St. Cloud State University

The Repository at St. Cloud State

Culminating Projects in Computer Science and Information Technology

Department of Computer Science and Information Technology

7-2011

Using Secondary Fitness to Break Ties in a Genetic Algorithm for the One-Dimensional Bin-Packing Problem

Justin W. Benjamin

Follow this and additional works at: https://repository.stcloudstate.edu/csit_etds

Part of the Computer Sciences Commons

USING SECONDARY FITNESS TO BREAK TIES IN A GENETIC ALGORITHM FOR THE

ONE-DIMENSIONAL BIN-PACKING PROBLEM

by

Justin W. Benjamin

B.A., Ithaca College, Ithaca, NY, 2002

A Thesis

Submitted to the Graduate Faculty

of

St. Cloud State University

in Partial Fulfillment of the Requirements

for the Degree

Master of Science

St. Cloud, Minnesota

July, 2011

This thesis submitted by Justin W. Benjamin in partial fulfillment of the requirements for the Degree of Master of Science at St. Cloud State University is hereby approved by the final evaluation committee.

Chairperson

Dean
School of Graduate Studies

USING SECONDARY FITNESS TO BREAK TIES IN A GENETIC ALGORITHM FOR THE ONE-DIMENSIONAL BIN-PACKING PROBLEM

Justin W. Benjamin

The one-dimensional bin-packing problem (BPP) is a well known problem in the realm of operations research. In the BPP, objects with pre-defined "weights" are packed into bins, each having a maximum weight capacity. The goal is to minimize the number of bins needed to hold objects.

In a Genetic Algorithm for this problem, an individual candidate solution consists of a permutation of objects representing their order of placement. A heuristic is then used to place objects in bins according to their specified order. The heuristic chosen is generally First Fit, Best Fit, or Worst Fit. The number of bins needed to store the objects is a solution's fitness. In Genetic Algorithms, the fitness value of a solution represents its likelihood to pass on its genetic material to new generations. In the case of the BPP, the less the number of bins needed, the greater a solution's fitness. To create new generations, solutions are "crossed-over" to combine features of their particular solutions and create a new organism. New generations are then created in a cycle until a predetermined limit has been reached and the best organism found becomes the solution to the problem.

Many situations arise where two solutions may have the exact same fitness value according to the number of bins. However, these two solutions may not be equivalent in terms of solving the BPP. Solutions which pack objects more tightly in bins are more desirable because they have more potential to be improved in future generations. To distinguish these solutions, we propose a second fitness criterion: the free space in the final bin of a solution.

By adding this secondary fitness criterion, the performance of the standard GA algorithm for the BPP can be improved by a significant margin. This has implications for any implementation of bin-packing GAs, as the modification is relatively minor and produces better results; therefore being a cost-effective improvement.

June, 2011
Month Year

Approved by Research Committee:

Bryant Julstrom Chairperson

TABLE OF CONTENTS

	Page
LIST OF TABLES	vi
LIST OF FIGURES	vii
Chapter	
I. INTRODUCTION	1
Evolutionary Algorithms	1
Genetic Algorithms	4
Evolution Strategies and Evolutionary Programming	6
An Example: A GA for the 0-1 Knapsack Problem	7
The One-Dimensional Bin Packing Problem	9
Secondary Fitness	12
Genetic Algorithms for BPP	13
The Implementation	15
Testing	16
Results	17
Discussion	19
Application to Other Problems	20
Conclusion	23
REFERENCES	24

Chapter Page

APPENDICES

A. Sample Data File 28

B. Program Code 30

LIST OF TABLES

Table Page

1. Results for All Test Instances 17

2. Results for Selected Test Instances 18

LIST OF FIGURES

Figure		Page
1.	Pseudo-Code Summary of Gas	6
2.	0-1 Knapsack	9
3.	Differences in "Quality" between Solutions in the One-dimensional Bin Packing Problem	13
4.	First-fit Heuristic in the One-dimensional Bin-packing Problem	14

Chapter I

INTRODUCTION

This paper begins by providing a summary of the concept of evolutionary algorithms and the three major types: genetic algorithms, evolutionary strategies and evolutionary programming.

The one-dimensional bin-packing problem, a well-known operations research problem is then examined. The paper then describes various evolutionary algorithms historically applied to the bin-packing problem.

A typical genetic algorithm for the bin-packing problem is outlined as well as the limitations of its single-fitness evaluation criterion.

A secondary fitness strategy is proposed which can help to take advantage of previously unused information to influence the flow of the GA toward a better overall solution set. This secondary fitness is shown in empirical testing to produce significantly better results than the typical GA using a single fitness criterion. The results of these tests are then examined as well as their implications.

Finally, potential applications of this concept to other similar problems are discussed.

Evolutionary Algorithms

An evolutionary algorithm (EA) is a search heuristic inspired by natural evolutionary processes. According to DeJong [1, p. 24], the concept of EAs "drew inspiration from nature,

rather than faithful (or even plausible) modeling of biological processes. The key issue was identifying and capturing computationally useful aspects of evolutionary processes."

An EA consists of a population of organisms which "compete" in the sense that there is some comparison of their relative values. These organisms "breed" to create new populations and pass on features of their genetic material; namely, their solution to the problem being solved.

This genetic material is in the form of a "chromosome", which represents a possible solution to the problem being examined. The initial population's chromosomes are generally randomly generated; however, they can also be pre-seeded with solutions that have the potential to present good long-term outcomes.

Attached to each chromosome is its numerical fitness which indicates how well the solution the chromosome represents solves the target problem instance. Better solutions will have more opportunities to pass on their genetic information. Genetic information is usually passed on by one of two ways: crossover or mutation.

Crossover, also known as Recombination, combines the solutions of two parent organisms in some way so as to pass on genetic information from each parent. There are various versions of this operator, depending on the type of problem being solved. However, the underlying principle of any crossover operator is to "mate two individuals with different but desirable features ... to produce an offspring that combines both of those features" [2, p. 22].

Mutation modifies a single parent organism (prospective solution) in (typically) a small way so as to retain a majority of its original information but also to differ slightly from

the original organism. The solution produced is always the result of some combination of random choices [2, p. 21].

Typically parent candidates are selected randomly, but the selection process is weighted toward choosing parents who have better overall fitness values in relation to other organisms [1, pp. 54-55]. Choosing parent candidates randomly but putting a slight bias toward ones with a better fitness improves results "without converging on a suboptimal solution" too quickly [1, p. 55].

Another way for an organism to pass on its genetic information is through elitism. Elitism is a process in some evolutionary algorithms where a best-rated solution from a previous generation will "survive" to the next generation and therefore be more likely to pass on its genetic information to future offspring [3, p. 66]. This prevents the loss of the "current fittest member" due to random parent selection [2, p. 66]. This also allows good solutions to persist without being modified through recombination or mutation; it also has the side effect of removing one of the newly created solutions/organisms from the new generation. It also prevents too many parent solutions from staying alive indefinitely [1, p. 40].

New generations of solutions are produced repeatedly and new organisms are built from their combined parents. As the candidates improve slowly over time, better solutions to the problem evolve.

During the EA process, generations continue being created until some particular criterion is met [1, p. 78]. Such criteria might consist of the EA recognizing no further changes in the population, how different the population is as a whole, how close the best fitness is to the best known fitness, or a generational limit [1, pp. 78-79].

This method of problem solving is often applied, as here, to NP-hard problems, where on large problem instances, algorithms guaranteed to find optimum solutions take unfeasibly long [4]. Because evolutionary algorithms are autonomous, human interaction is not needed and therefore large numbers of possible solutions can be processed. This has significant impact for scientific and commercial applications as, once a suitable software simulation can be constructed, the cost of running the simulation is low.

Genetic Algorithms

Genetic Algorithms (GAs) are "the most widely known type of EA" [2, p. 37]. GAs were created by Holland in 1975 in order to study adaptive behavior in natural and artificial systems [2, p. 37], [3, p. 64], [5]. GAs closely model natural evolutionary processes in that they reference organisms/individuals, chromosomes, genes, etc when describing their structures [6, p. 15]. They are widely considered to be "function optimization methods" [2, p. 38]. GAs were intended to be "application-independent" and universal [1, p. 26].

The "classic" GA structure is as follows. Chromosomes consist of strings of bits that represent candidate solutions to the target problem instance [2, p. 38]. This value-vector approach allows GAs to have problem-independent implementations and thus makes them more general [1, p. 41]. The recombination operator retains some genetic aspects of both parents to create a new hybrid solution [2, p. 38]. The classic GA uses one-point crossover, where one section of one parent's chromosome is used and then the remaining values of the second parent's chromosome are used to complete the chromosome [2, p. 38]. This emphasis on crossover is one factor that distinguishes GAs from other EAs [3, p. 64]. The classic GA mutates via a bit-flip, or a change in a single position in the chromosome [2, p. 38]. Mutation rates in the classic GA are usually low [2, p. 38].

The probability that a chromosome will reproduce is proportional to its fitness and there are varied methods of choosing parent candidates [2, p. 38]. This ensures that parents with higher fitness pass on more genetic material to future generations [1, p. 26]. This feature is unique to GAs [3, p. 64]. A popular alternate method of selection is tournament selection, where a number of chromosomes are selected at random from the population and the candidate with the best fitness value is chosen to be a parent [3, p. 66]. This technique mitigates the rapid convergence sometimes encountered when selecting parents via fitness-proportional probabilities [3, p. 66].

Holland's original model called for only one pair of organisms to be crossed per generational cycle and the resulting generation would be selected from the entire pool of parents and the newly developed child organism [3, p. 65], [5]. However, this depends on the implementation and newer GAs instead replace the entire parent population with newly-created child organisms [3, p. 65], [1, p. 26]. Replacing the entire parent population with newly created child solutions solves the problem of a lack of diversity when parents are allowed to continue on through many generations [1, p. 40]. However, to retain good solutions from one generation to the next, many GAs use the previously mentioned method of "elitism".

Newer GAs use chromosomal representations other than bit-strings and recombination methods other than crossover [3, p. 64]. However, in terms of recombination, most newer methods retain the "spirit" of the crossover operator originally proposed by Holland in that they preserve features of both parents' chromosomes [3, p. 64].

GAs have proven to be highly effective and robust search methods despite being relatively non-problem-specific and have been applied in many areas of science and industry [1, p. 27], [6, p. 15].

1. Create a population of randomly generated chromosomes, each representing a solution to the problem.
2. Evaluate the fitness of each chromosome and save the best for reference.
3. For each chromosome in the next generation, do:
Choose two parent candidates at random (for 2-tournaments; more for other types)
Compare their fitnesses
Choose the best as the first parent
Repeat until all needed parents are chosen
Cross the two parents using crossover operator
4. If using elitism, place the saved "best" chromosome in a position in the next generation
5. Repeat steps 2-4 until the desired number of generations has been reached.
6. Report the best found chromsome/solution.

Figure 1

Pseudo-Code Summary of GAs

Evolution Strategies and Evolutionary Programming

Evolution Strategies (ES) is a version of evolutionary computation created by Rechenberg and Schwefel in 1964 while working on a project to minimize drag on a robot operating in a wind tunnel [3, pp. 48-51]. Current ES systems generate new populations by combining the groups of parents and offspring. This larger group is then reduced to the best N individuals where N is the size of the regular population. Descriptions of this strategy label it as: (μ + λ), where μ represents the parent population and λ represents the offspring [1, pp. 36-37], [2, p. 72]. Some modern ES systems also use the (μ, λ) method of population generation where μ parents produce λ offspring and the new generation is selected from the offspring only [6, pp. 162-163], [2, p. 72]. The basic difference between Evolutionary Strategies and traditional Genetic Algorithms is that ESs use a hill climbing procedure with

self-adapting step sizes and inclinations, whereas GAs are more general-purpose adaptive search algorithms [6, p. 164], [2, p. 72].

Evolutionary Programming (EP) was created by Fogel in 1960 while doing research in the area of artificial intelligence [3, p. 41]. Originally, EP searched spaces of finite-state machines, where each machine would process input and the resulting output from the machine determined the fitness of the machine [3, p. 42]. New generations are created using the $(\mu + \mu)$ method, where each parent creates a new offspring and then the best half (by fitness) of the combined parent and child pool is retained [3, p. 43], [1, p. 25]. An interesting effect of this method of population generation is that low-fitness individuals have no chance of being retained, unlike most other EA algorithms [1, p. 35]. However, this can be undesirable in some cases, as it can cause fitness values to converge on a suboptimal value too quickly [1, p. 35]. Also, because parents and children are competing for their place in the next generation, the average ratio of children to parents in the new generation can actually be less than 1:1, something not possible in other EAs [1, p. 35]. Eventually, EP algorithms were modified to allow real-value vectors and ordered lists [3, p. 43].

An Example: A GA for the 0-1 Knapsack Problem

The 0-1 Knapsack Problem (KP) is a simple problem for which GAs are well-suited. In KP, there exists a set of n items, each having a value v and cost c, and a knapsack of capacity C. The objective of the problem is to select a subset of the n items that maximizes the sum of the objects' values and whose sum of costs does not exceed C [2, p. 27], [6, p. 81]. This problem has applications to situations where items must be packed into a container, such as a traveler packing his or her suitcase for a trip and industrial packing and loading [2, p. 27].

In a GA for this problem, the chromosomes representing solutions are binary strings of length N in which 0 indicates that an object is omitted and a 1 represents its inclusion in the knapsack. The ith chromosome value determines the ith object's inclusion in the knapsack. During evaluation, the chromosome is scanned and for every value 1 found, the corresponding object's value and cost are added to the respective sums. The fitness of the solution is based on whether the cost exceeds maximum allowable capacity (fitness would then be zero) and if not, the sum of the included objects' values. Higher sums would indicate that the knapsack contains a higher overall value and would therefore be more desirable.

Figure 2 illustrates the relationships between the chromosome and the array of objects (and their respective weights) in the 0-1 Knapsack problem. Objects referenced with a value of 1 in the chromosome are included in the knapsack and their respective weights are then added to determine the final capacity of the knapsack.

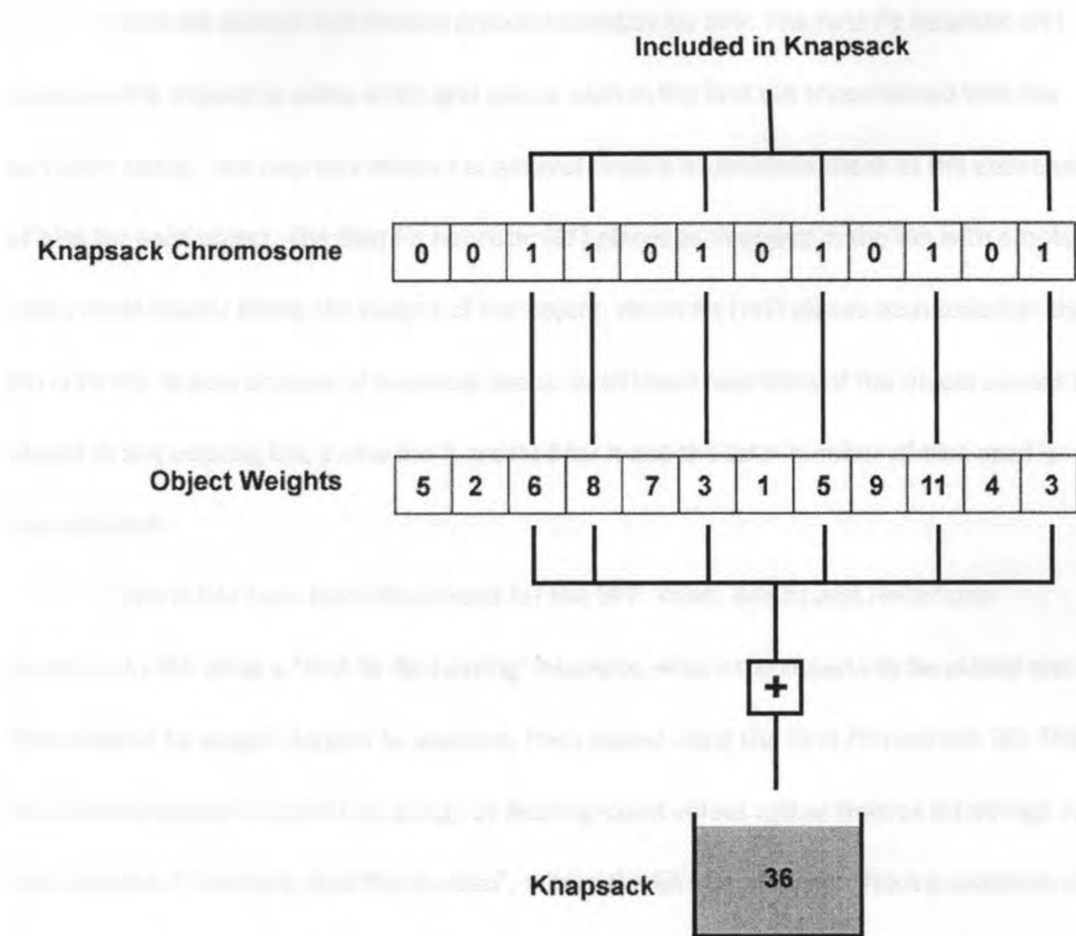


Figure 2

0-1 Knapsack

The One-Dimensional Bin Packing Problem

The One-Dimensional Bin-Packing Problem (BPP) is similar to the 0-1 Knapsack problem. In an instance of BPP, there are n items, each with a size or weight W_i and an unbounded number of bins, each with capacity C. The goal is to place the objects in the smallest possible number of bins without the sum of the weights in any bin exceeding its capacity. The BPP is known to be NP-hard and is therefore a candidate for the application of heuristics, including evolutionary algorithms [7, p. 1].

There are several well-known greedy heuristics for BPP. The First Fit heuristic (FF) examines the objects in some order and places each in the first bin encountered that has sufficient space. This heuristic doesn't in general require a complete check of the entire set of bins for each object. The Best Fit heuristic (BF) places each object in the bin with empty space most closely fitting the weight of the object. Worst Fit (WF) places each object in the bin with the largest amount of available space. In all these heuristics, if the object cannot be placed in any existing bin, a new bin is created for it and the total number of bins used is incremented.

Several EAs have been developed for the BPP. Khuri, Schütz and Heitkötter developed a GA using a "first fit decreasing" heuristic, where the objects to be placed are first ordered by weight, largest to smallest, then placed using the First Fit heuristic [8]. They encoded candidate solutions as strings of floating-point values rather than as bit-strings. Ross et al. applied a "learning classifier system", where the EA would choose from a selection of simple heuristics and "learn" when to apply each of them to solve the BPP problem more effectively [9]. A standard EA seeks to solve the problem, where Ross et al.'s EA seeks to find the best combination of heuristics to achieve the best *method* of finding a solution. Lima and Yakawa [10] describe a GA where genes in each chromosome are groups of objects assigned to a bin. In this case, changing the position or ordering of the genes in the chromosome makes no difference in the fitness of the solution; only changing the content of the gene itself affects the outcome. The solutions represented in the GA's initial population are created by applying the First Fit heuristic to the objects in random orders. Alvim et al. [11] developed a hybrid GA using the "Best Fit Decreasing" heuristic. Their GA is multifaceted and

utilizes pre-seeding of solutions, load redistribution, and solution improvement involving tabu search.

Two other examples of EA solutions to the BPP have similarities to the algorithm described in this paper. The first is the GA created by Falkenauer and Delchambre [12]. When developing the cost (fitness) function for their GA, Falkenauer and Delchambre recognized the need for taking into account how good a solution is. They observe that merely using a count of the number of bins as a fitness criterion is mathematically insufficient; solutions with the same fitness may differ in overall quality [12]. This does not help to guide the algorithm toward better solutions and "the algorithm would have to run into the optimal solution by mere chance" [12]. Falkenauer and Delchambre conclude that given two bins and a set of contents between them, "the situation where one of the bins is nearly full (leaving the other one nearly empty) is better than when the two bins are about equally filled. This is because the nearly empty bin will more easily accommodate additional objects which could otherwise be too big to fit into either of the half-filled bins" [12]. Falkenauer and Delchambre's cost function accounted for this by averaging the sum of the fractions of space used per bin over the number of bins used. This fitness function was applied universally. Their approach differs from the GA described in this paper in the fact that they measure an average of space across all bins and apply the fitness function universally, whereas the GA proposed here applies a modified function only when solutions occupy the same number of bins.

The second GA, given by Reeves, is a hybrid GA that uses a permutation encoding of objects for its chromosomes [13]. Reeves' method of placing objects in bins is essentially the same as the GA described in this paper. The chromosome represents an ordering of objects,

where each successive object is placed in a bin based on either the First Fit or Best Fit heuristic methods [13]. It is interesting to note that Reeves finds that the Best Fit heuristic has improved performance over the First Fit heuristic (the heuristic used in this paper) [13]. Future research should explore this application.

As with the Knapsack problem, the BPP has obvious applications in shipping and packing.

Secondary Fitness

Because the traditional bin-packing GA measures fitness via the total number of used bins, it loses important information about the potential a particular solution can have to produce an improved offspring. One such measure is how tightly "packed" a solution may be. For instance, two solutions may use the same number of bins, but one solution may have its unused space scattered freely about the bins, while the other has most of its free space in its last bin. In this case, the second solution would be preferred as it is likelier to improve under the genetic operators. Since this information and the overall number of bins are both important for evaluating the solution, the proposed modification is to create a dual-fitness GA. The secondary fitness will come into play as a "tiebreaker" when the number of bins in the two solutions is the same and traditional GAs would choose a solution at random. This modification should rescue solutions with the potential to produce better offspring from being discarded for lesser solutions. Overall, this should effect a slight performance increase from the standard method, as it does not aggressively cause convergence of solutions but only slightly tips the scales in favor of better solutions during comparisons. This modification can be used when comparing two parent solutions for crossover as well as when comparing solutions for elitism. Finally, it is unclear whether the accumulation of space in the final bin

alone is a deciding factor, or whether space accumulated in any particular bin should be examined. It is prudent to test both in this case. Figure 3 illustrates the difference between loosely and tightly packed solutions.

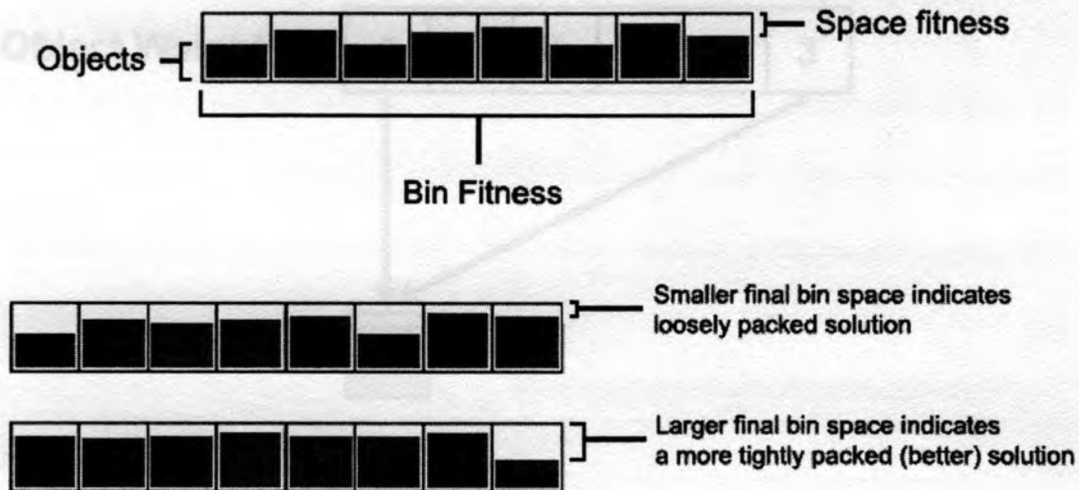
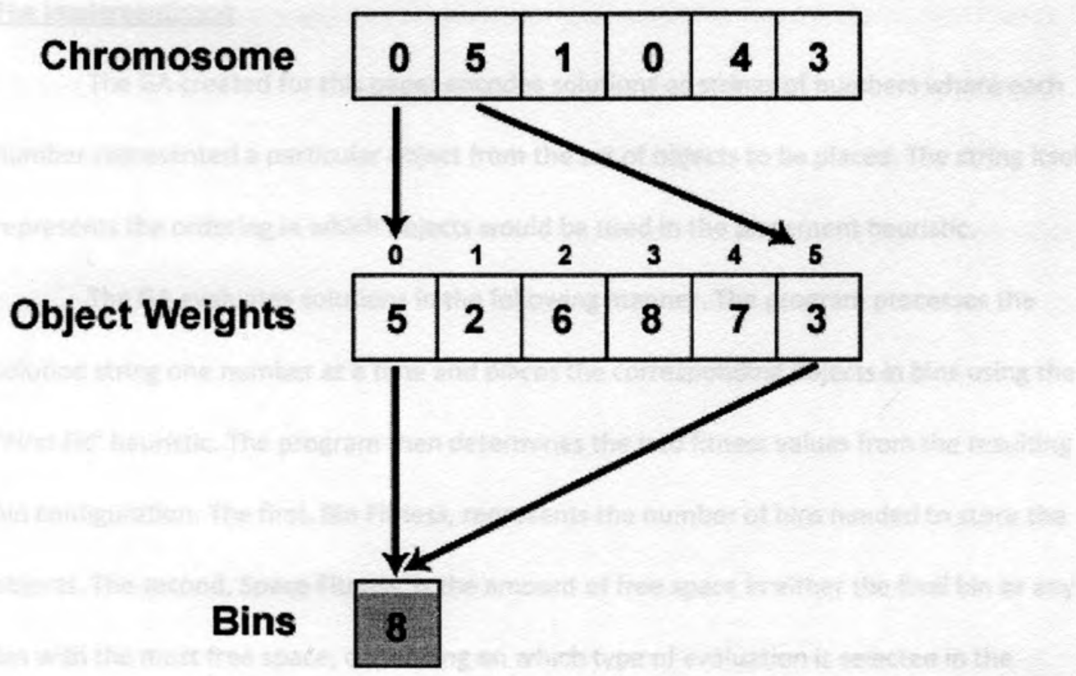


Figure 3

Differences in "Quality" between Solutions in the One-dimensional Bin Packing Problem

Genetic Algorithms for BPP

In a GA for BPP, chromosomes are permutations of integers that represent the objects, as in Reeves [13]. The numbers index the array of objects. When a chromosome is evaluated, the GA iterates through the chromosome and places objects in bins according to a placement heuristic, typically First Fit, Worst Fit, or Best Fit [7]. At the end of the evaluation, the fitness of the solution is determined by the overall number of bins used. Fewer bins indicate a better (more fit) solution. Figure 4 illustrates object placement using the First Fit heuristic.



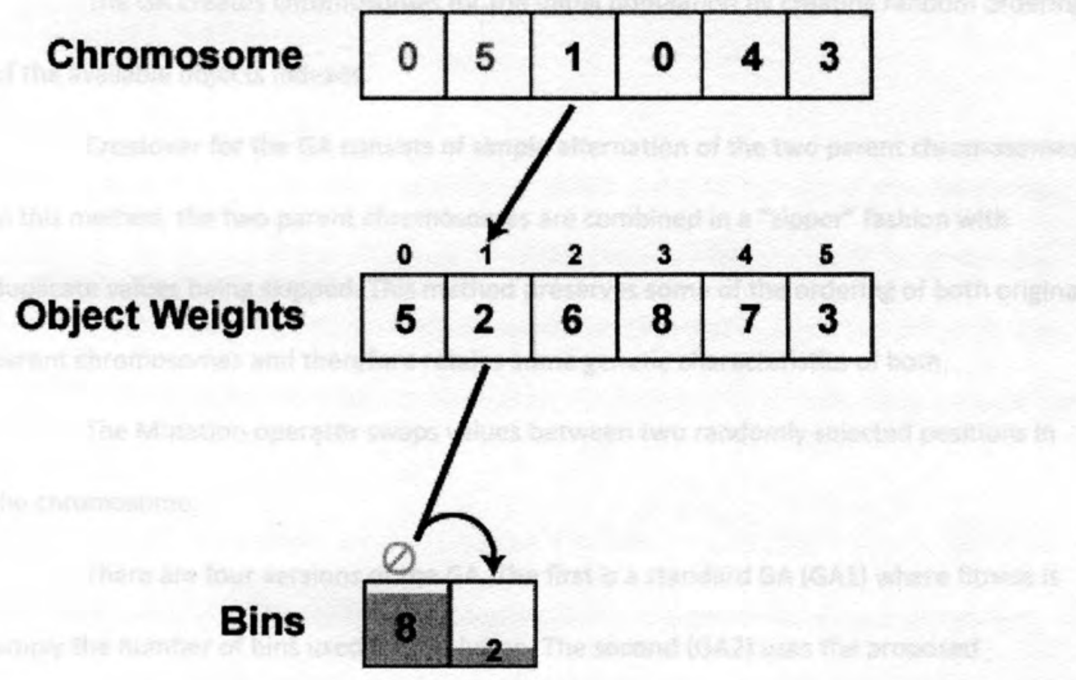


Figure 4

First-fit Heuristic in the One-dimensional Bin-packing Problem (Bin capacity = 9)

The Implementation

The GA created for this paper encodes solutions as strings of numbers where each number represented a particular object from the set of objects to be placed. The string itself represents the ordering in which objects would be used in the placement heuristic.

The GA evaluates solutions in the following manner. The program processes the solution string one number at a time and places the corresponding objects in bins using the "First Fit" heuristic. The program then determines the two fitness values from the resulting bin configuration. The first, Bin Fitness, represents the number of bins needed to store the objects. The second, Space Fitness, is the amount of free space in either the final bin or any bin with the most free space, depending on which type of evaluation is selected in the parameters of the GA.

The GA creates chromosomes for the initial population by creating random orderings of the available objects indexes.

Crossover for the GA consists of simple alternation of the two parent chromosomes. In this method, the two parent chromosomes are combined in a "zipper" fashion with duplicate values being skipped. This method preserves some of the ordering of both original parent chromosomes and therefore retains some genetic characteristics of both.

The Mutation operator swaps values between two randomly selected positions in the chromosome.

There are four versions of the GA. The first is a standard GA (GA1) where fitness is simply the number of bins used for a solution. The second (GA2) uses the proposed secondary fitness value in addition to the number of bins; however, this is used when comparing organisms for parent selection only. The third GA (GA3) uses the secondary

fitness for both parent selection and elitism comparisons. The final GA version (GA4) uses secondary fitness for both parent selection and elitism, but instead of using only the final bin's free space, it uses the largest free space in any bin as the secondary fitness value.

Each version initializes its population with random chromosomes. Parents for each new organism in the next generation are chosen using 2-tournaments. This method selects two parent candidates for each final parent organism at random from the population and compares them; the fitter of the two is selected as a parent. Comparisons of solutions are performed according to the GA type selected in the parameters (i.e., GA1, GA2, GA3, GA4). Each offspring is generated using either crossover or mutation, not both. The GA incorporates 1-elitism, where the best organism of each generation is preserved into the next. Again, comparisons of solutions for elitism are done according to the GA type selected. The GA is halted after a fixed number of generations.

Testing

The size of the test GA's population was 200. The total number of generations was set to 100. The chance of mutation was set at 6%; therefore the chance of crossover was 94%. The number of times to run the selected GA on each test instance was set to 50. The entire set of test instances were run through the GA a total of four times; once for each type of GA.

The GAs were implemented in Java on a PC with an Intel Core 2 Duo processor and 2GB of RAM, running Ubuntu Linux. Parameter information was read from the keyboard. The program then iterated through the list of available test instances, running a full GA 50 times for each and writing the results to a comma-delimited data file.

A total of 738 test instances were used; 20 from Beasley Operations Research Library [14] and 718 from the Jena University Operations Research department website [15]. Minimum numbers of bins are known for all of these instances. Because the OR Library's test set was used by [11], [8], [7], [13], and the Jena University OR test set was used by [11], [9], it was appropriate to use them for this study.

For each test instance, the best found, the mean, and the standard deviation of the fitnesses, as well as the number of test runs (out of 50) that found the best known fitness were recorded.

Results

The testing showed a significant effect of the dual-fitness modification on the results of the GA. The first run of all test instances was meant as a benchmark against which the other tests could be evaluated. The results of all runs are as follows:

Table 1

Results for All Test Instances

GA types:
GA1: Standard GA
GA2: GA with Dual Fitness, Selection Only
GA3: GA with Dual Fitness Selection and Dual Fitness Elitism
GA4: GA with Dual Fitness Selection, Dual Fitness Elitism and Check of Largest Max Bin Space For Any Bin

GA Type	Avg # Test Runs (of 50) Hitting Best Fitness:	Avg % GA Hit Best Fitness For Test Runs (of 50):	Avg Standard Deviation Of Best Found From Mean
GA1	8.090541	16.18108	0.255238
GA2	8.659459	17.31892	0.257387
GA3	8.766216216	17.53243	0.252353
GA4	8.418919	16.83784	0.253802

The dual-fitness modification for selection only (GA 2) showed a 1.13% increase over the standard GA in the proportion of test runs that hit the best fitness. Furthermore, using dual-fitness evaluation for both crossover and elitism raised this percentage to 1.35%. Surprisingly, including a full-chromosome bin check in the fourth run actually decreased the improvement to only 0.65% better than the standard GA. On average, the proportion of comparisons invoking secondary fitness to the overall number of comparisons was rather high, around 0.88 (88%). The table below shows a sample of the results of the testing.

Table 2

Results for Selected Test Instances

		GA1				GA2				GA3				GA4			
TestSet	Best Known	Best Found	# Hit Best	Mean	Std Dev	Best Found	# Hit Best	Mean	Std Dev	Best Found	# Hit Best	Mean	Std Dev	Best Found	# Hit Best	Mean	Std Dev
u120_03	49	49	1	49.98	0.140	49	1	49.98	0.140	49	6	49.88	0.325	49	1	49.98	0.140
u120_05	48	48	31	48.38	0.485	48	37	48.26	0.439	48	46	48.08	0.271	48	34	48.32	0.466
u120_06	48	48	8	48.84	0.367	48	16	48.68	0.466	48	17	48.66	0.474	48	13	48.74	0.439
u120_07	49	50	0	50.00	0.000	50	0	50.00	0.000	49	1	49.98	0.140	49	1	49.98	0.140
u120_12	48	49	0	49.00	0.000	49	0	49.00	0.000	48	1	48.98	0.140	49	0	49	0.000
u120_15	48	48	34	48.32	0.466	48	40	48.20	0.400	48	45	48.10	0.300	48	39	48.22	0.414
u120_17	52	53	0	53.00	0.000	52	4	52.92	0.271	52	9	52.82	0.384	52	6	52.88	0.325
u120_18	49	49	24	49.52	0.500	49	25	49.50	0.500	49	38	49.24	0.427	49	26	49.48	0.500
N1C1W2_L.BPP	31	31	42	31.16	0.367	31	46	31.08	0.271	31	48	31.04	0.196	31	50	31	0.000
N1C2W1_F.BPP	22	22	9	22.82	0.384	22	32	22.36	0.480	22	36	22.28	0.449	22	20	22.6	0.490
N1C2W1_L.BPP	25	25	48	25.04	0.196	25	49	25.02	0.140	25	50	25.00	0.000	25	49	25.02	0.140
N1C2W2_A.BPP	24	24	3	24.94	0.237	24	13	24.74	0.439	24	18	24.64	0.480	24	8	24.84	0.367
N1C2W2_P.BPP	23	23	45	23.10	0.300	23	49	23.02	0.140	23	50	23.00	0.000	23	46	23.08	0.271
N1C2W2_R.BPP	25	25	9	25.82	0.384	25	32	25.36	0.480	25	41	25.18	0.384	25	26	25.48	0.500
N2C1W1_K.BPP	55	56	0	56.88	0.325	56	0	56.76	0.427	56	0	56.68	0.466	56	0	56.72	0.449
N2C1W1_L.BPP	55	56	0	56.00	0.000	56	0	56.00	0.000	56	0	56.00	0.000	56	0	56	0.000
N2C1W1_M.BPP	46	48	0	48.00	0.000	48	0	48.00	0.000	48	0	48.00	0.000	48	0	48	0.000
N2C1W1_N.BPP	48	49	0	49.52	0.500	49	0	49.26	0.439	49	0	49.28	0.449	49	0	49.22	0.414
N2C3W1_M.BPP	31	32	0	32.00	0.000	31	3	31.94	0.237	31	5	31.90	0.300	31	2	31.96	0.196

Discussion

Clearly the addition of the dual-fitness evaluation made an impact on the performance of the GA. Although the improvement was slight, the results show that only a small change to the way a GA evaluates organisms can cause a significant difference in the number of times the GA finds an optimum solution to the problem. By adding this new criterion in each evaluation, the GA has been able to reclaim a potentially better contributor to the upcoming generations, which would have been otherwise lost, and to therefore improve the gene pool.

An additional factor to note is the frequency of ties in the number of comparisons made. Eighty-eight percent of the total number of comparisons experienced a tie in the bin fitness of the two candidate solutions. These ties represent potential turning points for the direction of the GA's flow and therefore adding the second fitness criterion would then obviously greatly change the overall outcome. It may seem counterintuitive that ties would be so common, but with a finite selection of objects and a limited space for placement, many solutions are very similar. Comparing very similar solutions often results in a bin-fitness tie and thus reiterates the need for a secondary criterion.

The method proposed here clearly improves the performance of a standard GA for BPP. It is a relatively minor change to the implementation of the classic bin-packing GA algorithm, requiring only the addition of a few logical structures. Because of this, users with existing GA systems, as well as users looking to implement new systems, should find this improvement easy to implement and in the long term quite cost-effective.

Application to Other Problems

The dual fitness tie-breaker method proposed here, if applicable to only the one-dimensional (and multi-dimensional) bin packing problem, would obviously be of limited utility. However, the concept is general and it should be applicable to many similar problems. The fundamental concern of the modification is some secondary evaluation criterion that indicates the "improvability" of each chromosome. The crucial step in applying this concept to any problem is identifying that second criterion.

Secondary fitness is effective on the Bin-Packing problem, which deals with arrangements of objects. There are many problems in Operations Research that deal with arrangements or scheduling of objects and secondary fitness should make a similar contribution on these. One common example of this is the problem of Job Shop Scheduling.

The Job Shop Scheduling Problem (JSSP) consists of a set of N jobs which are to be processed by M machines. Each job is made up of a set of operations, that must be processed in order, and each operation requires the use of its machine for an uninterrupted period of time. A "schedule" is an ordering of these operations on machines in order to minimize the overall time to process the entire set of jobs, a value called the "makespan" [16].

In a GA for the JSSP, there are several methods that can be chosen, some quite complex. However, the one that seems to be the closest to Bin-Packing is a permutation-based encoding of chromosomes [16]. A chromosome consists of a permutation of the operations to be completed for all jobs, placed in the order in which they would need to be processed for each job, but with the chromosome divided into subsections for each machine. So for example, if we have jobs:

Job1: op1 op2 op3 op4

Job2: op1 op3 op4 op2

Job3: op4 op3 op1 op2

A chromosome might look something like [2 1 3 1 3 2 3 2 1] where the first three items are the jobs to be processed on machine 1, the second three items for machine 2 and the last three items for machine 3. The number indicates that an operation from that particular job number needs to be processed next. If the job is not yet ready (i.e. it is not done processing on the previous machine), the current machine waits until the job is available.

In order to apply the dual-fitness/tie-breaking concept to this problem, the previously mentioned stall period is an area of focus. Because the overall fitness for the problem is the makespan (the total time to complete all jobs), we need a criterion by which to break ties between solutions having the same makespan. The stall times fit this application nicely. An effective measure might be the maximum stall time for any given machine; the larger the stall time, the worse the solution. This trait would indicate how little wasted space there is in a solution, directly corresponding with the quality of being "tightly packed" in the Bin Packing problem. This would indicate that this new quantity could be used in the same manner. However, it is important to note that finding the proper measurement of this trait is essential; in Bin Packing, a measure of the largest space in any bin in the chromosome actually decreased effectiveness, whereas only using the space in the final bin had the desired improvement. The key in this new application is to find which measurement of stall time should affect the overall outcome and direct experimentation is likely the best way to discover this.

Another possible application of the tie-breaker concept is with a second well-known Operations Research problem called the "Constrained Circular Cutting" problem (CCP). The CCP is a modified version of the "2D Rectangular Cutting Stock" problem. In the 2D Rectangular problem, a piece of rectangular stock material exists. Set numbers of different sized rectangles are cut from the stock material; the objective is to minimize the left over material. In the CCP, instead of rectangles, circles are cut from the stock material [17]. Obviously, this increases the potential waste, as the shapes do not fit as uniformly into the available stock material.

This problem is similar in structure to both BPP and JSSP in that it involves organizing shapes/objects into a finite space and attempting to gain the best output. However, the CCP is different, as its fitness measure is the amount of "wasted" stock material, not the number of items gleaned from it. Despite this, it is possible to adapt the tie-breaker method to fit this scenario. As the fitness value for a possible solution is determined by the overall amount of wasted stock material, and this is determined by subtracting the overall area of the placed circles from the total area of the stock, it is difficult to use individual waste as an effective secondary fitness. However, a better method can be used to determine how "tightly packed" a solution might be. Because the circles are not placed if they overlap on the stock material, any two circles should be at least as far apart as the sum of their radii (a distance of r1 + r2 would indicate the circles are touching). Also, this requirement would indicate the existence of some mechanism for detecting overlapping circles. If the distance between the centers of the two circles is measured (dc) and the sum of the radii of the two circles is measured (sr), then the difference between the two (dc - sr) would indicate how far apart the two circles are on the stock material. The largest difference value between any two adjacent circles

would show the two circles with the largest "empty" space between them: an indication of how "tightly packed" the solution is. This would correspond with the use of the secondary fitness in the other two previously discussed problems.

The tie-breaker concept is a general one and clearly there are many ways it can be applied to various problems, as well as more than one way to the same problem. Although it has been shown here that the idea can be used in the realm of "packing" or "scheduling" type problems, it is possible that it can be applied to other problem sets.

Conclusion

The addition of a secondary fitness value to a genetic algorithm can reclaim possible solutions which would have been lost when using a single fitness criterion. We've seen that ties between solutions in GAs using only one fitness can happen frequently and more desirable solutions can be neglected, causing the outcome to deviate from a more optimal path.

Clearly, this modification to the standard GA for the bin-packing problem is an improvement. Because of the simplicity of the modification, it will be easy to modify an existing GA. It should provide a significant improvement to the existing GA's results. In addition, this modification can be adapted to similar problems in other areas such as job-shop scheduling and stock-cutting and improve their performance as well. There also remains a possibility of adapting this method for use in other unrelated problems.

REFERENCES

REFERENCES

[1] K. De Jong, *Evolutionary Computation: A Unified Approach*. MIT Press, 2005.

[2] A. E. Eiben and J. E. Smith, *Introduction to Evolutionary Computing*. Berlin: Springer, 2003.

[3] T. Bäck, D. Fogel, and Z. Michalewicz, editors, *Evolutionary Computation 1: Basic Algorithms and Operators*. Bristol, UK: Institute of Physics Publishing, 2000.

[4] M. R Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[5] J. H. Holland, "Adaptation in Natural and Artificial Systems," University of Oregon Press, 1975.

[6] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs.* New York: Springer-Verlag, 1996.

[7] E. K. Burke, M. R. Hyde, and G. Kendall, "Evolving bin packing heuristics with genetic programming," In PPSN, vol. 4193 of Lecture Notes in Computer Science, T. P. Runarsson, H. G. Beyer, E. K. Burke, J. J. M. Guervos, L. D. Whitley, and X. Yao, Eds., pp. 860-869. Springer, 2006.

[8] S. Khuri, M. Schütz, and J. Heitkötter, "Evolutionary heuristics for the bin packing problem," in *Proceedings of the 2nd International Conference on Artificial Neural Networks and Genetic Algorithms*, D. W. Pearson, N. C. Steele, and R. F. Albrecht, Eds., pp. 285-288, Vienna: Springer, 1995.

[9] P. Ross, S. Schulenburg, J. G. Marin-Blazquez, and E. Hart, "Hyper-heuristics: Learning to combine simple heuristics in bin-packing problems" in *GECCO-2002: Proceedings of the Genetic and Evolutionary Computation Conference*, W. B. Langdon et al., Eds., pp. 942-948, San Francisco, CA: Morgan Kaufman, 2002.

[10] H. Lima and T. Yakawa, "A new design of genetic algorithm for bin packing," in *Proceedings of the 2003 Congress on Evolutionary Computation*, vol. 2, 2003, pp. 1044-1049.

[11] A. C. F. Alvim, C. C. Ribeiro, F. Glover, and D. J. Aloise, "A hybrid improvement heuristic for the one-dimensional bin-packing problem," *Journal of Heuristics*, vol. 10, no. 2, pp. 205-229, 2004.

[12] E. Falkenauer and A. Delchambre, "A genetic algorithm for bin-packing and line balancing" in *Proceedings of the 1992 IEEE International Conference on Robotics and Automation,* vol. 2, IEEE, Piscataway, IEEE Service Center, NJ, USA , 1992, pp.1186-1192.

[13] C. Reeves, "Hybrid genetic algorithms for bin-packing and related problems," *Annals of Operations Research,* vol. 63, 1996, pp. 371-396.

[14] J. R. Beasley, "OR library one dimensional bin packing instances," April 2010, [Online]. Available: http://people.brunel.ac.uk/~mastjjb/jeb/orlib/binpackinfo.html.

[15] A. Scholl and R. Klein, "Packing data aet 1 for BPP-1," April 2010, [Online]. Available: http://www.wiwi.uni-jena.de/Entscheidung/binpp/bin1dat.htm.

[16] T. Yamada and R. Nakano, "Job-shop scheduling," in *Genetic Algorithms in Engineering Systems*, vol. 55 of IEEE Control Engineering Series, pp. 134-160. Institution of Electrical Engineers, 1997.

[17] M. Hifi and R. M'Hallah, "Approximate algorithms for constrained circular cutting problems," *Computers and Operations Research*, vol. 31, no. 5, 2004, pp. 675-694.

APPENDICES

APPENDIX A

Sample Data File

[Sample Data File]

50
100
99
99
96
96
92
92
91
88
87
86
85
76
74
72
69
67
67
62
61
56
52
51
49
46
44
42
40
40
33
33
30

APPENDIX B

Program Code

[Program Code]

```
import java.lang.*;
import java.lang.Math;
import java.util.*;
import java.io.*;
import java.util.Random;

public class GA
{
 Random rand; /* Random number generator */

 /* Command line parameter storage */
 int cla_a, cla_b, cla_c, cla_d, cla_e, cla_f, cla_g, cla_h, cla_i, cla_j, cla_k;

 /* Params */
 int TESTDATA=0;
 int DUALFITNESS;
 int DUALFITNESSELITISM=0;
 int USEMAXSPACEALLBINS;
 int NUMOFTESTRUNS;
 int NUMOFORGANISMS;
 int NUMOFGENS;
 int MUTATIONPERCENTAGE;
 String TESTFILENAME="";
 int BESTKNOWNFITNESS;
 int TOURNEYPARTICIPANTS;

 /* GA Specific Variables */
 int[] ObjectWeights;
 int MAXBINCAPACITY;
 int NUMOFOBJECTS=0;

 /* Internal variables for processing */
 String TESTFILEPATH="";
 String TESTFILEWITHPATH="";
 String TESTSETNAME="";

 /* Variables for file io */
 Scanner in;
 PrintWriter out;

 /* Population variables */
 Organism[] CurrentGen;
```

```
 Organism[] NextGen;
 Organism[] SwapGen;

 /* Data collection variables */
 Organism BestOrganism; /* Holds the best organism found per test run */
 Organism BestOverallOrganism;        /* Holds the best organism found for the entire test
instance over all runs */
 Organism Parent1; /* Holds parent 1 temporarily for operations */
 Organism Parent2; /* Holds parent 2 temporarily for operations */
 long StartTime; /* Start time of GA run */
 long EndTime; /* End time of GA run */
 long ElapsedTime; /* Elapsed time of GA run */
 int MutationCount = 0;  /* Keeps track of number of times mutation was used in the
program execution */
 int CrossoverCount = 0; /* Keeps track of number of times crossover was used in the
program execution */
 double BBFMean; /* Holds the mean of Best Fitnesses for all test runs */
 double BBFStdDev; /* Holds standard deviation of Best Fitnesses for all test runs */
 int BBFFoundCount; /* Holds number of times Best Bin Fitness came up as best found for
test runs */
 int[] BestBinFitnesses; /* Holds the Best Bin Fitnesses found for each test run */
 int[] BestSpaceFitnesses; /* Holds the corresponding space fitnesses for BestBinFitnesses
array */

 public static void main (String[] args) throws FileNotFoundException, IOException
 {


       GA myga = new GA(); /* Create instance of GA object */


       myga.process_args(args); /* Check the command line arguments and set variables
accordingly */


       myga.loadfiledata(); /* Load the test file and fill appropriate variables */


       /* Prep data reporting for entire test instance */
       myga.BestBinFitnesses = new int[myga.NUMOFTESTRUNS];
       myga.BestSpaceFitnesses = new int [myga.NUMOFTESTRUNS];
       myga.BBFMean = 0.0;
```

```
/* Initialize BestOverallOrganism */
        myga.BestOverallOrganism = new Organism(myga.NUMOFOBJECTS);
        myga.BestOverallOrganism.BinFitness = myga.NUMOFOBJECTS; /* sets the value of
the best bin fitness to the total number of objects */
                                    /* i.e. one object per bin - worst case scenario */
  myga.BestOverallOrganism.SpaceFitness = 0; /* Sets space fitness to worst scenario */


        /* -------- GA Start -------- */
  myga.StartTime = System.currentTimeMillis();


        /* Test run loop - START OUTER TEST RUN LOOP */
  for (int runs = 0; runs < myga.NUMOFTESTRUNS; runs++)
  {

        /* Reset best organism object */
    myga.BestOrganism = new Organism(myga.NUMOFOBJECTS);
    myga.BestOrganism.BinFitness = myga.NUMOFOBJECTS; /* sets the value of the best bin
fitness to the total number of objects */
                                    /* i.e. one object per bin - worst case scenario */
    myga.BestOrganism.SpaceFitness = 0; /* Sets space fitness to worst scenario */


        /* Create list of organisms */
    myga.CurrentGen = new Organism[myga.NUMOFORGANISMS];
        myga.NextGen = new Organism[myga.NUMOFORGANISMS];
        for (int i=0; i<myga.NUMOFORGANISMS; i++)
        {
         myga.CurrentGen[i] = new Organism(myga.NUMOFOBJECTS);
        }


        /* Run for all generations - START GENERATIONAL LOOP */
    for (int gens = 0; gens < myga.NUMOFGENS; gens++)
        {

          /* Evaluate their fitnesses */
              for (int orgs=0; orgs<myga.NUMOFORGANISMS; orgs++)
              {
               myga.CurrentGen[orgs].evaluate_fitness(myga.ObjectWeights,
myga.MAXBINCAPACITY, myga.USEMAXSPACEALLBINS);
```

```
        }

        /* Get best organism */
    myga.BestOrganism = myga.getbestorganism(myga.CurrentGen, myga.BestOrganism);

            /* Set the best organism to organism in 0 slot (elitism) */
        myga.NextGen[0] = myga.BestOrganism.clone();

            /* Initialize random number generator */
        myga.rand = new Random();

        /* Create next generation, selecting parents for each organism based on
fitnesses, using 2-pt crossover or mutation */
            for (int i=1; i<myga.NUMOFORGANISMS; i++)
            {

                if (myga.rand.nextInt(100) < myga.MUTATIONPERCENTAGE)
                {
                    myga.Parent1 = myga.tournament(myga.CurrentGen,
myga.TOURNEYPARTICIPANTS, myga.DUALFITNESS);
                    myga.NextGen[i] = myga.Parent1.mutateandspawn();
                    myga.MutationCount++;
                }
                else
                {
                    myga.Parent1 = myga.tournament(myga.CurrentGen,
myga.TOURNEYPARTICIPANTS, myga.DUALFITNESS);
                    myga.Parent2 = myga.tournament(myga.CurrentGen,
myga.TOURNEYPARTICIPANTS, myga.DUALFITNESS);
                    myga.NextGen[i] = myga.crossover(myga.Parent1, myga.Parent2);
                    myga.CrossoverCount++;
                } /* end mutation if */

            } /* end for */

            /* Swap Generations */
            myga.CurrentGen = myga.NextGen;
            myga.NextGen = null;
            myga.NextGen = new Organism[myga.NUMOFORGANISMS];
```

```
        } /* Run for all generations - END GENERATIONAL LOOP */


        /* Record fitnesses for best object in this test run */
        myga.BestBinFitnesses[runs] = myga.BestOrganism.BinFitness;
        myga.BestSpaceFitnesses[runs] = myga.BestOrganism.SpaceFitness;
        if (myga.BestOrganism.BinFitness == myga.BESTKNOWNFITNESS) {
myga.BBFFoundCount++; }

        /* Update best overall organism with whichever is better - Using
DUALFITNESSELITISM instead of DUALFITNESS */
        myga.BestOverallOrganism = myga.compare(myga.BestOrganism,
myga.BestOverallOrganism, myga.DUALFITNESSELITISM);


       } /* Test one loop - END OUTER TEST RUN LOOP */

        /* Get end time for GA run */
        myga.EndTime = System.currentTimeMillis();

        /* -------- GA End -------- */



        /* -------- Calculations and reporting -------- */

        /* Calculate elapsed time, mean, std dev */
        myga.ElapsedTime = myga.EndTime - myga.StartTime;
        myga.BBFMean = myga.mean(myga.BestBinFitnesses);
   myga.BBFStdDev = myga.stddev(myga.BestBinFitnesses);


        /* Output results to console */
        System.out.print("Test Set: " + myga.TESTSETNAME + " - BK: " +
myga.BESTKNOWNFITNESS + " - BBF: ");
   System.out.print( myga.BestOverallOrganism.BinFitness + " - BSF: " +
myga.BestOverallOrganism.SpaceFitness);
        System.out.print(" - BBFFound: " + myga.BBFFoundCount);
        System.out.print(" - BBFFound%: " + ((myga.BBFFoundCount *
100)/myga.NUMOFTESTRUNS));
   System.out.print(" - BBFMean: " + myga.BBFMean);
        System.out.print(" - BBFStdDev: " + myga.BBFStdDev);
```

```
        System.out.print(" - Time(ms): " + myga.ElapsedTime + "\n");

    /* Output results to file */
        myga.out = new PrintWriter(new FileWriter(new File("testresults.csv"), true));
        myga.out.print(myga.TESTSETNAME + ",");
    myga.out.print(myga.BESTKNOWNFITNESS + ",");
        myga.out.print(myga.BestOverallOrganism.BinFitness + ",");
        myga.out.print(myga.BestOverallOrganism.SpaceFitness + ",");
        myga.out.print(myga.BBFFoundCount + ",");
        myga.out.print(((myga.BBFFoundCount * 100)/myga.NUMOFTESTRUNS) + ",");
        myga.out.print(myga.BBFMean + ",");
        myga.out.print(myga.BBFStdDev + ",");
        myga.out.print(myga.ElapsedTime + ",");
        myga.out.print(myga.TESTDATA + ",");
        myga.out.print(myga.DUALFITNESS + ",");
        myga.out.print(myga.DUALFITNESSELITISM + ",");
        myga.out.print(myga.USEMAXSPACEALLBINS + ",");
        myga.out.print(myga.NUMOFTESTRUNS + ",");
        myga.out.print(myga.NUMOFORGANISMS + ",");
        myga.out.print(myga.NUMOFGENS + ",");
        myga.out.print(myga.MUTATIONPERCENTAGE + ",");
        myga.out.print(myga.TOURNEYPARTICIPANTS + ",");
        myga.out.print("\n");
    myga.out.close();

        /* ------- End of GA program ------- */

        return;
  } /* end main */

/* ------- Function definitions ------- */
```

```
/* Checks the arguments to the program and sets variables accordingly */
void process_args(String[] args)
{

  /* Check the command line arguments */
  if (args.length != 11)
  {

        System.out.println("---------------------------------");
        System.out.println("|  Dual Fitness GA Test Suite  |");
        System.out.println("---------------------------------");
   System.out.println("Usage: ga a b c d e f g h i j k");
        System.out.println("---" );
        System.out.println("a: test set ID - 1 for OR library set, 2 for BPP set");
        System.out.println("b: dual fitness for parent selection on/off - 1 for on, 0 for off");
        System.out.println("c: dual fitness for elitism on/off - 1 for on, 0 for off");
        System.out.println("d: max value check for all bins on/off - 1 for on, 0 for off -
otherwise uses only final bin value for fitness");
        System.out.println("e: number of test runs per program - any integer value");
        System.out.println("f: population size - any integer value");
        System.out.println("g: number of generations - any integer value");
        System.out.println("h: mutation chance (%) - any value from 0 to 100, the rest is
given to crossover");
        System.out.println("i: file name - name of the test instance file (do not include path,
test set ID handles path)");
        System.out.println("j: best known fitness - the best known fitness value for the test
instance");
        System.out.println("k: tournament participants - number of participants to compete
to determine the parent candidate");
        System.out.println("---------------------------------");

   System.exit(1);

  } /* end if */


       /* Capture command line arguments into param variables */
       this.cla_a = Integer.parseInt(args[0]);
  this.cla_b = Integer.parseInt(args[1]);
  this.cla_c = Integer.parseInt(args[2]);
  this.cla_d = Integer.parseInt(args[3]);
  this.cla_e = Integer.parseInt(args[4]);
  this.cla_f = Integer.parseInt(args[5]);
  this.cla_g = Integer.parseInt(args[6]);
  this.cla_h = Integer.parseInt(args[7]);
  this.TESTFILENAME = args[8];
```

```
this.BESTKNOWNFITNESS = Integer.parseInt(args[9]);
        this.cla_k = Integer.parseInt(args[10]);


        /* Verify validity of command line args */
        switch (this.cla_a)
    {
    case 1: this.TESTDATA=1; break;
    case 2: this.TESTDATA=2; break;
    default: System.out.println("ERROR: You've chosen an incorrect value for test set, please
try again."); System.exit(1);
    }

    switch (this.cla_b)
    {
    case 0: DUALFITNESS=0; break; /* User chose to use the regular GA method for parent
selection */
    case 1: DUALFITNESS=1; break; /* User chose to use dual fitness GA for testing for parent
selection */
    default: System.out.println("ERROR: You've chosen an incorrect value for dual fitness
option for parent selection, please choose either 0 or 1."); System.exit(1);
    }

    switch (this.cla_c)
    {
    case 0: DUALFITNESSELITISM=0; break; /* User chose not to use dual fitness when
evaluating organisms for elitism */
    case 1: DUALFITNESSELITISM=1; break; /* User chose to use dual fitness when evaluating
organisms for elitism */
    default: System.out.println("ERROR: You've chosen an incorrect value for dual fitness
option for elitism, please choose either 0 or 1."); System.exit(1);
    }

    switch (this.cla_d)
    {
    case 0: USEMAXSPACEALLBINS=0; break; /* User chose to use final bin space only for
evaluation */
    case 1: USEMAXSPACEALLBINS=1; break; /* User chose to use any bin max space for
evaluation */
    default: System.out.println("ERROR: You've chosen an incorrect value for max free space
value check for all bins option, please choose either 0 or 1."); System.exit(1);
    }

    if (this.cla_e <= 0)
    {
    System.out.println("ERROR: You must enter a positive value for number of test runs.");
```

```
 System.exit(1);
}
else
{
 this.NUMOFTESTRUNS=this.cla_e; /* Set test runs from command line arguments */
}

if (this.cla_f <= 0)
{
 System.out.println("ERROR: You must enter a positive value for population size.");
 System.exit(1);
}
else
{
 this.NUMOFORGANISMS=this.cla_f; /* Set population size from command line arguments
*/
}

if (this.cla_g <= 0)
{
 System.out.println("ERROR: You must enter a positive value for number of generations.");
 System.exit(1);
}
else
{
 this.NUMOFGENS=cla_g; /* Set number of gens from command line arguments */
}

if ((this.cla_h > 100)||(this.cla_h < 0))
{
 System.out.println("ERROR: Your value for mutation chance percentage is out of range,
please choose a value between 0 and 100.");
 System.exit(1);
}
else
{
 this.MUTATIONPERCENTAGE=cla_h; /* Set mutation percentage from command line args
*/
}

if (this.cla_k < 2)
{
 System.out.println("ERROR: Your value for tournament participants is out of range, please
choose a value greater than 2.");
 System.exit(1);
}
```

```
else
{
 this.TOURNEYPARTICIPANTS=this.cla_k; /* Set tourney participants from command line
args */
}

/* --- Prep data file variables --- */
/* Set filelist to the proper test data list file */
if (this.TESTDATA == 1)
{
 /* ORLib data chosen */
 this.TESTFILEPATH = "./orlib/";
}
else if (TESTDATA == 2)
{
 /* BPP data chosen */
 this.TESTFILEPATH = "./bpp/";
}
else
{
 /* Invalid test file option */
 System.out.println("You have chosen an invalid test set option: " + this.TESTDATA + ".
Please try again. Exiting...");
 System.exit(1);
}

this.TESTFILEWITHPATH = this.TESTFILEPATH + this.TESTFILENAME;

} /* end process_args */


/* Loads data from test file */
void loadfiledata() throws FileNotFoundException
{

 if (this.TESTFILEWITHPATH == "")
 {
        System.out.println("Filename not specified, exiting...");
               System.exit(1);
       }
```

```
        /* Open file and load data */
  in = new Scanner(new File(TESTFILEWITHPATH));


        /* Get initial data to set variables */
   if (this.TESTDATA == 1)
   {
     this.TESTSETNAME = this.in.nextLine(); /* Get test set name from test file first line */
     this.MAXBINCAPACITY = this.in.nextInt();
     this.NUMOFOBJECTS = this.in.nextInt();
     this.BESTKNOWNFITNESS = this.in.nextInt(); /* Yes, this is redundant, but to make the
file loading smoother, I left it in */
   }
   else if (this.TESTDATA == 2)
   {
          this.TESTSETNAME = this.TESTFILENAME; /* Set the test set name with the file
name because in BPP they're the same, unlike ORLIB */

     switch(this.TESTFILENAME.charAt(1))
     {
       case '1': this.NUMOFOBJECTS = 50;
            break;
       case '2': this.NUMOFOBJECTS = 100;
            break;
       case '3': this.NUMOFOBJECTS = 200;
            break;
       case '4': this.NUMOFOBJECTS = 500;
            break;
       default: System.out.println("Invalid value for BPP first param. Exiting...");
            this.in.close();
            System.exit(1);
     } /* end switch */

     switch(this.TESTFILENAME.charAt(3))
     {
       case '1': this.MAXBINCAPACITY = 100;
            break;
       case '2': this.MAXBINCAPACITY = 120;
            break;
       case '3': this.MAXBINCAPACITY = 150;
            break;
       default: System.out.println("Invalid value for BPP second param. Exiting...");
            this.in.close();
            System.exit(1);
     } /* end switch */
```

```
    }
    else
    {
      /* Invalid test file option */
      System.out.println("You have chosen an invalid test file option: " + this.TESTDATA + ".
Please try again. Exiting...");
      this.in.close();
      System.exit(1);
    } // end if - else if - else

          /* Set up the object weights array */
          this.ObjectWeights = new int[this.NUMOFOBJECTS];

          /* Get the object weights from file */
          for (int objs=0; objs<this.NUMOFOBJECTS; objs++)
    {
      this.ObjectWeights[objs] = this.in.nextInt();
    }


          /* Close the input file */
          this.in.close();


  }       /* end loadfiledata */



  /* Does tournament selection and produces a parent */
  Organism tournament(Organism[] population, int numofcandidates, int DUALFITNESS)
  {
    /* Initialize random number generator */
    this.rand = new Random();

          /* Clone randomly chosen candidates */
    Organism[] candidates = new Organism[numofcandidates];
    for (int i=0; i<numofcandidates; i++)
    {
      candidates[i] = population[this.rand.nextInt(population.length)].clone();
    }

          Organism parent = candidates[0].clone(); /* Get the first one for comparison */
```

```
                /* Compare parents and take the best, using secondary fitness if specified */
        for (int i=0; i<numofcandidates; i++)
  {
    parent = this.compare(parent, candidates[i], DUALFITNESS);
  } /* end for */

        return parent;

 } /* end tournament */


 /* Compares two organisms and returns a copy of the best */
 Organism compare(Organism subject1, Organism subject2, int DUALFITNESS)
 {
   Organism better = subject1.clone(); /* Get the first one for comparison */

   /* Compare parent variable to each, using secondary fitness if specified */
        if ((better.BinFitness == subject2.BinFitness) && (better.SpaceFitness ==
subject2.SpaceFitness))
        {
          /* Both candidate pointers point to the same organism, do nothing */
        }
        else if ((better.BinFitness)==(subject2.BinFitness) && (DUALFITNESS==0))
        {
          /* Bin fitness is the same for both but dual fitness evaluation is turned off */
                /* Results in a tie, any one is fine */
                /* Do nothing, "better" is better */
        }
        else if ((better.BinFitness)==(subject2.BinFitness) && (DUALFITNESS==1))
        {
          /* Bin fitness is the same for both, dual fitness evaluation is turned on */
          if ((better.SpaceFitness) > (subject2.SpaceFitness))
          {
            /* Do nothing, "better" is better */
                }
                else
                {
                  better = subject2;
                }
        }
        else if ((better.BinFitness) < (subject2.BinFitness))
        {
          /* Bin fitnesses are not equal, "better" has less bins than subject2 */
          /* Do nothing, "better" is better */
```

```
        }
        else
        {
          /* Bin fitnesses are not equal, subject2 has less bins than "better" object
            i.e. - ((better->BinFitness) > (subject2->BinFitness))
                  So, replace "better" with subject2 */
          better = subject2;
        }

    return better;

  } /* end compare */


  /* Crossover function that returns reference to child object */
   Organism crossover(Organism parent1, Organism parent2)
   {

    boolean[] Used = new boolean[this.NUMOFOBJECTS]; /* Keeps track of which object
weights have been used */
          Organism childorganism = new Organism(NUMOFOBJECTS); /* The resulting
organism */
     int[] parent1chr = new int[this.NUMOFOBJECTS]; /* Copy of parent1 chromosome */
          int[] parent2chr = new int[this.NUMOFOBJECTS]; /* Copy of parent2 chromosome
*/

          /* Get copy of each parent's chromosome */
          for (int chr=0; chr < this.NUMOFOBJECTS; chr++)
          {
          parent1chr[chr] = parent1.Chromosome[chr];
          parent2chr[chr] = parent2.Chromosome[chr];
          }

          /* Initialize Used[] array to false */
          for (int i=0; i<this.NUMOFOBJECTS; i++)
          {
            Used[i] = false;
          }

          /* Crossover -> Alternation */
```

```
        for (int k=0,j=0; (j<this.NUMOFOBJECTS)&&(k<this.NUMOFOBJECTS); ++j)
        {
          if (!Used[parent1chr[j]])
                {
                  childorganism.Chromosome[k] = parent1chr[j];
                        ++k;
                        Used[parent1chr[j]] = true;
                }
                if (!Used[parent2chr[j]])
                {
                  childorganism.Chromosome[k++] = parent2chr[j];
                        Used[parent2chr[j]] = true;
                }
        }

        /* Return new child */
        return childorganism;

  }; /* end crossover */


  /* Returns the current best organism in the specified generation */
  Organism getbestorganism(Organism[] currentgeneration, Organism currentbest)
  {
    /* Make a copy of the current best organism for comparison */
     Organism newbest = currentbest.clone();

    /* Get best organism based on bin fitness only */
        for (int orgs=0; orgs<this.NUMOFORGANISMS; orgs++)
        {
          /* Note the DUALFITNESSELITISM variable in place of the DUALFITNESS one */
          if ((currentgeneration[orgs].BinFitness == newbest.BinFitness) &&
(this.DUALFITNESSELITISM == 1))
                {
                  if (currentgeneration[orgs].SpaceFitness > newbest.SpaceFitness)
                        {
                          /* Dual fitness check is turned on, Bin Fitnesses are equal and
current organism has more space in last bin than best organism */
                          newbest = currentgeneration[orgs].clone();
                        }
                }
```

```
                else if (currentgeneration[orgs].BinFitness < newbest.BinFitness)
                {
                  /* Bin fitness is less than best org bin fitness, replace current best org with
this one */
                  newbest = currentgeneration[orgs].clone();
                }
                else
                {
                  /* Keep best organism the same */
                } /* end if */

        } /* end for */

  return newbest;

 } /* end getbestorganism */



 /* Computes standard deviation of an array of integers */
 double stddev(int[] fitnesses)
 {

   double squaresum = 0; /* Holds sum of squares */
   double stddev = 0.0; /* Holds standard deviation */
   double groupmean = this.mean(fitnesses); /* Mean of integer fitness list parameter */

   for (int i=0; i<fitnesses.length; i++)
   {
          squaresum += Math.pow(((double) fitnesses[i] - groupmean), 2.0);
   }

   stddev = Math.sqrt(squaresum / (double) fitnesses.length);

   return stddev;

 } /* end stddev */



 /* Computes the mean of an array of integers */
 double mean(int[] fitnesses)
 {
```

```
    int groupsum = 0; /* Holds the sum of the entire group of integers */
   double groupmean = 0.0; /* Holds the mean of the group of integers */

   for (int i=0; i<fitnesses.length; i++)
   {
     groupsum += fitnesses[i];
   }

   groupmean = (double) groupsum / (double) fitnesses.length;

   return groupmean;

  } /* end mean */

} /* end GA class */
```

```
--- Organism.java---
import java.util.Random;


public class Organism
{

  public int[] Chromosome; /* Chromosome array */
  public int BinFitness; /* The smaller this number, the better */
  public int SpaceFitness; /* The bigger the number, the better */
  Random rand; /* Random number generator */


  Organism(int NUMOFOBJECTS)
  {

    this.Chromosome = new int[NUMOFOBJECTS]; /* Organism's chromosome array */
          this.BinFitness = NUMOFOBJECTS; /* Set to worst possible bin fitness */
          this.SpaceFitness = 0; /* Set to worst possible space fitness */
```

```
        this.rand = new Random(); /* Initialize random number generator */

    /* -- Initialize Chromosome With Random Permutation -- */

    int tempvalue; /* Holds the value being swapped */
        int targetindex; /* The random index returned by randomnumber */
        int chrlastused = 0; /* Holds the index of last filled position in child chromosome */
        int lastelement = NUMOFOBJECTS; /* Get last element index */
    int[] sourceindexes = new int[NUMOFOBJECTS]; /* Holds the indexes to choose from */

    /* Set up the source index array */
        for (int i=0; i<NUMOFOBJECTS; i++)
        {
          sourceindexes[i] = i;
        }

        /* Fill new chromosome with randomized index values into object weight array*/
        for (int chr=0; chr<NUMOFOBJECTS; chr++)
        {
          targetindex = rand.nextInt(lastelement); /* get random value between 0 and end
of available elements */
                this.Chromosome[chr] = sourceindexes[targetindex]; /* get chosen value */
      tempvalue = sourceindexes[lastelement-1]; /* copy last element */
      sourceindexes[lastelement-1] = sourceindexes[tempvalue];
                sourceindexes[targetindex] = tempvalue;
                lastelement--;
        }

        /* Chromosome is now set */

  } /* Constructor */


  /* Return a clone of the current organism */
  public Organism clone()
  {

    Organism doppelganger = new Organism(this.Chromosome.length);
        for (int i=0; i<this.Chromosome.length; i++)
        {
          doppelganger.Chromosome[i] = this.Chromosome[i];
```

```
        }
        doppelganger.BinFitness = this.BinFitness;
        doppelganger.SpaceFitness = this.SpaceFitness;

        return doppelganger;


  } /* Clone */


  /* Spawn a mutated copy of the current organism */
  public Organism mutateandspawn()
  {

    Organism doppelganger = this.clone();
        this.rand = new Random();
    int temp = 0;
        int randomindex1 = rand.nextInt(this.Chromosome.length);
        int randomindex2 = rand.nextInt(this.Chromosome.length);

    /* Swap values at a single position in the chromosome - i.e. mutate */
        temp = doppelganger.Chromosome[randomindex1];
        doppelganger.Chromosome[randomindex1] =
doppelganger.Chromosome[randomindex2];
        doppelganger.Chromosome[randomindex2] = temp;

    return doppelganger;

  } /* mutateandspawn */


  /* Evaluate the chromosome and set the fitness variables */
  public void evaluate_fitness(int[] objectlist, int MAXBINCAPACITY, int
USEMAXSPACEALLBINS)
  {
    int NUMOFOBJECTS = objectlist.length; /* Get the total number of objects */

        /* Error checking */
        if (NUMOFOBJECTS != this.Chromosome.length)
        {
          System.out.println("--------------------");
```

```
            System.out.println("ObjectWeights length and Chromosome length do not
match");
            System.out.println("ObjectWeights length: " + NUMOFOBJECTS);
            System.out.println("Chromsome length: " + this.Chromosome.length);
      }

   int binindex = 0; /* Holds the index of the currently used bin */
   int currentobjectweight = 0; /* Holds the weight of the next object to be placed */
      int[] binarray = new int[NUMOFOBJECTS]; /* Array to hold the bin values used */
      boolean placed = false; /* Flags the loop to stop trying to place the value once it's
been put in a bin */
      int j=0; /* Index to bin array items */

      /* Set binarray values to 0 */
      for (int i=0; i<NUMOFOBJECTS; i++)
      {
        binarray[i] = 0;
      }

      /* Loop through all values in the chromosome */
      for (int obj=0; obj<NUMOFOBJECTS; obj++)
      {

        currentobjectweight = objectlist[this.Chromosome[obj]]; /* Gets the weight of the
object indexed by the chromosome value */
            placed = false;
            j=0;

            while ((placed == false) && (j<NUMOFOBJECTS))
            {
              if ((binarray[j] + currentobjectweight) <= MAXBINCAPACITY)
                  { /* Object fits in the bin, place it */
                    binarray[j] = binarray[j] + currentobjectweight;
                    placed = true;

                  } /* end if */
                  else if (j == binindex)
                  {
                    /* Object won't fit in any bin, increase the number of bins used */
                    /* On the next iteration of the for loop, the value will fit and this
will not be activated */
                    binindex++;
                    binarray[binindex] = currentobjectweight;
                    placed = true;
                  }
                  else
```

```
                        {
                          j++; /* go to next bin */
                        }

                    } /* end while */

            } /* end for */

            this.BinFitness = binindex; /* Set the bin fitness to the number of bins used */

            if (USEMAXSPACEALLBINS == 0)
            {
              /* Only check the last bin used for secondary (space) fitness value */
              this.SpaceFitness = MAXBINCAPACITY - binarray[binindex]; /* Set space fitness to
the space left in the last bin */
            }
    else if (USEMAXSPACEALLBINS == 1)
    {
              /* Check all bins for the maximum unused space and report this as secondary
(space) fitness */
              int maxspacefound = 0;
      for (int i=0; i<binindex; i++)
                    {
                      if ((MAXBINCAPACITY - binarray[i]) > maxspacefound) { maxspacefound =
(MAXBINCAPACITY - binarray[i]); }
                    } /* end for */
                    this.SpaceFitness = maxspacefound;
    }
            else
            {
              /* Option for USEMAXSPACALLBINS was out of range, report error */
              System.out.println("ERROR: USEMAXSPACEALLBINS variable passed to
evaluate_fitness is not 0 or 1... exiting.");
                    System.exit(1);
            }

  } /* end evaluateFitness */




} /* end class */
```