The knowledge about the proper selection of GA parameters is still only fragmentary and has rather empirical background. Among these parameters, the population size seems to be the most important, since it has strong influence on the GA simulation cost. It might be that the best way for its setting is to let it to self-tune accordingly to the GA actual needs. This is the idea behind the GAVaPS method: at different stages of the search process different sizes of the population might be optimal. However, the reported experiments are preliminary and the allocation of the lifetime parameter deserves further research.

4.5 Genetic algorithms, constraints, and the knapsack problem

As discussed in the Introduction, the constraint-handling techniques for genetic algorithms can be grouped into a few categories. One way of dealing with candidates that violate the constraints is to generate potential solutions without considering the constraints and then to penalize them by decreasing the "goodness" of the evaluation function. In other words, a constrained problem is transformed to an unconstrained one by associating a penalty with all constraint violations; these penalties are included in the function evaluation. Of course, there are a variety of possible penalty functions which can be applied. Some penalty functions assign a constant as a penalty measure. Other penalty functions depend on the degree of violation: the larger violation is, the greater penalty is imposed (however, the growth of the function can be logarithmic, linear, quadratic, exponential, etc. with respect to the size of the violation).

Additional version of penalty approach is elimination of non-feasible solutions from the population (i.e., application of the most severe penalty: death penalty). This technique was used successfully in evolution strategies (Chapter 8) for numerical optimization problems. However, such approach has its drawbacks. For some problems the probability of generating (by means of standard genetic operators) a feasible solution is relatively small and the algorithm spends a significant amount of time evaluating illegal individuals. Moreover, in this approach non-feasible solutions do not contribute to the gene-pool of any population.

Another category of constraint handling methods is based on application of special repair algorithms to "correct" any infeasible solutions so generated. Again, such repair algorithms might be computationally intensive to run and the resulting algorithm must be tailored to the particular application. Moreover, for some problems the process of correcting a solution may be as difficult as solving the original problem.

The third approach concentrates on the use of special representation mappings (decoders) which guarantee (or at least increase the probability of) the generation of a feasible solution or the use of problem-specific operators which preserve feasibility of the solutions. However, decoders are frequently computa-

tionally intensive to run [73], not all constraints can be easily implemented this way, and the resulting algorithm must be tailored to the particular application.

In this section we examine the above techniques on one particular problem: the 0/1 knapsack problem. The problem is easy to formulate, yet, the decision version of it belongs to a family of NP-complete problems. It is an interesting exercise to evaluate the advantages and disadvantages of constraint handling techniques on this particular problem with a single constraint: the conclusions might be applicable to many constrained combinatorial optimization problems. It should be noted, however, that the main purpose of this section is to illustrate the concept of decoders, repair algorithms, and penalty functions (discussed briefly in the Introduction) on one particular example; by no means it is a complete survey of possible methods. For that reason we do not provide the optimum solutions for the test cases: we only make some comparisons between presented methods.

4.5.1 The 0/1 knapsack problem and the test data

There is a variety of knapsack-type problems in which a set of entities, together with their values and sizes, is given, and it is desired to select one or more disjoint subsets so that the total of the sizes in each subset does not exceed given bounds and the total of the selected values is maximized [252]. Many of the problems in this class are NP-hard and large instances of such problems can be approach only by using heuristic algorithms. The problem selected for these experiments is the 0/1 knapsack problem. The task is, for a given set of weights $W[i]$, profits $P[i]$, and capacity C, to find a binary vector $x = \langle x[1], \ldots, x[n] \rangle$, such that

$$\sum_{i=1}^{n} x[i] \cdot W[i] \leq C,$$

and for which

$$\mathcal{P}(x) = \sum_{i=1}^{n} x[i] \cdot P[i]$$

is maximum.

As indicated earlier, in this section we analyse the experimental behavior of a few GA-based algorithms on several sets of randomly generated test problems. Since the difficulty of such problems is greatly affected by the correlation between profits and weights [252], three randomly generated sets of data are considered:

- *uncorrelated*:

  $W[i] :=$ (uniformly) random($[1..v]$), and
  $P[i] :=$ (uniformly) random($[1..v]$).

- *weakly correlated*:

  $W[i] :=$ (uniformly) random($[1..v]$), and
  $P[i] := W[i] +$ (uniformly) random($[-r..r]$),

(if, for some i, $P[i] \leq 0$, such profit value is ignored and the calculations are repeated until $P[i] > 0$).

- *strongly correlated*:

  $W[i] :=$ (uniformly) random($[1..v]$), and
  $P[i] := W[i] + r$.

Higher correlation implies smaller value of the difference:

$$\max_{i=1..n}\{P[i]/W[i]\} - \min_{i=1..n}\{P[i]/W[i]\};$$

as reported in [252], higher correlation problems have higher expected difficulty.

Data have been generated with the following parameter settings: $v = 10$ and $r = 5$. For the tests we used three data sets of each type containing $n = 100$, 250, and 500 items, respectively. Again, following a suggestion from [252], we have taken under consideration two knapsack types:

- *restrictive knapsack capacity*

  A knapsack with the capacity of $C_1 = 2v$. In this case the optimal solution contains very few items. An area, for which conditions are not fulfilled, occupies almost the whole domain.

- *average knapsack capacity*

  A knapsack with the capacity $C_2 = 0.5 \sum_{i=1}^{n} W[i]$. In this case about half of the items are in the optimal solution.

As reported in [252], further increasing the value of capacity C does not significantly increase the computation times of the classical algorithms.

4.5.2 Description of the algorithms

Three types of algorithms were implemented and tested: algorithms based on penalty functions ($A_p[i]$, where i is the index of a particular algorithm in this class), algorithms based on repair methods ($A_r[i]$), and algorithms based on decoders ($A_d[i]$). We discuss these three categories of algorithms in turn.

Algorithms $A_p[i]$

In all algorithms in this category a binary string of the length n represents a solution x to the problem: the i-th item is selected for the knapsack iff $x[i] = 1$.

The fitness $eval(x)$ of each string is determined as:

$$eval(x) = \sum_{i=1}^{n} x[i] \cdot P[i] - Pen(x),$$

where penalty function $Pen(x)$ is zero for all feasible solutions x, i.e., solutions such that $\sum_{i=1}^{n} x[i] \cdot W[i] \leq C$, and is greater than zero, otherwise.

There are many possible strategies for assigning the penalty value. Here, three cases only were considered, where the growth of the penalty function is logarithmic, linear, and quadratic with respect to the degree of violation, respectively:

- $A_p[1]$: $Pen(\boldsymbol{x}) = \log_2(1 + \rho \cdot (\sum_{i=1}^{n} x[i] \cdot W[i] - C))$,
- $A_p[2]$: $Pen(\boldsymbol{x}) = \rho \cdot (\sum_{i=1}^{n} x[i] \cdot W[i] - C)$,
- $A_p[3]$: $Pen(\boldsymbol{x}) = (\rho \cdot (\sum_{i=1}^{n} x[i] \cdot W[i] - C))^2$.

In all three cases, $\rho = \max_{i=1..n}\{P[i]/W[i]\}$.

Algorithms $A_r[i]$

As in the previous category of algorithms, a binary string of the length n represents a solution to the problem $\boldsymbol{x}$: the i-th item is selected for the knapsack iff $x[i] = 1$.

The fitness $eval(\boldsymbol{x})$ of each string is determined as:

$$eval(\boldsymbol{x}) = \sum_{i=1}^{n} x'[i] \cdot P[i],$$

where vector $\boldsymbol{x}'$ is a repaired version of the original vector $\boldsymbol{x}$.

There are two interesting aspects here. First, we may consider different repair methods. Second, some percentage of repaired chromosomes may replace the original chromosomes in the population. Such replacement rate may vary from 0% to 100%; recently Orvosh and Davis [301] reported so-call 5% rule which states that if replacing original chromosomes with a 5% probability, the performance of the algorithm is better than if replacing with any other rate (in particular, it is better than with 'never replacing' or 'always replacing' strategies).

We have implemented and tested two different repair algorithms. Both algorithms are based on the same procedure, shown in Figure 4.9.

```
procedure repair (x)
begin
  knapsack-overfilled := false
  x' := x
  if Σ_{i=1}^{n} x'[i] · W[i] > C
    then knapsack-overfilled := true
  while (knapsack-overfilled) do
  begin
    i := select an item from the knapsack
    remove the selected item from the knapsack:
      i.e., x'[i] := 0
    if Σ_{i=1}^{n} x'[i] · W[i] ≤ C
    then knapsack-overfilled := false
  end
end
```

Fig. 4.9. The repair procedure

The two repair algorithms considered here differ only in selection procedure **select**, which chooses an item for removal from the knapsack:

- $A_r[1]$ (random repair). The procedure **select** selects a random element from the knapsack.
- $A_r[2]$ (greedy repair). All items in the knapsack are sorted in the decreasing order of their profit to weight ratios. The procedure **select** chooses always the last item (from the list of available items) for deletion.

Algorithms $A_d[i]$

A possible decoder for the knapsack problem is based on integer representation.[6] Here we used the ordinal representation (see Chapter 10 for more details on this representation) of selected items. Each chromosome is a vector of n integers; the i-th component of the vector is an integer in the range from 1 to $n - i + 1$. The ordinal representation references a list L of items; a vector is decoded by selecting appropriate item from the current list. For example, for a list of items $L = (1, 2, 3, 4, 5, 6)$, the vector $\langle 4, 3, 4, 1, 1, 1 \rangle$ is decoded as the following sequence of items: 4, 3, 6 (since 6 is the 4-th element on the current list after removal of 4 and 3), 1, 2, and 5. Clearly, in this method a chromosome can be interpreted as a strategy of incorporating items into the solution. Additionally, one-point crossover applied to any two feasible parents would produce a feasible offspring. A mutation operator is defined in a similar way as for the binary representation: if the i-th gene undergoes mutation, it takes a value random value (uniform distribution) from the range $[1..n - i + 1]$. The decoding algorithm is presented in Figure 4.10.
The two algorithms based on decoding techniques considered here differ only in the procedure **build**:

- $A_d[1]$ (random decoding). In this algorithm the procedure **build** creates a list L of items such that the order of items on the list corresponds to the order of items in the input file (which is random).
- $A_d[2]$ (greedy decoding). The procedure **build** creates a list L of items in the decreasing order of their profit to weight ratios. The decoding of the vector x is done on the basis of the sorted sequence (there are some similarities with the $A_r[2]$ method). For example, $x[i] = 23$ is interpreted as the 23-rd item (in the decreasing order of the profit to weight ratios) on the current list L.

4.5.3 Experiments and results

In all experiments the population size was constant and equal to 100. Also, probabilities of mutation and crossover were fixed: 0.05 and 0.65, respectively.

[6] There are, of course, other possibilities. We can stay, for example, with binary representation, and interpret a string from the left to the right (i.e., from $i = 1$ to n) in the following fashion: take the i-th item if (1) $i = 1$, and (2) there is a room in the knapsack for this item. Such interpretation always results in a feasible solution. Moreover, if items are sorted by the profit to weight ratio, a solution of all 1's correspond to the solution by the greedy algorithm. An example of such decoder is given in section 15.3, part I.

```
procedure decode (x)
begin
  build a list L of items
  i := 1
  WeightSum := 0
  ProfitSum := 0
  while i ≤ n do
  begin
    j := x[i]
    remove the j-th item from the list L
    if WeightSum + W[j] ≤ C then
    begin
      WeightSum := WeightSum + Weight[j]
      ProfitSum := ProfitSum + Profit[j]
    end
    i := i + 1
  end
end
```

Fig. 4.10. The decoding procedure for the ordinal representation

As a performance measure of the algorithm we have collected the best solution found within 500 generations. It has been empirically verified that after such number of generations no improvement has been observed. The results reported in the Table 4.2 are mean values of the 25 experiments. The exact solutions are not listed there; the table compares only the relative effectiveness of different algorithms. Note, that the data files were unsorted (arbitrary sequences of items, not related to their $P[i]/W[i]$ ratios). The capacity types C_1 and C_2 stand for restrictive and average capacities (section 2), respectively.

Results for the methods $A_r[1]$ and $A_r[2]$ have been obtained using the 5% repair rule. We have also examined whether the 5% rule works for the 0/1 knapsack problem (the rule was discovered during experiments on two other combinatorial problems: network design problem and graph coloring problem [301]). For the purpose of comparison we have chosen the test data sets with the weak correlation between weights and profits. All parameters settings were fixed and the value of the repair ratio varied from 0% to 100%. We have observed no influence of the 5% rule on performance of the genetic algorithm. The results (algorithm $A_r[2]$) have been collected in the Table 4.3.

The main conclusions drawn from the experiments can be summarized as follows:

- Penalty functions $A_p[i]$ (for all i) do not produce feasible results on problems with restrictive knapsack capacity (C_1). This is the case for any number of items ($n = 100$, 250, and 500) and any correlation.

Correl.	No. of items	Cap. type	method						
			$A_p[1]$	$A_p[2]$	$A_p[3]$	$A_r[1]$	$A_r[2]$	$A_d[1]$	$A_d[2]$
none	100	C_1	*	*	*	62.9	94.0	63.5	59.4
		C_2	398.1	341.3	342.6	344.6	371.3	354.7	353.3
	250	C_1	*	*	*	62.6	135.1	58.0	60.4
		C_2	919.6	837.3	825.5	842.2	894.4	867.4	857.5
	500	C_1	*	*	*	63.9	156.2	61.0	61.4
		C_2	1712.2	1570.8	1565.1	1577.4	1663.2	1602.8	1597.0
weak	100	C_1	*	*	*	39.7	51.0	38.2	38.4
		C_2	408.5	327.0	328.3	330.1	358.2	333.6	332.3
	250	C_1	*	*	*	43.7	74.0	42.7	44.7
		C_2	920.8	791.3	788.5	798.4	852.1	804.4	799.0
	500	C_1	*	*	*	44.5	93.8	43.2	44.5
		C_2	1729.0	1531.8	1532.0	1538.6	1624.8	1548.4	1547.1
strong	100	C_1	*	*	*	61.6	90.0	59.5	59.5
		C_2	741.7	564.5	564.4	566.5	577.0	576.2	576.2
	250	C_1	*	*	*	65.5	117.0	65.5	64.0
		C_2	1631.9	1339.5	1343.4	1345.8	1364.4	1366.4	1359.0
	500	C_1	*	*	*	67.5	120.0	67.1	64.1
		C_2	3051.6	2703.8	2700.8	2709.5	2748.1	2738.0	2744.0

Table 4.2. Results of experiments; symbol '*' means, that no valid solution has been found within given time constraints

- Judging only from the results of the experiments on problems with average knapsack capacity (C_2), the algorithm $A_p[1]$ based on logarithmic penalty function is a clear winner: it outperforms all other techniques on all cases (uncorrelated, weakly and strongly correlated, with $n = 100$, 250, and 500 items). However, as mentioned earlier, it fails on problems with restrictive capacity.

- Judging only from the results of the experiments on problems with restrictive knapsack capacity (C_1), the repair method $A_r[2]$ (greedy repair) outperforms all other methods on all test cases.

These results are quite intuitive. In the case of restrictive knapsack capacity, only very small fraction of possible subsets of items constitute feasible solutions; consequently, most penalty methods would fail. This is generally the case for many other combinatorial problems: the smaller the ratio between feasible part of the search space and the whole search space, the harder it is for the penalty function methods to provide feasible results. This was already observed in [332], where the authors wrote:

> "On sparse problems, [harsh penalty functions] seldom found solutions. The solutions which it did occasionally manage to find were poor. The reason for this is clear: with a sparse feasible region, an initial population is very unlikely to contain any solutions. Since [the harsh penalty function] made no distinction between infeasible

correlation	weak					
no. of items	100		250		500	
capacity type	C_1	C_2	C_1	C_2	C_1	C_2
repair ratio						
0	94.0	371.3	134.1	895.0	158.0	1649.1
5	94.0	371.7	135.1	891.4	155.8	1648.5
10	94.0	370.2	135.1	889.5	157.3	1640.3
15	94.0	368.3	135.3	895.4	156.6	1646.2
20	94.0	372.0	135.6	905.7	155.8	1643.6
25	94.0	370.2	135.1	894.0	155.8	1644.0
30	94.0	367.2	135.1	895.7	157.3	1648.0
35	94.0	370.3	136.1	896.5	156.3	1643.3
40	94.0	368.6	134.3	886.5	156.1	1648.4
45	94.0	369.0	135.3	891.5	156.6	1649.0
50	94.0	371.7	134.6	891.0	156.1	1641.5
55	94.0	371.3	135.0	895.7	157.0	1647.0
60	94.0	369.6	135.0	894.0	156.1	1645.0
65	94.0	370.0	135.1	893.2	156.6	1642.8
70	94.0	367.6	135.0	893.4	156.1	1640.9
75	94.0	367.7	135.3	895.7	157.1	1648.1
80	94.0	368.2	134.3	898.5	155.8	1648.5
85	94.0	364.7	135.6	897.4	156.1	1646.2
90	94.0	368.7	134.3	885.2	156.1	1648.9
95	94.0	371.2	135.0	890.5	155.3	1642.3
100	94.0	370.2	134.6	901.0	156.3	1646.1

Table 4.3. Influence of the repair ratio on the performance of the algorithm A_r[2]

> solutions, the GA wandered around aimlessly. If through a lucky mutation or strange crossover, an offspring happened to land in the feasible region, this child would become a *super-individual*, whose genetic material would quickly dominate the population and premature convergence would ensue".

depending on selection: worse with FPS

On the other hand, the repair algorithms perform quite well. In the case of average knapsack capacity, logarithmic penalty function method (i.e., small penalties) is superior: it is interesting to note that the size of the problem does not influence the conclusions.

As indicated earlier, these experiments do not provide a complete picture yet; many additional experiments are planned in the near future. In the category of penalty functions, it might be interesting to experiment with additional formulae. All considered penalties were of the form $Pen(x) = f(x)$, where f was logarithmic, linear, and quadratic. Another possibility would be to experiment with $Pen(x) = a + f(x)$ for some constant a. This would provide a minimum penalty for any infeasible vector. Further, it might be interesting to experiment with *dynamic* penalty functions, where their values depend on additional parameters, like the generation number (this was done in [267] for numerical

optimization for continuous variables) or characteristic of the search (see [360], where dynamic penalty functions were used for the facility layout problem: as better feasible and infeasible solutions are found, the penalty imposed on a given infeasible solution will change). Also, it seems worthwhile to experiment with *self-adaptive* penalty functions. After all, probabilities of applied operators might be adaptive (as in evolution strategies); some initial experiments indicate that adaptive population sizes may have some merit (section 4.4); so the idea of adaptive penalty functions deserves some attention. In its simplest version, a penalty coefficient would be part of the solution vector and undergo all genetic (random) changes (as opposed the idea of dynamic penalty functions, where such penalty coefficient is changed on regular basis as a function of, for example, generation number).

It is also possible to experiment with many repair schemes, including other heuristics than the ratio of the profit and the weight. Also, it might be interesting to combine the penalty methods with repair algorithms: infeasible solutions for algorithms $A_p[i]$ could have been repaired into feasible ones.

In the category of decoders it would be necessary to experiment with different (integer) representations (as it was done for the traveling salesman problem — Chapter 10): adjacency representation (with alternating-edges crossover, subtour-chunks crossover, or heuristic crossover), or path representation (with the PMX, OX, and CX crossovers, or even the edge recombination crossover). It would be interesting to compare the usefulness of these representations and operators for the 0/1 knapsack problem (as it was done for the traveling salesman problem and scheduling [370]). It is quite possible, that some new problem-specific crossover would provide with the best results.

4.6 Other ideas

In the previous sections of this chapter we have discussed some issues connected with removing possible errors in sampling mechanisms. The basic aim of that research was to enhance genetic search; in particular, to fight the premature convergence of GAs. During the last few years there have been some other research efforts in the quest for better schema processing, but using different approaches. In this section we discuss some of them.

The first direction is related to the genetic operator, crossover. This operator was inspired by a biological process; however, it has some drawbacks. For example, assume there are two high performance schemata:

$$S_1 = (0\ 0\ 1 * * * * * * * * 0\ 1) \text{ and}$$
$$S_2 = (* * * * 1\ 1 * * * * * * *).$$

There are also two strings in a population, v_1 and v_2, matched by S_1 and S_2, respectively:

$$v_1 = (0010001101001) \text{ and}$$
$$v_2 = (1110110001000).$$