Genetic Algorithms for the Multiple Container Packing Problem

Günther R. Raidl, Gabriele Kodydek

Department of Computer Graphics
Vienna University of Technology
Karlsplatz 13/1861, 1040 Vienna, Austria
e-mail: {raidl,kodydek}@eiunix.tuwien.ac.at

Abstract. This paper presents two variants of Genetic Algorithms (GAs) for solving the Multiple Container Packing Problem (MCPP), which is a combinatorial optimization problem comprising similarities to the Knapsack Problem and the Bin Packing Problem. Two different representation schemes are suggested, namely direct encoding and order based encoding. While order based encoded solutions are always feasible, a repair algorithm is used in case of direct encoding to ensure feasibility. Additionally, local improvement operators have been applied to both GA variants. The proposed algorithms were empirically compared by using various sets of differently sized test data. Order based encoding performed better for problems with fewer items, whereas direct encoding exhibited advantages when dealing with larger problems. The local improvement operators lead in many cases not only to better final results but also to shorter running times because of higher convergence rates.

1 Introduction

The *Multiple Container Packing Problem* (MCPP) is a combinatorial optimization problem which involves finding the most remunerative assignment of n items with given weights and values to C containers such that each item is assigned to one container or remains unassigned, and the total weight of each container does not exceed a given maximum. This problem has applications in various fields as e.g. in air baggage handling and many other important sectors of a modern economy. In detail, it can be formulated as follows:

$$\text{maximize} \quad f = \sum_{i=1}^{C} \sum_{j=1}^{n} v_j x_{i,j}, \tag{1}$$

$$\text{subject to} \quad \sum_{i=1}^{C} x_{i,j} \leq 1, \quad j = 1, \ldots, n, \tag{2}$$

$$\sum_{j=1}^{n} w_j x_{i,j} \leq W_{\max}, \quad i = 1, \ldots, C, \tag{3}$$

$$x_{i,j} \in \{0, 1\}, \quad i = 1, \ldots, C, \; j = 1, \ldots, n,$$

$$\text{with} \quad w_j > 0, \; v_j > 0, \; W_{\max} > 0\,.$$

Let w_j be the weight and v_j be the value of item j. The variables searched for are $x_{i,j}$ ($i = 1, \ldots C$, $j = 1, \ldots, n$): If item j is assigned to container i, $x_{i,j}$ is set to 1, otherwise to 0. The goal is to maximize the total value of all assigned items (1). The n constraints in (2) ensure that each item is assigned to one container at maximum. According to (3), each of the C containers has a total maximum weight $W_{\max}$ which must not be exceeded by the sum of the weights of all items assigned to this container.

The next section gives a short survey of combinatorial problems related to the MCPP and of *Genetic Algorithms* (GAs) for solving them. In Sect. 3, two GA variants differing in their solution encoding techniques are presented, and specific local improvement operators are introduced. An empirical comparison of the new GAs using various test problem sets follows in Sect. 4. Finally, some conclusions are drawn in Sect. 5.

2 Related Problems

There are some other combinatorial optimization problems which are closely related to the MCPP. The well-known *Knapsack Problem* (KP) can be seen as the variant of the MCPP with only one container ($C = 1$): Which items should be selected for packing into a single knapsack to get the highest possible total value while not exceeding a given total weight? The KP is not strongly NP-hard [8], and efficient approximation algorithms have been developed for obtaining near optimal solutions, see e.g. [13]. The more general and strongly NP-hard *Multiconstrained Knapsack Problem* (MKP) involves more than one ($m > 1$) limited resources leading to m constraints. E.g. additionally to the weight, the volume might be a second constrained resource. Various exact algorithms and heuristics for the MKP can be found in [13]. A comprehensive review is given in [3, 5].

Another combinatorial optimization problem related to the MCPP is the *Bin Packing Problem* (BPP): In this problem, the goal is to minimize the number of containers necessary to pack all n items while not violating any weight constraint. The values of items do not play a role. Like the MKP, the BPP in its general form is NP-hard, see [6]. Note that the MCPP can also be seen as a complex combination of the KP and the BPP, since the MCPP can be divided into two strongly depending parts which must be solved simultaneously: (a) Select items for packing, and (b) distribute chosen items over the available containers.

One more related problem, which can be seen as a more general form of the BPP, is known under the term *General Assignment Problem* (GAP), see [3, 4]: A set of jobs ($\hat{=}$ items) must be assigned to a set of agents ($\hat{=}$ containers). Each possible assignment has its individual capacity requirements and costs, and each agent has its individual capacity limits. The goal is to distribute all jobs in a way to pay minimal costs while satisfying all constraints.

In the last years, Genetic Algorithms [1, 2, 7, 10, 14] have proven to be very well suited for finding nearly optimal solutions to difficult instances of the (M)KP, BPP, GAP, and similar combinatorial problems. Olsen presented in

[15] a GA for the KP using a bit string representation for solutions. Infeasible solutions containing constraint violations are penalized by adding a suitable term to the objective function. Difficulties lie in the selection of the penalty function and its coefficients to prevent premature convergence and infeasible final solutions. Olsen compared various penalty functions for such a GA. In a similar GA of Khuri et al. [12], a graded penalty term was used; only moderate results were reported on a small number of standard test problems. Rudolph and Sprave [17] presented a GA in which parent selection is restricted to happen between "neighboring" solutions. Infeasible solutions were penalized as in [12].

Another approach to handle infeasible solutions is to incorporate a repair algorithm which transforms each infeasible solution into a feasible one (see [10, 14] for a general introduction). In the (M)KP, this can be done by setting some genes to 0. Chu describes such a GA in his PhD thesis [3] and together with Beasley in [5]. This GA uses a heuristic algorithm based on the shadow prices of the LP-relaxed solution[1] for selecting the genes which are set to 0 in case of unsatisfied constraints. Additionally, the GA includes a local optimization operator for improving each newly generated solution by setting previously unset genes to 1 as long as no constraints are violated. Empirical comparisons of the GA to other approaches using various standard test problems are documented in [3, 5]. The results show that Chu's GA performs superior to the other methods concerning the quality of the final solutions. In [16], we presented another GA for the MKP also including a heuristic repair algorithm and local improvement. The starting population is generated by using a greedy heuristic, which speeds up the convergence to high quality solutions essentially. For the same test data, this improved GA finds much faster slightly better solutions than that of Chu.

A different technique for representing solutions of the (M)KP is to use an order based encoding scheme, which is well known from GAs for the *Traveling Salesman Problem* (TSP), see [14]. A first fit algorithm is used as a decoder to get the selected items. This approach guarantees that only feasible solutions are generated. Hinterding presented in [11] a GA with such a representation for the KP. He used *uniform order based crossover* (see [2, 10, 14]) as recombination operator and realized that disallowing duplicates in the population significantly improves results. Sun and Wang [18] proposed a very similar order based GA for the more general *0–1 integer programming problem* in which the weights w_j may also be negative. Owing to this property, a more complex decoding function is necessary to guarantee feasibility.

For the BPP, an efficient hybrid GA is presented in [6]. A GA for the GAP is discussed in [3, 4]. This approach uses a repair algorithm for improving infeasible solutions, but reaching feasibility cannot be guaranteed for all cases. Furthermore, a greedy heuristic is used to locally improve solutions by reassigning jobs to different agents involving lower costs. Regarding the quality of final solutions, the GA outperformed several other optimization techniques for most test problems.

[1] The solution of the *Linear Programming* relaxation of the original problem, where discrete parameters are substituted by continuous ones.

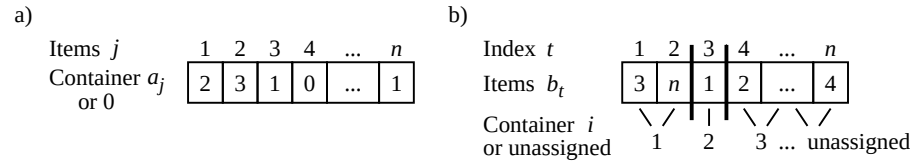

Fig. 1. (a) Direct encoding versus (b) order based encoding

3 GAs for the MCPP

Inspired by [3, 4, 5, 16], a steady-state GA with tournament selection and a replacement scheme which eliminates the worst solution or the last one generated in case of duplicates is used as a basis instead of the traditional generational GA.

According to the experiences from the previous GAs for the (M)KP, BPP, and GAP, two different solution encoding schemes seemed suitable for the MCPP: *Direct encoding* (DE), similar to [4, 5, 12, 15, 16, 17], and *order based encoding* (OBE), compare [11, 18]. These two techniques are described in detail in the following.

3.1 Direct Encoding

A solution is encoded as a vector $\mathbf{a}$ consisting of n genes a_j ($j = 1, \ldots, n$). Each a_j represents the number i ($i = 1, \ldots, C$) of the container to which item j is supposed to be assigned or the special value 0 if no assignment to any container should be done, see Fig. 1a.

With this representation it is easily possible that solutions are generated which violate constraints (2) concerning the maximum total weight $W_{\max}$ of containers. Results of the mentioned GAs for related problems suggest the usage of a repair mechanism within the chromosome decoding function rather than penalizing such infeasible solutions.

The algorithm for decoding and possibly repairing a chromosome $\mathbf{a}$ is shown in Fig. 2a. First, the current weights of all containers (vector $\mathbf{s}$) are initialized with 0. Then, all items are processed in a random, always different order so that not the same items are favored every time. Each item j is checked if it fits into the container possibly specified in a_j, in which case the item is actually assigned and the current weight of the container s_{a_i} is increased accordingly. If adding item j would result in exceeding the total maximum weight $W_{\max}$, the value of the corresponding gene a_j is set to 0 meaning that the item is not assigned to any container.

Note that this encoding scheme allows the usage of the standard recombination and mutation operators as uniform crossover and flip mutation, which behaved best in performed experiments.

a)

```
procedure DecodeDE (a);
s ← 0;
for all items j in random order do
  if a_j ≠ 0 then
    if s_{a_j} + w_j ≤ W_max then
      s_{a_j} ← s_{a_j} + w_j;
      assign item j to container a_j;
    else
      a_j ← 0;
done;
```

b)

```
procedure ImproveDE (a);
for all unassigned items j
            in random order do
  for all containers i
              in random order do
    if s_i + w_j ≤ W_max then
      s_i ← s_i + w_j;
      a_j ← i;
      assign item j to container a_j;
      skip remaining containers;
done;
```

Fig. 2. Direct encoding: (a) chromosome decoding including repair mechanism and (b) heuristic improvement

a)

```
procedure DecodeOBE (b);
i ← 1; s ← 0; t ← 1;
while i ≤ C do
  if s_i + w_{b_t} ≤ W_max then
    s_i ← s_i + w_{b_t};
    assign item b_t to container i;
  else
    e_i ← t;
    i ← i + 1;
    if i ≤ C then
      s_i ← w_{b_t};
      assign item b_t to container i;
  t ← t + 1;
done;
```

b)

```
procedure ImproveOBE (b);
for t ← e_C to n do
  for all containers i
            in random order do
    if s_i + w_{b_t} ≤ W_max then
      s_i ← s_i + w_{b_t};
      j ← b_t;
      for k ← t − 1 downto e_i do
        b_{k+1} ← b_k;
      b_{e_i} ← j;
      for r ← i to C do
        e_r ← e_r + 1;
      assign item j to container i;
      skip remaining containers;
done;
```

Fig. 3. Order based encoding: (a) chromosome decoding and (b) local improvement

3.2 Order Based Encoding

A solution is represented by a permutation of all items $j = 1, \ldots, n$ stored in a chromosome $\mathbf{b} = (b_1, \ldots, b_n)$, see Fig. 1b. The first fit algorithm shown in Fig. 3a is used as a decoder to get the item/container assignments: All available containers i ($i = 1, \ldots, C$) are consecutively filled with the items in the order given by permutation $\mathbf{b}$. If an item b_t does not fit into the current container i, the algorithm will proceed with the next container, and the index t of this first item not fitting into container i is stored in e_i for later usage during local improvement. If all containers are packed in this way, the remaining items are treated as unassigned. This method ensures that only feasible solutions are created, and no repair mechanism is necessary.

When using order based encoding, special recombination and mutation operators are needed to prevent solutions with duplicate or missing items. Various recombination methods known from the TSP such as order crossover, partially matched crossover, uniform order based crossover, and cycle crossover satisfy those requirements, see [14]. Preliminary experiments indicated a slightly better performance for the MCPP when order crossover was used. As mutation operator the exchange of two randomly chosen items performed better than insertion or inversion.

3.3 Local Improvement Algorithms

As already mentioned, many GAs benefit from the inclusion of local or heuristic improvement techniques applied to some or all newly generated solutions, see e.g. [3, 4, 5, 16]. Such a hybridization should therefore also be considered for the MCPP. The basic idea is to improve each newly generated solution by trying to assign its unassigned items to a container that has not reached its maximum total weight yet.

Figure 2b shows the algorithm when using direct encoding in detail: All previously unassigned items j ($a_j = 0$) are processed in random order, and each container is checked in random order if enough space is available to hold item j. In this case, the item is assigned to the found container and the algorithm proceeds with the next unassigned item.

When using order based encoding, the local improvement gets a little more complicated, see Fig. 3b: Again, the algorithm processes all unassigned items, now starting with the first one at position e_C in $\mathbf{b}$. Each container is checked in random order if there is enough space left for putting item b_t into it. In this case, the item is moved from its current position to position e_i and is now considered the last element of container i. For this purpose all items between these two positions need to be moved one gene up, and the container border indexes e_i to e_C must be incremented.

4 Implementation and Experimental Results

Miscellaneous randomly generated test data sets were used to practically examine a GA using direct encoding with and without local improvement (DEI, DE) and order based encoding with and without local improvement (OBEI, OBE). These four GA variants have been implemented on a Pentium-II PC (266 MHz) using Linux, the GNU C++ compiler and the publicly available Genetic Algorithms Library GAlib by M. Wall [19].

Various characteristics and parameters of the steady-state GA, which were determined by preliminary experiments and found to be robust and well suited for the MCPP in general, are summarized in Table 1. It is essential that duplicate solutions are disallowed in the population (by using the replacement scheme already described in Sect. 3) because otherwise the GA converges too fast to only poor solutions. For the KP a similiar behavior has already been observed by

Table 1. Characteristics of the GA with the two solution encoding variants

GA:	steady state, no duplicate solutions
Goal:	maximize total value of assigned items (f)
Selection:	tournament ($k = 2$)
Recombination:	DE: uniform crossover ($p_c = 0.5$), OBE: order crossover ($p_c = 0.5$)
Mutation:	DE: flip mutation ($p_m = 1/n$), OBE: swap mutation ($p_m = 1/n$)
Population size:	100
Termination:	200,000 solutions evaluated without finding a new best solution

Table 2. Gaps of best-of-run solutions with needed numbers of evaluations *evals* and CPU-times t for problems with different numbers of containers C and fixed total container weights $W_{\max} = 100$ (average values from 10 runs/problem)

$W_{\max} =$	$n = 30$				$n = 50$				$n = 200$			
100	C	%-*gap*	*evals*	t[sec]	C	%-*gap*	*evals*	t[sec]	C	%-*gap*	*evals*	t[sec]
DE	3	2.74	11140	0.9	5	2.60	270720	37.7	20	1.91	377300	162.9
	6	2.69	139260	12.0	10	1.43	379180	53.2	40	1.94	675140	295.2
	9	3.31	98860	9.2	15	2.53	162220	23.7	60	1.99	730200	322.7
	12	2.48	92640	8.2	20	2.58	244820	36.3	80	2.89	644500	295.6
DEI	3	2.74	5620	0.5	5	2.58	119620	17.4	20	1.65	711080	336.0
	6	2.45	38900	3.5	10	1.10	408280	62.2	40	1.64	755220	356.5
	9	3.01	19260	1.8	15	1.99	262800	42.5	60	1.48	935840	452.4
	12	1.58	81680	8.0	20	1.50	305340	48.7	80	2.09	823100	401.5
OBE	3	3.16	61200	4.9	5	2.96	194300	25.6	20	2.84	706000	383.9
	6	2.82	183560	13.1	10	1.73	352680	44.9	40	2.52	1371840	723.9
	9	3.25	140780	9.8	15	2.35	397320	48.8	60	2.10	1431300	698.8
	12	1.42	167120	12.4	20	2.03	358880	43.7	80	2.36	1431560	621.4
OBEI	3	2.74	78400	9.0	5	2.28	193980	37.1	20	2.57	717940	713.9
	6	2.32	72000	7.4	10	1.58	299900	52.4	40	2.49	778920	860.9
	9	2.90	32520	3.2	15	2.00	231300	39.0	60	2.28	1262140	1274.7
	12	1.05	184500	15.9	20	1.67	231660	33.4	80	2.62	1009600	833.5

Hinterding [11]. Note that each GA run was terminated when no improvements were encountered within the last 200,000 evaluations. This condition ensures that the GA has enough time to converge. In general, we were primarily interested in finding high-quality solutions and only secondary in the needed CPU-time.

Test problems were generated in three different sizes, namely with n=30, 50, and 200 items. Item weights w_j were randomly chosen out of the interval $[5, 95]$ giving an average item weight of $\overline{w} = 50$. The item values v_j were generated by multiplying the weight w_j of each item by a relative item value randomly taken from $[0.8, 1.2]$. In a first set of test problems, the total container weight $W_{\max}$ was set to 100 allowing two items of average weight to be packed in a single

Table 3. Gaps of best-of-run solutions with needed numbers of evaluations *evals* and CPU-times t for problems with different total container weights $W_{\max}$ and fixed numbers of containers C (average values from 10 runs/problem)

		$n = 30, C = 3$			$n = 50, C = 5$			$n = 200, C = 20$		
	$W_{\max}$	%-*gap*	*evals*	t[sec]	%-*gap*	*evals*	t[sec]	%-*gap*	*evals*	t[sec]
DE	100	2.74	11140	0.9	2.60	270720	37.7	1.91	377300	162.9
	200	0.68	80180	6.8	0.42	573420	80.7	0.71	1045960	455.3
	300	0.50	64400	5.5	0.51	454560	64.4	0.45	933200	407.1
	400	0.39	208120	17.7	0.40	240400	34.2	0.32	1105640	484.7
DEI	100	2.74	5620	0.5	2.58	119620	17.4	1.65	711080	336.0
	200	0.66	33360	3.0	0.35	331180	49.2	0.51	760720	371.8
	300	0.47	17040	1.5	0.30	230960	35.2	0.27	1021280	474.6
	400	0.37	160660	14.7	0.39	315820	47.8	0.25	1028240	476.7
OBE	100	3.16	61200	4.9	2.96	194300	25.6	2.84	706000	383.9
	200	0.96	129980	9.8	0.88	368880	47.5	1.26	742280	408.9
	300	0.53	84100	6.0	0.48	434700	54.0	0.76	773060	393.1
	400	0.45	99400	7.2	0.53	286480	35.5	0.41	744180	314.6
OBEI	100	2.74	78400	9.0	2.28	193980	37.1	2.57	717940	713.9
	200	0.68	113240	11.7	0.60	299620	48.3	2.49	558300	462.7
	300	0.44	131040	12.0	0.70	243020	37.8	1.64	514560	341.7
	400	0.33	199720	16.7	0.53	505960	70.2	0.64	565960	291.3

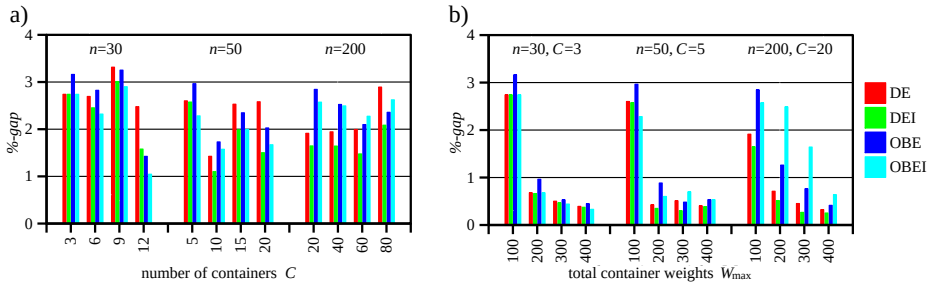

Fig. 4. Average gaps for problems with (a) varying number of containers C and (b) varying total container weights $W_{\max}$

container, and the number of containers C was varied in a way that roughly 20, 40, 60, and 80 percent of the n items could be packed in total: $n = 30$: $C \in \{3, 6, 9, 12\}$, $n = 50$: $C \in \{5, 10, 15, 20\}$, $n = 200$: $C \in \{20, 40, 60, 80\}$. This test series includes therefore 12 problems, and the four GA variants were run 10 times for each problem.

Since the optimal solution values for most of these problems are not known, the quality of a final solution is measured by the percentage gap of the GA's solution value f with respect to the optimal value of the LP-relaxed problem

$f_{\max}^{\mathrm{LP}}$. This upper bound can easily be determined for any MCPP by sorting all items according to their relative values v_j/w_j and summing up the item values v_j starting with the best item until a total weight of $CW_{\max}$ is reached. The last item is counted proportionately. Knowing the LP optimum, the gap is determined by $\%\text{-}gap = 100(f - f_{\max}^{\mathrm{LP}})/f_{\max}^{\mathrm{LP}}$. Table 2 shows average results derived from 10 runs per problem instance. Beside the gaps of the best-of-run solutions, the numbers of evaluated solutions *evals* and CPU-times t in seconds until these best-of-run solutions were found are presented. The average gaps are also depicted in Fig. 4a.

In a second test series the number of containers was fixed ($n = 30$: $C = 3$, $n = 50$: $C = 5$, $n = 200$: $C = 20$), and the total container weight $W_{\max}$ was varied from 100 to 400 in steps of 100. Table 3 and Fig. 4b show average results for these 12 problem instances.

In general, direct encoding did always benefit from local improvement, order based encoding in case of small or medium sized problems. Clearly, local improvement increases the CPU-time needed for a single evaluation. Nevertheless entire runs using DEI and OBEI were most of the time only slightly slower and sometimes even faster than runs with DE and OBE because the locally improved GAs usually needed fewer evaluations to converge.

For small-sized problems ($n = 30$), OBEI lead nearly always to the smallest gaps (best results), while OBE gave the worst final solutions. For large-scale problems ($n = 200$), the results are surprisingly different: Generally, the direct encoding GA variants were better than the order based ones. Especially DEI outperformed all other approaches by far. The results of OBEI are in this case often even worse than those of OBE. Furthermore, the execution times were significantly higher for OBEI and OBE (except for the problems with larger $W_{\max}$).

A reason for the different behaviors of the two encoding schemes seems to be that the order based approach with its order crossover is more disruptive and introduces therefore more diversity into the population. Smaller problems with fewer items benefit from this property because the GA can easier escape from local optima. For larger problems this property turns into a disadvantage because the GA converges slower.

5 Conclusions

Two GA variants using different encoding schemes and recombination and mutation operators were introduced for the MCPP. While infeasible solutions never appear in OBE(I), a repair algorithm has been incorporated into the decoding function of DE(I). Both encoding approaches performed generally well, but the order based method, which seems to be more disruptive, exhibited advantages for problems with fewer items. Direct encoding performed better for larger problems. The introduction of the local improvement operators lead in many cases not only to better results, but also to shorter total running times because of higher convergence rates.

References

1. Bäck T.: Evolutionary Algorithms in Theory and Practice, Oxford University Press, New York (1996)
2. Bäck T., Fogel D. B., Michalewicz Z.: Handbook of Evolutionary Computation, Oxford University Press (1997)
3. Chu P. C.: A Genetic Algorithm Approach for Combinatorial Optimization Problems, Ph.D. thesis at The Management School, Imperial College of Science, London (1997)
4. Chu P. C., Beasley J. E.: A Genetic Algorithm for the Generalized Assignment Problem, Computers & Operations Research **24**(1) (1997) 17–23
5. Chu P. C., Beasley J. E.: A Genetic Algorithm for the Multidimensional Knapsack Problem, working paper at The Management School, Imperial College of Science, London (1997)
6. Falkenauer E.: A Hybrid Grouping Genetic Algorithm for Bin Packing, working paper at CRIF Industrial Management and Automation, CP 106-P4, 50 av. F. D. Roosevelt, Brussels, Belgium (1994)
7. Fogel D. B.: Evolutionary Computation – Toward a New Philosophy of Machine Intelligence, IEEE Press, Piscataway, NJ (1995)
8. Garey M. D., Johnson D. S.: Computers and Intractability: A Guide to the Theory of NP-Completeness, Freeman, San Francisco (1979)
9. Gavish B., Pirkul H.: Efficient Algorithms for Solving Multiconstraint Zero-One Knapsack Problems to Optimality, Mathematical Programming **31** (1985) 78–105
10. Goldberg D. E.: Genetic Algorithms in Search, Optimization and Machine Learning, Addison–Wesley (1989)
11. Hinterding R.: Mapping, Order-independent Genes and the Knapsack Problem, in Proc. of the 1st IEEE Int. Conference on Evolutionary Computation 1994, Orlando, FL (1994) 13–17
12. Khuri S., Bäck T., Heitkötter J.: The Zero/One Multiple Knapsack Problem and Genetic Algorithms, in Proc. of the 1994 ACM Symposium on Applied Computing, ACM Press (1994) 188–193
13. Martello S., Toth P.: Knapsack Problems: Algorithms and Computer Implementations, J. Wiley & Sons (1990)
14. Michalewicz Z.: Genetic Algorithms + Data Structures = Evolution Programs, Springer, Berlin (1992)
15. Olsen A. L.: Penalty Functions and the Knapsack Problem, in Proc. of the 1st International Conference on Evolutionary Computation 1994, Orlando, FL (1994) 559–564
16. Raidl G. R.: An Improved Genetic Algorithm for the Multiconstrained 0–1 Knapsack Problem, in Proc. of the 1998 IEEE International Conference on Evolutionary Computation, Anchorage, Alaska (1998) (to appear)
17. Rudolph G., Sprave J.: Significance of Locality and Selection Pressure in the Grand Deluge Evolutionary Algorithm, in Proc. of the International Conference on Parallel Problem Solving from Nature IV (1996) 686–694
18. Sun Y., Wang Z.: The Genetic Algorithm for 0–1 Programming with Linear Constraints, in Proc. of the 1st ICEC'94, Orlando, FL (1994) 559–564
19. Wall M.: GAlib – A C++ Genetic Algorithms Library, Version 2.4, Massachusetts Institute of Technology, http://lancet.mit.edu/ga (1996)

This article was processed using the LaTeX macro package with LLNCS style