Optimizing Highly Constrained Truck Loadings Using a Self-Adaptive Genetic Algorithm

Sander van Rijn
LIACS, Leiden University
Niels Bohrweg 1, 2333 CA
Leiden, The Netherlands
Email: svr003@gmail.com

Edgar Reehuis
DENC Netherlands B.V.
Slochterenlaan 12, 1405 AM
Bussum, The Netherlands
Email: ereehuis@denc.nl

Michael Emmerich
Natural Computing Group
LIACS, Leiden University
Niels Bohrweg 1, 2333 CA
Leiden, The Netherlands
Email: m.t.m.emmerich@liacs.leidenuniv.nl

Thomas Bäck
Natural Computing Group
LIACS, Leiden University
Niels Bohrweg 1, 2333 CA
Leiden, The Netherlands
Email: t.h.w.baeck@liacs.leidenuniv.nl

***Abstract*—Most research into the Container Loading problem has been done on theoretical problem sets and while taking one or two constraints into account. In this paper we discuss the successful implementation of a self-adaptive Genetic Algorithm applying only mutation, with a variable mutation rate. This is applied to a real-world problem with actual problem instances from industry. We introduce an abstract, indirect representation for the considered loadings together with two mutation strategies. Solutions of these different strategies are compared with each other, a static mutation rate GA, and with solutions created by human planners as used in industry, for a set of over 500 real-world problem instances. Furthermore, we examine how our automated results compare to those generated by experienced human planners, showing that they are valid loadings and match fitness values.**

I. Introduction

The *Container Loading* and *Bin-Packing Problem* (CLP and BPP) are closely related optimization problems, requiring the packing of three-dimensional objects into a larger three-dimensional space or set of such spaces respectively. Volume efficiency is the dominant, and often only measure of solution quality in most literature concerning these problems, as has been noted in [2], [3].

Rather than studying instances of these problems, we examine a solver using a new, indirect solution representation for an actual real-world problem, namely loading products for hardware stores into truck trailers. Products range from fragile glass shower cabins to 500 kg pallets of wooden floorboards and 5 meter long pipes. The selection of products that have to be loaded into the trailer has been determined in advance.

A defining requirement for our real-world problem is that the products are placed into *stacks* that can easily be moved by a forklift truck. This requirement in combination with such a *strongly heterogeneous* set of products makes the methods described in [1], [4], [5], [7], [8], [9], [11], [12], [13] difficult to apply in practice, as they rely heavily on three-dimensional freedom in the placement of the products.

978-1-4799-7492-4/15/$31.00

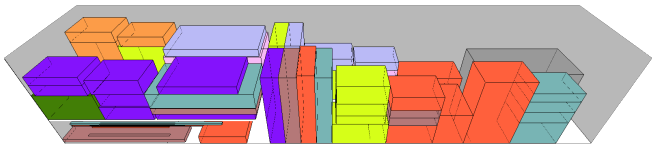

Fig. 1: *Example Loading*. Example of a container loaded with boxes for multiple clients. Each color represents a different client to which boxes have to be delivered.

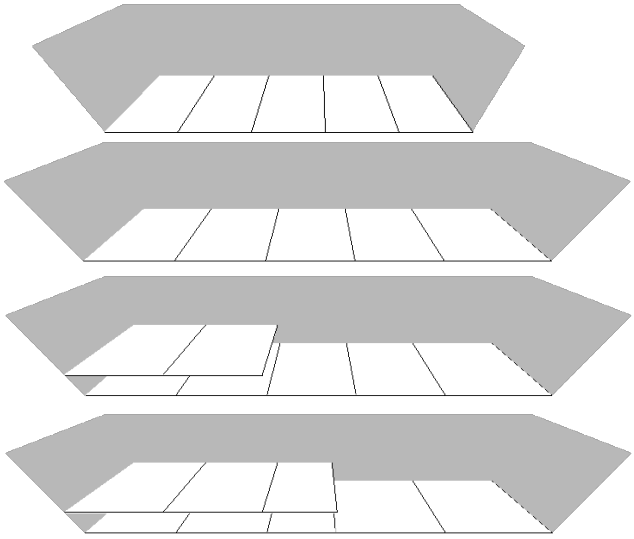

Fig. 2: *Container Types*. The four container types for which loadings are to be generated. The front of a container is at the left. The vertical lines divide the trailer into multiple zones that are used to indicate the intended location of a box for the loading process. A so-called *bridge* is present in the last two trailers.

Described in Section II, we encounter *six* constraints and

objectives, as identified by Bortfeldt and Wäscher [3], concerning *weight distribution, stacking constraints, positioning, stability, orientation*, and *complexity*, explained in detail in Sections II-A and II-B. The container types vary in size and may consist of several separate sections, see Fig. 2.

In Section III we elaborate on the abstract, indirect solution representation created for this problem and the self-adaptive Genetic Algorithm used. Furthermore, we explain the fitness function used in our experiments, which was provided by industry. The setup of our experiments is explained in Section IV, with some analysis of the results in Section IV-B. Further discussion of the results and suggestions for future research can be found in Section V.

II. Problem Definition

We examine the real-world case of loading truck trailers with products of various sizes to be supplied to hardware stores. A single truck may deliver products to up to 25 different clients. The products are all loaded, and unloaded from one of four different types of trailers from the sides by a forklift truck. We will refer to the trailer as a *container*, to the products as *boxes*, and to a configuration of boxes in a container as a *loading*. In a loading, safety and convenience are to be considered via objectives and constraints. An example of a loading can be seen in Fig. 1. The goal is to create a loading using a given container and set of boxes, such that the boxes can be transported as safely and quickly as possible to their intended destinations.

Containers can include a so-called *bridge*, a raised area at the front providing additional surface area over boxes that are fragile. Fig. 2 shows the four different types of containers available. The container is effectively divided into three sub-containers when such a bridge is present, but as interaction is possible between these sub-containers we cannot examine them separately.

Boxes can vary greatly in dimensions and weight. This is referred to as a *strongly heterogeneous* assortment. Boxes are categorized in *product groups* that determine what can and cannot be stacked on top of each other. A *problem instance* consists of a (strongly heterogeneous) set of boxes, usually for multiple different clients, that have to be delivered on a single trip. The number of trips required, which clients should be visited on a single trip and the optimal route along all clients, is determined in advance, externally.

Loading is done in terms of *stacks* that can be loaded or unloaded in a single action, similar to [6], in which a set of disjunctive box towers is generated. Methods producing tight loadings, do not take this requirement into account. Instead, they produce loadings that consist of layers of boxes, and are therefore not a possible solution within the considered problem class.

As boxes in a single container are delivered to multiple clients, all constraints and objectives have to be checked for *all* intermediate sections of the trip. A proper loading has to conform to all constraints and objectives, not only at the start of a trip, but also after the boxes for each client have been unloaded. Alternatively, the boxes would have to be reorganized after each delivery.

A. Hard Constraints

There are hard constraints that limit what boxes can be stacked directly on top of each other. These constraints are by *product type* and by *weight*. Most boxes may not be stacked on top of fragile boxes, and the weight of a box may rarely be more than that of the box below it. These are implemented as hard constraints: If they are violated, the loading is invalid.

B. Penalties

Primarily, a loading should fit all boxes intended for that trip into the container. If this is not possible, all boxes for the client in question will be left out. These then have to be added to another container, disrupting the scheduled delivery routes, which is undesirable.

The generated loadings have to be *safe* to drive with. This concerns both safety for the driver and for the transported boxes. For the driver's safety, *weight distribution* of the boxes in the container is the determining factor. If one side is far heavier than the other, this will impair the handling of the truck.

Safety for the boxes is determined by how they are stacked in the container. Boxes that can slide forward when braking are likely to fall and sustain damage. The general shape of the loaded boxes in the container should therefore be like a triangle: High at the front, low at the back.

The remaining penalties concern convenience when unloading the boxes. Some clients have a strong preference for the side that their boxes should be loaded on, usually because of limited space when unloading. If this preference is then violated, unloading will become more challenging, taking up valuable extra time.

How boxes are ordered within stacks is also taken into account. If boxes for client A are covered by boxes for another client B when delivering at A, the whole stack has to be taken out of the trailer first. The boxes for client B are then placed back into the container. These actions take time and are therefore undesirable, but often unavoidable due to the hard stacking constraints.

III. Approach

Driven by the complexity of the task at hand, a heuristic method was chosen to tackle it: A discrete representation Evolutionary Algorithm, or Genetic Algorithm (GA).

In order to use a GA for this problem, a representation for the loadings had to be constructed. The details of our representation can be found in Section III-A. Using this representation, the GA as specified in Section III-B was implemented. Finally, we shortly discuss the fitness function, produced by industry, that was used by the GA in applying its selection operator in Section III-C.

A. Representation

The solution representation was devised by studying the steps required to create a loading manually. A loading can be represented by a set of these steps. Each step contains at least two pieces of information: *Which box* is planned, and *where* this box is planned.

As sizes of the boxes are listed in centimeters, we use milimeter precision to prevent rounding errors during calculation. Using 3D-coordinates in millimeter precision, however, for containers of size 13,600 × 2,500 × 2,000 mm would result in a total of 6.8×10^{10} possible coordinates. By using abstract, relative coordinates instead, this number is greatly reduced without excluding possible valid loadings.

A representation of a loading has to:

- Deterministically construct a loading,
- Allow enough degrees of freedom for the GA to explore the fitness landscape.

First we divide the container into a set of *areas*

$$A = \begin{cases} \{\mathrm{L}, \mathrm{R}\} & \text{without a bridge} \\ \{\mathrm{L_u}, \mathrm{L_o}, \mathrm{L_b}, \mathrm{R_u}, \mathrm{R_o}, \mathrm{R_b}\} & \text{with a bridge} \end{cases}.$$

L and *R* stand for *left* and *right*, and the subscripts *u, o, b* refer to *under, on*, and *behind* the bridge respectively. This first separation of the container into 2 or 6 areas allows a single, small mutation to quickly switch between left and right, and the parts of the container in the presence of a bridge. Each of these parts limits which boxes can be placed by maximum height.

Because all boxes are loaded and unloaded from the side, there is no benefit to gain from specifying the exact position along the width of the container, i.e., the short side. Boxes may end up in the middle of the container if there is no more space left-over along the sides. This means that only the position along the length of the container has to be specified, with a location at the front of the container always being preferable to one at the back. We do this in terms of a *stack index* s.

Definition III.1. *Representation*
Given a set of n boxes B, a set of areas A in the container and a set of stack indices S, a loading L of n boxes is represented by a set of triples

$$L = \{\{b_1, a_1, s_1\}, \{b_2, a_2, s_2\}, \ldots, \{b_n, a_n, s_n\}\},$$

where $b_i \in B$ is a box from the set of boxes to be planned, $a_i \in A$ is one of the possible areas in the container, $s_i \in S$ is the index of the intended stack and $\forall b_i, b_j \in B \mid b_i = b_j$ **iff** $i = j$, i.e., all boxes are unique. We will refer to a triple $\{b_i, s_i, a_i\}$ as a *step*.

A loading will have 6 areas times 20 stacks per area in a worst-case scenario, resulting in 120 possible coordinates. This reduces the complexity by a factor of $5 \cdot 10^8$ compared to specifying coordinates in millimeter precision. Because there is no physical limit on the number of stacks in an area, a value of 20 was chosen based on the highest occurring value from a set of loadings created by human planners.

Algorithm 1 shows how a loading is built from the used indirect representation. The representation does not directly describe the resulting loading: If a box does not fit at the specified location, all other stacks in that area are tried, and eventually all stacks in the other areas can be attempted. The algorithm constructs the loading from the representation, one step $\{b, a, s\}$ at a time. In lines 4–11, if stack index s matches a stack currently in a, we try to place box b on top of this stack. Otherwise, in lines 12–20, if stack index s is greater than the number of stacks currently in area a, we try to place box b as a new stack in area a. In both cases, if the box fits at that location, we place it and update the step with s and a.

When a box does not fit at the given location, lines 10 and 18 make s iterate over all other valid stack indices, and the above process is repeated for each index until the box fits. Once all stacks in the area a have been tried, in line 22 a new area is selected, and per area, s iterates over all stacks in that area. If no fit was found after all stack indices for all areas have been checked, the box b cannot be placed and the algorithm continues with the next step $\{b, a, s\}$.

This placement algorithm is essential for making sure that the GA converges in a reasonable amount of time. As mutations are likely to produce *invalid* steps $\{b, a, s\}$ in the abstract, indirect representation, the greedy loading construction is used

Algorithm 1 Building a Loading from a Representation

```
1:  for step {b, a, s} ∈ L do
2:      a', s' ← a, s // Make a local copy
3:      while box not placed and not all areas tried do
4:          if stack s' exists in a' then
5:              if b fits at (a', s') then
6:                  placeAt(b, a', s')
7:                  a, s ← a', s' // Copy locals back into L
8:                  break while
9:              else
10:                 s' ← s' + 1
11:             end if
12:         else
13:             if b fits at numStacks(a') as a new stack then
14:                 placeAt(b, a', s')
15:                 a, s ← a', s' // Copy locals back into L
16:                 break while
17:             else
18:                 s' ← 0
19:             end if
20:         end if
21:         if all options for s' in a' have been tried then
22:             a' ← next(a') // Next area from A
23:         end if
24:     end while
25: end for
```

Algorithm 2 (μ, λ)-self-adaptive GA

```
t ← 0
P^(0) ← generate μ individuals r⃗_1, ..., r⃗_μ, randomly
while not terminate do
    for i = 1 to λ do // Create λ offspring
        (L_i, p_m,i) = r⃗_i ← copy(random choice from P^(t))
        p_m,i ← mutate_p(p_m,i)      // Update mutation rate
        L_i ← mutate_L(L_i, p_m,i) // Update steps in loading
                                    // with mutation rate p_m,i
        f_i ← evaluate(L_i)
    end for
    P^(t+1) ← {r⃗_1:λ, ..., r⃗_μ:λ}, select μ best from λ total
    t ← t + 1
end while
```

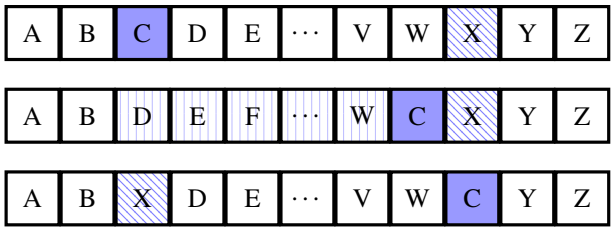

Fig. 3: *Mutation Example*. Demonstration of *insert* and *swap* mutation operators respectively. *Top*: A loading L consisting of genes A – Z. *Middle*: Loading L after gene **C** was *inserted* before gene **X**. *Bottom*: Loading L after gene **C** was *swapped* with gene **X**.

to keep the GA searching with valid loadings. Storing the found coordinates back into the representation is done in lines 7 and 15 to omit redoing all the searching that the build algorithm already performed, in future iterations.

B. Self-Adaptive Genetic Algorithm

A GA commonly uses both mutation and crossover operations to evolve the representations into the final solution, but using only a mutation operator also provides enough freedom to explore the search space. Algorithm 2 shows the general evaluation loop of the GA used in this paper. In a single evolution cycle, λ offspring are generated by mutating one of the μ parents. The μ best of the λ generated form the parents for the next generation.

We compare a fixed mutation rate and a self-adaptive mutation rate approach, referred to as *SA3* by Kruisselbrink *et al.* [10]. The SA3 strategy defines the mutation rate as

$$p_{\text{m}} = p_{\text{m,sa}} + p_{\text{m,min}}, \tag{1}$$

where

$$p_{\text{m,min}} = \frac{1}{n} \tag{2}$$

is the lower bound. The self-adaptive part $p_{\text{m,sa}}$ of the mutation rate is updated according to the rule

$$p'_{\text{m,sa}} = \min\left(\frac{1}{2}, \frac{1}{1 + \frac{1-p_{\text{m,sa}}}{p_{\text{m,sa}}} \cdot \exp(\gamma \cdot \mathcal{N}(0,1))}\right). \tag{3}$$

The parameter γ was empirically defined as

$$\gamma = 0.22. \tag{4}$$

The mutation rate p_{m} will be initialized to 0.2 through initializing $p_{\text{m,sa}}$ to

$$p_{\text{m,sa}} = 0.2 - p_{\text{m,min}}. \tag{5}$$

An individual $\vec{r}$ is defined as a loading L for the static mutation-rate GA, and as a pair

$$\vec{r} = (L, p_{\text{m}}) \tag{6}$$

for the self-adaptive GA, consisting of both a loading L and a mutation rate p_{m}. During the mutation phase of the self-adaptive GA, p_{m} is mutated first according to Eq. 3, followed by the mutation of L, according to p_{m}.

Because our representation combines a permutation with integer parameters, a regular bit flip mutation would not produce the required results. Instead, the mutation rate is used to determine if a *step* has to be mutated. When this is the case, there is a $\frac{1}{3}$-chance for each part of the step to be mutated. When the box of a step is mutated, the entire step is *swapped with*, or *inserted before*, another randomly chosen step. An illustration of the different mutations can be seen in Fig. 3. The random choice of this step is done independent of the mutation rate. The area and stack coordinates are new, randomly generated from the ranges $[1, 2]$ or $[1, 6]$, and $[1, 20]$ respectively.

Early experiments indicated that the self-adaptive mutation rate occasionally prematurely converges to $\frac{1}{n}$, causing stagnation, even before spending 50 percent of the evaluation budget. To prevent this stagnation, we reset the mutation rate p of all individuals in the next generation to the initial value of 0.2 if no improvement has been found in the last 10 percent of the total evaluation budget, since the last improvement or last stagnation. This forces more aggressive mutation and encourages further exploration of the search space.

C. Fitness Function

The different objectives that have to be taken into account by the GA are combined into a single fitness function. For this research, a function provided by industry was adapted slightly. The final penalty is the sum of seven *penalties*. Penalties are calculated for each intermediate section along the route. A violation that remains in the loading between multiple stops is therefore counted multiple times. This causes a violation concerning boxes that are unloaded at the first stop, to have a lower total penalty than a similar violation between boxes that are not unloaded until the last stop.

To evaluate client-related conveniences, the *ClientSide* penalty indicates how often boxes for a client are not on the preferred side. This includes the boxes being spread out over both sides of the trailer without a preference being present. Clients for which the total weight of all boxes exceeds 4,000 kg are exempt from this penalty, because unloading that weight from a single side would destabilize the container. This penalty was adapted slightly to include a small additional penalty for the amount of boxes violating the preference, in order to encourage convergence when a mutation towards a better solution occurs. Additionally, whenever the order of boxes does not match the order in which they are to be unloaded, a *ClientOrder* penalty is given.

Safety is evaluated indirectly by penalizing potentially dangerous situations. Having the overall shape of the loading approximating a triangle, as explained in Section II-B, is done to decrease the risk of boxes sliding. The corresponding penalties are calculated based on the height difference between stacks. When the forward stack is taller, a *StackPlus* penalty is added, otherwise a higher *StackMinus* penalty is given. Together, they make up the *StackPattern* penalty. For stacks consisting of more than 4 boxes, a *StackHeight* penalty is added. Another *WeightBalance* penalty is added depending on the weight distribution in the container. As roads tend to have a slightly convex shape, the penalty for a heavier left side is lower than for a heavier right side, when driving on the right side of the road.

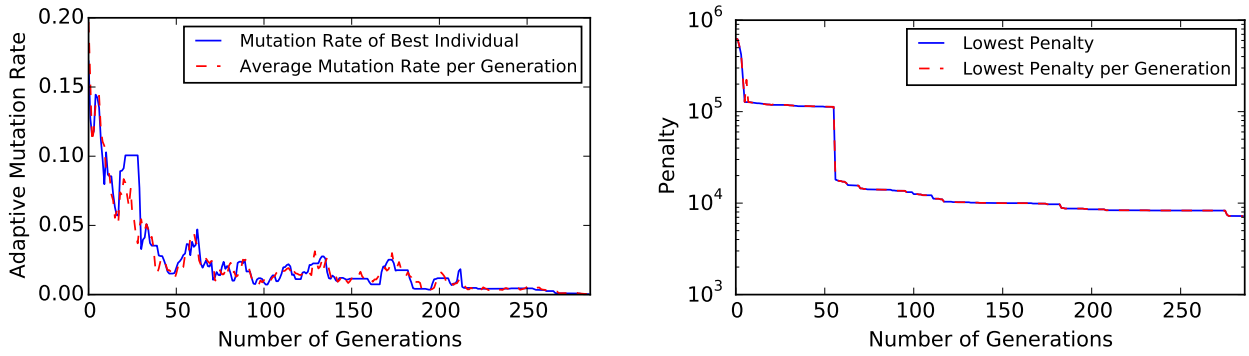

Fig. 4: *Penalty and Mutation Rate Development*. Development of the penalty and self-adaptive mutation rate for a typical problem instance using the self-adaptive GA, with a (5,35) strategy and 50/50 mutation operator. *Left*: The adaptive mutation rate $p_{\mathrm{m,sa}}$ over time. *Right*: Penalty of the constructed loading over time. Penalty values are plotted on a log scale.

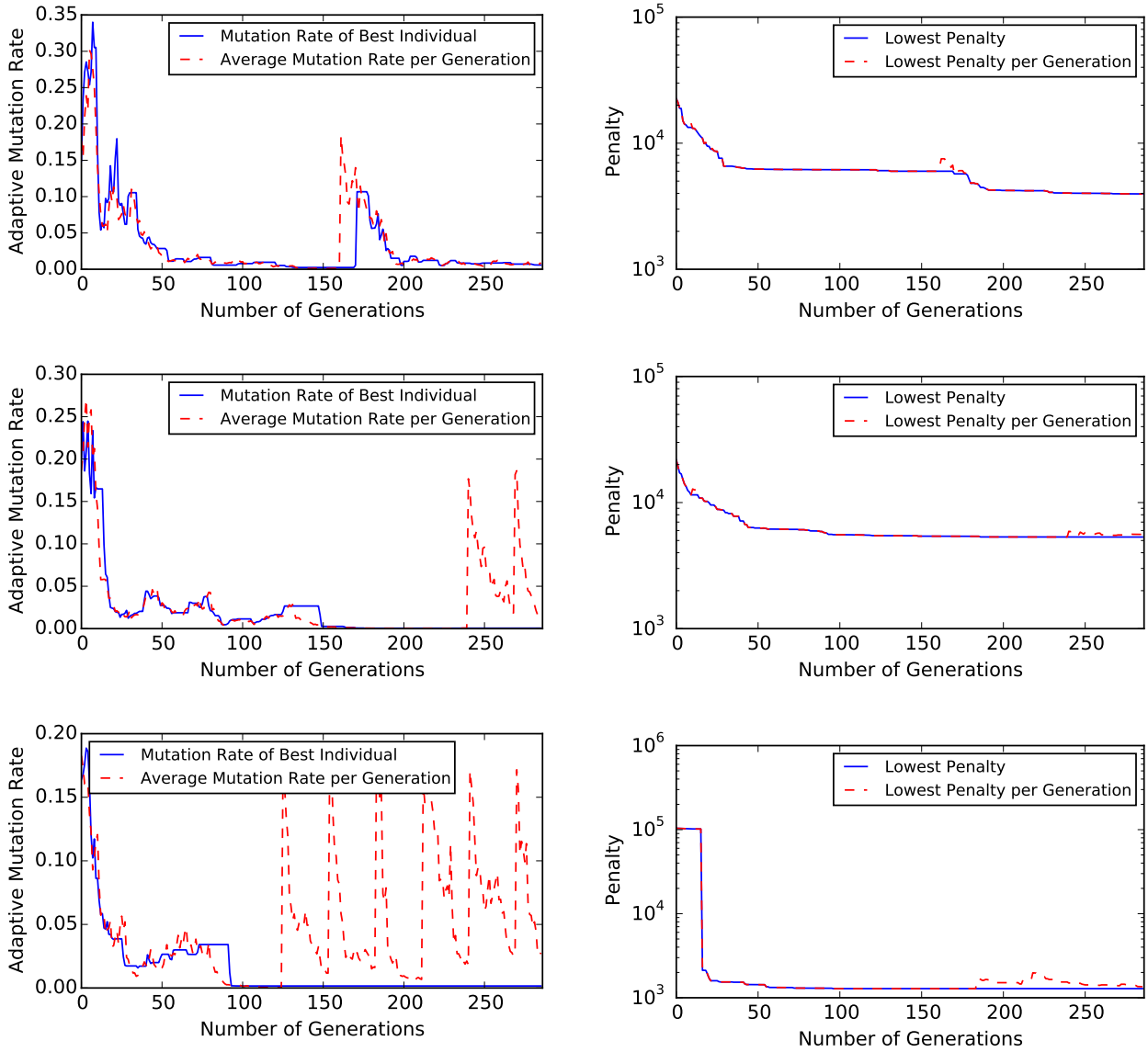

Fig. 5: *Influence of Mutation Rate Reset*. Development of the self-adaptive mutation rate for runs in which one or multiple resets occurred. *Top row*: Mutation rate is reset once, improving the final result. *Middle row*: Consecutive resets towards the end, when a low penalty has been reached. *Bottom row*: Continuous resets starting before 50% of the evaluation budget has been spent, likely due to faster than expected convergence.

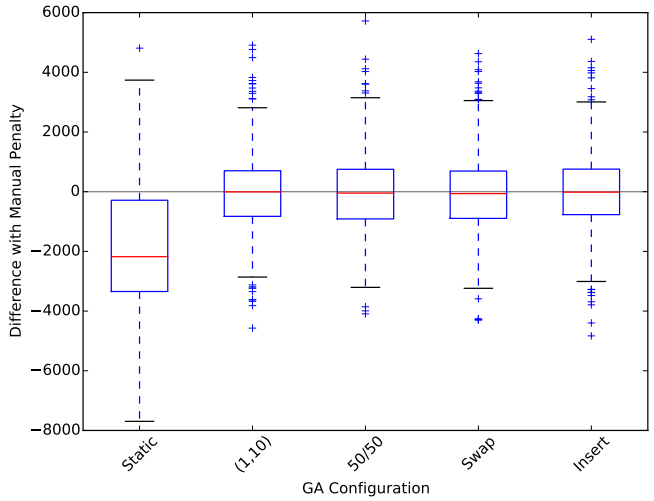

Fig. 6: *Mutation Strategy Comparison*. Box plot of the GA generated penalty values, excluding penalty for boxes that could not be placed, subtracted from the manual results. A positive result indicates the GA outperforming the manual planning.

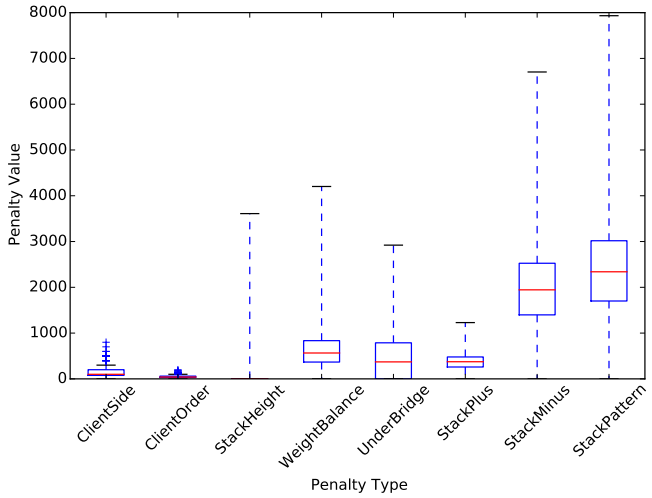

Fig. 7: *Penalty Composition*. Box plot of the penalty value per objective for the self-adaptive, (5,35), 50/50 strategy. The StackPattern penalty is the sum of the StackPlus and StackMinus penalties.

TABLE I: *Boxes Not Placed.* The number of runs resulting in a loading without all boxes being placed, and the percentage of the 32486 boxes in total that could not be placed, for each of the GA configurations.

	Static	(1,10)	50/50	Insert	Swap
Incomplete Loadings	247	76	76	80	76
% Boxes Not Placed	1.481	0.268	0.277	0.299	0.268

Because the area under the bridge is intended for long, fragile and small boxes, an artificial *UnderBridge* penalty was added to discourage placement of heavy, large boxes under the bridge. As these larger boxes could not receive StackPattern or StackHeight penalties under the bridge, the GA would place as many boxes there as possible, resulting in the small and fragile boxes ending up spread out through the container.

Because the penalties for these different objectives have not been expressed in a common unit, they have no intrinsic meaning. Common penalty values for a manually created loading can be found between 10^3 and 10^5. For every box that does not fit in the container, an additional penalty of 10^5 is added, as not delivering a box is the least desirable possibility.

IV. Experiments

We perform a number of experiments to compare the different available options for this GA. A fixed mutation rate is compared with the self-adaptive mutation rate strategy. Furthermore, values (1,10) and (5,35) for (μ, λ) of the GA are evaluated, and a comparison is made between three different mutation strategies: *Swap, insert* and a *50/50* hybrid mutation strategy that randomly picks between the first two. All options are tested with respect to the default *self-adaptive, (5,35), 50/50* configuration. This gives a set of 5 configurations to be tested.

In the experiments we use a set of **528** problem instances for which a solution by a human planner is available. Combined, this set consists of **32486** boxes to be placed into a container. For each configuration the evaluation budget is $10,000$ evaluations; $1,000$ or 286 generations for the (1,10) and (5,35) strategies respectively. With 5 configurations of the GA to run for each problem instances, this means a total of $\mathbf{5 \times 528}$ runs. Combined with the manually created solutions, we obtain 6 possible loadings for each problem instance with the corresponding total and objective specific penalties.

The experiments are performed on several desktop computers. The least powerful PC is equipped with an Intel Core i3 dual-core processor with 4 GB RAM, while the software is written in *Python*. On average, a single experiment of $10,000$ fitness evaluations takes around 10 minutes to complete. This is comparable to the time required by a human planner for these problem instances.

A. Behavior

Figure 4 shows the behavior of the self-adaptive GA for a typical problem instance. Initially, the mutation rate offset $p_{\text{m,sa}}$ increases slightly, but as time progresses, it drops further towards zero. Occasionally, the mutation rate will increase again, characterizing a properly functioning self-adaptive GA.

The development of the penalty for a typical run can be separated into two parts: The *fitting* and *optimizing* phase. For the first 60 generations in the run seen in Fig. 4, not all boxes could be fit in the container, indicated by a penalty of more than 10^5. During this phase, fitting an additional box into the container is always preferred over reducing penalties of an existing loading. As the loading becomes more and more optimized during this fitting phase, the volume of spaces between stacks and boxes decreases, allowing for more boxes to fit into the container. Placing a new box into the container somewhere generally causes the loading to be shuffled, increasing the penalties.

TABLE II: *Improvements per Penalty.* Number of runs per GA configuration in which the GA performed better than the manually generated loading. Note that runs where both loadings resulted in an equal penalty are not included in these results. The numbers in bold indicate the best performance for that penalty.

Objective	Static	(1,10)	50/50	Insert	Swap
ClientSide	119	203	208	201	**219**
ClientOrder	**122**	115	109	95	108
StackHeight	39	**42**	41	37	41
WeightBalance	231	224	**234**	226	209
UnderBridge	81	163	165	164	**173**
StackPlus	146	181	**210**	173	188
StackMinus	205	248	258	250	**263**
StackPattern	185	241	**259**	234	242
Average	141	177	**185**	172	180

Once all boxes fit into the container, the optimizing phase begins. Individuals that leave a box out again are least likely to survive to the next generation, and the GA can focus on reducing the remaining penalties. Figure 7 illustrates which penalties are most decisive in this convergence process, namely *StackPattern* and *WeightBalance*.

The differing outcomes of resetting the mutation rate can be seen in Fig. 5. Ideally, resetting the mutation rate makes the GA continue to explore when stagnating prematurely. This behavior is illustrated in the top row of Fig. 5, where stagnation occurred after approximately 130 generations. The reset is clearly visible by the spike in the average mutation rate per generation after 160 generations. A small increase in the average fitness can also be seen around this point. A new optimum is found soon after, allowing the GA to continue without further resets.

Shown in the middle row of Fig. 5 is the behavior that the GA no longer converges in the final 30 percent of the given evaluations. In these cases, it is unlikely that the GA resumes converging, causing the mutation rate to reset multiple times without improvements. The reset of the mutation rate is again marked by a rise in the value of the best fitness per generation, as elitism is not allowed. However, convergence to the best value found so far does not occur, resulting in consecutive resets through the rest of the process. Note that the mutation rate is reset before it converges back to its former value. This behavior is very common whenever resets occur.

In problem instances consisting of few boxes, behavior as in the bottom row of Fig. 5 is seen. After early convergence to a low penalty value within the first half of the given evaluation budget, no mutation manages to find an improvement, resulting in mutation rate being reset continuously throughout the rest of the process. Problem instances with a small number of boxes are especially likely to display this behavior.

B. Results

Figure 6 shows the distribution of the penalty for the result of each GA configuration, subtracted from the penalty for the manual loading. The static $\frac{1}{n}$-mutation rate clearly performs significantly worse than the other options that all use the self-adaptive strategy. Close to 75 percent of the runs result in a higher penalty than the manual loading. In contrast, the average of the difference between the penalties for the other strategies is close to zero.

TABLE III: *Stagnations.* Data concerning the number of times a run stagnated. *Stagnating Runs* is the amount of runs in which stagnation occurred. *Stagnations/Run* is the average number of times a stagnation occurred.

Strategy	Stagnating Runs	Stagnations/Run
Static	110	3.0
(1,10)	418	3.5
50/50	308	3.1
Insert	337	3.0
Swap	277	3.0

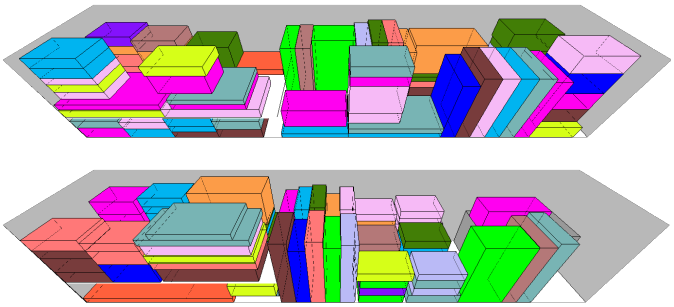

Fig. 8: *Comparison of Loadings*. Two valid loadings for the same problem instance. *Top*: A loading as generated by the self-adaptive GA. *Bottom*: A loading as created by a human planner.

Given that the common range of penalties for loadings is in the order of 10^3–10^4, notice the spread between $-5 \cdot 10^3$ and $6 \cdot 10^3$ in the differences between the penalties seen in Fig. 6. While this spread is too large to argue that any configuration of the GA will consistently generate better loadings, this even distribution around zero confirms that results will be useful as is or after manual improvements.

Of the set of boxes per problem instance, Table I lists the percentage of boxes that the GA was not able to fit into a loading. The static-mutation strategy has the lowest success, with almost half of all runs encountering at least one box not being placed. All self-adaptive mutation rate strategies resulted in such a case for 15 percent of the problem instances. The percentage of the total number of boxes that could not be fit into a loading, is very low for all mutation strategies.

Table II shows the number of runs for which a GA configuration generated a lower penalty for a certain objective than the loading by the human planner. The low numbers for the *ClientOrder, StackHeight* and *UnderBridge* penalties are supported by Fig. 7, where it can be seen that those penalties have the highest probability of scoring zero, which cannot be improved upon. The remaining values for all self-adaptive strategies show that the GA regularly earns a lower penalty in the most influential penalties, contributing to overall penalty values that can compete with those of manual loadings.

Table III lists the number of runs for which no stagnation occurred. It becomes clear immediately that the static mutation

rate performed best in terms of continually finding improvements. Only 20 percent of runs encountered stagnation, with an average of 3.0 stagnations per run. The (1,10) strategy performs worst, with around 80 percent of the runs encountering stagnation, significantly worse than the 50–60 percent for each of the (5,35) strategies. Although the difference is small when considering the average number of stagnations per run, the (5,35) strategy outperforms the (1,10) strategy.

An example of the generated loadings can be seen in Fig. 8. The differences between these loadings are due to several best practices that a human planner uses when creating a loading, such as placing tall boxes against the bridge, and combining large, flat boxes together into stacks on or under the bridge. The GA has partially emulated these, by placing the same tall boxes close together and creating stacks with several large, flat boxes. With a few minor adjustments, the shortcomings can be solved to create a loading that could meet the quality of loadings created by human planners.

V. Conclusions and Outlook

We have shown that it is possible to employ a self-adaptive GA for generating solutions to a highly constrained Truck Loading Problem, using an abstract, indirect representation to reduce complexity while maintaining the required degrees of freedom for optimization. The solutions can be generated in reasonably short amounts of time on regular desktop computers, which is essential for use of the method in industry. Furthermore, we have demonstrated that solutions generated by the GA are valid and match the penalty values of loadings created by human planners.

Most solutions generated by the GA fit all boxes into the container, which is the most important objective. Exactly how the boxes are ordered in the container can still differ significantly between loadings generated with the GA and those created by human planners. This can be seen in Fig. 8. Both the loadings of the GA and human planner are valid. The man-made loading shows more signs of having been planned using a set of best practices. This suggests that the generated loadings can be used directly, or used as a starting point. If the loading shows enough promise, a human planner can save time by only correcting those situations which are undesirable. Otherwise, it can easily be dismissed and the loading can be planned manually.

The used fitness function consists of an aggregation of multiple, unit-less penalties, that was not designed with use by a GA in mind. It contains all elements that are to be considered for evaluating a loading, but it does not properly reflect the preference for the desired structure within loadings. As Fig. 7 shows, some of these penalties rarely produce any high values, while others dominate the total penalty value. These inconsistent penalties are a sign that the fitness function is not properly balanced; the priorities of the human planner do not fully match the penalties given to loadings.

In most cases, the properties that define a good loading are more easily described in terms of the best practices used by human planners than in terms of objectives or penalty values. Certain boxes are commonly placed together, and can have a preferred place in the container. These preferences are not well defined and will often conflict with other penalties in the fitness function. This was further confirmed in an interview with two experienced human planners, as they did not agree over the preference order of a set of example loadings consisting of only a small number of boxes.

A more accurate fitness function is required to further improve the generated loadings. First, the penalties should be expressed in the same unit, such as a monetary value for additional time taken on a ClientOrder penalty. Next, the remaining and combined penalties have to be aggregated according to the correct ratios. As manually defining ratios for such a function is difficult, self-learning could be used to identify the ratios that human planners implicitly use in applying their best practices.

Self-learning can also be applied in emulating the best practices themselves, as they are used during the planning process of human planners. Preferences of which types of boxes are stacked on each other could be identified and used to create a directed mutation operator, that emulates the preferrential planning of certain boxes in man-made loadings.

Acknowledgments

The authors would like to thank Paul Dragstra and Joost Leuven for the many useful discussions, and the anonymous reviewers for their comments.

References

[1] S. Allen, E. Burke, and G. Kendall. A Hybrid Placement Strategy for the Three-Dimensional Strip Packing Problem. *European Journal of Operational Research*, 209(3):219–227, 2011.

[2] E. Bischoff and M. Ratcliff. Issues in the Development of Approaches to Container Loading. *Omega*, 23(4):377–390, 1995.

[3] A. Bortfeldt and G. Wäscher. Constraints in Container Loading – A State-of-the-Art Review. *European Journal of Operational Research*, 229(1):1–20, 2013.

[4] T. Dereli and G. Sena Das. A Hybrid Simulated Annealing Algorithm for Solving Multi-Objective Container-Loading Problems. *Appl. Artif. Intell.*, 24(5):463–486, may 2010.

[5] M. Eley. Solving Container Loading Problems by Block Arrangement. *European Journal of Operational Research*, 141(2):393–409, 2002.

[6] H. Gehring and A. Bortfeldt. A Genetic Algorithm for Solving the Container Loading Problem. *International Transactions in Operational Research*, 4(5-6):401–418, 1997.

[7] J. F. Gonçalves and M. G. Resende. A Biased Random Key Genetic Algorithm for 2D and 3D Bin Packing Problems. *International Journal of Production Economics*, 145(2):500–510, 2013.

[8] H. Hasni and H. Sabri. On a Hybrid Genetic Algorithm for Solving the Container Loading Problem with no Orientation Constraints. *Journal of Mathematical Modelling and Algorithms in Operations Research*, 12(1):67–84, 2013.

[9] M. Juraitis, T. Stonys, A. Starinskas, D. Jankauskas, and D. Rubliauskas. Randomized heuristic for the container loading problem: Further investigations.

[10] J. Kruisselbrink, R. Li, E. Reehuis, J. Eggermont, and T. Bäck. On the Log-Normal Self-Adaptation of the Mutation Rate in Binary Search Spaces. In *GECCO'11*, pages 893–900. ACM, 2011.

[11] F. Parreño, R. Alvarez-Valdes, J. Oliveira, and J. Tamarit. Neighborhood Structures for the Container Loading Problem: a VNS Implementation. *Journal of Heuristics*, 16(1):1–22, 2010.

[12] L. Wei, W.-C. Oon, W. Zhu, and A. Lim. A Reference Length Approach for the 3D Strip Packing Problem. *European Journal of Operational Research*, 220(1):37–47, 2012.

[13] Y. Wu, W. Li, M. Goh, and R. de Souza. Three-Dimensional Bin Packing Problem with Variable Bin Height. *European Journal of Operational Research*, 202(2):347–355, 2010.