# A Note on the Performance of Genetic Algorithms on Zero-One Knapsack Problems

V. Scott Gordon, A. P. Wim Böhm, and Darrell Whitley
Department of Computer Science, Colorado State University

*Keywords: zero-one knapsack, dataflow computing, performance comparisons of genetic algorithms vs. traditional search methods.*

## Introduction

For small zero-one knapsack problems, simple branch-and-bound and depth-first methods generate solutions much faster than our genetic algorithms. For large problems, branch-and-bound and depth-first methods outperform the genetic algorithms both for finding optimal solutions and for finding approximate solutions quickly. The 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.

## 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 $X$, such that:

$$\sum_{i=1}^{n} X_i W_i \leq M \quad and \quad \sum_{i=1}^{n} X_i P_i \quad maximal$$

A *greedy estimate* is found by inserting objects by profit weight ratio until the knapsack cannot be filled any further. Problems with poor greedy estimates tend to be harder to solve.

Random zero-one knapsack problems were generated based on five parameters: number of objects ($n$), knapsack capacity ($p$) 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 $p = 80\%$, 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.

The test cases include a 20-object problem, an 80-object problem, and several 500 and 1000-object problems. The 20-object problem was built by generating a random 15-object problem (as described above) and adding 5 more objects in such a way as to generate a poor greedy estimate. This problem was used previously in experiments by Böhm and Egan [1]. The 80-object problem was built entirely by using the random method described earlier.

## Genetic Encoding

A simple genetic encoding scheme for zero-one knapsack problems is as follows. Let each bit represent the inclusion or exclusion of one of the $n$ objects from the knapsack, by profit/weight ratio from left to right. Note that it is possible to represent infeasible solutions by setting so many bits to "1" that the weight of the corresponding set of objects overflows the capacity of the knapsack.

We consider two ways of handling overflow. The first *penalty* method assigns a penalty equal to the amount of overflow. The second method, *partial scan*, adds items to the knapsack one at a time, scanning the bitstring left to right, stopping when the knapsack overflows. Then, the last item that was added is removed.

Since the objects are ordered by profit weight ratio, the greedy estimate appears as a series of "1" bits followed by a series of "0" bits. The partial scan method has the interesting property that a string of all "1" bits evaluates to the greedy estimate. This provides an easy way of seeding the greedy estimate into the population.

Our 20-object problem has a global optimum of 445 and a greedy estimate of 275. The 80-object problem has a global optimum of 25729 and a closer greedy estimate of 25713. Using our genetic algorithms, the 20-object knapsack problem is *harder* to solve than the 80-object knapsack problem. Further, the penalty evaluation method works better on the 20-object problem, and the partial scan method works better on the 80-object problem [4].

## Exact Methods

Bohm and Egan [1] describe five algorithms for solving zero-one knapsack problems: *divide and conquer*, *dynamic programming*, *depth first with bound*, *memo functions*, and *branch and bound*. We consider only depth-first and branch-and-bound since they were reportedly the most effective.

Points in the search space 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 part of the object that exceeded the knapsack capacity, such that the knapsack is filled to capacity. The *depth first with bound* algorithm starts out with the greedy estimate. At a certain point in the search, the upper bound is used to determine whether the subtree under this point needs to be searched further. The *branch and bound* algorithm searches the state space breadth first. Sub-trees are cut by calculating the upperbound of a partial solution and comparing it to a shared variable containing the current best lower bound.

We compare performance by executing the depth-first, branch-and-bound, and the genetic algorithm on a simulator of the Manchester Dataflow Machine [5], which provides statistics such as total instructions executed, critical path, and average parallelism.

## Comparative Results – Small Knapsack Problems

Gordon and Whitley [4] reported results for knapsack problems using several genetic algorithm implementations, including a Simple Genetic Algorithm, Genitor, a simplified CHC, several Island Models, and a Fine Grain Cellular Genetic Algorithm. 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 cellular genetic algorithm (CGA). With a population size of 25, and the partial scan encoding method, CGA requires 26 generations (averaged over 30 runs) to solve the 80-object knapsack. The results reported here are therefore based on the cellular algorithm, since they represent the lowest total instructions executed and critical path for our genetic algorithms.

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 2 and 3 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 2 and 3 generations.

For CGA with population size 25, the dataflow simulator reports $n_2 = 713174$, $n_3 = 991405$, $cp_2 = 32871$, and $cp_3 = 34520$. Estimates for $n_k$ and $cp_k$ are shown in Table 1, along with corresponding values for depth-first and branch-and-bound (as reported by the simulator).

| Algorithm | Total Inst | Crit Path |
|---|---|---|
| Massively Parallel GA | 7390718 | 42874 |
| Depth-First Search | 1142637 | 355711 |
| Branch-and-Bound | 128636 | 16582 |

Table 1: GA vs. other methods on 80-object knapsack

Both 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 also has greater potential parallelism than the genetic algorithm. Overall, depth-first is about six times faster than the genetic algorithm, and branch-and-bound is about sixty times faster.

## 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 likely also true for genetic algorithms, perhaps the genetic algorithm can provide better solution estimates part way through the search than standard methods. On the other hand, for larger problems the greedy estimate is often close to the global optimum, which helps simple exact methods prune the search space more effectively. We found it necessary to generate *thirty* knapsack problems of 500 and 1000 objects in order to find *four* that the exact methods did not solve within one second.

"Best-so-far" values for each algorithm at various times during the search are compared in Table 2. Several population sizes for the genetic algorithm are tested. On 500-object problems, the genetic algorithm requires 3 minutes *just to reach the greedy estimate*. On 1000-object problems the genetic algorithm requires 20 minutes to reach the greedy estimate. Branch-and-bound performs significantly faster than either algorithm, but space demands cause it to fail on one of the problems. Depth-first also clearly beats the genetic algorithm, since the genetic algorithm never reaches the estimate which the depth-first algorithm finds after 10 seconds. It is interesting to note that on problem $ks_2^{1000}$, the 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 this long 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 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 2.

| | $ks_1^{500}$ | $ks_2^{500}$ | $ks_1^{1000}$ | $ks_2^{1000}$ |
|---|---|---|---|---|
| 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 hr | $>$ 3 hrs | $>$ 3 hrs | 25 min |
| estimate @10 sec | 55,927 | 56,446 | 336,982 | 630,972 |
| Branch-bound | | | | |
| time-to-solve | 1 sec | 7 sec | 24 sec | never* |
| estimate @10 sec | solved | solved | — | — |
| GA, popsize = 25 | | | | |
| time-to-solve | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| estimate @90 sec | 55,879 | 56,349 | 334,963 | 626,398 |
| GA, popsize = 100 | | | | |
| time-to-solve | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| estimate @90 sec | 55,820 | 56,315 | 329,672 | 616,064 |
| estimate @180 sec | 55,912 | 56,419 | 336,583 | 630,154 |
| GA, popsize = 400 | | | | |
| time-to-solve | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| estimate @20 min | 55,924 | 56,437 | 336,852 | 630,594 |

(*) Branch-and-bound exceeds memory for the $ks_2^{1000}$ problem.

Table 2: GA vs. other search methods on large knapsack problems.

## 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 report solving much larger knapsack problems than ours in under one minute using more specialized forms of branch and bound [6]. It seems reasonable to infer that pure genetic search could not match this kind of performance since the cost of evaluating a population large enough to adequately sample such a huge space would be excessive. Gendreau *et al* are producing similarly superior results (over genetic algorithms) on traveling salesman problems [2].

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. 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.

## References


[1] A. Böhm and G. Egan. Five Ways to Fill Your Knapsack. *Colorado State University technical report CS-92-127*, 1992.

[2] M. Gendreau, A. Hertz, and G. LaPorte. New Insertion and Postoptimization Procedures for the Traveling Salesman Problem. *Operations Research*, Vol 40 #6, Nov-Dec 1992.

[3] V. Gordon, D. Whitley, and A. Böhm. Dataflow Parallelism in Genetic Algorithms. *PPSN2*, North Holland, 1992

[4] V. Gordon, D. Whitley. Serial and Parallel Genetic Algorithms as Function Optimizers. *Proceedings of the 5th ICGA*, Morgan Kaufmann, 1993

[5] J. Gurd, C. Kirkham, and W. Böhm. The Manchester Dataflow Computing System. in J. Dongarra, *Exper. Parallel Comp. Arch*, Special Topics in Supercomputing 1, North Holland, 1987

[6] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*, J. Wiley and Sons, ©1990