A Fuzzy Bacterial Evolutionary Solution for Crisp Three-Dimensional Bin Packing Problems

Dányádi Zsolt, Péter Földesi
Dept. of Logistics and Forwarding, Faculty of Engineering Sciences
Széchenyi István University
Győr, Hungary
danyadi@sze.hu foldesi@sze.hu

László T. Kóczy
Institute of Electrical, Mechanical Engineering and Computer Science, Faculty of Engineering Sciences
Széchenyi István University
Győr, Hungary
koczy@sze.hu

Abstract—This paper presents an evolutionary quasi-solution for a problem commonly occurring in practical logistics, the three-dimensional version of the bin packing problem. The algorithm presented here is a variation of the bacterial evolutionary approach, and utilizes fuzzy logic in the fitness calculation. The goal is to give a useful alternative method to the basic problem, and to demonstrate that the addition of fuzzy logic elements to the fitness function increases the speed of the evolutionary process. The paper first describes the specific problem, then moves on to the details of every key part of the algorithm. Finally, the results from a number of test runs are used to show the general efficiency, and the contrast between the crisp and fuzzy fitness functions. It is clearly shown that the application of fuzzy approach in the fitness function can improve the speed of convergence, so the fuzzy logic can be helpful even in solving crisp problems.

Keywords—bin packing; bacterial evolutionary algorithm; fuzzy logic

I. INTRODUCTION

The general bin packing problem is difficult to solve efficiently due to the fact that this task is NP-hard [14]. Even in its simplest cases, without any additional restrictions, barring some special cases, only approximate solution methods can be given. It is difficult to propose even a polynomial time approximation scheme (PTAS) to this problem, even though polynomial time asymptotic PTAS algorithms are known [13].

The basic problem may have multiple interpretations, the one presented in this paper is utilizing a certain three dimensional space known as the container, by placing objects inside it, in order to cover as much of the container as possible. The restrictions are that neither overlapping of the objects is allowed, nor can they have parts sticking out of the container [4, 5, 7, 8].

In our approach an evolutionary algorithm will be applied. The Bacterial Evolutionary Algorithm (BEA) is a versatile optimization method that imitates phenomena observed in the reproduction of microbial organisms. It is a member of the family of evolutionary algorithms, which means that it uses stochastic mutations and inheritance to produce new candidate solutions to a problem. These candidates are named as individuals, and together they form the population on which the bacterial algorithm operates. The individuals can be broken down to segments known as genes, which, when combined, describe a full solution candidate. The goal of the evolutionary process is to create successive generations, each of them containing better solutions than earlier ones [1, 2].

Applying fuzzy operations to the evaluation of individual fitness in evolutionary algorithms can be beneficial [11, 12]. In one of the implementations of bacterial evolution shown here, fuzzy elements are introduced to the fitness calculations by assigning membership values to every one of the genes. This is done because most of the candidates have redundant genes, which have to be culled in order to form the real solution. The paper presents all important components of the algorithm, and demonstrates its efficiency both in the fuzzy and the non-fuzzy (crisp) versions by applying it on benchmark problems.

II. THE PROBLEM

The three-dimensional bin packing task in our case is limited to rectangular objects. The container and the bins can be defined with three numbers, denoting their dimensions. Thus, the full description of the problem has the following parameters:

- The dimensions of the container
- The dimensions of every type of bin object
- The number of bins of each particular type (size)

The problem itself can take multiple forms, for example to use as few containers as possible to pack a certain set of bins, or to utilize a single container to the fullest extent possible. Although these variations might have a lot in common, the best solutions for them do not necessarily correlate well. The goal of the algorithm here is the latter, to use as much of the
container space as possible, only picking from bins still in the supply. Another restriction is that the edges of the bins have to be either parallel or perpendicular to those of the container, which limits the number of possible orientations, but greatly simplifies calculating when and how much they overlap.

### III. THE BACTERIAL EVOLUTIONARY ALGORITHM

As mentioned before, the bacterial evolutionary algorithm works on principles of reproduction as seen in the case of real bacteria. The method works on a population of individuals, each of which fully describes a solution to the problem. The population is constantly refreshed through cycles known as generations. The individuals in the new generation inherit their parts from older members of the population they were created from. These blocks of heredity are way to model genetic information in biological organisms. The interpretation of what a gene represents varies from one evolutionary algorithm to the other.

Simulating the phenomenon of natural selection makes sure that individuals that are better get to take more part in creating the next generation, by having more offspring than the ones that are less suited to the problem. Also through a process known as horizontal gene transfer, better individuals can also spread parts of themselves within the population, in the hope that these will correlate with better solutions [1, 2].

In order to measure how good a solution is, a scale called “fitness value” is introduced. Members of the population that have a higher fitness value are considered to be better solutions, and they have more chance of producing offspring. Performing a generation of a bacterial evolutionary algorithm consists of the following:

- **Cloning:** In this step copies of individuals are made, which will then be subjected to mutation.
- **Mutation:** This part of the algorithm performs random alterations on the clones.
- **Replacement:** The best clones are then used to replace the parent individuals, if there is any improvement.
- **Gene transfers:** In this process, members of the population that fall into the better portion can copy some of their genome into worse individuals, improving them in the process [1, 2].

Since any individual created by these operations has to represent a solution candidate, special care must be taken to ensure that they do not result in individuals with flawed syntax. For example, the gene transfer step, instead of copying over random bits, has to work with bins as units in order to make sense in the context of the problem. The bin packing problem can be put into evolutionary terms in a number of ways [5, 6, 8, 9, 10], the following interpretation operators are based on these ideas.

#### A. An Individual

The way an individual is stored is relatively simple. Given the description of the problem (the container size and the supply of different types of bins), the following are needed:

- The number and indices of which bins from the supply are used.
- The position and orientation of each of these bins in the container.
- Various other values, such as stored fitness measurements.

The position, orientation, and supply index of every bin is considered as a gene, and they may be changed independently of other bins, however, they are bound in the sense that none of them may be removed or copied without also taking care of the other parts.

#### B. Introducing New Members into the Population

The simplest new solution candidate would be a container having no bins in it, and while it definitely works for this algorithm, randomized individuals are used at the start, and they are also introduced into the population during later generations, to help keeping up genetic diversity. They are created by randomly choosing a number of bins to be placed in them (based on an average), and using the addition operator, which will be described later.

#### C. Cloning and Mutation

This operation has multiple versions, because of the different alterations that an individual can go through. The cloning portion is simply done by copying a solution bin by bin, but the mutation part consists of multiple actions, which include adding new bins to an individual, changing the position of bins already in the container, and removing some of them. The number of these smaller operations within a single mutation is random, with a geometric distribution, determined by an average. The operators presented here are mostly based on common bin packing heuristics [3, 4, 7, 8].

#### D. Adding New Bins to an Individual

Adding a new bin to a solution candidate is a relatively simple task. We choose one from the supply, then find a random position and orientation for it. The orientation is handled by creating a random permutation of its three dimensions, but the positioning requires some help in order to be efficient. From the randomly generated initial place, the bin is moved towards the nearest other bin or a wall of the container, as close as possible. This, however, is only done if the bin is not at a sufficient distance from everything else, so the solution can be built starting from multiple points at once.

The probability of adding new bins is directly proportional to the difference of sum volumes of the existing bins and the total volume of the container:

$$P_{\text{add}} = r_{\text{add}} \frac{V_{\text{container}} - \sum V_{\text{bins}}}{V_{\text{container}}},$$

where:

- $P_{\text{add}}$ is the probability of adding a new bin during the mutation.
- $V_{\text{container}}$ is the total volume of the container.
- $V_{\text{bins}}$ values represent the volumes of each bin used in the current solution.
• \( r_{\text{add}} \) can be used to modify the rate of the addition operations.

E. Moving Bins

The probability of this specific operation depends on the number of bins in the solution:

\[
P_{\text{move}} = r_{\text{move}} \frac{\text{V}_{\text{avg}} \cdot n_{\text{bins}}}{\text{V}_{\text{container}}},
\]

(2)

where:

• \( P_{\text{move}} \) is the probability of moving a random bin inside the solution.
• \( V_{\text{avg}} \) denotes the average volume of the bins found in the supply.
• \( n_{\text{bins}} \) is the number of bins in the current solution.
• \( r_{\text{move}} \) is a constant we can use to set of bin-moving operation.

The chance of choosing a particular bin is inversely proportional to its average distance from the closest other bin or wall along all six directions. If this value is sufficiently small, the probability of moving that bin is zero. This helps keep tightly packed sections the way they are.

Moving bins consists of two parts; first, an offset vector is generated randomly, and if it exceeds a certain number in magnitude, the bin is moved to that exact direction and distance. However, if the offset vector is sufficiently small, the destination will be the nearest side of another bin or wall. This might mean moving in the opposite direction, in case of an overlap. Each component of the offset vector is generated with a uniform distribution in a \([0, \text{offset}_{\text{max}}]\) range.

F. Removing Bins

The removal of a bin from the solution is a straightforward operation. The only questions are how frequently it should be done, and which bins to choose. There is a minimum non-zero chance for removing any of them in a given mutation, but for the most part, the probability is determined by the total volume of bins in the solution. The closer it is to the total volume of the container, the more useless bins there are in it, therefore the higher the probability of bin removal, similarly to the addition operation:

\[
P_{\text{remove}} = r_{\text{remove}} \frac{\sum V_{\text{bins}}}{\text{V}_{\text{container}}},
\]

(3)

where:

• \( P_{\text{remove}} \) is the probability of performing a bin removal.
• \( V_{\text{bins}} \) and \( \text{V}_{\text{container}} \) represent the same values as in (1).
• \( r_{\text{remove}} \) is a constant that can be used to set of the removal operation.

The probability of choosing a particular bin is directly proportional to how much it overlaps with others, measured in volume. If all of them are completely separate, there is no need for removing any bins.

G. Gene Transfers

The purpose of horizontal gene transfers in this case is to make the receiver individual, the one that the genome segment is copied to, better. For this, it makes sense to group parts of the solution that are spatially connected, since the goal here is effective utilization of space. This translates to bins that are within a certain section of the container. The section is chosen randomly, and any bin from the source individual that is at least partially within this section is placed in the same place in the destination individual.

To make room for the new bins, any of them in the destination individual that is not completely outside the section is removed, as well as any bin that is outside and shares the same index in the supply of bins with one that is about to be copied over. This prior correction makes sure that there are no irregular solutions along the way. The gene transfer may introduce additional overlaps, which could be avoided by only copying bins that are completely inside the section, and only keeping ones that are completely outside of it. This, however, would introduce problems such as often creating holes in the destination individual, by cleaning a portion of the container space, and not adding anything in return.

As an illustration, in Fig. 1, if we are looking at the source individual, bins 1 and 2 are the ones that will be selected for transfer. However, if this were a destination individual, bin 3 would be the only one of the three bins still included after the transfer.

IV. THE FITNESS FUNCTION

Since it is the fitness function that drives the entire evolutionary process towards better individuals, it is important to choose it appropriately. In our case, because there are fewer restrictions on the candidate solutions than there are in the final one (due to the permitted overlapping), two separate measures are used. The first one represents the goodness of the solution after temporarily correcting for overlapping, either by repositioning or deleting the problematic bins from it. The second one, in contrast, calculates the usefulness of the candidate before these corrections. There is a difference in priority between these two measures; the first one is more important for determining the viability of an individual, but due
to its discrete nature, the other, secondary fitness value is needed as well.

The primary fitness value, which represents the viability of the actual, corrected solution, is calculated after a series of operations that remove any remaining overlapping between the bins. Once this is done, a summation of the selected bin volumes gives an indicator as to how well the container space is utilized:

$$\text{fitness}_{\text{primary}} = \frac{\sum V_{\text{remaining, bins}}}{V_{\text{container}}},$$  \hspace{1cm} (4)

where the $V_{\text{remaining, bins}}$ values represent the volumes of all the bins that are still in the solution after the elimination of overlaps.

The elimination process itself is stochastic; in each cycle, out of the bins that have any overlapping, one is chosen randomly. The probability of removal is directly proportional to the number of how many others the particular bin overlaps with. All individuals are evaluated in every generation, not only after modifications (mutations and gene transfers), so that even if there is no change, a different random elimination order might yield better coverage, which updates the primary fitness value as well.

Calculating the secondary fitness value has two separate versions; one that uses fuzzy membership functions to describe bins that are fully or partially covered by others, the crisp version, on the other hand, treats all bins as full members of the solution, and handles the overlaps separately.

A. Crisp Fitness

The secondary fitness value, in the crisp case, imposes a penalty on any overlapping between bins (or pairs of bins, to be more precise), and is calculated in the following way:

$$\text{fitness}_{\text{secondary}} = \frac{\sum (V_{\text{bins}} - C \sum V_{\text{overlaps}})}{V_{\text{container}}},$$  \hspace{1cm} (5)

where $V_{\text{bins}}$ are the volumes of all the bins in the solution, $V_{\text{container}}$ is the total volume of the container, the $V_{\text{overlaps}}$ values represent the common volume for every overlapping pair of bins, and the constant $C$ might be any real number above 2. The higher $C$ is, the less lenient the algorithm is towards bins that share common space. The value used in the test runs was 8.

B. Fuzzy Fitness

The fuzzy variant of the secondary fitness function is somewhat similar to the crisp version, but it deals with overlapping in a different way:

$$\text{fitness}_{\text{secondary}} = \frac{\sum (F_{\text{bin}} \cdot \Delta V_{\text{bin}})}{V_{\text{container}}},$$  \hspace{1cm} (6)

where $F_{\text{bin}}$ is the measure of how much of the volume of a bin is not covered by another bin, a ratio between 0 and 1. Here, it is used as an upper estimate for a membership function, expressing to what degree the bin can be considered a part of the solution. A conjunction of all the $F_{\text{bin}}$ values for a bin (from each of its overlaps) is taken as the final membership value, an operation that in its simplest version is performed by taking the minimum of all values. The membership value multiplies the volume of the corresponding bin, $V_{\text{bin}}$, and the sum of these values is divided by that of the container, $V_{\text{container}}$, to acquire the secondary fitness function.

In both cases, fitness comparisons between individuals are done by first taking the primary value into consideration, and the secondary values are only used if there is no significant difference in the primary ones (this means more than a sufficiently small non-zero number, to account for floating point imprecision).

V. THE BENCHMARK PROBLEMS AND PARAMETERS USED FOR COMPARISON

The specific problems were generated based on problem classes, similarly to [15]:

- **Class 1**: the container size is 10 in each dimension, and bin sizes are uniformly random in $[1, 5]$.
- **Class 2**: the container size is 30 in each dimension, and bin sizes are uniformly random in $[1, 10]$.
- **Class 3**: the container size is 100 in each dimension, and bin sizes are uniformly random in $[1, 20]$.

The population size for the bacterial algorithms was 20. The constants $t_{\text{add}}$, $t_{\text{move}}$, and $t_{\text{remove}}$ were set to 0.25, 0.5 and 0.25 respectively, and the $r_{\text{max}}$ value for the bin moving operator was set to half the container size (5, 15 and 50 for the three problem classes). The populations underwent a total of 5 gene transfers in each generation, and a new individual was placed in the population when none of the fitness values improved over the last 10 generations. The best member of the population at any given time was immune to replacement (except by even better clones), which approach is known in the literature as “elitist strategy”. It was put into the algorithm to ensure that the best solution never gets lost because of alterations to the individual. The generation limit was 10000 for class 1, 3000 for class 2, and 1000 for class 3 problems.

The benchmark solution for each problem used in the comparisons was generated using the branch-and-bound algorithm [16]. There were a total of 5 instances generated for every problem class, in every instance, 10 executions of both evolutionary algorithms were then compared to the benchmark solution in terms of total bin volume percentage. These percentages were first averaged between the 10 runs, then the 5 instances, to calculate the average fill ratios for each problem class, as seen in (Table 1).

VI. OBSERVED RESULTS

Although there is a great amount of variation in the performance of the algorithms due to the randomness, the general tendency is that the test runs that utilized the fuzzy descriptors in the secondary fitness function managed to reach a higher fill rate than their crisp counterparts. The difference for class 2 problems is not that significant, but it is much more pronounced for problems of classes 1 and 3 (see Table 1).
<table>
<thead>
<tr>
<th>Problem Class</th>
<th>Percentage-Crisp</th>
<th>Percentage-Fuzzy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>65%</td>
<td>70%</td>
</tr>
<tr>
<td>2</td>
<td>68%</td>
<td>69%</td>
</tr>
<tr>
<td>3</td>
<td>71%</td>
<td>74%</td>
</tr>
</tbody>
</table>

VII. CONCLUSIONS AND FUTURE IMPROVEMENTS

In general, the inclusion of the fuzzy membership functions describing the bins was shown to be beneficial when it came to the effectiveness of the algorithm. Although the fill ratios achieved within the generation limit are smaller than the benchmark values, the difference between the fuzzy and non-fuzzy approaches is clear in terms of results.

Allowing overlapping bins during the evolutionary process can keep potential solutions that are not fully acceptable in their current form, and they may have a chance of becoming suitable solutions after a minor mutation. The secondary fitness value using the fuzzy descriptors seems to predict these opportunities better, which may be behind the increase in performance.

One can make several alterations and extensions to the core methods of the bacterial algorithm implemented here, namely in the mutation and gene transfer operations. New heuristics might improve their effectiveness, making them more suited to this specific problem of bin packing. Also, the fitness function may also undergo changes, to reflect different packing heuristics than the ones used here. Finally, aspects of other evolutionary methods, such as crossover-based genetic variants might also be included as an extension to the bacterial one [12].

ACKNOWLEDGEMENT

This work is partially supported by the project TAMOP 421B, two Széchenyi István University Main Research Direction Grants, further National Scientific Research Fund Grant OTKA K75711.