Implementing Algorithms for the Binary (0-1) Knapsack Problem

Dušan B. Gajić[1]

***Abstract* – The knapsack problem is one of the most important and, at the same time, most fascinating NP-complete problems in the field of combinatorial optimization. From a practical point of view, it is interesting to estimate which knapsack problem instances can be efficiently handled, in terms of both qualitative and quantitative parameters. Another problem is to determine how much decisions made on implementation issues, e.g., application of code tuning techniques and compiler optimizations, affect actual performance of algorithms for the knapsack problem. This paper presents the results and conclusions drawn from the computational experiments done with C/C++ implementations of both classical and recent algorithms for the binary (0-1) knapsack problem. These conclusions can be useful as guidelines in practical applications. Further, analyzing these results could be helpful in creating directions for future research.**

***Keywords* – binary (0-1) knapsack problem, computational experiments, C/C++ algorithm implementations, code tuning techniques, compiler optimizations**

I. INTRODUCTION

When discussing algorithm performance in computer science, we are accustomed to dealing with terms that inhabit the realm of mathematical abstraction, like the asymptotic notation and the worst-case running time. But what is the real-world performance of algorithms that are actually implemented and executed on a computer? What types of problem instances can be practically solved, measured both quantitatively and qualitatively? Which decisions can a programmer make in order to speed up a computational process guided by an implemented algorithm? Thinking about these questions, stated solely in the context of the binary knapsack problem, motivated the research reported in this paper. In order to see the actual experimental results from an appropriate standpoint, a reader should be first introduced to the landscape of the binary knapsack problem.

The classical binary or 0-1 knapsack is an NP-hard [12] maximization problem defined by the following setting: given a list of n items, characterized by their respective values v_i ($i = 1, 2,\ldots, n$) and weights w_i ($i = 1, 2,\ldots, n$), find a subset of these items that can be packed in a knapsack, with maximum carrying weight of W, whose total value is a maximum. Without any loss of generality, we can assume that all of the weights and values of items are positive integers. The Integer Linear Programming (ILP) model of the 0-1 knapsack is one of the simplest and it is created by introducing the Boolean decision variables x_i, with $x_i = 1$ if the item i is selected to be included, and $x_i = 0$ otherwise. Formally, the ILP model of the binary knapsack can be formulated as follows:

$$\text{maximize} \sum_{i=1}^{n} v_i x_i \tag{1}$$

$$\text{subject to} \sum_{i=1}^{n} v_i x_i \le W \tag{2}$$

$$x_i \in \{0,1\}, i \in \{1,\ldots, n\}$$

The sum in Eq. (1) is called *the objective function*, and the inequality in Eq. (2) is *the constraint*.

The 0-1 knapsack problem is actually a member of a large family of knapsack problems, presented in detail in [12], and, more recently, in [6], [8]. Knapsack problems arise naturally in many real-world applications, e.g., scheduling problems, capital budgeting, resource allocation with financial constraints, cargo loading, and cutting stock. In applied mathematics, especially operations research and cryptography, knapsack problems were studied and used in numerous contexts (e.g., Merkle-Hellman knapsack cryptosystem).

The binary knapsack problem can be solved by means of exact and approximation algorithms. These algorithms have different performances, the study of which, in the light of various implementation issues, is reported in this paper. The chosen representative of the family of the exact algorithms for the 0-1 knapsack is the state of the art combo algorithm, described in [10]. An approximation algorithm chosen for implementation and testing is the fully polynomial time approximation scheme, described in [16]. Its selection was motivated by two different reasons. First, it offers a representative behavior of the approximation algorithms with errors arbitrarily close to zero at an acceptable implementation cost for testing purposes. Second, it is a computationally and memory intensive algorithm, and thus it is suitable as the benchmark for testing different programming implementations.

The data acquired in the presented research can be useful at least for the following purposes:

i) Creating recommendations for selection and implementation of the algorithms for solving 0-1 knapsack instances that arise in practical applications.
ii) Developing guidelines for possible future research through understanding the limitations and bottlenecks of the current algorithms.

[1]Dušan B. Gajić is with the Computational Intelligence and Information Technology Laboratory, Department of Computer Science, Faculty of Electronic Engineering, A. Medvedeva 14, 18000 Niš, Serbia, E-mail: dusan.gajic@elfak.rs

II. Exact Algorithms for the 0-1 Knapsack Problem

First exact algorithms for the 0-1 knapsack problem were developed in the 1950s, using the dynamic programming paradigm developed by Bellman [4].

Kolesar in 1967 was the first one to construct a branch-and-bound algorithm for the 0-1 knapsack problem (for the best algorithm created through this approach see [12]). Branch-and-bound algorithms can effectively deal only with small and easy problem instances [11].

Balas and Zemel [3] proposed to randomly set the optimal values of some of the decision variables, and then focus the enumeration on the most promising ones. The subset of the items built in this way is called the core. The core problem can then be solved by either heuristics or branch-and-bound algorithms. This approach proved to be helpful in solving larger problem instances.

Martello, Pisinger and Toth [10] proposed a hybrid technique of combining dynamic programming with tight bounds. This algorithm, called the combo algorithm, has the theoretical worst-case time complexity of $O(nW)$, but it has proven in practice to be the state of the art in solving 0-1 knapsack problem instances to optimality [8], [10], [11]. As a consequence of this fact, an implementation of the combo algorithm in C by Pisinger, available from [7], was chosen as the one of the subjects of the tests that were conducted in the presented research.

III. Approximation Algorithms for the 0-1 Knapsack Problem

An approach based on settling for an approximate solution to the 0-1 knapsack problem offers reasonable algorithm running times, even for the problem instances with exponentially-growing values and weights.

A reasonable way to measure the distance between an approximate and an optimal solution is the relative performance ratio ρ which bounds the value of maximum ratio between an approximate and an optimal solution. For maximization problems, relative difference or error ε is defined as $1-\rho$. Thus we refer to $(1-\varepsilon)$-approximation algorithms when the performance ratio of an approximate solution to an optimal one is larger than or equal to $1-\varepsilon$. An algorithm is an ε-approximation scheme if it is a $(1-\varepsilon)$-approximation algorithm for every input value of $\varepsilon > 0$. Further, an algorithm is said to be a polynomial time approximation scheme, abbreviated PTAS, if, for each fixed $\varepsilon > 0$, its running time is polynomial in the input size n. If the previous definition is expanded in a way to require that the running time of an algorithm is polynomial both in n and in $1/\varepsilon$, we get a fully polynomial time approximation scheme, abbreviated FPTAS.

The binary knapsack problem is a member of a "fortunate" class of NP-complete problems in terms of approximation, i.e., polynomial time approximations with errors arbitrarily close to zero exist for it. The first FPTAS for the knapsack problem was presented in 1975 [5]. A FPTAS by Keller and Pferschy [8] has recently improved on all of the previous schemes, but only under the assumption that $n \geq 1/\varepsilon$ and at a high implementation cost, due to the fact that the algorithm is fairly complicated (as stated by the authors themselves in [8]). Based on this fact, as well as the other reasons already stated in Section 1, a FPTAS described by Vazirani [16] was selected for implementation.

Basically, all of the fully polynomial time approximation schemes for the 0-1 knapsack problem follow the same approach of scaling values of items in order to reduce the number of different total sums of values, and then apply dynamic programming by values to a scaled instance. For the implemented algorithm, this scaling of the values of items is done in the following way:

$$v_i^* = \left\lfloor \frac{v_i}{v_{\max}} \cdot \frac{n}{\varepsilon} \right\rfloor \tag{3}$$

where $\varepsilon > 0$ and $v_{\max} = \max v_i$, $i \in \{1,...,n\}$

To sum up, the design of almost all known approximation schemes is based on the idea of trading accuracy for time. The given problem instance is transformed into a less "precise" one, depending on the concrete value for ε, which is then solved exactly by means of dynamic programming. The question whether PTAS or FPTAS is the best scheme we can hope for when dealing with an NP-hard problem is difficult and it has no clear and straightforward answer [16].

IV. Programming Implementations and Computational Experiments

In order to conduct the experiments, a test generator for different types of input instances had to be first built. The C++ test environment using this generator was constructed in a generic way, so that it can be reused. It allows observations of behavior of a wide range of algorithm implementations for different problem sizes, instance types, and data ranges. Problem instance sizes used in testing span from 50 to 10 000 items, and data range R takes value from the following set: $\{10^3, 10^5, 10^7\}$. Generated instance types are in accordance with test data sets presented in [8], and they are as follows:

a) Uncorrelated: the weights w_i and the values v_i are distributed uniformly random in $[1, R]$.

b) Weakly correlated: the weights w_i are distributed uniformly random in $[1, R]$, and the values v_i are set in $[w_i - R/10, w_i + R/10]$ and $v_i \geq 1$.

c) Strongly correlated: the weights w_i are distributed uniformly random in $[1, R]$, and the values v_i are set to $v_i = w_i + R/10$.

d) Subset-sum: the weights w_i are distributed uniformly random in $[1, R]$, and the values v_i are set to $v_i = w_i$.

For each of the instance types, data ranges, and problem sizes, 100 input instances were generated. The knapsack capacity was set to a value of: $W = 0.5 \sum_{i=1}^{n} w_i$. All of the experiments were done using a PC with *Intel Core i7 920* quad-core processor and 4 GBs of RAM, running a 64-bit *Kubuntu 9.10* operating system with *Linux kernel 2.6.31.19*. The C/C++ source code was compiled using the *GNU Compiler Collection* (*gcc*), version 4.4.1. The CPU times,

appearing in all of the tables in this section, correspond only to the execution of the algorithms themselves and do not include time used for test generation and I/O operations.

The combo algorithm was tested first. The test results for the computationally easy instances are presented in Table I, and for the computationally hard ones in Table II. An in-depth explanation of why the uncorrelated and the weakly correlated instances are easier for computation than the strongly correlated and the subset-sum instances can be found in [8]. It is important to state here that the problem instances that are most often met in practice correspond to the weakly correlated type. The test results clearly show that the combo algorithm has an outstanding performance for almost all of the instances. But, it also has its limitations, as we can see from Table II. It cannot deal with instances that are both computationally hard and large in range ($R \geq 10^7$). This was expected as a consequence of the NP-hardness of the 0-1 knapsack problem.

TABLE I
COMBO ALGORITHM, EASY INSTANCES, CPU TIMES [MILLISECONDS], n - INPUT SIZE, R - DATA RANGE

n \ R	*uncorrelated*			*weakly correlated*		
	10^3	10^5	10^7	10^3	10^5	10^7
50	0.01	0.01	0.01	0.03	0.03	0.04
100	0.02	0.02	0.02	0.04	0.06	0.06
500	0.05	0.06	0.06	0.10	0.22	0.25
1000	0.07	0.09	0.08	0.13	0.46	0.47
5000	0.27	0.35	0.37	0.30	2.26	2.56
10000	0.38	0.72	0.76	0.73	4.42	6.02

TABLE II
COMBO ALGORITHM, HARD INSTANCES, CPU TIMES [MILLISECONDS], n - INPUT SIZE, R - DATA RANGE, – STANDS FOR ALGORITHM TERMINATED WITH NO RESULT

n \ R	*strongly correlated*			*subset-sum*		
	10^3	10^5	10^7	10^3	10^5	10^7
50	0.15	3.59	-	0.09	7.83	-
100	0.26	5.15	-	0.09	2.61	-
500	0.42	2.80	-	0.08	1.58	-
1000	0.50	2.85	-	0.06	1.65	-
5000	1.13	3.21	-	0.11	1.81	-
10000	1.82	3.71	-	0.19	1.98	-

The second in line for testing was the FPTAS. The results of the tests done with the fully optimized C++ implementation operating on the easy and the hard instances are given in Tables III and IV, respectively. The tests clearly show that satisfying for an approximate solution makes the job much easier, when dealing with large data. The FPTAS has stable performance for all of the instance types, while the running time grows fast with the size of the input. The identified bottleneck of the algorithm is the memory consumption in the dynamic programming phase, which limits application of this concrete implementation to instances with $n < 1000$. More advanced and complicated schemes, e.g., the one described in [8], share the same general type of limitation, but improve on the upper limit value for n.

TABLE III
FPTAS, EASY INSTANCES, CPU TIMES [MILLISECONDS], $\varepsilon = 0.1$, n - INPUT SIZE, R - DATA RANGE, – STANDS FOR ALGORITHM TERMINATED (OUT OF MEMORY)

n \ R	*uncorrelated*			*weakly correlated*		
	10^3	10^5	10^7	10^3	10^5	10^7
50	2.12	2.33	2.44	2.27	2.20	2.30
100	16.99	18.84	18.42	17.97	17.86	17.75
500	2.2×10^3	2.3×10^3	2.1×10^3	2.2×10^3	2.1×10^3	2.0×10^3
1000	-	-	-	-	-	-

TABLE IV
FPTAS, HARD INSTANCES, CPU TIMES [MILLISECONDS], $\varepsilon = 0.1$, n - INPUT SIZE, R - DATA RANGE, – STANDS FOR ALGORITHM TERMINATED (OUT OF MEMORY)

n \ R	*strongly correlated*			*subset-sum*		
	10^3	10^5	10^7	10^3	10^5	10^7
50	2.76	2.77	2.79	2.37	2.43	2.49
100	20.32	20.62	20.66	18.87	18.50	18.74
500	2.5×10^3	2.5×10^3	2.3×10^3	2.2×10^3	2.2×10^3	2.1×10^3
1000	-	-	-	-	-	-

The second batch of experiments was conducted in order to acquire information on how much decisions concerning implementation issues affect actual performance of algorithms for the 0-1 knapsack problem. More precisely, the following effects on performance were measured and analyzed:

- application of templates from the C++ STL
- application of code tuning techniques
- application of compiler optimizations

The Standard Template Library, abbreviated STL, is the first library of generic algorithms and data structures for C++, created by Stepanov and Lee [15]. The STL was developed with four main ideas in mind [15]: generic programming, abstractness without loss of efficiency, the Von Neumann computation model, and value semantics. In the light of the presented research, it sounded interesting to see how well does the idea of enabling high level abstraction without loss of efficiency hold in the context of an algorithm implementation for the binary knapsack.

Code tuning is a single name for a set of techniques with a common goal of improving performance of an implemented algorithm. Code tuning techniques are mostly motivated by the Pareto principle, and they are described in detail in [13], [14]. The techniques applied in the experiments reported here include: minimization of array references and work inside loops, loop fusion, strength reduction, elimination of system routines and common sub-expressions, etc.

Modern compilers offer various levels of compile-time optimizations [2]. Some authors suggest that these optimizations are superior to any of the code tuning techniques [13] and that a programmer should just write clear code and leave optimizations to compilers. Therefore, it was of matter of interest to this research, to test these claims in practice, in the context of the 0-1 knapsack problem.

Table V presents the results of the tests conducted with C++ implementations of the FPTAS. The FPTAS was chosen as the benchmark because it features intense numerical computations and has a high demand on memory. The tests were done using weakly correlated 0-1 knapsack instances, where $R = 10^5$, $\varepsilon = 0.1$, which best correspond to problems most frequently met in practice. Compiler optimizations used were invoked with the flag *–O3* for *gcc* compiler, and they include options like the function in-lining, loop unswitching, tree vectorization, etc. Compiler optimization techniques are described in detail in [2]. The columns in Table V represent the following algorithm implementations:

A. code using the STL, without code tuning and compiler optimizations
B. code not using the STL, without code tuning and compiler optimizations
C. code not using the STL and compiler optimizations, with code tuning
D. code using the STL and compiler optimizations, without code tuning
E. code not using the STL, with code tuning and compiler optimizations

TABLE V
CODE IMPROVEMENT, CPU TIMES [MILLISECONDS], *n* - INPUT SIZE

n	*A*	*B*	*C*	*D*	*E*	*time saved A-E*
50	20.56	10.86	8.31	2.43	2.20	89%
100	146.86	80.02	67.23	18.62	17.86	88%
500	17924.01	9616.97	8110.82	2293.84	2088.03	88%

The most important conclusions that can be drawn from the presented data are as follows:

i) Application of the STL classes without the compiler optimizations leads to the significant loss of performance (compare columns *A* and *B* in Table V). Modern C++ compilers are tuned to minimize any abstraction penalty arising from heavy use of the STL. Therefore, the STL classes do offer higher abstraction without loss of efficiency, but only when coupled with compile-time optimizations.
ii) Code tuning techniques offer some speedup, but their effects are almost completely canceled when compiler optimizations are enabled. This happens because most of the tuning techniques are actually on the list of optimizations that modern compilers can offer. Therefore, code tuning proved to be a time-consuming method that provides only a small speedup in the case of the compile-time optimized C++ FPTAS implementation.
iii) Application of the compiler optimizations alone leads to the time savings of 87-88%, so it can be stated that they are the single most important improvement factor in the case of the C++ FPTAS.

We can conclude from the experimental data that an increase in performance of almost an order of magnitude can be achieved through optimization solely at the implementation level, at least for the tested algorithm.

V. CONCLUSION

The first set of the computational experiments with the programming implementations of the algorithms for the 0-1 knapsack problem identified some of the principal limitations and bottlenecks of the two analyzed algorithms, one exact and one approximate. This information can be helpful in estimating which problem instances can be efficiently handled in today's practical applications. Further research in algorithms that combine tight bounds with dynamic programming, like the combo algorithm, looks promising.

The second part of the research dealt with the implementation issues and their effects on the performance of the 0-1 knapsack FPTAS. The recorded results are in line with the statement from Knuth [9]: "Premature optimization is the root of all evil". Compile-time optimizations proved to be by far the most important factor in the improvement of the C++ FPTAS implementation. Code tuning techniques, applied in the code construction phase, offer some speedup, but often at the price of lowering code readability. Further, modern compilers are better at optimizing straight than intricate code, and intricacies are frequently created through tuning. So, another sound advice might also come in the following words [1]: "Programs must be written for people to read, and only incidentally for machines to execute".

REFERENCES

[1] H. Abelson, G. Sussman, *Structure and Interpretation of Computer Programs*, MIT Press, 2nd edition, 1996.
[2] A. Aho, M. Lam, R. Sethi, J. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison Wesley, 2nd edition, 2006.
[3] E. Balas, E. Zemel, "An algorithm for large zero-one knapsack problems", Operations Research, no. 28, pp. 1130-1154, 1980.
[4] R. E. Bellman, *Dynamic Programming*, Princeton University Press, Princeton, NJ, 1957.
[5] O. Ibarra, C. Kim, "Fast approximation algorithms for the knapsack and sum of subset problems", Journal of the ACM, no. 22, pp. 463-468, 1975.
[6] D. Gajić, *Algorithms for NP-complete Problems*, Master's thesis, Faculty of Electronic Engineering, Niš, 2009.
[7] *http://www.diku.dk/~pisinger/codes.html*
[8] H. Kellerer, U. Pferschy, D. Pisinger, *Knapsack Problems*, 1st edition, Springer, 2004.
[9] D. Knuth, „Structured Programming with *go to* Statements", Computing Surveys, vol. 6, no. 4, 1974.
[10] S. Martello, D. Pisinger, P. Toth, "Dynamic Programming and Strong Bounds for the 0-1 Knapsack Problem", Management Science, vol. 45, no. 3, pp. 414-424, 1999.
[11] S. Martello, D. Pisinger, P. Toth, "New trends in exact algorithms for the 0-1 knapsack problem", Technical Report 97/10, DIKU, University of Copenhagen, 1997.
[12] S. Martello, P. Toth, *Knapsack Problems: Algorithms and Computer Implementations*, John Wiley & Sons, 1990.
[13] S. McConnell, *Code Complete: A Practical Handbook of Software Construction*, Microsoft Press, 2nd edition, 2004.
[14] M. Scott, *Programming Language Pragmatics*, Morgan Kaufmann, 3rd edition, 2009.
[15] A. Stepanov, M. Lee, "The Standard Template Library", HP Laboratories Technical Report 95-11(R.1), 1995.
[16] V. Vazirani, *Approximation Algorithms*, Springer, 1st edition, 2001.