A Genetic Algorithm for the Bin Packing Problem

Jordan Junkermeier

Department of Computer Science, St. Cloud State University, St. Cloud, MN 56301 USA

Abstract

The bin packing problem (BPP) is an NP-hard problem of combinatorial optimization. Given a set of numbers and a set of bins of fixed capacity, the problem is to find the minimum number of bins needed to contain all of the numbers (Korf, 2002).

In this paper, a genetic algorithm is proposed, which generates random permutations of the input set and then itself uses the *first-fit* algorithm, an approximate algorithm to the BPP, to generate valid solutions. The goal of the genetic algorithm is to produce optimal permutations of the input set on which the *first-fit* algorithm will be applied.

The comparative results of the two algorithms on three groups of test problem instances are compared and evaluated, and the genetic algorithm is found to produce better solutions than those of the *first-fit* algorithm alone for two-thirds of the problem instances. For the other problem instances, the two algorithms produced solutions with equivalent fitnesses.

1. Introduction

The *first-fit* approximate algorithm has been implemented as an alternative to computationally difficult exact algorithms to produce adequate solutions to the bin packing problem (BPP).

In this paper, a genetic algorithm is proposed, which generates random permutations of the input set and then itself uses the *first-fit* algorithm to generate valid solutions to the BPP. What matters here is the order of the elements in the input set. Therefore, the goal of the genetic algorithm is to produce optimal permutations of the input set on which the *first-fit* algorithm will be applied.

The comparative results of the two algorithms on three groups of test problem instances are compared and evaluated at the end of the paper.

2. The Problem

2.1. Background

The bin packing problem (BPP) is an NP-hard problem of combinatorial optimization. Given a set of numbers and a set of bins of fixed capacity, the problem is to find the minimum number of bins needed to contain all of the numbers (Korf, 2002).

Formally, given n items and n bins, with $w_j = weight$ of item j and $c = capacity$ of each bin, the problem is to minimize $z = \sum_{i=1}^{n}(y_i)$ subject to $\sum_{j=1}^{n}(w_j x_{ij}) \leq cy_i : i \in N = \{1, \dots, n\}$, $\sum_{i=1}^{n}(x_{ij}) = 1 : j \in N$, $y_i = 0$ or $1 : i \in N$, $x_{ij} = 0$ or $1 : i \in N$ and $j \in N$, where $y_i = 1$ if bin i is used, and 0 otherwise, and $x_{ij} = 1$ if item j is assigned to bin i, and 0 otherwise (U. of Bologna: *ORG*).

The following examples demonstrate the bin packing problem. Take the set of numbers $\{1, 2, 3, 4, 5\}$ and a bin capacity of 5. A solution for this problem instance forms bins $\{2, 3\}$, $\{1, 4\}$, and $\{5\}$, for a total of three bins, each at their capacity. In another example, given the set $\{6, 12, 15, 40, 43, 82\}$ and a bin capacity of 100, a minimum of two bins, $\{6, 12, 82\}$ and $\{15, 40, 43\}$, can be formed (Korf, 2002).

In bin packing problem instances, the minimum number of bins in a solution must be at least $\left\lceil \frac{\sum(a_i \in A)}{C} \right\rceil$, where A is the input set and C is the bin capacity. If the number of bins in a solution is equivalent to this expression, then that solution is an optimal solution to the problem instance (Korf, 2002). In both of the above examples, the solutions presented are optimal solutions for their respective problem instances.

2.2. Applications

The bin packing problem can be applied to many areas, both in computing and otherwise. In general, the BPP is applicable to any situation in which identical bins with the same capacity are to be filled with items of a specified weight, such as packing trucks that have a weight capacity C (Malkevitch, 2004).

Specifically, bin packing is used in VLSI circuit clustering, partitioning, and technology mapping, where the area of a gate corresponds to the size of an item and the area of the clusters corresponds to the capacity of the bins (Izumi et al., 1998).

In addition, bin packing is directly related to *machine scheduling* problems, in which a set of independent tasks and a set of identical machines on which to run the tasks are given. Each task requires a certain amount of time to complete on a machine. One might wish to determine the minimum number of machines needed to complete a collection of tasks with

times $t_1, t_2, \ldots, t_n$ in time T. This problem is, in essence, a bin packing problem in which T is the capacity of the bins and each t_i is the weight of an item to be packed. (Malkevitch, 2004).

3. Algorithms

This section briefly describes a few exact algorithms for the bin packing problem, and then examines the *first-fit* approximate algorithm, which is the algorithm used in comparison to the proposed genetic algorithm. Finally, the genetic algorithm for which this paper was written is itself presented and described.

3.1. Exact Algorithms

Very little can be found in the literature on the exact solution of the BPP (Schwerin & Wäscher, 1997; U. of Bologna: *ORG*). Eilon and Christofides (1971) presented a depth-first enumerative algorithm and Hung and Brown (1978) presented a branch-and-bound algorithm, yet both teams indicated that their respective algorithms can only solve small-size problem instances (U. of Bologna: *ORG*).

Another exact algorithm, MTP, proposed by Martello and Toth (1989) is based on a "first-fit decreasing" branching strategy (U. of Bologna: *ORG*), and is the best existing algorithm for optimal packing (Korf, 2002). However, since the goal of this paper is to present and compare approximate heuristic approaches to the bin packing problem, this algorithm will not be discussed further.

3.2. The *First-Fit* Approximate Algorithm

Although many approximate algorithms for the bin packing problem exist, only one, the *first-fit* (FF) algorithm, will be described here. This algorithm is later compared to the genetic algorithm presented in this paper.

The *first-fit* algorithm considers the items in the input set in the order in which they appear in the set. Each item is assigned to the lowest indexed initialized bin into which it fits. If the current item cannot fit into any initialized bin, a new bin is initialized, and the item is placed into that bin (U. of Bologna: *ORG*). **Figure 1** shows the *first-fit* algorithm applied to the set $A = \{4, 5, 6, 3, 9, 2\}$ with a bin capacity of 13.

$$A = \{4, 5, 6, 3, 9, 2\}; C = 13$$

Time ↓

$B_1 = \{\mathbf{4}\}$
$B_1 = \{4, \mathbf{5}\}$
$B_1 = \{4, 5\}$ $B_2 = \{\mathbf{6}\}$
$B_1 = \{4, 5, \mathbf{3}\}$ $B_2 = \{6\}$
$B_1 = \{4, 5, 3\}$ $B_2 = \{6\}$ $B_3 = \{\mathbf{9}\}$
$B_1 = \{4, 5, 3\}$ $B_2 = \{6, \mathbf{2}\}$ $B_3 = \{9\}$

Figure 1. The *first-fit* algorithm applied to the set $A = \{\mathbf{4}, \mathbf{5}, \mathbf{6}, \mathbf{3}, \mathbf{9}, \mathbf{2}\}$ with a bin capacity of 13.

The *first-fit* algorithm's time complexity is $O(N \log N)$, which can be achieved by using a 2–3 tree whose leaves store the current capacities of the initialized bins. Therefore, each iteration of FF requires $O(\log n)$ time, since the number of leaves in the tree is bounded by n. (U. of Bologna: *ORG*). Furthermore, for all instances I of the bin packing problem, $\mathrm{FF}(I) \leq \frac{17}{10}\mathrm{opt}(I) + 2$, where $\mathrm{opt}(I)$ denotes the optimal solution for instance I. Similarly, there exist instances I, with arbitrarily large $\mathrm{opt}(I)$, for which $\mathrm{FF}(I) > \frac{17}{10}\mathrm{opt}(I) - 8$. (Johnson et al., 1974).

It should be noted that along with the *first-fit* algorithm, the *best-fit*, *next-fit-decreasing*, *first-fit-decreasing*, and *best-fit-decreasing* algorithms all have equivalent worst-case time complexities of $O(N \log N)$ (Coffman et al., 1984).

3.3. The Genetic Algorithm

In addition to greedy heuristics like the *first-fit* algorithm, genetic algorithms (GA) can also be used to produce adequate solutions to NP-hard problems. Genetic algorithms are heuristics based on biological evolution that simulate reproduction with variation and selection according to fitness, like that of a true biological population (Eiben & Smith, 2010).

The genetic algorithm created to solve the BPP is implemented in the C# programming language and follows the general structure of a typical genetic algorithm, shown in **Figure 2** (Eiben & Smith, 2010). In this GA, the program iterates through a set number of generations and then halts, reporting the best overall solution.

```
Generate random initial population;
While (not done)
{
     For i=1 to population size
     {
          Select two parents;
          Crossover to produce an offspring;
          Mutate the offspring;
          Insert offspring into new generation;
     }

     Offspring replace parents;
     Report the best solution in the population;
}

Report the best overall solution;
```

Figure 2. The general structure of a genetic algorithm.

3.3.1. Encoding Candidate Solutions

In a GA designed for the bin packing problem, each candidate solution can be represented by a permutation of the elements in the input set.

For each chromosome, the input set is randomly shuffled, creating a random permutation. The question now is how do these permutations apply to the BPP? Since the *first*-fit approximate algorithm is the algorithm being compared to, the *first*-fit algorithm itself will be applied to each permutation. This way, all candidate solutions will undoubtedly be valid.

In short, the two algorithms that will be compared are the *first-fit* algorithm itself and the genetic algorithm, which operates on solutions generated from the *first-fit* algorithm. Because the underlying algorithm in both applications is identical, only the order of the elements in the input set will differ.

Therefore, the goal of the genetic algorithm is to produce optimal permutations of the input set on which the *first-fit* algorithm will be applied. The permutations are generated by randomly shuffling the elements in the input set. An example of the underlying *first-fit* algorithm is shown in **Figure 1**.

A *Chromosome* `class` with data members *permutation* and *fitness* hold each candidate solution's permutation and its associated fitness, respectively. Additionally, the *first-fit* algorithm is applied to each chromosome during its creation, as to promptly calculate its fitness so that it is available throughout the remainder of the program's execution. A *population* array holds every *Chromosome* in the population.

3.3.2. Fitness

In this genetic algorithm, a chromosome's fitness is equal to the number of bins needed to hold the elements in the input set, as defined by the bin packing problem (see **2. The Problem**). Therefore, fitness should be minimized, such that chromosomes with smaller fitnesses are better solutions.

The fitness for each chromosome is also stored within the *Chromosome* `class`. Fitnesses are calculated when each chromosome is created, after the underlying *first-fit* algorithm assigns the input elements to bins. The resulting number of bins is the chromosome's fitness.

At the end of each generation, once the offspring chromosomes replace the parent chromosomes, the chromosome with the smallest fitness is reported as output to the program. This chromosome is also compared to an overall best chromosome kept throughout the program. If the best chromosome in the current population has a smaller fitness than the overall best, the local best chromosome becomes the overall best. At the end of the program's execution, the overall best chromosome and its fitness are reported as output.

3.3.3. Selection

The genetic algorithm uses k-tournament selection to determine which chromosomes in the population will become parents, with $k = 2$ for the problem instances used during testing and during comparison with the *first-fit* algorithm.

To determine a parent, an array of size k of candidate parents is initialized, and chromosomes are randomly chosen from the population and added to the array until it is full. The candidates' fitnesses are then compared, and the chromosome with the smallest fitness becomes the parent.

3.3.4. Crossover

Crossover in the genetic algorithm is accomplished via alternating-position crossover (Larrañaga et al., 1996). In this crossover method, the first element in the first parent chromosome is added to the offspring chromosome. Then, ignoring duplicates, the first element in the second parent chromosome is added to the offspring chromosome. This continues for each element in each parent, with all duplicates ignored. This process is illustrated in **Figure 3**.

$$parent_0 = \{1\ 7\ 8\ 2\ 3\ 4\ 6\ 5\}$$
$$parent_1 = \{2\ 7\ 1\ 5\ 6\ 3\ 4\ 8\}$$

$$offspring = \{1\ 2\ 7\ \sout{7}\ 8\ \sout{1\ 2}\ 5\ 3\ 6\ 4\ \sout{3\ 6\ 4\ 5\ 8}\}$$
$$offspring = \{1\ 2\ 7\ 8\ 5\ 3\ 6\ 4\}$$

Figure 3. An example of alternating-position crossover in a genetic algorithm.

3.3.5. Mutation

After an offspring chromosome has been created, a random swap mutation is performed on that chromosome's permutation encoding. To perform this mutation, two indices in the array holding the permutation are randomly selected, and the contents of the two indices are swapped. This way, each new chromosome that enters the population still maintains some heritability from its parents, while also allowing for the introduction of new traits into the population.

3.3.6. Parameter Values

Parameter values for the genetic algorithm were chosen through trial-and-error, based on results from small test instances. **Figure 4** lists the genetic algorithm's parameters and the values of each.

- Population size: **100**
- Number of generations: **100**
- K-tournament k: **2**
- Probability of crossover: **100%**
- Probability of mutation: **10%**

Figure 4. Chosen parameter values for the genetic algorithm

4. Comparison of Algorithms

In this section, several BPP problem instances are described, and the results of both the genetic algorithm and the *first-fit* algorithm on those instances are stated and compared. For these tests, both of the algorithms have been implemented in the C# programming language and run as console applications.

The algorithms were first tested on three problem instances, each with an input set of 50 randomly generated unique integers and varying bin capacities. Similar groups of sets of size 100 and 500 were also tested. For each instance, the genetic algorithm was executed 50 times.

4.1. Test Group 1: Input sets of size 50

As shown in **Table 1** below, the genetic algorithm outperformed the *first-fit* algorithm in two of the three test problem instances. In both of these instances (sets 50_B and 50_C), the genetic algorithm achieved solutions whose fitnesses are one less than the fitnesses produced by the *first-fit* algorithm. In the other test instance (50_A), both algorithms' best solutions had equivalent fitnesses.

Table 1. Fitness Summary Statistics for Test Group 1.

Input Set	Algorithm	Best Fitness
50_A (capacity: 200)	*First-Fit*	20
	Genetic	20
50_B (capacity: 250)	*First-Fit*	18
	Genetic	17
50_C (capacity: 350)	*First-Fit*	12
	Genetic	11

4.2. Test Group 2: Input sets of size 100

The algorithms produced similar results in the second test group. The genetic algorithm's solutions for problems 100_A and 100_B were better than the solutions produced by the *first-fit* algorithm, by a single bin. The best solutions from the two algorithms for problem 100_C had identical fitnesses. **Table 2** shows these results.

Table 2. Fitness Summary Statistics for Test Group 2.

Input Set	Algorithm	Best Fitness
100_A (capacity: 500)	*First-Fit*	29
	Genetic	28
100_B (capacity: 500)	*First-Fit*	32
	Genetic	31
100_C (capacity: 500)	*First-Fit*	34
	Genetic	34

4.3. Test Group 3: Input sets of size 500

Results for test group three matched the results of the previous two test groups. The genetic algorithm produced a better-by-one solution for two of the problem instances (500_B and 500_C) compared to the *first-fit* algorithm. The two algorithms produced an identical best fitness for problem instance 500_A. The results are shown in **Table 3**.

Table 3. Fitness Summary Statistics for Test Group 3.

Input Set	Algorithm	Best Fitness
500_A (capacity: 1,500)	*First-Fit*	136
	Genetic	136
500_B (capacity: 1,500)	*First-Fit*	134
	Genetic	133
500_C (capacity: 1,500)	*First-Fit*	134
	Genetic	133

4.4. Summary of Results

In each test group, the genetic algorithm produced a better solution than the *first-fit* algorithm for two-thirds of the problem instances in the group. For these instances, solutions obtained by the genetic algorithm had a fitness that was one better than the fitness of the solution produced by the *first-fit* algorithm. This was the case for all instances in which the genetic algorithm produced better solutions.

For the remaining problem instances, the two algorithms produced solutions whose fitnesses were identical. There weren't any cases in which the genetic algorithm produced a final solution that was worse than the solution produced by the *first-fit* algorithm for the same problem instance.

5. Conclusion

From the results collected from the test problem instances described in **4. Comparison of Algorithms**, shown in **Table 1**, **Table 2**, and **Table 3**, it is clear that of the two algorithms, the genetic algorithm managed

to produce superior solutions to those of the *first-fit* algorithm for two-thirds of the problem instances.

In cases in which the solutions produced by the genetic algorithm were better than the solutions produced by the *first-fit* algorithm, the improvement in fitness was minimal. That being said, there was in fact improvement. The degree of improvement did not change with the size of the input set and the bin capacity. Regardless of the input parameters, the genetic algorithm always produced a final solution with a fitness equal to, or one better than the result of the *first-fit* algorithm.

It should be noted that the results of the genetic algorithm are probabilistic, in that each successive execution of the genetic program may produce a different overall best solution. This is dependent on the random initial population, the population size, the randomly selected parents, random mutation, and more. Therefore, there is a chance that the genetic program will not produce the optimal solution for a given input set in a single execution of the program. To increase the chances of producing a superior final solution, subsequent executions of the program should be performed, as demonstrated in this paper.

Though not mentioned in the testing and results, the genetic algorithm's superior solutions were contrasted by its inferior execution time. This can be attributed to the large amount of work required in the genetic algorithm. Much work is required to complete merely one generation, and all of this work must be repeated for each generation. Furthermore, the entire process must be repeated for each subsequent execution of the program.

In conclusion, this genetic algorithm can be used to generate improved BPP solutions to those produced by the *first-fit* algorithm alone.

References

Bin-packing problem. University of Bologna. Department of Electronics, Computer Science, and Systems. Operations Research Group.

Coffman, E.G., M.R. Garey, & D.S. Johnson (1984). *Approximation Algorithms for Bin-Packing – An Updated Summary*. In Algorithm Design for Computer System Design.

Eiben, A.E. & Smith, J.E. (2010). *Introduction to Evolutionary Computing*. Springer. Germany.

Eilon, S. & N. Christofides (1971). *The Loading Problem*. Management Sci. 17, 259-268.

Hung, M.S. & J.R. Brown (1978). *An algorithm for a class of loading problems*. Naval Research Logistics Quarterly 25, 289-297.

Izumi, T., T. Yokomaru, A. Takashashia, & Y. Kajitani (1998). *Computational Complexity Analysis of Set-Bin-Packing Problem*. Special Section on Discrete Mathematics and Its Applications. Tokyo Tech Research Repository.

Johnson, D.S., A. Demers, J.D. Ullman, M.R. Garey, & R.L. Graham (1974). *Worst-Case Performance Bounds for Simple One-Dimensional Packing Algorithms*. SIAM. J. Comput. 3, 4.

Korf, Richard (2002). *A New Algorithm for Optimal Bin Packing*. AAAI-02 Proceedings. Los Angeles, CA.

Larrañaga, P., C.M.H. Kuijpers, M. Poza, & R.H. Murga (1996). *Decomposing Bayesian Networks: Triangulation of the Moral Graph with Genetic Algorithms*. Statistics and Computing.

Malkevitch, Joseph (2004). *Bin Packing*. American Mathematical Society. Feature Column Archive, May 2004.

Schwerin, P. & G. Wäscher (1997). *The Bin-Packing Problem: A Problem Generator and Some Numerical Experiments with FFD Packing and MTP*. Int. Trans. Op Res. 4, 377-389. Elsevier Science Ltd. Great Britain.