Published in *The Application Handbook of Genetic Algorithms*, Volume I, Lance Chambers, editor, pages 143-172, CRC Press, 1995

Using LibGA to Develop Genetic Algorithms for Solving Combinatorial Optimization Problems*

Arthur L. Corcoran
artc@ons.octel.com

Roger L. Wainwright
rogerw@penguin.mcs.utulsa.edu

Department of Mathematical and Computer Sciences
The University of Tulsa
600 South College Avenue
Tulsa, OK 74104-3189

Phone: (918) 631-2228
Fax: (918) 631-3077

Abstract

In this paper we provide an introduction to genetic algorithms and how they are used to solve combinatorial optimization problems. We describe LibGA, a genetic algorithm development library. LibGA is used to solve three simple combinatorial optimization problems: bin packing, the traveling salesman problem, and multiprocessor scheduling. Sufficient details of LibGA are provided to enable the reader to easily use LibGA as a tool for solving additional combinatorial optimization problems.

1 Introduction

Combinatorial optimization problems are among the most difficult problems faced by Computer Scientists. These problems collectively belong to the class of NP-complete problems. The genetic algorithm is a search technique which borrows ideas from natural evolution to effectively find good solutions for combinatorial optimization problems.

This paper is organized as follows: Section 2 provides a brief introduction to genetic algorithms. Section 3 describes how combinatorial optimization and genetic algorithms are related. Section 4 introduces LibGA, a genetic algorithm development library developed for solving combinatorial problems. Section 5 describes three different combinatorial optimization problems: bin packing, the traveling salesman problem, and multiprocessor scheduling. Code and sample output are provided for a simple implementation of these problems using LibGA. Conclusions are presented in Section 6.

2 Genetic Algorithms

A *genetic algorithm* (GA) is an adaptive search technique based on the principles and mechanisms of natural selection and 'survival of the fittest' from natural evolution. GAs grew out of Holland's [21] study of adaptation in artificial and natural systems. By simulating natural evolution in this way, a GA can effectively search the problem domain and easily solve complex problems. Furthermore, by emulating biological selection and reproduction techniques, a GA can perform the search in a general, representation-independent manner.

The genetic algorithm operates as an iterative procedure on a fixed size population or pool of candidate solutions. The candidate solutions represent an encoding of the problem into a form that is analogous to the chromosomes of biological systems. Each chromosome represents a possible solution for a given objective function. Associated with each chromosome is a fitness value, which is found by evaluating the chromosome with the objective function. It is the fitness of a chromosome which determines its ability to survive and produce offspring. Each chromosome is made up of a string of genes (whose values are called alleles). The chromosome is typically represented in the GA as a string of bits. However, integers and floating point numbers can easily be used.

*Research partially supported by OCAST Grant AR2-004 and Sun Microsystems, Inc.

```
procedure GA
begin
  t = 0;
  initialize P(t)
  evaluate structures in P(t);
  while termination condition not satisfied do
  begin
    t = t + 1;
    P(t) = select from P(t-1)
    alter structures in P(t);
    evaluate structures in P(t);
  end
end.
```

Figure 1: Genetic algorithm

Figure 1 illustrates a 'canonical' genetic algorithm. The GA begins by generating an initial population, $P(t = 0)$, and evaluating each of its members with the objective function. While the termination condition is not satisfied, a portion of the population is selected, somehow altered, evaluated, and placed back into the population. At each step in the iteration, chromosomes are probabilistically selected from the population for reproduction according to the principle of the 'survival of the fittest'. Offspring are generated through a process called crossover, which can be augmented by mutation. The offspring are then placed back in the pool, perhaps replacing other members of the pool. This process can be modeled using either a 'generational' [20, 21] or a 'steady-state' [39] genetic algorithm. The generational GA saves offspring in a temporary location until the end of a generation. At that time the offspring replace the entire current population. Conversely, the steady-state GA immediately places offspring back into the current population.

The genetic algorithm relies on genetic operators for selection, crossover, mutation, and replacement. The selection operators use the fitness values to select a portion of the population to be parents for the next generation. Parents are combined using the crossover and mutation operators to produce offspring. This process combines the fittest chromosomes and passes superior genes to the next generation, thus providing new points in the solution space. The replacement operators ensure that the 'least fit' or weakest chromosomes of the population are displaced by more fit chromosomes.

While the fundamental concepts of genetic algorithms are fairly simple and straightforward, there are numerous implementation variations and options to incorporate into a genetic algorithm. For example, there are numerous ways to parameterize a model and encode it into a finite length chromosome. There are numerous selection techniques for determining chromosomes for crossover. There are literally dozens of possible crossover operators that have been developed in recent years depending on the problem type and chromosome encoding scheme. There are also several techniques for introducing some random changes to a chromosome (i.e., mutation). In this paper, we assume the reader is familiar with the fundamental concepts of genetic algorithms.

3 Combinatorial Optimization

Methods to solve difficult combinatorial problems can be divided into two types. The first type includes those methods which try to find optimal solutions through an 'intelligent' exhaustive search. This includes techniques such as backtracking, branch and bound, implicit enumeration, and dynamic programming. Such techniques are only useful for solving combinatorial problems with small sizes. The other type of method for solving combinatorial problems relys on optimization. That is, rather than finding the absolute optimal solution, a 'good' solution is desired within an acceptable time period. These are known as combinatorial optimization techniques. These methods usually employ heuristic algorithms which are problem specific. Since the true optimal is often unknown and impossible to determine, the 'optimal' solution is usually considered the best one obtainable.

Combinatorial optimization methods have as their goal the minimization or maximization of a problem. They are composed of three parts. First, there is a set of problem *instances*. Second, for each problem instance, there is a finite set of *candidate solutions*. Finally, there is a function which assigns to each instance and candidate solution a positive rational number called the *solution value* for the candidate solution. Notice how these elements correspond to those found in genetic algorithms. For a particular problem instance, the GA maintains a set of candidate solutions which are evaluated by the problem specific evaluation function. The solution value returned by the function is used by the GA to measure the relative fitness of that candidate solution. This information is used with the idea of 'survival of the fittest' to conduct the genetic search. As a result GAs are very successful in finding good near-optimal solutions

for combinatorial optimization problems.

4 LibGA

The LibGA software package [11] was developed primarily because of the noticeable deficiencies of existing GA packages at the time. LibGA is a collection of routines written in the C programming language. It can run on a variety of workstations and PC's. However, since everything in LibGA is in double precision and since genetic algorithms are inherently CPU bound, we elected to implement LibGA in a workstation environment using Unix. We found that executing LibGA on a PC was extremely slow. LibGA provides a user-friendly workbench for genetic algorithm research. It is especially useful for working with combinatorial problems which are often *order-based*. That is, problems for which the representation depends on a certain order being preserved, as with a permutation representation, for example. LibGA includes a rich set of genetic operators for selection, crossover and mutation. An important feature of LibGA is the ability to implement both a generational and steady-state genetic algorithms using the genetic operators. This allows researchers the ability to compare between the two approaches. Other features of LibGA include a generation gap, elitism, and the ability to implement a dynamic generation gap. Other routines are provided for initialization, reading a configuration file and generating various statistical reports. LibGA has been requested and sent to locations all over the world. In addition, many have obtained LibGA via anonymous ftp from the 'GA Digest' archive. Information for obtaining LibGA can be found at the end of this paper.

The operators in LibGA include selection, replacement, crossover, and mutation. Selection and replacement can be augmented with elitism. This ensures that the best member of a population survives into the next generation. The selection operators included in LibGA are: *uniform-random*, *roulette*, and *rank-biased*. Uniform-random selection picks a member of the pool at random, completely ignoring fitness or other factors. Thus each chromosome in the pool is equally likely to be selected. Roulette is the classic selection method used in generational GAs and rank-biased is the classic selection method used in steady-state GAs.

The replacement operators in LibGA are: *append*, *by-rank*, *first-weaker*, and *weakest*. The append replacement operator appends new chromosomes to an existing pool. This operator is used in the classical generational GA to place offspring in the new pool. In the by-rank operator the pool is ranked by sorting the fitness values. If the chromosome has a 'high' fitness, it will be placed in the pool, displacing 'weaker' chromosomes. If its fitness is worse than the weakest member of the pool, it dies and is not placed in the pool. Note the weakest and first-weaker operators are somewhere between the append and by-rank operators.

LibGA's crossover operators include *simple*, *uniform*, *order1*, *order2*, *position*, *cycle*, *PMX*, and *asexual*. Simple crossover and uniform crossover are used for traditional bit string encodings of the chromosome. In simple crossover, a random crossover point is selected which divides each parent chromosome into two parts. Alternate parts are contributed by each parent to generate two offspring. This is also known as single point crossover. Uniform crossover selects genes uniformly from either parent to create offspring. The choice of parent is determined randomly for each gene and each parent is equally likely to be selected. Note, however, these crossover operators do not work for order-based problems, since order is not preserved. The other crossover operators preserve order information. Order1, order2, position, cycle, and PMX operators are described in Starkweather *et al.* [35]. The asexual operator is a simple swap of two randomly selected genes, which also is suitable for order-based problems.

LibGA currently offers the following mutation operators: *simple-invert*, *simple-random*, and *swap*. Simple-invert and simple-random are both used with bit string representations. They both randomly select a gene for mutation. The difference is that simple-invert inverts the bit while simple-random selects a random bit value for the gene, which may be the same as the original bit value. Thus, simple-random has an effective mutation rate of half of the mutation rate for simple-invert. Swap mutation can be used for any representation. It simply swaps two randomly selected genes, which is similar to asexual crossover.

We developed LibGA to be straight forward and easy to use. Figure 2 and Figure 3 show `ga-test.c`, one of the files provided with LibGA. This file provides the best illustration of how a GA is developed with LibGA. The goal of this test program is to find a sequence of genes which are in sorted order. That is, it is a GA for sorting.

Figure 2 shows the main function. The file `ga.h` must be included to obtain all of the necessary definitions for the LibGA library routines. A forward definition for the objective function is found here as well. The main program begins with a call to `GA_config()`, which reads the configuration file `ga-test.cfg` and registers the objective function `obj_fun()`. The complete text of `ga-test.cfg` can be found in the Appendix. Memory is allocated for all of the global information required by LibGA and a pointer to this information is returned by `GA_config()` and is assigned to `ga_info`. If a command line argument is specified, it is assumed to be the name of a crossover function to use instead of the one set by `GA_config()`. In this case, the crossover is changed by using the LibGA internal library routine `X_select()`. When the desired configuration is established, `GA_run()` is then called to run the genetic algorithm.

The remainder of Figure 2 has been commented out through the use of the preprocessor directive. This code illustrates the possibility of rerunning the GA with LibGA. Calling `GA_run()` a second time restarts the GA with the same pool and configuration which was present at the end of the previous run. The GA can be reset and a new configuration established by first calling `GA_reset()`, setting any additional parameters, and then calling `GA_run()` again. In Figure 2, the GA is reset and run with a different chromosome length.

Figure 3 shows the objective function. This function gives the lowest fitness value when the values in the chromosome are ordered by nondescending allele value. A penalty is used when genes are encountered which are not in the proper position. At the beginning of the objective function, the penalty is set to 1. A 'fudge factor' is computed as $1/(10l)$ where l is the length of the chromosome. The fudge factor is a small fractional value between 0 and 0.1 which is used to indicate how far off a gene is from the desired position. This helps the GA by giving it better 'resolution' when comparing two chromosomes. The *for* loop in Figure 3 performs a comparison between the allele value and the one it expects to be in each gene position. If there is no match, it computes a positive distance for how far off the allele is from the desired value. This is multiplied by the fudge factor to get a fractional value which is combined with the penalty and added to the fitness. The final fitness of the chromosome then is a real number where the integer part is the number of alleles out of place, and the fractional part is a measure of how far off each allele is from the desired position. Higher fitness values indicate more undesirable chromosomes, thus, the GA's goal should be to minimize the objective function.

Figure 4 shows the output of the test program, `ga-test`. LibGA first prints the configuration used in this run. A value of 1 was used to seed the random number generator. An integer permutation representation was used. The initial population was generated randomly with a pool size of 100 and a chromosome length of 10. The GA's objective was to minimize the objective function and elitism was used. A generational GA was run using roulette selection, position based crossover, no mutation, and append replacement. A short report was generated with output interval set at one generation. Next LibGA prints out statistics during the run. Iteration zero indicates the statistics from the initial pool, and statistics for successive iterations follow. When the GA terminates, a report is made indicating the best chromosome ever encountered as well as its fitness. Note that in generation 6 in Figure 4 the best fitness in the pool becomes zero. This value corresponds to a chromosome which is ordered by nondescending allele value, that is, a 'most desirable' chromosome. Unfortunately, a fitness value of zero spells disaster for a proportional selection scheme like roulette. However, LibGA automatically scales all of the fitness' by one when computing the percentage of total fitness, so roulette can still be used. The fitness values printed by LibGA are the unscaled values. As seen in Figure 4, the GA converged after 10 generations with a final fitness of 0, which for this problem is optimal. This is apparent when examining the final resulting chromosome, which is sorted.

```
/*============================================================================
| (c) Copyright Arthur L. Corcoran, 1992, 1993.  All rights reserved.
|
| Genetic Algorithm Test Program
============================================================================*/
#include "ga.h"

int obj_fun();    /*--- Forward declaration ---*/

/*----------------------------------------------------------------------------
| main()
----------------------------------------------------------------------------*/
main(argc, argv)
   int  argc;
   char *argv[];
{
   GA_Info_Ptr ga_info;

   /*--- Initialize the genetic algorithm ---*/
   ga_info = GA_config("ga-test.cfg", obj_fun);

   /*--- Select crossover ---*/
   if(argc > 1) {
      X_select(ga_info, argv[1]);
   };

   /*--- Run the GA ---*/
   GA_run(ga_info);

#if 0
   /*--- Rerun the GA ---*/
   GA_run(ga_info);

   /*--- Reset and rerun the GA ---*/
   GA_reset(ga_info, "ga-test.cfg");
   ga_info->chrom_len = 15;
   GA_run(ga_info);
#endif
}
```

Figure 2: `ga-test.c` (part 1)

```
/*-----------------------------------------------------------------------
| obj_fun() - user specified objective function
-----------------------------------------------------------------------*/
int obj_fun(chrom)
   Chrom_Ptr chrom;
{
   int i, how_far_off;
   double val = 0.0, penalty, fudge_factor;

   /*--- Penalty for not being in correct position ---*/
   penalty = 1.0;

   /*--- Fudge factor for variance from optimal
         Ensure this is never more than penalty ---*/
   fudge_factor = (1.0 / (double)(chrom->length * 10.0));

   /*--- Fitness is number of genes out of place for sorted order ---*/
   for(i = 0; i < chrom->length; i++) {
      if(chrom->gene[i] != i+1) {

         how_far_off = chrom->gene[i] - (i+1);
         if(how_far_off < 0) how_far_off = -how_far_off;

         val += penalty + how_far_off * fudge_factor;
      }
   }

   chrom->fitness = val;
}
```

Figure 3: `ga-test.c` (part 2)

```
LibGA Version 1.00
(c) Copyright Arthur L. Corcoran, 1992, 1993.  All rights reserved.

GA Configuration Information:
-----------------------------
Basic Info
   Random Seed       : 1
   Data Type         : Integer Permutation
   Init Pool Entered : Randomly
   Chromosome Length : 10
   Pool Size         : 100
   Number of Trials  : Run until convergence
   Minimize          : Yes
   Elitism           : Yes
   Scale Factor      : 0

Functions
   GA          : generational (Gap = 0)
   Selection   : roulette
   Crossover   : position (Rate = 1)
   Replacement : append

Reports
   Type     : Short
   Interval : 1
-----------------------------

Gener    Min      Max      Ave    Variance  Std Dev  Tot Fit    Best
-----  -------  -------  -------  --------  -------  -------  -------
    0     4.16    10.46     9.29      1.44      1.2   928.74     4.16
    1     4.08    10.44     8.21      2.66     1.63   821.04     4.08
    2     4.08    10.44     7.13      3.51     1.87   713.28     4.08
    3     3.04     9.32     6.11      3.16     1.78   611.14     3.04
    4     2.02      9.3     5.05       2.1     1.45   505.38     2.02
    5     2.02     7.26     4.16      1.08     1.04   415.78     2.02
    6        0     5.18     3.47      1.09     1.04   346.86        0
New scale factor = 1
    7        0     5.14     2.63      1.41     1.19   263.18        0
    8        0     5.14     1.27      2.12     1.46   126.66        0
    9        0     3.14    0.205     0.512    0.715    20.52        0
   10        0        0        0         0        0        0        0

The GA has converged after 10 iterations.

Best: 1 2 3 4 5 6 7 8 9 10  (0)
```

Figure 4: `ga-test` output

5 Examples

In this section, we present three simple, related combinatorial optimization problems: bin packing, the traveling salesman problem, and multiprocessor scheduling.

5.1 Bin Packing

The Bin Packing Problem is one of the classic NP-complete problems. Except for trivial cases, it is impossible to optimally solve any of these problems. As a result, researchers have focused on approximation techniques which provide efficient, near optimal solutions. Some of these techniques, which are applicable to bin packing and related problems, include heuristic techniques, simulated annealing, neural networks, tabu search and genetic algorithms. Papadimitriou and Steiglitz [30] and Parker and Rardin [31] present several classical techniques for solving the bin packing problem.

In the classic bin packing problem, a finite collection of packages is packed into a set of bins. The packages and bins are characterized by their weights and capacities, respectively. The problem can be stated either as a decision problem or as an optimization problem. In the decision problem it is necessary to determine whether or not there is a disjoint partitioning of the set of packages such that each partition fits into a bin. Thus, given an integer number of bins, determine if all of the packages fit into the bins. On the other hand, the optimization problem attempts to minimize the number of bins required, or equivalently, to minimize the amount of wasted bin capacity in the packing.

The bin packing problem is a generalization of the Partition Problem. The partition problem is stated as follows: Let p_1, p_2, ..., p_n be a set of real numbers each between 0 and 1; the goal is to partition the numbers into as few subsets as possible such that the sum of numbers in each subset is at most 1.

In the bin packing problem, the bin represents a partition size and the packages must be optimally placed in these partitions. In most cases, the standard approximation algorithms (first fit, next fit, best fit, etc.) are nearly optimal. However, there are worst case examples which are far from optimal. Some algorithms like the Modified First Fit Decreasing algorithm have tried to improve absolute bounds by special treatment of these worst cases. See Hu [23] for more details on bin packing.

The bin packing problem is applicable in a variety of situations. In computer systems it is used in allocation problems, such as allocating core memory to programs, or space on a disk or tape. The two dimensional bin packing is equivalent to solving the problem of multiprocessor scheduling with time and memory constraints. The packages represent the time and memory requirements of tasks, and the bins represent processors. The knapsack problem is another closely related problem. In other disciplines, bin packing can be used in such problems as packing trucks, allocating commercials to station-breaks on television, and cutting pipe from standardized lengths. The two dimensional problem can be used for stock cutting, where the packages are patterns which must be cut from a fixed width roll of material (the bin). Other classical problems related to bin packing include Job Scheduling, Network Routing, various other layout problems, and the vehicle routing problem (VRP) [9].

The vehicle routing problem (VRP) involves determining minimum cost vehicle routes for a fleet of vehicles originating and terminating from a central location. The fleet of vehicles service a set of customers with a known set of constraints. All customers must be assigned to vehicles such that each customer is serviced exactly once and each vehicle cannot exceed its capacity. The vehicle routing problem has been studied extensively. Bodin *et al.* [5] provides a comprehensive survey of VRP and several variations.

The vehicle routing problem with time windows (VRPTW) adds the additional constraint to the VRP where each customer provides a time window for servicing. Hence, in the presence of time windows, optimization of routing and scheduling of vehicles involves not only total distance traveled, but time costs for waiting when a vehicle has arrived too early to service a customer. Time windows arise frequently in business applications. Examples include bank deliveries, postal deliveries, school bus routing, industrial refuse collection, over night delivery services, passenger and freight operations such as airline, railway and bus routing and scheduling. Time windows also arise in most retail distribution systems. This is an extremely practical problem; efficient routing and scheduling can save industry and government millions of dollars each year. Solomon [33, 34] provides an excellent survey of the vehicle routing problem with time windows. The VRPTW is a variation of the bin packing problem, where each vehicle corresponds to a 'bin' and each customer to be serviced corresponds to a 'package'. The need for multiple vehicles to satisfy the requirements corresponds to the need for multiple 'bins'. The goal is the same: to minimize the number of vehicles (bins) required, while meeting all of the constraints. Blanton and Wainwright have developed special GA crossover operators called MX1 and MX2 specifically for the VRPTW [4].

The classic bin packing problem is expressed using one dimensional packages. This approach blindly generates partitions so that the sum of the one dimensional package parameters in each partition does not exceed the bin capacity. This parameter is typically stated as the package weight. For every package attribute there is a corresponding

capacity or maximum value for the bin. Generally, there can be no single package with an attribute exceeding the maximum value set by the bin. Otherwise, the package could never be packed. Interesting cases occur when the parameters differ in relation to one another. For example, when the bin is rectangular and the packages are all square.

In the classic problem, the bin has a fixed capacity. For multiple dimensions, the analogous bin would have fixed capacity in every dimension. Thus, a two dimensional bin would define a rectangle, and a three dimensional bin would define a 'closed box'. A common variation of the classic problem is to use a single, open-ended bin. Used primarily for two or more dimensions, the problem is to minimize the value of the open dimension subject to all other constraints. When using the level technique, this method can be transformed to the classical problem. This is done by packing each level of the single bin into the multiple closed bins, as if each level were a single package. A less common variation uses multiple, dissimilar bins. The distribution of the bins could be like the distribution of package sizes. They could be random, uniform, skewed, etc. It is analogous to packing a fleet of trucks of different sizes and capacities.

Many near-optimal heuristic techniques have been developed for the one-dimensional bin packing problem. None of these techniques guarantee an optimal packing. However, many techniques approximate the optimal packing within a constant bound. The three best-known approximation algorithms for bin packing are Next Fit, First Fit, and Best Fit. There are many other algorithms, however, most are variations or refinements of these basic methods and only offer modest improvements in packing efficiency. For more details, see Floyd and Karp [17], Garey and Johnson [19] and Johnson *et al.* [25].

Next Fit heuristic is described as follows: beginning with a single bin, the packages are taken from the list in order, and placed in the next available position. If there is not enough room to pack the current package, the bin is considered 'closed' and a new bin is started. Packages are considered in the order they appear in the list, and once a bin is 'closed' no additional packages can be placed into it. Waste can obviously occur since a new bin is started even when a package may fit into a 'closed' bin. This is a linear algorithm in both time and space. Furthermore, it has been shown that the Next Fit algorithm generates packings no worse than twice optimal. The First Fit algorithm places each package in the first bin in which it will fit. A new bin is added only when all of the previous bins have been examined and no space can be found for a package. In this algorithm there are no 'closed' bins. This algorithm can use quadratic time in the worst case and is $O(n \log n)$ on average for n packages. However, it has been shown the packings produced are no worse than 1.7 times optimal.

The Best Fit algorithm places each package in the 'best' bin in which it will fit. The best bin is the one with the least amount of space left over when the package is added. Surprisingly, this algorithm's asymptotic performance and packing efficiency is identical to that for First Fit. Minor differences in packing efficiency are related to package distribution. That is, for package sizes larger than 1/6 of the bin size (distributed uniformly in the range [1/6..1]), Best Fit is more efficient than First Fit. For package sizes larger than 1/5 of the bin size (distributed uniformly in the range [1/5..1]) the packing efficiencies are identical. Sorting the packages before applying these methods can lead to improved results. For example, sorting by decreasing package size before applying First Fit results in packings which are no worse than 11/9 times optimal, a 28% improvement. This variation is called First Fit Decreasing. Similar improvements can be found in Next Fit Decreasing and Best Fit Decreasing [9].

Expanding the problem to two dimensions (rectangle packing) demands different techniques. The first technique uses a 'bottom up - left justified' packing rule, or simply 'bottom-left'. Each package is packed as close to the bottom of the bin and as far to the left of the bin as it can go. This differs from the one dimensional cases where there exists a permutation of the packages for which the methods generate an optimal packing. There are instances where the best bottom-left method produces packings which are 5/4 times optimal. That is, no matter how the packages are ordered, the optimal packing can not be found. The best absolute packing bounds are obtained by sorting the packages by decreasing width. Classical two dimensional packing generally requires the packing to be orthogonal and disallows rotation of the packages. However, some applications may allow rotation or translation of the packages. Packing efficiency may be improved if each package is rotated so that its width exceeds its height, or vice versa. Several theoretical and practical results are presented in Baker *et al.* [1], Carpenter and Dowsland [7], Coffman *et al.* [8], Dowsland [14] and Leung *et al.* [27].

When extending the problem to three dimensions, it is desirable to apply the results of two dimensional research to obtain similar efficiency. Ideally, the packages would be presorted, then placed level by level, using a two dimensional method to pack each level. Unfortunately, it is difficult to extend the packing efficiency in this way. For example, sorting by decreasing height does not guarantee decreasing width or length. Thus, the two dimensional packing may be inefficient. On the other hand, ordering the packages to make the two dimensional packing efficient may cause wasted space to appear in the height. Unlike the purely two dimensional problem, the two dimensional packing stage must deal with a boundary on the second dimension (the length). Clearly, three dimensional packing is a very practical problem, yet proves to be a very difficult problem to solve [9].

In this paper, we will concentrate on the simple example of solving the one-dimensional bin packing problem using LibGA. To apply genetic algorithms to the one dimensional packing problem, one must define the encoding of the chromosome, the evaluation function, and the recombination operator. The most natural encoding is to use a string of integers which form an index into the set of packages. The first package is denoted '1', the second '2', and so on. A random reordering of the string represents a random permutation of the packages. The evaluation function returns the number of bins obtained by applying a one dimensional next fit packing algorithm.

The recombination operator must produce a permutation of the packages using partial orderings contained in the two parents. The resulting chromosome must include all of the packages with no duplicates. Fortunately, there are several general purpose crossover functions which meet the requirement. The possible crossover functions that can be used for order based problems include Order1, Order2, Cycle, Position, and Partially Mapped Crossover (PMX). These are described by Whitley and Starkweather [40]. In addition, we have developed an asexual crossover operator which simply exchanges two packages in the list. This is precisely what is done by swap mutation.

Figure 5, Figure 6, and Figure 7 show `gabp.c`, a genetic algorithm for one dimensional bin packing. This program is adapted from `ga-test.c`. Figure 5 shows the main program. The GA is configured with `GA_config()`, using the configuration file `gabp.cfg` and registering the objective function `next_fit()`. The configuration file is nearly identical to `ga-test.cfg` found in the Appendix. After the GA has been configured, the packages are read from the data file specified by the `user_data` directive in `gabp.cfg`. The chromosome length is set to the number of packages read. Finally, `GA_run()` is called to run the GA.

Figure 6 shows the objective function, `next_fit()`. The chromosome in this problem is a permutation of the list of packages read. The objective function packs these packages using a simple next fit heuristic and returns the total number of bins required in the packing for the chromosome's fitness. In the trivial case of no packages the objective function returns a fitness of zero. Otherwise, it initializes a single empty bin. As seen in the *for* loop in Figure 6, the weight of the next package to add to the bin is determined by the index provided by each gene in the chromosome. The weight is a fractional value between 0 and 1 which represents a value normalized with respect to the bin capacity. If the package fits into the bin it is placed there. Otherwise, a new bin is initialized and the package is placed there. The final fitness is the number of bins and the GA's goal is to minimize this value.

Figure 7 shows the code used to read the data file. The data file consists of a single line indicating the number of packages, n, followed by n lines which are the normalized package weights. As the packages are read, their weights are summed to give an indication of what the optimal might be.

Figure 8 shows the output from running `gabp` on a 50 package data set. The configuration is the same as for `ga-test` with the exception of user data `r50.bp`, asexual crossover, and report interval 10. The data file used, `r50.bp`, was a randomly generated set of 50 packages. From Figure 8 we see the GA begins with a pool of solutions with fitnesses in the range from 31 bins to 38 bins. The average is 34.5 bins. After 62 generations, the GA has converged to a solution with fitness value of 28 bins. The sum of package weights is nearly 26, indicating an optimal packing can use no less than 26 bins. Thus, the GA has found a solution that is within 2 bins of optimal in this case. Since the true optimal is unknown and impossible to calculate in a reasonable time, and the random packages may not necessarily fit in 26 bins, it is quite possible that the GA has found the optimal. In any case, the GA has found a very good solution. Additional details about genetic algorithms for bin packing can be found in Smith [32], and in our previous work [9, 10, 11].

```
/*==========================================================================
| (c) Copyright Arthur L. Corcoran, 1993.  All rights reserved.
|
| Genetic Algorithm For One Dimensional Bin Packing
==========================================================================*/
#include "ga.h"

int next_fit();      /* Objective function */

#define MAXPKGS 100  /* Maximum number of packages */

float Pkgs[MAXPKGS], /* Packages */
      Sum_Pkgs;      /* Sum of all package weights */
int   Num_Pkgs;      /* Actual number of packages */

/*--------------------------------------------------------------------------
| Entry point
--------------------------------------------------------------------------*/
main(argc, argv)
   int  argc;
   char *argv[];
{
   GA_Info_Ptr ga_info;

   /*--- Initialize the genetic algorithm ---*/
   ga_info = GA_config("gabp.cfg", next_fit);

   /*--- Read packages from data file ---*/
   read_packages(ga_info->user_data);

   /*--- Set chromosome length to number of packages ---*/
   ga_info->chrom_len = Num_Pkgs;

   /*--- Run the GA ---*/
   GA_run(ga_info);

   /*--- This gives us some idea of optimal ---*/
   printf("Sum of package weights = %f\n\n", Sum_Pkgs);
}
```

Figure 5: `gabp.c` (part 1)

```
/*-------------------------------------------------------------------------
| Use simple next fit heuristic for objective function
-------------------------------------------------------------------------*/
int next_fit(chrom)
   Chrom_Ptr chrom;
{
   int i, num_bins;
   float pkg_weight, weight;

   /*--- Trivial case: no packages ---*/
   if(chrom->length < 1) {
      chrom->fitness = 0;
      return;
   }

   /*--- Initialize ---*/
   num_bins = 1;  /* First bin */
   weight   = 0;  /* Its empty */

   /*--- Place each package using next fit ---*/
   for(i = 0; i < chrom->length; i++) {

      pkg_weight = Pkgs[(int)chrom->gene[i]-1];

      if(weight + pkg_weight > 1.0) {  /* Oops, too big */
         weight = pkg_weight;
         num_bins++;
      } else {                         /* Ahh, it fits */
         weight += pkg_weight;
      }
   }

   /*--- Goal is to minimize the number of bins ---*/
   chrom->fitness = num_bins;
}
```

Figure 6: `gabp.c` (part 2)

```
/*-------------------------------------------------------------------------
| Read packages from data file
-------------------------------------------------------------------------*/
read_packages(filename)
   char *filename;
{
   FILE *fid;
   int i;

   /*--- Open data file ---*/
   if((fid = fopen(filename,"r")) == NULL) {
      printf("Error opening package data file <%s>\n", filename);
      exit(1);
   }

   /*--- Get number of packages ---*/
   fscanf(fid,"%d", &Num_Pkgs);
   if(Num_Pkgs < 1 || Num_Pkgs > MAXPKGS) {
      printf("Number of packages, %d, out of bounds [1..%d]\n",
         Num_Pkgs, MAXPKGS);
      exit(1);
   }

   /*--- Get package weights and sum them ---*/
   Sum_Pkgs = 0;
   for(i=0; i < Num_Pkgs; i++) {
      fscanf(fid,"%f", &Pkgs[i]);
      Sum_Pkgs += Pkgs[i];
   }

   /*--- Close data file ---*/
   fclose(fid);
}
```

Figure 7: `gabp.c` (part 3)

```
LibGA Version 1.00
(c) Copyright Arthur L. Corcoran, 1992, 1993.  All rights reserved.

GA Configuration Information:
-----------------------------
Basic Info
   User Data         : r50.bp
   Random Seed       : 1
   Data Type         : Integer Permutation
   Init Pool Entered : Randomly
   Chromosome Length : 50
   Pool Size         : 500
   Number of Trials  : Run until convergence
   Minimize          : Yes
   Elitism           : Yes
   Scale Factor      : 0

Functions
   GA          : generational (Gap = 0)
   Selection   : roulette
   Crossover   : asexual (Rate = 1)
   Replacement : append

Reports
   Type     : Short
   Interval : 10
-----------------------------

Gener    Min      Max      Ave    Variance  Std Dev  Tot Fit    Best
-----  -------  -------  -------  --------  -------  -------  -------
    0       31       38     34.5      1.46     1.21    17256       31
    1       31       37     33.9     0.938    0.968    16968       31
   10       30       32     30.9     0.226    0.476    15455       30
   20       29       31     29.9       0.1    0.317    14956       29
   30       28       30     29.1     0.193    0.439    14573       28
   40       28       29     28.7     0.214    0.462    14346       28
   50       28       29     28.1     0.104    0.323    14059       28
   60       28       29       28     0.002   0.0447    14001       28
   62       28       28       28         0        0    14000       28

The GA has converged after 62 iterations.

Best: 36 48 41 10 49 43 37 39 25 26 1 11 22 33 5 2 40 24 6 29
      21 47 23 35 14 50 19 15 45 12 4 20 8 31 42 32 44 7 30 3
      34 46 27 13 28 16 18 9 17 38  (28)

Sum of package weights = 25.928408
```

Figure 8: `gabp.c` output

5.2 Traveling Salesman Problem

Another well known combinatorial optimization problems is the Traveling Salesman Problem (TSP). Given a set of n points in a plane corresponding to the location of n cities, find the minimum distance closed path that visits each city exactly once. This is called the traveling salesman problem. The traveling salesman problem belongs to a class of minimization problems for which the objective function has many local minima. The objective function is simply the total length of the tour. The traveling salesman route can be thought of as a circular arrangement of n cities, or as a permutation of a list of n cities. Solving this problem requires $O(n!)$ computation time since the number of possible tours for n cities is $(n-1)!$. In this paper it is assumed each city is directly connected to every other city by Euclidian distance. That is, distances satisfy the triangle inequality, which means that the direct route between any two cities is never more than an indirect route between two cities. This assumption helps in the design of approximation algorithms for this problem.

We chose the traveling salesman problem to demonstrate LibGA because it is a representative problem for a wide variety of combinatorial optimization problems where the solution space is all permutations of n objects. Other combinatorial optimization problems that fall into this category include the bin packing problem, job scheduling problems, stock cutting, vehicle routing and transportation scheduling problems, etc. Developing efficient genetic algorithms to solve this problem will have direct applications for solving a host of other practical combinatorial optimization problems [3, 37, 38, 41].

Figure 9, Figure 10, and Figure 11 show `gatsp.c`, a genetic algorithm for the traveling salesman problem. This program is adapted from `gabp.c` and `ga-test.c`. The main program is shown in Figure 9. As in `gabp`, the GA is configured with `GA_config()`, the city information is read from the data file specified in `gatsp.cfg`, and the chromosome length is set to the number of cities read. Finally, the GA is run by calling `GA_run()`.

The objective function, `eval_tour()`, is shown in Figure 10. In the trivial case of no cities, the fitness returned is zero. Otherwise, the Euclidean distance between each successive city in the list is added to the fitness. In the *for* loop in Figure 10, each gene is an index into the original list of cities. After the loop, the distance from the last city indexed by the chromosome to the first city indexed by the chromosome is added to the fitness. Thus, the fitness is the total cost of the tour. The GA must minimize this total cost.

Figure 11 shows the routine which reads the data file. This file is much like the one used in the bin packing problem. The first line indicates the number of cities, and the remaining lines give the x and y coordinates for each city. These coordinates are assumed to be normalized so they fall in the range from 0 to 1.

Figure 12 shows the output of `gatsp`. The parameters are the same as before in Figure 4 and Figure 8 except for user data `r50.tsp` and report interval 100. The data file `r50.tsp` is a randomly generated set of 50 cities. As Figure 12 shows, the initial pool contained solutions whose fitness' ranged from about 21 to about 30. After 1309 generations, the GA converged to a fitness of about 7.

Figure 13 shows an example random tour for the data. Figure 14 shows the final tour found by the GA. We see again that the GA has been able to find a very good solution.

```
/*============================================================================
| (c) Copyright Arthur L. Corcoran, 1993.  All rights reserved.
|
| Genetic Algorithm For The Travelling Salesman Problem
============================================================================*/
#include "ga.h"

int eval_tour();      /* Objective function */

#define MAXCITIES 100  /* Maximum number of cities */

struct {
   float x, y;
} City[MAXCITIES];     /* Cities */
int   Num_Cities;      /* Actual number of cities */

/*----------------------------------------------------------------------------
| Entry point
----------------------------------------------------------------------------*/
main(argc, argv)
   int  argc;
   char *argv[];
{
   GA_Info_Ptr ga_info;

   /*--- Initialize the genetic algorithm ---*/
   ga_info = GA_config("gatsp.cfg", eval_tour);

   /*--- Read cities from data file ---*/
   read_cities(ga_info->user_data);

   /*--- Set chromosome length to number of cities ---*/
   ga_info->chrom_len = Num_Cities;

   /*--- Run the GA ---*/
   GA_run(ga_info);
}
```

Figure 9: `gatsp.c` (part 1)

```
/*-------------------------------------------------------------------------
| Objective function evaluates tour cost based on Euclidean distance
-------------------------------------------------------------------------*/
int eval_tour(chrom)
   Chrom_Ptr chrom;
{
   int i, idx1, idx2;
   float dx, dy, cost;

   /*--- Trivial case: no cities ---*/
   if(chrom->length < 1) {
      chrom->fitness = 0;
      return;
   }

   /*--- Initialize ---*/
   cost = 0;

   /*--- Add Euclidean distance from each city to next city ---*/
   for(i = 0; i < chrom->length - 1; i++) {
      idx1  = (int)chrom->gene[i  ] - 1;
      idx2  = (int)chrom->gene[i+1] - 1;
      dx    = City[idx1].x - City[idx2].x;
      dy    = City[idx1].y - City[idx2].y;
      cost += sqrt(dx * dx + dy * dy);
   }

   /*--- Add cost from last city to first ---*/
   idx1  = (int)chrom->gene[chrom->length-1] - 1;
   idx2  = (int)chrom->gene[0              ] - 1;
   dx    = City[idx1].x - City[idx2].x;
   dy    = City[idx1].y - City[idx2].y;
   cost += sqrt(dx * dx + dy * dy);

   /*--- Goal is to minimize the tour cost ---*/
   chrom->fitness = cost;
}
```

Figure 10: `gatsp.c` (part 2)

```
/*-------------------------------------------------------------------------
| Read cities from data file
-------------------------------------------------------------------------*/
read_cities(filename)
   char *filename;
{
   FILE *fid;
   int i;

   /*--- Open data file ---*/
   if((fid = fopen(filename,"r")) == NULL) {
      printf("Error opening city data file <%s>\n", filename);
      exit(1);
   }

   /*--- Get number of cities ---*/
   fscanf(fid,"%d", &Num_Cities);
   if(Num_Cities < 1 || Num_Cities > MAXCITIES) {
      printf("Number of cities, %d, out of bounds [1..%d]\n",
         Num_Cities, MAXCITIES);
      exit(1);
   }

   /*--- Get city coordinates ---*/
   for(i=0; i < Num_Cities; i++) {
      fscanf(fid,"%f %f", &City[i].x, &City[i].y);
   }

   /*--- Close data file ---*/
   fclose(fid);
}
```

Figure 11: `gatsp.c` (part 3)

```
LibGA Version 1.00
(c) Copyright Arthur L. Corcoran, 1992, 1993.  All rights reserved.

GA Configuration Information:
-----------------------------
Basic Info
   User Data         : r50.tsp
   Random Seed       : 1
   Data Type         : Integer Permutation
   Init Pool Entered : Randomly
   Chromosome Length : 50
   Pool Size         : 500
   Number of Trials  : Run until convergence
   Minimize          : Yes
   Elitism           : Yes
   Scale Factor      : 0

Functions
   GA          : generational (Gap = 0)
   Selection   : roulette
   Crossover   : asexual (Rate = 1)
   Replacement : append

Reports
   Type     : Short
   Interval : 100
-----------------------------

Gener    Min      Max      Ave    Variance  Std Dev  Tot Fit    Best
-----  -------  -------  -------  --------  -------  -------  -------
    0  21.0887  30.2088     26.7      2.24      1.5  13374.2  21.0887
    1  21.0887  29.5356     25.7      1.77     1.33  12873.9  21.0887
  100  10.9397  12.3754     11.6    0.0843     0.29  5819.41  10.9397
  200   9.4759  10.1167     9.82    0.0231    0.152  4907.61   9.4759
  300  8.89632   9.5162     9.27     0.018    0.134  4633.49  8.89632
  400  8.38697  8.93954     8.65   0.00876   0.0936  4324.46  8.38697
  500  8.15736  8.48956      8.3   0.00555   0.0745  4149.75  8.15736
  600  8.05966  8.25668     8.12    0.0031   0.0557  4061.96  8.05966
  700  7.96856  8.15455     8.05   0.00178   0.0422  4025.26  7.96856
  800   7.4698   7.9195      7.6   0.00759   0.0871  3799.99   7.4698
  900  7.40255  7.66445     7.51   0.00219   0.0468  3753.72  7.40255
 1000  7.34219  7.48982     7.42   0.00236   0.0486  3707.87  7.34219
 1100  7.34219  7.42519     7.35  0.000118   0.0109  3674.05  7.34219
 1200  7.34219   7.3614     7.35  4.85E-05  0.00696  3672.66  7.34219
 1300  7.34219   7.3614     7.34  3.66E-06  0.00191  3671.19  7.34219
 1309  7.34219  7.34219     7.34         0        0   3671.1  7.34219

The GA has converged after 1309 iterations.

Best: 7 30 41 33 45 15 27 42 16 24 38 9 22 37 5 13 17 12 35 49
      28 14 47 40 43 31 8 4 25 46 26 29 10 34 23 19 36 48 20 1
      6 32 50 18 39 21 11 44 2 3  (7.34219)
```

Figure 12: `gatsp` output

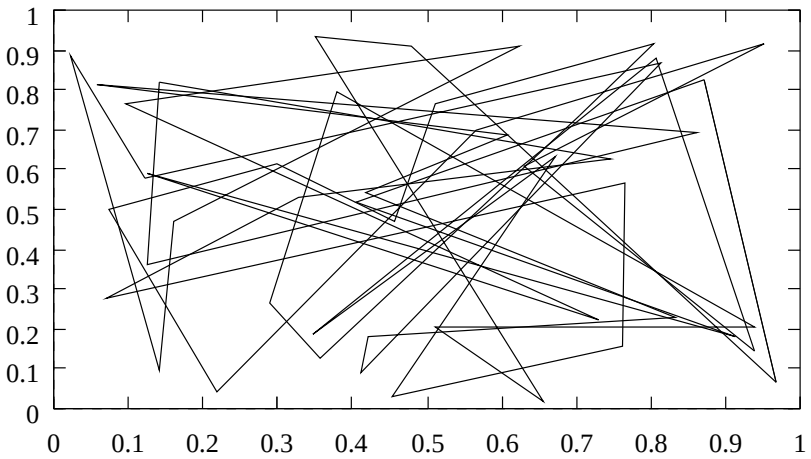

Figure 13: Random TSP Tour

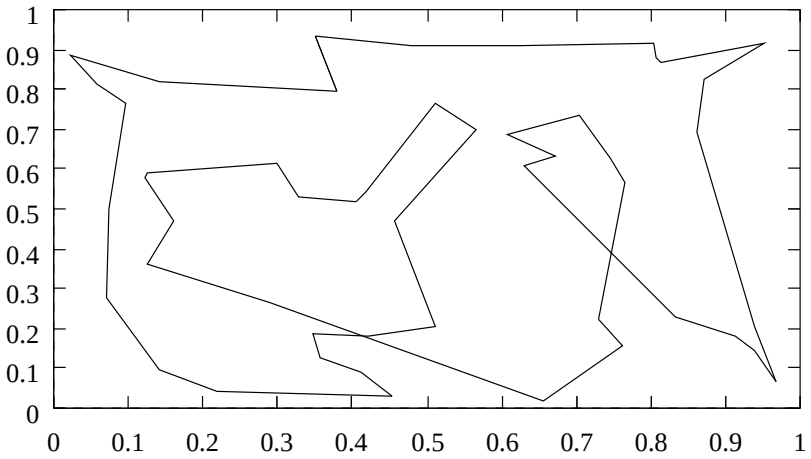

Figure 14: GA's TSP Tour

5.3 Multiprocessor Scheduling

The Multiprocessor Scheduling Problem is defined as follows: a set of n jobs is to be scheduled on a set of m identical processors. Each job J is specified as $J = (t, c)$, where c is the capacity (memory) requirement of the job and t is its running time. Note that only nonpreemptive job scheduling is considered. That is, once a job is started it remains in the processor until it is finished. The objective is to determine a schedule of jobs on the machines so as to minimize the total processing time. For example, consider the case where $m = 3$, each processor has memory capacity = 5, and the 14 jobs to be scheduled are:

Job	1	2	3	4	5	6	7
	(2,2)	(1,3)	(1,3)	(3,2)	(4,1)	(3,1)	(3,4)
Job	8	9	10	11	12	13	14
	(2,3)	(1,4)	(2,2)	(1,2)	(2,2)	(2,1)	(3,1)

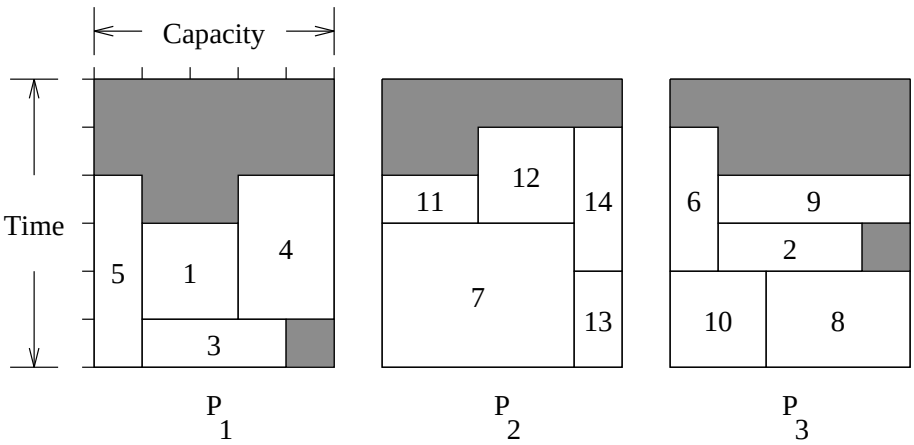

Figure 15: An Example Schedule

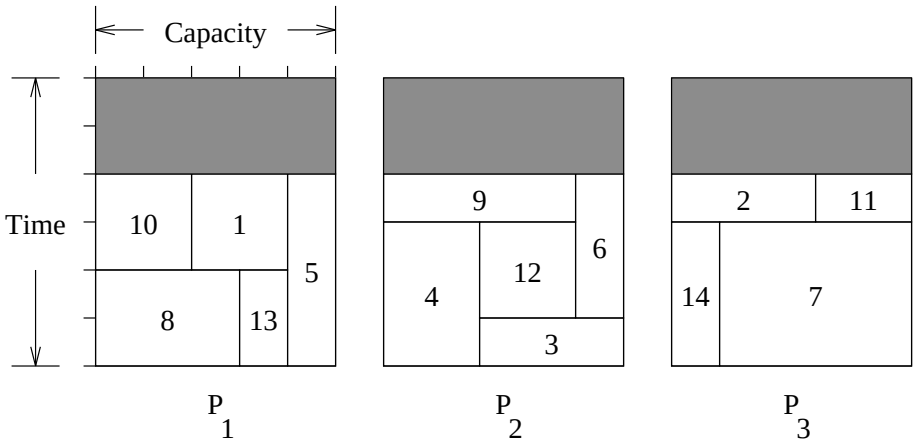

Figure 16: An Optimal Schedule

An example schedule is shown in Figure 15 requiring 5 time units. An optimal schedule requiring 4 time units is shown in Figure 16. Note the Job Shop Scheduling problem is a variation of the multiprocessor scheduling problem. In the job shop scheduling problem each job, J_i requires the completion of several tasks, $T_{j,1}$, $T_{j,2}$, ..., $T_{j,n}$. The tasks for any job J_i are to be carried out in the order 1, 2, 3, ..., etc., where each task j cannot begin until task $j-1$ ($j > 1$) has been completed [22]. In the job shop scheduling problem, the processor capacity is not considered. It is assumed every task in every job uses the entire capacity of a given processor.

The multiprocessor scheduling problem is also similar to the two dimensional bin packing problem, where each processor is a bin, and each job is a package. The multiprocessor scheduling problem and its variations of other scheduling problems are economically very important problems, especially in industrial applications.

For the interested reader, Yamada and Nakano [42] present a GA implementation for large-scale job shop problems. Davidor *et al.* [13] investigated GAs as a technique for solving the job shop scheduling problem. Kidwell [26] developed a GA to schedule distributed tasks on a bus-based system. Li and Cheng [28] developed a job shop scheduling algorithm to partition a mesh connected system where jobs require meshes an the system itself is a square mesh of size a power of two. For other related work see [6, 15, 16, 24, 36, 41]. The LibGA implementation for the multiprocessor scheduling problem is shown below. This is an extremely simplified version of the one the authors studied in more detail in [12].

Figure 17, Figure 18, and Figure 19 show the GA for multiprocessor scheduling, `gams.c`. As before, the GA is configured with `GA_config()`, the set of tasks is read from the data file, and the chromosome length is set to the number of tasks. The objective function in Figure 18 evaluates the total time required for the task list ordering indicated by the chromosome. In the trivial case of an empty task list, the fitness is zero. Since this problem is equivalent to a two dimensional bin packing problem, each task is otherwise placed using a two dimensional, level oriented next fit heuristic. Note, the 'width of the bin' in this case corresponds to the memory capacity of the processors. The goal of the GA is to minimize the total time. Figure 19 shows the routine to read the initial task list, which is straight forward.

Figure 20 shows the output of `gams`. The configuration is the same as before except the user data is `r50.ms` and the report interval is 30. The GA begins with an initial pool of chromosomes which represent solutions whose total time is in the range from about 16 to about 21 time units. After 297 generations, the GA has obtained a solution of about 14 time units, which is quite close to the optimal's lower bound of about 12 time units.

```
/*==========================================================================
| (c) Copyright Arthur L. Corcoran, 1993.  All rights reserved.
|
| Genetic Algorithm For Multiprocessor Scheduling
==========================================================================*/
#include "ga.h"

int eval_tasks();      /* Objective function */

#define MAXTASKS 100   /* Maximum number of tasks */

struct {
   float time,       /* Time requirement */
         mem;        /* Memory requirement */
} Task[MAXTASKS];    /* Tasks */
int   Num_Tasks;     /* Actual number of tasks */

double Sum_Area;     /* Sum of task "area" (time * memory) */

/*--------------------------------------------------------------------------
| Entry point
--------------------------------------------------------------------------*/
main(argc, argv)
   int  argc;
   char *argv[];
{
   GA_Info_Ptr ga_info;

   /*--- Initialize the genetic algorithm ---*/
   ga_info = GA_config("gams.cfg", eval_tasks);

   /*--- Read tasks from data file ---*/
   read_tasks(ga_info->user_data);

   /*--- Set chromosome length to number of tasks ---*/
   ga_info->chrom_len = Num_Tasks;

   /*--- Run the GA ---*/
   GA_run(ga_info);

   /*--- This gives us some idea of optimal ---*/
   printf("Total task area = %G\n", Sum_Area);
}
```

Figure 17: `gams.c` (part 1)

```
/*-------------------------------------------------------------------------
| Objective function evaluates task list using next fit placement
-------------------------------------------------------------------------*/
int eval_tasks(chrom)
   Chrom_Ptr chrom;
{
   int i;
   float tot_time, max_time, tot_mem, task_time, task_mem;

   /*--- Trivial case: no tasks ---*/
   if(chrom->length < 1) {
      chrom->fitness = 0;
      return;
   }

   /*--- Initialize ---*/
   tot_time = 0;
   max_time = 0;
   tot_mem  = 0;

   /*--- Place each task using next fit ---*/
   for(i = 0; i < chrom->length; i++) {

      task_time = Task[(int)chrom->gene[i]-1].time;
      task_mem  = Task[(int)chrom->gene[i]-1].mem;

      /*--- Place task on a "level" ---*/
      if(tot_mem + task_mem > 1.0) {  /* Oops, too much memory */
         tot_mem   = task_mem;
         tot_time += max_time;
         max_time  = 0;
      } else {                        /* Ahh, it fits */
         tot_mem += task_mem;
      }

     /*--- Find longest task time on a "level" ---*/
     if(task_time > max_time) max_time = task_time;
   }

   /*--- Goal is to minimize the total time ---*/
   chrom->fitness = tot_time;
}
```

Figure 18: `gams.c` (part 2)

```
/*-------------------------------------------------------------------------
| Read tasks from data file
-------------------------------------------------------------------------*/
read_tasks(filename)
   char *filename;
{
   FILE *fid;
   int i;

   /*--- Open data file ---*/
   if((fid = fopen(filename,"r")) == NULL) {
      printf("Error opening task data file <%s>\n", filename);
      exit(1);
   }

   /*--- Get number of tasks ---*/
   fscanf(fid,"%d", &Num_Tasks);
   if(Num_Tasks < 1 || Num_Tasks > MAXTASKS) {
      printf("Number of tasks, %d, out of bounds [1..%d]\n",
         Num_Tasks, MAXTASKS);
      exit(1);
   }

   /*--- Get task time and memory requirements ---*/
   Sum_Area = 0;
   for(i=0; i < Num_Tasks; i++) {
      fscanf(fid,"%f %f", &Task[i].time, &Task[i].mem);
      Sum_Area += Task[i].time * Task[i].mem;
   }

   /*--- Close data file ---*/
   fclose(fid);
}
```

Figure 19: gams.c (part 3)

```
LibGA Version 1.00
(c) Copyright Arthur L. Corcoran, 1992, 1993.  All rights reserved.

GA Configuration Information:
-----------------------------
Basic Info
   User Data         : r50.ms
   Random Seed       : 1
   Data Type         : Integer Permutation
   Init Pool Entered : Randomly
   Chromosome Length : 50
   Pool Size         : 500
   Number of Trials  : Run until convergence
   Minimize          : Yes
   Elitism           : Yes
   Scale Factor      : 0

Functions
   GA          : generational (Gap = 0)
   Selection   : roulette
   Crossover   : asexual (Rate = 1)
   Replacement : append

Reports
   Type     : Short
   Interval : 30
-----------------------------

Gener    Min      Max      Ave    Variance  Std Dev  Tot Fit    Best
-----  -------  -------  -------  --------  -------  -------  -------
    0  16.1001  20.7399     18.6     0.566    0.752  9311.39  16.1001
    1  15.7861  20.0496     18.1     0.344    0.587   9074.3  15.7861
   30  14.2964   15.199     14.8    0.0236    0.154  7412.93  14.2964
   60  13.9174  14.2645     14.1    0.0035   0.0592  7037.35  13.9174
   90  13.8126  14.0378     13.9   0.00139   0.0372  6958.99  13.8126
  120  13.7852  13.8808     13.8  0.000255    0.016  6903.08  13.7852
  150  13.6964  13.7908     13.7    0.0015   0.0388  6862.18  13.6964
  180  13.6655  13.6964     13.7  0.000219   0.0148  6841.08  13.6655
  210  13.6655  13.6964     13.7  2.36E-05  0.00486  6833.32  13.6655
  240  13.6655  13.6655     13.7  1.37E-12  1.17E-06  6832.73  13.6655
  270  13.6655  13.6655     13.7  6.71E-13  8.19E-07  6832.73  13.6655
  297  13.6655  13.6655     13.7         0         0  6832.73  13.6655

The GA has converged after 297 iterations.

Best: 28 43 42 12 24 31 14 41 33 40 49 13 30 29 36 38 10 46 45 27
      15 2 7 11 5 34 26 20 6 21 8 35 22 3 39 9 19 23 4 50
      25 48 32 47 44 17 16 37 18 1  (13.6655)

Total task area = 12.4629
```

Figure 20: gams output

6 Conclusions

We have given several examples to show that genetic algorithms are ideally suited for solving combinatorial optimization problems. We used LibGA to implement three such problems: bin packing, the traveling salesman problem, and multiprocessor scheduling. The use of LibGA allows the problems to be coded with a minimal knowledge of genetic algorithms. Parameters can be easily changed by simply editing a configuration file. One of the objectives of this paper is to show how easy LibGA is to use, and provide enough example code to allow the reader the ability to easily begin to use LibGA.

Acknowledgements

This research has been partially supported by OCAST Grant AR2-004. The authors also wish to acknowledge the support of Sun Microsystems, Inc. We also extend heart felt thanks to those who have taken the time to use LibGA and have sent us comments, questions, and suggestions.

LibGA Availability

LibGA is available at no cost by sending an email request to the authors. The authors respective email addresses are *corcoran@penguin.mcs.utulsa.edu*, and *rogerw@penguin.mcs.utulsa.edu*. Conventional mail should be addressed to the authors at:

Department of Mathematical and Computer Science
University of Tulsa
600 South College Avenue
Tulsa, OK 74104-3189
USA

LibGA can also be obtained via anonymous ftp from *ftp.aic.nrl.navy.mil* as */pub/galist/src/ga/libga100.tar.Z*.

References

[1] B. S. Baker, E. G. Coffman, and R. L. Rivest. Orthogonal packings in two dimensions. *SIAM Journal of Computing*, 9(4):846–855, Nov. 1980.

[2] R. K. Belew and L. B. Booker, editors. *Proceedings of the Fourth International Conference on Genetic Algorithms*, San Diego, California, 1991. Morgan Kaufmann.

[3] J. L. Blanton and R. L. Wainwright. Vehicle routing with time windows using genetic algorithms. In W. A. Coberly, editor, *Proceedings of the Sixth Oklahoma Symposium on Artificial Intelligence*, pages 242–251, Tulsa, Oklahoma, Nov. 1992.

[4] J. L. Blanton and R. L. Wainwright. Multiple vehicle routing with time and capacity constraints using genetic algorithms. In Forrest [18], pages 452–459.

[5] L. Bodin, B. Golden, A. Assad, and M. Ball. Routing and scheduling of vehicles and crews: The state of the art. *Comput. Opns. Res.*, 10:62–212, 1983.

[6] R. Bruns. Direct chromosome representation and advanced genetic operators for production scheduling. In Forrest [18].

[7] H. Carpenter and W. B. Dowsland. Practical considerations of the pallet-loading problem. *Journal of the Operational Research Society*, 36(6):489–497, 1985.

[8] E. G. Coffman, M. R. Garey, D. S. Johnson, and R. E. Tarjan. Performance bounds for level-oriented two-dimensional packing algorithms. *SIAM Journal of Computing*, 9(4):808–826, Nov. 1980.

[9] A. L. Corcoran and R. L. Wainwright. A genetic algorithm for packing in three dimensions. In H. Berghel, E. Deaton, G. Hedrick, D. Roach, and R. Wainwright, editors, *Proceedings of the 1992 ACM/SIGAPP Symposium on Applied Computing*, pages 1021–1030, New York, 1992. ACM Press.

[10] A. L. Corcoran and R. L. Wainwright. A heuristic for improved genetic bin packing. *Information Processing Letters*, May 1993. Submitted.

[11] A. L. Corcoran and R. L. Wainwright. LibGA: A user-friendly workbench for order-based genetic algorithm research. In E. Deaton, K. M. George, H. Berghel, and G. Hedrick, editors, *Proceedings of the 1993 ACM/SIGAPP Symposium on Applied Computing*, pages 111–118, New York, 1993. ACM Press.

[12] A. L. Corcoran and R. L. Wainwright. A parallel island model genetic algorithm for the multiprocessor scheduling problem. In E. Deaton, K. M. George, H. Berghel, and G. Hedrick, editors, *Proceedings of the 1994 ACM/SIGAPP Symposium on Applied Computing*, New York, 1994. ACM Press. Submitted.

[13] Y. Davidor, T. Yamada, and R. Nakano. The ECOlogical framework II: Improving GA performance at virtually zero cost. In Forrest [18].

[14] K. A. Dowsland. An exact algorithm for the pallet loading problem. *European Journal of Operational Research*, 31:78–84, 1987.

[15] F. F. Easton and N. Mansour. A distributed genetic algorithm for employee staffing and scheduling problems. In Forrest [18].

[16] H.-L. Fang, P. Ross, and D. Corne. A promising genetic algorithm approach to job-shop scheduling, re-scheduling, and open-shop scheduling problems. In Forrest [18].

[17] S. Floyd and R. M. Karp. Ffd bin packing for item sizes with uniform distributions on [0, 1/2]. *Algorithmica*, 6(2):222–239, 1991.

[18] S. Forrest, editor. *Proceedings of the Fifth International Conference on Genetic Algorithms*, Urbana-Champaign, Illinois, July 1993. Morgan Kaufmann.

[19] M. R. Garey and D. S. Johnson. Approximation algorithms for bin packing problems: A survey. In G. Ausiello and M. Lucertini, editors, *Analysis and Design of Algorithms in Combinatorial Optimization*, pages 147–172. Springer-Verlag, New York, 1981.

[20] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, Reading, Massachusetts, 1989.

[21] J. H. Holland. *Adaptation in Natural and Artificial Systems.* The University of Michigan Press, Ann Arbor, Michigan, 1975.

[22] E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms.* Computer Science Press, 1984.

[23] T. C. Hu. *Combinatorial Algorithms.* Addision-Wesley, 1982.

[24] P. Husbands and F. Mill. Simulated co-evolution as the mechanism for emergent planning and scheduling. In Belew and Booker [2].

[25] D. S. Johnson, A. Demers, J. D. Ullman, M. R. Garey, and R. L. Graham. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM Journal of Computing*, 3(4):299–325, Dec. 1974.

[26] M. D. Kidwell. Using genetic algorithms to schedule distributed tasks on a bus-based system. In Forrest [18].

[27] J. Y. Leung, T. W. Tam, C. S. Wong, G. H. Young, and F. Y. Chin. Packing squares into a square. *Journal of Parallel and Distributed Computing*, 10:271–275, 1990.

[28] K. Li and K.-H. Cheng. Job scheduling in a partitionable mesh using a two-dimensional buddy system partitioning scheme. *IEEE Transaactions on Parallel and Distributed Systems*, 2(4):413–422, Oct. 1991.

[29] R. Männer and B. Manderick, editors. *Parallel Problem Solving from Nature, 2.* North-Holland, Amsterdam, 1992.

[30] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, Englewood Cliffs, New Jersey, 1982.

[31] R. G. Parker and R. L. Rardin. *Discrete Optimization.* Academic Press, New York, 1988.

[32] D. Smith. Bin packing with adaptive search. In J. J. Grefenstette, editor, *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, pages 202–207, Hillsdale, New Jersey, 1985. Lawrence Erlbaum.

[33] M. M. Solomon. Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations Research*, 35(2):254–265, 1987.

[34] M. M. Solomon and J. Desrosiers. Time window constrained routing and scheduling problems: A survey. *Transportation Science*, 22(1):1–11, 1988.

[35] T. Starkweather, S. McDaniel, K. Mathias, D. Whitley, and C. Whitley. A comparison of genetic sequencing operators. In Belew and Booker [2], pages 69–76.

[36] H. Tamaki and Y. Nishikawa. A paralleled genetic algorithm based on a neighborhood model and its application to the jobshop scheduling. In Männer and Manderick [29].

[37] S. R. Thangiah. *GIDEON: A Genetic Algorithm System for Vehicle Routing with Time Windows*. PhD thesis, North Dakota State University, May 1991.

[38] S. R. Thangiah, K. E. Nygard, and P. L. Juell. Gideon: A genetic algorithm system for vehicle routing with time windows. In *Proceedings of the Seventh Conference on Artificial Intelligence Applications*, pages 322–325, Miami, Florida, 1991.

[39] D. Whitley and J. Kauth. GENITOR: A different genetic algorithm. In *Proceedings of the Rocky Mountain Conference on Artificial Intelligence*, pages 118–130, Denver, Colorado, 1988.

[40] D. Whitley and T. Starkweather. GENITOR II: A distributed genetic algorithm. *Journal of Experimental and Theoretical Artificial Intelligence*, 2:189–214, 1990.

[41] D. Whitley, T. Starkweather, and D. Fuquat. Scheduling problems and traveling salesman: The genetic edge recombination operator. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, Arlington, Virginia, 1989. Morgan Kaufmann.

[42] T. Yamada and R. Nakano. A genetic algorithm applicable to large-scale job-shop problems. In Männer and Manderick [29].

Appendix: ga-test.cfg

```
#============================================================================
# (c) Copyright Arthur L. Corcoran, 1992, 1993.  All rights reserved.
#
# Genetic Algorithm configuration file
#============================================================================
#----------------------------------------------------------------------------
# User data file
#    This information is not used by the GA, however, it is a convenient
#    way to input a data file name or other information to your application.
#----------------------------------------------------------------------------
# user_data datafile

#----------------------------------------------------------------------------
# Seed for random number generator
#
# Usage: rand_seed my_pid
#        rand_seed number
#
#    my_pid = use system pid as random seed
#    number = seed for random number generator, a positive integer
#
# DEFAULT: rand_seed 1
#----------------------------------------------------------------------------
# rand_seed my_pid
# rand_seed 1

#----------------------------------------------------------------------------
# The data type of the allele
#
# Usage: datatype [bit | int | int_perm | real]
#
#    bit      = bit string
#    int      = integers
#    int_perm = permutation of integers
#    real     = real numbers
#
# DEFAULT: int_perm
#----------------------------------------------------------------------------
# datatype bit
# datatype int
# datatype int_perm
# datatype real
```

```
#-------------------------------------------------------------------------
# How to initialize the pool
#
# Usage: initpool [random | from_file filename | interactive]
#
#    random      = generate at random based on
#                      datatype, chrom_len, & pool_size
#    from_file   = read from a file
#       filename = the name of the file to read from
#    interactive = read from stdin
#
# DEFAULT: initpool random
#-------------------------------------------------------------------------
# initpool random
# initpool from_file   initpool.dat
# initpool interactive

#-------------------------------------------------------------------------
# Chromosome length, needed when "initpool random" selected
#
# Usage: chrom_len length
#
#    length = chromosome length, a positive integer
#
# DEFAULT: chrom_len 10
#-------------------------------------------------------------------------
# chrom_len 25

#-------------------------------------------------------------------------
# Pool size, needed when "initpool random" selected
#
# Usage: pool_size size
#
#    size = pool size, a positive integer
#
# DEFAULT: 100
#-------------------------------------------------------------------------
# pool_size 200
```

```
#-----------------------------------------------------------------------------
# When to stop the GA
#
#    Convergence means when the variance = 0, or equivalently, when
#    all the fitness values in the pool are identical.
#
#    Iterations means the number of generations for the generational model
#    and the number of trials for the steady state model.  Numbers must
#    be given as positive integers.  It takes roughly pool_size/2
#    iterations of the steady state model to equal one iteration of
#    the generational model.
#
# Usage: stop_after convergence
#        stop_after number [use_convergence | ignore_convergence]
#
#    convergence           - stop when the GA converges
#    number                - stop after specified number of iterations
#       use_convergence    - will stop early if GA converges (default)
#       ignore_convergence - WILL NOT stop early even if GA converges
#
# DEFAULT: stop_after convergence
#-----------------------------------------------------------------------------
# stop_after convergence
# stop_after 500
# stop_after 500 use_convergence
# stop_after 500 ignore_convergence

#-----------------------------------------------------------------------------
# GA Type:
#
# Usage: ga [generational | steady_state]
#
#    generational = generational GA
#    steady_state = steady-state GA
#
# WARNING: This directive has the following side effects:
#
#         GA type         Directives set as a side effect
#       ------------      -------------------------------
#       generational      selection        roulette
#                         replacement      append
#                         rp_interval      1
#
#       steady-state      selection        rank_biased
#                         replacement      by_rank
#                         rp_interval      100
#
# DEFAULT: ga generational
#-----------------------------------------------------------------------------
# ga generational              # most commonly used
# ga steady_state              # used by Genitor
```

```
#-----------------------------------------------------------------------------
# Generation gap:
#
#    The generation gap  represents a percentage of the population to copy
#    (clone) to the new pool at each generation.  This only makes sense in
#    a GA with two pools as in the generational model.  A gap of 0.0
#    is the traditional generational algorithm.  As the gap increases,
#    it becomes more like a steady-state algorithm.  A gap of 1.0
#    essentially disables crossover since only reproduction occurs.
#
# Usage: gap number
#
#    number = generation gap, valid range = [0.0 .. 1.0]
#
# DEFAULT: gap 0.0
#-----------------------------------------------------------------------------
# gap 0.3

#-----------------------------------------------------------------------------
# Selection method:
#
# Usage: selection [roulette | rank_biased | uniform_random]
#
#    roulette       = Roulette wheel
#    rank_biased    = Ranked, biased selection as in Genitor
#    uniform_random = Pick one at random
#
# DEFAULT: selection roulette
#-----------------------------------------------------------------------------
# selection roulette           # use with generational GA
# selection rank_biased        # use with steady-state GA
# selection uniform_random     # experimental

#-----------------------------------------------------------------------------
# Selection bias
#
# Usage: bias number
#
#    number = selection bias, valid range = [1.0 .. 2.0]
#             Only used for rank_biased selection
#
# DEFAULT: bias 1.8
#-----------------------------------------------------------------------------
# bias 1.1
```

```
#-------------------------------------------------------------------------
# Crossover method:
#
# Usage: crossover [simple | uniform | order1 | order2 | position | cycle |
#                   pmx | uox | rox | asexual]
#
#    simple    = children get alternate "halves" of parents
#    uniform   = alleles swapped uniformly
#    order1    = order based
#    order2    = order based
#    position  = order based
#    cycle     = order based
#    pmx       = order based
#    uox       = uniform order
#    rox       = relative order
#    asexual   = swap two alleles
#
# DEFAULT: crossover order1
#-------------------------------------------------------------------------
# crossover simple
# crossover uniform
# crossover order1            # use ony with integer permutations
# crossover order2            # use ony with integer permutations
  crossover position          # use ony with integer permutations
# crossover cycle             # use ony with integer permutations
# crossover pmx               # use ony with integer permutations
# crossover uox               # use ony with integer permutations
# crossover rox               # use ony with integer permutations
# crossover asexual

#-------------------------------------------------------------------------
# Crossover Rate
#
# Usage: x_rate number
#
#    number = crossover rate (percentage), valid range = [0.0 .. 1.0]
#             A crossover rate of 0.0 disables crossover
#
# DEFAULT: x_rate 1.0
#-------------------------------------------------------------------------
# x_rate 0.6

#-------------------------------------------------------------------------
# Mutation method:
#
# Usage: mutation [simple_invert | simple_random | swap]
#
#    simple_invert = invert a bit
#    simple_random = random bit value
#    swap          = swap two alleles
#
# DEFAULT: mutation swap
#-------------------------------------------------------------------------
# mutation simple_invert      # use only with bits
# mutation simple_random      # use only with bits
# mutation swap               # use with any datatype
```

```
#-----------------------------------------------------------------------------
# Mutation Rate
#
# Usage: mu_rate number
#
#    number = mutation rate (percentage), valid range = [0.0 .. 1.0]
#             A mutation rate of 0.0 disables mutation
#
# DEFAULT: mu_rate 0.0
#-----------------------------------------------------------------------------
# mu_rate 0.1

#-----------------------------------------------------------------------------
# Replacement method:
#
# Usage: replacement [append | by_rank | first_weaker | weakest]
#
#    append       = append to new pool, as in generational GA
#    by_rank      = insert in sorted order, as in Genitor
#    first_weaker = replace first weaker found in linear scan of pool
#    weakest      = replace weakest member of the pool
#
# DEFAULT: replacement append
#-----------------------------------------------------------------------------
# replacement append           # use with roulette (generational GA)
# replacement by_rank          # use with rank_biased (steady-state GA)
# replacement first_weaker     # experimental
# replacement weakest          # experimental

#-----------------------------------------------------------------------------
# Objective of GA:
#
# Usage: objective [minimize | maximize]
#
#    minimize = minimize evaluation function
#    maximize = maximize evaluation function
#
# DEFAULT: objective minimize
#-----------------------------------------------------------------------------
# objective minimize
# objective maximize
```

```
#-------------------------------------------------------------------------
# Elitism
#
#    Elitism has two actions.  For a generational GA, elitism makes two copies
#    of the best performer in the old pool and places them in the new
#    pool, thus ensuring the most fit chromosome survives.  The other action
#    works with both models.  In this case, elitism picks the best two
#    chromosomes from the parents and children.  Thus, if a child is not as
#    fit as either parent, it will not be placed in the new pool.  Selecting
#    elitism in LibGA performs both actions.
#
# Usage: elitism [true | false]
#
#    true  = ensure best members survive until next generation
#    false = no guarantee best will survive
#
# DEFAULT: elitism true
#-------------------------------------------------------------------------
# elitism true
# elitism false

#-------------------------------------------------------------------------
# Report type
#
# Usage: rp_type [none | minimal | short | long]
#
#    none     = output nothing
#    minimal  = output configuration and final result
#    short    = output minimal + statistics only
#    long     = output short + dump pool
#
# DEFAULT: rp_type short
#-------------------------------------------------------------------------
# rp_type none
# rp_type minimal
# rp_type short
# rp_type long

#-------------------------------------------------------------------------
# Report interval
#
# Usage: rp_interval number
#
#    number = interval between reports, a positive integer
#
# DEFAULT: rp_interval 1
#-------------------------------------------------------------------------
# rp_interval 10
```

```
#-------------------------------------------------------------------------
# Output report filename
#
# Usage: rp_file file_name [file_mode]
#
#    file_name = name of report file
#    file_mode = optional file mode for fopen()
#       a      = append (DEFAULT)
#       w      = overwrite
#
# DEFAULT: (write to stdout)
#-------------------------------------------------------------------------
# rp_file ga.out
# rp_file ga.out a
# rp_file ga.out w
```