G9.7 Knapsack problems

Darrell Whitley, V Scott Gordon and A P Willem Böhm

Abstract

Genetic algorithms are applied to various knapsack problems and compared to bounded depth-first search and a form of parallel branch and bound. Two methods of constructing a fitness function are also considered. The genetic algorithms produce poor results compared to the algorithms using bounds; furthermore, parallel branch and bound can potentially execute faster in parallel than fine-grain cellular genetic algorithms.

G9.7.1 Zero–one knapsack problems

The zero–one knapsack problem is defined as follows. Given n objects with positive weights W_i and positive profits P_i, and a knapsack capacity M, determine a subset of the objects, represented by a bit vector $\boldsymbol{X}$ of length n, such that

$$\sum_{i=1}^{n} X_i W_i \leq M \qquad \text{and} \qquad \sum_{i=1}^{n} X_i P_i \text{ is maximal.}$$

An example of a real-world problem which can be modeled as a zero–one knapsack problem is the utilization of communication channels. Given a fixed capacity and a surplus of customers that have different-size communication packets that generate differing amounts of revenue, sell access to a subset of customers in a way that maximizes total profit.

The zero–one knapsack problem is nondeterministic polynomial-time (NP) complete, but a different version of this problem, the *fractional knapsack problem* (Cormen *et al* 1990) has a greedy polynomial-time solution. In the fractional knapsack problem one can place a fraction of an object into the knapsack. Thus, to solve the fractional knapsack problem, objects are iteratively placed in the knapsack by picking the next available object in the unselected set with the highest profit–weight ratio. When no more whole objects fit into the knapsack, select a fractional part of that object in the unselected set with the best profit–weight ratio so that the knapsack is exactly full.

For the zero–one knapsack only whole objects can be placed into the knapsack. In this case, a *greedy approximate solution* is found by inserting objects by profit–weight ratio until the knapsack cannot be filled any further. This also yields a lower-bound estimate on the optimal solution. Problems with poor greedy estimates tend to be harder to solve. It is also easy to see that a greedy solution is not always optimal. Consider the case of three objects where the object with the best profit–weight ratio fills 60% of the knapsack and the other two objects each fill exactly 50% of the knapsack: it is now easy to assign profit–weight ratios such that total profit is maximized by picking the two objects that exactly fill the knapsack rather than the single object with the best profit–weight ratio.

As noted, the greedy solution provides a lower bound on the profit which can be obtained for a zero–one knapsack problem. Any competitive solution should be better than the greedy solution. On the other hand, if we treat the zero–one knapsack problem *as if* it were a fractional knapsack problem, we can also obtain a upper bound on the solution. The solution to the zero–one version of the knapsack problem can be no better than the solution to the fractional knapsack problem.

We can also generate upper and lower bounds with respect to partial solutions. Assume a partial solution has been specified. The partial solution fills part of the knapsack and also removes some objects

from consideration. Treat the residual knapsack capacity as a new knapsack and the unselected objects as the objects in the new *residual* knapsack problem. The upper bound on the residual knapsack problem combined with the profit associated with the partial solution provides an upper bound on the potential profit that can be achieved by extending the partial solution. Bounding algorithms use this information to discard partial solutions that have upper bounds inferior to the current best solution, thus pruning the search space.

B1.2 C1.2 At first glance, *genetic algorithms* would appear to be relatively well suited to the zero–one knapsack problem. The most straightforward problem representation is a *binary string*. Traditional recombination operators also work with this representation. In practice, however, the existence of good upper and lower bounds makes it possible to solve knapsack problems with great speed using exact methods such as branch-and-bound algorithms. More analytical methods have also been developed that solve very large knapsack problems (e.g. 250 000 objects) to optimality (Babayer *et al* 1996, Martello and Toth 1990). These particular results were obtained for the *integer knapsack problem*. This version of the knapsack problem can be characterized as follows:

$$\sum_{i=1}^{n} Y_i W_i \leq M \qquad \text{and} \qquad \sum_{i=1}^{n} Y_i P_i \text{ is maximal}$$

where $\boldsymbol{Y}$ is a vector of nonnegative integers; thus, we have multiple copies of the objects being placed in the knapsack. Of course, it is possible to generate even better greedy solutions to this problem than is possible when the problem is the zero–one knapsack problem. This implies even better lower bounds; nevertheless solving 250 000-object problems to optimality is very impressive.

Small problems are particularly well solved by exact methods. For larger problems, branch-and-bound and bounded depth-first methods with pruning outperform the genetic algorithms both for finding optimal solutions and for finding approximate solutions quickly. These simple methods perform much better than genetic algorithms on this class of problem in spite of the existence of a genetic encoding scheme which exploits useful local information. The results highlight the need for a better understanding of which problems are suitable for genetic algorithms and which problems are not.

E2.1.6 One argument that is often offered in favor of genetic algorithms over other methods is the potential for *parallel execution*. However, our results also suggest that it is unclear whether fine-grain parallel genetic algorithms offer a greater potential for parallelism than branch-and-bound methods for a small 80-object test case examined in this paper. At the same time, the kind of parallelism that is available is different for the two approaches.

The following documents one study of the zero–one knapsack problem. Other work includes that of Khuri *et al* (1994).

G9.7.2 The experiments

Random zero–one knapsack problems were generated based on five parameters: number of objects (n), knapsack capacity (k) as a percentage of total weight of the objects, minimum weight and profit of any object (o), range of weight and profit of any object (v) such that $o + v$ represents the maximum weight (profit) of an object, and a random seed (s). We always use $k = 80\%$ and try to adjust o and v to create a small variance in profit–weight ratio. This seems to generate knapsack problems with poor greedy estimates, although it also generates problems for which the greedy approximation tends to approach the global optimum as the problem size increases. The pseudocode used to generate these problems is as follows. The function randvec(n, l, u) produces a vector of n random integers between l and u:

- $\boldsymbol{M}$ = randvec($n, o, o + v$);
- $\boldsymbol{P}$ = randvec($n, o, o + v$);
- Sort $\boldsymbol{M}$ and $\boldsymbol{P}$ according to P_i/M_i;
- Capacity = $0.8 \sum_{i=1}^{n} M_i$.

The test cases include a 20-object problem, an 80-object problem, and several 500-object and 1000-object problems. The 20-object problem was built such that only one of the five objects with the highest profit–weight ratio will fit into the knapsack. This problem was used previously in experiments by Böhm and Egan (1992). The 80-, 500-, and 1000-object problems were built using the random problem generator.

G9.7.3 Binary encodings for zero–one knapsack problems

A simple encoding scheme for zero–one knapsack problems is to let each bit represent the inclusion or exclusion of one of the n objects from the knapsack. Thus, a bitstring of length n can be used to represent candidate solutions. Sorting the bit vector by profit–weight ratio is useful because, for larger problem instances, greedy estimates are often fairly close to the global optimum in terms of their bit representations. If the objects are ordered in the bitstring by profit–weight ratio, then the greedy approximation appears as a series of 1 bits followed by a series of 0 bits. Solutions that improve upon the greedy solution typically have 1 bits as a prefix, 0 bits as a suffix, and a small region of mixed 1s and 0s at the location in the bitstring that corresponds to the 1s to 0s transition in the greedy solution. Because most good solutions are relatively similar, strings are less likely to be disrupted by crossover.

```
Capacity = 50
Weight   = [25,40,20,70]
Profit   = [70,20,5,10]
Optimum  = 75 at 1010
Greedy   = 70 at 1000
```

Figure G9.7.1. A four-object knapsack problem.

Consider the four-object knapsack problem shown in figure G9.7.1 and note that the greedy method used here is more simplistic than necessary. A better greedy method would continue to try inserting objects until reaching the end of the string. The simplistic method used here was chosen for consistency with previous studies (Böhm and Egan 1992).

The problem with this representation is that it is possible to generate infeasible solutions. In other words, setting too many bits to 1 might overflow the capacity of the knapsack. In the above example, the string 1011, which could easily appear during the normal course of genetic search, is an infeasible candidate solution. Two methods of handling overflow are considered: (i) allow infeasible strings and assign a penalty to the evaluation and (ii) ignore the overflow bits.

The first method is implemented by assigning a penalty equal to the amount of overflow. Thus in the above example the string 1011 would have the value of $-(115 - 50) = -65$. This is referred to as the *penalty* method of knapsack evaluation. C5.2

The second method is implemented by adding items one at a time by profit–weight ratio, scanning the bitstring left to right, stopping when the knapsack overflows. Then, the last item that was added is removed. In the above example the string 1011 would be effectively the same as string 1010, since the fourth bit which caused the overflow would be ignored. This is referred to as the *partial-scan* method of knapsack evaluation. The partial-scan method has the interesting characteristic that a string of all 1 bits always evaluates to the greedy approximation. This provides an easy way of seeding the greedy approximation into the population, if desired.

The partial-scan method produced the best results for all of the problems except the 20-object problem. The 20-object problem has a global optimum of 445 and a greedy approximation of 275. The 80-object problem has a global optimum of 25 729 and a close greedy approximation of 25 713. The 500- and 1000-object problems were also generated by using the random method. The profit and weight arrays for the 20- and 80-object problems, and for one of the 500-object problems, are shown in figures G9.7.2–G9.7.4.

```
Profit   = [275,268,260,250,230,63,40,31,41,25,18,23,42,21,25,16,4,5,6,7]
Weight   = [ 55, 54, 53, 52, 51,15,10, 8,13, 7, 6, 8,16, 9,12,12,5,6,7,8]
Capacity = 100
Optimum  = 445  at   01000111011000000000
```

Figure G9.7.2. The 20-object knapsack problem.

```
Profit   = [955,863,753,533,969,725,581,733,975,865,859,499,671,505,827,433,
            707,947,983,557,213,681,891,999,899,401,591,521,543,927,545,985,
            653,101,569,823,431,447,695,303,539,685,839,885,517,153,195,419,
            381,571,357,215,313,225,481,327,359,285,161,391,267,239,181,117,
            281,229,313,179,259,291,313,179,125, 91,219,179, 77, 53, 31,  1]
Weight   = [  4, 48, 98, 86,218,174,150,190,256,242,252,156,232,186,308,162,
            268,388,424,254,102,378,500,624,572,274,408,370,400,720,434,802,
            542, 86,490,736,392,416,656,288,548,702,904,990,606,194,276,596,
            550,836,558,360,602,442,970,680,784,630,378,920,692,624,478,334,
            818,678,930,540,796,900,978,628,478,364,980,940,718,958,944,786]
Capacity = 12000
Optimum  = 25729 at
11111111111111111111111111111111110100000000000000000000000000000000000000
```

Figure G9.7.3. The 80-object knapsack problem.

G9.7.4 Exact methods

Böhm and Egan (1992) describe five algorithms for solving zero–one knapsack problems: *divide and conquer*, *dynamic programming*, *bounded depth-first*, *memo functions*, and *branch and bound*. We consider only bounded depth first and branch and bound since they were reportedly the most effective. Horowitz and Sahni (1978) also review most of these methods for the zero–one knapsack problem in a tutorial fashion, with both example problems and pseudocode.

Bounded depth-first and branch-and-bound algorithms explore a search tree, where points in the search tree represent partial solutions. A lower bound for the best total solution from this point is computed by adding objects with decreasing profit–weight ratio until an object exceeds the knapsack capacity. An upper bound is computed by adding a fractional part of the first object that exceeded the knapsack capacity, such that the knapsack is filled to capacity. The *bounded depth-first* algorithm starts out with the greedy estimate. The upper bound on the potential solution of subtrees representing partial solutions can be compared to the lower bound, the current best solution. Subtrees whose upper bound is worst than the lower bound can be pruned away. The *branch-and-bound* algorithm searches the state space breadth first. Subtrees are cut by calculating the upper bound of a partial solution and comparing it to a shared global variable containing the current best lower bound.

We compare performance by executing the bounded depth-first, branch-and-bound, and genetic algorithms on a simulator of the Manchester Dataflow Machine (Gurd *et al* 1987), which provides statistics such as total instructions executed, critical path, and average parallelism.

G9.7.5 Comparative results: small knapsack problems

Gordon and Whitley (1993) reported results for knapsack problems using several genetic algorithm implementations, including a simple genetic algorithm (SGA), Genitor, a simplified CHC algorithm, several parallel *island models*, and a parallel fine-grain *cellular genetic algorithm* (CGA). Although Genitor performed the best overall, additional experiments using smaller population sizes determined that the optimal performance of any of the genetic algorithms on the 80-object knapsack problem was obtained by using the CGA. With a population size of 25, and the partial-scan encoding method, the CGA requires 26 generations (averaged over 30 runs) to solve the 80-object knapsack. The results reported here are therefore based on the CGA.

C6.3, C6.4

Although it is too slow in practice to run the genetic algorithm to optimality on the 80-object knapsack within the dataflow simulator, it is possible to estimate the total number of instructions required for such execution as follows. First, the number of instructions required for a single generation is estimated by running the algorithm for two and three generations, then finding the difference of the reported number of instructions n for each run. The number of instructions n_g for a single generation is approximately $n_3 - n_2$. It follows that the number of instructions n_i for the initialization phase is approximately $n_2 - 2n_g$. The total number of instructions n_k then for a complete run of k generations is $n_i + kn_g$. Similarly, the critical path cp_k can be estimated using critical paths cp_2 and cp_3 reported for runs of two and three generations.


```
Weight   = [101,100,104,108,101,108,110,110,110,101,106,115,115,116,115,103,114,118,
    116,118,114,124,105,105,107,113,117,122,119,107,126,108,107,113,110,101,117,100,
    114,120,104,118,120,129,103,123,117,133,121,117,126,119,133,124,111,133,136,117,
    129,136,100,140,102,140,111,127,135,142,126,105,118,124,143,125,127,134,133,148,
    133,143,103,116,126,124,115,139,139,150,107,121,127,139,153,101,155,145,126,115,
    146,133,149,142,111,152,145,159,156,115,115,143,102,149,103,125,110,124,138,145,
    109,110,124,131,164,129,153,130,131,157,140,145,105,122,129,124,124,126,148,165,
    100,117,163,169,158,167,162,105,111,159,118,143,168,102,126,123,151,132,154,111,
    169,149,115,155,159,107,174,166,168,170,146,172,164,121,169,172,147,170,105,150,
    167,159,109,112,180,113,125,104,107,107,110,136,152,116,129,182,135,180,184,129,
    119,105,151,155,143,176,179,102,103,181,185,114,115,154,170,175,139,165,170,148,
    174,105,178,188,136,138,143,150,174,137,139,193,147,101,165,179,155,187,135,147,
    150,156,156,174,167,157,145,122,193,184,181,169,144,189,189,121,171,167,159,105,
    195,194,122,194,148,110,104,162,187,154,122,144,108,168,116,139,136,178,154,182,
    116,135,168,198,142,192,188,167,167,161,176,155,168,196,174,158,186,135,189,188,
    157,152,113,150,145,179,196,155,199,118,147,166,154,199,158,148,177,192,170,151,
    112,193,136,188,148,173,164,195,192,182,170,193,181,149,141,145,198,182,163,191,
    191,198,186,185,148,190,179,169,173,158,171,143,169,168,120,119,181,140,140,152,
    149,172,142,158,198,157,162,148,176,153,135,191,195,160,171,193,193,181,169,163,
    157,150,134,174,194,198,146,165,185,191,132,146,198,187,147,146,183,164,179,128,
    160,137,146,173,140,147,178,195,158,140,192,183,194,175,132,135,143,180,150,132,
    196,154,186,197,153,146,194,167,159,140,155,181,156,193,185,198,172,150,153,189,
    156,141,184,188,195,193,162,186,148,189,147,188,191,152,176,198,182,172,194,186,
    175,181,173,197,169,193,158,188,198,197,185,157,195,163,160,193,184,167,160,163,
    198,191,196,175,182,176,174,178,173,189,171,196,183,199,196,182,184,195,185,195,
    188,199]
Profit   = [197,187,192,199,186,196,196,194,194,176,182,197,196,194,191,171,188,194,
    188,191,184,199,167,167,170,178,184,188,183,164,193,165,163,172,167,153,177,151,
    172,181,156,177,180,193,154,183,174,197,179,173,186,175,195,181,162,193,197,169,
    186,196,144,198,143,196,155,177,188,196,173,144,161,169,194,169,171,180,178,197,
    176,189,135,152,165,161,149,180,180,194,138,156,162,177,194,128,196,183,159,145,
    184,167,187,178,139,190,181,198,194,143,143,177,126,184,127,154,135,152,169,177,
    133,134,151,159,199,156,185,157,158,189,168,174,126,146,154,148,148,149,175,195,
    118,138,192,199,186,196,190,123,130,186,138,167,196,119,146,142,174,152,177,127,
    193,170,131,176,179,120,195,186,188,190,163,192,183,135,188,191,163,188,116,165,
    183,174,119,122,196,123,136,113,116,116,119,147,164,125,139,196,145,193,197,138,
    127,112,161,165,152,187,190,108,109,191,195,120,121,162,178,183,145,172,177,154,
    181,109,184,194,140,142,147,154,178,140,142,197,150,103,168,182,157,189,136,148,
    151,157,157,173,166,156,144,121,191,182,179,167,142,186,186,119,168,164,156,103,
    191,190,119,189,144,107,101,157,181,149,118,139,104,161,111,133,130,170,147,173,
    110,128,159,187,134,181,177,157,157,151,165,145,157,183,162,147,173,125,175,174,
    145,140,104,138,133,164,179,141,181,107,133,150,139,179,142,133,159,172,152,135,
    100,172,121,167,131,153,145,172,169,160,149,169,158,130,123,126,172,158,141,165,
    165,171,160,159,127,163,153,144,147,134,145,121,143,142,101,100,152,117,117,127,
    124,143,118,131,164,130,134,122,145,126,111,157,160,131,140,158,158,147,137,132,
    127,121,108,140,156,159,117,132,148,152,105,116,157,148,116,115,144,129,140,100,
    125,107,114,135,109,114,138,151,122,108,148,141,149,134,101,103,109,137,114,100,
    148,116,139,147,114,108,143,123,117,103,114,133,114,141,135,144,125,109,111,137,
    113,102,133,135,140,138,115,132,105,134,104,133,135,107,123,138,126,119,133,127,
    119,123,117,133,114,130,106,126,132,131,123,104,129,107,104,124,118,107,102,103,
    122,117,120,107,110,106,104,106,103,112,101,114,106,115,110,102,103,109,103,106,
    102,106]
```

Figure G9.7.4. One of the 500-object problems, having capacity 49 117 and optimum 55 928.

For the CGA with population size 25, the dataflow simulator reports $n_2 = 713\,174$ and $n_3 = 991\,405$. It follows that $n_g = 991\,405 - 713\,174 = 278\,231$, $n_i = 713\,174 - 2(278\,231) = 156\,712$. For critical path, the dataflow simulator reports $cp_2 = 32\,871$ and $cp_3 = 34\,520$. It follows that $cp_g = 34\,520 - 32\,871 = 1649$, $cp_i = 32\,871 - 2(1649) = 29\,573$. Therefore the total number of instructions n_k and the critical path length cp_k for solving the problem are approximately given by

$$n_k = n_i + 26n_g = 156\,712 + 26(278\,231) = 7\,390\,718$$
$$cp_k = cp_i + 26cp_g = 29\,573 + 26(1649) = 42\,874.$$

Table G9.7.1 compares the statistics generated by the dataflow simulator.

Table G9.7.1. The genetic algorithm versus other methods on the 80-object knapsack.

Algorithm	Total instructions	Critical path
Massively parallel GA	7 390 718	42 874
Bounded depth first	1 142 637	355 711
Branch and bound	128 636	16 582

Both bounded depth-first and branch and bound find solutions faster (fewer total instructions) than the genetic algorithm. The short critical path for branch and bound indicates that this algorithm could execute faster in parallel than the genetic algorithm. At the same time, the parallelism of the branch-and-bound approach is irregular and a bottleneck exists in terms of the need to communicate lower-bound information. The genetic algorithm has more regular parallelism, only local communication, and no bottleneck. At the same time, the figure for the critical path of the genetic algorithm may be optimistic, since some of the parallelism is bit level parallelism that may not be realizable in an actual implementation. Thus, the question of which approach is best suited to parallel implementation is unclear. It is nonetheless true that bounded depth first is about six times faster than the genetic algorithm, and branch and bound is about 60 times faster.

G9.7.6 Large knapsack problems

Since knapsack problems define a search space of 2^n combinations of objects, exhaustive search methods will eventually fail on very large problems. While this is probably also true for genetic algorithms, perhaps the genetic algorithm can provide better solution estimates part way through the search than standard methods can provide. To test this, the CGA and the bounded depth-first and branch-and-bound algorithms were run on a set of four larger knapsack problems, two of size 500 and two of size 1000. All of the algorithms were run on a Sparc 2 with 32 Mbytes of memory. The 'best-so-far' values for each algorithm at various times during the search were compared. Several population sizes for the genetic algorithm were tested.

Recall that for larger problems the greedy estimate is close to the global optimum. While one would expect this to benefit the genetic algorithm performance, it also helps simple exact methods prune the search space more effectively. In fact, it was necessary to generate 30 knapsack problems of 500 and 1000 objects in order to find *four* that the exact methods did not solve within one second on the Sparc 2. Performance results are shown in table G9.7.2.

Table G9.7.2. Genetic algorithm against other search methods on hard knapsack problems.

		ks_1^{500}	ks_2^{500}	ks_1^{1000}	ks_2^{1000}
Global optimum		55 928	56 448	336 983	630 972
Greedy estimate		55 907	56 366	336 699	630 397
Depth first	time to solve	1 h	> 3 h	> 3 h	25 min
	estimate @ 10 s	55 927	56 446	336 982	630 972
Branch and bound	time to solve	1 s	7 s	24 s	never[a]
	estimate @ 10 s	solved	solved	—	—
CGA popsize = 25	time to solve	∞	∞	∞	∞
	estimate @ 90 s	55 879	56 349	334 963	626 398
CGA popsize = 100	time to solve	∞	∞	∞	∞
	estimate @ 90 s	55 820	56 315	329 672	616 064
	estimate @ 180 s	55 912	56 419	336 583	630 154
CGA popsize = 400	time to solve	∞	∞	∞	∞
	estimate @ 20 min	55 924	56 437	336 852	630 594

[a] Branch and bound exceeds memory for the ks_2^{1000} problem.

'Best-so-far' values for each algorithm at various times during the search are compared. On 500-object problems, the genetic algorithm requires 3 minutes just to reach the greedy estimate (note that the greedy estimate can always be determined directly, requiring only a sort which can be done in polynomial time). Branch and bound performs significantly faster than either of the other algorithms, but space demands cause it to fail on one of the problems. Bounded depth-first also clearly beats the genetic algorithm, since the genetic algorithm never reaches the estimate which the bounded depth first algorithm finds after 10 seconds. It is interesting to note that on problem ks_2^{1000}, the bounded depth-first algorithm finds the global optimum after only 10 seconds, but requires 25 more minutes to *know* that it is the optimum. *The genetic algorithm takes 25 minutes just to find the greedy estimate.*

The genetic algorithm results can be improved slightly by seeding the population with a copy of the greedy estimate. As stated earlier, this is easily done by setting all of the bits in one of the strings to 1. Tests determined that doing this does not change the comparison with bounded depth first with regards to solving the problems to optimality, or obtaining better estimates. For the longer time periods, the estimates produced by the genetic algorithm were close to those shown in table G9.7.2.

G9.7.7 Conclusions

These findings strengthen the notion that genetic algorithms are general-purpose algorithms *not intended to supplant existing methods for solving all problems*. The algorithms used here are also rather simple general purpose search algorithms. Martello and Toth (1990) report solving much larger knapsack problems than ours in under 1 minute using more specialized forms of branch and bound; code for branch and bound is also readily available (see e.g. Horowitz and Sahni 1978). The analytical methods proposed by Babayev *et al* (1996) appear to be even better. It seems reasonable to infer that pure genetic search could not match the kind of performance these researchers report since the cost of evaluating a population large enough to adequately sample such a huge space would be excessive.

Application domains such as the zero–one knapsack may not be well suited to blind genetic search, and genetic approaches to such problems may instead require a hybrid approach. On the other hand, problem representation can make a huge difference and perhaps a better representation could dramatically improve the performance of the genetic algorithm. Yet, the bounded depth-first and branch-and-bound methods exploit a great deal of problem specific knowledge and run extremely fast. These algorithms also do not have the overhead of a population-based search. We clearly need a better understanding of which problems cannot be solved practically by exact methods, and which may lend themselves instead to genetic or hybrid approaches.

Some knapsack problems are included in the current paper. We would advise researchers interested in working with the knapsack problem to use these problems only as a starting point. Any serious studies of the knapsack problem should look at much larger problems (e.g. 100 000 variables) and should compare results with methods such as branch and bound.

References

Babayev D, Glover F and Ryan J 1996 A new knapsack solution approach by integer equivalent aggregation and consistency determination *Mathematical Programming* at press

Cormen T, Leiserson C and Rivest R 1990 *Introduction to Algorithms* (Cambridge, MA: MIT Press)

Böhm A and Egan G 1992 Five ways to fill your knapsack *Proc. 2nd Sisal Workshop* LLNL CONF-9210270

Gordon V and Whitley D 1993 Serial and parallel genetic algorithms as function optimizers *Proc. 5th Int. Conf on Genetic Algorithms (Urbana-Champaign, July 1993)* ed S Forrest (San Mateo, CA: Morgan Kaufmann)

Gurd J, Kirkham C and Böhm W 1987 The Manchester dataflow computing system *Exper. Parallel Comput. Arch. (Special Topics in Supercomputing 1)* ed J Dongarra (Amsterdam: North-Holland)

Horowitz E and Sahni S 1978 *Fundamentals of Computer Algorithms* (Computer Science Press)

Khuri S, Bäck T and Heitkötter F 1994 The zero–one multiple knapsack problem and genetic algorithms *Proc. 1994 ACM Symp. on Applied Computing* ed E Deaton, D Oppenheim, F Urban and H Berghel (ACM)

Martello S and Toth P 1990 *Knapsack Problems: Algorithms and Computer Implementations* (New York: Wiley)