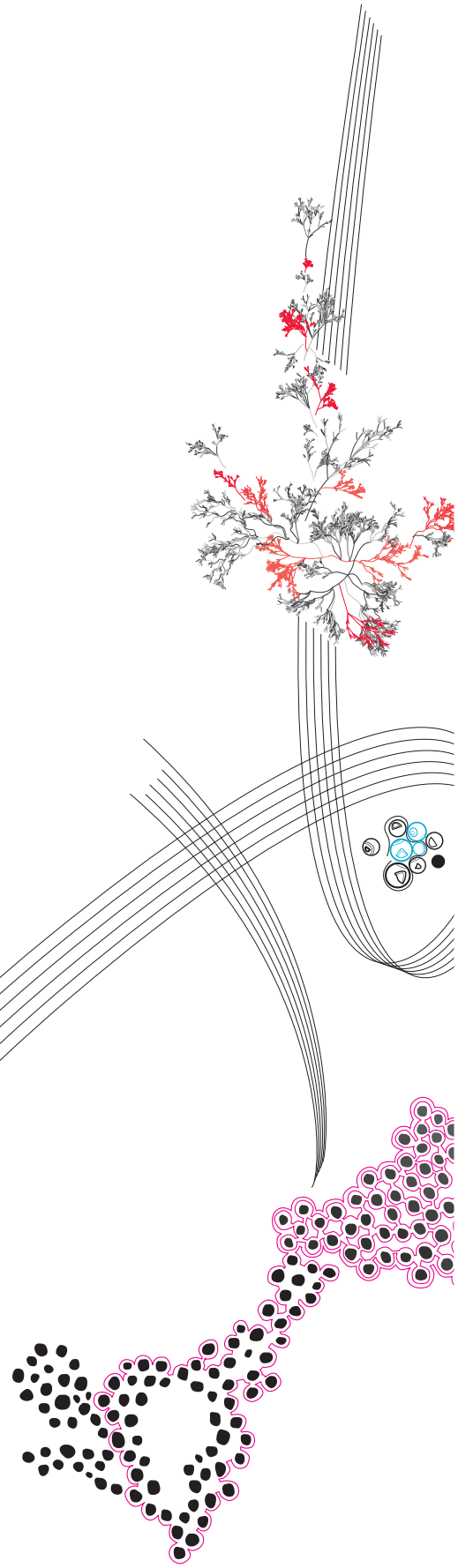


BSc Thesis Applied Mathematics

Solution quality analysis of a genetic algorithm on the carton set optimization problem using mixed-integer programming

Milo Wullink

Supervisor: dr. Matthias Walter

January, 2025

Department of Applied Mathematics
Faculty of Electrical Engineering,
Mathematics and Computer Science

UNIVERSITY OF TWENTE.

Preface

I would like to thank my supervisor dr. Matthias Walter for his supervision and guidance throughout my research. Moreover, I would like to thank him for his useful feedback.

Solution quality analysis of a genetic algorithm on the carton set optimization problem using mixed-integer programming

Milo Wullink*

January, 2025

Abstract

The carton set optimization problem is about finding a variety of carton sizes that minimize the total shipping costs within a warehouse. In 2020, Singh and Ardjmand [10] solved this problem using a genetic algorithm. A genetic algorithm is however not guaranteed to provide an optimal solution. In this paper, the solution quality of the genetic algorithm suggested by Singh and Ardjmand is analyzed. We compare the best-found solutions of the genetic algorithm using an exact mixed-integer programming approach to find the approximation error of the genetic algorithm. The approximation error of the genetic algorithm increases as the number of orders increases, but for order sizes where the mixed-integer program is unable to solve the problem, the genetic algorithm is a great alternative.

Keywords: carton set optimization problem; genetic algorithm; mixed-integer programming

1 Introduction

With sustainability becoming more important and shipping volumes increasing, optimizing carton usage is a pressing concern in the packaging industry as carton usage highly influences shipping costs. One potential solution for warehouses to minimize shipping costs is optimizing the variety of carton sizes they use. By increasing the range of carton sizes, warehouses can generally provide better-fitting options for each order, which in turn means lower shipping costs. However, having too many options can overwhelm packers, making it harder to select the most suitable one. Balancing these factors has proven challenging.

1.1 Related work

The problem has been addressed in many papers but was first introduced as the packaging problem by Wilson in 1965 [11]. The paper formulates the problem as an integer programming problem and suggests a heuristic approach by rationally selecting carton sizes and eliminating them one by one until the desired number of carton sizes is achieved. Lee, Chew, Lee and Thio [7] tackled a very similar problem in 2015, but the objective of their paper was to minimize the total wasted space in the cartons instead of minimizing shipping costs. Gurumoorthy and Hinge [5] suggested a decision-tree-based clustering method to solve the problem that tries to minimize the costs. In 2020, Singh and Ardjmand [10] also

*Email: m.wullink-1@student.utwente.nl

addressed this problem, naming it the carton set optimization problem. In their paper, they propose a genetic algorithm (GA) that can find a set of cartons that reduces the total shipping costs of diverse analyzed warehouses. They used a sample of real-world data across multiple warehouses. Their research focuses on two parts. The first part of their paper is about optimally packing multiple items into a cuboid. The second part of their paper, which is the part that this paper will strictly be focused on, uses the dimensions of the found cuboids to optimally select cartons such that the total shipping costs of the orders are minimized. They proposed a genetic algorithm that uses an iterative approach to optimize the shipping costs. Keeping the number of carton sizes fixed, the algorithm achieved a 1.3–7.8% reduction in carton usage across the three different warehouses compared to the currently implemented carton sizes in the warehouse. Moreover, the algorithm achieved a 5.03-7.0% decrease in the total shipping costs compared to what is currently implemented at DHL [8].

1.2 Contribution

While the paper [10] by Singh and Ardjmand outlines why the packaging industry needs to look into optimizing the carton sizes that they use, the paper does not grasp how much optimization is possible. The genetic approach suggested is iterative and is not guaranteed to result in an optimal solution. Although the algorithm was able to achieve better results than the currently implemented carton types at the warehouse, it is still unclear how far from optimal the newly found set of carton types is. The primary focus of this paper is evaluating the solution quality of the genetic algorithm applied to the carton set optimization problem.

Research question: What is the solution quality of the genetic algorithm approach by [10] on the carton set optimization problem?

In this paper, the suggested genetic algorithm is implemented. Moreover, an exact method is implemented, namely a mixed-integer programming (MIP) approach to solve the problem. The issue with an exact approach, as has been touched upon by Singh and Ardjmand [10], is that this method is unsuitable for larger quantities of orders. However, using the exact approach on medium-sized instances, such that both methods can find solutions, one can say something about the solution quality of the genetic algorithm. In this paper, both methods are implemented and their results are compared on medium-sized instances, ensuring that the MIP method could provide a solution.

2 Problem description

The name used by Singh and Ardjmand for the problem at hand is the *carton set optimization problem*. As input, we are given the dimensions and weights of all orders to be packed. Namely, we are given a set K of all orders (l_k, w_k, h_k, x_k) where l_k, w_k, h_k represent the dimensions and x_k the weight of each order k. Furthermore, we are given the dimensions of available cartons to be selected for the carton set. Namely, we are given a set I of all available cartons (l_i, w_i, h_i) where l_i, w_i, h_i represent the dimensions of each order i. Finally, the required number of carton types n in the final set is also given. The problem is to find a subset $A \subseteq I$ of cartons, such that the total shipping costs for packing all orders are minimized, assuming that once the cartons are chosen, each order is packed using the

carton that minimizes the cost for it. Namely, we are looking for a set A containing n number of cartons selected from the set of available cartons.

To find such a set, using the given set of orders K and the set of cartons I, one can create a mapping $f : K \to \mathcal{P}(I)$ between the orders K and the power set of cartons I where each order k is mapped to all cartons i that fit the order. This mapping can be visualized as a bipartite graph with two disjoint, independent sets (K and I) of vertices. An edge is created between two vertices, an order k and a carton i, if and only if the carton i is larger in all dimensions than the order k, more formally defined in (1). The value associated with the edge $c_{k,i}$ corresponds to the cost to ship order i using carton k as defined in (2). Two new values are introduced here: the dimensional weight factor d and the price per kilogram P_{kg} for shipping an order. The dimensional factor [9] and price per kilogram vary per shipping company. Using this mapping we can easily refer back to it when we need to calculate the incurred shipping costs of a set of cartons. In Figure 1 we can see a small-sized example of how such a mapping would look.

$$f : k \mapsto f(k), \quad f(k) = \{(l_i, w_i, h_i) \in I : l_k \leq l_i,\ w_k \leq w_i,\ h_k \leq h_i\} \quad k \in K, f(k) \in I \tag{1}$$

$$c_{k,i} = P_{kg} \cdot \max\left(x_k, \frac{l_i \cdot w_i \cdot h_i}{\mathrm{d}}\right) \tag{2}$$

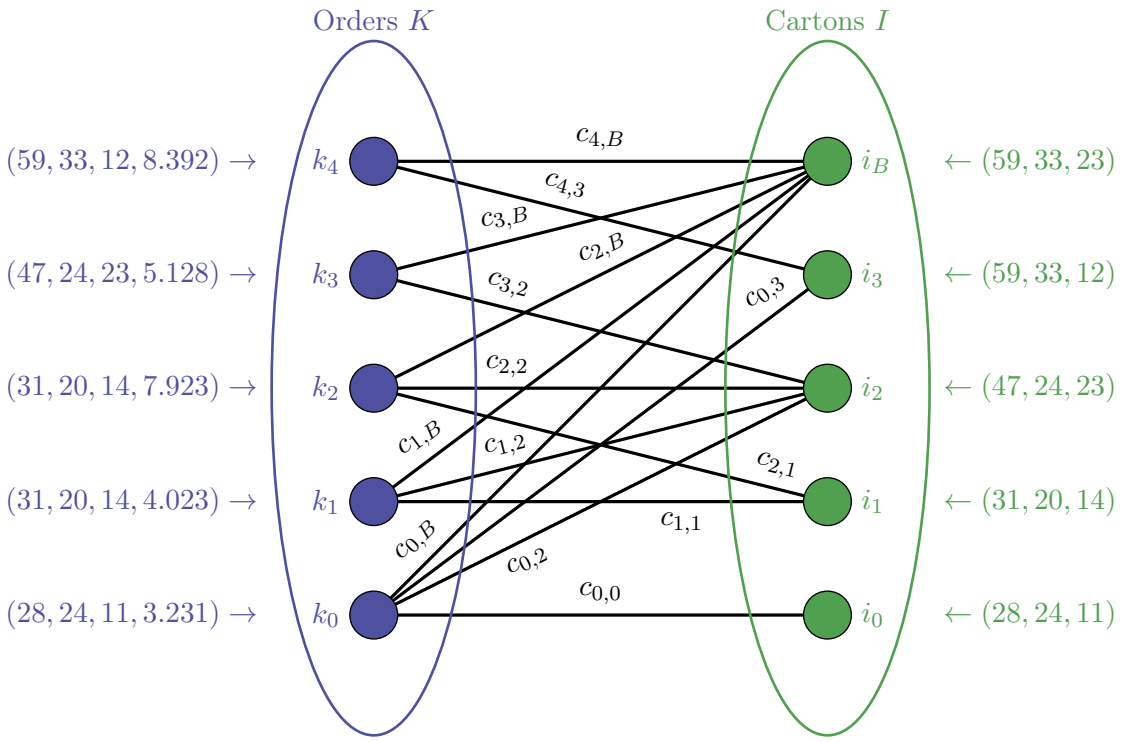


FIGURE 1: Bipartite graph that visualizes the mapping from orders K to cartons I. Vertices on the left correspond to orders with their corresponding dimensions and weight, and vertices on the right correspond to cartons and their corresponding dimensions. An edge $c_{k,i}$ indicates that order k can be packed using carton i. The value assigned to the edge $c_{k,i}$ corresponds to the cost of shipping order k with carton i.

3 Instance generation

In this paper, no real-world data is used as there was no such data publicly available. To compensate for this, a set of orders $K = \{(l_1, w_1, h_1, x_1), \ldots, (l_{|K|}, w_{|K|}, h_{|K|}, x_{|K|})\}$ is generated. More specifically, the dimensions are generated by assigning random variables using a probability distribution that tries to mimic real-world data. The value l_k is a random value that combines a Poisson distribution and a discrete uniform distribution given by (3). The values of w_k and h_k are random variables that depend on l_k, scaled by a normal distribution, as shown in (4) and (5) respectively. All three variables are rounded to the nearest integer and represent the dimensions of the order in centimetres. The dimensions of the order are then reordered in such a way that (6) holds. This corresponds to turning an order and is mainly important for the implementation of the algorithm. The weight variable x_k (kg) is determined as the maximum between a fixed lower bound (0.5 kg) and a random variable with a normal distribution that depends on the volume of the order in cm^3 as can be seen in (7). This ensures that the weight is proportional to the order volume as we would expect in real life.

$$l_k = 10 \times Pois(\lambda = 4) + \mathcal{U}\{0, 9\} \qquad l_k \in \mathbb{N} \tag{3}$$

$$w_k = \mathcal{N}(0.5, 0.1) \times l_k \qquad w_k \in \mathbb{N} \tag{4}$$

$$h_k = \mathcal{N}(0.5, 0.1) \times l_k \qquad h_k \in \mathbb{N} \tag{5}$$

$$l_k \geq w_k \geq h_k \tag{6}$$

$$x_k = \max(0.5, \mathcal{N}(\frac{l_k \cdot w_k \cdot h_k}{10,000}, \frac{l_k \cdot w_k \cdot h_k}{100,000})) \tag{7}$$

Together, these equations generated the dimensions and weight of each order k. For the experiments, the choice of cartons is restricted to the dimensions of all of the orders, including a carton whose dimensions are the maximum of all possible dimensions. Therefore, the set of cartons I use the same dimensions as the orders in set K, but without duplicates. The set of available cartons also includes the biggest carton i_B defined as:

$$i_B = (\max_{(l_i, w_i, h_i) \in I} l_i, \max_{(l_i, w_i, h_i) \in I} w_i, \max_{(l_i, w_i, h_i) \in I} h_i) \tag{8}$$

This ensures a feasible solution exists for any number of cartons to be selected. The running example presented in Figure 1 is a great illustration of this. Let the objective be to find the single best carton to ship all orders K. In other words, let $n = 1$. In this case, none of the cartons i_0 up until i_4 would suffice, as none of the cartons can fit all orders. In the case of the running example, $i_B = (59, 33, 23)$ and can fit all orders. Therefore, all edges between i_B and the cartons $k_0, \ldots, k_4$ exist in Figure 1.

4 Genetic Algorithm

4.1 Method

One way of solving the carton set optimization problem is using a *genetic algorithm* as shown in [10]. In general, a genetic algorithm is used for optimization problem sets where an exact solution is not feasible as computation times are unrealistic. A genetic algorithm uses a heuristic approach to evaluate the given problem and is therefore not guaranteed to provide an optimal solution. A genetic algorithm applies a trial-and-error approach using generations. Each generation consists of a population $\mathscr{A} = \{A_1, A_2, \ldots, A_R\}$ which is a set of feasible solutions, also called individuals. Each individual A represents a feasible solution to the problem.

In the context of the carton set optimization problem, this means that an individual A consists of n cartons that can pack all orders K. To ensure the feasibility of an individual, i_B as defined in (8) is always in the individual. Therefore, if $n = 1$ then $A = \{i_B\}$ as we had already established for the running example.

Each generation of the genetic algorithm consists of a new population. This population is generated using the previous population. To initialise the genetic algorithm, an initial population $\mathscr{A}_1$ is created using Algorithm 1. The initial population $\mathscr{A}_1$ consists of R unique individuals A that are generated independently. One of the parameters of a genetic algorithm is the value of R. An individual always includes the largest carton i_B. The other $n-1$ cartons are randomly selected from the set of available cartons I while avoiding duplicate selection of cartons.

Algorithm 1 Generate population (*generate_population*)

Input: size of population R, size of individual n, set of available cartons I
Output: population $\mathscr{A}$
$\mathscr{A} \leftarrow \emptyset$
while $|\mathscr{A}| < R$ **do**
    Let $i_B \in I$ be as in (8).
    $A \leftarrow \{i_B\}$
    while $|A| < n$ **do**
        Let y be chosen uniformly at random from $I \setminus \{i_B\}$.
        Append y to A.
    end while
    Append A to $\mathscr{A}$.
end while
Return: population $\mathscr{A}$

After an initial population $\mathscr{A}_1$ has been generated, the genetic algorithm uses the initial population to generate a new population of individuals. In the case of the carton set optimization problem, we are looking for a feasible solution that minimizes the total shipping costs for all orders. In more mathematical terms, we are looking for an individual A that minimizes the fitness function $\mathscr{F}$ as defined in (9).

$$\mathscr{F}(A) = \sum_{k \in K} c_{k,i_k^*} \qquad \text{where } i_k^* = \text{argmin}_{i \in A}\{c_{k,i}\} \quad \forall k \in K \tag{9}$$

Using Algorithm 2, we can evaluate individuals and sort them on this fitness value, where a lower value of $\mathscr{F}$ corresponds to a more desirable solution to the problem. The general idea behind a genetic algorithm is to evaluate the individuals within the population and use a subset of individuals with a low fitness value to generate a new population to hopefully obtain even lower fitness values.

Algorithm 2 Fitness of individual (*fitness_ind*)

Input: individual A, orders K
Output: fitness value $\mathscr{F}(A)$
$\mathscr{F}(A) \leftarrow 0$
for each order $k \in K$ **do**
    $i \leftarrow \text{argmin}_{i \in A}\{c_{k,i}\}$ ▷ Select carton from A with minimal cost
    $\mathscr{F}(A) \leftarrow \mathscr{F}(A) + c_{k,i}$
end for
Return: $\mathscr{F}(A)$

The reproduction and mutation algorithms are what make genetic algorithms distinct from each other. These algorithms vary immensely and can be customised to fit the type of individual. In order to analyze the state-of-the-art, we have chosen to replicate the algorithm suggested in [10]. In Algorithm 3, you can see that the reproduction process involves keeping the best half of the individuals. All other individuals are discarded as their fitness value $\mathscr{F}$ was too high. The other half is kept for generating new individuals. More specifically, new individuals A' are generated by choosing random cartons from the best half of the individuals from population $\mathscr{A}$. The new individual A' contains the largest carton i_B. This process repeats until the new population $\mathscr{A}'$ contains R individuals.

Algorithm 3 Reproduce Population (*reproduce*)

Input: population $\mathscr{A}$, set of available cartons I, size of individual n, fitness value $\mathscr{F}$
Output: reproduced population $\mathscr{A}'$
Initialize old population $\mathscr{A}^{old} \leftarrow \emptyset$.
Append half of the individuals with the lowest fitness value $\mathscr{F}$ from $\mathscr{A}$ to $\mathscr{A}^{old}$.
Initialize new population $\mathscr{A}^{new} \leftarrow \emptyset$.
while $|\mathscr{A}^{new}| < \frac{1}{2}|\mathscr{A}|$ **do**
    Initialize new individual $A' \leftarrow \emptyset$.
    Append i_B as in (8) to A'.
    while $|A'| < n$ **do**
        Let A be chosen uniformly at random from $\mathscr{A}^{old}$.
        Let i_q be chosen uniformly at random from $A/\{i_B\}$.
        Append i_q to A'.
    end while
    Add A' to $\mathscr{A}^{new}$.
end while
Let the reproduced population be $\mathscr{A}' = \mathscr{A}^{old} \cup \mathscr{A}^{new}$.
Return: $\mathscr{A}'$

As one can imagine, this method of reproducing population might shift into a bias towards specific cartons, as the more individuals that include this carton there are, the more chance this carton has of becoming one of the selected cartons for the new individual. To prevent biases, we also mutate some individuals.

Mutation takes place after reproduction of the population. Each individual A within the population $\mathscr{A}$ has a probability p_m of getting mutated as can be seen in Algorithm 5 This mutation probability is another parameter of the genetic algorithm. If an individual is mutated, a carton i_y is randomly selected to be removed, and a carton i_z is randomly selected to replace carton i_y. During the selection process, carton i_y is chosen in such a way that $i_y \neq i_B$, as carton i_B guarantees the feasibility of an individual. This mutation process is shown in Algorithm 4 and 5.

Algorithm 4 Mutate individual (*mutate_individual*)

Input: individual A, set of available cartons I
Output: mutated individual A
Let $i_B \in I$ be as in (8).
Let i_y be chosen uniformly at random from $A \setminus \{i_B\}$.
Let i_z be chosen uniformly at random from $I \setminus A$.
Remove i_y from A.
Append i_z to A.
Return: individual A

Algorithm 5 Mutation (*mutate_population*)

Input: population $\mathscr{A}$, mutation probability p_m
Output: mutated population $\mathscr{A}'$
$\mathscr{A}' \leftarrow \emptyset$
for each individual A in $\mathscr{A}$ **do**
    Mutate A using Algorithm 4 with probability p_m.
    Append A to $\mathscr{A}'$.
end for
Return: $\mathscr{A}'$

All of these algorithms form the genetic algorithm for the carton set optimization problem, (largely) unchanged from the genetic algorithm that is suggested in [10]. The general steps that the genetic algorithm takes are; generating an initial population $\mathscr{A}_1$ and repeatedly reproducing and mutating the current population. Let $\mathscr{A}_L$ be the last population of the genetic algorithm. At the end the GA reports the best-found individual $\operatorname{argmin}_{A \in \mathscr{A}_L} \mathscr{F}(A)$ and its fitness value $\min_{A \in \mathscr{A}_L} \mathscr{F}(A)$. When the genetic algorithm should stop running is a parameter of the genetic algorithm. This can be defined in a unit of time or the number of generations that the genetic algorithm should complete.

Algorithm 6 Genetic Algorithm (*GA*)

Input: number of generations g / maximum time T
Output: individual A and corresponding, lowest found fitness value $\mathscr{F}(A)$
Generate initial population $\mathscr{A}_1$ using Algorithm 1.
repeat g **times or until** T **time has passed**
    $\mathscr{A} \leftarrow$ Reproduce population $\mathscr{A}$ using Algorithm 3.
    $\mathscr{A} \leftarrow$ Mutate population $\mathscr{A}$ using Algorithm 5.
end
Return: $\min_{A \in \mathscr{A}} \mathscr{F}(A)$; $\operatorname{argmin}_{A \in \mathscr{A}} \mathscr{F}(A)$

4.2 Implementation

Besides the formal mathematical side of the genetic algorithm, there is also a lot of programming involved. The programming of the model is quite important for the efficiency of the algorithm, as a bad implementation can lead to a slow algorithm. To run the program as fast as possible some techniques are implemented that speed up the program. The genetic algorithm is implemented within Python [3], as this was also done by Singh and Ardjmand [10]. The complete code is uploaded to GitLab [12].

The first technique is a dictionary `costs` for storing the values of $c_{k,i}$. Although the function for $c_{k,i}$ (2) is not very complex, for large instances of set K and set I, the number of edges $c_{k,i}$ will increase quadratically in $|K|$ under the assumption that K does not contain too many duplicates. Each time an individual has to be evaluated, the values of $c_{k,i}$ are needed. To save the time of recalculating each value $c_{k,i}$ every time, a dictionary is created within Python that saves these values, such that the values can be accessed when needed. The dictionary `costs` is structured as follows; the key is the order k that needs to be packed, and the value assigned to that key is a list of all possible cartons i that fit the order k, together with their corresponding values $c_{k,i}$. Using this method, all values $c_{k,i}$ need to be calculated only once. While you lose a bit of time looking up the needed value $c_{k,i}$ in the dictionary `costs`, this does not weigh up to the time it takes to recalculate the value.

To calculate the fitness of an individual A, all orders k need to be packed using a carton i. As stated previously, we assume that once the cartons A are chosen, each order is packed using the carton that minimizes the cost for it. As can be seen in Algorithm 2, the selection of the best-fitting carton is done using $i \leftarrow \text{argmin}_{i \in A}\{c_{k,i}\}$. However, this operation is computationally expensive. To lower the computational costs, the dictionary `costs` is sorted. That is, the value associated with a key is not only a list of all possible cartons i that fit the order k but they are also sorted based on their corresponding values of $c_{k,i}$. When looking at the running example, the dictionary `costs` would look as shown in Figure 2.

$$
\begin{aligned}
&\{\\
&k_0: \quad [\,(\,i_0\,, c_{0,0} = 3.231\,)\,, (\,i_3\,, c_{0,3} = 4.672\,)\,, (\,i_2\,, c_{0,2} = 5.189\,)\,, (\,i_B\,, c_{0,B} = 8.956\,)\,]\,;\\
&k_1: \quad [\,(\,i_1\,, c_{1,1} = 4.023\,)\,, (\,i_2\,, c_{1,2} = 5.189\,)\,, (\,i_B\,, c_{1,B} = 8.956\,)\,]\,;\\
&k_2: \quad [\,(\,i_1\,, c_{2,1} = 7.923\,)\,, (\,i_2\,, c_{2,2} = 7.923\,)\,, (\,i_B\,, c_{2,B} = 8.956\,)\,]\,;\\
&k_3: \quad [\,(\,i_2\,, c_{3,2} = 5.189\,)\,, (\,i_B\,, c_{3,B} = 8.956\,)\,]\,;\\
&k_4: \quad [\,(\,i_3\,, c_{4,3} = 8.392\,)\,, (\,i_B\,, c_{4,B} = 8.956\,)\,]\,;\\
&\}
\end{aligned}
$$

FIGURE 2: A visual representation of the dictionary `costs` of $c_{k,i}$ for the running example of Figure 1. The values of $c_{k,i}$ have been calculated using (2) where $d = 5000$ and $P_{kg} = 1$. Note that the dictionary is sorted based on the values of $c_{k,i}$.

The reason for sorting the dictionary in this manner is that we do not need to loop over the whole list of possible cartons i that fit order k. Instead, we move through the list until we have found a carton i that is in the individual A of which we are calculating the fitness value and take the corresponding value of $c_{k,i}$. In the worst case, we need to loop over the whole list and only the last carton in the list is in individual A, but more often we will find a fitting carton that is in the individual much earlier on. The computational time

of sorting the dictionary `costs` once is much lower than that for looping over all lists every time you evaluate an individual. The first carton i that you find that is in the individual A is guaranteed to be the carton that minimizes the cost of packing order k.

The final technique that is applied is similar to the first technique. Again a dictionary is created that stores values such that they do not have to be recalculated. This time the dictionary is created for storing the calculated values of $\mathscr{F}(A)$. The key of the dictionary is individual A and the value associated with the key is the fitness value of that individual $\mathscr{F}(A)$. The best half of the individuals stay in the next population. Recalculating their fitness values in the next generation would not make any sense. Therefore, we store the values of individuals that have been evaluated in a dictionary. If the individual is in the dictionary, we can access the value. Otherwise, we calculate it and add it to the dictionary.

5 Mixed-Integer Programming

5.1 Method

The second approach that is implemented to solve this problem is *mixed-integer programming*. MIP is a mathematical approach for solving optimization problems. These optimization problems are generally given as in (10).

$$\begin{aligned} \min_{\mathbf{x}} \quad & \mathbf{c}^\top \mathbf{x} \\ \text{s.t.} \quad & \mathbf{A}\mathbf{x} \geq \mathbf{b} \\ & \mathbf{l} \leq \mathbf{x} \leq \mathbf{u} \\ & \mathbf{x} \in \mathbb{R}^n \end{aligned} \tag{10}$$

The matrix $\mathbf{A}$ has a size of $m \times n$. The vector $\mathbf{b}$ has a length of m and $\mathbf{l}, \mathbf{u}$ and $\mathbf{c}$ are vectors of length n. The optimization problem is to find a vector $\mathbf{x}$ of length n that minimizes (or maximizes) the objective value $\mathbf{c}^\top \mathbf{x}$ and satisfies the constraints $\mathbf{A}\mathbf{x} \geq \mathbf{b}$ and $\mathbf{l} \leq \mathbf{x} \leq \mathbf{u}$. Mixed-integer programming has the advantage of being able to find an optimal solution to the problem, contrary to the genetic algorithm, which is not guaranteed to find an optimal solution. However, the trade-off for this is that the instance sizes it can handle are very limited as the computational time for solving a problem becomes unrealistic.

The carton set optimization problem can also be formulated as a mixed-integer programming problem. The objective of the problem is to minimise the total shipping costs for all packages. We can formulate this objective as a function by introducing new variables. First, we introduce a binary variable y_i for each carton i. The value of y_i is 1 if the carton is selected in the individual and 0 otherwise. Moreover, we introduce a binary variable $x_{k,i}$ for each pair of k and i. The value of $x_{k,i}$ is 1 if carton i is used for packing order k, otherwise the value of $x_{k,i}$ is 0. Now that we have defined our variables, we can formulate the objective function as the product of $c_{k,i}$ and $x_{k,i}$ for all k and i. The carton set optimization problem is to find an individual that minimizes this objective value as denoted in (11). There are of course also some constraints that should be added to this model, such that the model finds a feasible solution. The first constraint is that the number of selected cartons in the individual should equal n, as this is a given problem constraint. Secondly, to make sure that all orders get packaged (exactly once), we add the constraint that the sum of all $x_{k,i}$ for all $i \in I$ should equal one. No package should be packed twice or not packed,

but it should be packed exactly once. Finally, the constraint that an order k can only be packaged using carton i, if that carton is selected within the individual is added. Therefore $x_{k,i}$ should be smaller or equal to y_i for all k and i. These variables, constraints and the objective together, outline the problem as the mixed-integer programming problem shown in (11) which can be solved using MIP.

$$
\begin{aligned}
\min \quad & \sum_{i \in I} \sum_{k \in K} c_{k,i} \cdot x_{k,i} \\
\text{s.t.} \quad & \sum_{i \in I} y_i = n \\
& \sum_{i \in I} x_{k,i} = 1 && \forall k \in K \\
& x_{k,i} \leq y_i && \forall k \in K, \forall i \in I \\
& x_{k,i} \in \{0,1\} && \forall k \in K, \forall i \in I \\
& y_i \in \{0,1\} && \forall i \in I
\end{aligned}
\tag{11}
$$

After having formulated our problem as a MIP (11), one needs to solve it. There exist several implementations of the state-of-the-art solution methods. One of these methods is the branch and bound algorithm which was developed by [6].

Branch and bound is a search tree algorithm that breaks down the problem into sub-problems that are easier to solve. To explain branch and bound, consider the linear problem given by the equation and inequalities of (12). The problem involves maximizing a function with certain constraints and variables, just like the carton set optimization problem.

$$
\begin{aligned}
\max_{x_1, x_2} \quad & y = 8x_1 + 5x_2 \\
\text{s.t.} \quad & x_1 + 7x_2 \leq 28 \\
& 7x_1 + 3x_2 \leq 35 \\
& x_1 \geq 0, x_2 \geq 0 \\
& x_1 \in \mathbb{Z}, x_2 \in \mathbb{Z}
\end{aligned}
\tag{12}
$$

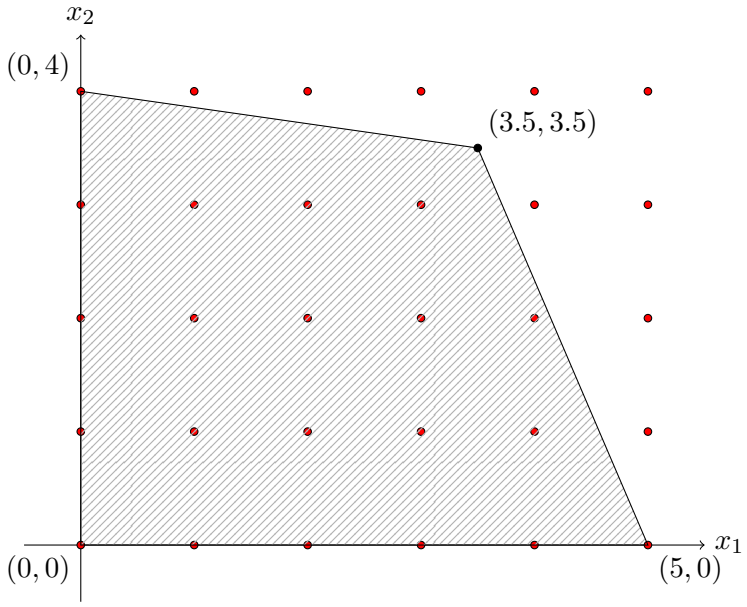


FIGURE 3: The visualised feasible region for the example linear problem defined by (12).

The constraints given by (12) form the feasible region as illustrated by the shaded area in Figure 3. The coordinates of the vertices of the shaded polygon are $(0,0)$, $(0,4)$, $(3.5,3.5)$ and $(5,0)$, with objective values 0, 32, 45.5 and 40 respectively. The linear problem is now solved with the relaxation that variables x_1 and x_2 are real numbers instead of integers. For this example, an optimal solution is found at $(3.5,3.5)$. Note, however, that $(3.5,3.5)$ is not a feasible solution for the linear program defined by (12) as both x_1 and x_2 are not integers. This is where branching is needed. We can subdivide the problem by looking at x_1 first. For the problem to be feasible, x_1 should be an integer. Therefore, $x_1 \leq 3$ or $x_1 \geq 4$ must hold which creates the two different subproblems that are given by the equation and constraints (13) and (14) respectively.

$$\begin{aligned} \max_{x_1,x_2} \quad & y = 8x_1 + 5x_2 \\ \text{s.t.} \quad & x_1 + 7x_2 \leq 28 \\ & 7x_1 + 3x_2 \leq 35 \\ & x_1 \leq 3, x_2 \geq 0 \\ & x_1 \in \mathbb{Z}, x_2 \in \mathbb{Z} \end{aligned} \tag{13}$$

$$\begin{aligned} \max_{x_1,x_2} \quad & y = 8x_1 + 5x_2 \\ \text{s.t.} \quad & x_1 + 7x_2 \leq 28 \\ & 7x_1 + 3x_2 \leq 35 \\ & x_1 \geq 4, x_2 \geq 0 \\ & x_1 \in \mathbb{Z}, x_2 \in \mathbb{Z} \end{aligned} \tag{14}$$

These subproblems can be solved accordingly and provide new coordinates for the best-found objective within those bounds. For $x_1 \leq 3$ this is $(3, 3.57)$ with the optimal value of $y = 41.86$ and for $x_1 \geq 4$ these values are $(4, 2.33)$ and $y = 43.67$ respectively. This step is then repeated for x_2, as x_2 is still not an integer value, which results in even more subproblems. When this is finished we are left with the binary tree that can be seen in Figure 4. Interesting to note, is that the combinations of the newly added constraints $x_1 \geq 4$ and $x_2 \geq 3$ do not leave any feasible region and therefore no solution is found and we set $y = \infty$. In the end, the best-found integer solution to the problem is $(4, 2)$ with objective value $y = 42$.

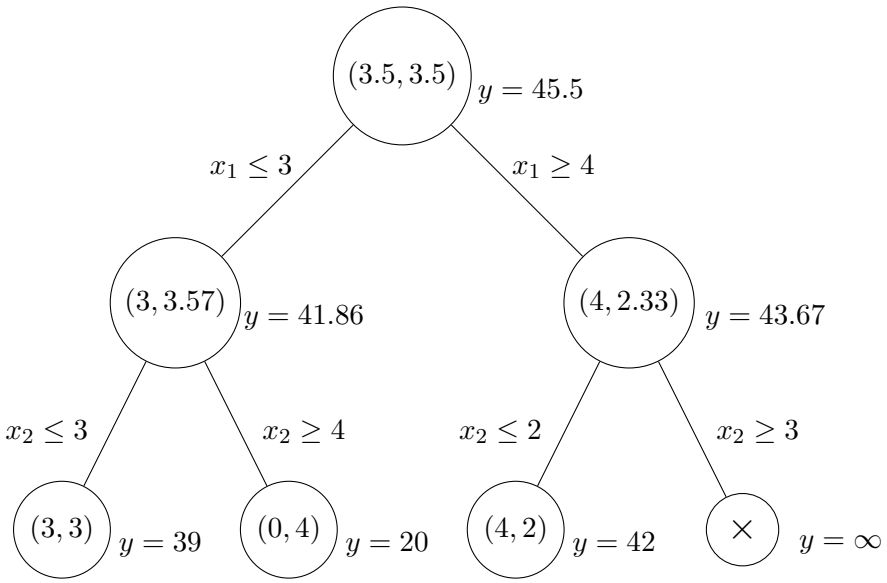


FIGURE 4: Visualisation of the branch and bound algorithm on the example linear problem defined by (12).

This is the general idea behind the branch and bound algorithm. Of course, this problem was easy enough to calculate all feasible integer solutions by hand, but for the carton set optimization problem, the number of variables is $|K| \cdot (|I| + 1)$ which quadratically increases in $|K|$. In the worst-case scenario, the branch and bound algorithm will only terminate after having inspected all of the finite subsets A of cartons I. Because the feasible region is finite, the branch and bound algorithm is guaranteed to find an optimal solution. However, the potential of the algorithm stems from the hope to be able to discard many of the subproblems as their objective value is too high and can therefore be completely discarded. Next to branch and bound mixed-integer programming also utilizes techniques like presolve, cutting planes, heuristics and parallelism [2].

5.2 Implementation

To solve the mixed-integer programming problem in Python, the model is implemented using the MIP solver Gurobi [1]. Gurobi takes the variables x_{ji} and y_i, the objective function and constraints mentioned in (11) as inputs. Gurobi finds an optimal solution to the problem, given that the instance size can be handled by Gurobi.

6 Computational results

Before we can sufficiently compare the genetic algorithm with the mixed-integer program, we need to choose the right parameters for the genetic algorithm. The genetic algorithm has two parameters that can largely influence the efficiency of the algorithm. The first parameter is the population size R, which is the number of individuals evaluated in each generation. The second variable is the mutation probability p_m, which denotes the probability of an individual to be mutated in the mutation phase of a generation. To find what value to use for each parameter preliminary experiments are run in which the approximation error of the genetic algorithm is compared against the mutation probability p_m and the population size R, respectively. To effectively compare the results, mixed-integer programming is used to compute the exact solution. The preliminary experiments are run on the problem of selecting the $n = 20$ cartons that minimize the shipping costs of $|K| = 500$ orders. These parameters enable the mixed-integer program to confidently solve the problem within 10 minutes. Moreover, the dimensional factor d is set to 5000 and the price per kilogram P_{kg} to €1.29 for all experiments, unless stated otherwise. The average approximation error is taken over 5 different order sets to compensate for outliers. The maximum running time for the genetic algorithm is set to 2 minutes.

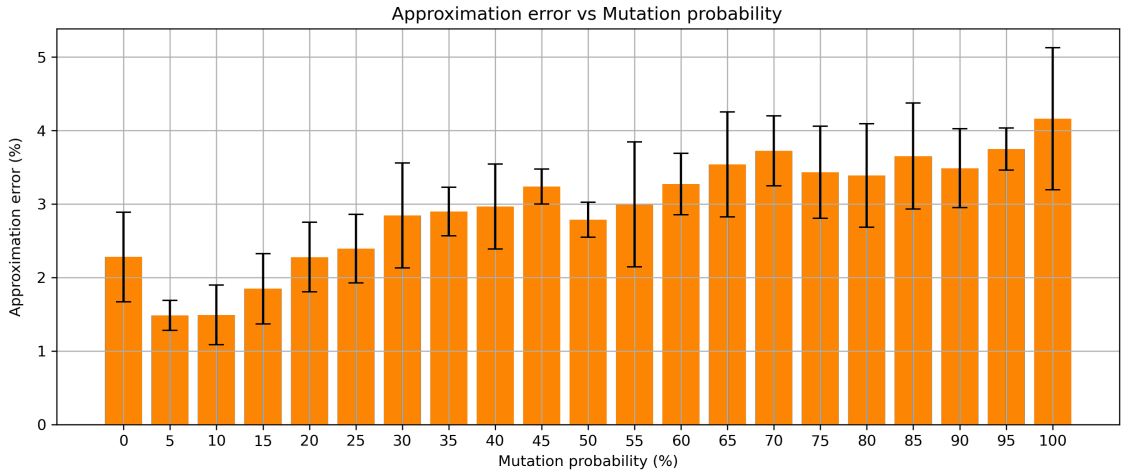


FIGURE 5: The average approximation error of the genetic algorithm over 5 instances for different mutation probabilities. The approximation error represents the percentual difference to the optimal solution that is calculated on the same instance using mixed-integer programming.

The experiment is done on all mutation probabilities in increments of 5%. The results in Figure 5 show that for higher values of the mutation probability p_m the approximation error gets larger. This means that the objective value found by the genetic algorithm is further away from the optimal solution found by the mixed-integer programming. In Figure 5 we see that the best value for the mutation probability lies somewhere around 5% or 10% as their corresponding approximation errors are the lowest.

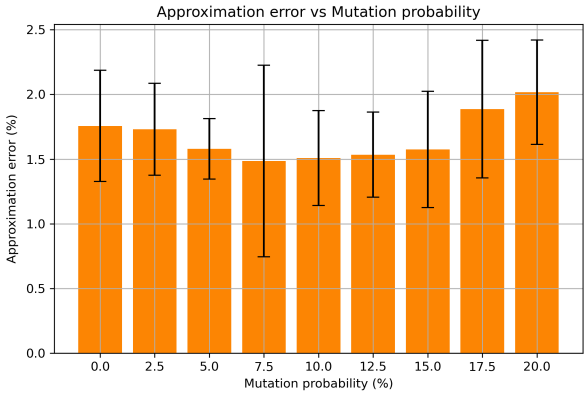


FIGURE 6: The average approximation error of the genetic algorithm over 5 instances for different mutation probabilities p_m between 0% and 20%. The approximation error represents the percentual difference to the optimal solution that is calculated on the same instance using mixed-integer programming.

To pick the best value for mutation probability p_m the experiment is rerun for mutation probabilities between 0% and 20% with smaller increments of 2.5%. Using the results of Figure 6, the mutation probability p_m of the genetic algorithm is set to 10%. This value is chosen as it has a low approximation error on average and the variance is low compared to a mutation probability of 7.5%.

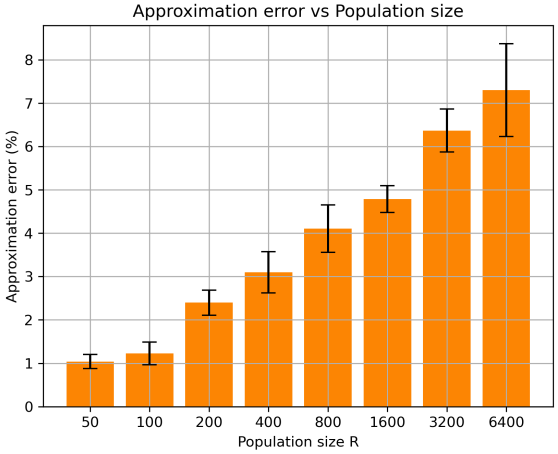


FIGURE 7: The average approximation error of the genetic algorithm over 5 instances for different population sizes R. The approximation error represents the percentual difference to the optimal solution that is calculated on the same instance using mixed-integer programming.

When running the experiment of the approximation error compared to population size we see an initially unexpected result. A smaller population size seems to correspond to a lower approximation error, where $R = 50$ gives the best result. This can be explained by the fact that each population is sorted to reproduce and mutate the population. A larger population size increases the complexity of sorting and has a longer computation time. The trend line in Figure 7 suggests that a population size smaller than $R = 50$ might be even better.

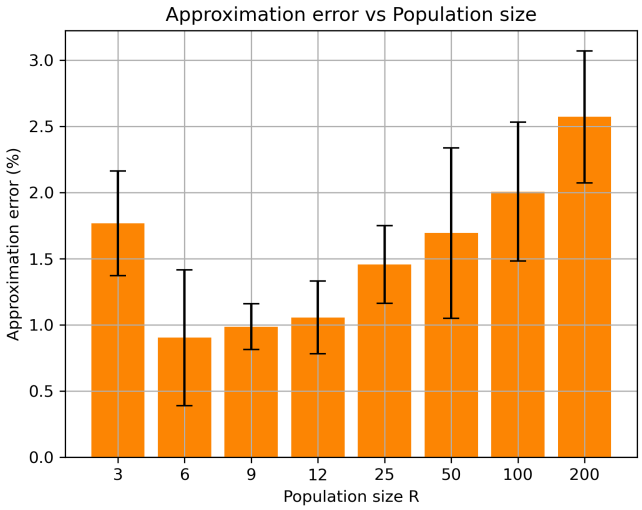


FIGURE 8: The average approximation error of the genetic algorithm over 5 instances for smaller population sizes R. The approximation error represents the percentual difference to the optimal solution that is calculated on the same instance using mixed-integer programming.

To confirm if this is the case, an additional experiment is run on smaller population sizes. The results of Figure 8 show that indeed smaller population sizes correspond to a smaller approximation error, while $R = 3$ may not be able to provide enough variance in the population. Based on the results of this experiment, the population size R of the genetic algorithm is set to $R = 6$, as it provided the lowest approximation error in these preliminary experiments.

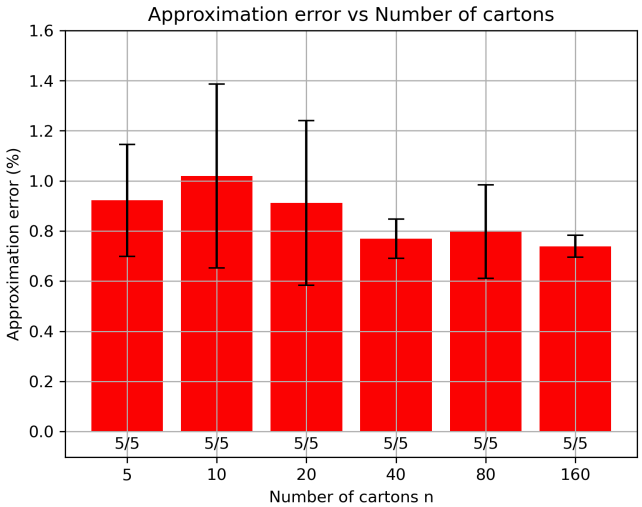


FIGURE 9: The average approximation error over 5 instances of the genetic algorithm for different numbers of cartons n. The fraction below each bar represents the number of times the mixed-integer program was able to find an optimal solution within the 10-minute time limit.

Now that the parameters of the genetic algorithm are set, the experiments that evaluate the solution quality of the genetic algorithm were initialized. Two main experiments were run. First, an experiment was run on the number of cartons selected. Just like in the preliminary tests, the number of orders $|K|$ is set to 500, but the number of unique cartons n that pack these orders is the variable in this experiment. In Figure 9, the approximation error is plotted against the number of cartons. Although the number of cartons decreases the total shipping costs as can be seen in Table 1, it seems like there is no strong relation between the variable n and the actual approximation error of the genetic algorithm. In other words, although more carton sizes decrease the costs, the solution quality of the genetic algorithm does seem to be affected.

TABLE 1: The average cost of shipping 500 orders over 5 instances using a fixed cost of P_{kg} =€1.29/kg. The maximum is taken between the actual weight and the dimensional weight of the carton to determine the weight.

n	Total price	Price per order
5	€9780.61	€19.56
10	€7101.86	€14.20
20	€6555.32	€13.11
40	€5739.50	€11.48
80	€5230.34	€10.46
160	€5043.65	€10.09

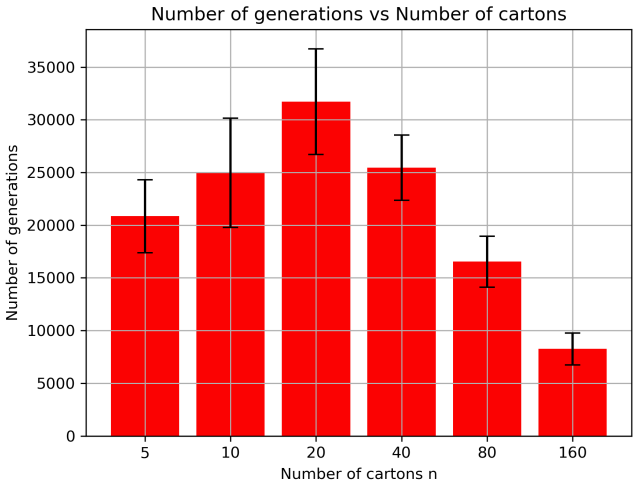


FIGURE 10: The average number of generations that the genetic algorithm was able to run within the time limit over 5 instances for different numbers of cartons n.

In Figure 10, the number of generations is plotted against the number of cartons. The genetic algorithm was able to run the most number of generations for $n = 20$ which is quite odd. To clarify why this is the case, an additional experiment should be run on how much of the running time is spent on which process of the genetic algorithm. This experiment could give more insight into the time spent on reproducing the population, mutating the population and sorting the population, which might explain the behaviour that is seen in Figure 10.

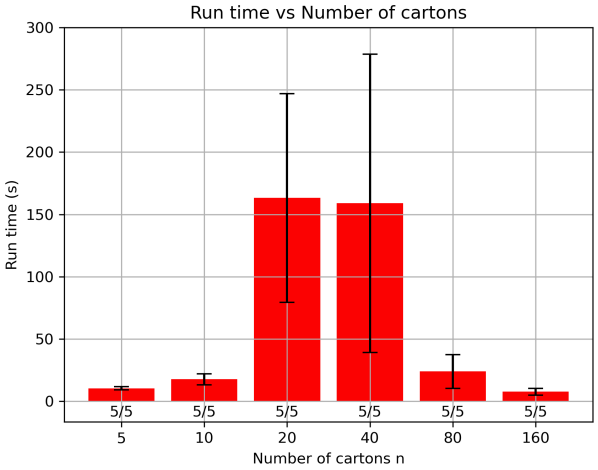


FIGURE 11: The average run time of the mixed-integer program over 5 instances for different numbers of cartons n. The fraction below each bar represents the number of times the mixed-integer program was able to find an optimal solution within the 10-minute time limit.

In Figure 11, the run time of the mixed-integer program is plotted against the number of cartons. What is notable about this plot is that the run times for $n = 20$ and $n = 40$ are significantly higher. This behaviour can be explained by the number of possible combinations there are of picking a subset A of size n from the set of available cartons I. However, it should be noted that the sudden decrease in runtime for $n = 80$ and $n = 160$ can not be explained using the same logic as the possible number of combinations is even larger. An explanation for this might be that an excessive number of carton sizes for 500 orders makes it easier for the mixed-integer program to find a near-optimal solution as any selection of a large number of carton sizes can pack the orders effectively. And using a near-optimal solution the mixed-integer program can discard many subproblems. This is confirmed when looking at the approximation error over time graph for $n = 160$ in Figure 33d. It shows that the genetic algorithm started with a solution that had an approximation error of around 7% for this instance, while the initial approximation error was higher for all other values of n as can be seen in Figure 28d, 29d, 30d, 31d and 32d.

The limitation and reason why the carton set optimization problem is not solved exclusively using a mixed-integer program is because of the computational time. If the instance size becomes too big, the mixed-integer program is unable to solve the problem within the time limit. This is nicely illustrated in Figure 13, where you can see that the run time increases as the number of orders increases. It should be noted that the mixed-integer program was only able to solve 3 out of 5 problems for an instance size of 750 cartons within the 10-minute time limit, and none of the larger instances. For the instance of 750, the average time was calculated only using the runs that finished before the time limit, not including the two runs of more than 10 minutes. In Figure 12, the correlation between the approximation error of the genetic algorithm and the instance size of the problem is shown. There seems to be an increasing trend line within this plot, although it is not very clear. One reason for this might be that taking a 5-time average is not enough to compensate for outliers.

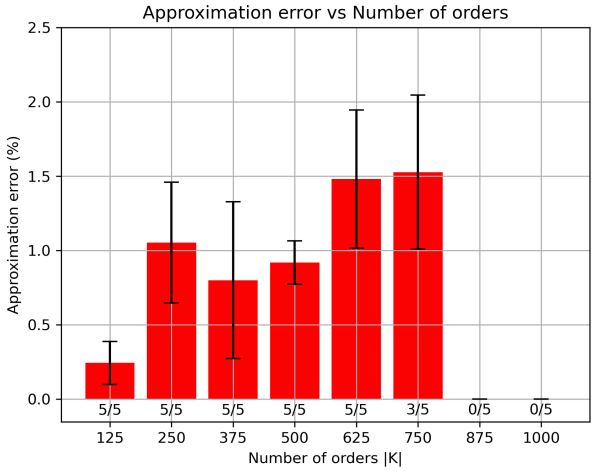


FIGURE 12: The average approximation error over 5 instances of the genetic algorithm for different numbers of orders $|K|$. The fraction below each bar represents the number of times the mixed-integer program was able to find an optimal solution within the 10-minute time limit.

FIGURE 13: The average run time of the mixed-integer program over 5 instances for different numbers of orders $|K|$. The fraction below each bar represents the number of times the mixed-integer program was able to find an optimal solution within the 10-minute time limit.

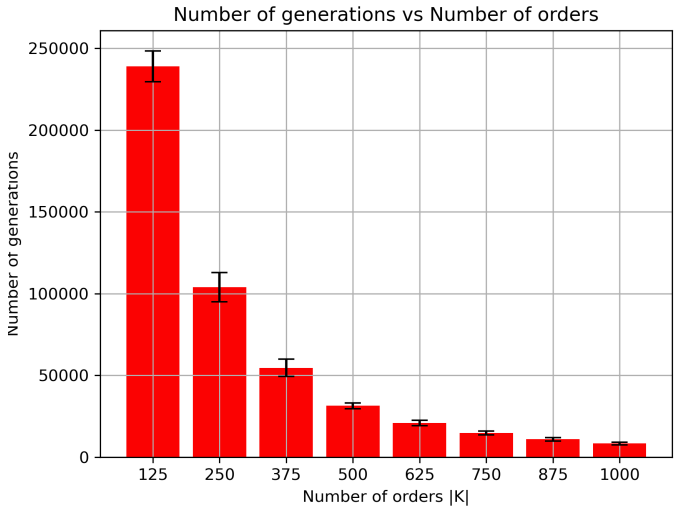


FIGURE 14: The average number of generations over 5 instances of the genetic algorithm for different numbers of orders $|K|$.

While running the main experiments, experiments were also run using another cost function that represents a real-world scenario more closely than having a factor P_{kg} that makes costs linear to the weight. To be more specific, the experiments were run on the step-wise cost system of UPS Standard delivery in the Netherlands [4]. The price of shipping a package increases at certain weight thresholds. All experiments using this new step-wise cost function are coloured blue. The principle of the problem stays the same, but the prices do differ from the linear cost function, as can be seen in Table 2. The dimensional factor d is kept the same, as this is also the dimensional factor that UPS wields.

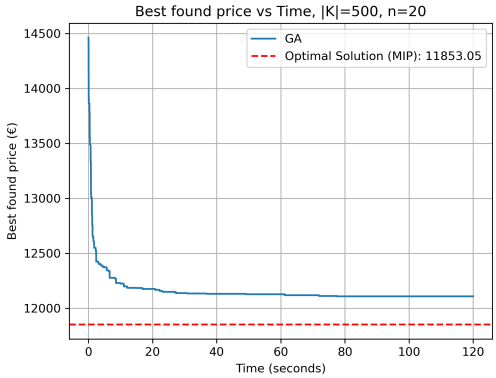
The experiments gave some surprising results, as it turned out that the mixed-integer program was able to handle way larger instance sizes than before as can be seen in Figure 16. The mixed-integer program was able to provide an optimal solution for 2 out of the 5 instances for 2000 orders, which is a significant difference. This difference might be explained by the fact that the price is rounded to 2 decimals, while in the previous experiments, the factor P_{kg} was multiplied directly with the weight, which was not rounded to 2 decimals. Rounding the costs to 2 decimals enables the mixed-integer program to discard all solutions that have more decimals, which significantly reduces the complexity of the problem. Another explanation might be that the number of possible costs $c_{k,i}$ is significantly decreased, which also reduces the complexity of the problem.

TABLE 2: The average cost of shipping 500 orders over 5 instances using the step-wise cost function of UPS Standard shipping in the Netherlands [4].

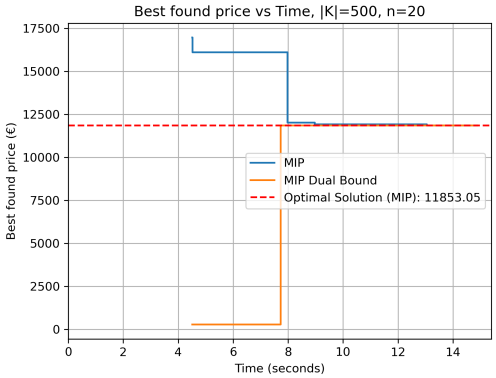
n	Total price	Price per order
5	€13588.59	€27.18
10	€11948.54	€23.90
20	€11384.38	€22.77
40	€10871.39	€21.57
80	€10501.25	€21.00
160	€10463.61	€20.93

Because of this observation, all main experiments were also done using the step-wise cost function to analyze the differences. The experiments on the number of cartons provided very similar results to the experiments on the linear cost function as can be seen in Figures 34, 35 and 36. The only notable difference in that experiment was that the run time was way lower for the step-wise cost function than for the linear cost function. The experiments on the number of cartons do however provide major new insights.

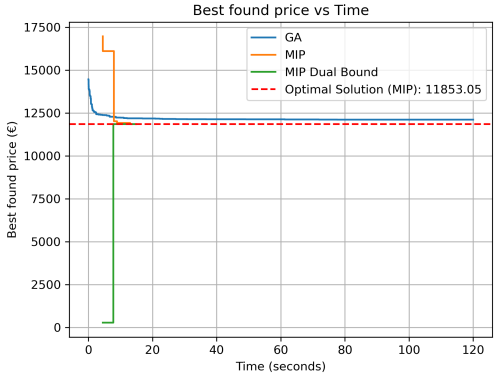
In Figure 15, the plots of an individual instance are shown for the base case $|K| = 500$ and $n = 20$ using the step-wise cost function of UPS. In Figure 15a it can be seen that the genetic algorithm can quickly improve to a good solution but needs a lot of additional time to improve further. Therefore, for order sizes |K| where the mixed-integer program is unable to solve the problem, the genetic algorithm is a great alternative. Figure 15b clearly shows that the MIP needs some initial time for defining variables and constraints, before finding a first solution to the problem. For this instance size of 500 orders that process takes about 4.4 seconds, but for larger instance sizes like $n = 1000$ or $n = 1500$ this takes about 15 seconds and 40 seconds respectively as can be seen in Figures 25 and 27. Figure 15d clearly shows that, whenever the MIP can solve the problem, the approximation error is initially smaller for the GA, but after a while the MIP can find the better carton set. For this specific experiment, the found carton sets are shown in Table 3.



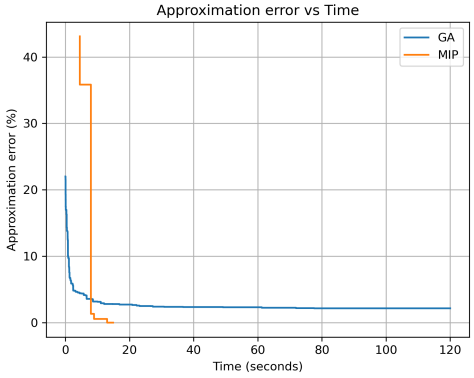
(A) Best-found price using the GA over time.



(B) Best-found price and lower bound using the MIP over time.



(C) The best-found price for both methods combined in one figure.



(D) The solution quality of both methods over time.

FIGURE 15: An example of the plots for $|K| = 500$ and $n = 20$ using the step-wise cost function.

TABLE 3: The dimensions of the carton sets found by the genetic algorithm and the mixed-integer program on the instance $|K| = 500$ and $n = 20$ that is shown in Figure 15. The total shipping costs are €12109.55 and €11853.05 for the GA and the MIP respectively. The approximation error of the genetic algorithm was 2.16% in this instance.

(A) Genetic algorithm

Length (cm)	Width (cm)	Height (cm)
28	14	12
35	19	15
45	26	21
46	23	18
53	28	26
55	24	22
61	32	30
69	39	29
74	48	38
78	54	44
84	41	37
88	56	41
90	58	32
95	54	43
95	65	47
97	50	38
104	63	48
110	56	47
120	54	25
120	70	62

(B) Mixed-integer program

Length (cm)	Width (cm)	Height (cm)
29	14	12
35	19	15
46	23	18
49	29	24
53	25	20
61	31	26
63	37	30
71	40	27
74	48	38
78	54	44
84	41	37
85	56	35
93	59	45
95	54	43
95	65	47
97	50	38
104	63	48
110	56	47
120	54	25
120	70	62

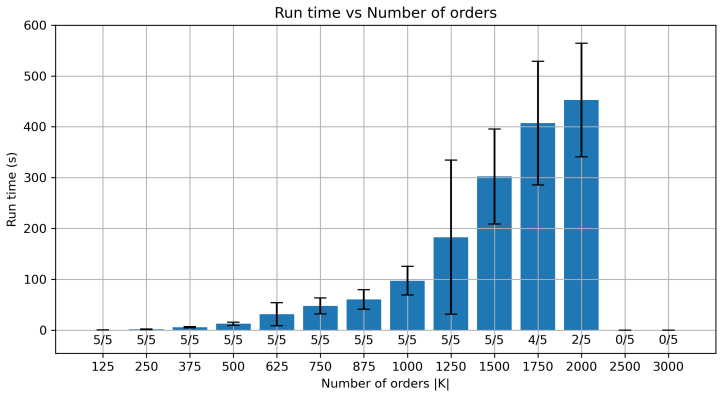


FIGURE 16: The average run time of the mixed-integer program over 5 instances for different numbers of orders $|K|$. The fraction below each bar represents the number of times the mixed-integer program was able to find an optimal solution within the 10-minute time limit. The experiments were run using the step-wise cost function.

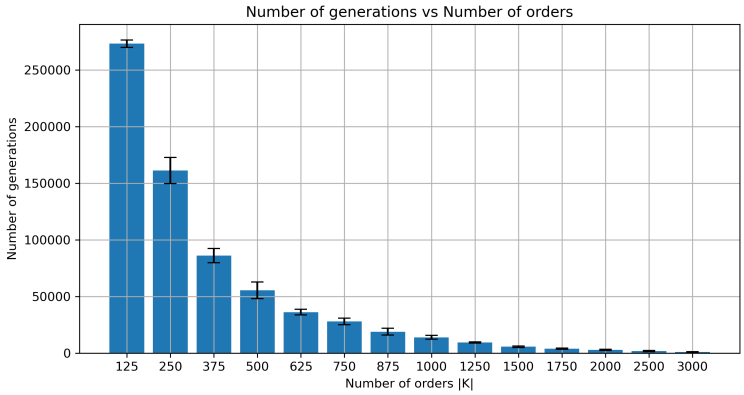


FIGURE 17: The average number of generations over 5 instances of the genetic algorithm for different numbers of orders $|K|$. The experiments were run using the step-wise cost function.

The plots of the number of generations against the number of orders for both the linear cost function in Figure 14 and the step-wise cost function in Figure 17 seem to closely follow the reciprocal function. This can be explained by the fact that having double the cartons requires double the calculations and is thus able to run half as many generations.

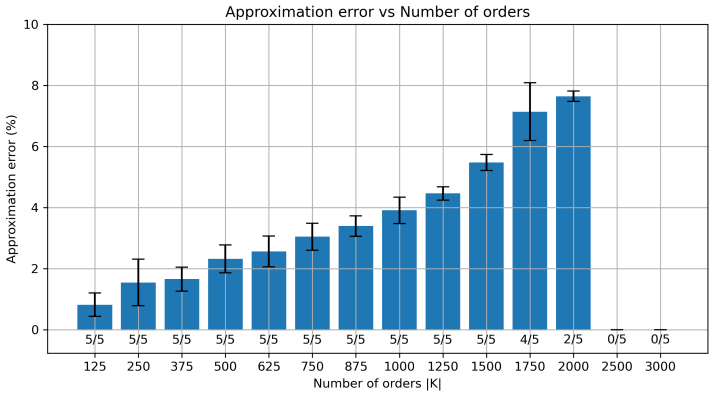


FIGURE 18: The average approximation error over 5 instances of the genetic algorithm for different numbers of orders $|K|$. The fraction below each bar represents the number of times the mixed-integer program was able to find an optimal solution within the 10-minute time limit. The experiments were run using the step-wise cost function.

The trend line in Figure 18 is clearer than that in Figure 12 which uses the linear cost function, because the experiments could be done for more instance sizes. Therefore, the experiments conclude with more confidence that there is an increasing trend line in the approximation error for larger instance sizes. Note that the horizontal axis in Figure 18 is not linear.

7 Conclusion

In this paper, two methods have been proposed to solve the carton set optimization problem. The mixed-integer program was implemented to evaluate the solution quality of the genetic algorithm. For smaller instances, the genetic algorithm is not able to find the optimal solution, in contrast to the mixed-integer program. For order sizes where the mixed-integer program is unable to solve the problem, the genetic algorithm is a great alternative. We see that the genetic algorithm can provide a near-optimal solution to the problem if the genetic algorithm is given enough time.

Although there does not seem to be a correlation between the number of cartons and the solution quality of the genetic algorithm, we can conclude that increasing the number of cartons decreases the average shipping costs of all orders. However, inventory costs and initial set-up costs for such carton sizes are not considered in this paper. Additionally, the prices of certain cartons might be much higher than other more standard carton sizes. To get a more accurate representation of a real-world situation, one should consider updating the model to include the costs of varying carton sizes, initial set-up costs and upkeep costs.

In this paper, the available carton sizes were limited to the generated order dimensions. To find the actual optimal carton sizes one should consider all possible combinations of dimensions. Note that this drastically increases the number of available cartons and thus increases the computation times of the mixed-integer program. This may also harm the solution quality of the genetic algorithm.

In the model, it is assumed that an order can be packed using a carton of the same dimensions. This is not completely accurate as the cartons have a thickness themselves. Additionally, for calculating the fitness value it is assumed that each order is packed using the optimal carton. In real-world scenarios, this might not be the case as packing might be done by humans or outsourced to machines who are both not flawless.

It is noted that taking the average values over 5 instances still seems to cause relatively big outliers within the data. To compensate for outliers, follow-up experiments are recommended to be run on more instances to ensure representative data. Moreover, to give an adequate representation of the real world, one should consider obtaining and using real-world data for follow-up experiments.

References

[1] Gurobi. https://www.gurobi.com/.

[2] Mixed-integer programming (MIP) - A primer on the basics. https://www.gurobi.com/resources/mixed-integer-programming-mip-a-primer-on-the-basics/.

[3] Python v3.10. https://www.python.org/.

[4] 2025 UPS Tarievengids - Basis [2025 UPS Rate Guide - Base]. https://www.ups.com/assets/resources/webcontent/en_GB/tariff-guide-base-NL.pdf, December 2024.

[5] K. S. Gurumoorthy and A. Hinge. Go Green: A Decision-Tree Framework to Select Optimal Box-Sizes for Product Shipments. In *Machine Learning and Knowledge Discovery in Databases*, pages 598–613, March 2023.

[6] A. H. Land and A. G. Doig. An Automatic Method of Solving Discrete Programming Problems. *Econometrica*, 28(3):497–520, 1960.

[7] S. J. Lee, E. P. Chew, L. H. Lee, and J. Thio. A study on crate sizing problems. *International Journal of Production Research*, 53(11):3341–3353, June 2015.

[8] M. Leonard. How DHL optimized packaging at warehouses to cut shipping costs. https://www.supplychaindive.com/news/dhl-carton-utilization-set-optimization-research-warehouse-box-shipping-cost-ecommerce-order/593164/, January 2021.

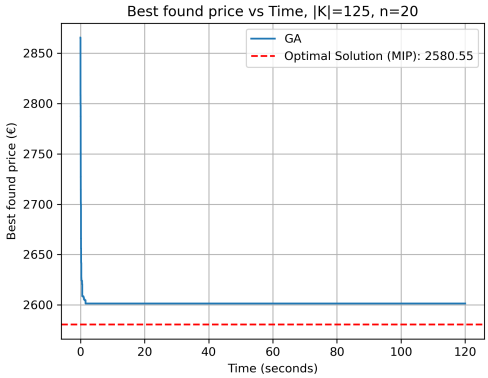
[9] L. Liska. How to Calculate Dimensional Weight for Shipping, July 2022.

[10] M. Singh and E. Ardjmand. Carton Set Optimization in E-commerce Warehouses: A Case Study. *Journal of Business Logistics*, 41(3):222–235, October 2020.

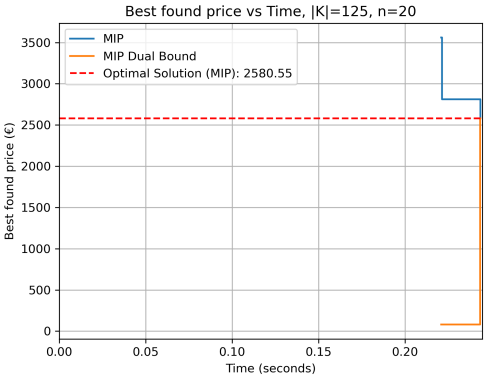
[11] R. C. Wilson. A Packaging Problem. *Management Science*, 12(4):B–135, December 1965.

[12] M. Wullink. Code for my thesis. https://gitlab.utwente.nl/milowullink/bsc-thesis/, January 2025.

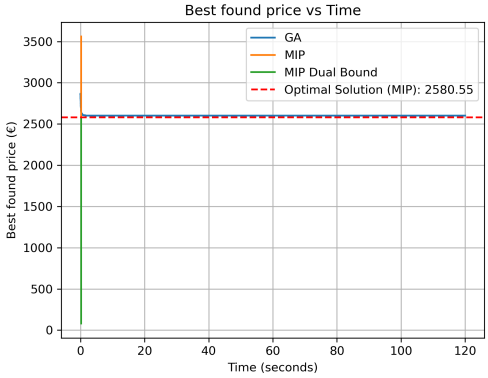
A Figures



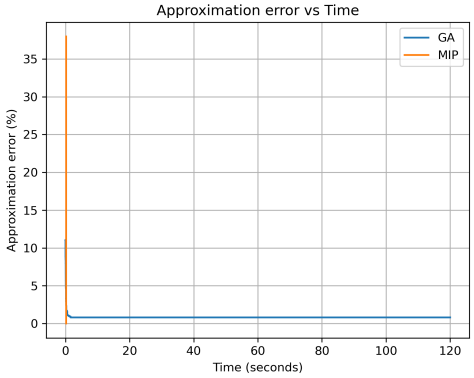
(A) Best-found price using the GA over time.



(B) Best-found price and lower bound using the MIP over time.

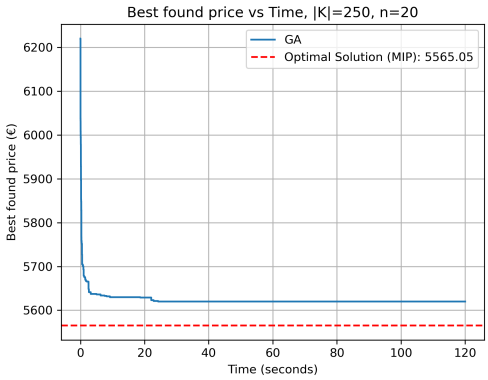


(C) The best-found price for both methods combined in one figure.

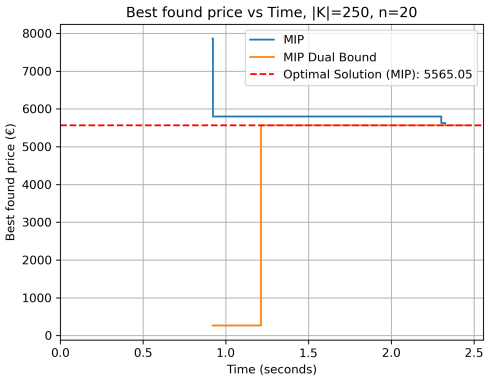


(D) The solution quality of both methods over time.

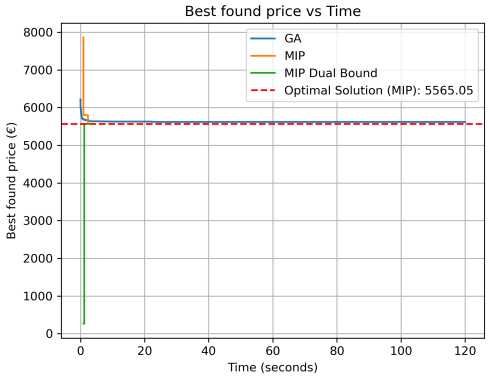
FIGURE 19: An example of the plots for $|K| = 125$ and $n = 20$ using the step-wise cost function.



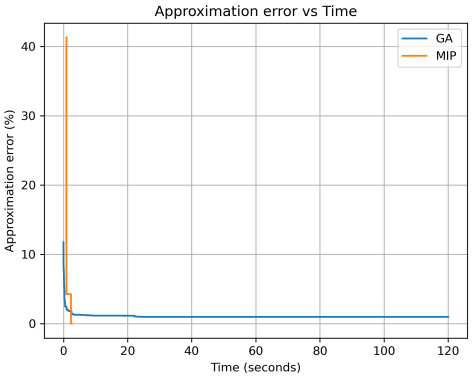
(A) Best-found price using the GA over time.



(B) Best-found price and lower bound using the MIP over time.

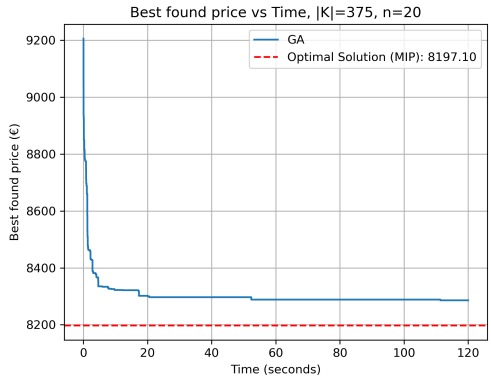


(C) The best-found price for both methods combined in one figure.

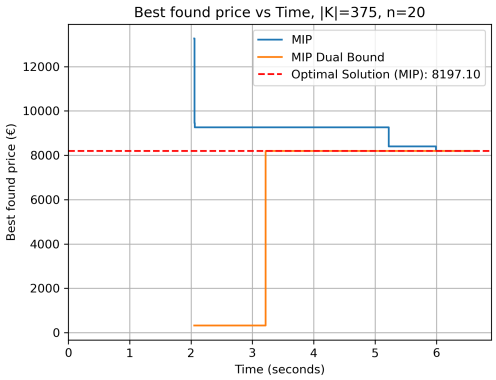


(D) The solution quality of both methods over time.

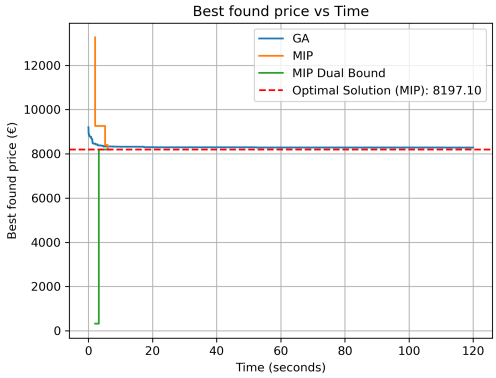
FIGURE 20: An example of the plots for $|K| = 250$ and $n = 20$ using the step-wise cost function.



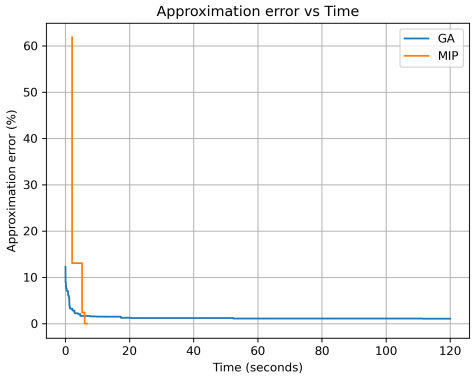
(A) Best-found price using the GA over time.



(B) Best-found price and lower bound using the MIP over time.

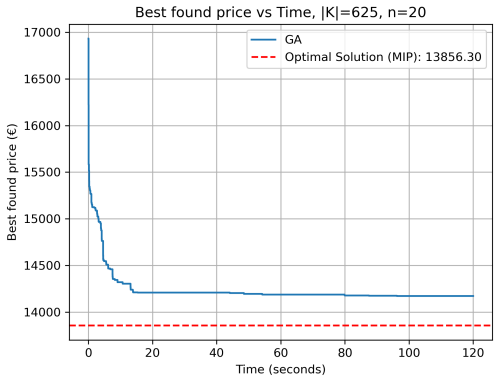


(C) The best-found price for both methods combined in one figure.

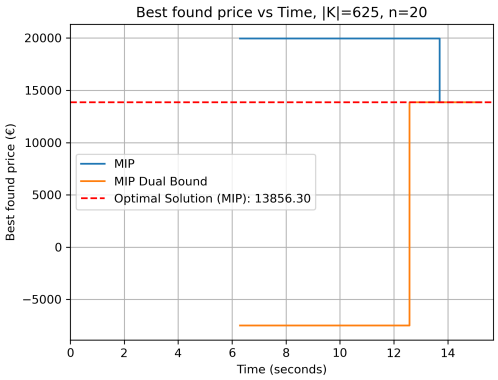


(D) The solution quality of both methods over time.

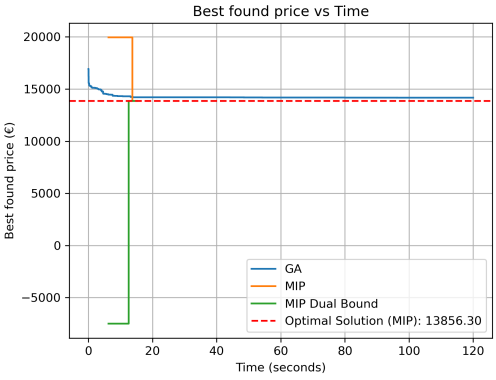
FIGURE 21: An example of the plots for $|K| = 375$ and $n = 20$ using the step-wise cost function.



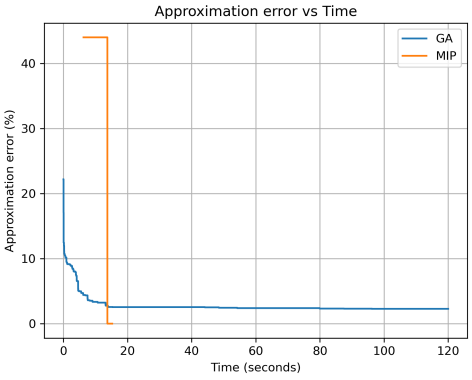
(A) Best-found price using the GA over time.



(B) Best-found price and lower bound using the MIP over time.

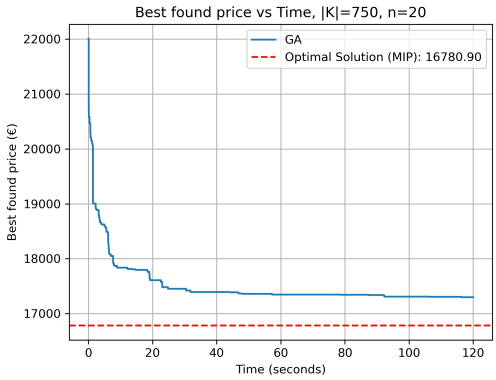


(C) The best-found price for both methods combined in one figure.

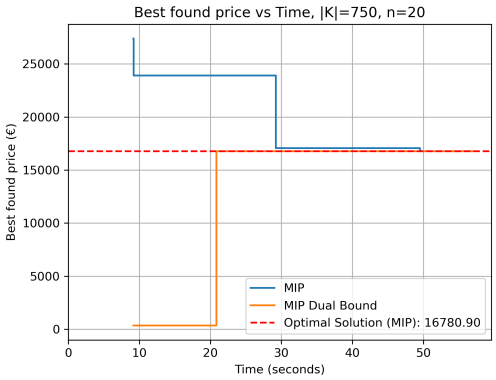


(D) The solution quality of both methods over time.

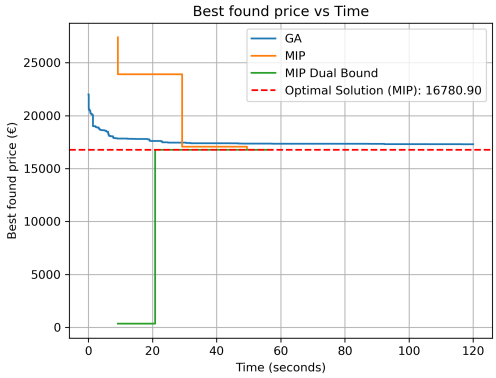
FIGURE 22: An example of the plots for $|K| = 625$ and $n = 20$ using the step-wise cost function.



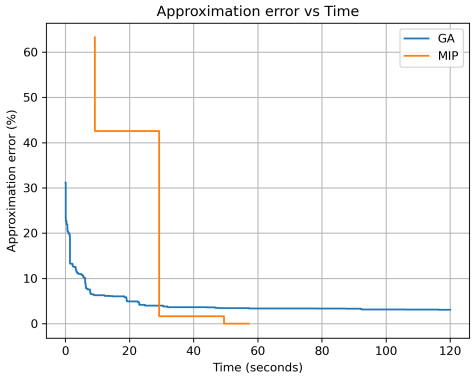
(A) Best-found price using the GA over time.



(B) Best-found price and lower bound using the MIP over time.

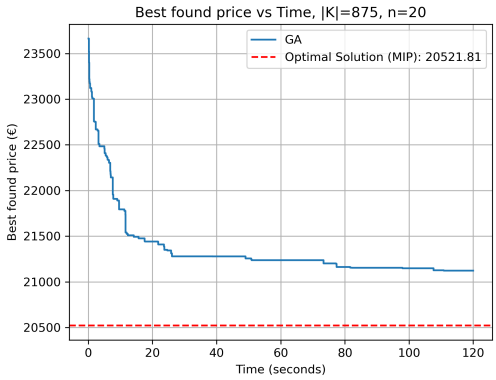


(C) The best-found price for both methods combined in one figure.

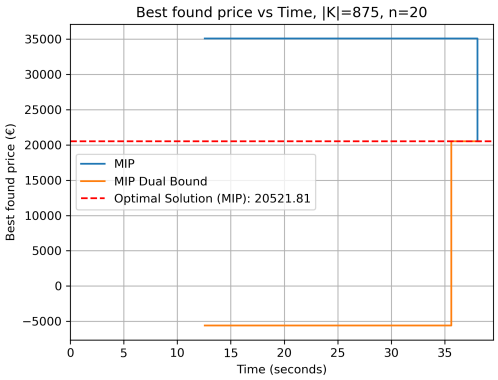


(D) The solution quality of both methods over time.

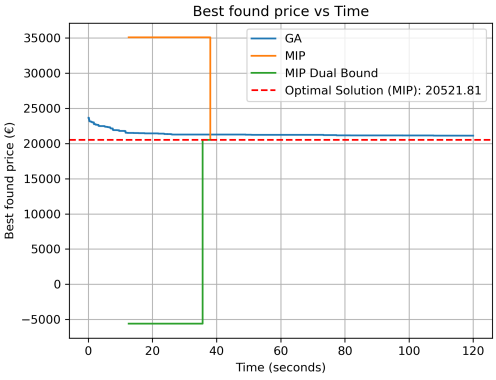
FIGURE 23: An example of the plots for $|K| = 750$ and $n = 20$ using the step-wise cost function.



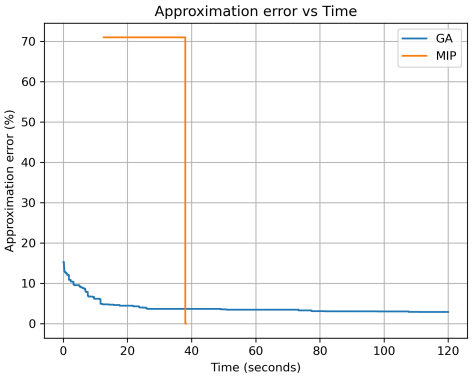
(A) Best-found price using the GA over time.



(B) Best-found price and lower bound using the MIP over time.

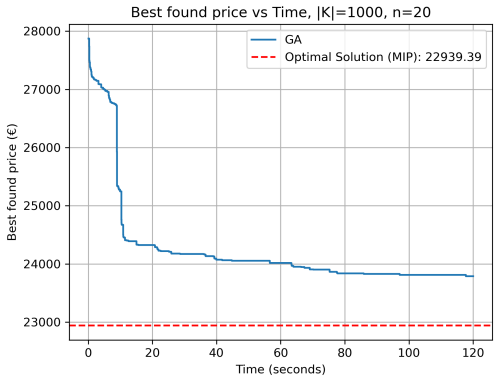


(C) The best-found price for both methods combined in one figure.

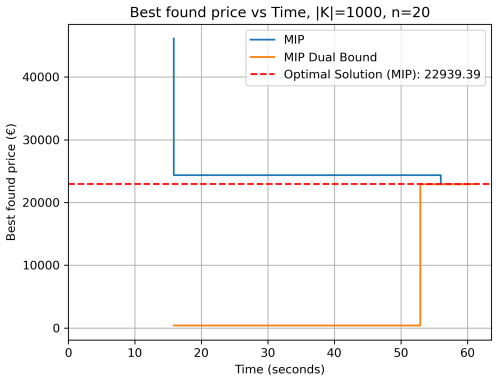


(D) The solution quality of both methods over time.

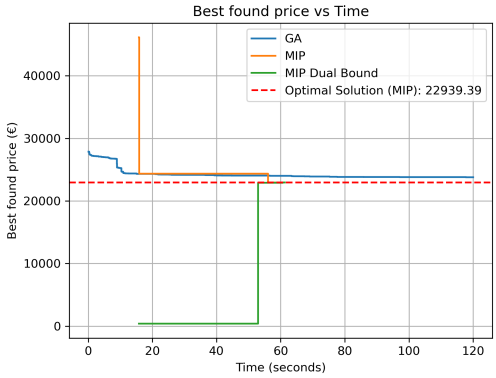
FIGURE 24: An example of the plots for $|K| = 875$ and $n = 20$ using the step-wise cost function.



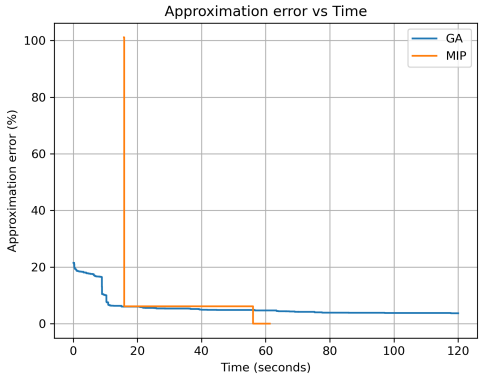
(A) Best-found price using the GA over time.



(B) Best-found price and lower bound using the MIP over time.

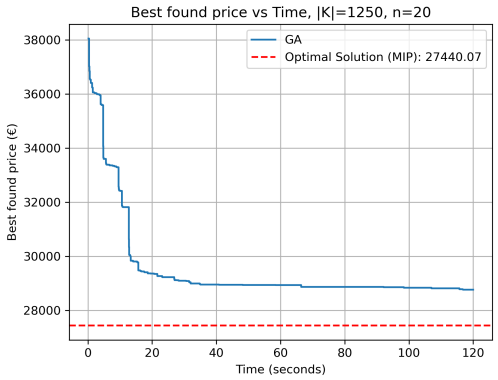


(C) The best-found price for both methods combined in one figure.

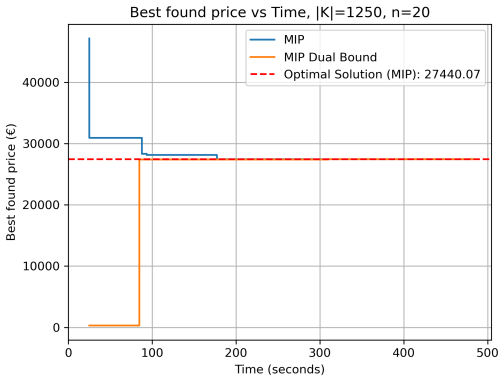


(D) The solution quality of both methods over time.

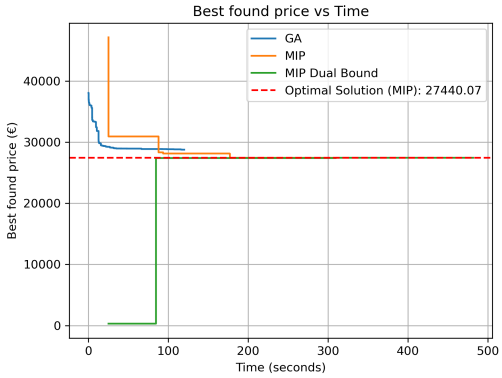
FIGURE 25: An example of the plots for $|K| = 1000$ and $n = 20$ using the step-wise cost function.



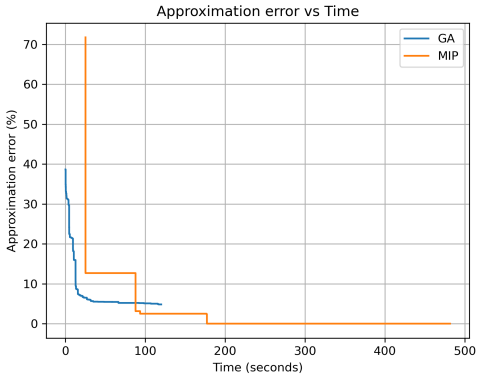
(A) Best-found price using the GA over time.



(B) Best-found price and lower bound using the MIP over time.

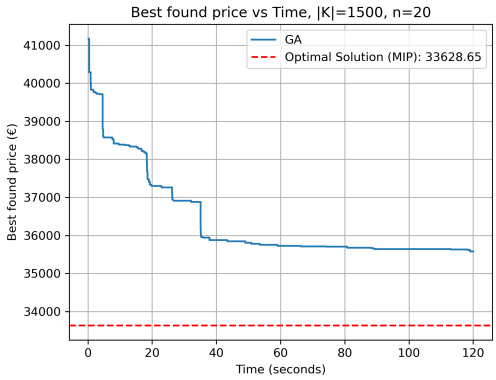


(C) The best-found price for both methods combined in one figure.

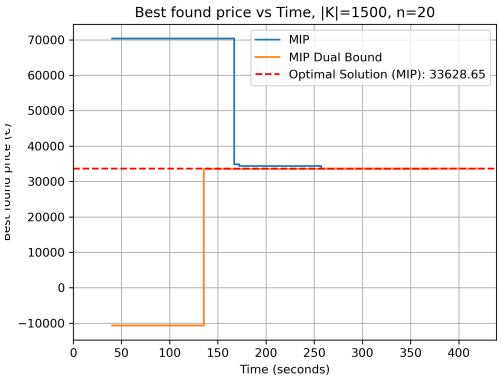


(D) The solution quality of both methods over time.

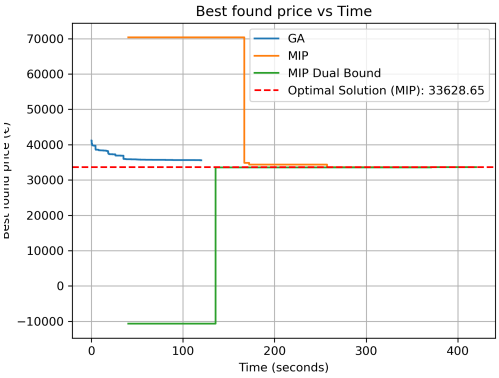
FIGURE 26: An example of the plots for $|K| = 1250$ and $n = 20$ using the step-wise cost function.



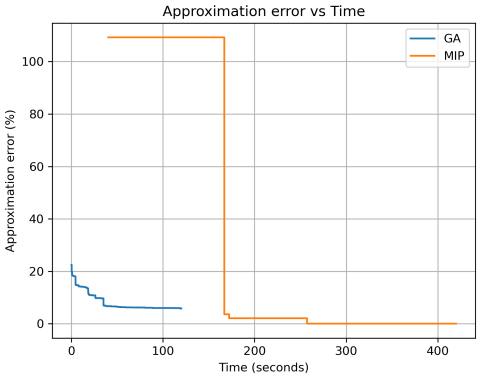
(A) Best-found price using the GA over time.



(B) Best-found price and lower bound using the MIP over time.

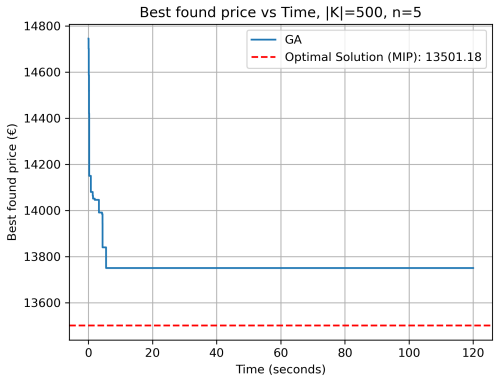


(C) The best-found price for both methods combined in one figure.

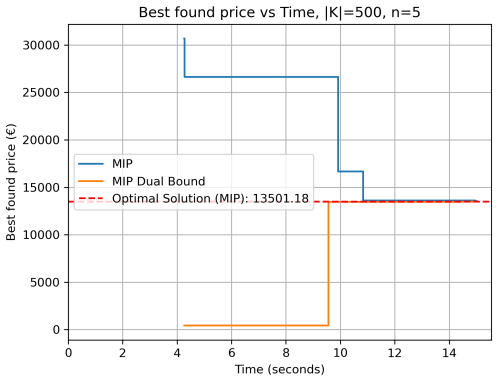


(D) The solution quality of both methods over time.

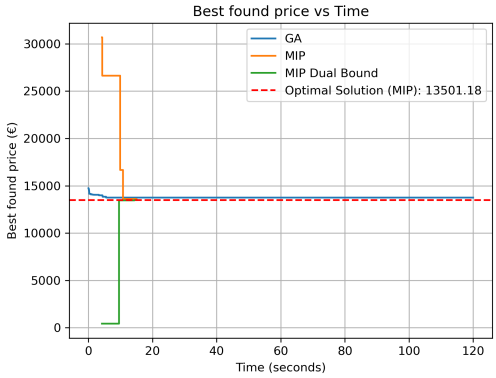
FIGURE 27: An example of the plots for $|K| = 1500$ and $n = 20$ using the step-wise cost function.



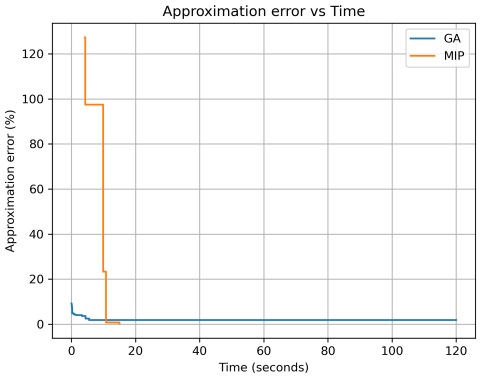
(A) Best-found price using the GA over time.



(B) Best-found price and lower bound using the MIP over time.

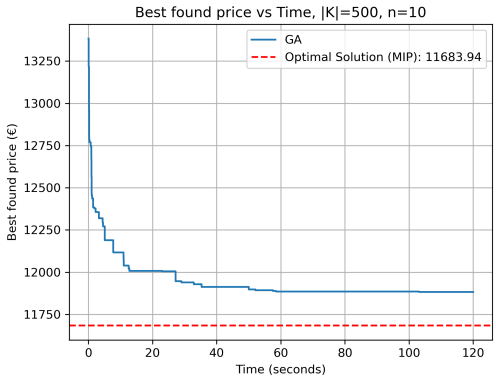


(C) The best-found price for both methods combined in one figure.

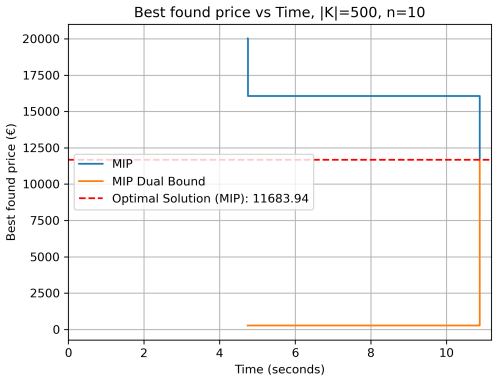


(D) The solution quality of both methods over time.

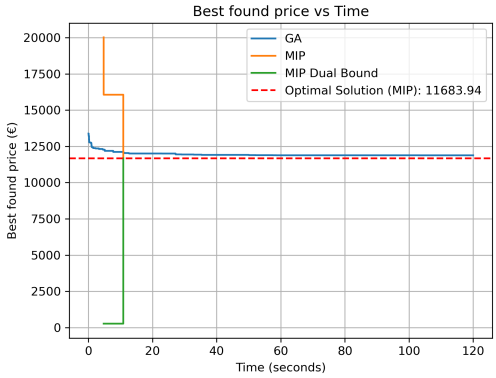
FIGURE 28: An example of the plots for $|K| = 500$ and $n = 5$ using the step-wise cost function.



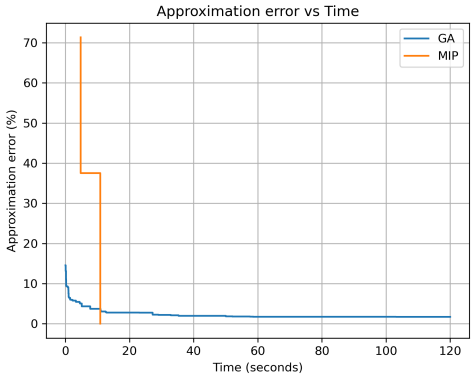
(A) Best-found price using the GA over time.



(B) Best-found price and lower bound using the MIP over time.

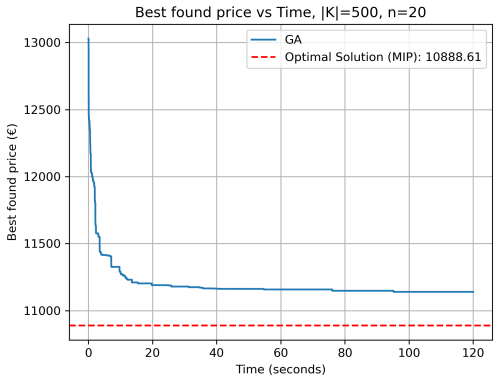


(C) The best-found price for both methods combined in one figure.

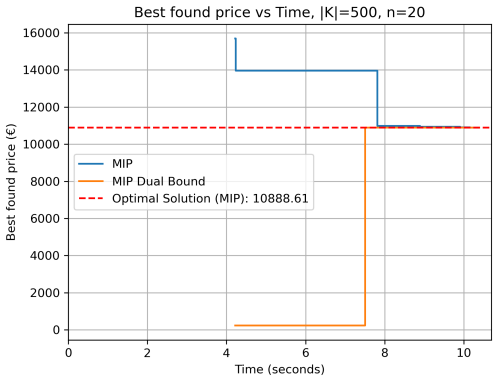


(D) The solution quality of both methods over time.

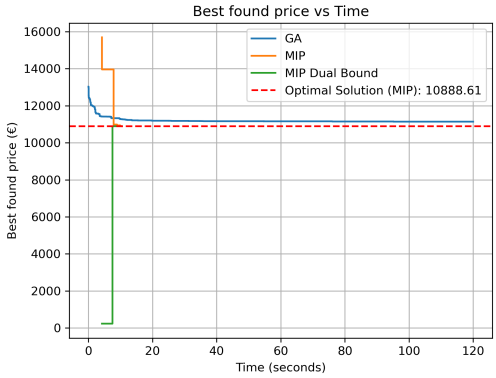
FIGURE 29: An example of the plots for $|K| = 500$ and $n = 10$ using the step-wise cost function.



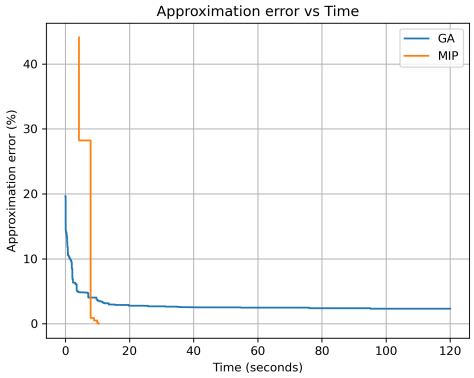
(A) Best-found price using the GA over time.



(B) Best-found price and lower bound using the MIP over time.

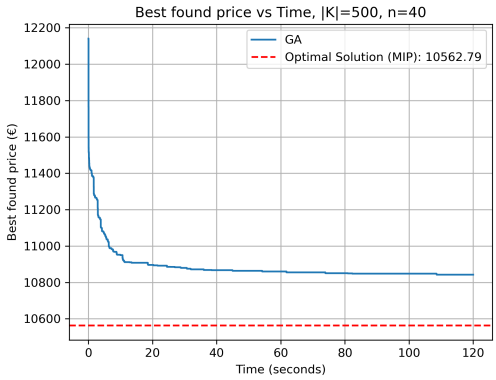


(C) The best-found price for both methods combined in one figure.

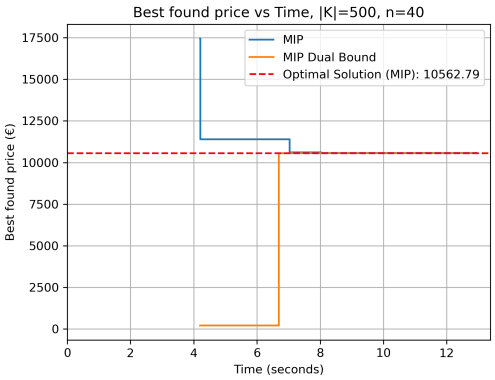


(D) The solution quality of both methods over time.

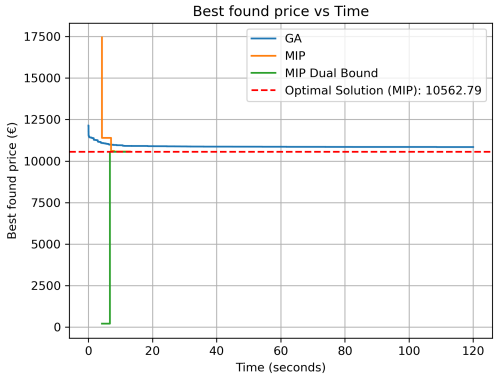
FIGURE 30: An example of the plots for $|K| = 500$ and $n = 20$ using the step-wise cost function.



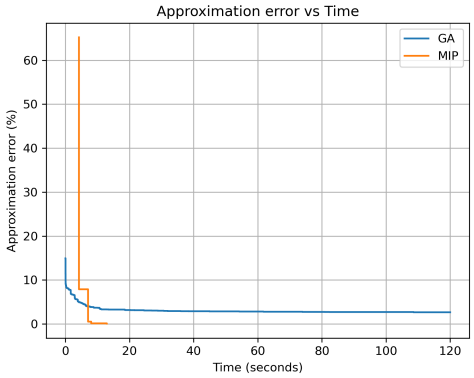
(A) Best-found price using the GA over time.



(B) Best-found price and lower bound using the MIP over time.

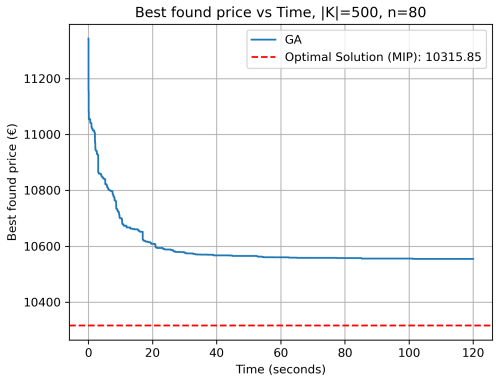


(C) The best-found price for both methods combined in one figure.

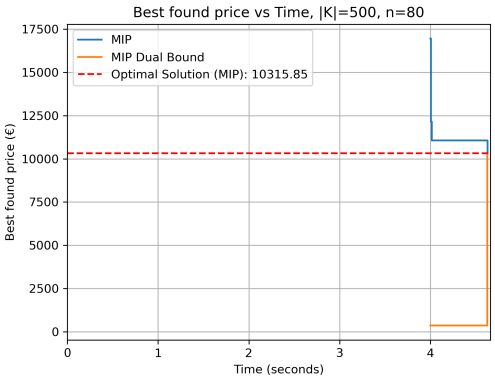


(D) The solution quality of both methods over time.

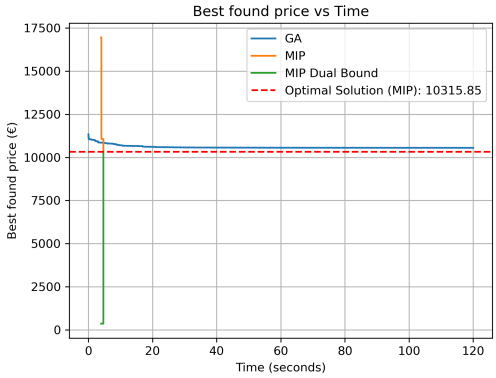
FIGURE 31: An example of the plots for $|K| = 500$ and $n = 40$ using the step-wise cost function.



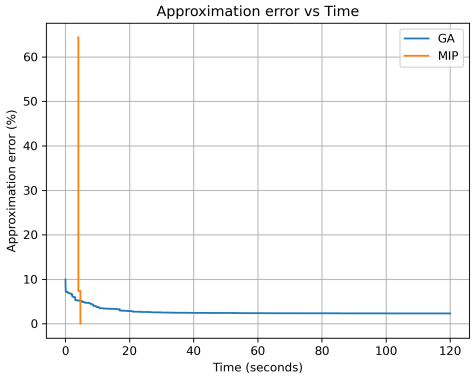
(A) Best-found price using the GA over time.



(B) Best-found price and lower bound using the MIP over time.

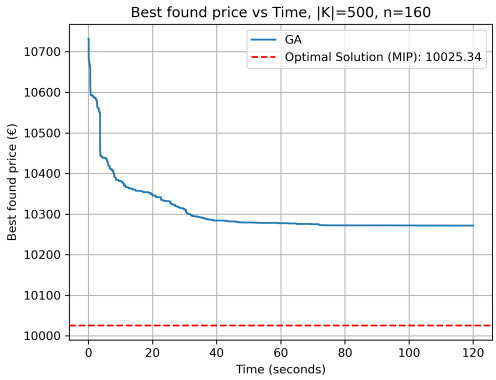


(C) The best-found price for both methods combined in one figure.

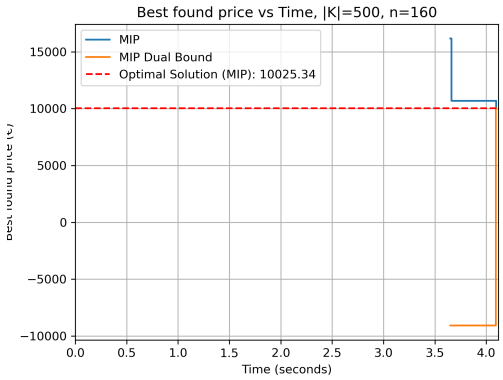


(D) The solution quality of both methods over time.

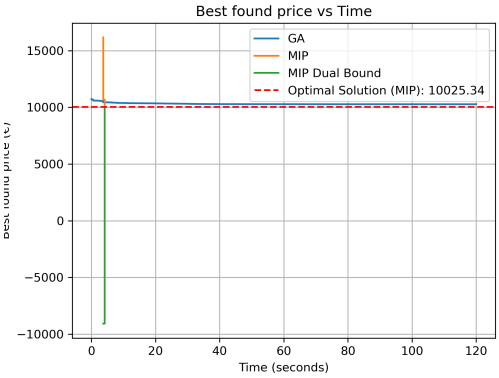
FIGURE 32: An example of the plots for $|K| = 500$ and $n = 80$ using the step-wise cost function.



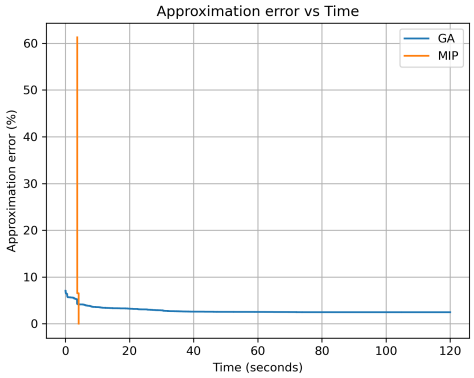
(A) Best-found price using the GA over time.



(B) Best-found price and lower bound using the MIP over time.



(C) The best-found price for both methods combined in one figure.



(D) The solution quality of both methods over time.

FIGURE 33: An example of the plots for $|K| = 500$ and $n = 160$ using the step-wise cost function.

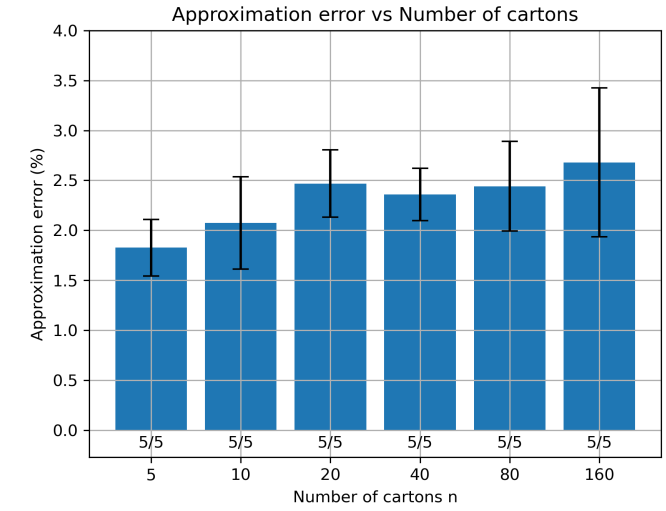


FIGURE 34: The average approximation error over 5 instances of the genetic algorithm for different numbers of cartons n. The fraction below each bar represents the number of times the mixed-integer program was able to find an optimal solution within the 10-minute time limit. The experiments were run using the step-wise cost function.

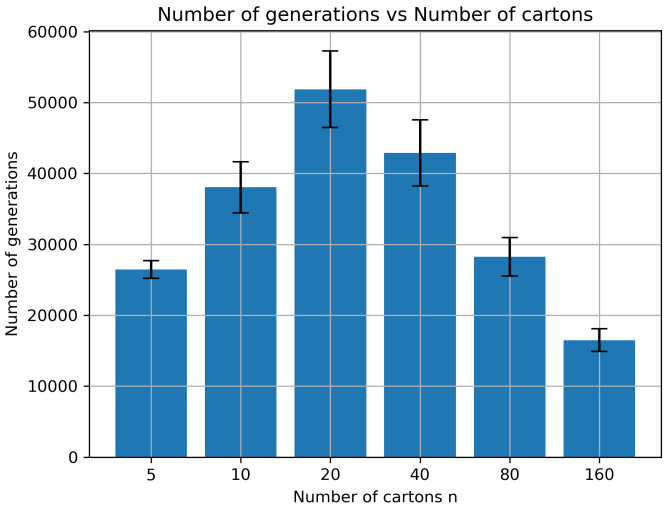


FIGURE 35: The average number of generations over 5 instances of the genetic algorithm for different numbers of cartons n. The experiments were run using the step-wise cost function.

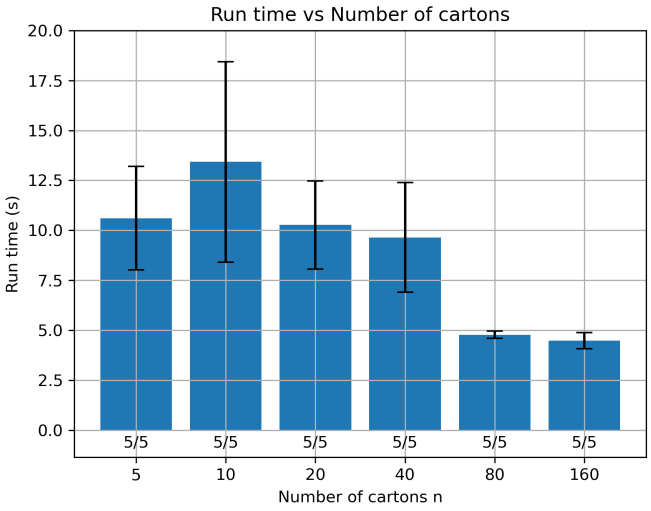


FIGURE 36: The average run time of the mixed-integer program over 5 instances for different numbers of cartons n. The fraction below each bar represents the number of times the mixed-integer program was able to find an optimal solution within the 10-minute time limit. The experiments were run using the step-wise cost function.