A Flipping Local Search Genetic Algorithm for the Multidimensional 0-1 Knapsack Problem

César L. Alonso[1] *, Fernando Caro[1], and José Luis Montaña[2] **

[1] Centro de Inteligencia Artificial, Universidad de Oviedo
Campus de Viesques, 33271 Gijón, Spain
calonso@aic.uniovi.es
[2] Departamento de Matemáticas, Estadística y Computación,
Universidad de Cantabria
montana@matesco.unican.es

Abstract. In this paper we present an evolutionary strategy for the multidimensional 0–1 knapsack problem. Our algorithm incorporates, a flipping local search process in order to locally improve the obtained individuals and also, a heuristic operator which computes problem-specific knowledge, based on the surrogate multipliers approach introduced in [12]. Experimental results show that our evolutionary algorithm is capable of obtaining high quality solutions for large size problems and that the local search procedure significatively improves the final obtained result.

Keywords: Evolutionary computation, genetic algorithms, knapsack problem, linear 0–1 integer programming.

1 Introduction

The multidimensional 0–1 knapsack problem (MKP) is a NP-complete combinatorial optimization problem which captures the essence of linear 0–1 integer programming problems. It can be formulated as follows. We are given a set of n objects where each object yields p_j units of profit and requires a_{ij} units of resource consumption in the i-th knapsack constraint. The goal is to find a subset of the objects such that the overall profit is maximized without exceeding the resource capacities of the knapsacks.

The MKP is one of the most popular constrained integer programming problems with a large domain of applications. Many practical problems can be formulated as a MKP instance: the capital budgeting problem in economy or the allocation of databases and processors in a distributed computer system ([7]) are some examples. Most of the research on knapsack problems deals with the simpler one-dimensional case ($m = 1$). For this single constraint case, the problem is not strongly NP-hard and some exact algorithms and also very efficient

* Partially supported by the Spanish MCyT and FEDER grant TIC2003-04153
** Partially supported by the Spanish MCyT, under project MTM2004-01167 and Programa de Movilidad PR2005-0422

approximation algorithms have been developed for obtaining near-optimal solutions.

In the multidimensional case several exact algorithms that compute different upper bounds for the optimal solutions are known. For example that in [8]. But this method, based on the computation of optimal surrogate multipliers, becomes no applicable for large values of m and n and other strategies must be introduced. In this context heuristic approaches for the MKP have appeared during the last decades following different ideas: greedy-like assignment ([6], [12]); LP-based search ([2]); surrogate duality information ([12]) are some examples. Also an important number of papers using genetic algorithms and other evolutionary strategies have emerged. The genetic approach has shown to be well suited for solving large MKP instances. In [11] a genetic algorithm is presented where infeasible individuals are allowed to participate in the search and a simple fitness function with a penalty term is used. Thiel and Voss (see [15]) presented a hybrid genetic algorithm with a tabu search heuristic. Chu and Beasley (see [5]) have developed a genetic algorithm that searches only into the feasible search space. They use a repair operator based on the surrogate multipliers of some suitable surrogate problem. Surrogate multipliers are calculated using linear programming. Also in [1] an evolutionary algorithm based on the surrogate multipliers is presented, but in this case a good set of surrogate multipliers is computed using a genetic algorithm.

In this present paper we propose an evolutionary strategy for MKP which takes as starting point the surrogate multipliers approach appeared first in [12] and later in [5] and also in [1]. Our algorithm introduces a flipping search strategy that includes random walking in hypercubes of the kind $\{0,1\}^n$. Our experimental results show that the local search procedure increases the quality of the solutions with respect to those obtained by other evolutionary algorithms based on the surrogate problem. The rest of the paper is organized as follows. Section 2 deals with the surrogate problem associated to an instance of the MKP. In section 3 we present our evolutionary algorithm for solving the MKP that incorporates our local search procedure. In section 4 experimental results and comparisons over problems taken from the OR-library(see [4][3]) and a set of large instances (proposed by Glover and Kochenberger[4])are included. Finally, section 5 contains some conclusive remarks.

2 The MKP and the Surrogate Problem

This section resumes the mathematical background and the formulation of a genetic heuristic for computing surrogate multipliers.

Definition 1. *An instance of the MKP is a 5-tuple:*

$$K = (n, m, \overline{p}, A, \overline{b}) \tag{1}$$

[3] Public available on line at http://people.brunel.ac.uk/∼mastjjb/jeb/info.html
[4] Public available at http://hces.bus.olemiss.edu/tools.html

where $n, m \in \mathbb{N}$ are both natural numbers representing (respectively) the number of objects and the number of constraints; $\overline{p} \in (\mathbb{R}^+)^n$ is a vector of positive real numbers representing the profits; $A \in M_{m\times n}(\mathbb{R}^+ \cup \{0\})$ is $m \times n$-matrix of non-negative real numbers corresponding to the resource consumptions and $\overline{b} \in (\mathbb{R}^+)^m$ is a vector of positive real numbers representing the knapsack capacities.

Remark 2. Given an instance of the MKP $K = (n, m, \overline{p}, A, \overline{b})$ the objective is

$$\text{maximize } f(\overline{x}) = \overline{p} \cdot \overline{x}' \tag{2}$$

$$\text{subject to } A \cdot \overline{x}' \leq \overline{b}' \tag{3}$$

where $\overline{x} = (x_1, \ldots, x_n)$ is a vector of variables that takes values in $\{0, 1\}^n$. Notation $\overline{v}'$ stands for the transposition of the vector $\overline{v}$.

Definition 3. *Let $K = (n, m, \overline{p}, A, \overline{b})$ be an instance of the MKP. A bit-vector $\alpha \in \{0, 1\}^n$ is a feasible solution of instance K if it verifies the constraint given by equation 3. A bit-vector $\alpha_K^{opt} \in \{0, 1\}^n$ is a solution of instance K if it is a feasible solution and for all feasible solution $\alpha \in \{0, 1\}^n$ of instance K it holds*

$$f(\alpha) \leq f(\alpha_K^{opt}) = \sum_{j=1}^{n} p_j \alpha_K^{opt}[j] \tag{4}$$

Here $\alpha[j]$ stands for the value of variable x_j.

Along this section we shall deal with a relaxed version of the MKP where the vector of variables $\overline{x}$ can take values in the whole interval $[0, 1]^n$. We will refer to this case as the LP-relaxed MKP. An instance of the LP-relaxed MKP will be denoted by K^{LP}. A solution of some instance K^{LP} of the LP-relaxed MKP will be denoted by $\alpha_{K^{LP}}^{opt}$.

Definition 4. *Let $K = (n, m, \overline{p}, A, \overline{b})$ be an instance of MKP and let $\overline{\omega} \in (\mathbb{R}^+)^m$ be a vector of positive real numbers. The surrogate constraint for K associated to $\overline{\omega}$, denoted by $Sc(K, \overline{\omega})$, is defined as follows.*

$$\sum_{j=1}^{n} (\sum_{i=1}^{m} \omega_i a_{ij}) x_j \leq \sum_{i=1}^{m} \omega_i b_i \tag{5}$$

The vector $\overline{\omega} = (\omega_1, \ldots, \omega_m)$ is called the vector of surrogate multipliers.
The surrogate instance for K, denoted by $SR(K, \overline{\omega})$, is defined as the 0–1 one-dimensional knapsack problem given by:

$$SR(K, \overline{\omega}) = (n, 1, \overline{p}, \overline{\omega} \cdot A, \overline{\omega} \cdot \overline{b}') \tag{6}$$

We shall denote by $\alpha_{SR(K,\overline{\omega})}^{opt}$ the solution of the surrogate instance $SR(K, \overline{\omega})$. As an easy consequence of the definition of the surrogate problem we can state that:

$$f(\alpha_{SR(K,\overline{\omega})}^{opt}) \geq f(\alpha_K^{opt}) \tag{7}$$

Remark 5. The best possible upper bound for $f(\alpha_K^{opt})$ using the inequality 7 is computed by finding the minimum value $min\{f(\alpha_{SR(K,\overline{\omega})}^{opt}) : \overline{\omega} \in (\mathbb{R}^+)^m\}$. Next proposition motivates the genetic strategy to approximate the optimal values of the surrogate multipliers $\overline{\omega}$. The interested reader can find the detailed proof in [1].

Proposition 6. *Let $K = (n, m, \overline{p}, A, \overline{b})$ be any instance of MKP. Then the following inequalities are satisfied.*

$$min\{f(\alpha_{SR(K,\overline{\omega})^{LP}}^{opt}) : \overline{\omega} \in (0,1]^m\} = min\{f(\alpha_{SR(K,\overline{\omega})^{LP}}^{opt}) : \overline{\omega} \in (\mathbb{R}^+)^m\} \quad (8)$$

$$min\{f(\alpha_{SR(K,\overline{\omega})^{LP}}^{opt}) : \overline{\omega} \in (0,1]^m\} \geq f(\alpha_K^{opt}), \quad (9)$$

Here $\alpha_{SR(K,\overline{\omega})^{LP}}^{opt}$ denotes the solution of the relaxed surrogate problem $SR(K,\overline{\omega})^{LP}$.

2.1 A Genetic Algorithm for Computing the Surrogate Multipliers

The following is a simple genetic algorithm (GA) to obtain approximate values for $\overline{\omega}$ (see [1]). In this GA the individuals will be binary 0–1 strings, representing $\overline{\omega} = (\omega_1, \ldots, \omega_m)$. Each ω_i will be represented as a q-bit binary substring, where q determines the desired precision of $\omega_i \in (0, 1]$.

Definition 7. *Given an MKP instance $K = (n, m, \overline{p}, A, \overline{b})$, $q \in \mathbb{N}$, and a representation $\gamma \in \{0,1\}^{qm}$ of a candidate vector of surrogate multipliers $\overline{\omega} = (\omega_1, \ldots, \omega_m)$ the fitness value of γ is defined as follows:*

$$fitness(\gamma) = f(\alpha_{SR(K,\overline{\omega})^{LP}}^{opt}) = \sum_{j=1}^{n} p_j \alpha_{SR(K,\overline{\omega})^{LP}}^{opt}[j] \quad (10)$$

Remark 8. The solution $\alpha_{SR(K,\overline{\omega})^{LP}}^{opt}$ can be obtained by means of the well known greedy algorithm for one-dimensional instances of the LP–relaxed knapsack problem.

Note that our objective is to minimize the $fitness$ function defined by equation 10. The remainder operators of the genetic algorithm have been chosen as follows: the roulette wheel rule as selection procedure, uniform crossover as recombination operator and bitwise mutation according to a given probability p_m (see [10] for a detailed description of these operators).

3 The Evolutionary Algorithm for the MKP

Given an instance of MKP, $K = (n, m, \overline{p}, A, \overline{b})$, and a set of surrogate multipliers, $\overline{\omega}$, computed by the genetic algorithm described in the previous section, we will run an steady state evolutionary algorithm for solving K that searches only into the feasible search space. So, it includes a repair operator for the infeasible individuals, that uses as heuristic operator the set of surrogate multipliers, and also an improvement technique. As main contribution, we incorporate to this algorithm a local search procedure in order to locally improve the individuals of the population.

3.1 Individual Representation and Fitness Function

We choose the standard 0–1 binary representation since it represents the underlying 0–1 integer values. A chromosome α representing a candidate solution for $K = (n, m, \overline{p}, A, \overline{b})$ is an n-bit binary string. A value $\alpha[j] = 0$ or 1 in the j-bit means that variable $x_j = 0$ or 1 in the represented solution. The fitness of chromosome $\alpha \in \{0,1\}^n$ is defined by

$$f(\alpha) = \sum_{j=1}^{n} p_j \alpha[i] \tag{11}$$

3.2 Repair Operator and Improve Procedure

The repair operator lies on the notion of utility ratios. Let $K = (n, m, \overline{p}, A, \overline{b})$ be an instance of MKP and let $\overline{\omega} \in (0,1]^m$ be a vector of surrogate multipliers. The utility ratio for variable x_j is defined by $u_j = \frac{p_j}{\sum_{i=1}^{m} \omega_i a_{ij}}$. Given an infeasible individual $\alpha \in \{0,1\}^n$ we apply the following repairing procedure .
Procedure DROP ([5], [1])
input: $K = (n, m, \overline{p}, A, \overline{b})$; $\overline{\omega} \in (0,1]^m$ and a chromosome $\alpha \in \{0,1\}^n$

```
begin
    for j=1 to n compute u_j
    P:=permutation of (1,...,n) with u_P[j] <= u_P[j+1]
    for j=1 to n do
        if (alpha[P[j]]=1 and infeasible(alpha)) then
            alpha[P[j]]:=0
end
```

Once we have transformed α into a feasible individual α', a second phase is applied in order to improve α'. This second phase is called the ADD phase.
Procedure ADD ([5], [1])
input: $K = (n, m, \overline{p}, A, \overline{b})$; $\overline{\omega} \in (0,1]^m$ and a chromosome $\alpha \in \{0,1\}^n$

```
begin
    P:=permutation of (1,...,n) with u_P[j] >= u_P[j+1]
    for j=1 to n do
        if alpha[P[j]]=0 then alpha[P[j]]:=1
        if infeasible(alpha) then alpha[P[j]]:=0
end
```

3.3 Genetic Operators

We use the roulette wheel rule as selection procedure, the uniform crossover as replacement operator and bitwise mutation with a given probability p_m. So when mutation must be applied to an individual α, a bit j from α is randomly selected and flipped from its value $\alpha[j] \in \{0,1\}$ to $1 - \alpha[j]$.

3.4 Local Search Procedure

Let $K = (n, m, \overline{p}, A, \overline{b})$ be an instance of MKP, $\overline{\omega} \in (0,1]^m$ a vector of surrogate multipliers and $\alpha \in \{0,1\}^n$ a chromosome representing a feasible solution of K. We describe below a local search procedure that can be applied to α. During the execution of this procedure, chromosomes representing infeasible solutions of K could be generated. So, the repair and improve operators described above should be applied. We will refer to $DROP_j$ as the operator that behaves as DROP but with the property that it does not modify the gen $\alpha[j]$ in chromosome α. We will refer to ADD_j in an analogous way.

Assuming that the iteration begins with a chromosome α, at each iteration step, the local search performs a random walk inside the hypercube $\{0,1\}^n$ generating a permutation P of length n. Then, for each $j \in \{1, \ldots, n\}$ makes a flip in $\alpha[P[j]]$ if and only if there is a gain in the fitness. Note that after that the flip was made, the generated chromosome could represent an infeasible solution. If it is the case we apply first $DROP_{P[j]}$, and then the procedure ADD. In other case –when the flip does not turns the chromosome infeasible– we just apply the procedure $ADD_{P[j]}$. The described process iterates until there is no gain in the fitness.

Procedure LocalSearch
input: $K = (n, m, \overline{p}, A, \overline{b})$; $\overline{\omega} \in (0,1]^m$ and a chromosome $\alpha \in \{0,1\}^n$

```
begin
    repeat
        beta:=alpha
        P:=permutation of (1,...,n)
        for j=1 to n do
            alpha_aux:=alpha
            alpha:=flip(P[j],alpha)
            if f(alpha) <= f(alpha_aux) then alpha:=alpha_aux
    until alpha=beta
end
```

Where the pseudo–code of the procedure $flip\,(i, \alpha)$ is as follows:

```
begin
    if alpha[i]=1 then
        alpha[i]:=0
        alpha:=ADD_i(K,omega,alpha)
    else
        alpha[i]=1
        alpha:=DROP_i(K,omega,alpha)
        alpha:=ADD(K,omega,alpha)
end
```

Our evolutionary algorithm follows the performance of the steady state GA described in [1] but introducing the above local search procedure. In this sense

we do not apply the local search procedure to all generated individuals but only periodically, each time that a number t of generations were produced. In this case we apply local search to all individuals in the population. The period t is given by the user and also the local search procedure is applied to the initial and final populations. To maintain diversity of individuals during the execution, the strategy used while applying the local search procedure to a population $P(kt)$, $k \in \mathbb{N}$, is the following:

```
begin
    P(kt+1):=empty_set
    for each alpha in P(kt) do
        beta:=local_search(alpha)
        if (beta belongs to (P(kt) or P(kt+1))) then
            insert alpha in P(kt+1)
        else
            insert beta in P(kt+1)
end
```

Remark 9. Individuals of the initial population are constructed generating random permutations from $(1, \ldots, n)$ applying the ADD procedure to the chromosome $\alpha = (0, \ldots, 0)$ and following the permutation order.

4 Experimental Results

We have executed our double genetic algorithm with local search (DGALS) on two sets of MKP instances. The first set of instances is included in the OR-Library proposed in [4][5]. They are randomly generated instances of MKP with number of constraints $m \in \{5, 10, 30\}$, number of variables $n \in \{100, 250, 500\}$ and tightness ratios $r \in \{0.25, 0.5, 0.75\}$. The tightness ratio r fixes the capacity of i-th knapsack to $r \sum_{j=1}^{n} a_{ij}$, $1 \leq i \leq m$. There are 10 instances for each combination of m, n and r giving a total of 270 test problems. The second set is a set of 11 instances proposed by Glover and Kochenberger[6]. These are large instances, up to $n = 2500$ and $m = 100$. After a previous experimentation we have set the parameters to the following values. The GA computing the surrogate multipliers uses population size 75, precision $q = 10$, probability of mutation 0.1 and 15000 generations to finish. The steady state GA solving the MKP instances uses population size 100, probability of mutation equal to 0.1 and the algorithm finishes when 1500000 evaluations have been performed. The period t to apply local search is set to 10^4 generations as a good trade-off between quality of the final solutions and computational effort.

For the first set of instances, since the optimal solutions values are unknown, the quality of a solution α is measured by the percentage gap of its fitness value with respect to the fitness values of the optimal solution of the LP-relaxed

[5] Public available on line at http://people.brunel.ac.uk/∼mastjjb/jeb/info.html
[6] Public available at http://hces.bus.olemiss.edu/tools.html

problem: $\%gap = 100\frac{f(\alpha^{opt}_{KLP})-f(\alpha)}{f(\alpha^{opt}_{KLP})}$. We will compare our algorithm with that presented in [5] (CHUGA), and the percentage gap is also the quality measurement used in that work. For the second set of 11 instances, we compare the best known solutions with that obtained by our evolutionary algorithm and also we compute the $\%gap$ with respect to the fitness values of that best known solutions.

Table 1. Computational results for CHUGA, DGA and DGALS. Values for DGA and DGALS are based on 10 runs for each instance.

Problem		GHUGA		DGA		DGALS	
m	n	A. %gap	A.E.B.S	A. %gap	A.E.B.S	A. %gap	A.E.B.S
5	100	0.59	24136	0.58	58045	0.58	99549
5	250	0.16	218304	0.15	140902	0.14	381840
5	500	0.05	491573	0.06	185606	0.05	675086
10	100	0.94	318764	0.98	70765	0.94	175630
10	250	0.35	475643	0.32	153475	0.29	495312
10	500	0.14	645250	0.15	179047	0.13	705148
30	100	1.74	197855	1.71	106542	1.70	217794
30	250	0.73	369894	0.71	184446	0.67	532870
30	500	0.40	587472	0.44	233452	0.36	716827

We measure the time complexity by means of the number of evaluations required to find the best obtained solution, as the individual representation and fitness function are the same in CHUGA, DGA and DGALS. We have executed our evolutionary algorithm on a Pentium IV; 3GHz. Over this platform the execution time for a single run ranges from 5 minutes, for the simplest problems, to 10 hours, for the most complex ones.

The results of our experiments are displayed in Tables 1 and 2 based on 10 independent executions for each instance. In table 1, our algorithm (DGALS), executed over the set of 270 instances is compared with that of Chu et. al (CHUGA) ([5]) and also with our first version without local search (DGA) ([1]). The first two columns identify the 30 instances that corresponds to each combination of m, n. In the remainder columns we show for the compared algorithms the average %gap and the average number of evaluations required until the best individual was encountered (A.E.B.S). We have taken the values corresponding to CHUGA from [5] and [13]. As the authors have pointed out in their work these results are based on only one run for each problem instance whereas in the case of our genetic algorithms 10 runs were executed. In table 2 we display the results concerning to the set of 11 large instances. In this case we present only results for DGALS compared with the best known solutions obtained by Glover and Kochenberger (GLOVERA) ([9]). We also display the average percentage gap of the fitness of our solutions with respect to the fitness of the best known solutions

Table 2. Computational results for the 11 large instances. We compare DGALS with GLOVERA. Values A.E.B.S. and A. %gap of DGALS are based on 10 runs for each instance.

Problem		GLOVERA	DGALS		
m	n	Best Sol.	Best Sol.	A.E.B.S.	A. %gap
15	100	3766	3766	98857	0
25	100	3958	3958	196649	0
25	150	5650	5656	502719	0.01
25	200	7557	7557	234675	0.01
25	500	19215	19211	875213	0.02
25	1500	58085	58078	1128350	0.01
50	150	5764	5764	273140	0
50	200	7672	7672	360145	0.03
50	500	18801	18796	950548	0.03
50	1500	57292	57295	1459603	0.05
100	2500	95231	95378	1487329	-0.14

From table 1 we conclude that our DGALS performs better than CHUGA and DGA in terms of the average %gaps and it seems that this better performance increases with the instance size. We can see also that the use of our local search procedure obviously suppose an increase in the amount of computational effort in order to reach the best solution. Nevertheless this is a reasonable increment and, as we have mentioned, the found solutions are significatively better. Although it is not showed in a table, we have done some executions only with local search applied to random generated individuals, obtaining poor results. This also agrees with the fact that using local search in all generations does not guarantees better individuals after 1500000 evaluations.

The results presented in table 2 agree with our conjecture that DGALS performs better with large instances. We think that the reason is that in some of these problems we have not reached the best known solution because we would need more than 1500000 evaluations. Specially surprising are the results for the largest instances, where we have obtained considerably better solutions in all executions of DGALS.

5 Conclusive remarks

In this paper we have presented an evolutionary algorithm for solving multidimensional knapsack problems based on genetic computation of surrogate multipliers incorporating a flipping local search procedure. On a large set of problems, we have shown that our evolutionary algorithm is capable of obtaining high-quality solutions for large problems of various characteristics. Clearly, the use of a local search procedure improves the quality of the individuals and directs the GA to the best solutions. Periodical application of the local search procedure reduces the amount of evaluations per generation and maintains diversity

in the population. Both aspects seem to be crucial for the performance of our evolutionary algorithm.

References

1. Alonso, C. L., Caro, F., Montaña, J. L.: An Evolutionary Strategy for the Multidimensional 0–1 Knapsack Problem based on Genetic Computation of Surrogate Multipliers. To appear in the Proceedings of IWINAC–2005. LNCS series (2005)
2. Balas, E., Martin, C. H.: Pivot and Complement–A Heuristic for 0–1 Programming. Management Science 26 (1) (1980) 86–96
3. Balas, E., Zemel, E.: An Algorithm for Large zero–one Knapsack Problems. Operations Research 28 (1980) 1130–1145
4. Beasley, J. E.: Obtaining Test Problems via Internet. Journal of Global Optimization 8 (1996) 429–433
5. Chu, P. C., Beasley, J. E.: A Genetic Algorithm for the Multidimensional Knapsack Problem. Journal of Heuristics 4 (1998) 63–86
6. Freville, A., Plateau, G.: Heuristics and Reduction Methods for Multiple Constraints 0–1 Linear Programming Problems. Europena Journal of Operationa Research 24 (1986) 206–215
7. Gavish, B., Pirkul, H.: Allocation of Databases and Processors in a Distributed Computing System. J. Akoka ed. Management od Distributed Data Processing, North-Holland (1982) 215–231
8. Gavish, B., Pirkul, H.: Efficient Algorithms for Solving Multiconstraint Zero–One Knapsack Problems to Optimality. Mathematical Programming 31 (1985) 78–105
9. Glover, F., Kochenberger, G. A.: Critical event tabu search for multidimensional knapsack problems. Metaheuristics: The Theory and Applications. Kluwer Academic Publishers (1996) 407–427
10. Goldberg, D. E.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley (1989)
11. Khuri, S., Bäck, T., Heitkötter, J.: The Zero/One Multiple Knapsack Problem and Genetic Algorithms. Proceedings of the 1994 ACM Symposium on Applied Computing (SAC'94), ACM Press (1994) 188–193
12. Pirkul, H.: A Heuristic Solution Procedure for the Multiconstraint Zero–One Knapsack Problem. Naval Research Logistics 34 (1987) 161–172
13. Raidl, G. R.: An Improved Genetic Algorithm for the Multiconstraint Knapsack Problem. Proceedings of the 5th IEEE International Conference on Evolutionary Computation (1998) 207–211
14. Rinnooy Kan, A. H. G., Stougie, L., Vercellis, C.: A Class of Generalized Greedy Algorithms for the Multi-knapsack Problem. Discrete Applied Mathematics 42 (1993) 279–290
15. Thiel, J., Voss, S.: Some Experiences on Solving Multiconstraint Zero–One Knapsack Problems with Genetic Algorithms. INFOR 32 (1994) 226–242