An Empirical Study of Algorithms for the Subset Sum Problem

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

Abstract

The Subset Sum problem is a well-known NP-complete problem. It can be 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? This problem is particularly interesting because, depending on what parameter is used to characterize the size of the problem instance, it can be shown to have polynomial, sub-exponential, or strongly exponential worst-case time complexity. Let n represent the size of the input set S, and let m be the maximum value in the set. A standard dynamic programming algorithm for the problem can be shown to have polynomial time complexity $O(n^2m)$. On the other hand, there is an algorithmic model that includes both backtracking and dynamic programming in the research literature that is shown to have a strongly exponential lower bound of $2^{\Omega(n)}$ on the closely related Knapsack problem when n alone is used as the complexity parameter. And finally, a variant of dynamic programming called dynamic dynamic programming has been shown to have a worst-case sub-exponential time complexity of $2^{O(\sqrt{x})}$ when the total bit length x of the input set is used as the complexity parameter. In addition to the contrast in complexities of different algorithms, it is also known that each algorithm is very sensitive to the density (quantified as n/m) of the problem instance. Dynamic programming is expected to be most efficient for very dense instances, while backtracking is expected to be most efficient for sparse instances.

This paper presents the results of an empirical study that illustrates the comparative performance and characteristics of the three algorithms mentioned above. Various experiments are designed to test the analytical complexity results, to find what density defines the critical region of the problem space, to determine what target sums are most difficult to find, and to compare and contrast the performance of the three algorithms in varied regions of the problem space.

1 Introduction

Subset Sum is a well-known hard problem in computing that can be informally defined as follows: Given a set of positive integers and a target value, determine whether some subset has a sum equal to the target. In different versions of the problem, the input set may or may not contain duplicate values, and the problem can also be expressed as an optimization problem. All such versions are NP-complete. We will use the formal definition below, which coincides with the definition in [4]. As a notational convenience, let $\Sigma(S)$ represent the sum of all elements of set S.

Definition 1.1. The *Subset Sum* problem is defined as follows: given a set of positive integers S and an integer t, determine whether there is a set S' such that $S' \subseteq S$ and $\Sigma(S') = t$.

The Partition problem – determining whether a set of numbers can be partitioned into two subsets that have the same sum, can be defined as a special case of Subset Sum.

Definition 1.2. The *Partition* problem is defined as follows: given a set of positive integers S, determine whether S has a subset with sum $\lfloor \Sigma(S)/2 \rfloor$.

The Partition problem is also NP-complete. In discussing Subset Sum or Partition, we will denote the size of the input set S as n and the maximum value in S as m, and given the assumption that S does not contain duplicates, we have $n \leq m$.

The Subset Sum problem is particularly interesting because, depending on what parameter is used to characterize the size of the problem instance, it can be shown to have polynomial, sub-exponential, or strongly exponential worst-case time complexity. A standard dynamic programming algorithm for the problem can be shown to have polynomial time complexity $O(n^2m)$. On the other hand, there is an algorithmic model sufficiently general to include both backtracking and dynamic programming in the research literature [1] that is shown to have a strongly exponential lower bound of $\Omega(2^{n/2}/\sqrt{n})$ on the closely related Knapsack problem when n alone is used as the complexity parameter. And finally, a variant of dynamic programming called dynamic dynamic programming has been shown to have a worst-case sub-exponential time complexity of $O(x^2 2^{\sqrt{x}})$ when the total bit length x of the input set is used as the complexity parameter [8]. In addition to the contrast in complexities of different algorithms, it is also known that Subset Sum algorithms are very sensitive to the density (quantified as n/m) of the problem instance. Dynamic programming is expected to be most efficient for very dense instances, while backtracking is expected to be most efficient for sparse instances.

This paper presents the results of an empirical study that illustrates the comparative performance and characteristics of the three algorithms mentioned above. The *AlgoLab* software application [7] is used to generate batches of randomly generated problem instances and automatically chart the results. Section 2 below provides a description of the algorithms and sketches of their worst-case time analysis. Sections 3 describes various experiments that test the analytical complexity results, find what density defines the critical region of the problem space, determine what target sums are most difficult to find, and compare the performance of the three algorithms.

2 Algorithms for Subset Sum

We will examine three algorithms for Subset Sum: backtracking (BT), dynamic programming (DP), and dynamic dynamic programming (DDP). All three algorithms can take advantage of the fact that the problem is symmetric around $\Sigma(S)/2$. Specifically, S has a subset S' with sum t if and only if S has a subset $S - S'$ with sum $\Sigma(S) - t$. For dynamic programming, for example, a search for a smaller target value generally requires fewer steps than a search for a larger one.

As with any combinatorial problem, backtracking is a standard approach to solving Subset Sum. It has a simple recursive formulation, and with the proper bounding conditions, it is competitive with any other exact algorithm. The logic is simply to branch on the numbers in the set S. For any element y of S, if there is a subset S' with sum t, it either contains y or it doesn't. If S' contains y, we can put y in the subset we get by recursive call on $S - \{y\}$ and $t - y$. Otherwise, we skip y and find S' by recursive call on $S - \{y\}$ and t. The BT algorithm, a simple version of backtracking, is shown in Figure 1. The input set S is represented as an array (indexed 0 to $n - 1$), and an index parameter identifies which element of the array is the current branching value. Backtracking does not require and ordered input set, but we find that it is significantly more efficient if the array is ordered and the values are processed from high to low. This maximizes the benefit of the bounding condition that terminates a subtree search when the sum of the rest of the elements is insufficient to achieve the target (line 1). Lines 5 and 6 implement a symmetry-motivated heuristic to look for the smaller of targets t and $\Sigma(S) - t$. The rationale is that a smaller target should have a shallower depth of recursion, but choosing a smaller target may also minimize the benefit bounding condition in line 1. Experiments have shown that there is a small improvement in step counts when the smaller target is chosen if the set S is a sparse set, and there is a small benefit in choosing the larger target for dense sets.

A BT computation can be modeled as a binary tree where each node represents a single activation of the recursive code. Each activation processes one element of S, and it makes at most two recursive calls. So the total number of recursive calls cannot exceed the number of nodes in a full binary tree of depth n, and the worst-case time complexity is $O(2^n)$.

Subset Sum is a prototypical "pseudo-polynomial-time" NP-complete problem. It has an algorithm that operates in time $O(m \cdot n^2)$, which is considered to be an example of the dynamic programming (DP) method. Instead of enumerating all possible subsets of S, it enumerates all possible sums of subsets of S, up to and including the target sum. Since S is a subset of $\{1, 2, ..., m\}$, the sum of all its elements cannot exceed $n \cdot m$. Any target sum larger than this is trivially not achievable. In defining DP, an implementation of dynamic programming, we set up a bit array T of length t to represent the sums that can be produced by subsets of S, initially all zeros except $T[0]$, which is set to 1 (representing the sum of the empty subset of S). When it is determined that some subset has sum r, $T[r]$ is set to one. The sums of subsets are recorded while processing one element of S at a time. To process a number y, T is traversed from high index to low, and for each $T[i] = 1$, $T[i + y]$ is set to 1. After processing all n elements of S, the array T indicates all sums of subsets of S.

```
boolean function BT
        (set S,          // an array of positive integers
        int index,       // the index of the current number
        int target,      // the target sum
        int sumofrest)   // the sum of the numbers yet to be processed
1)  if (index < 0 ∨ target < 0 ∨ target > sumofrest)
2)      return false;
5)  else if (target > setsum/2)
6)      target ⇐ setsum − target;
3)  int curnum ⇐ S[index];
4)  if (sumofrest = target ∨ curnum = target ∨ target = 0)
5)      return true;
6)  else if (BT(S, index − 1, target − curnum, sumofrest − curnum))
7)      return true;
8)  else
9)      return BT(S, index − 1, target, sumofrest − curnum);
```

Figure 1: A backtracking algorithm (BT) for Subset Sum

Pseudocode for DP is shown in Figure 2. The implementation employs bitwise *shift* and *or* operations to implement the processing of each element of S. We note that the algorithm does not require that the set S is ordered, and that the symmetry heuristic (lines 5 and 6) can result in significant savings of time and space, since the size of the sum map depends directly on the target t.

The step count for a DP computation is $n \cdot t$, and since t is $O(n \cdot m)$, we achieve the complexity of $O(m \cdot n^2)$, which is polynomial in n and m. We call the complexity pseudo-polynomial, however, since it is possible that m is much larger than n, even exponential in n. If $m = 2^n$, $O(m \cdot n^2)$ is really $O(n^2 \cdot 2^n)$, which is actually worse than $O(2^n)$ – worse than backtracking. Another limitation of DP is its space complexity, which is $O(n \cdot m)$, also potentially exponential in n. Backtracking, by contrast, has $O(n)$ space complexity.

In addition to backtracking and dynamic programming, the research literature contains algorithms that split the input array into two subarrays, perform dynamic programming on the lower portion to produce a sum map, and perform backtracking to enumerate the sums of subsets of the upper portion. The search terminates successfully when an upper subset S' is discovered such that $t - \Sigma(S')$ is in the sum map. The hybrid approach has some of the advantages of each previous method, while quadratically reducing the space complexity. Woeginger [10] provides a description of this method and its complexity. If we divide the input array into two equal-size pieces, the complexity is $2^{O(n/2)}$. Note that the input array does not need to be ordered for the algorithm to be successful.

If we use the total bit length x of the input set as the complexity parameter instead of the number of values n in the set, the hybrid approach can be exploited to achieve sub-exponential time complexity. The trick is to use an ordered array for the input and separate

```
boolean function DP(set S, int target)   // searches for a subset of S
                                         // with specified target sum
1)   if (target = 0 or target = Σ(S))
2)       return true;
3)   else if (target > Σ(S))
4)       return false;
5)   else if (target > Σ(S)/2)
6)       target ⇐ Σ(S) − target;
7)   if (target ∈ S)
8)       return true;
9)   BitMap summap: summap[0] ⇐ 1;
10)  for each num in S from high to low
11)      BitMap newmap ⇐ summap >> num;
12)      summap ⇐ summap or newmap;
13)      if (summap[target] = 1)
14)          return true;
15)  return false;
```

Figure 2: A dynamic programming (DP) algorithm for Subset Sum

the values with less than $\sqrt{x}$ bits from those with $\sqrt{x}$ or more bits. This analysis was apparently first published in [9]. It has recently been shown that dynamic programming with a dynamically allocated list of sums (instead of a fixed-size array) can achieve the same time complexity as the hybrid algorithm [8]. Pseudocode for this algorithm, called DDP, is shown in Figure 3. The algorithm requires that the numbers in the input set are ordered, and the numbers are processed from high to low. A list of sums, initially containing only the target sum (line 9), is maintained as the input numbers are processed. A number is processed by subtracting it from each sum on the current sum list to get a new sum list (lines 14-18). The current and new sum lists are then merged cosequentially, removing duplicates and keeping the sum list in order for the next iteration (line 20). The list is also pruned to remove the unreachable sums, which are greater than the sum of the remaining unprocessed numbers (line 19).

The DDP algorithm achieves sub-exponential time without explicitly partitioning the input set. The step count is approximated within a constant factor as sum of the lengths of the sum list over all iterations of the outer loop. The sum list initially has length 1, and it can at most double in length with each iteration. At the same time, its length can never exceed the sum of the remaining unprocessed numbers. So the length of the sum list in the i^{th} iteration of the outer loop (lines 11-20) is bounded by the smaller of 2^i and the sum of the smallest $n - i$ values in the set.

The sub-exponential time analysis relies on the property of an ordered number list that value of the number at position $n - \lceil \sqrt{x} \rceil$ in an ordered list of numbers with total bit length x is less than $2^{\sqrt{x}}$. Bounding the k^{th} value allows us to bound the sum of the first k values as well. This, in turn, leads to a sub-exponential bound on the length of the sum list, and the

```
boolean function DDP(set S, int target)   // searches for a subset of S
                                          // with specified target sum
1)   if (target = 0 or target = Σ(S))
2)       return true;
3)   else if (target > Σ(S))
4)       return false;
5)   else if (target > Σ(S)/2)
6)       target ⇐ Σ(S) − target;
7)   if (target ∈ S)
8)       return true;
9)   List sumlist ⇐ {target}
10)  sumofrest ⇐ Σ(S)
11)  for each num in S from high to low
12)      sumofrest ⇐ sumofrest − num;
13)      List newlist ⇐ {};
14)      for each sum in sumlist
15)          if (sum − num = 0 or sum − num = sumofrest)
16)              return true;
17)          else if (sum − num > 0 and sum − num < sumofrest)
18)              newlist.append(sum − num);
19)      sumlist.truncate(sumofrest);
20)      sumlist.mergewith(newlist);
21)  return false;
```

Figure 3: The DDP algorithm for Subset Sum

total step count is determined by the cumulative lengths of the sum list over all iterations of the outer loop. Details of the analysis can be found in [8].

3 An Empirical Comparison

The algorithms of the previous section were tested to compare the worst-case analytical complexity results with empirically derived average step counts for randomly generated problem instances. When conducting experiments with algorithms on randomly generated problem instances, it is important to know where the easier and harder instances are found in the problem space. For the Subset Sum problem, the relevant parameters are the size n of the input set and the maximum value m in the set. The ratio of these numbers n/m defines the density of the set, and this property can be used to define regions of the problem space. Combinatorial decision problems typically have critical regions where the hardest instances reside. The critical region is normally defined to be the region just below and above a crossover point. At the crossover point, random instances have a 50% chance of having a solution. And in the critical region, the probability of finding a solution changes rapidly from close to 0 (the over-constrained side of the crossover point) to almost 1 (the under-constrained side). The metaphor of phase transition (from physics) is frequently used

in the research literature to describe the critical region. We expect that a backtracking algorithm with appropriate bounding conditions will show the highest step counts for instances in the critical region. Studies of the critical region for the Partition problem began to appear in about 1996 (e.g. [5]). Crossover was predicted to occur at $m = 2^n$, or $n = \lg m$. Stephan Mertens [6] published a more detailed model a few years later. His analysis parameterized the average bit length b of the numbers in the set and predicted crossover when $b = \kappa n$ where κ is a value that approaches 1 as n grows. In order to use the Mertens model, we would have to make some simple some simple assumptions about b and calculate κ for given values n. For the purposes of our experiments, however, the simpler model where $b = \lg m$ and κ is constant at $\kappa = 1$ provides sufficient guidance for finding the critical region.

A critcal region experiment was conducted to validate the three algorithms of the previous section. To illustrate the critical region, we can either fix n and vary m, or fix m and vary n. Figure 4 shows the results when n varies from 10 to 30, and m is fixed at 100,000. The target sum for the experiment (by default) is set to half the sum of the input set, so the algorithms are tested on instances of the Partition problem. Each data point for step counts or space untis on the charts represents the average result from 1000 instances. The upper left plot in Figure 4 shows the decision results as *%Yes* and *%No*. The critical region and crossover point are clearly illustrated, with crossover very close to $n = 18$. The experiment confirms the correctness of the three algorithms, with 100% consistency in their results for the 1000 instances for each value of n. The upper right plot in Figure 4 illustrates (on a logarithmic scale for the y-axis) the space required for each of the three algorithms. The lowest line is BT, requiring $O(n)$ space, and the highest is DP, requiring $O(t)$ space, where t for each trial is half the sum of the input set. DDP uses less space than DP, but the growth rate of the space metric approaches the growth rate for DP. This illustrates the known time-space trade-off of variants of dynamic programming. A savings in time is achieved by using a lot more space, super-polynomial in n. For sparse problem instances with large values of m, the space requirement can be a problem. The two charts in the lower half of Figure 4 show the average step counts for the three algorithms. For BT, the step count is the number of recursive calls. For DP and DDP, the step count is the cumulative lengths of the sum lists over all iterations of the outer loop. The two measures may not be directly comparable, but they both capture the super-polynomial factor in the respective algorithms' time functions. The top line in the lower left chart is DP with DDP and BT very close to each other at the bottom of the chart. The lower right presents the same data on a logarithmic scale showing that DDP out-performs BT until the crossover point is passed, and then the BT step counts level off while DDP step counts continue to grow.

The empirical results in Figure 4 do not necessarily reflect worst-case analytical results. Backtracking is better than DP for all problem instances, even as the density of the problem instance is increasing. The worst-case analyses lead to the expectation that DP will be better than BT for high-density problem instances. Apparently, the density never gets high enough to give DP the advantage. Of course, we need to keep in mind that the empirical data from randomly generated instances is an illustration of average-case complexity, not worst-case complexity. While it is not known that average time complexity differs from

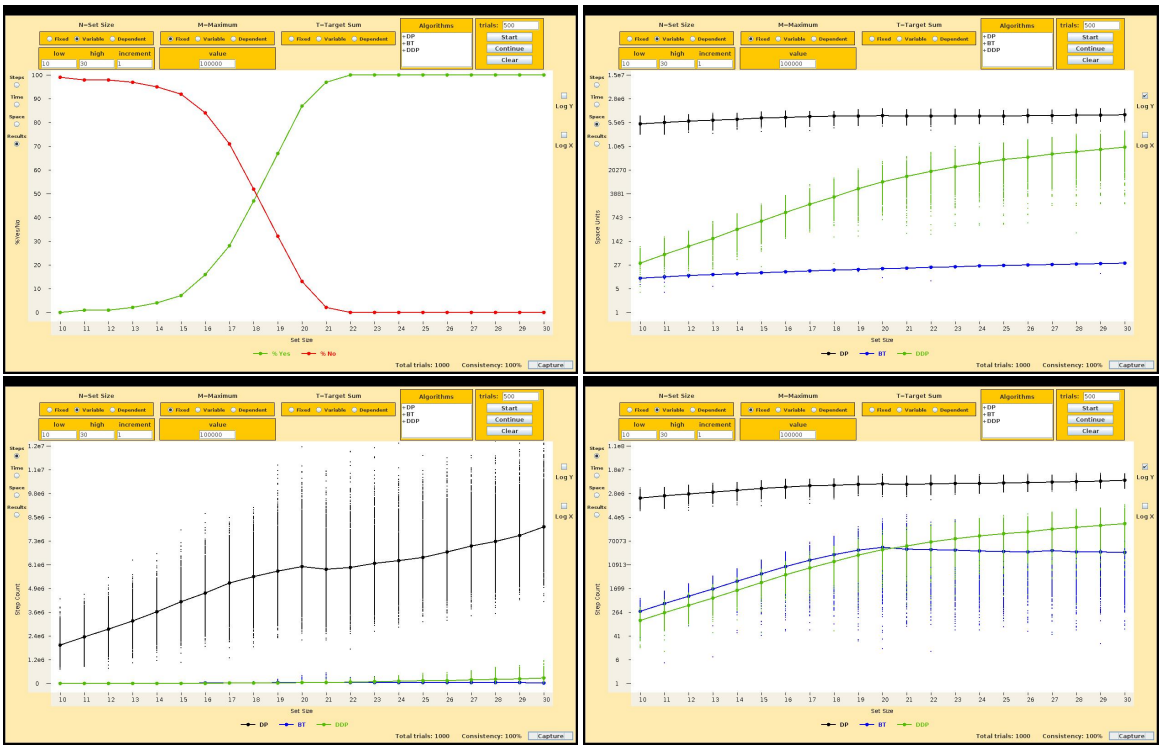

Figure 4: Crossover with fixed m using BT, DP, and DDP

worst-case time complexity for either backtracking or dynamic programming, it is known that many algorithms perform far better than worst-case for instances outside the critical region. This is particularly true for BT, which, unlike dynamic programming variants, is strictly a depth-first search with no additional overhead for storage of partial solutions. While we might expect dynamic programming's $O(n^2m)$ complexity to outperform backtracking's $O(2^n)$ for high-density instances, the greedy heuristic employed by BT is quite effective when there is a high probability that a solution can be found.

For the sake of comparison, we design another crossover experiment with n fixed at 20 and m varying from 100,000 to 1,000,000. As before, the target sum is half the sum of the input set by default. The DP algorithm was left out of the experiment, since its step count is significantly higher than both BT and DDP. The results of the experiment are shown in Figure 5. The chart on the left shows that we have again captured the crossover point, but from the opposite direction. The density and the probability of finding a solution are both decreasing as m grows larger. It also appears that the density changes more slowly than in the previous experiment. The region where backtracking out-performs DDP is not entered (on the left), and the problem instances remain in the critical region (on the right) as m grows. We find the crossover point at about $m = 334,000$.

The Partition problem has been labeled in the research literature as the easiest hard problem [6], apparently based on the differences between Partition and the strong NP-complete problems (those that do not have psuedo-polynomial-time algorithms). The literature does

not actually discuss how Partition is easier than the more closely related problems that also have psuedo-polynomial-time algorithms, such as Subset Sum, Knapsack, and Bin Packing. Partition is the special case of Subset Sum where the target value is $\Sigma(S)/2$. If Partition is really the easiest hard problem, it should be the easiest instance of Subset Sum. This expectation is reinforced by the fact that there are more subsets whose sum is $\Sigma(S)/2$ than any other target sum. We would expect backtracking to succeed sooner with more solutions in the solution space. We also have results from [2] and [3] saying that for dense enough sets, a symmetric cluster of subset sums can be found around $\Sigma(S)/2$, and the cluster grows wider as the density grows. So subsets with sums near $\Sigma(S)/2$ have a higher probability of occurrence than those with sums far from the center.

To test the hypothesis that Partition is the easiest special case of Subset Sum, we choose fixed values for n and m, and we vary the target sum from just above m to about $\Sigma(S)$. If Partition is easier, the step count curve will reach a low point in the middle. Figure 6 shows the results of two such experiments, one for sparse sets where $n = 20$ and $m = 500,000$ (upper charts), and the other for dense sets where $n = 100$ and $m = 200$ (lower charts). In both cases, the target sum varies from m to $n \cdot m/2$, and both BT and DDP are used. The rightmost charts show the step counts for both experiments. Each chart has a curve for BT and for DDP. We see a hump, not a trough, in the middle target range. The DDP algorithm shows a hump for both the sparse and dense experiments, and BT shows a hump for the sparse instances. A closer look at the step counts for BT on the dense sets reveals that the curve is very flat – no evidence of either a hump or a trough in the middle. The generally higher step counts for sums near $\Sigma(S)/2$ indicate that these sums are harder, not easier, to find. So we should probably call Subset Sum the easiest hard problem, with Partition as its hardest special case.

Another unexpected result from the experiments of Figure 6 is the algorithms' performance on sparse vs. dense problem instances. The published worst-case complexity analyses lead us to expect backtracking to be better for sparse instances and variants of dynamic programming to be better for dense instances. The step counts show the opposite results. For the sparse experiment, the higher step counts for backtracking are consistent with the results from Figures 4 and 5, which also have DDP better than BT near the crossover point.

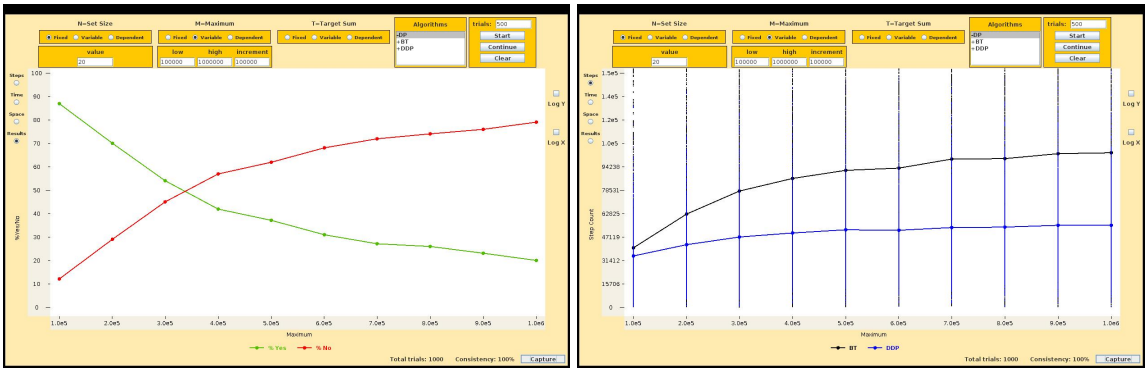

Figure 5: Crossover with fixed n using BT and DDP.

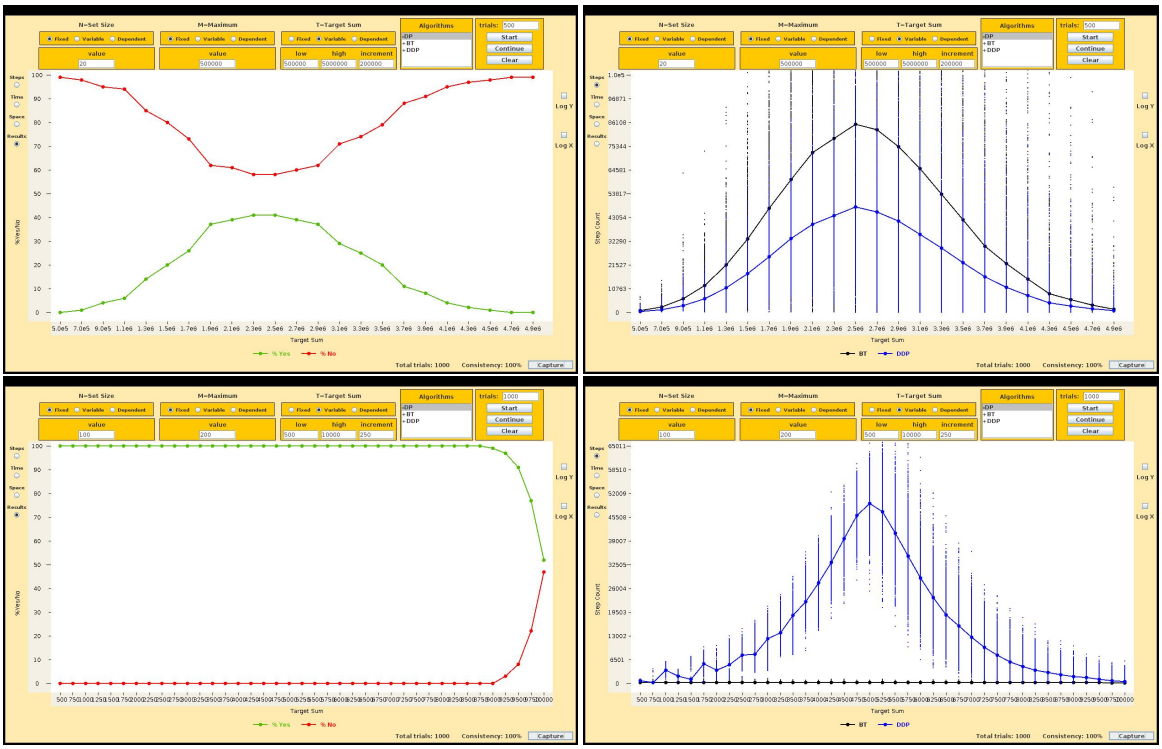

Figure 6: Exploring the target range using BT and DDP.

The upper left chart in Figure 6 shows that the fixed density of $20/500,000$ is close to the crossover point. The *%Yes* curve approaches 50% in the middle of the target range and then recedes as the targets become less central. We assume that our density is not yet sparse enough to show lower step counts for backtracking. For the dense experiment, we might speculate that backtracking with the hueristic of branching on the higher numbers first actually has lower than exponential average-case complexity. DDP has been shown to have sub-exponential complexity, but so far no proof has emerged that backtracking by itself can be sub-exponential in the worst case. While the experiments of Figure 6 raise questions about the true complexity of backtracking, they provide solid evidence that the middle sums are the hardest to find. This validates the approach of the previous two experiments, where target sums were set by default to be in the middle of their range as m or n varied. While the number of solutions is higher for middle sums, the number of subsets that remain feasible and resistant to the bounding heuristic is also higher, resulting in deeper, more costly failed sub-searches. As a final note regarding the experiment of Figure 6, the decision results (lower left) for the dense instances are 100% *Yes* until the very end of the target range. The instances for which no subset produced the target sum at the end of the range are the result of a target sum that exceeds the sum of the entire set. The calculation of the target sum was approximate, not exact, and some of the calculations exceeded the set sum.

The previous experiments have established where the critical region is, and which targets are the hardest. We design one more experiment to compare BT with DP and DDP on

Figure 7: DP, BT, and DDP comparison for growing n, $\%Yes \approx 50\%$

instances that grow in size but remain in the critcal region. To accomplish this, we make m and the target sum t dependent on n, hoping to keep t in the middle of the target range and to maintain a constant *%Yes* at about 50% for the decision results. Specifically, we vary n from 10 to 25, set m to $2^{0.95n}$, and let t default to half the sum of the input set. The results of this experiment are shown in Figure 7. The decision results are displayed on the left, and the step counts are on the right. The results were 100% consistent for all three algorithms. We see that the percentage of *Yes* decisions was below 35% for the smallest values of n, and it increased slowly to around 40% as n increased. This confirms that formula for the m parameter was sufficient to keep the problem instances in the critical region. The step counts for the three algorithms are shown in the chart on the right on a logarithmic scale. All the lines appear to be straight, indicating exponential growth in step counts. The top line is DP, the lowest line is DDP, and the second lowest is BT. The top line obviously has steeper slope than the bottow two, indicating that DDP significantly improves DP. It is interesting that the bottom two lines appear to be parallel, with step counts for BT less the twice the step counts for DDP, suggesting that the two algorithms have the same time complexity. While DDP [8] and a hybrid of dynamic programming and backtracking [9] have been shown to have sub-exponential worst-case complexity, this result has never been established for any variant of backtracking. DDP achieves sub-exponential complexity by suppressing duplication of sums on its growing sum list. BT, especially while conducting an unsuccessful search for a dense input set, might recompute the same sums multiple times for multiple subsets. Unsuccessful searches, however, are unlikely for dense sets, and for sparse sets, it is unlikely that two subsets will have the same sum. The step counts of Figure 7, where problem instances are sparse, show that the number of nodes in a backtracking tree corresponds directly to the number of sums on a DDP sum list.

4 Summary and Further Work

The experiments described here illustrate the problem space of the Subset Sum problem, corroborating published analytical results regarding the location of the critical region. They also provided evidence that Subset Sum is more deserving of the label ”easiest hard problem” than the Partition problem. A new variant of dynamic programming called DDP is

shown to be competitive with both standard dynamic programming and backtracking. DDP has lower step counts than both of the other algorithms for medium-density to low-density problem instances. The final experiment confirms that all three algorithms have strongly-exponential time complexity when the complexity parameter is the number of integers in the input set. At the same time, DDP is known to have sub-exponential worst-case time complexity when the complexity parameter is the total bit length of the input set. This suggests that an additional experiment, one in which step counts are plotted as a function of total bit length of the input, is needed to further corroborate published analytical results.

References

[1] M. Alekhnovich, A. Borodin, J. Buresh-Oppenheim, R. Impagliazzo, A. Magen, and T. Pitassi, Toward a Model for Backtracking and Dynamic Programming, *Proceedings of the 20th Annual IEEE Conference on Computational Complexity* (2005), pp. 308-322.

[2] N. Alon and G. Freiman, On Sums of Subsets of a Set of Integers, *Combinatorica* **8:4** (1988), pp. 297-306.

[3] 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.

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

[5] I. Gent and T. Walsh, Phase Transitioning and Annealed Theories: Number Partitioning as a Case Study, Instituto per la Ricerca Scientifica e Tecnologica (IRST), Technical Report #9601-06 (1996).

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

[7] T. E. O'Neil, A Virtual Laboratory for Study of Algorithms, *Proceedings of the 42nd Midwest Instruction and Computing Symposium* (Eau Claire, WI, 2009).

[8] 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* (CSREA Press, 2010), pp. 55–58.

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

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