0/1-Knapsack vs. Subset Sum: A Comparison using *AlgoLab*

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

Abstract

0/1-Knapsack and Subset Sum are two closely related, well-known NP-complete problems. There is a direct reduction from Subset Sum to Knapsack, and the methods for solving Knapsack are generally the same as for Subset Sum. So Subset Sum is a special case of Knapsack. Does this suggest that Knapsack is easier or harder than Subset Sum? A review of the literature shows that both problems have the same complexity when the complexity parameter is n, but that Knapsack appears to be easier when the complexity parameter is x, the total bit length of the input. This leads to an empirical study of the two problems to determine whether the problem spaces have the same characteristics and whether various algorithmic strategies exhibit the same relative performance.

The empirical study is conducted using *AlgoLab*, a software package written in Java that allows rapid comparison of algorithms for batches of randomly generated problem instances. The need to implement the Knapsack problem in *AlgoLab* led to a significant generalization of its capability. Earlier versions of the software allowed at most two parameters to drive the random instance generator at at most one constraint on the problem solution. This is adequate for Subset Sum, since set size and maximum value are sufficient parameters to generate random instances, and the target sum is the only constraint on the solution. With Knapsack, however, we must specify the number of objects, the maximum weight of any object, and the maximum value of any object for the instance generator. Also, the solution has two constraints, a knapsack capacity and a minimum value goal. The new version of *AlgoLab* allows any number of parameters to define a problem instance and any number of constraints to characterize the desired solution. This enables experimentation to show that when the value of an object is independent of its size, Knapsack instances are much easier to solve, and the problem space does not show any of Subset Sum's sensitivity to the density of the input set.

1 Introduction

0/1-Knapsack and Subset Sum are two closely related, well-known NP-complete problems. Subset Sum problem can be 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 the sum of integers in S' is t. The 0/1-Knapsack problem is more general: given a set S of n objects with weights $w[1..n]$ and values $v[1..n]$, a knapsack capacity C, and minimum value goal V, find a subset of objects whose weight is at most C and whose value is at least V. There is a direct reduction from Subset Sum to Knapsack, and the methods for solving Knapsack are generally the same as for Subset Sum. To reduce Subset Sum to Knapsack, we set the values of all objects to be equal to their weights and set the knapsack capacity to be the same as the minimum value goal. So Subset Sum is a special case of Knapsack. Does this suggest that Knapsack is easier or harder than Subset Sum?

A review of the literature shows that both problems have the same complexity when the complexity parameter is n, but that Knapsack appears to be easier when the complexity parameter is x, the total bit length of the input. The analysis using n dates back to the 1970s, when Horowitz and Sahni [2] defined a splitting algorithm for both problems with time complexity $O(\lg m \cdot 2^{n/2})$, where m is the largest number in the input set. This complexity has apparently not improved over the decades [11]. The analysis using x dates back to the 1990s, when Stearns and Hunt [10] defined a hybrid splitting algorithm for the Partition problem (a special case of Subset Sum) with complexity $2^{O(\sqrt{x})}$. This complexity is duplicated by the DDP algorithm for Subset Sum [8] and Knapsack [6]. Most recently, it has been shown that when DDP is applied to the Knapsack problem, the operation count depends on the bit length of the set of weights or the set of values, whichever is shorter, and the resulting complexity is $2^{O(\sqrt{x/2})}$[9]. This provides analytical evidence that Knapsack is generally an easier problem than Subset Sum, and we would expect empirical studies to show similar results. Martello, Pisinger, and Toth describe such results [3, page 414]: "Instances where a loose correlation, or no correlation at all, exists among the profit and weight of each item ... can be easily solved to optimality even for large values of n, while strongly correlated instances, as well as instances involving very large profit and weight values, may be very difficult." This provides motivation for the current study, which uses the *AlgoLab* software to illustrate the empirical differences between the Subset Sum and 0/1-Knapsack problems. A study of Subset Sum using *AlgoLab* was published a few years ago [7]. Here we provide an extension of that study to 0/1-Knapsack.

AlgoLab is a software package written in Java that allows rapid comparison of algorithms for batches of randomly generated problem instances. Earlier versions of the software allowed at most two parameters to drive the random instance generator and at most one constraint on the problem solution. This was adequate for Subset Sum, since set size n and maximum value m are sufficient parameters for the random instance generator, and the target sum t is the only constraint on the solution. With Knapsack, however, the problem instance contains two arrays of numbers, each with a distinct maximum, and two constraints on the solution: a knapsack capacity and a minimum value goal. *AlgoLab* was extended to

allow any number of parameters for the instance generator and any number of constraints on the solution. The changes to *AlgoLab* are discussed in more detail in Section 2 below, and the results of the experiments on the Knapsack problem are described in Section 3.

2 Modifications to *AlgoLab*

The *AlgoLab* software package automatically runs and charts experiments on batches of randomly generated problem instances as defined by user-specified parameters and constraints. The software was intended to be usable for any decision or optimization problem. For each problem to be studied, the user supplies the code for a random instance generator and some number of algorithms to be compared. The original release ([5] allows the user to specify two parameters to control the instance generator and one constraint on the problem solution. This is adequate for many problems including k-Sat, graph problems such as k-Clique, and Subset Sum. For k-Sat, the instance generator produces a Boolean expression with m clauses over n variables and there is no constraint on the solution. For k-Clique, a random graph is specified using a vertex count n and an edge count e, and the constraint on the solution is the clique size k. For Subset Sum, the input is an array of n integers with maximum value m, and the constraint on the solution is the target sum t. For problems like Bin Packing and Knapsack, however, more parameters and/or constraints are required. A Bin Packing instance requires two constraints on the solution: a bin capacity and the number of bins. For a Knapsack instance, three parameters are needed to specify the input set: a number of objects n, a maximum weight object *WMAX*, and a maximum object value *VMAX*; and there are two constraints on the solution: a knapsack capacity C and a minimum value goal V.

To extend the usability of *AlgoLab*, the *MakeInstance* method of the *InstanceGenerator* interface was redefined to take a seed and an arbitrarily long array of parameters. Also, the *DecisionAlgorithm* and *OptimizationAlgorithm* classes were redefined to have a *setConstraint* method that accepts both a constraint index and a constraint value. The index is used to indicate which constraint to set, thus allowing the algorithm to maintain an arbitrarily long array of constraints on the problem solution. The user specifies the parameters and constraints by adding lines to the *AlgoLab.prj* configuration file.

The modified *AlgoLab* user interface is illustrated in Figure 2. The upper left input box labeled "X-Axis Value" allows the user to choose one of the parameters or constraints to be variable, defining the data points along the x-axis of the chart panel. The other parameters and constraints are displayed in two tabbed panes. Each parameter or constraint can be defined to be a fixed constant or a value computed as a function of the x-axis parameter/constraint. The contents of the *AlgoLab.prj* file for the Knapsack problem are shown in Figure 1.

```
project "The Knapsack Problem"
generator SVListGenerator
parameter N "Number of Objects"
parameter WMAX "Maximum Weight"
parameter VMAX "Maximum Value"
constraint C "Knapsack Capacity"
constraint V "Value Goal"
algorithm KBT
algorithm HSK
algorithm KDDP
```

Figure 1: An *AlgoLab* configuration file for the 0/1-Knapsack Problem

3 Knapsack Experiments

Many combinatorial problems have critical regions in the problem space where the probability of finding a solution changes rapidly from 0 to 1, with a crossover point at 0.5. A previously published AlgoLab study of the Subset Sum problem illustrates critical regions related to both the density of the input set [7, Figures 4 and 5] and the value of the constraint [7, Figure 6]. The density-related crossover point matches the mathematical models in the research literature [4, 1], which predict crossover at $n \approx \lg m$. The constraint-related crossover point, where the target sum is half the sum of the input set, provides evidence that the Partition problem is the most difficult special case of Subset Sum. Since Subset Sum is a special case of 0/1-Knapsack, The experiments described below were designed to determine whether Knapsack shows the same critical regions as Subset Sum. All the experiments use three algorithms: a bounded backtracking algorithm called KBT, an implementation of the splitting algorithm from [2] called HSK, and an adaptation of the DDP algorithm from [8] called KDDP.

Experiment 1 illustrates the Subset Sum problem as a special case of Knapsack. The set size N ranges from 1 to 30, the maximum object weight is fixed at 100000, and both the capacity and the value goal are set to $N * (100000/4)$, which is half the expected sum of the set. The code for the instance generator assigns the value of each object to be equal to its weight, so that the experiment represents the reduction of Subset Sum instances to Knapsack instances. The step counts are shown on a log scale in Figure 2 and the decision results in Figure 3. The results match Figure 4 of [7], showing a density-induced critical region with a crossover point at about $n = 18$. The backtracking algorithm KBT shows a step-count peak at the crossover point, and as the density continues to increase, the KBT step counts continue to fall. The splitting algorithm HSK shows the best performance over the entire range of instances.

Experiment 2 is identical to Experiment 1, except that the instance generator is modified to make the random weight of each object completely independent of its random value. The step counts are shown on a log scale in Figure 4 and the decision results in Figure 5.

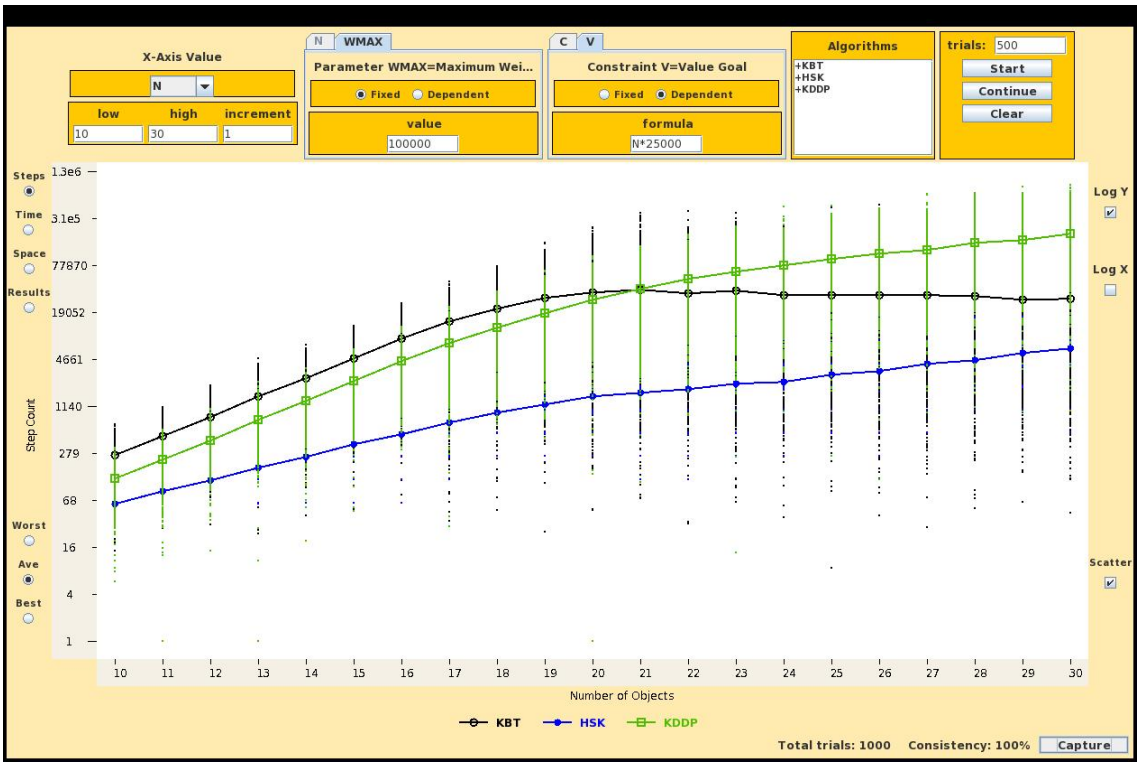

Figure 2: Step counts for equal weights and values with increasing density.

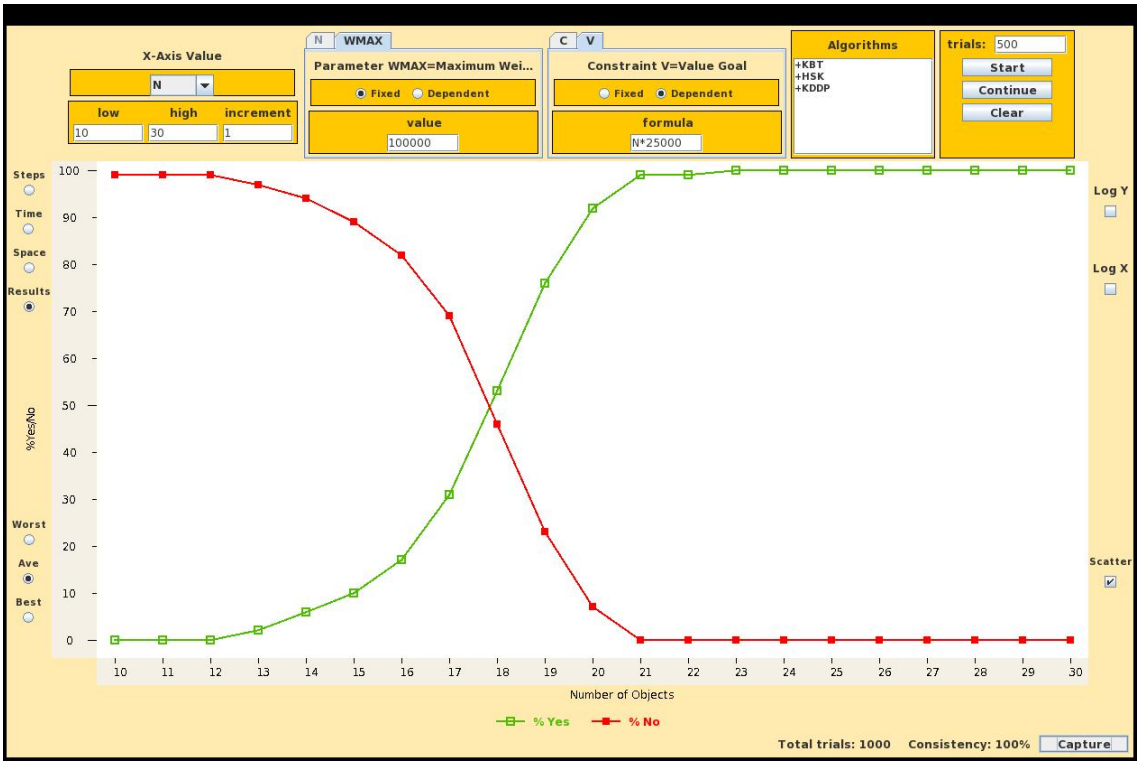

Figure 3: Decision results for equal weights and values with increasing density.

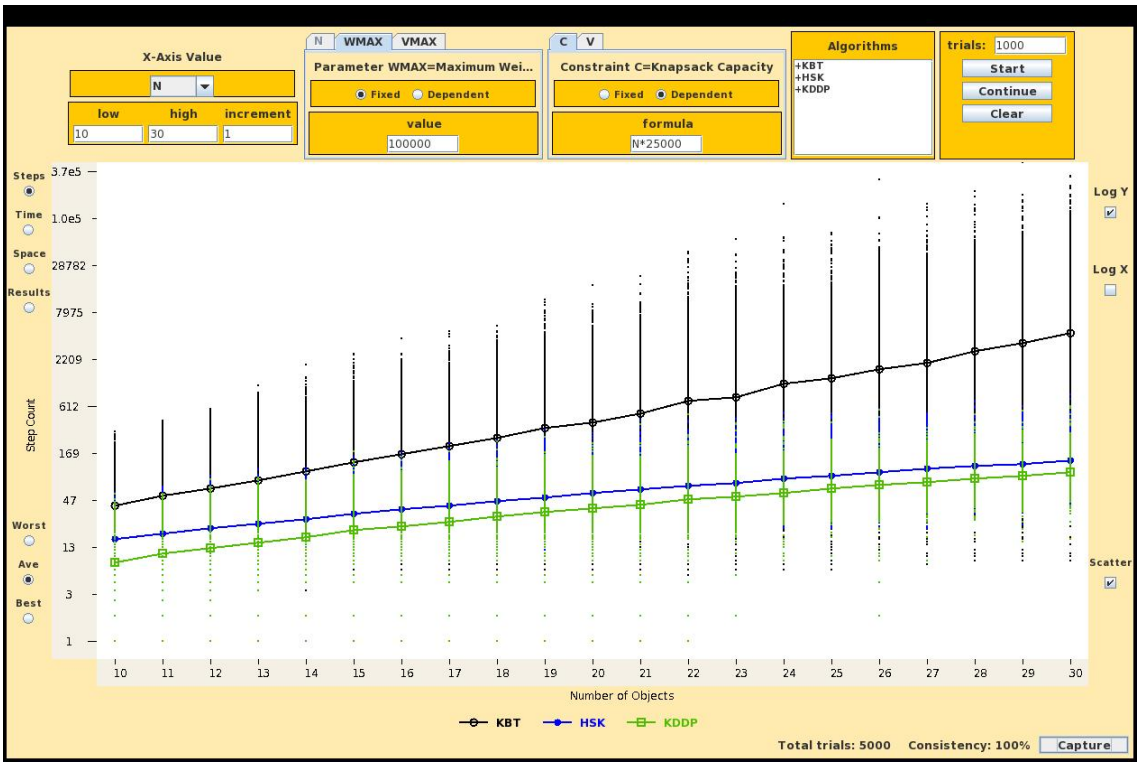

Figure 4: Step counts for independent weights and values with increasing density.

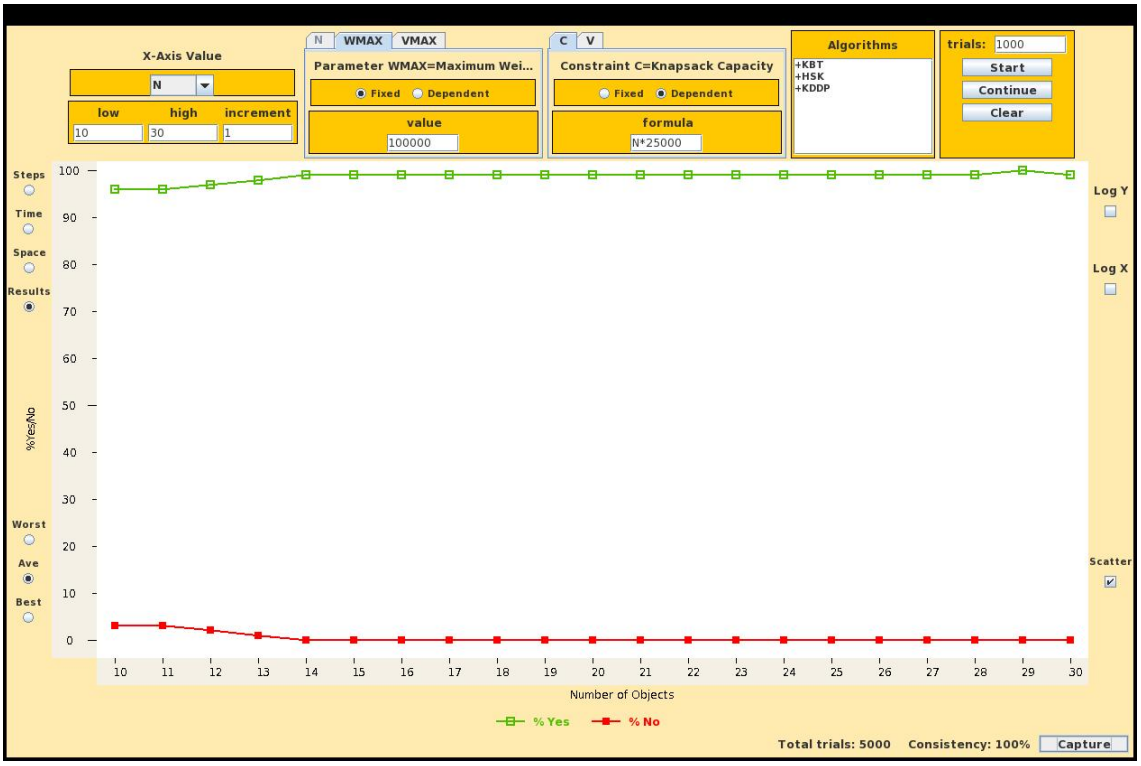

Figure 5: Decision results for independent weights and values with increasing density.

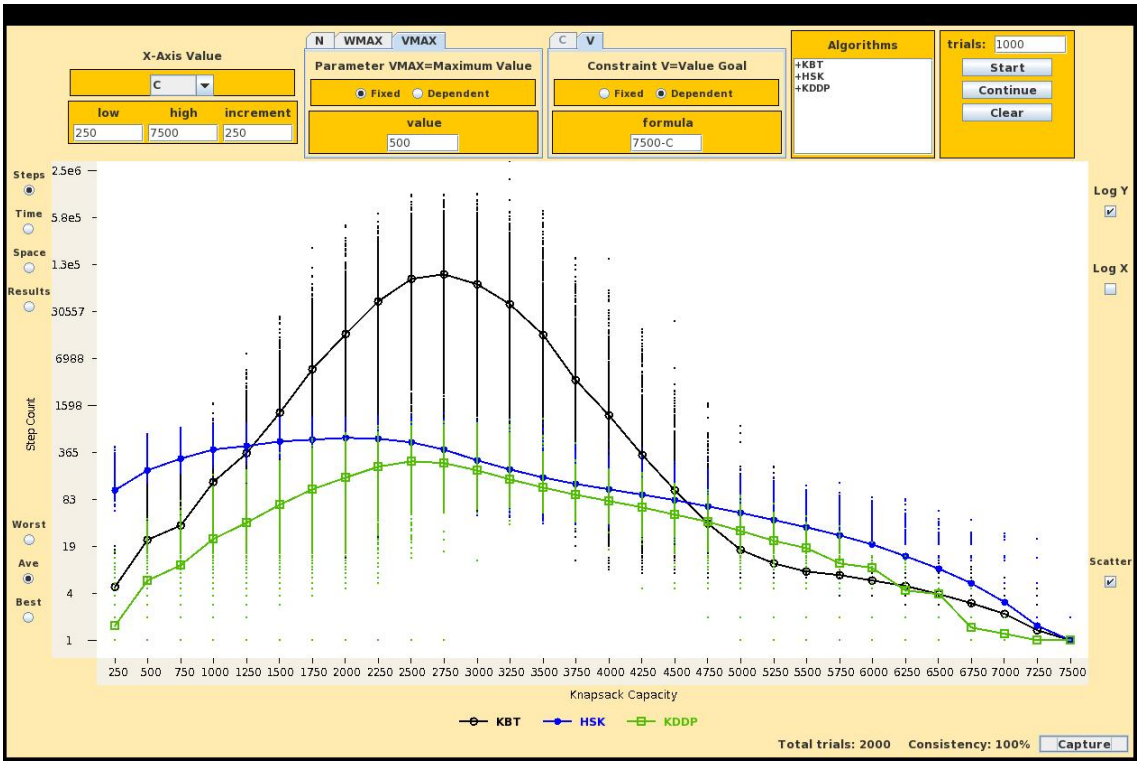

Figure 6: Step counts for a constraint-based crossover experiment.

The results show that the critical region is gone. The probability of finding a solution is near 1 over the entire range of instances. We also observe that the step counts are generally 10 times lower than in Experiment 1, and the ranking of the algorithms has changed. Backtracking shows the worst performance with no indication of lower step counts as the density grows, and KDDP outperforms both KBT and HSK over the entire range. When weights and values of objects are independent, it appears that density does not affect the probability of finding a solution.

Experiment 3 is designed to test for a constraint-based critical region. What constraint values will make the problem harder to solve? To answer this question, we fix N at 30 and set *WMAX* = *VMAX* = 500. This makes the expected sum of all weights to be $30{\cdot}500/2 = 7500$. We then vary the knapsack capacity C from 250 to 7500 while simultaneously varying the value goal V from 7250 to 0. The step counts are shown on a log scale in Figure 6 and the decision results in Figure 7. The decision results show a narrow critical region with crossover at about $C = 2500$, where the value of V would be 5000. Combining this information with data from another experiment that has C fixed at 2500 and V varying from 250 to 7500, we speculate that there is a 50% chance of finding a solution when have $V/vsum - C/wsum \approx .33$, where $vsum$ is the sum of all values and $wsum$ is the sum of all weights. This relation becomes the justification for setting the default constraints at $C = .33 \cdot wsum$ and $V = .66 \cdot vsum$ for the subsequent experiments. Regarding the step counts, we see that KBT and KDDP have peaks at the crossover point. KBT shows the worst performance, and KDDP shows the best.

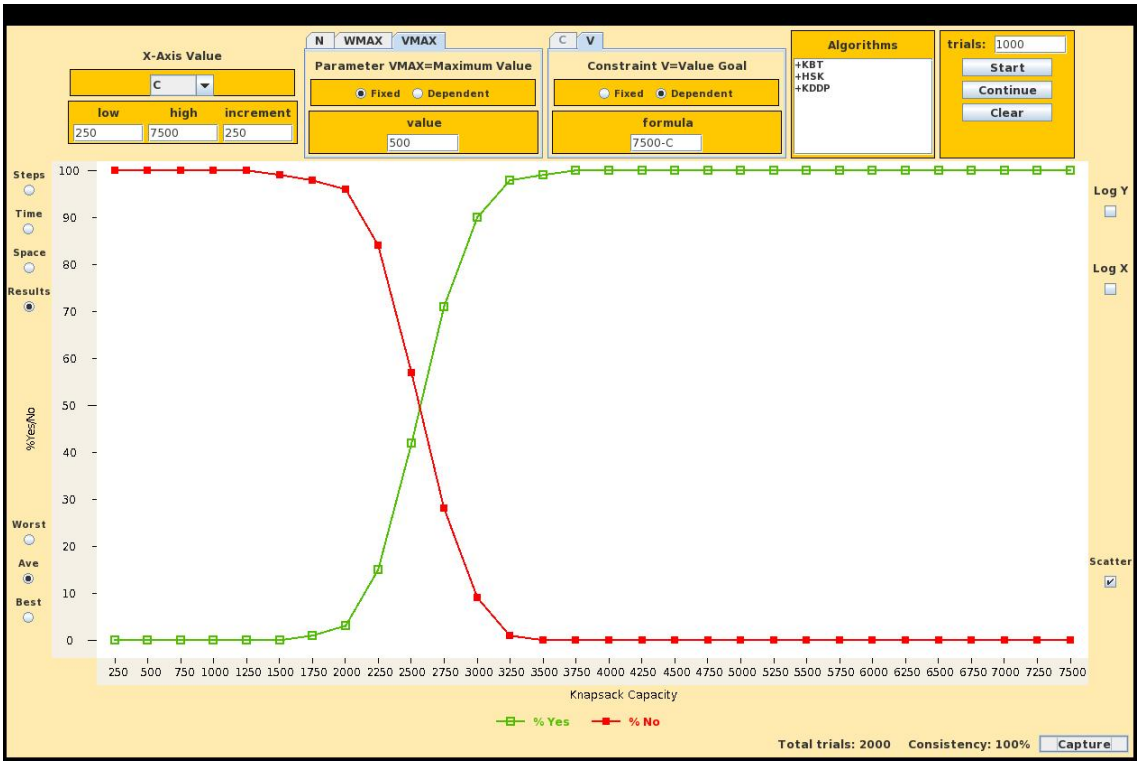

Figure 7: Decision results for a constraint-based crossover experiment.

The final experiment is intended to compare the performance of the three algorithms on problems that increase in size while remaining in the critical region. To achieve this requirement, we vary N from 10 to 40, set *WMAX* = *VMAX* = 100,000, and let the constraints assume their default values. The step counts are shown on a log scale in Figure 8 and the decision results in Figure 9. The results indicate that we are successful in keeping the instances near the middle of the critical region, at about 50% *yes*. The step count for KBT is clearly exponential, while the slope of the other two algorithms appears to decline on the log scale, indicating a sub-exponential growth rate. This experiment can be repeated for sets of size up to 1000 if KBT is excluded. KDDP shows consistently better performance than HSK, and the problems are much easier to solve than the corresponding Subset Sum instances. With Subset Sum, sets of size 50 trigger the Java "OutofMemoryError" exception.

4 Conclusion

The AlgoLab experiments confirm the observation of Martello, Pisinger, and Toth that Subset Sum, where object weights and values are equal, is much more difficult than the more general 0/1-Knapsack problem. The backtracking algorithm KBT, whose complexity is dictated by the set size n, shows step counts that grow exponentially for both problems.

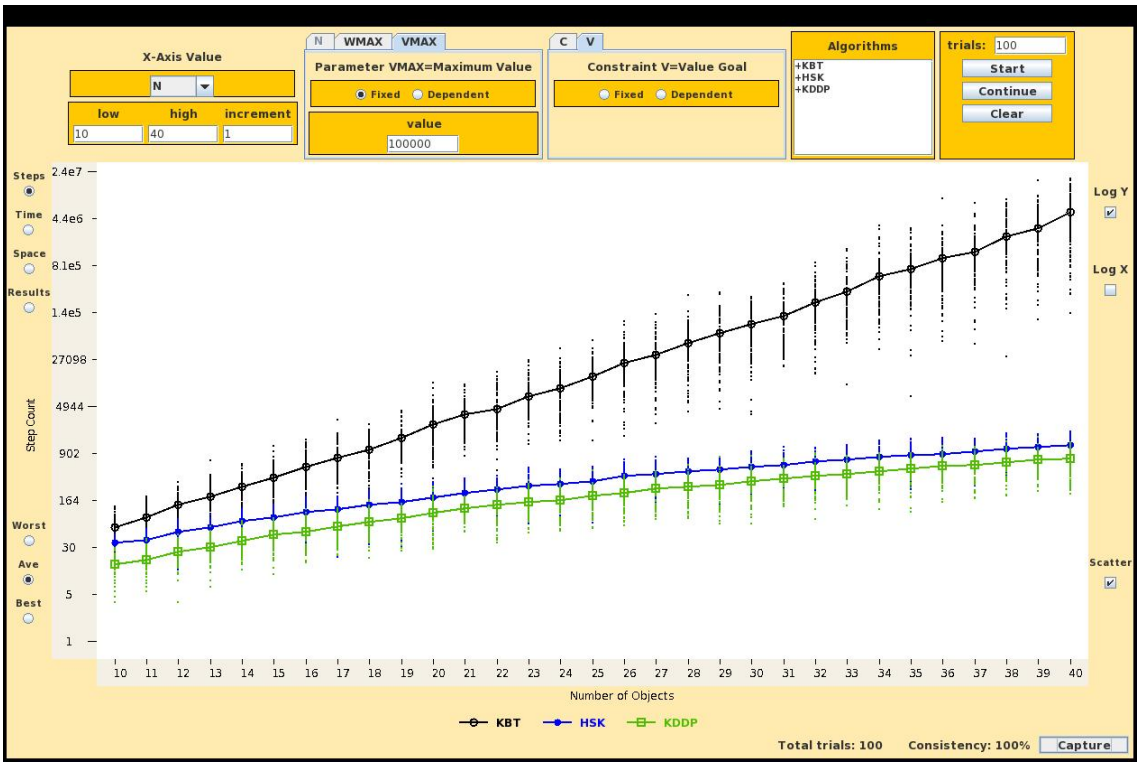

Figure 8: Step counts for growing set sizes in the critical region.

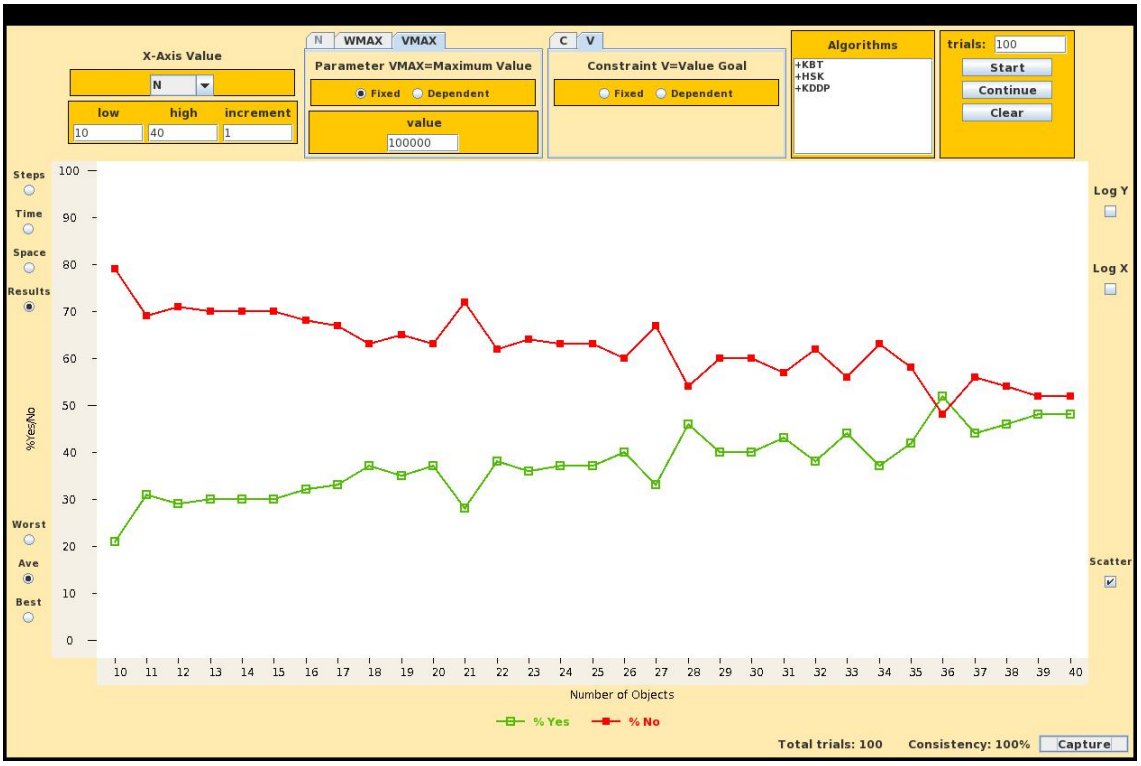

Figure 9: Decision results for growing set sizes in the critical region.

The step counts of the HSK and KDDP algorithms, however, which are variants of dynamic programming with dynamically allocated lists of partial solutions, exhibit more moderate growth rates for the Knapsack problem. It is interesting that the published complexities of the two problems are the same when the complexity parameter is n. A distinction is found, at least for the KDDP algorithm, only when the complexity parameter is x, the total bit length of the problem instance. The complexity classification using parameter x appears to be a better match for the empirical evidence.

References

[1] 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).

[2] E. Horowitz and S. Sahni, Computing Partitions with Applications to the Knapsack Problem, *Journal of the Association for Computing Machinery* **21:2**(1974), pp. 277-292.

[3] S. Martello, D. Pesinger, P. Toth, Dynamic Programming and Strong Bounds for the 0-1 Knapsack Problem, *Management Science* **45** (1999), pp. 414-424.

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

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

[6] T. E. O'Neil, Sub-Exponential Algorithms for 0/1 Knapsack and Bin Packing, *Proceedings of the 2011 International Conference on Foundations of Computer Science* (CSREA Press, 2011), pp. 209–214.

[7] T. E. O'Neil, An Empirical Study of Algorithms for the Subset Sum Problem, *Proceedings of the 46th Midwest Instruction and Computing Symposium* (LaCrosse, WI, 2013).

[8] T. E. O'Neil, Complement, Complexity, and Symmetric Representation, *International Journal of Foundations of Computer Science* **26:5** (World Scientific, August 2015), pp. 557-581.

[9] T. E. O'Neil, Improved Strongly Sub-Exponential Algorithms for Subset Sum and 0/1-Knapsack, submitted for publication, March 2016.

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

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