New Trials on Test Data Generation: Analysis of Test Data Space and Design of Improved Algorithm

So-Yeong Jeon[1] **and Yong-Hyuk Kim**[2,*]
[1]Department of Computer Science, Korea Advanced Institute of Science and Technology
Daejeon, Korea, E-mail: presentover@gmail.com
[2]Department of Computer Science and Engineering, Kwangwoon University
Seoul, Korea, E-mail: yhdfly@kw.ac.kr
* Corresponding author

Abstract – *Test (input) data generation is important for the algorithms/software/hardware testing. The previous researches on test data generation motivate us to find some meaningful information of generated test data. In this paper, we differentiate the test (input) data space (i.e., problem instance space) from the output data space (i.e., solution space), by examining the test data generated in terms of optimality (one of the performance measures). We investigate the problem instance space of the 0/1 knapsack problem, by calculating some kind of cost-distance correlation; the correlation was quite different from those in well-known combinatorial optimization problems. Also, we improved a greedy algorithm of the 0/1 knapsack problem by generated test data. The improved algorithm showed superiority to the original one under 10,000 random instances. This paper presents some promising values of the researches on the test data space and the test data generation for improving the tested module.*

Keywords: Problem instance space, 0/1 knapsack problem, greedy algorithm, algorithm performance analysis, cost-distance correlation.

1 Introduction

Test data generation or finding specific test data has been interest of many researchers in various fields. For the hardware testing, Saab *et al.* [1], Srinvas *et al.* [2], and Rudnick *et al.* [3] tried to generate test data finding some faults of sequential or combinatorial circuits. Corno *et al.* [4] tried to generate a test bench for a microprocessor design validation in terms of statement coverage. About software test data generation, McMinn surveyed those trials [5] for structural testing, non-functional testing and grey box testing. In fact, the test data generation is an undecidable problem in general. In the test data generation fore-mentioned, in fact, researchers use some ideas based on the meta-heuristic search algorithms. Then one of the important things is what objective function is used for the test data. But the objective value calculation in some case depends on the hardware environment. Execution time is the one of the examples (refer to [5] for the worst case execution time testing).

Therefore it would be general approach to define objective function in more abstract manner so that the calculation of objective value is independent from the hardware environment. *We regard a module of the software/hardware being tested as an algorithm.* The meaning of test data becomes the best/worst case defined for given testing objective function (e.g., the number of covered statements, the number of basic operations, or the degree to which the output is close to optimal one). The worst/best case is just some test data maximizing/minimizing the objective value. In the field of algorithm studies, we can see those researches. Johnson and Kosoresow [6] tried to find the worst case, in terms of optimality, of algorithms for online Taxicab problem. Cotta and Moscato [7] tried to find the worst case of the shell sort to approximate the worst-case time complexity. Hemert [8] tried to find the worst case of some algorithms for binary constraint satisfaction, Boolean satisfiability, and the travelling salesperson problem, in terms of time complexity. Jeon and Kim [9] tried to find the worst cases of well-known internal sorting algorithms as well as shell sort. They also tried to find the worst case, in terms of optimality, of algorithms for the 0/1 knapsack problem and the travelling salesperson problem.

Above researches give us some questions; this paper deals with two of those questions considering the 0/1 knapsack problem and the greedy algorithm as tested algorithm, as an extension of [9]. (From now on, the problem instances have the same meaning as the test data and we call problem instances just instances) The first question is, how different is generating test data of algorithms from finding solutions of problems? As the previous researches have focused on the cost-distance correlation (CDC) of solution space, we will calculate cost-distance correlation to investigate problem instance space, which is the space of the test data (Section 2). This will show how the test data space can be different. Note that, to the best of our knowledge, we are the first researchers using CDC for the investigation of *problem instance* space. The other question is, can we use the performance test (input) data for improving the tested algorithm? Previous researches did not focus on this question. Someone may underestimate the importance of the test data and want to see just the performance score. We will show the input data itself can be used for finding defects of the tested algorithm (Section 3). We observe the test data for which the tested greedy algorithm gives a far-from-optimal solution. Based on the observations, we devise an improved algorithm in terms of optimality. We make conclusions in Section 4.

2 Analysis of Test Data Space

```
U = list of all the given items;
W = given capacity of knapsack;
K = Ø; // items to be in the knapsack
Sort U as an array by profit density in non-increasing order†;
for each $item_i \in U$
{
  if ( weight sum of K + weight of $item_i \le W$ )
    Add $item_i$ to K;
}
return K;
```

Figure 1. Pseudo-code for tested greedy algorithm

†Put the lighter item at lower index when the same profit density occurs.

```
// (problem) instance is given as an array of items
for each i-th item
{
  // Let the i-th item be '(v,w)'; v is its value and w is its weight
  Consider its eight neighbors†: $(v+1,w)$, $(v-1,w)$, $(v,w+1)$,
   $(v,w-1)$,$(v+1,w+1)$, $(v+1,w-1)$, $(v-1,w+1)$, and $(v-1,w-1)$ ;
   Take the one improving the quality of the instance best;
  Update (v,w) to the new one;
}
```

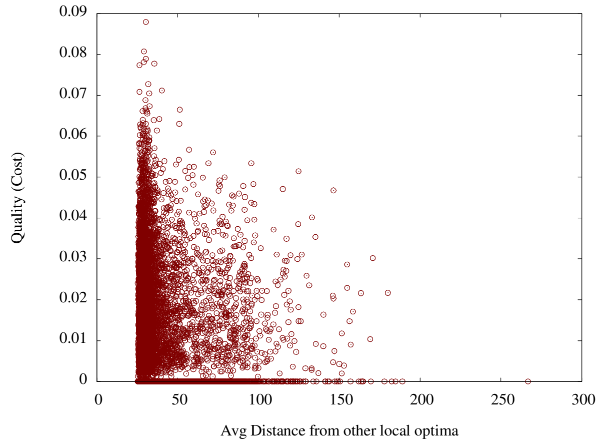
Figure 2. Local optimizer for 0/1KP instances

†We exclude any item of which the value or the weight is not in [1,100].

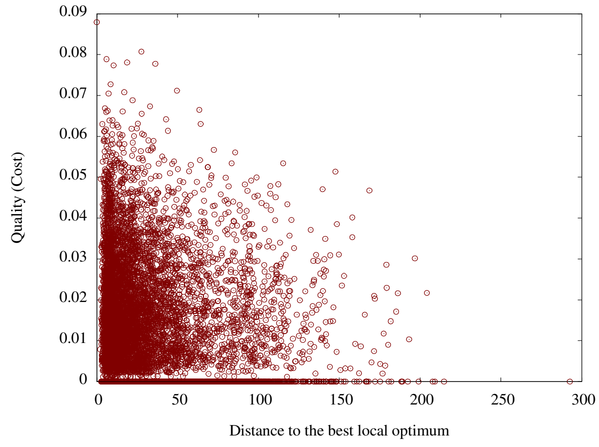
This section deals with the problem instance space of the 0/1 knapsack problem (0/1KP). In 0/1KP, we are given a knapsack and items. The knapsack has the capacity and each item has its own value and weight. The objective of 0/1KP is to choose items making their value sum be the highest, at the same time the weight sum not exceed the capacity of the knapsack.

We take as the tested algorithm the greedy algorithm which was dealt with in [9] (see Figure 1). The algorithm first sorts the given items by their profit density (value per unit weight) in non-increasing order. The lighter item is put first when the same profit density occurs. The algorithm tries to choose items in the sorted order. If the chosen item does not exceed the capacity of the knapsack, it is put into the knapsack. Otherwise, the algorithm tries to choose the next item according to the sorted order. The algorithm exits after considering the last item in the order.

This algorithm does not give an optimal solution in every case. As in [9], we define the quality of a problem instance as $1-P/O$, where P is the objective value of the solution obtained by the tested algorithm (the greedy algorithm fore-mentioned in this paper), O is the objective value obtained by the optimal algorithm (e.g., based on dynamic programming). In terms of optimality, an instance is the worst case if the quality is the highest. (This means the tested algorithm gives far-from-optimal solution for the instance.) The best case has the lowest quality.



(a) *dOthers* vs. quality



(b) *dBest* vs. quality

Figure 3. Cost-distance correlation (20 items, weight coefficient 0.5)

The method of getting cost-distance correlation is similar to one for the graph bi-partitioning problem given in [10]. Fore-mentioned 'quality' plays the role 'cost' plays in cost- distance correlation. In the search space, all the problem instances have the same number of items and the same weight coefficient. Here the weight coefficient is a real number in (0, 1) and determines the capacity of the knapsack by $\lfloor(\textit{weight coefficient})\times(\textit{weight sum of all the given items})\rfloor$. ($\lfloor.\rfloor$ means "the floor of".) Weights and values of items are limited to be integers in [1,100]. Observing the above constraints, we generate 10,000 instances randomly and take the local optimum of each by using the algorithm in Figure 2. Then we remove any redundant local optimum. For each local optimum, we calculate two kinds of numerical values. One is the average distance from all other local optima (we call it '*dOthers*'). The other one is the distance to the best local optimum (we call it

Table 1. Some statistics obtained from the method of [10] (# of local optima are 10,000)

#items	Weight coefficient	best Q	avg Q	std Q	max D	min D	avg D	std D	Corr(*dOthers*,Q)
10	0.25	0.389	0.035	0.054	271.708	0.346	23.185	24.241	–0.052
	0.5	0.207	0.019	0.031	274.740	0.247	23.160	24.202	–0.037
	0.75	0.156	0.010	0.019	271.610	0.234	23.169	24.216	0.010
20	0.25	0.176	0.021	0.024	301.207	0.858	37.670	30.634	–0.045
	0.5	0.088	0.012	0.014	304.889	0.924	37.591	30.587	–0.036
	0.75	0.058	0.006	0.008	301.276	0.884	37.677	30.631	–0.019
30	0.25	0.081	0.016	0.014	332.748	1.447	49.532	34.798	–0.035
	0.5	0.050	0.009	0.008	332.547	1.570	49.542	34.845	–0.019
	0.75	0.033	0.004	0.005	326.248	1.531	49.628	34.873	–0.015

'Q' means the quality, 'D' means distance. All statistics are from local optima. Local optimizer operates toward increasing the quality of the given instance. Corr(*dOthers*, Q) is the correlation coefficients between *dOthers* and quality.

'*dBest*'). For getting the distance, we regard a problem instance as an array of items, thus each item can be indexed. The definition of the distance is based on the Manhattan distance. We calculate the distance between *A* and *B* as follows:

$$d(A,B) = \sum_{i=1}^{n} \left| pd(a_i) - pd(b_i) \right|, \qquad (1)$$

where a_i and b_i are the *i*-th items of instances *A* and *B*, respectively. The function *pd*(.) returns the profit density of the given item. *N* is the number of items. (We denote '#items'.)

As the local optimizer to take a local optimum, we use the algorithm as in [9] (see Figure 2). The local optimizer tries to change the weight or value of each item in sequence, and test whether or not the quality of the problem instance increases. If it is true, the change is applied to the instance. Otherwise, the algorithm then considers the next item. It stops after it considers the last item. The changed instance becomes the local optimum for input instance of the local optimizer.

As a representative case, Figure 3 shows the correlation when #items is 20 and the weight coefficient is 0.5. Other correlations are similar to this case. Instances with high *dOthers* and *dBest* have low qualities. But the figures show different shapes from *solution* spaces of well-known combinatorial optimization problems such as graph bi-partitioning (see the shapes in [10]). In [10], when the values of *dBest* are small, the costs of solutions are also small. But Figure 3 shows variant qualities when the values of *dBest* are small. Instances with low *dOthers* and *dBest* could have low qualities. This indicates an obstacle in finding the worst case. On the other hand, there are many instances having the quality 0, which means the best case. This can be another obstacle in finding the worst case, because of the large number of the best cases. Also this result supports the result of [11]. In fact, [11] showed the distribution of this problem instance space is multi-modal.

Table 1 shows numerical data obtained by the fore-mentioned method. We calculate the correlation according to #items and weight coefficient. In our experiments, no local optimum was redundant. The average quality of local optima was quite close to 0. For the fixed #items, the average quality was decreasing as the weight coefficient is increasing. The correlation coefficients are all close to 0. If the coefficients were strong negative values, the search for the worst case would be straightforward.

3 Design of Improved Algorithm

Table 2. Worst-case instances found by the GA of [9] (# of items is 10)

Weight coefficient	Quality	Value sums obtained (Greedy algorithm / optimal one)
0.25	0.960	4 / 101
Instance	(1,1) (1,1) (100,100) (1,35) (1,37) (1,37) (1, 41) (1,46) (1, 51) (1,52) Capacity of knapsack = 101	
0.5	0.925	8 / 106
Instance	(1,1) (1,1) (1,1) (1,1) (1,1) (1,1) (1,1) (100,100) (66,100) (1, 4) Capacity of knapsack = 106	
0.75	0.916	9 / 107
Instance	(1,1) (1,1) (1,1) (1,1) (1,1) (1,1) (1,1) (1,1) (100,100) (1,34) Capacity of knapsack = 107	

*Items in each instance are represented by (*value*, *weight*).

In this section we show the generated test data can be used to find weak points of the tested algorithm and present an improved algorithm. Note that the main points are not the method for finding, but the improved algorithm based on the found. For finding the worst-case problem instance (as defined in Section 2), we use the same method as Jeon and Kim [9] used; a hybrid genetic algorithm (hybrid GA), where the tested algorithm is the greedy algorithm described in Figure 1. Also the local optimizer used in the hybrid GA is the same one as in Figure 2. For the fixed #items and weight coefficient, the method repeats the hybrid GA 50 times independently to achieve the statistical reliability. Jeon and Kim took as the worst case the problem instance showing the highest quality among 50 runs. The results were superior to those obtained by a random testing method.

Table 2 shows some of the worst cases obtained by the fore-mentioned hybrid GA. In Table 2, each item is represented as two-dimensional coordinates (v, w); v is its value and *w* is its weight. As an example, suppose that the greedy algorithm tries to solve the worst case in Table 2 when the weight coefficient is 0.5. (Refer to Figure 1 to see how this greedy algorithm works.) In the case, the algorithm takes seven (1,1)s and one (1,4). Then the value sum is 1×7+1×1=8. But six (1,1)s and one (100,100) makes the value sum 1×6+100=106. Since the algorithm prefers lighter items when items have the same profit density, it chooses (1,1) first before considering (100,100). One may suggest the modified algorithm that choose the item with higher value (although it is heavier item) first when considered items have the same profit density. His/her algorithm may overcome the problem instance in Table 2, but it does not perform better in every case than the original algorithm. Consider an instance having one (100,100), one (50,50), two (20,20)s, and four (1,73)s, with the weight coefficient 0.25 (the resulting capacity is 130). Then his/her algorithm will choose only one (100,100) whereas the original algorithm choose one (50,50) and four (20,20)s, which is a better solution. So we suggest a way conducting some post-processing after this algorithm, to perform better in every case than the original algorithm.

The proposed idea is as the following. (The pseudo-code is given in Figure 4.) Once the original greedy algorithm chooses items to be in the knapsack, the post-processing is done from the choice. It takes one of the subsets of the items in the knapsack, and swaps the subset for an item out of the knapsack, in the case that the swapping increases the value sum of the knapsack. Consider again the worst case in Table 2 when the weight coefficient is 0.5. In this case, the post-processing swaps (1,1) and (1,4) in the knapsack for (100,100) out of the knapsack; this achieves the increased value sum 106. This processing gives equal value sum to or higher value sum than one obtained by the original algorithm, at the same time it can overcome some of the worst cases of the original algorithm.

```
$U$ = list of all the given items;
$W$ = given capacity of knapsack;
$K = \{x \in U \mid x$ is chosen by the original greedy algorithm$\}$;
Sort $U-K$ as an array by profit density in non-increasing order[†];
for each $item_i \in U-K$
{
  for each $set_j \subset K$
  {
    $\Delta v_j$ = value of $item_i$ − $\sum$(value of each item$\in set_j$ );
    $\Delta w_j$ = weight of $item_i$ −$\sum$(weight of each item$\in set_j$ );
  }
  if($\forall j$, '$\Delta v_j < 0$' or $\sum$(weight of each item$\in K$ )+$\Delta w_j > W$ )
    break;
  $setB = set_j$ showing the highest $\Delta v_j$ ;[‡]
  Let $item_i$ be in $K$;
  Let $setB$ be in $U-K$;
  // we put $setB$ at the highest indices in $U-K$ (unsorted)
}
return $K$;
```

Figure 4. Pseudo-code for an improved greedy algorithm

†Put lighter item at lower index when the same profit density occurs.

‡When tie occurs, choose set_i showing the lowest Δw_i.

Table 3. Comparison between two algorithms

		Greedy		Greedy II		#samples: 10,000
#items	Weight coefficient	avg Q	sstd Q	avg Q	sstd Q	*p*-value
10	0.25	0.022	0.043	0.004	0.018	4.716E−304
	0.5	0.013	0.025	0.002	0.009	0[†]
	0.75	0.006	0.015	0.001	0.004	4.219E−273
20	0.25	0.012	0.019	0.003	0.009	0[†]
	0.5	0.007	0.011	0.002	0.005	0[†]
	0.75	0.003	0.006	0.001	0.002	0[†]
30	0.25	0.008	0.011	0.003	0.006	0[†]
	0.5	0.005	0.006	0.001	0.003	0[†]
	0.75	0.002	0.004	0.000	0.001	0[†]

†'0' means that it is quite less than 2.229E−308.
'Q' means quality. 'avg' and 'sstd' mean average and sample standard deviation, respectively. We obtained *p*-values by one-sided *t*-test.

But, considering combinations of the items in the knapsack takes so long time that the new algorithm is too slower than the original algorithm. To reduce the time, we limit the maximum size of the subset being considered to be 2 (i.e., in Figure 4, $|set_j| \leq 2$). Table 3 shows two kinds of the average quality of 10,000 random instances for each classification; one is obtained by taking the original algorithm (we denote 'Greedy') as the tested algorithm and the other one by taking the new algorithm (we denote 'GreedyII'). The low quality means the tested algorithm gives the solution close to the optimal one. The average qualities in the table indicate that the new algorithm is better than the original one.

4 Conclusions

In the approach that defines the performance of algorithms in abstract manner, the algorithm being tested need not to be well-known algorithms; the algorithm might be any part of the software being tested. This paper just illustrates the approach by taking the well-known problem and the well-known algorithm. (Of course, the problem and the algorithm we dealt with are important themselves and thus our experimental results can have some significance in the sense.) For our definition of the quality and given testing algorithm, the problem instance space was somewhat different from well-known solution space and seemed to be difficult to be searched. Also we showed how the test data can be used to devise an improved algorithm (in this paper, a greedy algorithm of the 0/1 knapsack problem); it could overcome some of the worst cases of the original greedy algorithm. Our results will accelerate further researches about the test data generation or empirical analysis of algorithms by generating test data using meta-heuristics.

We leave the following studies for future work: (i) finding again the worst case of the improved algorithm and examining the changes of the quality of the worst case, (ii) examining the cost-distance correlation of the improved algorithm, (iii) for the other problems and other definitions of the quality (e.g., one in terms of time complexity), investigating the problem instance space, (iv) applying different metrics and local optimizers to take the cost-distance correlation and comparing to the original correlation, (v) improving heuristic search algorithm by examining the problem instance space further and redefining the quality so that the landscape becomes more easy-to-search shape, and (vi) finding large-sized worst-case instances from small-sized ones to reduce the search time and the quality evaluation time.

Acknowledgments

This research was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education, Science and Technology(2011-0004215).

5 References

[1] D. Saab, Y. Saab, and J. Abraham. CRIS: A test cultivation program for sequential VLSI circuits. In *Proceedings of 1992 IEEE/ACM International Conference on Computer Aided Design*, pages 216- 219, 1992.

[2] M. Srinvas and L. Patnaik. A simulation-based test generation scheme using genetic algorithms. In *Proceedings International Conference VLSI Design*, pages 132-135, 1993.

[3] E. Rudnick, J. Patel, G. Greenstein, and T. Niermann. Sequential circuit test generation in a genetic algorithm framework. In *Proceedings of the 31st Annual Conference on Design Automation (DAC '94)*, pages 698-704, 1994.

[4] F. Corno, E. Sanchez, M. Sonza Reorda, and G. Squillero. Automatic test program generation - a case study. *IEEE Design & Test*, 21(2):102-109, 2004.

[5] P. McMinn. Search-based Software Test Data Generation: a Survey. *Software Testing Verification And Reliability*, 14(2): 105-156, 2004.

[6] M. Johnson and A. Kosoresow. Finding Worst-Case Instances of, and Lower Bounds for, Online Algorithms Using Genetic Algorithms. *Lecture Notes in Computer Science*, 2557:344-355, 2002.

[7] C. Cotta and P. Moscato. A Mixed-Evolutionary Statistical Analysis of an Algorithm's Complexity. *Applied Mathematics Letters*, 16(1):41-47, 2003.

[8] J. Hemert. Evolving combinatorial problem instances that are difficult to solve. *Evolutionary Computation*, 14(4):433-462, 2006.

[9] S.-Y. Jeon and Y.-H. Kim. A Genetic Approach to Analyze Algorithm Performance Based on the Worst-case Instances. *Journal of Software Engineering and Applications*, 3(8): 767-775, 2010.

[10] Y.-H. Kim and B.-R. Moon. Investigation of the Fitness Landscapes in Graph Bipartitioning: An Empirical Study. *Journal of Heuristics*, 10(2):111-133, 2004.

[11] S.-Y. Jeon and Y.-H. Kim. Finding the Best-case Instances for Analyzing algorithms: Comparing with the Results of Finding the Worst-case Instance. In *Proceedings of the Korea Computer Congress 2010*, vol. 37, no. 2(C), pages 145-150, 2010. (in Korean)