Worst-Case Execution Time Test Generation for Solutions of the Knapsack Problem Using a Genetic Algorithm

Maxim Buzdalov and Anatoly Shalyto

ITMO University
49 Kronverkskiy prosp.
Saint-Petersburg, Russia, 197101
mbuzdalov@gmail.com, shalyto@mail.ifmo.ru

Abstract. Worst-case execution time test generation can be hard if tested programs use complex heuristics. This is especially true in the case of the knapsack problem, which is often called "the easiest NP-complete problem". For randomly generated test data, the expected running time of some algorithms for this problem is linear. We present an approach for generation of tests against algorithms for the knapsack problem. This approach is based on genetic algorithms. It is evaluated on five algorithms, including one simple branch-and-bound algorithm, two algorithms by David Pisinger and their partial implementations. The results show that the presented approach performs statistically better than generation of random tests belonging to certain classes. Moreover, a class of tests that are especially hard for one of the algorithms was discovered by the genetic algorithm.

Keywords: knapsack problem, test generation, genetic algorithms, worst-case execution time.

1 Introduction

What is the input data which makes your program work too long? What is the maximum delay between an input event and an output action? These questions are related to the problem of determining the worst-case execution time of a program or a procedure.

It is impossible to write a program that determines the worst-case execution time of an arbitraty program, as follows from the Rice theorem [17]. However, one may hope to find an approximate answer. This makes it reasonable to apply search-based optimization techniques, such as genetic algorithms, to the problem of worst-case execution time test generation [3, 4].

Many works on evolutionary worst-case execution time test generation consider the case of real-time software testing [2, 10–12, 18, 19]. Another application where worst-case execution time matters is test generation for programming challenge tasks [7]. There are some papers on related topics of testing discrete mathematics algorithms [5, 6] and SAT solvers [13].

L. Pan et al. (Eds.): BIC-TA 2014, CCIS 472, pp. 1–10, 2014.

In many papers on worst-case execution time test generation, the execution time is optimized directly. However, measurements of the execution time contain noise introduced by the operating system scheduling and the measured time is often proportional to a certain time quantity [7]. To overcome these difficulties, paper [7] advises to instrument the tested code with special counters. Each of these counters is associated with a loop or a procedure and incremented each time the loop is entered or the procedure is called. The values of these counters, as well as the execution time, serve as optimization objectives.

The *0-1 knapsack problem* (will be referred to as simply *knapsack problem* in the rest of the paper) is a well-known combinatorial optimization problem. N items are given, each characterized by weight w_i and profit p_i, and a knapsack with a weight capacity of W. One needs to find a subset of items of the maximum total profit that can be placed into the knapsack, i.e. to determine $x_i \in \{0, 1\}$ such that $\sum_{i=1}^{N} w_i x_i \le W$ and $\sum_{i=1}^{N} p_i x_i$ is maximum possible.

The knapsack problem is weakly NP-hard [9], and is often called "the easiest NP-problem", because there exist algorithms that solve a large fraction of big instances in polynomial (even linear) time [15]. In [8], a way to construct hard test cases (in terms of the number of elementary operations) for a certain class of algorithms is given as follows:

$$p_j = w_j = 2^{K+N+1} + 2^{K+j} + 1,$$

where N is the number of items, $K = \lfloor \log 2N \rfloor$, $1 \le j \le N$.

In the test case construction scheme given above, weights and profits of items are exponential in the size of the problem. However, it may happen that in a particular setup weights and profits are bound by a constant, so such test cases are no longer valid.

We present an approach based on genetic algorithms to generate test cases against algorithms for the knapsack problem. The approach is compared with random test generation according to some known patterns [15] and it is shown that the evolutionary approach works statistically better. For one of the algorithms, the genetic algorithm discovered a new class of hard test cases.

The rest of the paper is structured as follows. In Section 2, the algorithms for the knapsack problem are described along with the fitness functions used to measure their performance. Section 3 outlines the ways to generate random test data for the knapsack problem, known from [15]. In Section 4, the genetic algorithm is described that is used to generate tests against the algorithms for the knapsack problem. Section 5 contains experimental results both for the genetic algorithm from Section 4 and for random tests from Section 3. Some analysis of the generated tests is included as well, which leads to a new class of knapsack problems that are hard for one of the algorithms. Section 6 concludes.

2 Algorithms for the Knapsack Problem

This section describes the algorithms for the knapsack problem that are tested in this paper. The first algorithm is a simple branch-and-bound algorithm, called SimpleBranch in this paper. The second one is the algorithm ExpKnap from [15].

The third one is the partial implementation of the ExpKnap, that is, an implementation with several heuristics omitted, which makes it run slower than ExpKnap under certain circumstances. The fourth one is the algorithm HardKnap, also from [15]. The fifth one is the partial implementation of HardKnap.

Below, we describe the SimpleBranch algorithm, outline the algorithms ExpKnap and HardKnap and partial implementations of the two latter algorithms. Performance measures of all these algorithms, which are used as fitness functions, are also discussed.

2.1 Simple Branch-and-Bound Algorithm

We start with an algorithm which we call SimpleBranch. The ideas of the algorithm are very simple and mostly come from common sense:

- sort the items in decreasing order of their profit-per-weight ratio (as is commonly done [15]);
- store the best known solution in a dedicated variable (initially an empty solution);
- for each item, starting from the first one, first try to put it to the knapsack, second try to put it aside;
- return from the current branch if the current profit plus the profit of the yet-to-consider items is less than or equal to the profit of the best known solution;
- don't ignore an item if the maximum possible weight for the current state does not exceed the knapsack capacity.

The algorithm is implemented as a single recursive function and a small number of preparation utilities. The amount of work for a single call to this function is constant, however, the number of calls can be large. The performance measure used for this algorithm is the number of calls to this function.

2.2 ExpKnap

The ExpKnap algorithm is described in detail in [15]. The name of the algorithm comes from the concept of *expanding core*. Consider all items in decreasing order of their profit-per-weight ratio. Let G be a solution to the knapsack problem constructed by taking items to the knapsack starting from the first one while the items can fit into the knapsack. The first item that does not fit is called the *break item*.

The algorithm ExpKnap considers solutions to the knapsack problem encoded as differences from solution G (i.e. a set of elements that are present in G but not present in the solution, or vice versa). The idea of the algorithm comes from the fact that in most problem instances the elements that differ from solution G are concentrated in a small area around the break item. This area is called the *core* [15]. Enumerating all the possible solutions with a small fixed core size is cheap. However, guessing the correct core size is a difficult problem, and the ExpKnap algorithm solves it by expanding the core when needed.

The partial implementation of the ExpKnap algorithm, which is used in this paper along with the original ExpKnap algorithm, uses a simple core expansion procedure, which just adds to the core the items that are adjacent to it. The original algorithm uses a more complicated procedure, which may change the order of elements to achieve better performance. It is thus expected to see the cases when the partial implementation performs worse than the original algorithm.

Implementations of ExpKnap, similarly to SimpleBranch, contain a recursive procedure which performs almost all computations. In both partial and full implementations of ExpKnap, the performance measure is the number of calls to this procedure.

2.3 HardKnap

The HardKnap algorithm is described in detail in [15]. This is an algorithm from the dynamic programming family. It solves the knapsack problem by solving subproblems of the following kind: enumerate all sets of items, which have indices from L to R, such that no set is dominated by any other set. Set A dominates set B if the profit of A is not smaller than the profit of B, while the weight of A is not larger than the weight of B. A subproblem $[L; R]$ can be solved by solving two subproblems $[L; M]$ and $[M + 1; R]$ and then merging the sets. For the top-level problem $[1; N]$, one does not need to find all the sets — it is enough to find a set with the larges profit and with a weight not exceeding the knapsack capacity, which can be done in linear time.

The original HardKnap algorithm constructs upper bounds on the possible solutions to the knapsack problem for each set and eliminates the sets for which this bound is too low, thus reducing the number of considered sets and decreasing the running time. The partial implementation of the HardKnap algorithm, which is considered in this paper along with the original one, does not do this kind of elimination. Effectively, this is an implementation of the Nemhauser and Ullmann algorithm [14].

The performance measure for both HardKnap implementations is the number of set merge events. Each time the upper bound construction and filtering, which is present in the original implementation but is missing in the partial implementation, makes the difference, the number of set merge events decreases, so we can compare both implementations by comparing their performance measures.

3 Tests for the Knapsack Problem

There are several strategies to generate tests for the knapsack problems. We consider three of them [15]:

– **Uncorrelated Tests.** In these tests, no dependency between weights and profits of items is specified. To generate random tests of this sort, one may set the weight w to a random integer between 1 and the maximum weight W_{max}, and set the profit p to a random integer between 1 and the maximum profit P_{max}.

- **Strongly Correlated Tests.** The difference between the profit and the weight of an item is equal to a small number d. To generate random tests of this sort, one may set the weight w to a random integer between $\max(1, 1-d)$ and $\min(W_{max}, P_{max} - d)$ and set the profit $p = w + d$.
- **Subset Sum Tests.** These are the tests where weights are equal to profits. Such instances of the knapsack problem are also the instances of the *subset sum* problem. To generate random tests of this sort, set the weight w to a random integer between 1 and $\min(W_{max}, P_{max})$ and set the profit $p = w$.

It should be noted that uncorrelated tests correspond to the general case, i.e. no constraints are put on tests. As in [15], we choose the capacity of the knapsack to be half the sum of weights of all items.

4 Genetic Algorithm

This section describes the encoding of a problem instance as an individual of the genetic algorithm, the evolutionary operators that are used, and some other settings of the genetic algorithm. The fitness functions to be used with different algorithms for the knapsack problem have been described in the corresponding subsections of Section 2.

We perform three series of experiments with the genetic algorithm, corresponding to three considered types of tests (Section 3). The new individual creation procedure and the mutation operator are different for each test type, all other operators and settings remain the same.

There are three parameters imposed by the knapsack problem: the number of items N, the maximum item weight W_{max} and the maximum item profit P_{max}. For the strongly correlated test type, the difference parameter d is also used. All these parameters are fixed during each run of an algorithm.

The **individual** is encoded as a list of items, where each item has a specified weight and profit. As said in Section 3, the capacity of the knapsack is chosen to be half the sum of weights.

The procedure of **new individual creation** creates a list of N items, where each item is generated at random as specified in Section 3 for the corresponding type of test.

The **mutation operator** works as follows. It selects a random item and mutates it according to the procedure described below. Then it terminates with the probability of 0.5, otherwise it repeats the process. Such operator is capable of making an arbitrarily large mutation, but prefers small mutations. The single item mutation procedure, depending on the test type, does the following:

- for uncorrelated tests — adds $d_w \sim \lfloor N(0,1) \cdot W_{max}/3 \rfloor$ to the weight, $d_p \sim \lfloor N(0,1) \cdot P_{max}/3 \rfloor$ to the profit, and then fits the weight to the $[1; W_{max}]$ interval and the profit to the $[1; P_{max}]$ interval.
- for strongly correlated tests — adds $d_w \sim \lfloor N(0,1) \cdot \min(W_{max}, P_{max})/3 \rfloor$ to the weight w, fits it to the $[\max(1, 1-d); \min(W_{max}, P_{max}+d)]$ interval and sets the profit to $w + d$.

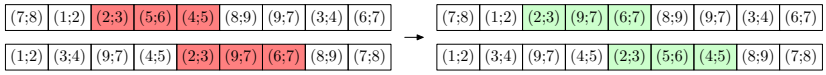

Fig. 1. Illustration of the two-point crossover with shift

– for subset sum tests — adds $d_w \sim \lfloor N(0,1) \cdot \min(W_{max}, P_{max})/3 \rfloor$ to the weight, fits it to the $[1; \min(W_{max}, P_{max})]$ interval and sets the profit equal to the weight.

The mutation operator is applied with a probability of 0.1.

The **crossover operator** is a two-point crossover with shift similar to the one proposed in [5]. This variant of crossover also performed well in [7] on a problem which is similar to the knapsack problem. This operator takes two individuals and returns also two individuals. It selects an exchange length L randomly from 1 to $N-1$. Then it selects offsets O_1 and O_2 for the first and second individuals randomly and independently from 1 to $N-L$. After that, the subsequences of length L at offset O_1 in the first individual and at offset O_2 in the second individual are exchanged (see Fig. 1). The crossover operator is applied with a probability of 1.0.

It is worth noting that, in the case of subset sum tests and strongly correlated tests both mutation and crossover operators always produce tests of the type that is equal to the type of the arguments. For example, a mutation test will produce a strongly correlated test if a strongly correlated test is given. This means that if the genetic algorithm is initialized with, for example, strongly correlated tests, the results will be strongly correlated as well. In the case of uncorrelated tests, that is, in the general case, tests can become structured spontaneously during the evolution.

The **reproduction selection** is a variant of tournament selection. For each individual to be selected, eight individuals are selected at random. Then the individuals are grouped in pairs, and in each pair a tournament is conducted — with a probability of 0.9, the individual with higher fitness wins. The process is repeated with the winning individuals until only one is left. We borrow this selection operator from [7]. The **survival selection** is the elitist selection with the elite rate of 20%.

5 Experiments

This section describes the experiment setup and results.

5.1 Setup

In our experiments we considered three types of tests described in Section 3. For each type and each algorithm for the knapsack problem, we compared the genetic

algorithm with random test generation. To do that, we performed 100 runs of each algorithm, and the number of fitness evaluations was limited to 500 000. Thus, the computational budget was the same for random test generation and for the genetic algorithm. For each run, the best individual is considered to be the result.

The generation size in the genetic algorithm was set to 50. The knapsack problem parameters were $N = 20$, $W_{max} = P_{max} = 10000$, the difference for strongly correlated tests $d = 5$. We chose such a small problem size because we wanted even the weakest algorithms to terminate in reasonable time.

The source code is available on GitHub [1].

5.2 Results

The minima, maxima, averages and means of best fitness values are presented in Table 1 for all considered configurations. To measure statistical difference between results for different configurations, the Wilcoxon rank sum test implemented in the R system [16] is used with the alternative hypothesis set to "greater" when the mean of the results for the first configuration is greater than the mean for the second one, and to "less" otherwise. We reject the null hypothesis if the p-value is less than 0.001.

Randomly Generated Tests. First, it can be seen that random uncorrelated tests are very simple for all considered algorithms. Even for SimpleBranch the average fitness value for random uncorrelated tests is approximately 10 times smaller than for the hardest test found. On the other hand, random strongly correlated tests and random subset sum tests are rather hard. Note that for SimpleBranch strongly correlated tests are noticeably (p-value less than $2.2 \cdot 10^{-16}$, as reported by R) harder than subset sum tests, while for other algorithms subset sum tests are harder (p-value less than $2.2 \cdot 10^{-16}$ for ExpKnap, partial ExpKnap and partial HardKnap; p-value equal to $1.785 \cdot 10^{-14}$ for HardKnap).

Random Tests vs. Genetic Algorithm. For each algorithm for the knapsack problem and for each type of tests, we compared statistically the results for random test generation and for genetic algorithms. For all configurations except for two, the p-value was reported to be less than $2.2 \cdot 10^{-16}$. For HardKnap and strongly correlated tests, it was equal to $1.717 \cdot 10^{-15}$, and for partial HardKnap and subset sum tests, it was equal to 0.01209.

This means that the genetic algorithm consistently produces statistically better results than random test generation. It seems that the small difference for the last configuration can be explained by the fact that the hardest possible problem instances (fitness value 15 119) were generated with high probability.

Table 1. Experimental results

Optimizer	Individual	Min	Max	Median	Mean	Deviation
SimpleBranch						
best of random	uncorrelated	90882	211530	108851.0	112820.74	17516.91
	strongly correlated	773333	881402	804713.0	809474.83	23793.61
	subset sum	671104	784717	708195.5	714033.65	27280.72
genetic	uncorrelated	520685	1048576	1048576.0	984013.90	157357.85
	strongly correlated	1048576	1048613	1048576.0	1048584.09	8.77
	subset sum	1048576	1048604	1048576.0	1048580.81	6.92
ExpKnap						
best of random	uncorrelated	393	1269	539.0	583.51	165.13
	strongly correlated	62627	156475	76391.5	84656.27	21268.04
	subset sum	176555	447333	348194.5	337375.98	64381.87
genetic	uncorrelated	428298	688129	447746.0	496336.91	81824.28
	strongly correlated	428298	512278	429182.0	459105.40	32676.90
	subset sum	696321	699041	698881.0	698603.88	419.75
Partial implementation of ExpKnap						
best of random	uncorrelated	492	2223	683.0	724.38	199.72
	strongly correlated	63915	164171	80573.5	86059.53	21265.52
	subset sum	154812	445699	322171.0	312934.88	77359.58
genetic	uncorrelated	369226	688129	447746.0	487032.44	72677.54
	strongly correlated	428298	541348	485316.0	465577.64	32425.55
	subset sum	698369	699009	698881.0	698726.52	260.21
HardKnap						
best of random	uncorrelated	771	1252	941.0	963.03	104.89
	strongly correlated	13700	14401	13789.0	13811.16	103.32
	subset sum	13068	15113	15022.5	14645.37	756.27
genetic	uncorrelated	3686	11457	8075.5	7892.86	1885.56
	strongly correlated	13682	15113	14062.5	14205.19	449.63
	subset sum	15113	15113	15113.0	15113.00	0.00
Partial implementation of HardKnap						
best of random	uncorrelated	5660	6970	6230.5	6278.98	333.20
	strongly correlated	15054	15074	15054.0	15054.35	2.29
	subset sum	15114	15119	15119.0	15118.80	0.94
genetic	uncorrelated	9859	13790	12318.5	12219.61	699.25
	strongly correlated	15060	15114	15114.0	15113.46	5.37
	subset sum	15119	15119	15119.0	15119.00	0.00

Partial Implementations. We cannot compare algorithms from different families by performance, but we can compare algorithms with their partial implementations. In this section, unlike the others, we use the two-sided alternative hypothesis.

For ExpKnap algorithms, statistic analysis reports surprising results — it is possible to distinguish the full and the partial implementations only at uncorrelated random tests (p-value is $1.83 \cdot 10^{-12}$). For subset sum tests, p-values are 0.02643 and 0.01595 for random tests and the genetic algorithm correspondingly.

For strongly correlated tests by the genetic algorithm it is 0.1346, and for other configurations it exceeds 0.5. It can be deduced that the difference between the implementations improves the average-case performance, but does not influence the worst case.

For HardKnap algorithms, however, full and partial versions are always distinguishable ($p < 2.2 \cdot 10^{-16}$).

Hard Test Analysis. We analyzed the hard test cases for ExpKnap, because the genetic algorithm constructs tests that are much harder than random tests.

The hardest test for ExpKnap with a fitness value of 699041 belongs to the group of subset sum tests. It consists of five items with weight and value of 1, three items of 34, one item of 83, two items of 265, one item of 335, two items of 614, one item of 1696, two items of 3842, one item of 5887, and two items of 10 000. Many other hard tests from the same group feature many items of 1 and of 10 000, but no particular structure is seen.

One of the hardest tests for ExpKnap among the strongly correlated tests (fitness value 512278) has only two types of items: one item with weight 9596 and profit 9601 and 19 items with weight 3030 and profit 3035.

A small experiment was performed to test if the latter type of tests is hard enough. For each algorithm, ten runs were performed, each for 100 000 fitness evaluations. The best fitness for each algorithm was the same for all runs. For SimpleBranch it was 1048576, for ExpKnap family — 541348, for HardKnap — 799 and for partial HardKnap — 803. It appears that such tests are hard, but not the hardest, for SimpleBranch and the ExpKnap family and easy for HardKnap.

6 Conclusion

We presented an approach to worst-case execution time test generation for algorithms solving the knapsack problem. This approach is based on genetic algorithms. Statistical analysis of the experimental results showed that the genetic algorithm consistently produces harder tests than random test generation.

A new class of tests for the knapsack problem was also found — tests with two types of items — that appeared to be very hard for the ExpKnap algorithm family, but they are easy for another algorithm, HardKnap.

This work was partially financially supported by the Government of Russian Federation, Grant 074-U01.

References

1. Source code for experiments (a part of this paper), `https://github.com/mbuzdalov/papers/tree/master/2014-bicta-knapsacks`
2. Alander, J.T., Mantere, T., Moghadampour, G.: Testing Software Response Times Using a Genetic Algorithm. In: Proceedings of the 3rd Nordic Workshop on Genetic Algorithms and their Applications, pp. 293–298 (1997)

3. Alander, J.T., Mantere, T., Turunen, P.: Genetic Algorithm based Software Testing. In: Proceedings of the 3rd International Conference on Artificial Neural Networks and Genetic Algorithms, Norwich, UK, pp. 325–328 (April 1997)
4. Alander, J.T., Mantere, T., Turunen, P., Virolainen, J.: GA in Program Testing. In: Proceedings of the 2nd Nordic Workshop on Genetic Algorithms and their Applications, Vaasa, Finland, August 19-23, pp. 205–210 (1996)
5. Arkhipov, V., Buzdalov, M., Shalyto, A.: Worst-Case Execution Time Test Generation for Augmenting Path Maximum Flow Algorithms using Genetic Algorithms. In: Proceedings of the International Conference on Machine Learning and Applications, vol. 2, pp. 108–111. IEEE Computer Society (2013)
6. Buzdalov, M.: Generation of Tests for Programming Challenge Tasks on Graph Theory using Evolution Strategy. In: Proceedings of the International Conference on Machine Learning and Applications, vol. 2, pp. 62–65. IEEE Computer Society (2012)
7. Buzdalova, A., Buzdalov, M., Parfenov, V.: Generation of Tests for Programming Challenge Tasks Using Helper-Objectives. In: Ruhe, G., Zhang, Y. (eds.) SSBSE 2013. LNCS, vol. 8084, pp. 300–305. Springer, Heidelberg (2013)
8. Chvatal, V.: Hard Knapsack Problems. Operations Research 28(6), 1402–1411 (1980)
9. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York (1979)
10. Gross, H.G.: A Prediction System for Evolutionary Testability Applied to Dynamic Execution Time Analysis. Information and Software Technology 43(14), 855–862 (2001)
11. Gross, H.G., Jones, B.F., Eyres, D.E.: Structural Performance Measure of Evolutionary Testing Applied to Worst-Case Timing of Real-Time Systems. IEEE Proceedings - Software 147(2), 25–30 (2000)
12. Gross, H.G., Mayer, N.: Search-based Execution-Time Verification in Object-Oriented and Component-Based Real-Time System Development. In: Proceedings of the 8th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems, pp. 113–120 (2003)
13. Moreno-Scott, J.H., Ortíz-Bayliss, J.C., Terashima-Marín, H., Conant-Pablos, S.E.: Challenging Heuristics: Evolving Binary Constraint Satisfaction Problems. In: Proceedings of Genetic and Evolutionary Computation Conference, pp. 409–416. ACM (2012)
14. Nemhauser, G., Ullmann, Z.: Discrete dynamic programming and capital allocation. Management Science 15(9), 494–505 (1969)
15. Pisinger, D.: Algorithms for Knapsack Problems. Ph.D. thesis, University of Copenhagen (February 1995)
16. R Core Team: R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria (2013), `http://www.R-project.org/`
17. Rice, H.G.: Classes of recursively enumerable sets and their decision problems. Transactions of the American Mathematical Society 74(2), 358–366 (1953)
18. Tlili, M., Wappler, S., Sthamer, H.: Improving Evolutionary Real-Time Testing. In: Proceedings of Genetic and Evolutionary Computation Conference, pp. 1917–1924. ACM (2006)
19. Wegener, J., Mueller, F.: A Comparison of Static Analysis and Evolutionary Testing for the Verification of Timing Constraints. Real-Time Systems 21(3), 241–268 (2001)