Department of Computer Science

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

V. Scott Gordon, A. P. Wim Bohm,
and Darrell Whitley

Technical Report CS-93-108

June 21, 1993

Colorado State University

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
Fort Collins, Colorado 80523 USA
gordons/bohm/whitley@cs.colostate.edu

INTRODUCTION

For small zero-one knapsack problems, simple depth-first and branch-and-bound methods generate solutions much faster than our genetic algorithms. For large problems, simple depth-first and branch-and-bound 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 $\sum_{i=1}^{n} X_i W_i \leq M$ *and* $\sum_{i=1}^{n} X_i P_i$ *maximal*. A *greedy approximation* is found by inserting objects by profit weight ratio until the knapsack cannot be filled any further. The greedy approximation tends to be closer to the global optimum for larger problems.

Random instances of the zero-one knapsack problem were generated based on five user-defined parameters: number of objects (n), knapsack capacity (p) as a percentage of total weight of the objects, minimum weight/profit of any object (o), variance of weight/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 harder 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.

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 corresponding set of objects overflows the capacity of the knapsack.

We consider two methods 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 at the end of the string or when the knapsack overflows, in which case the last item added is removed.

Since the objects are sorted by profit weight ratio, the greedy approximation 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 always evaluates to the greedy approximation. This provides an easy way of seeding the greedy approximation into the population if desired.

Our 20-object problem has a global optimum of 445 and a greedy approximation of 275. The 80-object problem has a global optimum of 25729 and a closer greedy approximation of 25713. Using our genetic algorithms, the 20-object knapsack problem is *harder* to solve than the 80-object knapsack problem. Further, we find that the *penalty* evaluation method works better on the 20-object problem, and the *partial scan* method works better on the 80-object problem [GW93].

COMPARATIVE RESULTS

Bohm and Egan have described five Sisal implementations of existing algorithms for exactly solving zero-one knapsack problems [BE92]: *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 these were reported as being the most effective. Since our genetic algorithms are also written in Sisal [GWB92], we can compare performance by executing the various algorithms on a simulator of the Manchester Dataflow Machine [GKB87]. The simulator provides useful statistics such as total instructions executed, critical path, average parallelism, etc.

We tried several genetic algorithm implementations [GW93], including a Simple Genetic Algorithm, Genitor, a simplified CHC, several Island Models, and a Fine Grain Cellular Genetic Algorithm. For our knapsack problems, the cellular version achieved better solutions and required less running time. The results reported here are therefore based on the cellular algorithm.

The genetic algorithm takes an average of 26 generations (over 30 runs) to solve the 80-object knapsack. Our fastest time-to-solve was achieved with a population size of 25. Table 1 compares the statistics generated by the dataflow simulator for the various algorithms.

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

Table 1: Comparison of 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.

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, recall that for larger problems the greedy estimate is 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 exhaustive methods did not solve immediately.

"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*. 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

	$knapsack_1^{500}$	$knapsack_2^{500}$	$knapsack_1^{1000}$	$knapsack_2^{1000}$
Global solution	55928	56448	336983	630972
Greedy estimate	55907	56366	336699	630397
Depth-first				
time-to-solve	1 hour	$>$ 3 hours	$>$ 3 hours	25 min
est @ 10 sec	55927	56446	336982	630972
Branch-bound				
time-to-solve	1 sec	7 sec	24 sec	*never**
Genetic Algorithm				
$N = 25,\ T = 90s$	55879	56349	334963	626398
$N = 100,\ T = 90s$	55820	56315	329672	616064
$N = 100,\ T = 180s$	55912	56419	336583	630154
$N = 400,\ T = 20m$	55924	56437	336852	630594

(*) Branch-and-bound exceeds memory for the $knapsack_2^{1000}$ problem.

Table 2: GA vs. other methods on large knapsacks. N = population size, T = running time.

genetic algorithm never reaches the estimate which the depth-first algorithm finds after 10 seconds. It is interesting to note that on $knapsack_2^{1000}$, the depth-first algorithm finds the global optimum after only 10 seconds, but requires 25 more minutes to *know* that it is the optimum. Further tests indicate that seeding the population with the greedy estimate does not affect the results.

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 [MT90]. It seems reasonable to infer that a genetic algorithm 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 [GHL92]. We clearly need a better understanding of which problems are suitable for genetic algorithms, and which problems are not.

[BE92] A. Böhm and G. Egan. Five Ways to Fill Your Knapsack. *Colorado State University technical report CS-92-127*, 1992.

[GHL92] 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.

[GWB92] V. Gordon, D. Whitley, and A. Böhm. Dataflow Parallelism in Genetic Algorithms. *PPSN2*, North Holland, 1992

[GW93] V. Gordon, D. Whitley. Serial and Parallel Genetic Algorithms as Function Optimizers. *ICGA-5*, Morgan Kaufmann, 1993.

[GKB87] J. Gurd, C. Kirkham, and W. Böhm. The Manchester Dataflow Computing System. in J. Dongarra, *Experimental Parallel Comp Arch*, Special Topics in Supercomputing 1, North Holland, 1987

[MT90] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*, J. Wiley and Sons, ©1990