Universiteit Leiden
Opleiding Informatica

Introducing User-Derived Information in the Optimization of Highly Constrained Truck Loading

Name:	Joost Leuven
Date:	31/8/2015
1st supervisor:	Thomas Bäck
2nd supervisor:	Michael Emmerich

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

Truck loading problems from practice tend to involve a lot of constraints. In automatically solving these problems, a rule set has to be compiled that governs the generation of valid solutions. This rule set is either defined manually, or distilled automatically by analyzing solutions of human planners. Van Rijn *et al.* describe an approach for automatically solving these problems using a manually defined set, determined in consultation with industry. In a follow-up paper, we reported on an addition to this approach that introduced user-derived information into the mutation operator. This thesis extends the existing approach further by introducing this user-derived information into the function that transforms the used indirect representation into a solution that can be evaluated. Furthermore, a method is proposed to correct the penalty function used to evaluate candidate solutions. With these extensions we are able to generate solutions that accommodate more boxes into a container, as well as improve on quality, in the sense that the automatically generated solutions conform more to how a human planner solves these problems.

Acknowledgements

The author would like to extend special thanks to Edgar Reehuis for his tremendous help in the process of writing and the realisation of this thesis. The author would also like to thank Sander van Rijn, Paul Dragstra, and Frits Ullmann for their contributions to this thesis.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

When solving instances of a *Truck Loading Problem* in logistics, constraints and objectives are to be defined, in order to generate valid and desirable solutions. In the instance of the TLP, the so called *Highly Constrained Truck Loading* problem, a lot of constraints and objectives arise that have been predefined, that are implicitly used when a human planner generates a solution. These constraints and objectives however have proven to be either false and/or incomplete.

A paper by Van Rijn *et al.* [21] proposes an *uninformed mutation operator* as well as an *indirect representation* so that the HCTL problem instances can be solved with the use of a *Genetic Algorithm*, more specifically a *Self-Adaptive Genetic Algorithm* (SAGA) as put forward by Kruisselbrink *et al.* [15]. In our earlier work we extended this approach with the introduction of an *informed mutation operator* that uses statistics derived from user-accepted solutions.

In this thesis we will further extend this approach for automatically solving instances of HCTL, by introducing user-derived information into the function that converts the indirect representations of solutions into solutions that can be evaluated. Furthermore, the function that evaluates is adjusted by a user-derived *correction factor*.

Chapter 2 gives an overview of the *Highly Constrained Truck Loading* (HCTL) problem studied in this thesis. In Chapter 3 an overview is given of the SAGA that has been implemented by Van Rijn *et al.* to solve instances of the HCTL problem automatically.

Experienced human planners are solving these problems by hand. These human planners use some combination of best practices from experience. To emulate this behaviour when solving the HCTL problem automatically some useful statistics are extracted from the user-generated solutions, by determining how often certain types of boxes are placed on top of each other. A method for extracting this information is described in Chapter 4.

In solving instances of this optimization problem automatically with the use of a *Genetic Algorithm*, three main components need to be defined: representation, variation operator, solution evaluation. These three components are used in a SAGA, where the representation is used to effectively store a solution, the variation operator to modify *candidate solutions*, and the solution evaluation to differentiate solutions of good and bad quality.

The indirect representation describes which box needs to be placed on which stack in which area of the trailer (Chapter 5). However, this representation does not represent an absolute place for the boxes, but a place relative to other boxes. This reduces the search space for the GA, by eliminating similar solutions.

For converting a candidate solution into a solution that can be evaluated for quality, we use an *Uninformed Building Function*, first proposed by Van Rijn *et al.* [21]. The building function tries to place each of the boxes on its intended stack. If this is not possible the building algorithm will search the rest of the container for an alternative placement. The uninformed building function is extended to incorporate statistics derived form user-provided solutions to form the *Informed Building Function.*

During the course of the optimization algorithm, solution candidates evolve towards solutions of better quality. To accomplish the evolution of the candidate solutions Van Rijn *et al.* [21] propose an uninformed variation operator, in the sense that no statistics derived from user-generated solutions. In our follow-up study, an informed extension to this variation operator is introduced, that makes use of placement-statistics derived from user-generated solutions [16]. Another variation operator that is introduced changes the placement location of a stacked *group of boxes.* A description of these extensions can also be found in Chapter 6.

To distinguish the solution candidates on quality, the candidate solution is evaluated by the use of a *penalty function*, described in Chapter 7. This penalty function yields one aggregated *penalty score.* The factors (sub-penalties) contributing to this penalty score are described in the same chapter.

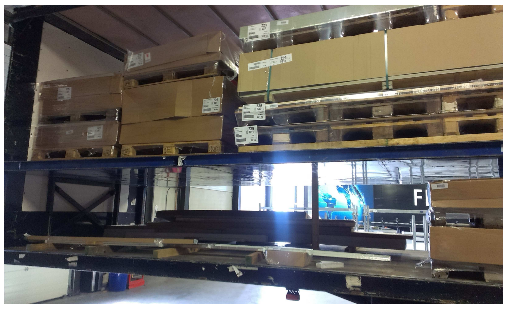
In experiments reported in [21] and [16], the automated solver is able to generate solutions which according to the penalty score are close to the solutions put forward by experienced human planners, but when evaluating the solutions by hand show some shortcomings. In Chapter 7 a *correction factor* for the sub-penalty weights is proposed.

Experiments indicate that the corrected sub-penalty weights can identify the some of the shortcomings found in the automatically generated solutions. A further subjective analysis of solution provided by the automated solver that uses the penalty function with the corrected weights is performed. This analysis indicates that a lot of the loading patterns observed in user-accepted solutions are also present.

Chapter 2

Highly Constrained Truck Loading

In this chapter, a description is given of the *Highly Constrained Truck Loading* (HCTL) problem studied in this thesis. The problem under investigation is a combination of the *Bin-Packing Problem* (BPP) and the *Container Loading Problem* (CLP), as encountered in practice. The instances we intend to solve have a *strongly heterogeneous* set of *boxes* that needs to be loaded into one *container* (see Figure 2.1), and a lot of different *constraints* originating from practice.


(a) *Front Part of the Trailer.* The front part of the trailer showing the raised area (bridge) present in some of the trailers under investigation.

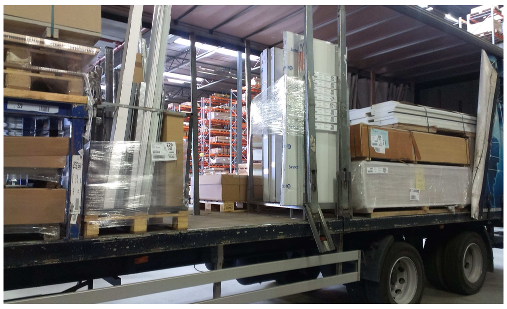

(b) *Rear Part of the Trailer.* The rear part of the trailer showing part of the flaps that will cover the sides of the trailer during travel.

FIGURE 2.1: *Loaded Truck Trailer.* Photos show a trailer as it is loaded in practice.

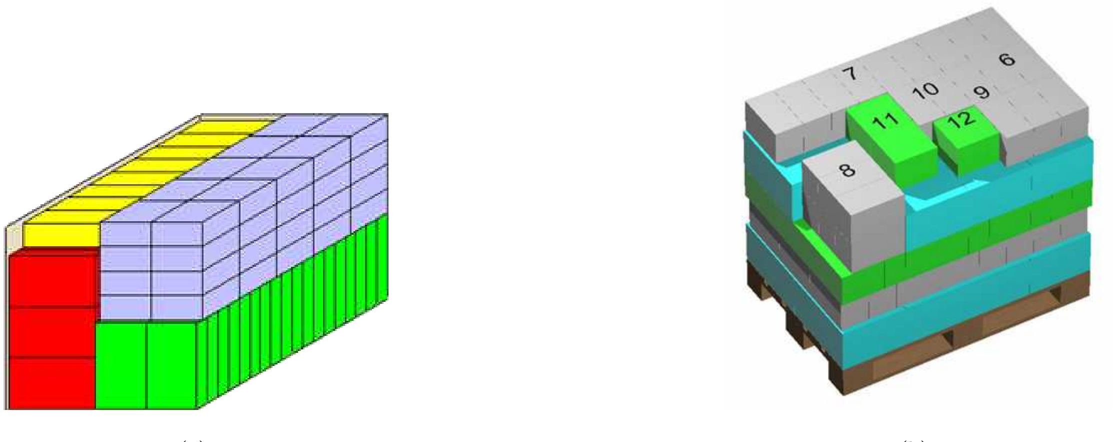
Most of the problem instances considered in this thesis have a raised area, called a *bridge* (see Figure 2.1). The raised area divides the trailer into three main sections: On the bridge, under the bridge, and adjacent to the bridge. This makes the problem studied a combination of CLP and BPP. In CLP boxes need to be loaded into a single container.

Each of the sections in the trailer can be considered a separate instance of CLP. In BPP items need to be distributed over a number of containers (bins), in our case the sections.

Solvers that are available for these types of problems, however, deal with theoretical problems involving only a few constraints and homogeneous sets of boxes [3, 4, 14, 24]. These solvers produce packing patterns unsuitable (see Figure 2.2) for the problem considered, where all boxes are to be organized in *stacks* that are loaded and unloaded from the sides of the container.

Each of the problem instances used in this thesis is accompanied by a corresponding solution made by an experienced human planner. In solving instances of BPP or CLP, a human operator is likely to use some combination of best practices from experience. Automated solvers described in literature (e.g., [13, 17, 18, 24]), commonly emulate best practices in the form of heuristics that can be used when volume-efficiency is the only governing objective, which is not the case in the problem we are studying here.

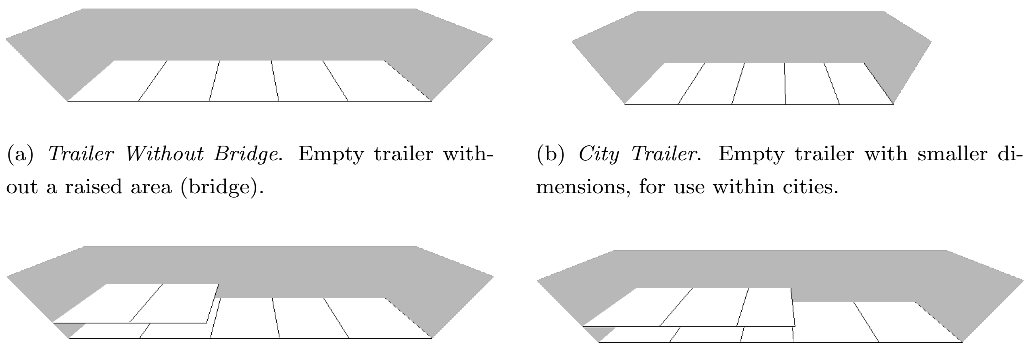
To account for the fact that loading and unloading actions are performed from the sides, each of the main sections is subdivided into a left and a right *area*. This means that when a bridge is present in the container, it will contain a total of six sections, whereas a container without such a bridge will consist of two sections.



(a) (b)

Figure 2.2: *Undesirable Stacking Pattern.* These patterns are unsuitable for the problem studied as boxes are supposed to be picked up with the use of a fork lift.

The boxes in the problem instances have greatly differing sizes and shapes, as well as varying weights. Each box is represented by a *cuboid*, a simplification, as in the real world boxes can be oddly shaped. However, the deviant information is retained by disallowing placement of other boxes on top of these. A problem instance is defined by a set of boxes intended for a number of *clients*. Preferably, these boxes are fit into a single container. The problem instances investigated in this thesis are real world, provided by industry.



(a) *Trailer Without Bridge.* Empty trailer without a raised area (bridge).

(b) *City Trailer.* Empty trailer with smaller dimensions, for use within cities.

(c) *Trailer With Bridge.* Empty trailer with raised area (bridge).

(d) *Trailer With Larger Bridge.* Empty trailer with larger raised area (extra bridge).

FIGURE 2.3: *The Four Types of Trailers.* Four types of trailers exist, two with a bridge and two without.

Some overhang and weight restrictions during placement were provided by industry, as well as some rudimentary rules about which types of boxes are allowed to be placed on top of each other. These rules prevent the placement of extremely heavy boxes on top of boxes that have been indicated as being fragile, for example the placement of a stack of heavy wooden plates on top of a glass shower cabin.

To be able to objectively evaluate suggested loadings, *quality measures* have been devised. These quality measures are defined as penalty functions and have been determined in consultation with industry. They represent stability of the loading, number of loading/unloading actions and the stability of the loaded trailer, among other things. These penalty functions are combined into a single penalty value by the use of *weighted aggregation*, where all the sub-penalties are summed into a single value by multiplying them with a factor reflecting their importance (weight).

Chapter 3

Automated Optimization of Highly Constrained Truck Loading

To automatically optimize instances of the HCTL problem, an optimization algorithm such as a *Genetic Algorithm* (GA) [2] can be used. A GA iteratively has its population of candidates evolving towards an optimal configuration. A GA typically operates on discrete-valued solutions and is non-deterministic by relying on randomness to steer its evolution.

To be able to do this, several components need to be defined. A *solution representation* is necessary that describes, in some shape or form, what a solution looks like (see Chapter 5). A *variation operator* is needed that is able to modify the solution represented, so that a different solution is created (see Chapter 6). Lastly, a method of evaluating the quality of a proposed solution are to be defined (see Chapter 7).

3.1 Self-Adaptive Genetic Algorithm

As core optimization algorithm, a *Self-Adaptive Genetic Algorithm* (SAGA) proposed by Kruisselbrink *et al.* [15] is used. The SAGA makes use of *individuals* within its *population* containing both a *candidate solution* and a *mutation magnitude* that governs the extent of variation that the application of the variation operator gives rise to.

Van Rijn *et al.* [21] provide problem-specific operators allowing application of such a SAGA to the HCTL problem under consideration. They define an *indirect* solution representation that derived from how a human planner solves the problem by making stacks of boxes and placing these stacks into the trailer from the side.

Algorithm 1 SAGA ([21])

```
1:  $t \leftarrow 0$
2:  $P^{(0)} \leftarrow$ generate $\mu$ individuals $\vec{a}_1, \dots, \vec{a}_\mu$, randomly
3:  while not terminate do
4:      for $i = 1$ to $\lambda$ do
5:          $\vec{a}_i \leftarrow$ copy randomly selected parent from $P^t$
6:          $\vec{a}_i \leftarrow$ mutate(mut. magn.($\vec{a}_i$))
7:          $\vec{a}_i \leftarrow$ mutate($\vec{a}_i$)
8:          $f_i \leftarrow$ evaluate($\vec{a}_i$)
9:      end for
10:     $P^{(t+1)} \leftarrow \{\vec{a}_{1:\lambda}, \dots, \vec{a}_{\mu:\lambda}\}$, select $\mu$ best from $\lambda$ total
11:     $t \leftarrow t + 1$
12: end while
```

In Algorithm 1, the size of the *parent* population is indicated by μ and the size of the *offspring* population is represented by λ. For each of the generations P^{t+1}, λ offspring are generated by mutating a randomly selected parent from generation P^t (n.b., no crossover is used). Each of the individuals has its own mutation magnitude, which defines the invasiveness of the variation operator. This variation operator is adapted, according to method SA3 from [15], before the variation operator is executed on the individual.

To select the μ best individuals to continue into the next generation, comma-selection is used, where only the newly generated individuals can become part of the new generation. Thus, our the selection operator makes sure that the SAGA does not suffer from the fact that a candidate with a good solution might not have a good mutation magnitude, and thus will not be able to produce offspring of sufficient quality, as put forward in [2].

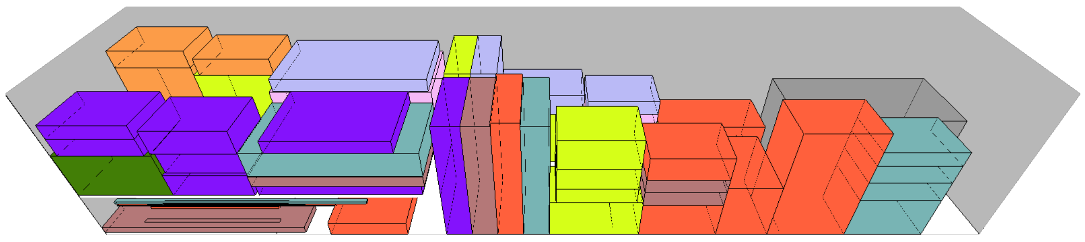


Figure 3.1: *Example of a container loaded with boxes for multiple clients.* Each color represents a different client to which boxes have to be delivered. Figure courtesy of [21].

3.2 Self-Learning

In trying to identify all factors that contribute to efficiently loading a container, it is very difficult to get the experienced human planners to clearly define all aspects of their solving process. Therefore, in Chapter 4, a method is presented that takes solutions generated by the planners and extracts the information needed from those, for solving new problem instances. The SAGA-based automated solver allows for several ways of user-derived information to be introduced into the search process.

Human planners generate solutions to instances of the truck loading problem. *Self-learning* is term we use to describe a method that as more of these solutions become available, an automatically derived rule set is further improved. To integrate self-learning into the existing automated solver, a clear definition of a *box* needs to be provided.

Box-types will be introduced to this end, defined by combinations of *box properties.* Having defined box-types, we can derive relations *between* box-types. These relations are in the form of allowed placement of a certain box-type on top of another. When these relations are derived from solutions by human planners, we can extend the SAGA-based approach to emulate human behaviour.

Chapter 4

User–Solver Exchange and Interaction

In this chapter, a method will be described that takes solutions provided by experienced human planners and extracts useful information from them, as we laid out before in [16]. In addition to that, an overview of factors will be given that need to be taken into account when setting up interplay between the user and the automated HCTL-solver.

4.1 Statistics Extraction

One of the important aspects of the planning of boxes into a container, is knowing what boxes are allowed to be placed on top of certain (other) boxes. To use the solutions generated by the human planners to this end, a method is devised that extracts useful information from them. Boxes get assigned *box-types* according to relevant box properties, after which relationships between these types are defined. These relationships and their *occurrence count* can be integrated into the optimization process, which is explained in Chapters 5 and 6. Numerical (that is, not categorical) box properties need their values to be grouped, so that they can be used more effectively when jointly representing an element of a box-type. Therefore, a method is laid out for grouping these values into *bins* with varying ranges.

4.1.1 Box Properties

Boxes that are placed on top of others have different *behaviour* from the boxes that they are placed upon. When we talk about behaviour of a box, we mean the way a user deals with a box when it needs to be planned in. To reflect this behaviour, two box-type categories are defined: An *above-box-type* and a *below-box-type*. Each box gets assigned both.

Box-types are defined by:

- *Pallet Configuration*, a short description (represented by an index value) of how the box is configured, and whether it has a smooth or rough surface. This parameter defines whether a box is likely to slide;
- *Product Group*, a short description (represented by an index value) of what type of product is in the box. For certain products it is disadvantageous to place anything on top of it;
- *Surface Area* of top face (continuous value), it gives an indication of how many other boxes could be placed on top of a box;
- *Weight* (continuous value) is used in the above-box-type. It represents how much pressure this box puts on the boxes below it;
- *Density* (continuous value) is used in the below-box-type. It is a representation of how solid a box is; a solid box can carry a bigger load on top of it.

It occurs that the properties surface area, weight, and density are unevenly distributed over their property ranges. Where a lot of property value instances occur close together, greater precision should be used to enable more detail in the resulting rules.

4.1.2 Binning of Property Values

Ideally, all bins are filled with the same number of items, such that all rules, to be derived from combinations of the property values, represent the same fraction of the data.

Algorithm 2 Determine Bins

```
1: function DetermineBins($DataSet$)
2:     $IdealSize \leftarrow$ CountMostFrequentItem($DataSet$)      // Count most frequent item
3:     $BinSep \leftarrow$ SplitInBins($DataSet, IdealSize$)      // Determine bin-separation values
4:     $BinSep \leftarrow$ MergeBins($DataSet, BinSep$)      // Merge bins where needed
5: end function
```

To do this in presence of unevenly spread values, Algorithm 2 first calculates the number of occurrences of the most frequent item. This number will be used as the *ideal* number of items in each of the bins.

Algorithm 3 Split Data Into Bins

```
1: function SplitInBins($DataSet$, $IdealSize$)
2:     $\epsilon = 0.0001$
3:     if Length($DataSet$) $> IdealSize$ then
4:         $DataSet_{\text{median}} \leftarrow$ Median($DataSet$)
5:         $DataSet_{\text{Before}} \leftarrow \{DP \in DataSet \mid DP < DataSet_{\text{median}}\}$
6:         $DataSet_{\text{Equal}} \leftarrow \{DP \in DataSet \mid DP = DataSet_{\text{median}}\}$
7:         $DataSet_{\text{After}} \leftarrow \{DP \in DataSet \mid DP > DataSet_{\text{median}}\}$
8:         $BinSep_{\text{Before}} \leftarrow$ SplitInBins($DataSet_{\text{Before}}$, $IdealSize$)
9:         if Length($DataSet_{\text{Equal}}$) $> IdealSize/2$ then
10:                                     // Median is frequent enough → separate bin
11:            $BinSep_{\text{Equal}} \leftarrow \{DataSet_{\text{median}} - \epsilon, DataSet_{\text{median}} + \epsilon\}$
12:        else
13:            $BinSep_{\text{Equal}} \leftarrow \emptyset$
14:        end if
15:        $BinSep_{\text{After}} \leftarrow$ SplitInBins($DataSet_{\text{After}}$, $IdealSize$)
16:        return $BinSep_{\text{Before}} \cup BinSep_{\text{Equal}} \cup BinSep_{\text{After}}$
17:    end if
18:    return $\emptyset$
19: end function
```

Initially, there is a single bin containing all the data. This bin is split according to Algorithm 3, until the number of items in each of the bins is smaller than the ideal number.

The function described in Algorithm 3 recursively determines bin-separator values, by calculating the median of the provided data set. If this median-value occurs frequently enough ($IdealSize/2$) in the provided data set, a separate bin will be created for this value, by introducing two bin-separator values: $DataSet_{\text{median}} - \epsilon$ and $DataSet_{\text{median}} + \epsilon$, where ϵ has a small enough value to only encompass the $DataSet_{\text{median}}$-value.

This is followed by a step in which adjacent bins might be merged. Per bin it is determined what leads to a number closest to the ideal: Merge with the bin before, leave as is, or merge with the bin after.

4.1.3 From Box Properties to Box-Types

After binning the non-categorical property values, we can combine the four parameters to create a box-type. Each *unique* combination of the parameter values will constitute a box-type. This is done for the above-box-type and below-box-type separately. Each box will be assigned an above-box-type for representing behaviour when being placed on top of another box, and a below-box-type for describing behaviour when being placed underneath another box.

A two-dimensional contingency table can now be created, in which the number of times that a box of a certain above-box-type is placed on top of another box of a certain below-box-type is counted. These *occurrence counts* can be used as guidance rules in the optimization, by taking the chance of a possible combination of stacked boxes proportional to its relative occurrence count.

4.2 Discussion of User–Solver Interaction

In this section a description is given of ways to introduce user-feedback into the optimization process. A distinction is made between *direct user-feedback* and *indirect user-feedback*. Direct questions that are answered by the user, and in that way provide feedback, are considered direct user-feedback. Implicit information that users impart by providing that solver with their solutions to HCTL problem instances is considered indirect user-feedback. Furthermore, possible pitfalls and areas of concern are discussed, when introducing user-feedback into the automated planning process.

4.2.1 Introduction of User–Solver Corrections

Having the automated solver suggest possible solutions and the user correct them is an efficient way of training the solver. It is those corrections that can be used to improve the solver quickly. Once more information becomes available, inferences can be made from the already learnt corrections that will possibly prevent additional future corrections the user will need to make, for example boxes with intermediate property values that should behave similarly.

4.2.1.1 Placement Rules

It is difficult to define every *single rule* (constraint) and *optimization measure* (objective) needed to build a valid loading of sufficient quality beforehand. To assist the automated

HCTL-solver in learning how to make loadings of good quality, the user can be asked to correct the suggested solutions put forward by the solver. These corrections will, in most cases, become hard constraints for the solver, in the form of *placement rules.*

Notion 4.1 Placement Rule. *A placement rule is defined by the combination of an above-box-type and a below-box-type, and states whether the placement of a box with the above-box-type is* allowed *or* disallowed *on top of a box with the below-box-type.*

Placement-rules can be derived in two ways:

1. Analyse final solutions made by users; if a user has made a certain placement, this will become a rule indicating the placement is **allowed**, which means the automated HCTL-solver is allowed to perform this placement (**indirect user-feedback**). This method is laid out in our earlier paper [16] and in Section 4.1.
2. The automated HCTL-solver is used to generate a suggestion-loading. Afterwards the user makes corrections. These corrections will indicate that the solver performed a placement that is **not allowed**. Thus, this invalid placement will become a placement-rule that indicates that the placement is not allowed (**direct user-feedback**).

4.2.1.2 Solution Evaluation

Another possibility for introducing user-feedback into the optimization process is to allow the users to influence how loadings are evaluated. This can be achieved by giving the user simple example-loadings with slight differences and let the user *rank* these loadings. These rankings can be used to adjust the weight of the sub-penalties in the penalty-function (**direct user-feedback**).

Additionally, the user can be introduced into the evaluation process by analysing solutions generated by the user. In this analysis, more *prevalent penalties* are apparently of less importance to the user. The weights assigned to each sub-penalty can then be adjusted according to the importance a user has implicitly assigned to each of those penalties (**indirect user-feedback**). This method is further explained in Chapter 7.

4.2.2 Considerations When Utilizing User-Feedback

Prompting of User. When asking any user for feedback, several factors must be observed. The feedback-frequency is the most imported of those factors; if a user is prompted too often, an adverse effect can be introduced where the user will no longer give the *correct* response, but rather the response that will cost the *least amount of time.* To this end the number of interactions with the user should be limited as much as possible.

When the user indicates that a certain placement is valid, the number of *future prompting* of the user should be further reduced by *analysing the feedback* the user has given. For example: when a user indicates that box of type A can be placed on top of a box of type B, and a box of type C is lighter than the box with type A, but has the same dimensions and has the same PC and PG (see Section 4.1.1), the box of type C should also be allowed on top of the box of type B.

Mistakes. Another area of concern that can be encountered is the fact that the user can provide unintended or intended but erroneous feedback. When this occurs a user should be able to correct it.

Conflict Between Users. A certain "mistake" might not be a mistake at all but just a preference of the user. This, of course, can only occur when multiple users provide conflicting corrections on suggested solutions. The conflict between users should be logged and reported back to the team of human planners. This is also a case that might be useful for the users to discuss amongst themselves, to come to a consensus on which of the views should be followed.

Reasons for Feedback. One of the major areas of concern with direct user feedback is the reason a certain direct user-feedback is given. Several reasons can be envisioned:

- The current box is not allowed to be placed on top of another;
- It is not allowed to place a box on top of the current box;
- The current box has too much *overhang* over the box below it;
- The weight of the current box is too great for the box below to handle;
- User preference; the user sees a better place for it.

The last of the reasons listed above is the most difficult to detect, as it does not indicate an error, but rather a better *alternative placement.* If a user only has to decide between right and wrong, the exact reason for it being wrong or right needs to be determined. The user can also be asked to give this exact reason, but this would mean that the user needs to spend even more time to give this feedback.

Therefore, it might be beneficial to present only *right–wrong questions* to the user. In this case, the exact reason for that given feedback needs to be determined automatically.

In practice, this means that a lot more true–false data entries are needed before the exact reason can be determined with any kind of certainty.

Chapter 5

Solution Representation

To effectively optimize an instance of the HCTL-problem under consideration, Van Rijn *et al.* [21] propose an *indirect solution representation*, which is derived from how human planners solve the problem. The representation indicates for each *box* in which *stack* it is to be placed, located in a certain *area* of the *container*. This is then translated into an exact (x, y, z)-coordinate. This significantly reduces the search space, as compared to directly operating on (x, y, z)-coordinates, without excluding feasible solutions. However, this representation does not reflect that placement on a certain stack is actually possible with respect to spatial constraints.

In our earlier work [16], we employed the translation routine provided by Van Rijn *et al.* [21], which takes *hard constraints* and *initial placement rules* into account. Here, we provide an alternative *informed* procedure that, next to these, adheres to *user-derived placement rules*, which are based on observed combinations of boxes that were planned by end-users.

5.1 Indirect Representation

With the *Self-Adaptive Genetic Algorithm* (SAGA) used for solving HCTL, we make use of the following tuple-notation to represent a candidate solution L:

$$L = (S_1, S_2, S_3, ..., S_n) \tag{5.1}$$

where n is the number of *boxes* in the current problem instance. Each *step* S_i represents the placement of one of the boxes in the problem instance and consists of three parts:

$$S_i = (b_i, a_i, s_i) \tag{5.2}$$

where b_i is an unique identifier for a given box, a_i is the *area* in which the box is to be placed, and s_i is the specific stack in that area on which the box is to be placed. It is exactly these steps that the user takes when solving an HCTL-problem: The user compiles the loading stack by stack, and usually not by first placing a lot of boxes on the floor of the container and constructing stacks on top of them layer–by–layer.

This representation was proposed to more effectively explore the search space [21]. By limiting the freedom of the *variation operation* (see Chapter 6) to the coordinates representing stacks and areas where a box is to be placed, a more restricted search space is defined. If a representation would be used where a placement of a box is described by the (x, y, z)-coordinates of its placement, many of the solutions would effectively be the same [21].

With trailer dimensions of 13 600 mm x 2500 mm x 2650 mm, each box could be placed at approximately $90 * 10^9$ different positions, of which most options would be invalid because boxes need to be placed on the floor or on top of other boxes. Assuming that no more than 25 stacks are present in each of the areas, and at most 6 areas in a container, this yields a total of at most 150 possible locations for a box to be placed. Thus, our search space is reduced by a factor of approximately $600 * 10^6$ per box.

5.2 Conversion from Representation to Solution

To obtain an actual solution from an indirectly represented candidate solution, a *deterministic* conversion routine is used. This *building function* [21] takes each of the steps, in the order of their indices, and tries to place a box b_i at its desired (a_i, s_i)-coordinate, as is described in Algorithm 4. If this is not possible due to any of the hard constraints ($\textsc{FitsAt}(b, a, s)$, see Section 5.2.1) and initial rule-set ($\textsc{AllowedInitial}(b, a, s)$), see Section 5.2.2), see line 5, the building function will look through the rest of the current area a_i for a location that the current box can be placed in, thus placing the box as close to its original (a_i, s_i)-coordinate as possible. It is of course possible that the box does not fit on any of the stacks in the desired area. In this case the rest of the trailer is searched for a fitting alternative placement (lines 21 to 23).

As this placement function is not in any way influenced by user-derived information, it will be referred to as the *uninformed building function*. Section 5.3 extends it with user-derived information to obtain an *informed building function*.

Algorithm 4 Uninformed Building Function [21]

```
for $(b, a, s) \in L$ do                                          // Step box, area, stack
    $a', s' \leftarrow a, s$                                       // Make a local copy
    while box not placed and not all areas tried do
        if stack $s'$ exists in $a'$ then
            if FitsAt$(b, a', s')$ and AllowedInitial$(b, a', s')$ then
                PlaceAt$(b, a', s')$
                $a, s \leftarrow a', s'$                           // Copy locals back into $L$
                break while
            else
                NextStack$(s')$                                    // $s'$ updated
            end if
        else
            if FitsAsNewStack$(b, a, s')$ then
                PlaceAt$(b, a', s')$
                $a, s \leftarrow a', s'$                           // Copy locals back into $L$
                break while
            else
                FirstStack$(s')$                                   // $s'$ updated
            end if
        end if
        if all options for $s'$ in $a'$ have been tried then
            NextArea$(a')$                                         // $a'$ updated
        end if
    end while
end for
```

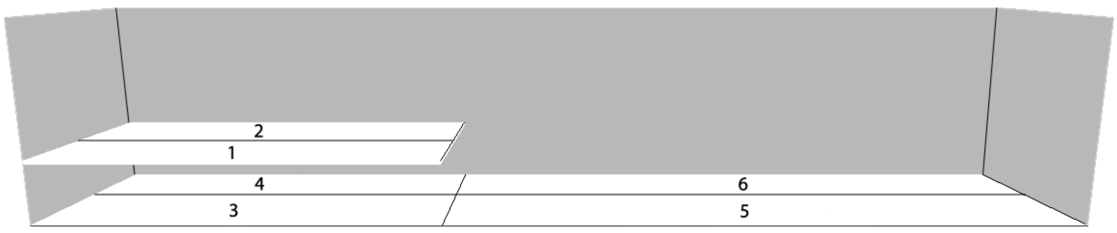


FIGURE 5.1: *Trailer With Numbered Areas.* The areas in the trailer are assigned an index, reflecting their ordering.

Area Order. Figure 5.1 provides the ordering assigned to each of the areas of the container; the uninformed building function tries to find alternative placements for a box (when no placement can be found within the preferred area) in the other areas according to a fixed order (introduced via NextArea(a)).

When a valid placement is found, the corresponding (a, s)-coordinate replaces the provided (a, s)-coordinate in the representation (lines 7 and 15). By applying the building function, we ensure not only that the maximum number of boxes is loaded into the container, but also, by reinserting the actual placement found back into the representation, that the one-to-one relationship between representation and solution is preserved. Since the building algorithm manipulates the representation, it is executed as part of the *variation operator*, further elaborated on in Chapter 6.

5.2.1 Hard Constraints

Several hard constraints have been defined in consultation with industry that need to be satisfied for a placement to be valid, looking at weight/density, overhang, i.e., how far does a box stick out over another.

1. $BoxA$ is allowed to be stacked on $BoxB$ if the following condition is met:

$$BoxA_{\text{Density}} \leq BoxB_{\text{Density}} * 2$$

2. $BoxA$ is allowed to be stacked on $BoxB$ if the following conditions are met:

$$\frac{BoxA_{\text{Width}} - BoxB_{\text{Width}}}{BoxB_{\text{Width}}} < 0.1 \text{ and } \frac{BoxA_{\text{Length}} - BoxB_{\text{Length}}}{BoxB_{\text{Length}}} < 0.1$$

   when the box is to placed not under the bridge. This ensures that the topmost box is at most 110% of the size of the box on the bottom. Under the bridge, a different set of conditions needs to be met before a box is allowed to be placed:

$$\frac{BoxA_{\text{Width}} - BoxB_{\text{Width}}}{BoxB_{\text{Width}}} < 0.5 \text{ and } \frac{BoxA_{\text{Length}} - BoxB_{\text{Length}}}{BoxB_{\text{Length}}} < 0.5$$

3. $BoxA$ is allowed to be placed on $BoxB$ if enough unfilled space is available above $BoxB$ to fit $BoxA$ in.

5.2.2 Initial Placement Rules

Initially, the automated solver makes use of rules drafted in consultation with industry. These rules have been defined as relations between manually defined box-types, where boxes with a certain type are not allowed to be placed on top of boxes of certain other box-types. These types are given in Table 5.1.

TABLE 5.1: *Overview of manual-box-types*

	Type Name	Description
0	NON STACKABLE	An id given to client to represent the order of servicing
1	CAGE	A metal cage that is placed in the trailer for smaller and more fragile items
2	CAGE PRODUCT	Products to be placed into the cage. These boxes are not part of the problem instance.
3	SOLID	Usually wooden floorboards (parquet and laminate)
4	DONT COUNT	Plates
5	SELF STACKABLE	Boxes with this type can be placed on top of each other
6	FRAGILE 1	Bath
7	FRAGILE 2	Radiator placed on a Euro 3 skid (1,000 mm 1,200 mm)
8	FRAGILE 3	Radiator placed on a Euro 1 skid (800 mm 1,200 mm)
9	REMAINING 1	Can be placed on top of Fragile 2
10	REMAINING 2	Can be placed on top of Fragile 2 and 3
11	REMAINING	All other boxes

Table 5.2 shows the placements rules in accordance with industry. For a lot of the box-types no placement rules have been defined. A large portion of the boxes is assigned one of the three REMAINING box-types, for which the initial rules do not define any restrictions to boxes that are placed on top of them. Nevertheless, the rules put forward in Table 5.2 prevent known undesirable placements.

TABLE 5.2: *Overview of manually defined rules*

Bottom Box	Top Box	Extra Requirement	Allowed / Disallowed
FRAGILE 1	FRAGILE 1	Weight difference $<$ 50 kg	Disallowed
FRAGILE 2	FRAGILE 1	$\text{Weight}_{\text{Top}} < \text{Weight}_{\text{Bottom}}$	Allowed
FRAGILE 2	FRAGILE 2	$\text{Weight}_{\text{Top}} < \text{Weight}_{\text{Bottom}}$	Allowed
FRAGILE 3	FRAGILE 3	$\text{Weight}_{\text{Top}} < \text{Weight}_{\text{Bottom}}$	Allowed
FRAGILE 2	REMAINING 1	$\text{Weight}_{\text{Top}} < \text{Weight}_{\text{Bottom}}$	Disallowed
FRAGILE 2	REMAINING 2	$\text{Weight}_{\text{Top}} < \text{Weight}_{\text{Bottom}}$	Allowed
FRAGILE 3	REMAINING 2	$\text{Weight}_{\text{Top}} < \text{Weight}_{\text{Bottom}}$	Allowed
FRAGILE 1	SELF STACKABLE	Weight difference $<$ 50 kg	Disallowed
Any	SELF STACKABLE	-	Disallowed
SELF STACKABLE	Any	-	Disallowed
SELF STACKABLE	SELF STACKABLE	-	Allowed
Any	NON STACKABLE	-	Disallowed

TABLE 5.2: *Overview of manually defined rules (Continued)*

Bottom Box	Top Box	Extra Requirement	Allowed / Disallowed
NON STACKABLE	Any	-	Disallowed

5.3 Introduction of User-Derived Information

The uninformed building function determines an a, s-coordinate for a given box b. However, it does not take any user-derived information into consideration in this process. Therefore, the *informed building function* is proposed where the uninformed building function is extended to incorporate the user-derived information, which will improve the adherence to the *user-derived* placement rules. The informed building function is described in Algorithms 5 to 7

Algorithm 5 Informed Building Function

```
1: for all $(b, a, s) \in L$ do                                  // Step box, area, stack
2:     if SingleStepInformed$(b, a, s)$ then    // Placement for box found, $b, a, s$ possibly updated
3:         PlaceAt$(b, a, s)$                                    // Place box $b$ in actual solution
4:     end if                                                    // Otherwise: try next box
5: end for
```

Algorithm 6 Single Step Informed

```
function SingleStepInformed(step, searchDepth)                // step is possibly updated
    $b, a, s \in$ step                                         // box, area, stack
    if FitsAt$(b, a, s)$ and AllowedInitial$(b, a, s)$ then    // Check preferred placement
        if AllowedUser$(b, a, s)$ then
            return True                                        // step unchanged
        end if
        step$'$ $\leftarrow$ step                              // Local step
    end if

    step$''$ $\leftarrow$ step       // Local copy. Check rest of current area for alternative placements
    switch SearchAreaInformed(step$''$) do                     // step$''$ possibly updated
        case ALLOWED_USER:
            step $\leftarrow$ step$''$; return True            // Update step
        case ALLOWED_INITIAL:
            if step$'$ equals $\emptyset$ then
                step$'$ $\leftarrow$ step$''$                  // Update step$'$
            end if
                        // We found an ALLOWED_INITIAL placement within current section
    if searchDepth == 0 and step$'$ not equal to $\emptyset$ then   // search depth already reached
        step $\leftarrow$ step$'$; return True                 // Update step
    end if

    numAreasSearched $\leftarrow$ 1
    while NextArea$(a')$ do                // Check other sections for an allowed user placement
        step$''$ $\leftarrow (b, a', s')$
        switch SearchAreaInformed(step$''$) do                 // step$''$ possibly updated
            case ALLOWED_USER:                  // The step we want (user) has been found
                step $\leftarrow$ step$''$;return True         // Update step
            case ALLOWED_INITIAL:
                if step$'$ equals $\emptyset$ then
                    step$'$ $\leftarrow$ step$''$
                end if
                if numAreasSearched >= searchDepth then
                    step $\leftarrow$ step$'$; return True     // Update step
                end if
        numAreasSearched $\leftarrow$ numAreasSearched + 1     // Search next area
    end while
    return False                        // No placement found, step is returned unchanged
end function
```

Algorithm 7 Search Area Informed

```
 1: function SearchAreaInformed(step)                    // stack $s$ in *step* possibly updated
 2:     $b, a, s \in$ step
 3:     foundAllowedInitial $\leftarrow \emptyset$
 4:     while NextStack($s'$) do                                         // $s'$ updated
 5:         if FitsAt($b, a, s'$) and AllowedInitial($b, a, s'$) then
 6:             if AllowedUser($b, a, s'$) then
 7:                 step $\leftarrow b, a, s'$
 8:                 return ALLOWED_USER
 9:             end if
10:             if foundAllowedInitial equals $\emptyset$ then
11:                 foundAllowedInitial $\leftarrow (b, a, s')$
12:             end if
13:         end if
14:     end while
15:
16:     if FitsAsNewStack($b, a, s'$) then        // $s'$ possibly updated to represent new stack
17:         foundAllowedInitial $\leftarrow (b, a, s')$
18:     end if
19:
20:     if foundAllowedInitial is not equals to $\emptyset$ then
21:         step $\leftarrow$ foundAllowedInitial
22:         return ALLOWED_INITIAL
23:     else
24:         return NONE
25:     end if
26: end function
```

The informed building function first checks if a box b can be placed at its preferred place a, s, adhering to the initial rules (line 3 in Algorithm 6). After this the adherence to the user-derived rules is checked (line 4 in Algorithm 6). If the preferred place is considered invalid, the building algorithm continues by checking the rest of the preferred section for an alternative valid placement (Algorithm 7).

If no placement in the preferred area is possible, the rest of the container is checked. For all these placements, conformance to the initial rules is checked first (line 5 in Algorithm 7), after which conformance to the user-derived rules is checked (line 6 in Algorithm 7).

An (a, s)-coordinate that is valid according to the user-derived rules will always get preference over an (a, s)-coordinate that is considered invalid, even if the invalid coordinate is closer to the original (preferred) coordinate in the representation. If no valid placement is found, then the first placement in accordance with the initial rules is used. Since the user-derived rules do not define whether it is considered valid to place a box on the floor of the trailer, a new stack needs to be created in an area to accommodate a box, is not considered a valid placement according to the user-derived rules.

Because of the extra restrictions in lines 4 and 6 of Algorithm 6, it takes the informed building algorithm longer than it takes the uninformed building algorithm to find a valid place for a certain package. To alleviate this problem, a hybrid version of the building function is proposed, where a maximum number of areas in the trailer is checked for a valid placement (lines 19 to 21 and lines 33 to 35 in Algorithm 6). After the search depth has been reached, the informed building algorithm falls back on placements that only adhere to the initial rules. The *search-depth* is one of the test-parameters described in Section 8.5.

Area Order. For the informed building function an alternative section order is used (Figure 5.1); the preferred starting area of a box determines the search order of the alternative areas. With this alternative ordering, the areas with a similar maximum height are searched first:

- When a box is to be placed *on the bridge*, the alternative area order is:
  Left: $1 \Longrightarrow 2 \Longrightarrow 5 \Longrightarrow 6 \Longrightarrow 3 \Longrightarrow 4$
  Right: $2 \Longrightarrow 1 \Longrightarrow 6 \Longrightarrow 5 \Longrightarrow 4 \Longrightarrow 3$
  where the other side of the bridge is checked, after which the areas behind the bridge and underneath the bridge are checked;

- When a box needs to be placed *below the bridge*, the area order is:
  Left: $3 \Longrightarrow 4 \Longrightarrow 1 \Longrightarrow 1 \Longrightarrow 5 \Longrightarrow 6$
  Right: $4 \Longrightarrow 3 \Longrightarrow 2 \Longrightarrow 2 \Longrightarrow 6 \Longrightarrow 5$
  where the other side under the bridge is checked first, then on the bridge, and finally behind the bridge;

- When a box has a preferred area *behind the bridge* the other areas are searched in the following order:
  Left: $5 \Longrightarrow 6 \Longrightarrow 3 \Longrightarrow 4 \Longrightarrow 1 \Longrightarrow 2$
  Right: $6 \Longrightarrow 5 \Longrightarrow 4 \Longrightarrow 3 \Longrightarrow 2 \Longrightarrow 1$
  where the other side behind the bridge is searched first, after which the container is checked for a place under the bridge and on the bridge. When a box does not

fit in one of the areas behind the bridge, it is likely that is oddly shaped (a long pipe for example), which is usually better fitted under the bridge.

The behaviour of NextArea(a) (line 24) in the informed building algorithm is defined by these orders.

Chapter 6

Variation Operator

User-derived information can be introduced into the SAGA-based solver in several places, one of which is the *variation operator*. We presented an approach for this in earlier work [16], giving rise to an *informed mutation operator*.

The original, *uninformed* variation in the automated HCTL-solver implemented by Van Rijn *et al.* [21], has two uses: To generate *new* solutions, and to *modify* existing candidate solutions. In generating new solutions, no user-derived information is introduced. This is to ensure that the automated solver is not restricted in the beginning of its optimization process. For both the uninformed and informed mutation operator this generation process of new solutions is kept completely random.

Furthermore, our earlier work [16] suggests that the *stack-mutation operator*, which swaps entire stacks in a solution, should be performed more often. This improvement is described in Section 6.3.

6.1 Uninformed Mutation Operator

Van Rijn *et al.* [21] provide a variation operator for HCTL, for use within a *Self-Adaptive Genetic Algorithm* (SAGA) [15]. This variation operator is purely random and does not make use of user-derived information. We termed this *uninformed box-mutation* [16].

The SAGA applies mutation as the only handle for generating new candidate solutions from existing ones. Thus the *mutation application probability* is 1.0. The *mutation magnitude* expresses the (expected) fraction of the total number of elements (here: boxes) n in the old candidate solution that gets mutated upon application of the mutation operator. It is defined as

$$mut_{\text{magn}} = mut_{\text{magn,sa}} + mut_{\text{magn,min}} \ , \tag{6.1}$$

where

$$mut_{\text{magn,min}} = \frac{1}{n} \tag{6.2}$$

is the lower bound of the total mutation magnitude. The *self-adaptive* mutation magnitude $mut_{\text{magn,sa}}$ is updated according to the rule (SA3 in [15])

$$mut'_{\text{magn,sa}} = \min\left(\frac{1}{2}, \frac{1}{1 + \frac{1-mut_{\text{magn,sa}}}{mut_{\text{magn,sa}}} \cdot \exp(\gamma \cdot \mathcal{N}(0,1))}\right), \tag{6.3}$$

where $\gamma = 0.22$ [15]. Updating the self-adaptive part of the mutation magnitude is performed before the actual mutation of the elements in the solution, i.e. $mut'_{\text{magn,sa}}$ is used for the mutation.

The total mutation magnitude $mut_{\text{magn,box}}$ for the *box-mutation* is initialized to 0.2, through initializing $mut_{\text{magn,sa,box}}$ to

$$mut_{\text{magn,sa,box}} = 0.2 - mut_{\text{magn,min}}. \tag{6.4}$$

The $mut_{\text{magn,box}}$ is part of the individual $\vec{r}$ in the SAGA, i.e.:

$$\vec{r} = (L, mut_{\text{magn,box}}), \tag{6.5}$$

$$L = (S_1, S_2, S_3, ..., S_n)\,, \tag{6.6}$$

$$S_i = (b_i, a_i, s_i)\,. \tag{6.7}$$

Each representation of a loading L is comprised of several steps S, each representing the placement of a box b on a stack s in an area a of the container. For each box in the solution a decision is made, based on the mutation magnitude $mut_{\text{magn,box}}$, whether the associated step (b, a, s) gets mutated. If this is the case, three parameters can be adjusted in mutating the step:

- *Placement order*, i.e., the position of the step associated with the box within the representation;

- *Area in the container*, where stack the stack that will contain the box is located;
- *Stack in the area*, in which the box is to be placed.

Each of the three parameters has an *independent* $\frac{1}{3}$-chance of being adjusted. For mutating the placement order, a random new location in the permutation of the steps $(S_1, S_2, ...)$ is chosen. The step associated with the box under consideration is then either *inserted* or *swapped* with the step currently at that location, see Sections 6.1.1 and 6.1.2.

For mutating the area in the container, the area index a currently in the step (b, a, s) is replaced by a random integer in the range $[1, 6]$, as there are 6 areas in a container when a bridge is present. For mutating the stack index s, a random integer in the range $[1, 25]$ is used, as 25 stacks was estimated to be the maximum number of stacks in an area.

When a bridge is not present a new random integer is chosen in the range $[1, 2]$ to indicate the area, and a stack number is randomly chosen in the range $[1, 75]$, since the areas are a lot bigger when a bridge is absent.

6.1.1 Insert

When performing an insert operation on a candidate solution, the current step is inserted into its representation before a random step.

For example, insert step 5 (S_5) before step 3 (S_3):

$$L = (S_1, S_2, S_3, S_4, S_5, S_6)$$
$$\downarrow$$
$$L' = (S_1, S_2, S_5, S_3, S_4, S_6)$$

6.1.2 Swap

A swap operation on a candidate solution is performed by selecting a random step from its representation and swapping the current step with that step.

For example, swap step 5 (S_5) with step 3 (S_3):

$$L = (S_1, S_2, S_3, S_4, S_5, S_6)$$
$$\downarrow$$
$$L' = (S_1, S_2, S_5, S_4, S_3, S_6)$$

6.2 Informed Mutation Operator

In Section 4.1.3, a method for extracting usable information from user-generated solutions is described. This extracted information is employed in emulating the user within the optimization approach, through an *informed box-mutation* operator. By definition, however, this mutation operation is unable to act on entire stacks. Therefore, a *stack-mutation operator* is introduced to remedy this limitation.

6.2.1 Informed Box-Mutation

With informed box-mutation, a box will be mutated according to user-derived statistics. Based on box-types, a box is either placed on top of (*above-mutation*), or underneath another box (*below-mutation*). Above-mutation has the ability to place a box on top of a stack, whereas below-mutation has the ability to place a box on the floor of the container.

To choose whether to apply above-mutation or below-mutation, counts are used of the number of times a box has been placed on the floor, and the number of times it has been placed on the top of a stack, in the user-solutions. This determines the chance of using either of the two mutation operators. In some cases, a box will make up the entire stack, in which it is both the topmost box *and* on the floor (see Figure 6.1). These *SingleBoxStack* cases are subtracted from the totals as follows:

$$\mathit{Floor}_{\text{normalized}} = \mathit{Floor} - \mathit{SingleBoxStack}, \tag{6.8a}$$

$$\mathit{Top}_{\text{normalized}} = \mathit{Top} - \mathit{SingleBoxStack}, \tag{6.8b}$$

$$\mathit{Ratio} = \frac{\mathit{Top}_{\text{normalized}}}{\mathit{Floor}_{\text{normalized}} + \mathit{Top}_{\text{normalized}}}. \tag{6.8c}$$

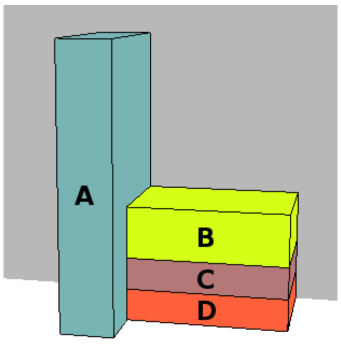


FIGURE 6.1: *Types of Stacks.* A is a stack consisting of a single box (i.e., it is the topmost box and on the floor), whereas B,C,D is a stack that is composed of multiple boxes.

In the available user-generated solutions, the number of times the current box has been above or below any of the other boxes in each of the problem instances, has been counted. With these counts, a *roulette wheel* selection routine can be created with each portion of the “wheel” proportional to count that combination of above-box-type and below-box-type. The roulette wheel is used to determine where the current box is to be placed. If for all combinations the count is zero, each placement will be assigned an equal portion of the roulette wheel.

To accomplish the actual placement within the representation, the area and stack indices need to be set to those associated with the box that the current box is being placed underneath or on top of. The order in the representation determines the order in which the boxes are placed in the container.

To place the current box underneath of another box, its step (b, a, s) is put in the representation directly before that of the box it is to be placed underneath of. To place the current box on top of another, it is inserted directly after it in the representation. When using this insert-implementation, the boxes that are to be placed on the same stacks will thus be sorted in the representation.

6.2.2 Stack-Mutation Operator

The informed box-mutation operator only moves boxes in relation to other boxes, and cannot operate on entire stacks. To alleviate this shortcoming, a stack-mutation operator is introduced that swaps entire stacks, as described in Algorithm 8. Before the actual stack-mutation is performed, the self-adaptive part $\mathrm{mut}_{\mathrm{magn,sa,stack}}$ of the stack-mutation magnitude $\mathrm{mut}_{\mathrm{magn,stack}}$ is updated according to Equation 6.3.

Algorithm 8 Stack-Mutation

```
1: function StackMutation($\vec{r}$)
2:     $mut_{\text{magn,stack}} \leftarrow$ UpdateMutMagn($mut_{\text{magn,stack}} \in \vec{r}$)
3:     $effectiveMutMagn \leftarrow mut_{\text{magn,stack}} * \frac{n}{numStacks}$
4:     for $(a, s) \in$ StackList($\vec{r}$) do                    // Get stack ID $(a, s)$
5:         if Random($[0, 1]$) $< effectiveMutMagn$ then    // Random value between 0 and 1
6:             $(a', s') \in$ StackList($\vec{r}$)                  // Get stack ID $(a', s')$
7:             SwapAll($(a, s), (a', s'), \vec{r}$)
8:         end if
9:     end for
10:    SwapFirstOccurrences($(a, s), (a', s')$)            // Swap stacks in building order
11: end function
```

Expressing the mutation magnitude for the stack-mutation directly as a fraction of the number of stacks is not possible as the number of stacks in a solution may vary over the course of the optimization run. It is therefore expressed in terms of the number of boxes n and converted before application using a conversion factor of $\frac{n}{numStacks}$.

For initializing this *effective* magnitude of the stack mutation to $\frac{2}{numStacks}$, we initialize $\text{mut}_{\text{magn,stack}}$ to $\frac{2}{n}$ by taking

$$mut_{\text{magn,sa,stack}} = \frac{2}{n} - mut_{\text{magn,min}} \tag{6.9}$$

(see Equation 6.2). An individual $\vec{r}$ in the SAGA is then extended to:

$$\vec{r} = (L, mut_{\text{magn,box}}, mut_{\text{magn,stack}}). \tag{6.10}$$

To bring about the actual mutation of two stacks, all occurrences of combinations of their area and stack indices a, s in the steps (b, a, s) are swapped with each other's, thereby swapping their contents in terms of the contained boxes. Also the location of the first steps concerning the stacks is swapped, so that the order in which the placement algorithm (Chapter 5) creates the stacks is also swapped.

6.3 Combining Mutation Operators

For every execution of the mutation operator, the algorithm needs to determine which of the different box-mutation operators is going to be used. For the box-mutation two possible mutations can be chosen: Uninformed box-mutation (see Section 6.1) and informed box-mutation (see Section 6.2.1). The choice between these two mutation-operators is determined by the $\mathit{frequency}_{\text{informed-box}}$ parameter provided to the algorithm, and is one of the parameters tested in the experiments (see Chapter 8).

The informed box-mutation manifests a sorting behaviour when applied. This is due to the fact that steps are inserted directly before or after other steps referring to the same stack and section. Since the stack mutation can be a very invasive mutation (it is able to change several boxes in one mutation step), a linearly increasing application frequency $\mathit{mut}_{\text{prob,stack}}$ is used for the decision to use stack-mutation. The $\mathit{mut}_{\text{prob,stack}}$ proposed in our earlier work [16] ranged from 0.05 to 0.25. When applying the stack-mutation with a linearly increasing rate, i.e. more stack-mutation towards the end of the optimization process, the negative influence is decreased since the informed-box-mutation has sorted the stacks in the representation.

It is beneficial to a apply the stack-mutation operator more frequently, as suggested in our earlier work [16]. Therefore, we increase the $\mathit{mut}_{\text{prob,stack}}$. Furthermore, we make the $\mathit{mut}_{\text{prob,stack}}$ dependent on the $\mathit{frequency}_{\text{informed-box}}$, since the stack-mutation operator has a negative effect on the uninformed box-mutation. This is accomplished by using

$$\mathit{mut}_{\text{prob,stack,start}} = 0.1 * (\mathit{frequency}_{\text{informed-box}}/0.25) \tag{6.11}$$

at the start of the optimization process, and linearly it increasing to

$$\mathit{mut}_{\text{prob,stack,end}} = 0.1 * (\mathit{frequency}_{\text{informed-box}}/0.25) + 0.15 \tag{6.12}$$

at the end of the process, where $\mathit{frequency}_{\text{informed-box}}$ is provided to the SAGA as a test-parameter. Thus, $\mathit{mut}_{\text{prob,stack}}$ can have starting values ranging from 0.0 to 0.4 increasing to end values ranging from 0.15 to 0.55. This means that when more informed-mutation is used, more stack-mutation is used.

In one mutation step, the stack-mutation operator is applied first. Due to the indirect representation, first applying the box-mutation would mean that in the actual loading, these boxes might not fit in their new stacks. Thus, the following stack-mutation would be applied to (possibly) invalid stacks.

Chapter 7

Penalty Function

In the optimization of HCTL-problem instances, a lot of *objectives* have to be taken into account. The performance of a solution in each of the objectives is expressed in the form of a *penalty*. These different penalties are combined using *weighted aggregation*, where each of these so-called *sub-penalties* is summed after being multiplied with a certain *weight*. The weight assigned to each a sub-penalties as well as the sub-penalties themselves have been developed in consultation with industry, but some are not optimal for use in automated solving.

Some sub-penalties are expressed as *non-continuous* functions, where a penalty is incurred only after a certain *threshold* has been reached. Other sub-penalties keep a fixed value once a certain threshold has been met. Both of these sub-penalties thus closely resemble *constraints*, that would indicate a true–false distinction. These types of sub-penalties, in combination with regular objective-functions, are very difficult to optimize with the use of a selection-based iterative search method, as the penalty value does not indicate a difference between two candidate solutions where one of the candidates is closer to the aforementioned threshold, yet both candidates incur the same penalty value.

In this chapter, a comprehensive overview of the used sub-penalties is given in Section 7.1, and a listing of the original weights assigned to those in Section 7.2. Furthermore a method is described as put forward by Eiben *et al.* [7], for the adaptation of these weights in a penalty function to solve *Constraint Satisfaction Problems*. However, this method is not applicable to the HCTL problem-instances we are trying to solve in this thesis. Therefore a method is described that *corrects* the weights by analysing user-generated solutions in Section 7.3.

7.1 Overview of Sub-Penalties

This section will give an overview of all sub-penalty functions contributing to the penalty-function. These sub-penalties have been determined in consultation with industry. For some sub-penalties a converge value has been defined that helps the GA rank solutions that would otherwise have the same penalty value.

7.1.1 Left Over

This sub-penalty is used to reflect the fact that when boxes are left out, another delivery needs to be scheduled to deliver the boxes to their respective clients.

Notion 7.1 Left Over Penalty. *The Left Over Penalty is defined as the number of boxes that have not been placed in the solution under evaluation.*

7.1.2 Client Order

To represent the number of loading/unloading actions a fork-lift driver has to perform to load/unload all boxes for a certain client, the client order penalty is introduced. This penalty is increased for each extra unloading action a fork-lift driver would have to perform. This means that clients that are to be serviced first are preferably placed on top of stacks.

Notion 7.2 Client Order Penalty. *The Client Order Penalty is determined by evaluating for each client, in the order in which they are to be serviced, how many packages need to be removed before all boxes for the current client can be unloaded.*

7.1.3 Client Side

This sub-penalty is two-fold. The first part deals with the fact that it takes a couple of minutes for the fork-lift driver to remove the flaps on one side of the trailer. If boxes for a client are loaded on both sides, both flaps need to be opened, which will take extra time. To reflect this, the Client Side Penalty is increased when a client has boxes on both sides of the trailer. However, when the total weight of all boxes of a client exceeds 4000kg, both parts of the penalty are ignored, due to the fact that if those 4000kg was to be loaded on one side of the trailer, the trailer would become very unstable once that client's boxes have been unloaded (Section 7.1.4).

Notion 7.3 Client Side Penalty. *The Client Side Penalty is expressed as the number of clients that have boxes on both sides of the container, or have boxes that are not on* *the client's preferred side. This is penalty only incurred if and only if the total weight of the client's boxes combined is below* 4000 kg.

Notion 7.4 Client Side Penalty Convergence Value. *The Client Side Penalty Convergence Value is defined as the number of boxes that are placed on the wrong side of the container. If a client's boxes are placed on both sides of the container, the* wrong *side is defined as the side with the least amount of boxes.*

The second part of the sub-penalty deals with the fact that some clients have a preferred side for unloading their boxes, due to available loading docks, for example. If no client side penalty is incurred for a given client, the client side preferred penalty is incurred.

Notion 7.5 Client Side Penalty Preferred. *The Client Side Penalty Preferred is the number of clients that have boxes that are not on the preferred side.*

Notion 7.6 Client Side Penalty Preferred Convergence Value. *The Client Side Penalty Preferred Convergence Value is defined as the number of boxes that are not on the preferred side.*

7.1.4 Weight Distribution

To ensure that the trailer does not have the risk of falling over, the following sub-penalty is used:

Notion 7.7 Weight Balance Side-to-Side Penalty. *The Weight Balance Side-to-Side Penalty is defined as the difference in weight of the boxes loaded on each side of the container. If the weight on the boxes on the left side is the biggest, then the penalty is halved.*

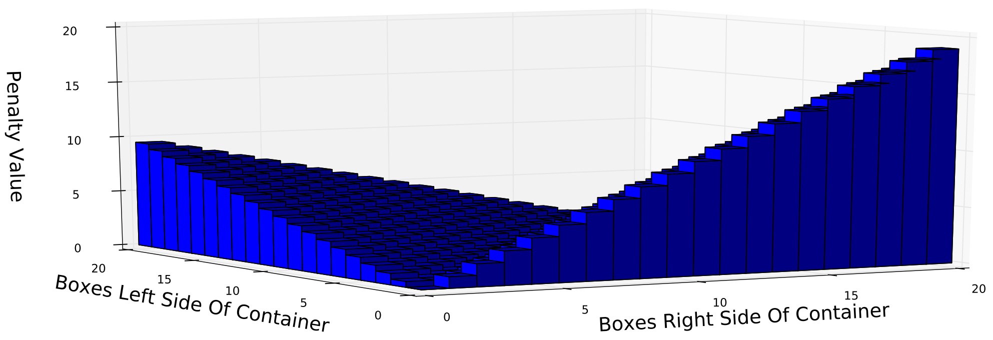


Figure 7.1: *Side-To-Side Weight Balance Penalty.* All boxes are of equal size and weight. The only thing changed is the number of boxes on each side of the container. The penalty weight balance penalty on the left side of the container is lower than on the right side.

The curvature of the road (when driving on the right side of the road) will put a higher strain on the right side of the suspension. The penalty function reflects this by penalising a weight imbalance on the right side more severely than on the left (Figure 7.1).

7.1.5 Under Bridge

If the container has a bridge, every box that is placed underneath the bridge will not be assigned certain penalties. Due to this nullification of certain penalties, the SAGA optimizes the penalty value by placing as many boxes as possible underneath the bridge. Since all these boxes need to be removed by hand (a fork-lift is unable to reach here) some boxes should not be placed underneath the bridge. To this end, the Under Bridge Penalty is introduced. This penalty is incurred when a box that should not be placed underneath the bridge is placed there.

Notion 7.8 SmallBox. *A box is considered a SmallBox when the length or width is smaller than* 400 mm.

Notion 7.9 Under Bridge Penalty. *The Under Bridge Penalty is expressed as the total volume of all boxes that are placed under the bridge, except for SmallBoxes.*

7.1.6 Stack Height

This sub-penalty is used to penalise the stacks that are too high and therefore unstable. This function grows exponentially, after certain conditions have been met.

Notion 7.10 Stack Height Penalty. *The Stack Height Penalty is defined by, for each stack in the container that is bigger than* 1500 mm *and contains more than 4 boxes. If these conditions have been satisfied, the penalty is calculated according to the* Stack Height Formula

$$\text{Stack Height Formula} = (\#\text{Boxes} - 1)^2 \tag{7.1}$$

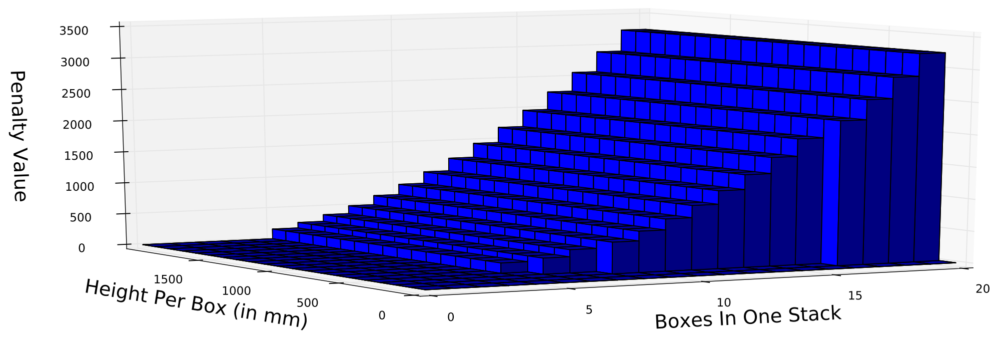


FIGURE 7.2: *Stack Height Penalty.* The penalty increases with the number of boxes in a stack. When boxes are smaller, more are needed to reach the 1500 mm height threshold.

7.1.7 Stack Pattern

This sub-penalty is used to represent the stability of a loading. It increases when stacks are able to slide or fall over. It consists of two parts: plus and minus. The plus part of the penalty reflects the possibility of stacks moving when the truck accelerates and the minus penalty for when the truck brakes. This penalty has been provided by industry.

Notion 7.11 Stack Pattern Penalty. *The stack pattern penalty is defined by going through all stacks from back to front. For each stack, the stacks that are directly behind it difference in height is multiplied by the width of the box that is behind the current box. This is multiplied by the "Distance Bonus" defined below. If the current box is bigger than the box behind it, the value is added to the plus-penalty, otherwise it is added to the minus-penalty.*

This penalty might be rather abstract, but can be viewed as weighted surface area of the sides of stacks, that are not supported by other stacks. The weighting is determined by the distance d between the current stack and the stacks are that do support it.

$$\text{Distance Bonus} = \frac{1}{1 + \epsilon^{-14*(d-400)}} \tag{7.2}$$

7.2 Sub-Penalty Weights

In the original penalty function, the following weights were assigned to the different sub-penalties. The weighting of all these sub-penalties, has been established in consultation with industry.

Table 7.1: *Initial Weights.* The initial weights assigned to each of the sub-penalties. Since the Client Side Penalty, Stack Patter Penalty and Balance Penalty are described by several components, each component has its own weighting. The approximate values of the sub-penalties (after multiplication with their respective weights) are listed in the last column.

Sub-Penalty	Weight	Approximate Value Range
left_over	100 000	100 000 - 1 000 000
under_bridge	500	0 - 1500
client_side	100	0 - 500
client_side, convergence	10	
client_side, preferrence	5	
client_side, preferrence, convergence	1	
client_order	10	0 - 200
stack_pattern_minus	50	1000 - 6000
stack_pattern_plus	10	
stack_height	10	0 - 500
balance,right	10	0 - 1000
balance,left	5	

7.3 Correction of Sub-Penalty Weighting

Given a set of penalties that describe certain undesirable traits of a solution, an importance (*weight*) needs to be assigned to each of these penalties. The initial weights have explicitly been determined by interviewing experienced human planners. To correct these initial weights of each of the objectives, the available user-generated solutions can be used to determine how often the user accepts a solution with certain penalty values.

If a user accepts high values for a sub-penalty, it is apparently of less importance to the user. Thus, it should be valued less when evaluating the quality of a solution generated with the use of the automated-solver. To accomplish this, we propose a *correction factor* for each of the sub-penalties.

7.3.1 Stepwise Adaptation of Weights

In work by Eiben *et al.* [7], a method is described to apply adaptation of weights assigned to sub-penalties in order to solve instances of the *Constraint Satisfaction Problems* (CSP). In CSP, a set of constraints all need to be satisfied for the problem instance to be considered solved. In their method, Eiben *et al.* describe that if a constraint is not satisfied for a certain amount of generations, i.e. it is hard to solve, the weight assigned to that sub-penalty is increased with a fixed value. Thus, the penalties with a higher weight will have more influence in the selection operator of the GA. Initially, all weights are equal.

The sub-penalties used by Eiben *et al.* are defined per constraint. If a constraint χ_i is not satisfied, a penalty of 1 is incurred otherwise the penalty value is 0. The sub-penalties for all constraints are combined into one penalty value using weighted aggregation, i.e. by multiplying them with their assigned dynamic weight, described above.

TABLE 7.2: *Method Comparison.* A comparison between the method proposed in this thesis and the SAW method as described by Eiben *et al.* [7].

	This Thesis	SAW
Sub-penalties	Varying scale	Equal scale
Importance of Sub-penalties	Varying importance	Equal importance
Goal of Weight Adaptation	Determine true weights	Artificial weighting so GA can solve CSP
Scope of Adaptation	Dependant on user	Within GA

Table 7.2 shows the differences in approach for adjusting weights between Eiben *et al.* [7] and the method proposed in this thesis Section 7.3.2. The method proposed by Eiben *et al.* deals with sub-penalties that are of equal weights and of equal importance, whereas the HCTL-problem we are studying is evaluated by sub-penalties that are of varying scale and varying importance. The goal of adapting the weights for Eiben *et al.* is to assist the GA in solving their problem, whereas we are determining the actual importance of each of the sub-penalties by analysing user-accepted solutions.

7.3.2 Correction Factor for Sub-Penalty Weight

The correction factor for the initial weight of a sub-penalty is determined using a collection of user-accepted solutions. Since solutions for problem instances containing a larger amount of boxes are likely to get higher *absolute* penalties, the *average* for each sub-penalty per box $P_{\text{box},i}$, weighted according to the initial setting, is calculated per solution instead. Otherwise, solutions with more boxes would implicitly get a larger vote, disturbing the per-solution accepted ratio between the sub-penalties.

Thus, per solution Υ, a weighted sub-penalty value per box is calculated,

$$P_{\text{box},i}(\Upsilon) = \frac{P_{\text{sol},i}(\Upsilon) * w_i}{\text{numboxes}(\Upsilon)}, \tag{7.3}$$

where $P_{\text{sol},i}(\Upsilon)$ denotes the i-th sub-penalty, taken times its initial weight, and numboxes(Υ) denotes the number of boxes in that solution.

Two ways of calculating the correction factors for the initial weights of the sub-penalties will be compared. For the first approach, we take the *sum* of an initially weighted sub-penalty i per box, over all user-accepted solutions $\Upsilon_1, \Upsilon_2, \ldots, \Upsilon_m$,

$$\sum_{j=1}^{m} \left(P_{\text{box},i}(\Upsilon_j)\right). \tag{7.4}$$

Next, we determine the total of these sums, of all initially weighted sub-penalties per box $P_{\text{box},1}, P_{\text{box},2}, \ldots, P_{\text{box},n}$ (expressed over all user-accepted solutions $\Upsilon_1, \Upsilon_2, \ldots, \Upsilon_m$),

$$\sum_{i=1}^{n} \left(\sum_{j=1}^{m} \left(P_{\text{box},i}(\Upsilon_j)\right) \right). \tag{7.5}$$

Of this total, the portion can be determined that resulted from a certain initially weighted sub-penalty. This portion reflects how important this weighted sub-penalty was deemed to be, namely, larger portion $\leftrightarrow$ less importance. As such, we take the correction factor for the initial weight of the i-th sub-penalty as the inverse of its fraction,

$$C_{\text{summed},i} = \frac{\sum_{i=1}^{n} \left(\sum_{j=1}^{m} \left(P_{\text{box},i}(\Upsilon_j)\right) \right)}{\sum_{j=1}^{m} \left(P_{\text{box},i}(\Upsilon_j)\right)}. \tag{7.6}$$

Some problem instances might require the user to drastically alter the weighting of the sub-penalties for that particular solution. To prevent these *outliers* from skewing the correction factors, a second method of determining the correction factors is proposed, based on the *averages* of the initially weighted sub-penalties per box, averaged over all user-accepted solutions $\Upsilon_1, \Upsilon_2, \ldots, \Upsilon_m$,

$$C_{\text{averaged},i} = \frac{\sum_{i=1}^{n} \left(\frac{\sum_{j=1}^{m} \left(P_{\text{box},i}(\Upsilon_j) \right)}{m} \right)}{\frac{\sum_{j=1}^{m} \left(P_{\text{box},i}(\Upsilon_j) \right)}{m}}. \tag{7.7}$$

After these inverted fractions have been determined, they need to be normalized. To this end the sum over all the non-normalized correction factors is calculated,

$$C_{\text{total, summed}} = \sum_{i=1}^{n} \left(c_{\text{summed},i} \right) \tag{7.8}$$

$$C_{\text{total, averaged}} = \sum_{i=1}^{n} \left(c_{\text{averaged},i} \right). \tag{7.9}$$

After which, the normalized correction factors $c_{\text{summed},i}$ and $c_{\text{averaged},i}$ can be calculated,

$$c_{\text{summed},i} = \frac{c_{\text{summed},i}}{c_{\text{total, summed}}} \tag{7.10}$$

$$c_{\text{averaged},i} = \frac{c_{\text{averaged},i}}{c_{\text{total, averaged}}}. \tag{7.11}$$

In determining the correction factors for the sub-penalties, the left over penalty (Section 7.1.1) cannot be taken into account, since the building function (described in Chapter 5) does not have the option to leave any of the boxes out. This is the reason behind the normalization of the correction factors, since using the non-normalized correction factors would disrupt the relationship between the left over penalty and the other sub-penalties. The convergence values will also be excluded from the calculation of the correction factors, since they were only introduced to assist the GA.

Chapter 8

Experimental Setup

In this chapter, a description is given of the experiments that have been performed to test the performance of the proposed methods put forward in this thesis. The experiments performed in our earlier work [16] are repeated. Furthermore, tests are performed on the *informed building function* (see Chapter 5) to determine its performance both in the terms of quality of a solution (number of boxes, stability etc.) as well as the number of *violations* of the *user-derived placement rules.*

8.1 The Data Set

The data set used in our experiments consists of 1449 problem instances provided by industry. For each of these problem instances, a solution by an experienced human planner is also provided.

For the experiments, the data set is split up into three parts: A *training set*, a *run set*, and a *test set.* The training set, 650 problem instances, is used to generate a two-dimensional contingency table (see 4.1.3) for training the solver, whereas the test set, using another 650 instances, gives rise to a contingency table for determining placement violations. The remainder of the data set, 149 instances, is used for running the solver on, after which performance with respect to placement violations and penalty values is determined.

To generate more reliable results, a cross-validation scheme is used in which the split into training, run, and test sets is performed five times. Per split, a certain problem instance can *only* occur in *one* of the sets, that is, the sets are completely disjunct. Through this cross-validation scheme, 745 runs of the solver are obtained for determining its performance over. For the experiments with the correction factors for the weights of the penalty function, the test set and run set have been combined into a set of 1300 problem instances, which are used to determine the correction factors.

8.2 Variation Operator

Our experiments for the *variation operator* (see Chapter 6) are three-fold: First, the influence of the *informed box-mutation* and *stack-mutation* operator on the *penalty value* is determined. Second, the effect of mutation operators is investigated on the number of *placement violations*. Another part of the experiments is the investigation of the *ratio* of combining the existing uninformed-mutation operator and the newly defined informed-mutation operator. For placement violation, the following definition is used:

Notion 8.1 Placement Violation. *A placement violation occurs when the placement of a box with above-box-type a on a box with below-box-type b was done less than twice by the human planner in the test data set.*

The human planner making a mistake once has, according to this definition, no effect on what is counted as a violation. Violations will be determined by comparing a generated loading to the rules defined by a separate *test set*.

8.3 Building Function

The experiments performed for testing the performance of the informed building function are two-fold: First, the influence of the search depth (see Section 5.3) for non-violating placements is tested. Second, various levels of search depth are combined with three different variation operators. These combinations should show if the building function has any effect on the performance of the variation operators.

8.4 Penalty Function

Van Rijn *et al.* [21] showed that their results are on par with solutions generated by the human planners. However, they conclude that the penalty function does not describe the undesirable traits of a solution accurately enough to help the automated solver in coming up with solutions that users would immediately accept. To determine whether the corrections proposed on the sub-penalty weighting have the desired effect of better representing validity of a loading, we will take the solver-generated solutions and the user-generated solutions, and evaluate these using the corrected penalty function.

8.5 Test Parameters

In the experiments, several ratios of using informed box-mutation will be tested. These range from completely uninformed to fully informed, in four steps. The ratios are tested with both the stack-mutation enabled and disabled.

For testing the informed building function, three different settings for the variation operator are used: uninformed without stack-mutation, 50% informed with stack mutation, and completely informed with stack mutation. The informed building function is tested as well as a *hybrid* version, where it only searches a number of areas for a non-violating placement. Two hybrid versions are tested: Only search the current area, and search both the current area and the corresponding area on the other side of the container. Table 8.1 presents an overview of the other test parameters.

Table 8.1: *Test Parameters.*

Parameter	Possible Values
Minimum number of bins	Automatically determined
Maximum number of bins	25 bins
Evaluation budget	$10,000$ evaluations
Population sizes	$(5, 35)$

8.5.1 System Specification

A run of the solver with 10,000 evaluations takes about five to eight minutes, depending on the number of boxes. It is a single threaded program. The system specifications of the systems used are comparable to a Intel i5 (4th generation) with 8 GB of system memory. The entire solver is run in a Linux environment. All the code used for the solver is written in Python [20].

Chapter 9

Empirical Study

This chapter gives an overview of the results of all *experiments* that have been performed to test the influence of the proposed improvements for the automated solver of *Highly Constrained Truck Loading* problems. These experiments have been divided into four parts: *Variation operator*, *building function*, *penalty function* and *run-time analysis*. All box-plots presented in this chapter represent a comparison between solutions generated with a certain default parameter setting (denoted by a grey line) and solutions generated with a test parameter setting.

9.1 Variation Operator

For each of the different test-parameter sets (Section 8.5), we need to measure the performance. This is done by making two comparisons:

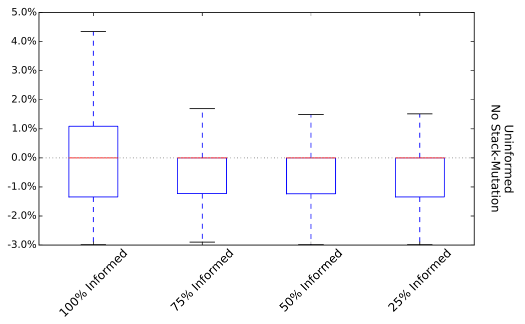
- For each level of informed box-mutation, executing the solver with and without stack-mutation is compared;
- The four levels where informed box-mutation are actually used (25%, 50%, 75%, 100%, with stack-mutation), are compared to the parameter settings used by Van Rijn *et al.* [21], i.e., uninformed without stack-mutation (*default parameters*).

Three metrics will be used to determine the performance of automated solver with each of the experimental settings.

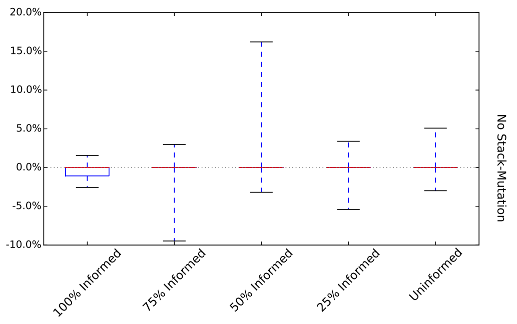
- Percentage of boxes left over.
- Quality of placement of loaded boxes.

- Percentage of violating placements.

9.1.1 Boxes Left Over



(a) *Difference in Percentage of Boxes Left Over (default).* Comparison of percentage of boxes left over between the four levels of informed mutation compared and unininformed mutation.



(b) *Difference in Percentage of Boxes Left Over (stack).* Comparison of percentage of boxes left over between mutation operators, with and without stack-mutation.

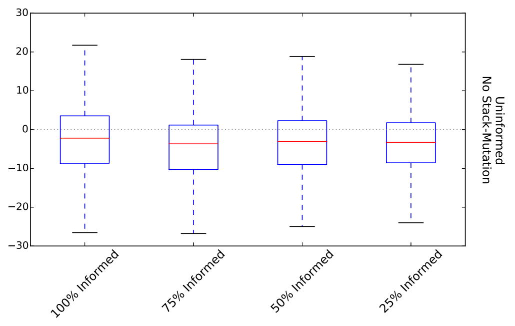
Figure 9.1: *Influence of Mutation Operators on Boxes Left Over.* The results obtained with each of the informed mutation rates, show that less boxes have been left over when 25 % to 75 % informed mutation was used. When 100 % informed mutation was used, the number of boxes that is left over is roughly the same as when the default parameters were used. Stack mutation has a positive effect when a 100% fraction of informed mutation is used.

The first and foremost quality measure of a solution is the number of boxes that could not be placed into a solution. The box plot in Figure 9.1(a) shows that when informed mutation is used (in combination with stack-mutation), percentage of boxes left over is decreased. Figure 9.1(b) shows the effect of stack-mutation in combination with the various mutation operators. Stack-mutation generally has an improving effect when more informed mutation operator is used. Also, a slight negative effect can be observed when the stack-mutation operator is used in combination with the uninformed mutation operator. Table 9.1 shows the number of boxes that have been left over by the various combinations of mutation operator, and show the same effects as described before.

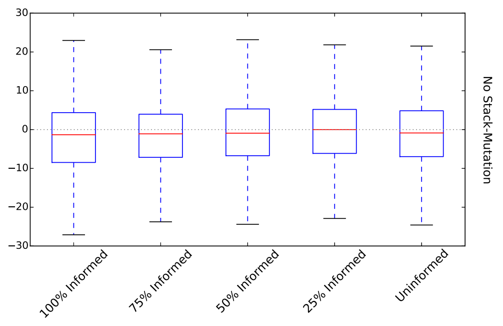
TABLE 9.1: *Number of Boxes Left Over.* Summed over all solutions. Stack mutation alleviates negative effect of informed mutation on the number of boxes left over. Stack mutation has a negative effect when combined with uninformed mutation.

	Without Stack Mutation	With Stack Mutation
Uninformed	625	630
25% Informed	615	617
50% Informed	622	622
75% Informed	631	609
100% Informed	660	619

9.1.2 Quality of Loaded Boxes



(a) *Difference in Penalty per Loaded Box (default).* Comparison of penalty per loaded box between the four levels of informed-mutation (with stack-mutation) compared and unininformed mutation.



(b) *Difference in Penalty per Loaded Box (stack).* Comparison of penalty per loaded box between mutation operators, with and without stack-mutation.

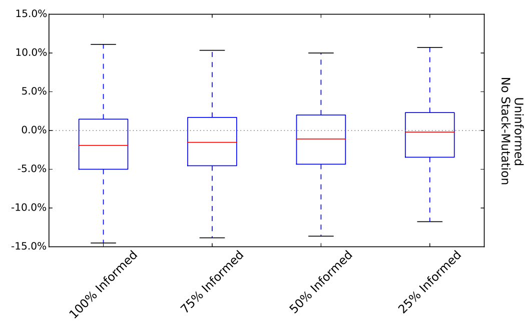
FIGURE 9.2: *Influence of Mutation Operators on the Penalty Per Loaded box.* Informed mutation yields a slight improvement in penalty value per loaded box when compared to uninformed mutation. Stack mutation no real effect on the penalty per loaded box.

The plot in Figure 9.2(a) describes the effect the informed mutation has on the quality of the placement of the loaded boxes. A slight improvement can be observed in the penalty per box, when any ratio of informed mutation is used. The same holds for when stack-mutation is used (see Figure 9.2(b)). Table 9.2 shows the average penalty per box for all the solutions generated with a certain set of parameters.

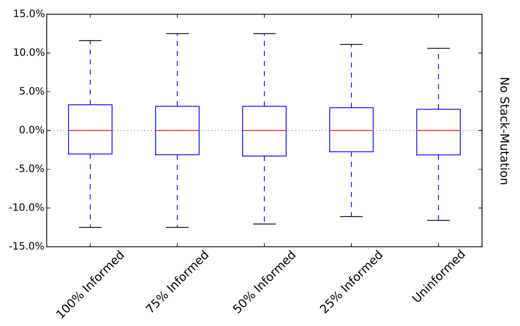
Table 9.2: *Average Penalty per Loaded Box.* Averaged over all solutions of the problem instances. Average penalty value is consistent across all test parameters.

	Without Stack Mutation	With Stack Mutation
Uninformed	48.04	47.34
25% Informed	44.98	44.61
50% Informed	45.51	44.52
75% Informed	45.47	43.77
100% Informed	47.61	45.73

9.1.3 Violating Placements



(a) *Difference in Percentage of Violating Placements (default).* Comparison of percentage of violating placement in solution between the four levels of informed mutation compared and unininformed mutation.



(b) *Difference in Percentage of Violating Placements (stack).* Comparison of percentage of violating placement in solution between mutation operators, with and without stack-mutation.

Figure 9.3: *Influence of Mutation Operators on Placement Violations.* More informed mutation causes less violations, stack mutation has no influence.

Figure 9.3(a) shows the improvement in the percentage of violating placements when more informed mutation is used. Additionally, Figure 9.3(b) shows that using stack-mutation in the automated solver has no noticeable effect on the percentage of violating placements.

Table 9.3: *Number Of Violations.* Summed over all solutions of the problem instances. More informed mutation reduces the number of violations.

	Without Stack Mutation	With Stack Mutation
Uninformed	6310	6214
25% Informed	5932	5992
50% Informed	5880	5844
75% Informed	5702	5707
100% Informed	5372	5509

Table 9.3 shows the total number of violating placements that are present in the solutions. The same effects can be observed as described before, where informed-mutation reduces the total number of violations by about 12 %.

9.1.4 Run-Time

Table 9.4: *Average Run-Time Uninformed Building Function.* Averaged over 10 runs of the same problem instance. Informed mutation operator has no noticeable influence on run-time.

	Average	Standard Deviation
100% Informed, No Stack-Mutation	316 s	12 s
75% Informed, No Stack-Mutation	311 s	14 s
50% Informed, No Stack-Mutation	315 s	8 s
25% Informed, No Stack-Mutation	320 s	9 s
Uninformed, No Stack-mutation	316 s	14 s
100% Informed, Stack-Mutation	326 s	15 s
75% Informed, Stack-Mutation	318 s	8 s
50% Informed, Stack-Mutation	315 s	6 s
25% Informed, Stack-Mutation	315 s	9 s
Uninformed, Stack-mutation	316 s	10 s

Table 9.4 shows the average run-time of the automated solver with the various mutation operators. No real difference in run-time can be observed.

9.2 Building Function

Section 8.5 describes three sets of experiments for testing the *informed building function*. These three set will be compared to the results of the *uninformed building function*. The same metrics are used to measure the performance of the proposed building function, as have been used for the variation operator (see Section 9.1).

9.2.1 Boxes Left Over

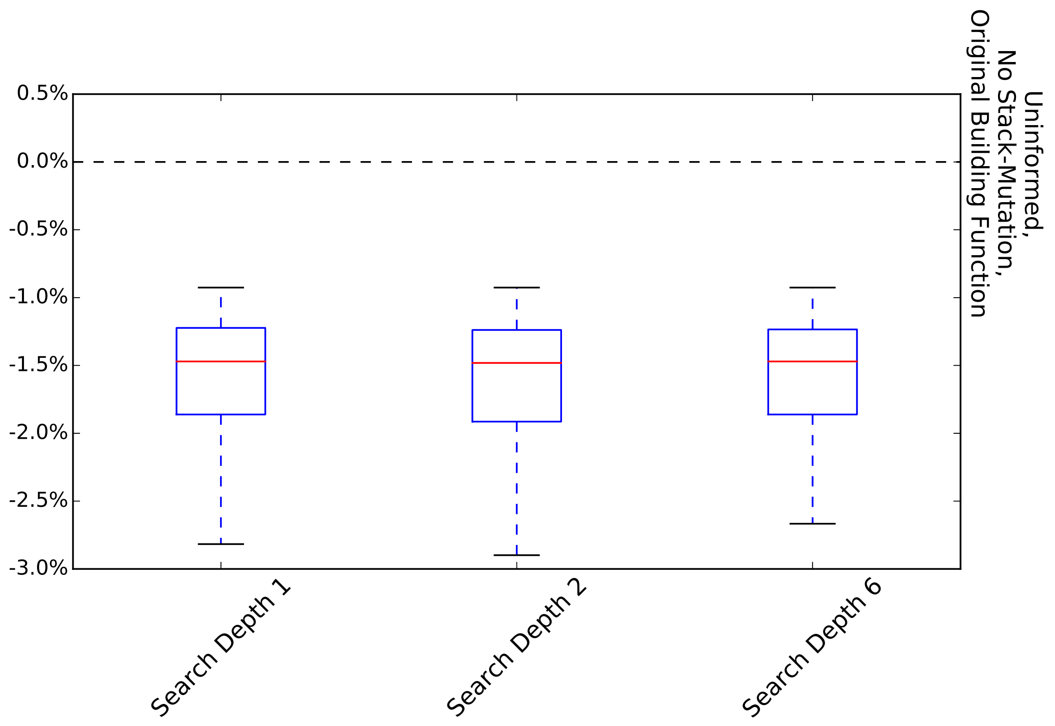


Figure 9.4: *Percentage of Boxes Left Over.* Comparison of percentage of boxes left over for the three levels of search depth. All three levels are able to fit about 1 %-2 % more boxes into a solution. The mutation operator used for the informed building function is: 100 %-informed, stack-mutation.

Both Figure 9.4 and Table 9.5 show that the percentage of boxes that are being left over is reduced when compared to the uninformed building function. Even though only the results from the informed building function in combination with the 100 %-informed stack-mutation is displayed, all three ratios of informed mutation (uninformed no stack-mutation, 50 %-informed stack-mutation and 100 %-informed stack-mutation) show the same performance increase in the number of boxes that have been left out of the solutions.

Table 9.5: *Number of Boxes Left Over.* Summed over all solutions. No noticable difference in number of boxes left over when looking at search-depth

	Search Depth 0	Search Depth 1	Search Depth 6
Uninformed, No Stack-mutation	516	516	507
50% Informed, Stack-mutation	513	516	508
100% Informed, Stack-mutation	518	512	513

9.2.2 Quality of Loaded Boxes

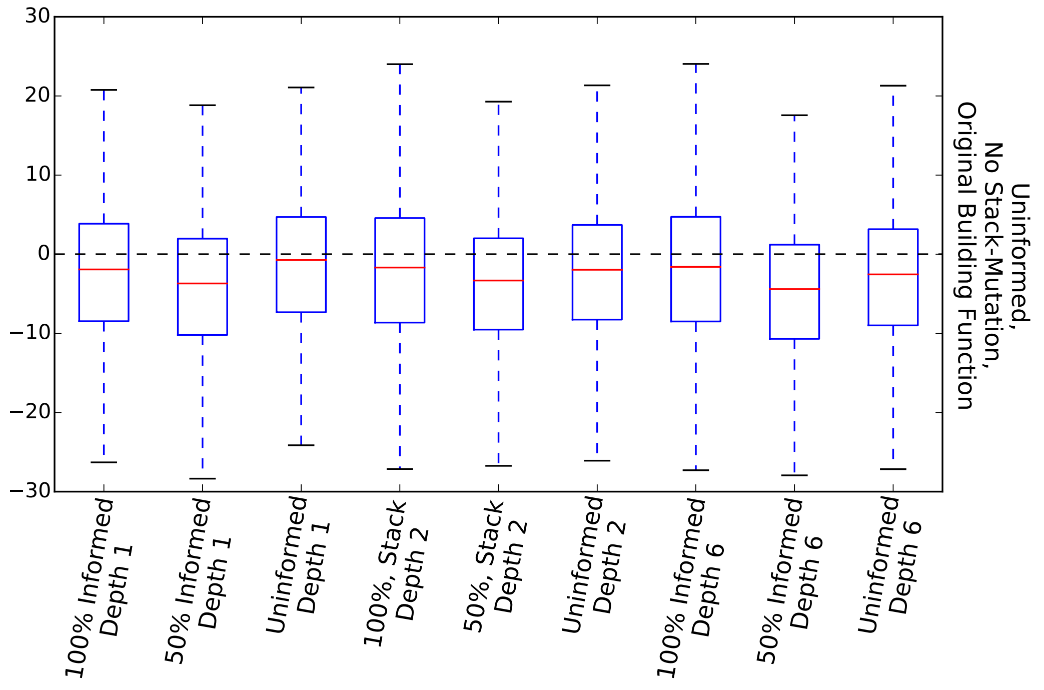


Figure 9.5: *Difference in Penalty Per Loaded Box.* The proposed building function has a slightly lower penalty loaded per box.

Figure 9.5 shows a slight decrease in the penalty per box that is incurred when the informed building function is used in the automated solver. Table 9.6 shows the average penalty per box for all the tested combinations for mutation operator and informed building function.

Table 9.6: *Average Penalty per Loaded Box.* Averaged over all solutions. Average penalty value is consistent across all test parameters.

	Depth 0	Depth 1	Depth 6
Uninformed, No Stack-mutation	47.01	45.73	45.20
50% Informed, Stack-mutation	44.03	44.16	43.11
100% Informed, Stack-mutation	45.95	45.99	46.00

9.2.3 Violating Placements

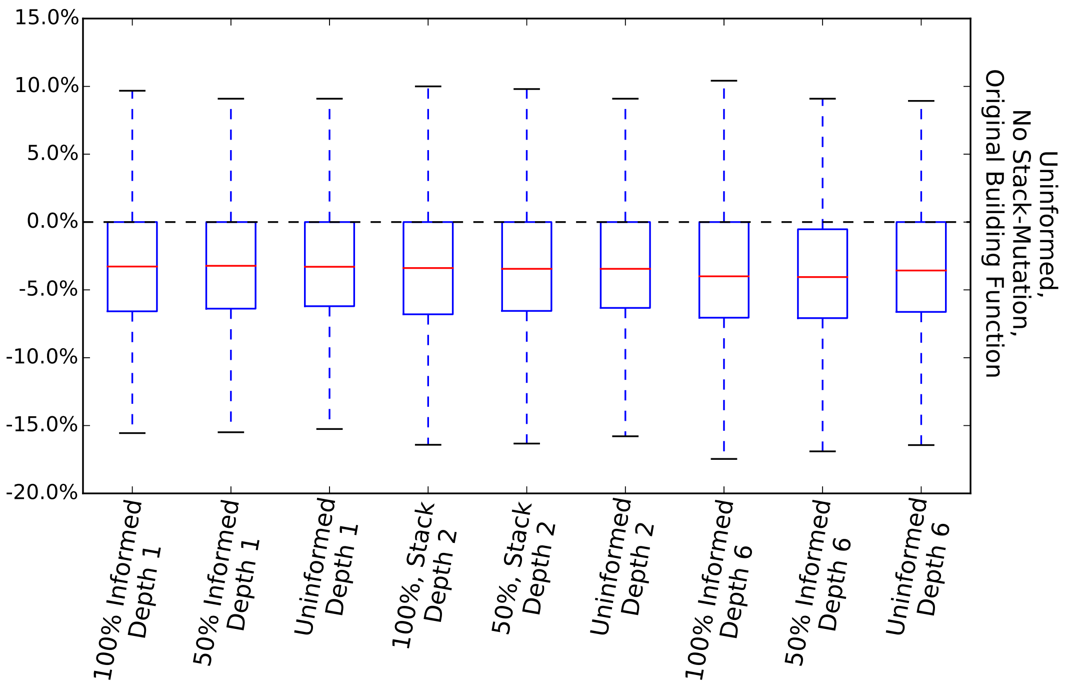


Figure 9.6: *Difference in Percentage of Violating Placements.* Comparison of the uninformed building function with the informed building function and the various mutation operators. All options get a similar score; less violations than the default parameter setting.

In the box plot of Figure 9.6 the difference in percentage of violating placements is displayed, where informed building function if able to decrease the percentage of violations. However, something that is difficult to observe from this figure is the fact that a deeper search prevents more violating placements. This result can more clearly be observed in Table 9.7.

Table 9.7: *Number of Violations.* Summed over all solutions. A deeper search reduces the number of violations.

	Search Depth 0	Search Depth 1	Search Depth 6
Uninformed, No Stack-mutation	4817	4753	4573
50% Informed, Stack-mutation	4774	4706	4434
100% Informed, Stack-mutation	4739	4603	4520

9.2.4 Run-Time

Table 9.8: *Average Run-Time Informed Building Function.* Averaged over 10 runs of the same problem instace. Deeper search takes longer.

	Average	Standard Deviation
Builing-Depth 0, 100% Informed, Stack-Mutation	809 s	56 s
Builing-Depth 0, 50% Informed, Stack-Mutation	844 s	40 s
Builing-Depth 0, Uninformed, No Stack-Mutation	840 s	62 s
Builing-Depth 1, 100% Informed, Stack-Mutation	898 s	68 s
Builing-Depth 1, 50% Informed, Stack-Mutation	911 s	40 s
Builing-Depth 1, Uninformed, No Stack-Mutation	939 s	54 s
Builing-Depth 6, 100% Informed, Stack-Mutation	991 s	65 s
Builing-Depth 6, 50% Informed, Stack-Mutation	1018 s	78 s
Builing-Depth 6, Uninformed, No Stack-Mutation	1004 s	66 s

When Table 9.8 is compared to Table 9.4, a significant increase in run-time can be seen, where the run time at least doubles. Furthermore, when the search depth is increased in the informed building function, the average run-time increases as well.

9.3 Penalty Function

To measure the influence of the correction factor for the sub-penalty weights as described in Chapter 7, we will use 2 sets of solutions. The first set is comprised of user-generated solutions. The second set consists of solutions generated by the automatic solver. Even though the automatically generated solutions perform well when the non-corrected penalty is used to evaluate them, domain experts have indicated that these solutions are of insufficient quality to be used in practice. Thus, we have one set of “good” solutions and one set with “bad” solutions namely, user-generated solutions and automatically generated solutions respectively. These two sets will be compared by their penalty values with the non-corrected sub-penalty weights and both the proposed corrected sub-penalty weights.

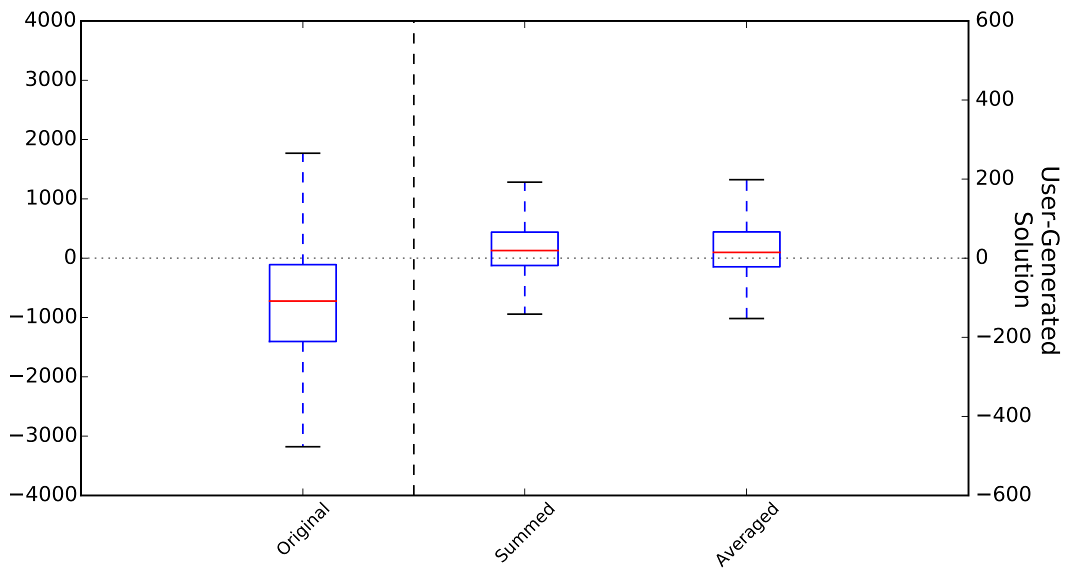


FIGURE 9.7: *Difference in Penalty Value.* Comparison of user-generated solutions with solver-generated solutions. Corrected sub-penalty weights rank solutions more accurately than non-corrected sub-penalty weights. Since penalty values are no longer in the same range, the left axis denotes the penalty value for the original and the right axis denotes the penalty values for the corrected penalties.

When we observe Figure 9.7, we can see that, when using the non-corrected sub-penalty weights, our solutions incur a lower penalty value for the loaded boxes. However, when the corrected sub-penalty values are used, the solver-generated solutions perform worse than the user-generated solutions. Moreover, a subjective analysis of the solutions provided by the automated solver that makes use of the corrected sub-penalty weights indicates that those solutions show a lot more of the characteristics seen in solutions generated by human planners.

Chapter 10

Conclusions and Future Work

Truck Loading problems from practice require a lot of objectives and constraints to be taken into account. Van Rijn *et al.* [21] propose to use a *Self-Adaptive Genetic Algorithm* (SAGA) to automatically solve instances of *Highly Constrained Truck Loading* (HCTL). This approach does not make use of any information provided by the user in generating and modifying solutions. In our earlier work [16] we introduce a method for deriving information from user-generated solutions. This information is then introduced into the optimization approach, to improve its performance. In this thesis we continue upon that work by integrating more user-derived information into the optimization process.

10.1 Building Function

Van Rijn *et al.* [21] propose an *uninformed building function* that converts *solution representations* into *actual solutions* that can be evaluated for quality. In Chapter 5 an *informed building function* is proposed that uses statistics derived from user-accepted solutions. This function gives preference to placements of boxes that have also been placed in that manner by the user.

The results presented in Section 9.2 show that the use of the informed building function reduces the percentage of violating placements in the automatically generated solutions. Another benefit of the informed building function is that it is able to load more boxes into the container, in many of the solutions. This is can possibly be attributed to two factors that are introduced by the proposed building function.

First, the informed building function gives preference to stacking boxes, as opposed to creating new stacks, to accommodate a box. The uninformed building function does not have this preference. The second factor that influences the number of boxes that can be loaded in a solution comes from the fact that placement of boxes is performed according to statistics that have been derived from the user. This causes boxes with similar dimensions to be stacked on top of each other, since this is how the user usually places these boxes. This is usually more space-efficient.

However, the informed building function comes with a drawback: It takes the automated solver over twice as long to produce a solution. This is due to the fact that the building function needs to investigate a lot more possible placements to find one that conforms to the user-derived placement rules.

The penalty per box that is incurred shows no real improvement by using the informed building function. This is remarkable, since each of the penalties can only increase when an extra box is placed into a solution, and we are able to fit more boxes into the solutions.

10.2 Variation Operator

The paper by Van Rijn *et al.* [21] uses an *uninformed mutation operator*. In our earlier work [16], we proposed an *informed mutation operator* that uses information from user-generated solutions (see Chapter 6). Moreover, a *stack-mutation* operator was introduced to alleviate the fact the informed mutation operator can only place boxes in relation to other boxes, which reduced the freedom of variation.

We put forward in our earlier paper [16] that it might be beneficial to increase the *frequency* of the stack-mutation operator, but that further investigation was needed to determine that with certainty. To this end, we proposed a new method that determines the frequency of stack-mutation. Not only is this rate higher than in our earlier paper, we also propose to make it depend on how much the informed mutation operator is used. The informed mutation operator has an *inherent sorting behaviour* for boxes that are in the same stack. Since boxes are planned in the order they appear in the representation and the informed mutation operator puts boxes that are in the same stack directly after each other, the invasiveness of the stack mutation operator is reduced.

In our earlier paper, the results showed that when only the informed mutation operator is used (in combination with stack mutation), the number of boxes that could be fit into a solution was less then when only the uninformed mutation operator was used. However, the results presented in this thesis (see Section 9.1) do not show this problem. These results were obtained with the use of the proposed stack-mutation frequency. This affirms the suspicion put forward in our earlier work [16], that the stack-mutation frequency was too low for the higher rates of informed mutation to alleviate the reduction of freedom in variation of the informed mutation operator.

Furthermore, the results reported in our earlier paper [16], are reaffirmed by the results presented in this thesis, that have been obtain over a different set of problem instances. The percentage of *violating placements* is reduced when more informed mutation is used in the optimization process. Moreover, a bigger number of boxes can be fit into solutions when informed mutation is used. The penalty per box that is incurred in the automatically generated solutions does not vary by a lot when comparing the various mutation operator strategies. The stack mutation operator seems to have no noticeable effect on the penalty per box.

10.3 Penalty Function

The penalty function described in Chapter 7 uses weighted aggregation (the multiplication of a sub-penalty with a certain factor) to combine the sub-penalties defined into one single penalty value. Experiments performed by Van Rijn *et al.* [21] and in our earlier work [16], indicate that automatically generated solutions are on par with or slightly better than solutions generated by the users. However, a subjective hand analysis of the automatically generated solutions shows that these solutions still have some shortcomings. This could indicate that the sub-penalties are not weighted properly, given that the sub-penalties themselves are accurate.

To alleviate this problem, two methods are proposed that determine a factor to correct the weighting of the sub-penalties. Both methods utilize user-generated solutions to determine the *correction factor*. Each solution is evaluated using the non-corrected sub-penalty weights. These values can then by used to calculate each correction factor by assuming that the user has accepted each of the sub-penalty values. Therefore, a sub-penalty with a high value is apparently of less importance to the user.

The results presented in Section 9.3 show that after the correction of the sub-penalty weights, automatically generated solutions generally get penalized more than the solutions put forward by the user. Further investigation of these observations is needed to determine the exact efficacy of each proposed correction factor.

10.4 Discussion

Even with the extensions proposed in our earlier work [16] and in this thesis, the automated solver still has shortcomings. These shortcomings are listed below.

New Types of Boxes. Whenever a new type of box is introduced, no information is available about it. By placing this new box, a user needs to indicate what needs to be done with this box. This gives a period wherein the automated solver is unable to load a box into a solution according to how it should be loaded. Furthermore, we still rely on the manually defined placement rules to at least give us some guidance as to which placement are undesirable.

Hard User-Derived Placement Rules. Another failing of the proposed approach occurs when the user-derived box-types are used to define hard placement rules, that indicate where boxes are allowed to be placed. Boxes that are quite similar could be given different box-types because one of the parameters that is used to determine the box-type is slightly different. This is due to the use of the binning algorithm (see Section 4.1.2), where close values can end up in different bins. To alleviate this problem, a different classification strategy might need to be applied to determine box-types.

Number of Boxes Left Over as Main Measure of Solution Quality. In the HCTL problems studied in this thesis, the number of boxes that are loaded into a solution is the biggest part of solution quality. However, a problem arises when a type of solution needs to be provided where this is not the case. The building function used to convert the indirect representation into a solution that can be evaluated, always tries to load as many boxes as possible into a solution. In order to solve instances of HCTL where the number of loaded boxes is not the main measure for solution quality, the building function should be extended to make a choice to place a box or not.

10.5 Future Work

For future work several subjects warrant further investigation

Improvements of the Variation Operator. An improvement for the variation operator would be a pre-processing step where, for each of the boxes in a problem instance, the areas are determined that the box actually fits in. Boxes that do not fit into certain area, should be excluded from being moved (via mutation or the building function) to that area. This should decrease the number of variations that are non-beneficial to the solution.

Another improvement that could be made to the variation operator is to determine, for each box-type, the frequency with which a box is placed in a certain area. This information can be used to further improve the variation operator by giving higher probabilities to areas a box is more frequently placed in.

User-Solver Interaction. A good area for further research is the user-solver interplay, where a system needs to be implemented that is able to give feedback to the user, and implements a method for the user to directly influence the solver. A description of factors that are of influence on such a system is given in Section 4.2.

Evaluation of Representation. Most of the run-time of the automated solver is spend on converting the solution representation to a solution that can be evaluated by the penalty function. An improvement to the solver can be made by defining a derivative of the penalty function that can by used to (partially) evaluate the representation, without the need for a conversion step.

Different Representation. The current indirect representation has one major drawback: The automated solver has no possibility to make a deliberate choice to leave a box out. This means that the building function will always try to place a box, even when it is better to leave a certain box out.

Sub-Penalty Weighting Even though results seem to indicate that the correction for the sub-penalty weight has a positive effect on how solutions are evaluated, investigation of solutions generated with these corrected sub-penalty weights should determine their efficacy.

Bibliography

[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] T. Bäck. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms.* Oxford University Press, Oxford, UK, 1996.

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

[4] 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.

[5] DENC Netherlands. Load-Optimizer Website. `http://www.load-optimizer.com/` `http://www.denc.com/`.

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

[7] A. Eiben, J. van der Hauw, and J. van Hemert. Graph coloring with adaptive evolutionary algorithms. *Journal of Heuristics*, 4(1):25–46, 1998.

[8] A. E. Eiben, J. van Hemert, E. Marchiori, and A. Steenbeek. Solving binary constraint satisfaction problems using evolutionary algorithms with an adaptive fitness function. In *Proceedings of the 5th international conference on Parallel Problem Solving from Nature (PPSN V)*, volume 1498, pages 196–205. Springer, 1998.

[9] M. Eley. Solving container loading problems by block arrangement. *European Journal of Operational Research*, 141(2):393–409, 2002.

[10] 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.

[11] 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.

[12] 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.

[13] M. Juraitis, T. Stonys, A. Starinskas, D. Jankauskas, and D. Rubliauskas. Randomized Heuristic for the Container Loading Problem: Further Investigations. *Information Technology and Control*, 35(1):7–12, 2006.

[14] K. Karabulut and M. M. İnceoğlu. A Hybrid Genetic Algorithm for Packing in 3D with Deepest Bottom Left with Fill Method. In *Proceedings of the Third International Conference on Advances in Information Systems*, ADVIS'04, pages 441–450, Berlin, Heidelberg, 2004. Springer-Verlag.

[15] 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 *Genetic and Evolutionary Computation Conference 2011*, pages 893–900. ACM, 2011.

[16] J. Leuven, E. Reehuis, M. Emmerich, and T. Bäck. User-Derived Mutation in Highly Constrained Truck Loading Optimization. In *2015 IEEE Congress on Evolutionary Computation, CEC 2015*, pages 235–242. IEEE, 2015.

[17] A. Lim, B. Rodrigues, and Y. Yang. 3-D Container Packing Heuristics. *Applied Intelligence*, 22(2):125–134, 2005.

[18] A. Lodi, S. Martello, and D. Vigo. Heuristic Algorithms for the Three-Dimensional Bin Packing Problem. *European Journal of Operational Research*, 141(2):441–450, 2002.

[19] 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.

[20] Python programming language. Official Python Website. https://www.python.org/.

[21] S. van Rijn, E. Reehuis, M. Emmerich, and T. Bäck. Optimizing Highly Constrained Truck Loadings Using a Self-Adaptive Genetic Algorithm. In *2015 IEEE Congress on Evolutionary Computation, CEC 2015*, pages 227–234. IEEE, 2015.

[22] 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.

[23] 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.

[24] W. Zhu, W.-C. Oon, A. Lim, and Y. Weng. The Six Elements to Block-Building Approaches for the Single Container Loading Problem. *Applied Intelligence*, 37(3):431–445, 2012.

Appendix A

Data Structure

The boxes in the problem instances under consideration are defined by a lot of parameters, listed in Table A.1.

TABLE A.1: *Parameters in box-array*

Variable Name	Description
Client ID	An id given to client to represent the order of servicing
Width	Width Of The Box
Height	Height of the Box
Length	Length Of The Box
Weight	Weight Of The Box
Pallet Configuration Hash	A hashed version of the pallet configuration variable (for efficient storage)
Product Group Hash	A hashed version of the product group variable (for efficient storage)
File Number	Unique identifier for problem instance the box is from
List ID	Unique identifier for box within problem instance
Current ID	Index of current row (to count the total number of boxes) even if part of data is lost
Below ID	Index of row in array that contains information of the box planned below
Calculated Volume	Length $*$ Width $*$ Height
Surface Area	Length $*$ Width

Parameters in box-array (Continued)

Variable Name	Description
Density	$\frac{\text{Weight}}{\text{Calculated Volume}}$
Pallet Configuration	Sequential indices for Pallet Configuration Hash
Product Group	Sequential indices for Product Group Hash
PC-PG Index	Sequential indices for each combination of Pallet Configuration and Product Group
Surface Area Index	Sequential indices to indicate Surface Area Bin (Section 4.1.2)
Weight Index	Sequential indices to indicate Weight Bin (Section 4.1.2)
Density Index	Sequential indices to indicate Density Bin (Section 4.1.2)
Above-Type Index	Sequential indices representing the above-box-type
Below-Type Index	Sequential indices representing the below-box-type
Not Bridge Or Small	Boolean variable that indicates if a box is not under the bridge or a ”small” box
Top Most	Boolean variable to store if current box is on top of its stack
Single Collo Stack	Boolean variable to store if current box is the only box in its stack (it is both the top and bottom box)

Columns 0 through 10 and 22 through 26 are actually saved, the other columns are recalculated on-the-fly. Column 22 through 26 are boolean parameters.