Algorithms for on-line bin covering problem

Govind, Hriday

Award date:
2022

BEP Report

Algorithms for on-line bin covering problem

Quartile 3-4 - 2021-2022

Full Name	Student ID	Study
Hriday Govind	1431625	Mechanical Engineering

Supervisors: dr.ir M.A Reniers & ir N. Paape

Eindhoven, June 23, 2022

Contents

1 Introduction

The food packaging industry is the largest domain in the entire packaging industry and is worth about $900 billion per year [McKinsey, 2020]. A key challenge in this sector is to optimally pack prescribed quantities of the required items in containers. Packing below the prescribed quantity is not acceptable to the customer and exceeding the limit leads to wastage of resources for the company. This paper aims to propose solutions to remedy the aforementioned situation of exceeding the prescribed limit (overfill) with a specific focus on packaging in the meat processing industry.

Bin-Packing is a classic optimization problem used in a variety of domains such as load balancing in networks and supply chain management [Angelopoulos et al., 2021]. It is an optimization technique where items of different sizes are packaged in a finite number of bins such that the total number of bins used is minimized. In this paper, we study the *Bin-Covering* (or dual Bin-Packing) variant of the Bin-Packing problem. Bin-Covering is the NP-hard problem of packing items from a given set into bins so as to maximize the number of bins used, subject to the constraint that each bin is filled to at least a given threshold [Assmann et al., 1984]. The NP problems are a set of problems whose solutions are hard to find but easy to verify and are solved by Non-Deterministic Machine in polynomial time [GeeksforGeeks, 2021]. A problem X is NP-Complete if there is an NP problem Y, such that Y is reducible to X in polynomial time [GeeksforGeeks, 2021]. A NP-hard problem is one that is at least as hard as an NP-Complete problem and a NP-hard problem X can be solved only if there exists a NP-Complete problem Y that is reducible to X in polynomial time. The Bin-Covering problem can be largely categorized into on-line and off-line problems. In the on-line variant, items arrive one at a time in random order and each item must be assigned in a bin before the next item can be assigned. On the other hand in the off-line variant, all items are known upfront after which algorithms are used to pack the items into the bins. This paper focuses on the on-line variant of the problem.

1.1 Case Description

This paper is inspired by a real-world problem encountered by Marel, an Icelandic company engaged in supplying full-line solutions for food processing systems. The focus here will be on their poultry processing systems. A frequently used machine in their poultry and fish processing systems is a grader which can be seen in Figure 1 below.

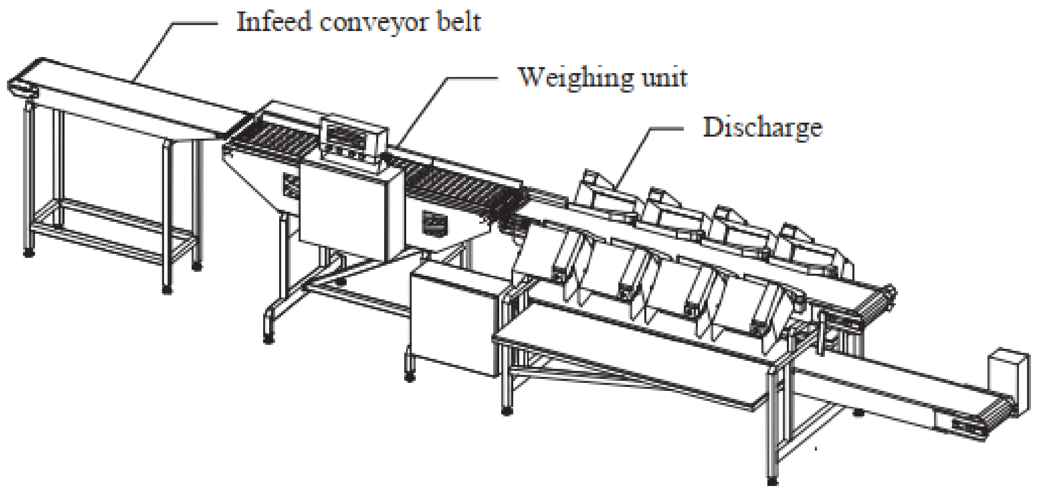


Figure 1: Grader Machine [Asgeirsson, 2014]

Before arriving at the grader, the poultry meat moves in shackles through a long conveyor belt in a continuous manner where they are cut into fillets by cutting machines placed at fixed points in the supply line. Once cut into fillets, the fillets are fed into the infeed conveyor belt of the grader. They eventually move to the weighing unit where each individual fillet is weighed. In the discharge section of the machine, trays (or bins) are present that receive the fillets and once a threshold weight is reached or some other condition such as the maximum time spent by the bin in the system is fulfilled, the trays are sent out and a new empty tray takes its place.

1.2 Research question

The purpose of the grader is to pack the fillets together to a guaranteed minimum weight. This packing is done through the use of an algorithm. The goal is then to minimize the excess of the minimum weight which is known as *overfill.* The weight distribution of the items is described by a Normal (Gaussian) Distribution and several algorithms are formulated and tested in order to obtain minimal overfill. Hence the research question is:

"Which algorithm is most suitable to minimize overfill, given a guaranteed minimum weight"

1.3 Related Literature

On the subject of Bin-Covering, a very limited amount of research has been performed on the on-line variant as opposed to the off-line variant of the subject. The most prominent literature in the sub-group of on-line algorithms are the works of Assmann, 1983 and Assmann et al., 1984. Most of the literature in the field largely build upon these works, some of which are mentioned below.

The work of Assmann et al., 1984 focuses on the dual Bin-Packing problem where a given set I is partitioned into a minimum number of sets, none of which exceeds the given threshold in size. An important remark made in this article is that efficient optimization algorithms are unlikely to exist for NP-hard problems such as Bin-Packing and that approximation algorithms are in most cases the best possible solution to the given problem. Jabari et al., 2016 discusses heuristic algorithms in order to find approximate solutions through the use of greedy algorithms.

Csirik has two notable works on the subject. The conclusion of the Csirik and Totik, 1988 paper was that for the dual Bin-Packing problem formulated by Assmann et al., 1984, there does not exist a performance ratio greater than $\frac{1}{2}$. The performance (or competitive) ratio of an on-line algorithm is the ratio of its performance on a given input in relation to the performance of an optimal off-line algorithm on the same input [Aspnes, 1998]. The work of Csirik et al., 2010 deals with on-line and semi on-line algorithms for a variety of different scenarios such as changing the number of bins, input sequences with (non)increasing item sizes etc.

[Boyar et al., 2021] improved upon the competitive ratio from 0.5 to 0.53 through the use of advice. Advice complexity is a formalized way of measuring how much knowledge of the future is required for an on-line algorithm to obtain a certain level of performance, as measured by the competitive ratio. When such advice is available, algorithms with advice could lead to semi on-line algorithms. A sub-category of advice is an approach called lookahead on which Grove, 1995 takes an interesting approach by analysing the cost function of items in the future as opposed to simply some number of items in the future. Besides lookahead, the concept of limited migration has also been investigated by Berndt et al., 2019. This involves the concept of a migration factor β that allows for some decisions made to be reversed and the items repacked.

Cardinality constraints are another aspect studied in this field. Epstein et al., 2013 describes this by stating that a given parameter k indicates a lower bound on the number of items that a covered bin must contain in addition to the condition of the total size. A bin is said to be covered when the allocation of items for that particular bin is complete.

Practical physical constraints such as geometric constraints are studied under subdomains like geometric bin covering and vector bin covering. Christensen et al., 2017 discusses an example of geometric bin packing by prescribing algorithms to pack rectangular items in a minimum number of unit-size square bins.

Zhang, 2001 builds on the work of Assmann et al., 1984 with an interesting variant of the on-line batching problem by describing algorithms for variable-sized Bin-Covering which closely resembles the scenario detailed in this paper.

However, the most directly related paper to the research question at hand is the doctoral thesis of Agni Asgeirsson [Asgeirsson, 2014]. Asgeirsson's work was based on the same grader machine described above and dealt with the same end goal of perfect packing and minimizing overfill. Asgeirsson defines a *Prospect Algorithm* and then proposes modifications such as item rejection, variable bin capacities and item constraints per bin to enable it to solve problems other than the traditional Bin-Packing and Bin-Covering problems. The key rationale behind the Prospect Algorithm is to use the information on the item distribution and the information of the empty space available in a bin in order to fill a bin with a minimal overfill. Asgeirsson also describes Markov modelling and Markov Decision Process (MDP) approaches for Bin-Covering. These were then extended by adding lookahead functions to the problem and investigating Subset-Sum problems.

While there exist clear similarities between the dissertation of Asgeirsson and this paper, key notable differences exist. The overall time frame of research of Asgeirsson, 2014 on the subject was over a much more extended period of time (13 years) and involves a great deal of complexity both in terms of practical modelling of the system as well as in the algorithms used. Firstly, the processes described in this paper are completely time-independent and focus purely on the performance of the algorithm with respect to the overfill. Asgeirsson's dissertation not only accounts for time in the general process but also other time-dependent parameters such as wastage and spoil rate of meat. Most of the results described in Asgeirsson's work regarding the performance of algorithms and the overfill involve plotting the overfill as a function of the bin size ranging from 200 to 800 for a particular algorithm. This paper however focuses on comparing the performance of different algorithms for a fixed bin size. The effect of changing bin size however will be discussed briefly in the Results section.

Asgeirsson's analysis revolves around lower weight distributions with most of the analysis being done with distributions having a mean of 100 grams. However, there are also heavier items of meat that often have to be packed into similar bin sizes. This has been depicted in Table 4.1 in Asgeirsson, 2014 along with one instance of analysis conducted. However, Asgeirsson's work lacks further depth on this subject. This is the domain that this paper really focuses on by testing algorithms on heavier weight distributions. This approach also presents an additional challenge that there will be fewer items per bin allocated for a similar threshold due to the increased weight of each individual. This means that the opportunity to optimize the packaging of items reduces leading to additional overfill. However, being a real-life scenario encountered, it remains an important area of interest to investigate.

The other question that this paper seeks to answer is whether the overfill can be reasonably contained using much more simpler algorithms. The algorithms used by Asgeirsson such as the Prospect Algorithm involve a high degree of mathematical complexity which could be computationally expensive, especially when used on a large scale.

1.4 Software Modelling

The aforementioned model had to be modelled using software which must possess the necessary features in order to answer the research question. These include aspects such as the ability to model the individual model components and their functions, the ability to store information about each individual object passing through the model and the ability to run simulations and gather results that are useful for interpretation. Each of the above is explained in detail below:

1.4.1 Required Model Components

The required model components and their functionalities are described below. The functionalities of each of the components have been visually modelled using Business Process Modelling Notation (BPMN) [Signavio, 2022] which is commonly used to model business processes and business information systems. A legend of all the symbols used can be found in Figure 2

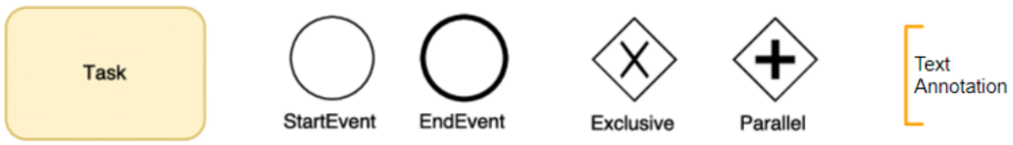


Figure 2: BPMN Legend

A task is used to represent the occurrence of any process in the model. A Start Event is used to define the start of a sequence of processes and an End Event defines the end of a sequence of processes. The next two symbols in the legend represent gateways namely exclusive and parallel gateways. Exclusive gateways act as a decision point in the process wherein the successive set of decisions are based on the fulfilment of a condition. This is very similar to if statements used in programming. Parallel gateways, on the other hand, are used to represent two or more processes happening simultaneously at a given instance of time. Finally, a text annotation to provide additional information regarding any of the symbols. Each of the diagrams are enclosed in dotted lines known as groups which indicate that a sequence of processes belongs to a particular model component.

Generator

A generator is needed in the model in order to create the object that passes through the model. It is imperative to have the ability to define the properties of the object during the generation of the object. **In this particular scenario, the ability to define the weight parameter during the generation of the object removes the need to separately model the weighing unit component of the system.** However in reality, the weighing unit gives an approximation of the actual weight. Information regarding the weight of the object can directly be obtained from the object itself and will be elaborated further in a later section.

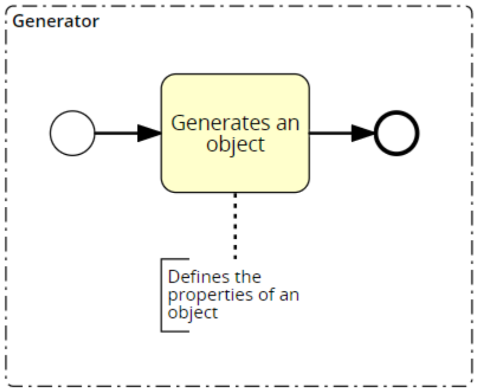


Figure 3: Working of Generator

Infeed Conveyor Belt

The infeed conveyor belt is used to transport the generated object to the next component of the model for further actions to be taken. In terms of modelling, this simply involves a component with the ability to move objects at a particular speed defined by the user.

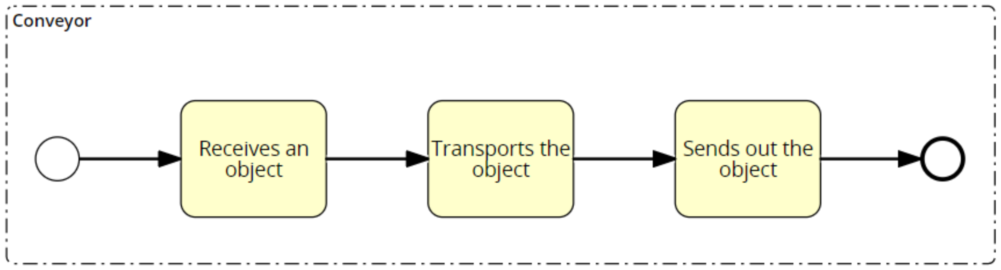


Figure 4: Working of Conveyor

Trays

Trays need to have the ability to hold all the objects that are placed in them and retain the cumulative properties of the objects throughout their movement through the model. While not a direct system component in itself, the overall modelling also takes a different angle with the introduction of trays since prior to the stage where the objects are placed into their respective trays, the flow of the individual objects through the model is modelled. However, after the placing of the objects into trays, the flow of the trays through the system is modelled.

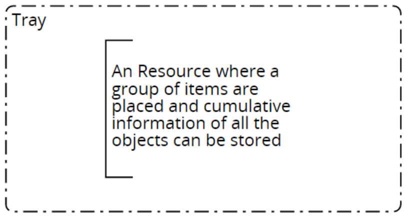

Tray
An Resource where a group of items are placed and cumulative information of all the objects can be stored

Figure 5: Working of Trays

Discharge

This is the most important and complex component of the model. It involves receiving an object from the conveyor, allocating the object to a respective tray through the use of a prescribed allocation algorithm, sending out trays when they reach their minimum guaranteed threshold weight followed by the creation of new empty trays in its place.

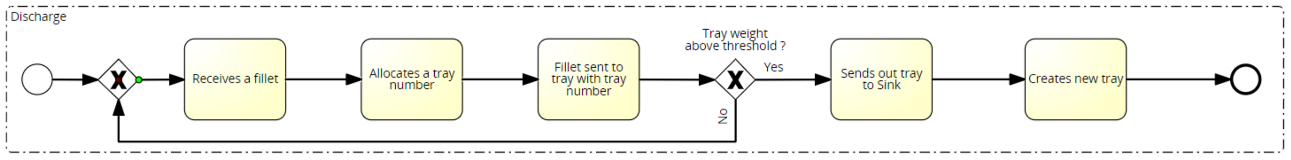


Figure 6: Working of Discharge

Sink

This is a simple component which represents the final destination of the final batched tray. In a physical sense, this can be a place where the trays are finally packed and shipped out to the end customer. In terms of its functionality, it simply receives the finally sent out tray from the Discharge.

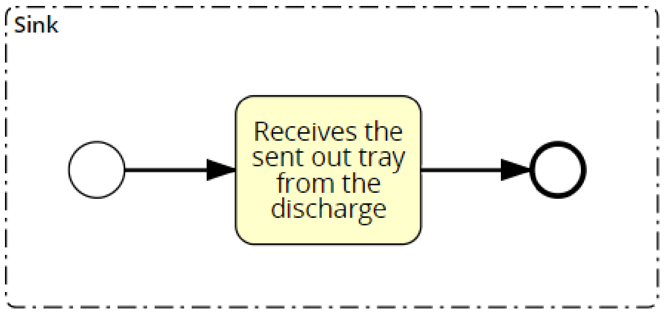


Figure 7: Working of Sink

Links

All of the system components when modelled in the software need to be linked to each other in order for the objects/ trays to flow to the next component and not get 'lost' in the process

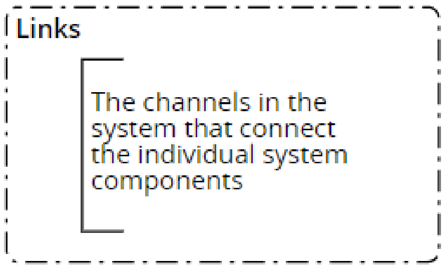

Links
The channels in the system that connect the individual system components

Figure 8: Working of Links

1.4.2 Ability to store and retrieve information of objects

Similar to the work of Asgeirsson, 2014 that involves the use of a Prospect Algorithm, the algorithms discussed in this paper also heavily rely on the information stored on the object in order to make decisions (with respect to tray allocation). Hence it is essential for the modelling software to possess the ability to store and make use of the information of the object in the decision-making process.

1.4.3 Ability to run simulations and retrieve results

The method of validation of the algorithms is through running simulations. The results that are retrieved from the simulations of the different algorithms are then put through statistical tests and compared with each other in order to obtain the most optimal algorithm among the ones compared.

1.5 Software Selection

Taking into account the aforementioned requirements, the search for a suitable software to fulfil these requirements had begun. At present, most manufacturing systems use expensive, customized licensed software in order to model their systems. Konverge is a good example of a company that develops custom software for a variety of industries including manufacturing [Konverge, 2022]. This is not only expensive to use and scale up when necessary but also poses a challenge to quickly add additional features on demand when deemed necessary. Hence opting for an open-source software in order to model the system would be the most robust and cost-efficient solution.

Among the various open-source programming languages, a Python-based software was chosen to be used due to its clear structure and syntax along with its wide prevalence in industry with a large number of ready-to-use libraries that can be directly implemented to achieve the intended task. Furthermore, detailed documentation and troubleshooting are present which allows for a smoother development process. Since the system under investigation can be modelled as a set of logically separate processes that autonomously progress over time, a Discrete Event Simulation (DES) can be used [Barrett et al., 2022]. A DES program takes into account those points in time (events) that are of importance to the further course of the simulation [Mes, 2017]. Some instances of these include entering a machine, leaving it, or moving to another machine. Any intermediary movement between those events is of little importance. Among the various softwares built on Python that allow for DES, the ones that were shortlisted and further investigated were SimPy, ManPy, PyCh and Salabim.

SimPy stands for 'Simulation in Python' and is a package for process-oriented DES written exclusively in Python [Dagkakis et al., 2015]. Processes in SimPy are defined by Python generator functions and may, for example, be used to model active components like customers, vehicles or agents [SimPy, 2022]. SimPy also provides various types of shared resources to model limited capacity congestion points (like servers, checkout counters and tunnels). It can also be used to produce visualizations of performance metrics by connecting to other Python libraries like *Matplotlib* or *Tkinter* [Brown, 2022].

ManPy stands for "Manufacturing in Python" and it is a layer of Discrete Event Simulation (DES) objects built in SimPy [Manpy, 2022]. It is designed to offer a similar world view in an OS coding environment. The user gets prefabricated components that he or she is able to connect as black boxes in order to form a model. The objects implement a well-defined API of methods so that an advanced user can easily customize their behaviour on specific occasions [Dagkakis et al., 2015].

Salabim is a new open-source object-oriented package specially developed for discrete event simulation of complex control in logistics and production environments [Ham, 2018]. The choice of Python as the host language means that simulations can be easily combined with powerful packages for statistical processing and presentation, web interfaces, machine learning, databases, etc. One of the key features of Salabim is the integrated animation engine. The package can be used for a wide range of simulation applications, such as (air)ports, hospitals, warehouses, job shop production, distribution systems and communication networks.

PyCh is a Python package based on SimPy for discrete-event simulation [Paape, 2022]. It is a tool developed for the course "Analysis of production systems" (4DC10) at Eindhoven University of Technology. A port represents the Transmission Control Protocol (TCP) port which is a virtual point where network connections start and end [CloudFlare, 2022]. A TCP is a connection-oriented communications protocol that facilitates the exchange of messages between computing devices in a network [SDX, 2022]. PyCh is a port of Chi3. It makes use of channels which easily perform the function of links between components as discussed earlier. A channel is a class that enables the sending and receiving of entities [Chi, 2022]. These entities can be any type of object. The processes are defined by send(entity) or receive(). A sender waits till a receiver is ready to receive and a receiver waits till a sender is ready to receive. In the case of multiple senders and receivers waiting, a non-deterministic (random) sender/receiver is chosen. Finally, the overall syntax of PyCh is quite easy to use in the context of modelling manufacturing systems.

1.5.1 Final Selection

Initially, ManPy was considered due to its libraries highly suited towards manufacturing systems along with its user-friendly block-based component modelling which somewhat resembles the BPMN diagrams used to explain the system components earlier. However, ManPy is not compatible with the latest Python version, because the software received its last update in 2016 [Lang et al., 2021]. In the initial stages of the research, it was difficult to predict whether this might cause challenging bottlenecks in terms of compatibility with other libraries and hence was decided not to be used.

Salabim was also a good option to proceed with. It allows for complex models to be easily modelled quickly [Lang et al., 2021]. It also supports additional components of the programming language SIMULA which simplifies the implementation of processing logic [Lang et al., 2021]. For similar university-level works using python-based simulations to perform analysis such as the thesis of [Kuipers, 2021], it was found that saving simulation models that have been executed after the use of resampling is very difficult. Resampling is a methodology of economically using a data sample to improve the accuracy and quantify the uncertainty of a population parameter [Brownlee, 2018]. While this paper purely focuses on the overfill and not the time involved in the entire process, resampling would not be necessary. However, should this be considered in the future as an extension of the work presented here, the use of Salabim would introduce challenges.

Among the remaining two options of SimPy and PyCh, both options were very similar which is to be expected since PyCh is based on SimPy. However, in the end, the ease of modelling of the system through the use of channels in PyCh along with the familiarity of its syntax proved to be the decider in choosing **PyCh** as the software to model the system.

2 Experimental Setup

The main aim of the experiments run is to test different batching algorithms in the Discharge and then determine the algorithm that results in the least overfill among the ones tested. Some of the assumptions that are taken in the model are as follows.

- The weight of the fillets generated is random on a Normal (Gaussian) Distribution with a mean of 200 grams and a standard deviation (SD) of 20 grams which is quite representative of real-life values.
- The model will use 8 trays and the target weight (threshold) of each tray will be a 1000 grams (the effect of changing these will be discussed in Section 4).
- Time is not being considered as a factor in the decision-making process. The results purely focus on the overfill parameter.
- Thereby the travel time of objects between system components is considered to be instantaneous.
- Other time-dependent factors such as spoil rate of meat etc. are also ignored in this simplified model.

2.1 Model

2.1.1 Object Oriented Programming

Since the model and algorithms are entirely based upon the object and its properties, the use of Object-Oriented Programming was most suited to accomplish the task. OOP is a computer programming model that organizes software design around data, or objects, rather than functions and logic. An object can be defined as a data field that has unique attributes and behaviour [Gillis and Lewis, 2022]. Some key concepts of OOP are discussed below.

- Classes are user-defined data types that act as the blueprint for individual objects, attributes and methods [Gillis and Lewis, 2022].
- Instances are objects that belong to a particular class. Objects can correspond to real-world objects or an abstract entity such as integers, lists, functions etc. [Gillis and Lewis, 2022].
- Attributes are defined in the class template and represent the state of an object. Objects will have data stored in the attributes field [Gillis and Lewis, 2022]. Class attributes belong to the class itself.

The principle of Inheritance is used quite frequently in the model. Inheritances are relationships between classes. They consist of a base class and a derived class (or multiple derived classes). The derived class inherits all the attributes of the base class. This can be best explained with a visualization and example. Let us consider a class Tree with attributes species, height, age, grow and reseed. There are two types of trees (or derived classes) namely Coniferous Trees with attribute reseed and Deciduous Trees with attribute dropLeaves. Now, should we take an instance of a Deciduous Tree, the object has all the attributes of the class Tree as well as the dropLeaves attribute. On the other hand, for an instance of a Coniferous Tree, all of the attributes of the Tree class are inherited besides the reseed attribute which is overwritten by the reseed attribute of the Coniferous Tree. The visualization of this example can be seen in Figure 9 below.

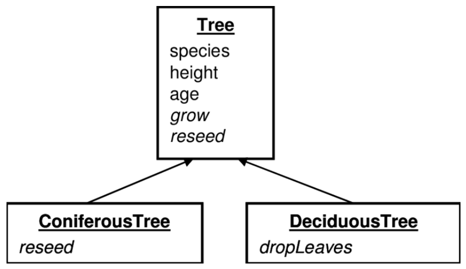


Figure 9: Inheritance in OOP [Parker et al., 2001]

2.1.2 Environment

All the processes run in the PyCh environment which is a slightly modified version of the SimPy environment. Besides initialization of an environment to work with, its primary role is to initiate communication channels which are used throughout to send and receive objects between the various system components.

2.1.3 Product and Resource Class

A Product class is initialized to represent the products that move through the system. Among the properties to be assigned to the Product class, since the scope of this experiment is limited only to the overfill which is dependent on the weight of the product, only the weight property is initialized to this class.

The Resource class is initialized to represent all the individual system components that are linked to each other by channels. The input and output channels are assigned to each instance of the Resource class and an arrival process is defined. The arrival process enables an instance of a Resource to receive a product.

The entire system is modelled in a resource library and the instances of the Resource class defined are Fillet Generator, Conveyor, Batcher, Tray and Sink.

The Fillet Generator Class generates the fillet object for the system and defines the weight of each object generated. This defined weight is then assigned to an instance of the Product class and the product is sent out using the output channel

The Conveyor class has the parameters to tweak the length and speed of the conveyor and just performs the role of receiving and sending out products. While these parameters do not influence the model due to the travel time of objects being considered as instantaneous, they leave room for future incorporation of a time-based system.

The Discharge component of the system is represented by the Batcher class and this is where the allocation algorithm is defined. Besides the algorithm, the trays as well as the performance metrics of the overfill are stored in this class.

Trays are Resources that hold a batch of products. The Tray class has three parameters defined; the cumulative weight of the products held in it, the target weight threshold which if exceeded prompts the algorithm to send out the tray, and the list of products in the tray.

Finally, the Sink class receives the trays sent out of the Batcher and stores the number of trays it receives. The flow of the entire system function can be visualized in Figure 10 below.

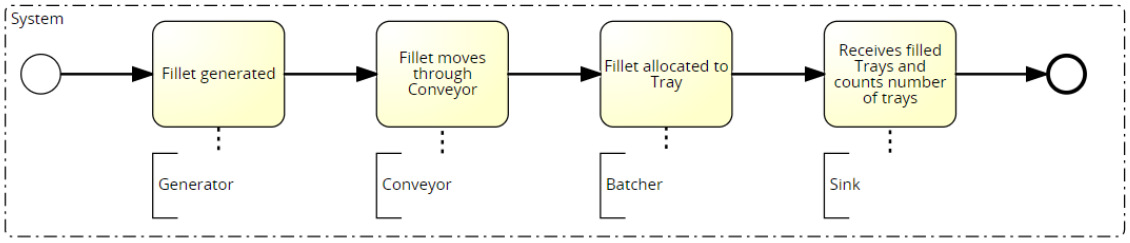

System
Fillet generated
Fillet moves through Conveyor
Fillet allocated to Tray
Receives filled Trays and counts number of trays
Generator
Conveyor
Batcher
Sink

Figure 10: Working of the entire system

2.2 Statistics

The comparison and validation of the results will be done using statistical tests and it is thereby important to introduce the statistical concepts that are used in the analysis of the results.

2.2.1 Confidence Intervals and Ideal Sample Size

In statistics, a confidence interval is an estimated range of likely values for a population parameter. Taking the commonly used 95% confidence level as an example, if the same population were sampled multiple times, and interval estimates made on each occasion, in approximately 95% of the cases, the true population parameter would be contained within the interval [SSC, 2022]. Note that the 95% probability refers to the reliability of the estimation procedure and not to a specific interval. A visualization of the same can be seen in Figure 11 below.

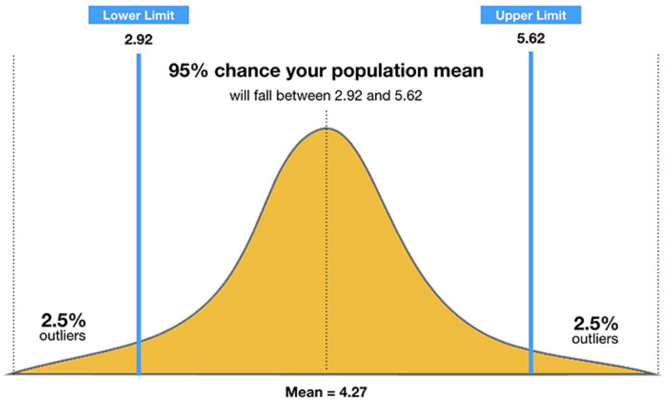


Figure 11: Confidence Interval [Mcleod, 2021]

Sample size is a statistical concept that involves determining the number of observations or replicates that should be included in a statistical sample [SSC, 2022]. It is an important aspect of any empirical study requiring that inferences be made about a population based on a sample. The equation to calculate the sample size (n) for a population is as follows.

$$n = \frac{z^2 * p(1-p)}{\epsilon^2} \tag{1}$$

Where: z - z score, ϵ - margin of error, p - population proportion

A sample size of 30 is fairly common across statistics sometimes referred to as the magic number [Sutiarso, 2022]. A sample size of 30 often increases the confidence interval of the population data set enough to warrant assertions against findings. The higher the sample size, the more likely the sample will be representative of the population set [Ganti, 2022].

2.2.2 Analysis of Variance Test

The Analysis of Variance Test or ANOVA is used to compare differences of means among more than two groups which is essential to the particular case discussed in this paper. This is due to the fact that more than two algorithms are being simultaneously compared to each other which needs a statistical test such as the ANOVA test that is capable of doing so. Other tests such as the commonly used t-test are not suitable for this case study since it compares the means of up to two groups. ANOVA compares the amount of variation between the means of different groups with the amount of variation within the different groups and this can be seen in Figure 12 below [ELL, 2022]. Like other classical statistical tests, we use ANOVA to calculate a test statistic (the F-ratio) with which we can obtain the probability (the P-value) of obtaining the data assuming the null hypothesis. A significant P-value (usually taken as $P < 0.05$) suggests that at least one group's mean is significantly different from the others.

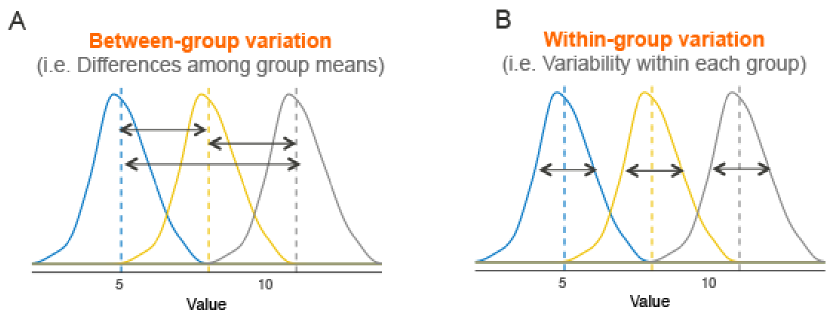


Figure 12: Variation between and within groups visualized [LSSG, 2022]

The definition of hypothesis should be clarified. A hypothesis is an educated guess about something and should be testable either by experiment or observation [Singh, 2018]. There are two hypotheses namely the null hypothesis and the alternate hypothesis. For the case of ANOVA, these are as follows.

- Null hypothesis - All population means are equal
- Alternative hypothesis - At least one population mean is different from the rest

An important distinction to be made here is that should the alternate hypothesis hold, it only indicates that one mean is different from the rest. It does not give any information as to which groups of means are different or even quantitative metrics such as by how much the mean of a particular group differs from the rest. There are two types of ANOVA tests as follows.

- A one-way ANOVA is a type of statistical test that compares the variance in the group means within a sample whilst considering only one independent variable or factor [Mackenzie, 2021].
- A two-way ANOVA is, as a one-way ANOVA, a hypothesis-based test. However, in the two-way ANOVA each sample is defined in two ways, and resultingly put into two categorical groups [Mackenzie, 2021].

A one-way ANOVA is primarily designed to enable equality testing between three or more means. A two-way ANOVA is designed to assess the interrelationship of two independent variables on a dependent variable. Within the scope of this research, since only a single performance metric namely the overfill is under investigation, the **one-way ANOVA** test is the one that is suitable to be used.

2.2.3 Tukey's Test

Since the ANOVA test does not give any additional information should the alternative hypothesis hold, it becomes essential to run a posthoc test to determine which groups have a difference in means. A Tukey's test is one such posthoc test. Tukey's test or Tukey's HSD (honest significant difference) test is a single-step multiple comparison procedure and statistical test [Passel, 2022]. Tukey's test compares the means of every group to the means of every other group; that is, it applies simultaneously to the set of all pairwise comparisons: $U_i - U_j$ and identifies any difference between two means that is greater than the expected standard error. **It works well when the sample sizes are equal and is quite conservative for unequal sample sizes.**

2.3 Experimental Plan

1. For each algorithm, the average overfill is calculated and recorded. This average should be the mean of a specific number of trays that is large enough to guarantee a reasonable estimate. Using Equation 1, in case we want to ascertain with a 90% confidence level that a given measurement lies $\pm$ 2% of its true value, we need around 1700 observations. This can be easily achieved by having the simulation run until around 1700 trays have reached the Sink.

2. This process is repeated 30 times for each algorithm since that is considered a large enough sample size for the ANOVA test.

3. Once complete, the confidence intervals of all the algorithms are plotted and the ANOVA test is conducted using Python. Should the P-value be more than 0.05, it can be ascertained that the efficacy of the algorithms is similar with no algorithm clearly performing better than the rest.

4. Should the P-value be less than 0.05, a Tukey's test should be run using Python to ascertain the statistically significant difference between the groups and determine the algorithm that results in the least average overfill.

3 Algorithms

The allocation algorithms used in the Discharge are mentioned and elaborated further below. From the illustration in Figure 6, the only process that changes between the different algorithms is the manner in which the tray number is allocated. Hence the figures in this section purely focus on the allocation of the tray number for a fillet

3.1 Random Allocation Algorithm

This is the algorithm that facilitates the baseline for comparison. While the name states that it is a random allocation algorithm, by principle, the algorithm is random allocation on a uniform distribution. This means that every tray has an equal chance of receiving the current fillet similar to how every number on a dice has an equal probability of showing up on a particular role. Hence an important distinction to make to avoid being misled by the name is that the **probability of allocation to each tray is equal and not random**.

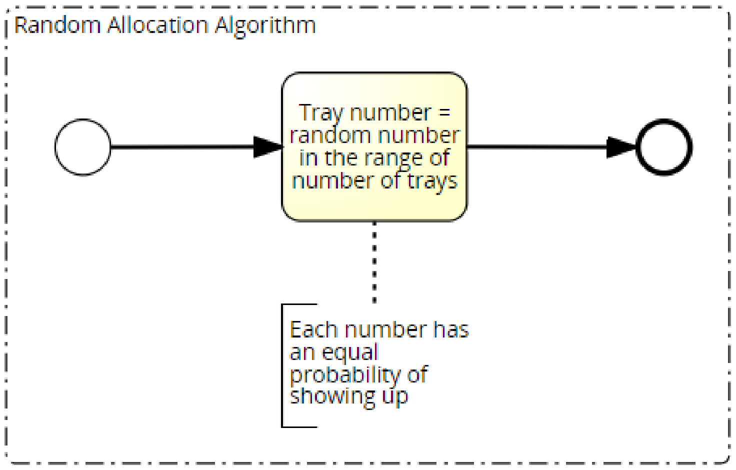

Random Allocation Algorithm
Tray number = random number in the range of number of trays
Each number has an equal probability of showing up

Figure 13: Random Allocation Algorithm

3.2 Worst Fit Algorithm

Worst Fit allocates a process to the partition which is the largest sufficient among the freely available partitions [GeekForGeeks, 2022]. In other words, the fillet is allocated to the tray which is most empty or least filled. There are two intuitive downsides to this algorithm that does not affect the metrics aspect of this research but more from a physical standpoint. Firstly, should a tray be filled just under the threshold (for instance a filled weight of 999.9 for a threshold weight of 1000), then it is unlikely that throughout the simulation that the particular tray would ever have the lowest weight. This effectively makes that tray redundant in the whole process by not being able to receive further fillets and be sent out. The second drawback is that once a tray exceeds that particular threshold and is sent out, the empty tray that takes its place automatically has the lowest weight and single-handedly receives the next couple of fillets until the weight exceeds that of one of the other trays but stays under the threshold weight (else it is sent again and the process repeats). In both cases it leads to the underutilization of the system since most of the components of the system are on standby. Remaining on standby in the case of the meat industry also introduces the risk of spoilage of meat from a physical standpoint. Nevertheless, the performance metrics would be an interesting aspect to investigate and if successful, then adaptations in the physical environment could be considered.

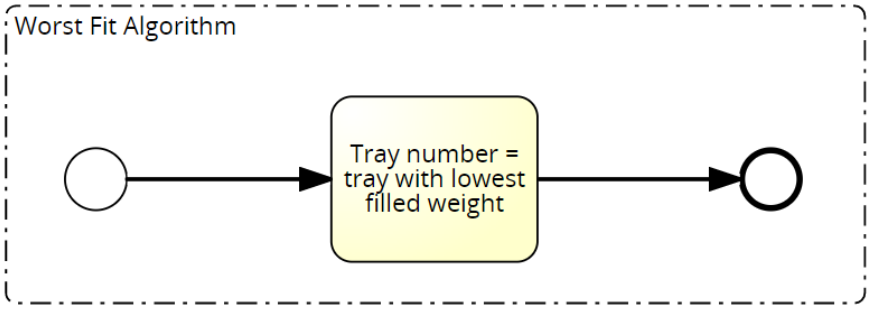


Figure 14: Worst Fit Algorithm

Custom Algorithms

The remainder of the algorithms discussed is individual work and while it draws some inspirations from the principles of certain algorithms found in literature, to the best extent of research conducted there have been no similar algorithms used in literature. [Assmann et al., 1984] mentions that efficient optimization algorithms are unlikely to exist for such an NP-hard problem and [Jabari et al., 2016] states that while heuristic (or custom) algorithms may not yield a globally optimal solution (or the best possible solution), it may yield locally optimal solutions that are 'good enough' for the task at hand within a reasonable time frame which is why further attempts were made in this direction.

3.3 Custom Algorithm 1

This algorithm takes inspiration from the work of Berndt et al., 2019 where separate approaches are taken for the allocation of 'small' and 'big' items. Since the distribution of the weights of the fillets follows a Normal (Gaussian) Distribution, the allocation algorithm was based on the principle of the Normal (Gaussian) Distribution.

Gaussian Distribution (also known as Normal Distribution) is a bell-shaped curve, and it is assumed that during any measurement, values will follow a normal distribution with an equal number of measurements above and below the mean value [Bruce and Lapsley, 2014]. Other characteristics of Gaussian distributions are as follows.

- 68.2% of all values are contained within 1 standard deviation from the mean.
- 95.5% of all values are contained within 2 standard deviations from the mean.
- 99.7% of all values are contained within 3 standard deviations from the mean.

A visual representation of the same can be seen in Figure 15 below

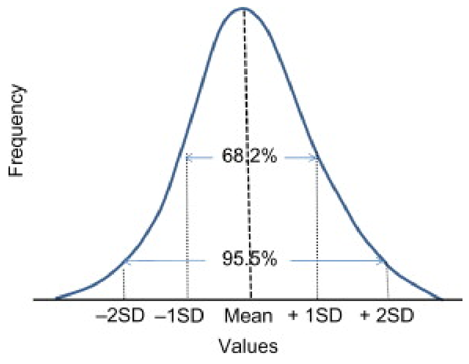


Figure 15: Gaussian (Normal) Distribution

3.3.1 Variant 1

Given the mean of the weight distribution of the fillets is 200 grams with a standard deviation of 20 grams, the first variant of the custom algorithm is as follows:

- All the items below 180 grams (below -1 SD) are allocated to the tray with the highest weight
- All the items between 180 and 200 grams (between -1 SD and mean) are allocated to the tray with the second-highest weight
- All the items between 200 and 220 grams (between mean and 1 SD) are allocated to the tray with the second-lowest weight
- All the items above 220 grams (above 1 SD) are allocated to the tray with the lowest weight

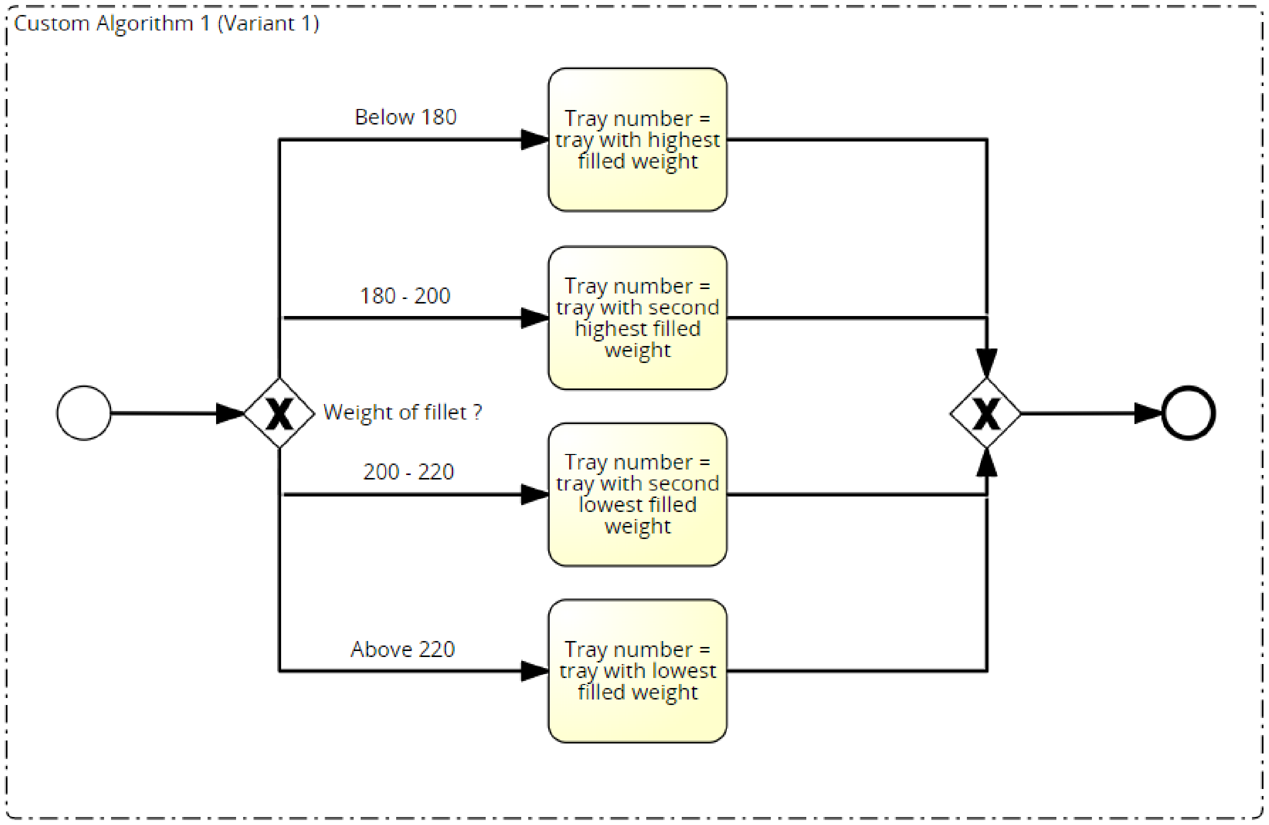

Custom Algorithm 1 (Variant 1)
Below 180
Tray number = tray with highest filled weight
180 - 200
Tray number = tray with second highest filled weight
Weight of fillet ?
200 - 220
Tray number = tray with second lowest filled weight
Above 220
Tray number = tray with lowest filled weight

Figure 16: Custom Algorithm 1 (Variant 1)

3.3.2 Variant 2

The principle remains the same but with a slight variation:

- All the items below 200 grams (below mean) are allocated to the tray with the highest weight
- All the items above 200 grams (above mean) are allocated to the tray with the lowest weight

The rationale behind both the variants and the algorithm is to minimize the overfill caused by allocating the heavier incoming items to the lighter filled trays and the lighter incoming items to the heavier filled trays. If a heavy item goes to a heavy tray that is just under the threshold, the overfill experienced is extremely large causing a lot of overfill. On the other hand, if a light item goes to a lighter tray, there runs the chance that the tray might be filled just under the required threshold leaving it prone to the aforementioned scenario to take place.

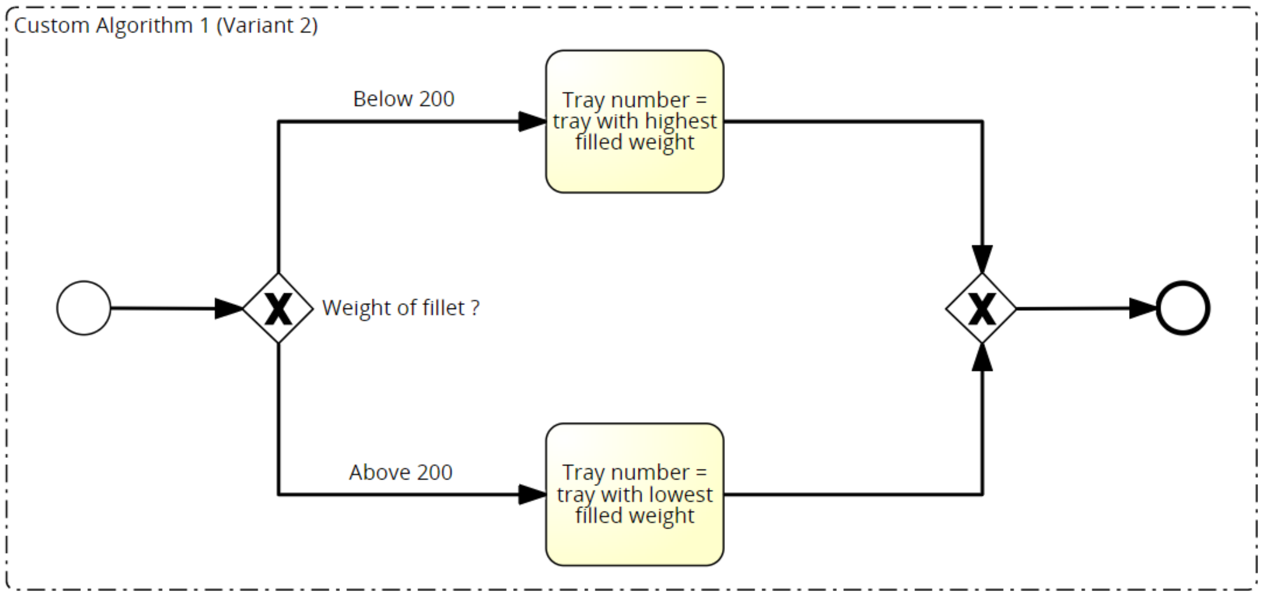

Custom Algorithm 1 (Variant 2)
Below 200
Tray number = tray with highest filled weight
Weight of fillet ?
Above 200
Tray number = tray with lowest filled weight

Figure 17: Custom Algorithm 1 (Variant 2)

3.4 Custom Algorithm 2

Following the previous algorithm, it is evident that it is most beneficial to avoid a scenario where a heavy item is placed in a tray that is almost near the threshold weight. Hence given the parameters, an algorithm was created such that a tray number is randomly generated (with a uniform distribution) and if the placing of an item on a tray with that tray number resulted in a filled weight that is between 0.85 and 1 times the threshold weight (or between 850 and 1000 grams), a new tray number is generated and the process is repeated (in order words, that particular tray is avoided). If none of the trays are able to receive the incoming fillet, a tray is randomly chosen. The factor 0.85 has been arbitrarily chosen and can be tweaked and optimized should the algorithm yield promising results.

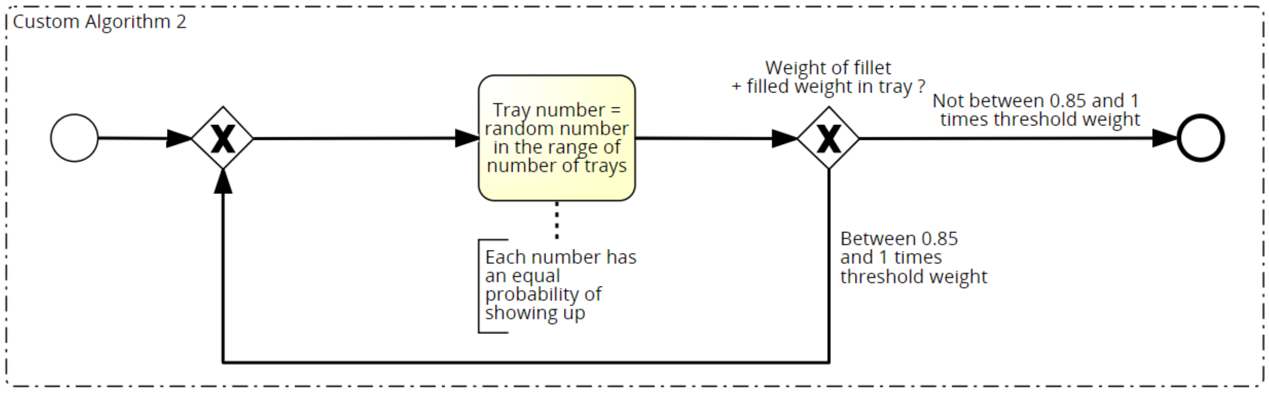

Custom Algorithm 2
Tray number = random number in the range of number of trays
Each number has an equal probability of showing up
Weight of fillet + filled weight in tray ?
Not between 0.85 and 1 times threshold weight
Between 0.85 and 1 times threshold weight

Figure 18: Custom Algorithm 2

3.5 Custom Algorithm 3

This algorithm aims at trying to restrict the overfill to a predetermined 'acceptable level' as much as possible. For the sake of the experiment, this level is set to 5% and just like the previous algorithm, this can be tweaked should the algorithm yield promising results. In essence, how the algorithm works is that should the weight of a tray after placing an item in it be between 1 and 1.05 times the threshold weight (or between 1000 and 1050 grams), the fillet is allocated to that tray.

3.5.1 Variant 1

Following the above, should the condition not be met then the fillet is randomly allocated (with equal probability) to any of the remaining trays.

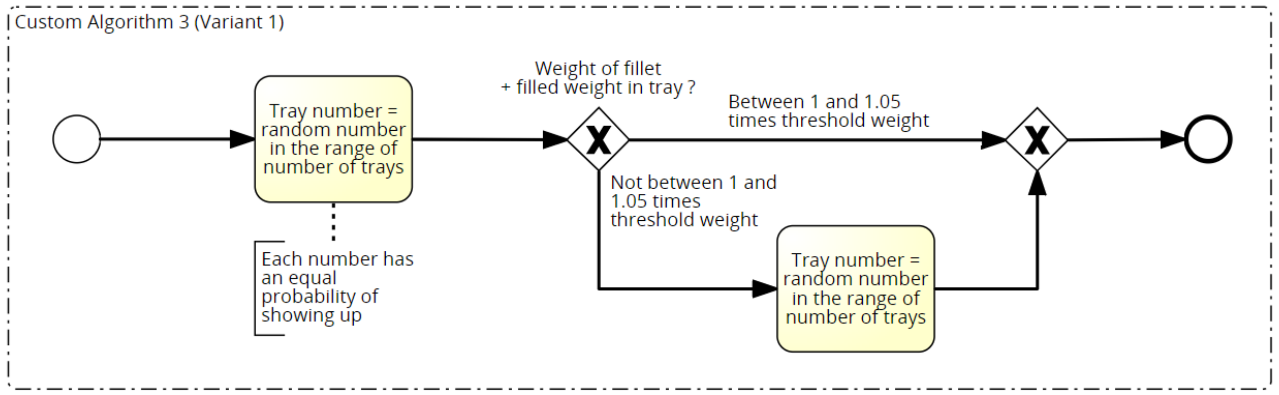

Custom Algorithm 3 (Variant 1)
Tray number = random number in the range of number of trays
Each number has an equal probability of showing up
Weight of fillet + filled weight in tray ?
Between 1 and 1.05 times threshold weight
Not between 1 and 1.05 times threshold weight
Tray number = random number in the range of number of trays

Figure 19: Custom Algorithm 3 (Variant 1)

3.5.2 Variant 2

In this variant, should the condition not be met then the fillet is allocated to the tray with the least weight.

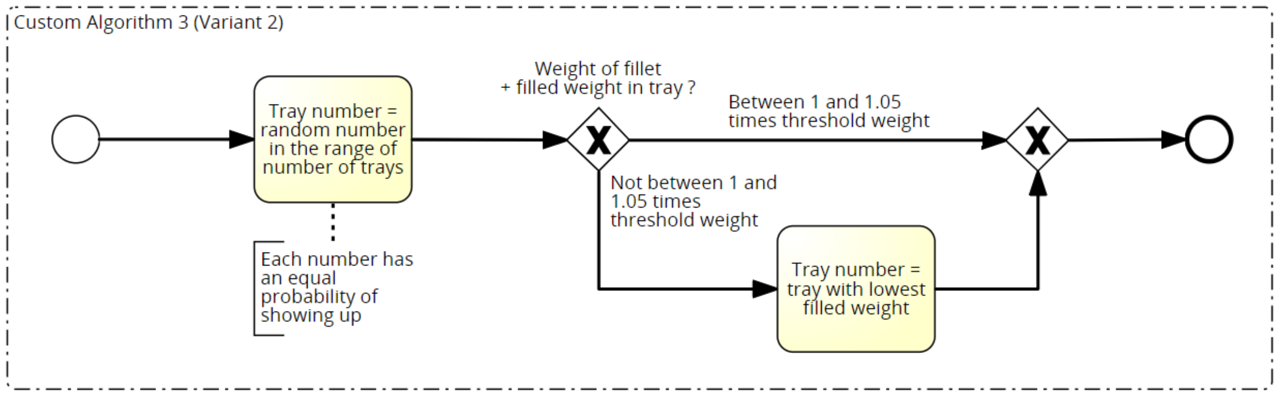

Custom Algorithm 3 (Variant 2)
Tray number = random number in the range of number of trays
Each number has an equal probability of showing up
Weight of fillet + filled weight in tray ?
Between 1 and 1.05 times threshold weight
Not between 1 and 1.05 times threshold weight
Tray number = tray with lowest filled weight

Figure 20: Custom Algorithm 3 (Variant 2)

3.6 Custom Algorithm 4

This algorithm aims to improve upon the drawback of the Worst Fit Algorithm. The challenge of the potential standby of items after a while of allocation was a primary bottleneck to the algorithm. This can best be demonstrated with an example. After a given amount of time has passed using the Worst Fit Algorithm for allocation (in Figure 21, this instance has been assumed as the point of time where 15 trays have been sent), the weights of the 5 trays could look something like 995, 997, 990, 992 and 940. Now, this leads to the scenario that if most trays receive another fillet, then a large overfill is experienced. Hence this algorithm prescribes that the moment a tray exceeds the threshold, all the other trays are sent out as well. This timer for the use of the Worst Fit Algorithm resets after this process has been done. This algorithm however leads to some trays being sent out before reaching the threshold weight and even if we consider only the ones that do exceed the threshold weight, the bias of allowing for trays not to be fully filled changes the fundamental condition of a guaranteed minimum weight altogether and therefore removes the ability to directly compare the algorithms to each other. **Hence, the result of this algorithm will not be discussed in the following section**. Since the overall working of the Discharge slightly differs for this algorithm as opposed to the rest, the entire Discharge has been modelled in Figure 21 below.

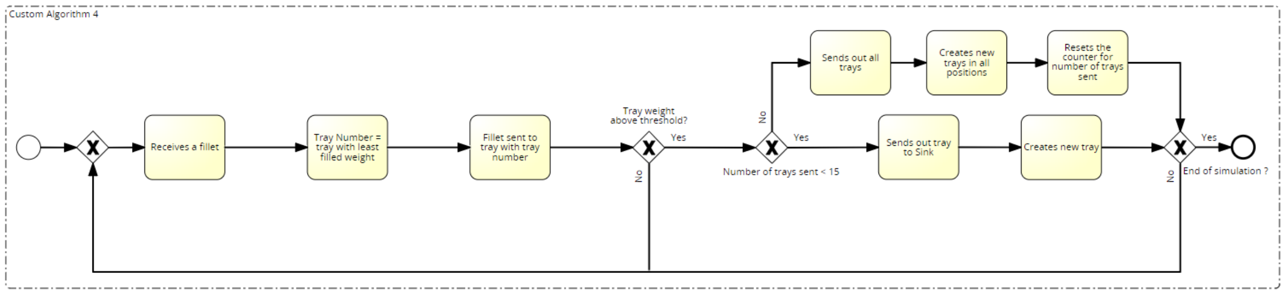

Custom Algorithm 4
Receives a fillet
Tray Number = tray with least filled weight
Fillet sent to tray with tray number
Tray weight above threshold?
Yes
No
Number of trays sent < 15
No
Sends out all trays
Creates new trays in all positions
Resets the counter for number of trays sent
Yes
Sends out tray to Sink
Creates new tray
Yes
No
End of simulation ?

Figure 21: Custom Algorithm 4

4 Results and Analysis

The results of the average overfill percentage by algorithm are summarized in the table below.

Reading No.	RAA	WFA	CA1_1	CA1_2	CA2	CA3_1	CA3_2
1	9.92	10.49	9.69	10.07	4.8	9.34	10.01
2	10.4	10	10.21	10.11	4.83	9.68	9.82
3	10.05	10.19	9.97	9.77	4.8	9.67	9.5
4	10	10.25	10	9.74	4.6	9.43	10.24
5	10.04	9.67	9.81	9.9	4.69	9.28	10
6	9.63	10.28	9.87	9.97	4.93	9.63	9.99
7	10.27	9.61	10.11	9.88	4.73	9.62	9.96
8	9.9	9.91	9.65	9.76	5.15	9.69	9.9
9	10.12	9.65	9.76	9.76	4.86	9.85	10
10	10.34	10.11	10.23	9.93	4.87	9.76	10.12
11	9.88	9.98	10.05	10.1	4.75	9.62	9.77
12	9.9	9.94	9.97	9.99	4.37	9.54	9.52
13	10.1	10.08	9.97	10.04	4.93	9.45	9.9
14	9.73	9.53	9.99	9.41	4.63	9.72	9.58
15	10.03	10.24	10.2	9.81	4.72	10.03	9.76
16	10.18	9.97	10.11	10.25	4.85	9.31	10.05
17	10.39	9.81	9.26	10.13	5.16	9.41	9.95
18	9.99	9.97	9.88	10.03	4.89	9.9	10.17
19	10.18	10	10.12	10.48	4.6	9.34	9.83
20	9.86	9.66	10.24	9.68	4.84	9.83	9.84
21	10.41	10.13	10.15	9.96	4.74	9.82	10.18
22	10.36	9.9	9.77	10.11	4.76	9.82	9.93
23	9.89	9.88	9.55	9.94	4.47	9.4	10.16
24	10.17	9.84	9.88	10.13	4.92	9.48	10.04
25	9.69	10.02	10.13	10.18	4.81	9.44	9.85
26	9.77	9.91	10.29	9.84	4.92	9.77	9.95
27	10.32	9.66	9.5	9.7	5.16	9.86	10.16
28	10.08	10.22	10.04	9.39	4.77	9.79	9.86
29	10.09	9.91	9.65	9.8	4.98	9.45	9.91
30	9.93	10.08	10.18	9.99	4.8	9.73	10.11

Key:-

- RAA - Random Allocation Algorithm
- WFA - Worst Fit Algorithm
- CA1_1 - Custom Algorithm 1 (Variant 1)
- CA1_2 - Custom Algorithm 1 (Variant 2)
- CA2 - Custom Algorithm 2
- CA3_1 - Custom Algorithm 3 (Variant 1)
- CA3_2 - Custom Algorithm 3 (Variant 2)

The means of the average overfill percentage along with their 95% confidence intervals indicated by the gray lines are shown in the Figure 22 below.

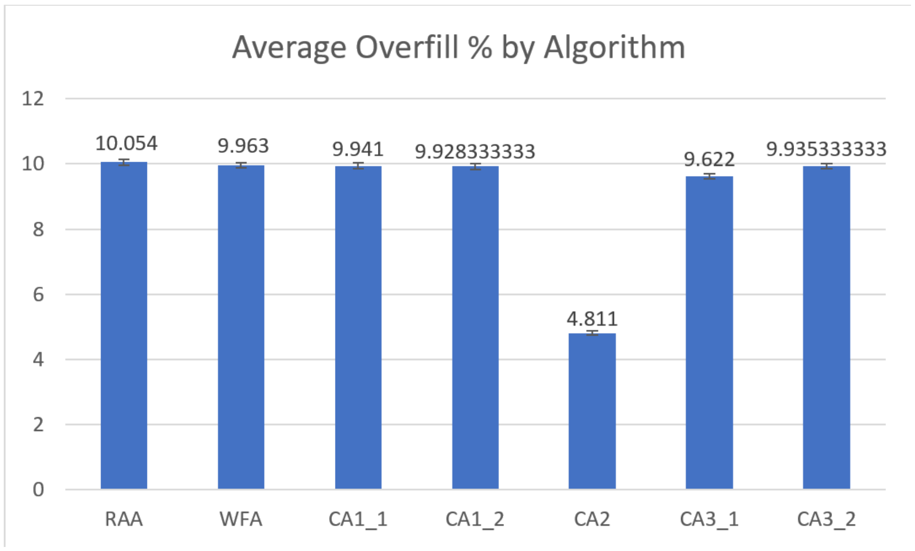


Figure 22: Average Overfill% by Algorithm

The top of the bar graphs are the means of the respective algorithm and the small grey lines that appear like dashes are the 95% confidence intervals. At first glance, it is clearly evident that **Custom Algorithm 2** greatly outperforms the other algorithms. However, before validating this statistically and fine-tuning the algorithm, a comparison will be made between the other algorithms and check if any of them have a non-overlapping confidence interval and thereby a better algorithm.

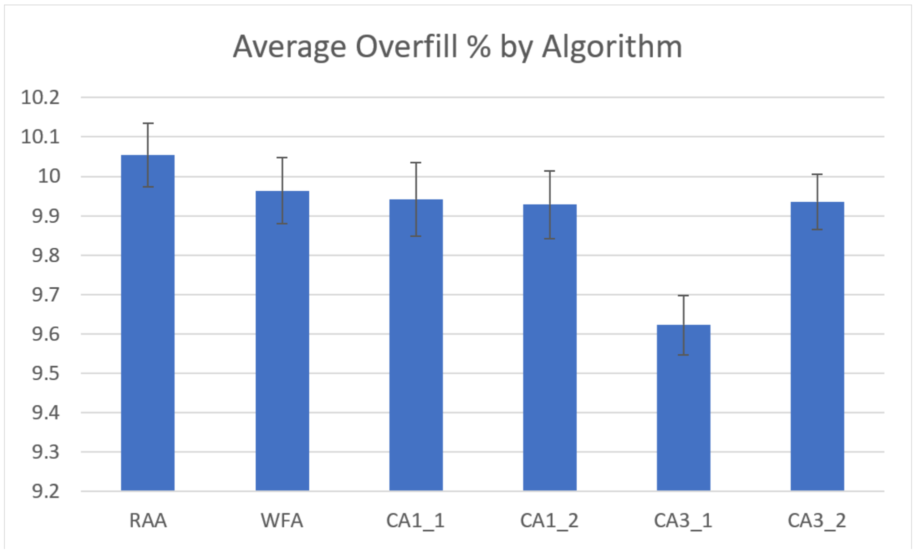


Figure 23: Average Overfill% by Algorithm without Custom Algorithm 2

From Figure 23, it can be seen that among the remaining algorithms, the first variant of Custom Algorithm 3 seems to perform better than the remaining with a high confidence level. Before the statistical tests are conducted, it is important to discuss the effects of changing the bin size and number of bins.

4.0.1 Effect of changing Bin Size

After experimenting with all the algorithms, it was found that all the algorithms performed better as the bin size or the maximum threshold weight was raised. This also coincides with intuition since the higher the threshold weight, the more the number of fillets that can be allocated which allows for more room for optimization. Hence, a larger bin size leads to lower overfill given the same fillet mean weight. On the contrary, this is also the same reason why lowering the threshold weight leads to worse performance. The explanation above can be visualized by taking the example of the Random Allocation Algorithm and the second variant of Custom Algorithm 3 and plotting its average overfill as a function of bin size seen in Figure 24 and Figure 25 below.

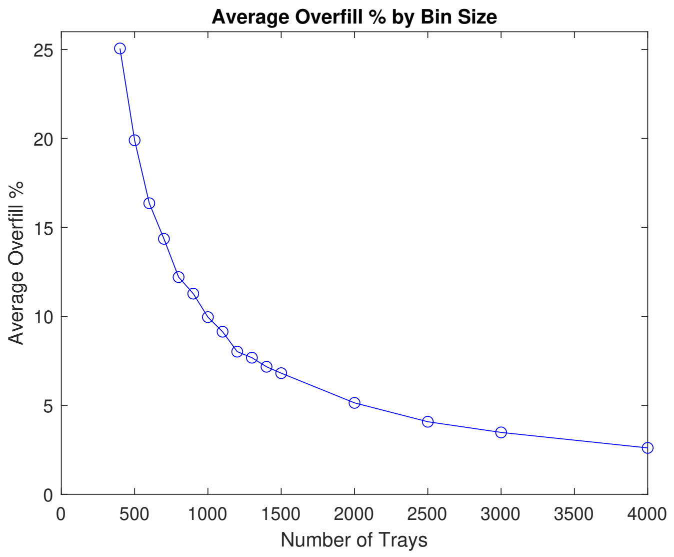


Figure 24: Average Overfill% by Bin Size for the Random Allocation Algorithm

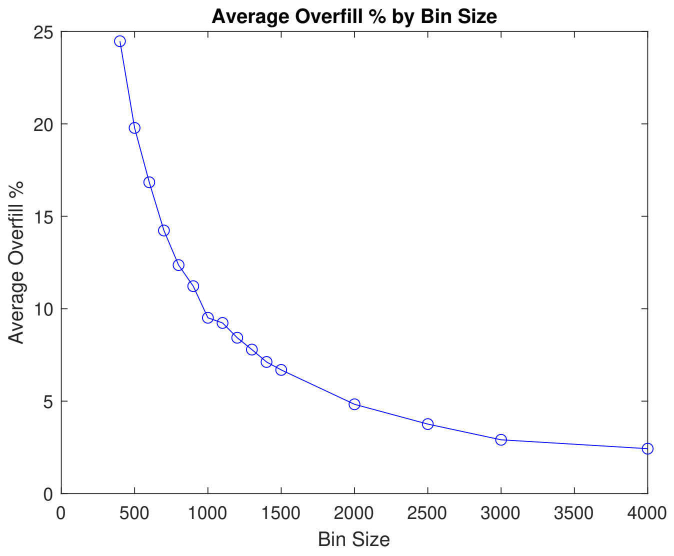


Figure 25: Average Overfill% by Bin Size for Custom Algorithm 3 (Variant 2)

As seen in the figures, the effect of increasing the bin size does have a positive influence on the performance of the algorithm however this additional benefit decreases in an exponential manner as the bin size increases.

4.0.2 Effect of changing Number of Trays

After running simulations with varying the number of trays in the system, no significant differences were found between the variations, while at first glance it seemed like increasing the number of trays improved the performance of the algorithm, it was found that by increasing the simulation time, the value of the average overfill of the system with a larger number of trays eventually converged to similar values of those with a smaller number of trays. The explanation can be shown by taking the example of the Random Allocation Algorithm and the first variant of Custom Algorithm 1 and the results can be seen in Figure 26 and Figure 27 below.

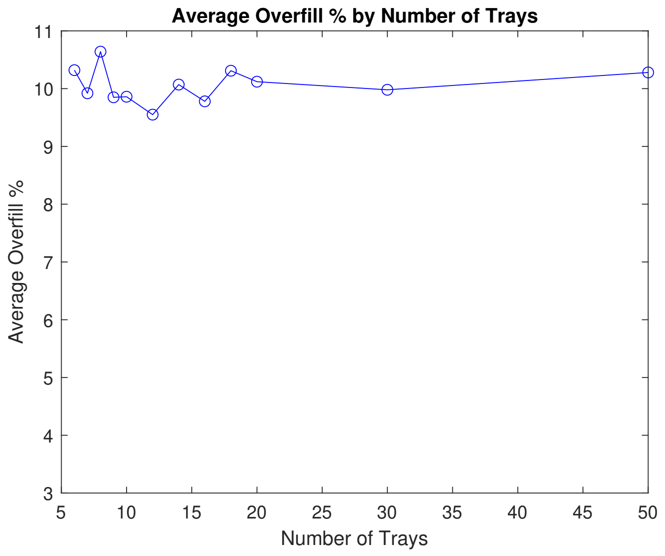


Figure 26: Average Overfill% by Number of Trays for the Random Allocation Algorithm

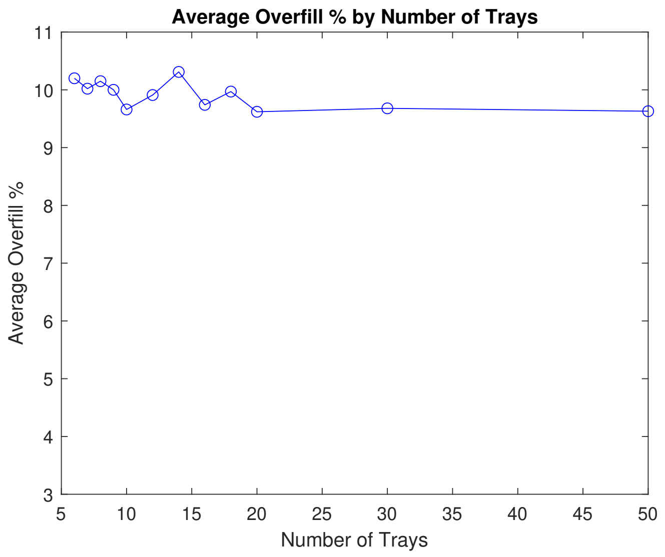


Figure 27: Average Overfill% by Number of Trays for Custom Algorithm 1 (Variant 1)

While there are peaks and through observed in the results, these simply arise due to the variation involved in stochastic simulations and not due to the influence of the induced variation. After investigating the website of Marel, it was found that graders often come in 6 or 8 bin sizes [Marel, 2022] and hence 8 bins is chosen due to the ability to process more volume (even though this does not affect this particular research question).

4.1 ANOVA Test

In order to statistically validate the results of the eye test, a one-way ANOVA test was conducted and the following results were yielded.

- F-statistic - 2425.89
- P-value - 5.837 e-186

The F-statistic is the ratio of two mean square values [Zar, 2010]. If the null hypothesis is true, F has a value that is close to 1. A large F ratio means that the variation among group means is more than that would normally occur by chance. With an F value much larger than 1 and a P-value much lower than 0.05, it can safely be concluded that there exists a statistically significant difference between the means of the various algorithms as expected.

4.1.1 Tukey's test

In order to find out which groups are statistically different in terms of their means, a Tukey test was and the results can be seen in Figure 28 below.

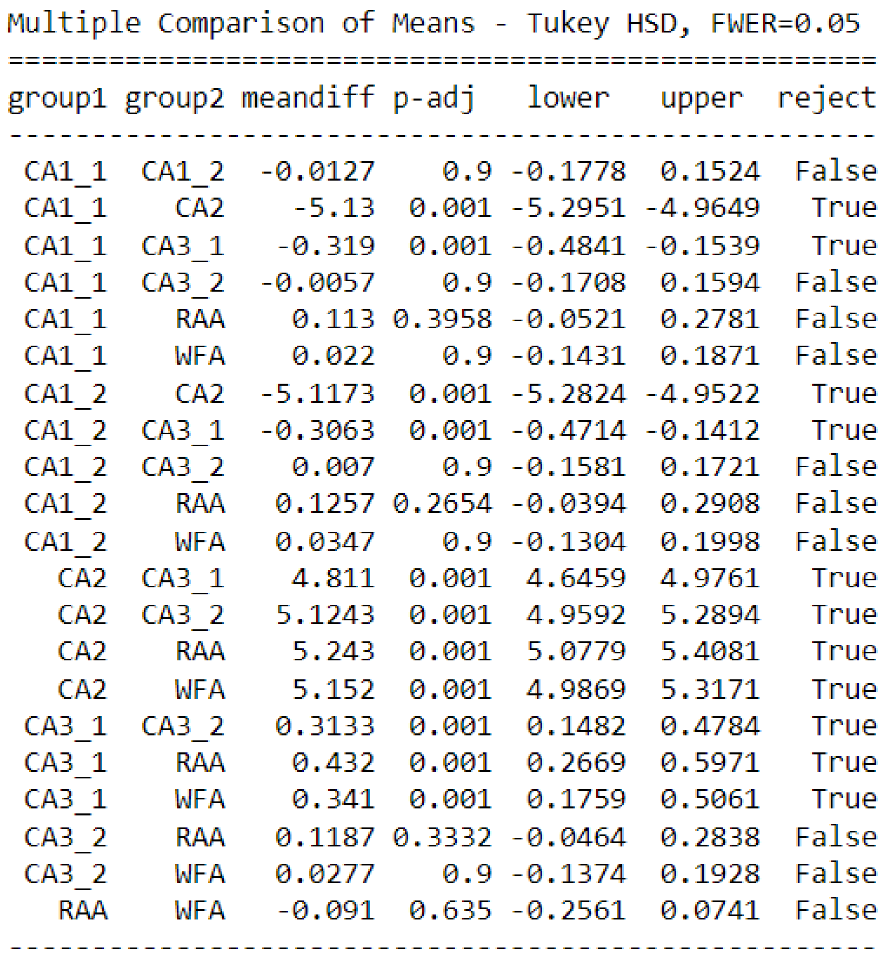

```
Multiple Comparison of Means - Tukey HSD, FWER=0.05
===================================================
group1 group2 meandiff p-adj   lower   upper  reject
---------------------------------------------------
 CA1_1  CA1_2  -0.0127    0.9 -0.1778  0.1524  False
 CA1_1    CA2    -5.13  0.001 -5.2951 -4.9649   True
 CA1_1  CA3_1   -0.319  0.001 -0.4841 -0.1539   True
 CA1_1  CA3_2  -0.0057    0.9 -0.1708  0.1594  False
 CA1_1    RAA    0.113 0.3958 -0.0521  0.2781  False
 CA1_1    WFA    0.022    0.9 -0.1431  0.1871  False
 CA1_2    CA2  -5.1173  0.001 -5.2824 -4.9522   True
 CA1_2  CA3_1  -0.3063  0.001 -0.4714 -0.1412   True
 CA1_2  CA3_2    0.007    0.9 -0.1581  0.1721  False
 CA1_2    RAA   0.1257 0.2654 -0.0394  0.2908  False
 CA1_2    WFA   0.0347    0.9 -0.1304  0.1998  False
   CA2  CA3_1    4.811  0.001  4.6459  4.9761   True
   CA2  CA3_2   5.1243  0.001  4.9592  5.2894   True
   CA2    RAA    5.243  0.001  5.0779  5.4081   True
   CA2    WFA    5.152  0.001  4.9869  5.3171   True
 CA3_1  CA3_2   0.3133  0.001  0.1482  0.4784   True
 CA3_1    RAA    0.432  0.001  0.2669  0.5971   True
 CA3_1    WFA    0.341  0.001  0.1759  0.5061   True
 CA3_2    RAA   0.1187 0.3332 -0.0464  0.2838  False
 CA3_2    WFA   0.0277    0.9 -0.1374  0.1928  False
   RAA    WFA   -0.091  0.635 -0.2561  0.0741  False
---------------------------------------------------
```

Figure 28: Results of Tukey test

The most important metrics in this table are the p values given by p-adj and based on it the reject column which states whether a pair of groups are significantly statistically different. A 'True' value indicates that a pair of groups are significantly statistically different from each other. As initially expected with the eye test, Custom Algorithm 2 and the first variant of Custom Algorithm 3 are significantly statistically different from the rest of the groups. Since Custom Algorithm 2 largely outperformed the other algorithms, the rest of the research will be focused on optimizing its parameters.

4.2 Optimization of Custom Algorithm 2

To begin testing with this algorithm, an arbitrary value of 0.85 was chosen. However, attempts will be made in order to optimize this value. In order to figure out whether to increase or decrease this value, a sample of 3 results of the average overfill from different levels were taken and their values are as follows.

- 0.75 - 6.68, 6.66, 6.95
- 0.8 - 5.11, 5.01, 5.38
- 0.85 - 4.74, 4.66, 4.97
- 0.9 - 4.29, 4.33, 4.00
- 0.95 - 5.94, 5.44,5.86

From the above results, it is evident that slightly increasing the value from 0.85 to around 0.9 gives better results, however, the performance significantly drops again when we increase it to about 0.95. Hence the values 0.86,0.88,0.9 and 0.92 will be compared to find the threshold that yields the best performance. The results of the means can be seen in Figure 29 below. The decimal point has been replaced with an underscore in the visuals and subsequent results.

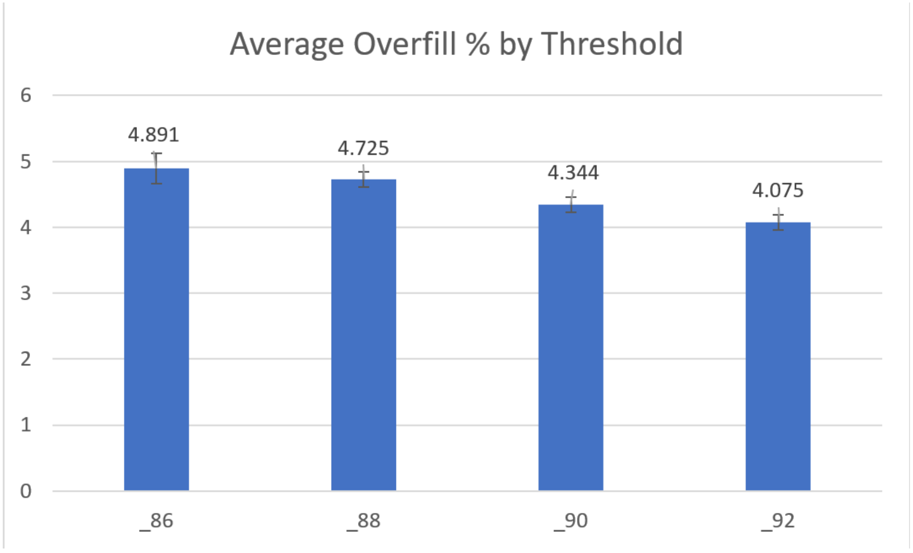
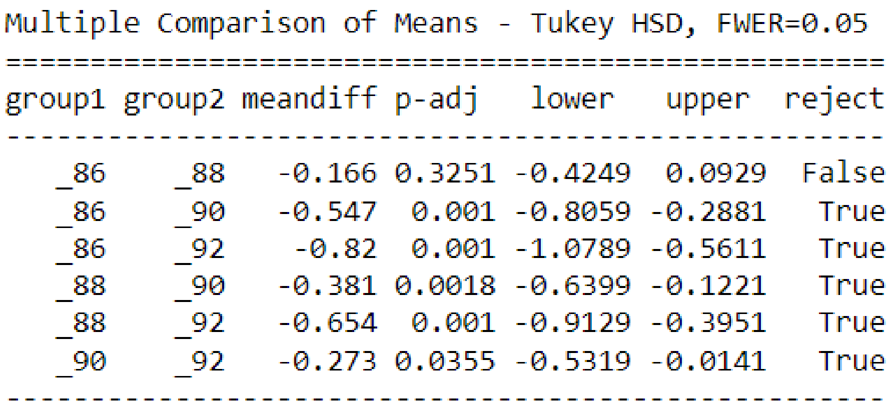


Figure 29: Average Overfill% by Threshold Value

It can be observed that while fairly similar, the threshold of 0.92 yielded the best value in terms of the mean of the average overfill. The same was done with values of 0.93 and 0.94 as well however these yielded worse performance with a higher average mean overfill. Just like in the previous case, an ANOVA test was conducted to check if any of the results were significantly statistically different from each other and the following were the results.

- F-statistic - 29.69
- P-value - 7.65 e-10

Since the P-value is below 0.05, it can be ascertained that the means of the different groups were significantly statistically different from each other. In order to investigate the differences between specific groups, a Tukey test was performed and the results can be seen in Figure 30 below.



```
Multiple Comparison of Means - Tukey HSD, FWER=0.05
=====================================================
group1 group2 meandiff p-adj   lower   upper  reject
-----------------------------------------------------
   _86    _88   -0.166 0.3251 -0.4249  0.0929  False
   _86    _90   -0.547  0.001 -0.8059 -0.2881   True
   _86    _92    -0.82  0.001 -1.0789 -0.5611   True
   _88    _90   -0.381 0.0018 -0.6399 -0.1221   True
   _88    _92   -0.654  0.001 -0.9129 -0.3951   True
   _90    _92   -0.273 0.0355 -0.5319 -0.0141   True
-----------------------------------------------------
```

Figure 30: Results of Tukey test

Here it can be seen that there exists a significantly statistical difference between all the groups except between the results of the values 0.86 and 0.88. Using this information and the results seen in Figure 29, we can conclude that **Custom Algorithm 2 with a threshold value of around 0.92 is the most optimal batching algorithm among all the algorithms tested for the given experimental conditions.**

5 Conclusion

In the domain of manufacturing, every small percentage improvement of optimizing a system process saves a lot of time and capital, especially in sectors such as meat processing which deals with a large amount of volume of items processed. This paper compares the efficacy of seven different batching algorithms and through statistical and empirical testing, narrows them down to the most optimal algorithm. The parameters of this algorithm were then tuned in order to yield a more optimal performance of the algorithm. It is important to note that this is not the best possible algorithm regarding a global optimal solution for perfect packing. Rather, it was the best outcome amidst the different algorithms tested in this particular research undertaken.

Hence for the experimental conditions taken, the answer to the initially proposed research question as to which algorithm is most suitable to minimize overfill given a guaranteed minimum weight is *Custom Algorithm 2* amongst the algorithms tested. Regarding the question of whether simpler algorithms can be used to minimize overfill in comparison to more complex ones used in Asgeirsson, 2014, there is no definitive answer. While Custom Algorithm 2 is more computationally simple, the average overfill of about 4% is still higher than the 1.5-2% of overfill achieved by the algorithms defined by Asgeirsson. Hence owing to this trade off between computational efficiency and overfill, a decision regarding the choice of algorithm will have to be made based on the amount of computational resources available and the maximum permissible overfill.

Since it can be concluded from literature such as [Assmann et al., 1984] that there is unlikely to exist any efficient optimization algorithm for such a problem, the best possible solutions would probably be generated through the system learning by itself what the best possible way to allocate items would be given a set of input parameters. This is the domain of Reinforcement Learning (RL) and the implementation of RL algorithms could be a very interesting follow-up step to the work conducted in this paper.

In order to obtain the most realistic representation of the real-life system, these algorithms should be used in conjunction with more complex parameters such as the total flow time of the items in the system or the spoil rate of meat. RL algorithms could be used to tackle those research questions as well. A single algorithm that takes into account all the complex requirements prescribed and yields a solution that fulfils all of them in a satisfactory manner would be the ideal algorithm for use in a real-life system. This however goes well beyond the scope of the research conducted in this paper.

Regarding the choice of modelling software, PyCh was able to perform all the required tasks as initially intended in this research. Furthermore, it is also robust enough to carry out the proposed recommendations of continuing the research. Two foreseeable drawbacks of using PyCh however are as follows. Firstly, should there be an interruption during simulations, the entire simulation restarts as opposed to resuming from the point of interruption. While this may not be a huge drawback for stochastic experiments, for cases such as the off-line batching problem where the input is known beforehand, the option to resume simulations from the point of interruption would be a useful functionality to have. The second challenge is applicable to any Python-based software and not just PyCh in the fact that it is computationally quite inefficient and slow in comparison to some other programming languages such as C. This would in particular cause some challenges to the implementation of a highly accurate RL algorithm due to the large amount of computation involved in training RL models.

Regarding the simulation and validation process itself, each reading for each algorithm was recorded manually by rerunning simulations each time and once all the readings were taken, statistical tests to validate them were performed separately. Hence it would be more efficient to investigate ways to streamline and automate at least some parts if not the entire process to minimize the number of manual tasks performed.

Bibliography

(2022). Chi 3. https://cstweb.wtb.tue.nl/chi/trunk-r9682/. Accessed: 23-06-2022.

Angelopoulos, S., Kamali, S., and Shadkami, K. (2021). Online bin packing with predictions.

Asgeirsson, A. (2014). On-line algorithms for bin-covering problems with known item distributions. https://smartech.gatech.edu/handle/1853/53413.

Aspnes, J. (1998). Competitive analysis of distributed algorithms. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1442:118–146.

Assmann, S. F. (1983). Problems in discrete applied mathematics. https://dspace.mit.edu/handle/1721.1/121905?show=full.

Assmann, S. F., Johnson, D. S., Kleitman, D. J., and Leung, J. Y. (1984). On a dual version of the one-dimensional bin packing problem. *Journal of Algorithms*, 5:502–525.

Barrett, J., Jayaraman, B., Patel, D., and Skolnik, J. (2022). Discrete event simulation. https://www.med.upenn.edu/kmas/DES.htm. Accessed: 23-06-2022.

Berndt, S., Epstein, L., Jansen, K., Levin, A., Maack, M., and Rohwedder, L. (2019). Online bin covering with limited migration *.

Boyar, J., Favrholdt, L. M., Kamali, S., and Larsen, K. S. (2021). Online bin covering with advice. *Algorithmica*, 83:795–821.

Brown, K. P. (2022). Simulating real-life events in python with simpy — dattivo software inc. — custom software development, systems integration, research and analytics in london, ontario, canada. https://www.dattivo.com/simulating-real-life-events-in-python-with-simpy/.

Brownlee, J. (2018). A gentle introduction to statistical sampling and resampling. https://machinelearningmastery.com/statistical-sampling-and-resampling/.

Bruce, H. and Lapsley, M. (2014). Acquisition and interpretation of biochemical data. *Clinical Biochemistry: Metabolic and Clinical Aspects: Third Edition*, pages 6–20.

Christensen, H. I., Khan, A., Pokutta, S., and Tetali, P. (2017). Approximation and online algorithms for multidimensional bin packing: A survey . *Computer Science Review*, 24:63–79.

CloudFlare (2022). What is a computer port? — ports in networking — cloudflare. https://www.cloudflare.com/learning/network-layer/what-is-a-computer-port/. Accessed: 23-06-2022.

Csirik, J., Epstein, L., Imreh, C., and Levin, A. (2010). On the sum minimization version of the online bin covering problem. *Discrete Applied Mathematics*, 158:1381–1393.

Csirik, J. and Totik, V. (1988). Online algorithms for a dual version of bin packing. *Discrete Applied Mathematics*, 21:163–167.

Dagkakis, G., Papagiannopoulos, I., and Heavey, C. (2015). Manpy: An open-source software tool for building discrete event simulation models of manufacturing systems. *Software: Practice and Experience*, 46.

ELL (2022). Anova explained: How to compare differences of means — edanz learning lab. https://learning.edanz.com/anova-explained/.

Epstein, L., Imreh, C., and Levin, A. (2013). Bin covering with cardinality constraints. *Discrete Applied Mathematics*, 161:1975–1987.

Ganti, A. (2022). Central limit theorem (clt) definition. https://www.investopedia.com/terms/c/central_limit_theorem.asp.

GeekForGeeks (2022). Program for worst fit algorithm in memory management - geeksforgeeks. `https://www.geeksforgeeks.org/program-worst-fit-algorithm-memory-management/`.

GeeksforGeeks (2021). Difference between np hard and np complete problem.

Gillis, A. S. and Lewis, S. (2022). What is object-oriented programming (oop)? `https://www.techtarget.com/searchapparchitecture/definition/object-oriented-programming-OOP`. Accessed: 23-06-2022.

Grove, E. F. (1995). Online bin packing with lookahead.

Ham, R. V. D. (2018). Salabim: Open source discrete event simulation and animation in python.

Jabari, S., Moazzami, D., and Ghodousian, A. (2016). Heuristic and exact algorithms for generalized bin covering problem.

Konverge (2022). Custom software for manufacturers — konverge. `https://www.konverge.com/custom-software-for-manufacturers/`. Accessed: 23-06-2022.

Kuipers, L. (2021). Towards the enhancement of supply chain visibility in the fmcg industry — tu delft repositories. `https://repository.tudelft.nl/islandora/object/uuid:b164af49-c453-4654-a88a-d81892187a10?collection=education`.

Lang, S., Reggelin, T., Müller, M., and Nahhas, A. (2021). Open-source discrete-event simulation software for applications in production and logistics: An alternative to commercial tools? *Procedia Computer Science*, 180:978–987.

LSSG (2022). Anova - lean six sigma groep. `https://leansixsigmagroep.nl/lean-agile-en-six-sigma/anova/`. Accessed: 23-06-2022.

Mackenzie, R. J. (2021). One-way vs two-way anova: Differences, assumptions and hypotheses — technology networks. `https://www.technologynetworks.com/informatics/articles/one-way-vs-two-way-anova-definition-differences-assumptions-and-hypotheses-306553`.

Manpy (2022). Overview - manpy documentation. `https://www.manpy-simulation.org/`.

Marel (2022). Compact grader — marel. `https://marel.com/en/products/compact-grader`. Accessed: 23-06-2022.

McKinsey (2020). The packaging industry and the coronavirus pandemic — mckinsey. `https://www.mckinsey.com/industries/paper-forest-products-and-packaging/our-insights/how-the-packaging-industry-can-navigate-the-coronavirus-pandemic`.

Mcleod, D. S. (2021). What are confidence intervals? - simply psychology. `https://www.simplypsychology.org/confidence-interval.html`.

Mes, M. (2017). *Simulation Modelling using Practical Examples: A Plant Simulation Tutorial*. University of Twente, Netherlands.

Paape, N. (2022). Github - nickp1993/pych. `https://github.com/Nickp1993/PyCh`. Accessed: 23-06-2022.

Parker, T., Berger, S., Manson, W., Mcconnell, d'Aquino, P., August, P., Balmann, A., Berger, T., Bousquet, F., Brondízio, E., Brown, D., Couclelis, H., Deadman, P., Goodchild, M., Gotts, N., Gumerman, G., Hoffmann, M., Huigen, M., Irwin, E., and Warren, K. (2001). Meeting the challenge of complexity.

Passel (2022). Part 2: Statistical analysis and modeling — introduction to experimental design - passel. `https://passel2.unl.edu/view/lesson/2e09f0055f13/14`. Accessed: 23-06-2022.

SDX (2022). What is transmission control protocol (tcp) — sdxcentral. `https://www.sdxcentral.com/resources/glossary/transmission-control-protocol-tcp/`. Accessed: 23-06-2022.

Signavio (2022). Business process model notation - bpmn introductory guide — signavio. `https://www.signavio.com/bpmn-introductory-guide/`. Accessed: 23-06-2022.

SimPy (2022). Overview — simpy documentation. `https://simpy.readthedocs.io/en/latest/index.html`. Accessed: 23-06-2022.

Singh, G. (2018). Analysis of variance (anova) — introduction, types techniques. `https://www.analyticsvidhya.com/blog/2018/01/anova-analysis-of-variance/`.

SSC (2022). Sample size calculator. `https://www.calculator.net/sample-size-calculator.html?type=1&cl=90&ci=2&pp=50&ps=&x=77&y=26`. Accessed: 23-06-2022.

Sutiarso, E. (2022). What is the rationale behind the magic number 30 in statistics — pdf — statistics — normal distribution. `https://www.scribd.com/document/422997431/What-is-the-Rationale-Behind-the-Magic-Number-30-in-Statistics`. Accessed: 23-06-2022.

Zar, J. (2010). *Biostatistical Analysis.* Pearson Prentice Hall, 5 edition.

Zhang, G. (2001). An on-line bin-batching problem. *Discrete Applied Mathematics*, 108:329–333.

A Appendix - Code

A.1 Environment, Resource and Product

```python
# ================================
# Imports
# ================================
import PyCh

import simpy
from simpy import AnyOf
from numpy import random
from PyCh import CommunicationEvent
# ================================
# Environment
# ================================
class Environment(PyCh.Environment):
    """ A slightly modified simpy Environment.

    All simpy processes ‘live‘ in this Environment.

    """
    def __init__(self):
        super().__init__()
        self.product_id = 0  # TODO: dit is een placeholder solution

    def generate_product_id(self):
        """
        a function for generating an unique product_id
        :return: product_id
        """
        current_product_id = self.product_id
        self.product_id += 1
        return current_product_id

    def dummy_event(self):
        """ A dummy event (due to how SimPy works, every process needs at least one event)
        If a process has no events, a dummy event should be added.
        TODO: maybe we can adjust how processes work to circumvent this problem.

        :return: a timeout event of 0.0 seconds.
        """
        return self.timeout(0.0)

    def select(self, *communication_events):  # TODO: documentation
        # Removes all communication_events of NoneType (for which the guard is false)
        communication_events = [c for c in communication_events if c]
        # Check if the correct input is given, and if not, give an error.
        for c in communication_events:
            if not isinstance(c, CommunicationEvent):
                if isinstance(c, simpy.Process):
                    raise TypeError(
                        'A process was passed to the Select statement, '
                        'Try a communication_event instead.'
                    )
                else:
                    raise TypeError(
                        'One of the communication_events is of an incorrect type.'
                    )
            if self != c.env:
                raise ValueError(
                    'It is not allowed to mix events from different '
                    'environments'
                )
            if c.communication_started:
                raise ValueError(
                    'The communication_event has already started its process,'
                    'which is not allowed when used with the select statement.'
                )

        # Only one communication_event is selected. Every communication_event must know who the other communication_events are.
        # The reason is that the communication_events must communicate to each other
        for c in communication_events:
            other_communication_events = [x for x in communication_events if x != c]
            c.mutual_exclusive_communication_events.extend(other_communication_events)

        def _select_process(env, communication_events):

            # start the send/receive processes for all communication_events.
            # The order in which processes are started determines their prioritization
            # We reshuffle this order randomly to randomize prioritization
            # Note: it would also be acceptable to keep the order of the list unchanged.
            random.shuffle(communication_events)

            for c in communication_events:
                c.start_process()

            # start waiting till one of the processes is selected
            events = [c.communication for c in communication_events]
            yield AnyOf(env, events)

            entity = None
            for c in communication_events:
                if c.selected:
                    entity = c.communication.value
            return entity

        return self.process(_select_process(self, communication_events))

    # ==========================================================
    # Selected function
    # ==========================================================
    def selected(self,communication_event):
        if communication_event is None:
            return False
        elif isinstance(communication_event, CommunicationEvent):
                return communication_event.selected
        else:
            raise TypeError(
                'The input is of the incorrect type.'
            )


# ==========================================================
# Process decorator
# ==========================================================
def process(func):
    def wrapper(*args, **kwargs):
        """"""
        # first check if any of the arguments is an environment
        env = None
        for arg in args:
            if isinstance(arg, Environment):
                env = arg

        if not env:
            raise TypeError(
                'At least one of the arguments of a process should be its Environment.'
            )
        return env.process(func(*args, **kwargs))

    return wrapper
```

Listing 1: PyCh Environment

```python

from PyCh import *
from src.MarelPython import *

# =================================
# Basic Resource
# =================================
class Resource:
    """ A simple version of the basic resource type

    """
    def __init__(self, env):
        self.env = env
        self.c_in = Channel(env)
        self.c_out = None
        self._arrival_process = self.arrival_process(env)

    def connected_to(self, Resource):
        self.c_out = Resource.c_in

    @process
    def arrival_process(self, env):
        while True:
            product = yield env.execute(self.c_in.receive())
            self.product_process(env, product)
```

Listing 2: PyCh Resource

```python
# =================================
# Product
# =================================
class Product:
    def __init__(self, env, weight=0.0 ):

        # simulation properties
        self.env = env  # The environment in which the product lives.
        self.weight = weight
```

Listing 3: PyCh Product

Algorithms

The entire library of all the components working together along with the algorithm will be shown in along with the Random Allocation Algorithm. For the rest of the algorithms, only the Discharge component (Batcher class) will be shown since the rest of the code is exactly the same.

```python
import random
from typing import List

from src.MarelPython.core.environment import Environment, process
from src.MarelPython.core.resources import Resource
from src.MarelPython.core.products import Product

import numpy


# =================================
# Fillet generator
# =================================

class FilletGenerator(Resource):
    def __init__(self, env, inter_arrival_time):
        super().__init__(env)
        self.inter_arrival_time = inter_arrival_time

        self.process = self.generator_process(self.env)

    def generate_weight(self):
        mean = 200.0
        std_dev = 20.0
        weight = numpy.random.normal(mean, std_dev)
        return weight

    @process
    def generator_process(self, env):
        while True:
            yield env.timeout(self.inter_arrival_time)
            fillet = Product(env,weight        = self.generate_weight())
            yield env.execute(self.c_out.send(fillet))



# =================================
# Conveyor
# =================================

class Conveyor(Resource):
    def __init__(self, env, conveyor_length, conveyor_speed):
        super().__init__(env)

        # Parameters
        self.conveyor_length = conveyor_length
        self.conveyor_speed = conveyor_speed

    @process
    def product_process(self, env, product):
        yield env.timeout(self.conveyor_length/self.conveyor_speed)
        yield env.execute(self.c_out.send(product))

# =================================
# Batcher
# =================================

class Batcher(Resource):
    def __init__(self, env, num_of_trays):
        super().__init__(env)
        self.wastage = []
        self.num_of_trays = num_of_trays
        self.trays = [Tray(env, 1000) for i in range(num_of_trays)]

    @process
    def product_process(self, env: Environment, product: Product):
        self.tray_num: int = random.randint(0, self.num_of_trays-1)

        self.trays[self.tray_num].weight += product.weight
        self.trays[self.tray_num].xs += [product]

        if self.trays[self.tray_num].weight > self.trays[self.tray_num].target_weight:
            self.wastage.append((self.trays[self.tray_num].weight - self.trays[self.tray_num
    ].target_weight)*100/(self.trays[self.tray_num].target_weight))
            print(f"Overfill % = {self.wastage}")
            print(f"Average overfill % = {numpy.mean(self.wastage)}")
            yield env.execute(self.c_out.send(self.trays[self.tray_num]))
            self.trays[self.tray_num] = Tray(env, 1000)
            self.trays[self.tray_num].weight = 0


# =================================
# Tray
# =================================

class Tray(Resource):
    def __init__(self, env, target_weight:int, weight: int = 0):
        super().__init__(env)
        self.weight: int = weight
        self.target_weight: int = target_weight
```

```python
        self.xs: List = []
    @process
    def product_process(self, env, product):
        yield env.timeout(0.0)

# =================================
# Sink
# =================================

class Sink(Resource):
    def __init__(self, env):
        super().__init__(env)
        self.product_counter = 0  # the number of products that have arrived at this Sink
    @process
    def product_process(self, env,product):
        self.product_counter += 1
        print(f"tray {self.product_counter} has reached the sink")
        yield env.timeout(0.0)
```

Listing 4: Random Allocation Algorithm

```python
class Batcher(Resource):
    def __init__(self, env, num_of_trays):
        super().__init__(env)
        self.wastage = []
        self.num_of_trays = num_of_trays
        self.trays = [Tray(env, 1000) for i in range(num_of_trays)]
        self.tray_weights = []

    @process
    def product_process(self, env: Environment, product: Product):
        for i in range(self.num_of_trays):
            self.tray_weights.append(self.trays[i].weight)
        self.min = min(self.tray_weights)
        self.tray_weights = []
        for x in range(self.num_of_trays):
            if self.trays[x].weight == self.min:
                self.tray_num = x


        self.trays[self.tray_num].weight += product.weight
        self.trays[self.tray_num].xs += [product]

        if self.trays[self.tray_num].weight > self.trays[self.tray_num].target_weight:
            self.wastage.append((self.trays[self.tray_num].weight - self.trays[self.tray_num
    ].target_weight)*100/(self.trays[self.tray_num].target_weight))
            print(f"Overfill % = {self.wastage}")
            print(f"Average overfill % = {numpy.mean(self.wastage)}")
            yield env.execute(self.c_out.send(self.trays[self.tray_num]))
            self.trays[self.tray_num] = Tray(env, 1000)
            self.trays[self.tray_num].weight = 0
```

Listing 5: Worst Fit Algorithm

```python
class Batcher(Resource):
    def __init__(self, env, num_of_trays):
        super().__init__(env)
        self.wastage = []
        self.num_of_trays = num_of_trays
        self.trays = [Tray(env, 1000) for i in range(num_of_trays)]
        self.tray_weights = []

    @process
    def product_process(self, env: Environment, product: Product):
        for i in range(self.num_of_trays):
            self.tray_weights.append(self.trays[i].weight)
        self.tray_weights.sort()
        counter = 1

        if counter <= self.num_of_trays:
            self.min = min(self.tray_weights)
            for x in range(self.num_of_trays):
                if self.trays[x].weight == self.min:
                    self.tray_num = x
            counter +=1
        else:
            if product.weight < 180:
                self.tray_num = self.num_of_trays - 1
            elif 180 <= product.weight < 200:
                self.tray_num = self.num_of_trays - 2
            elif 200 <= product.weight <= 220:
                self.tray_num = 1
            elif product.weight > 220:
                self.tray_num = 0

        self.tray_weights = []
        self.trays[self.tray_num].weight += product.weight
        self.trays[self.tray_num].xs += [product]

        if self.trays[self.tray_num].weight > self.trays[self.tray_num].target_weight:
            self.wastage.append((self.trays[self.tray_num].weight - self.trays[self.tray_num
    ].target_weight)*100/(self.trays[self.tray_num].target_weight))
            print(f"Overfill % = {self.wastage}")
            print(f"Average overfill % = {numpy.mean(self.wastage)}")
            yield env.execute(self.c_out.send(self.trays[self.tray_num]))
            self.trays[self.tray_num] = Tray(env, 1000)
            self.trays[self.tray_num].weight = 0
```

Listing 6: Custom Algorithm 1 (Variant 1)

```python
class Batcher(Resource):
    def __init__(self, env, num_of_trays):
        super().__init__(env)
        self.wastage = []
        self.num_of_trays = num_of_trays
        self.trays = [Tray(env, 1000) for i in range(num_of_trays)]
        self.tray_weights = []

    @process
    def product_process(self, env: Environment, product: Product):
        for i in range(self.num_of_trays):
            self.tray_weights.append(self.trays[i].weight)
        self.tray_weights.sort()
        counter = 1

        if counter <= self.num_of_trays:
            self.min = min(self.tray_weights)
            for x in range(self.num_of_trays):
                if self.trays[x].weight == self.min:
                    self.tray_num = x
            counter +=1
        else:
            if product.weight < 200:
                self.tray_num = self.num_of_trays - 1
            else:
                self.tray_num = 0

        self.tray_weights = []
        self.trays[self.tray_num].weight += product.weight
        self.trays[self.tray_num].xs += [product]

        if self.trays[self.tray_num].weight > self.trays[self.tray_num].target_weight:
            self.wastage.append((self.trays[self.tray_num].weight - self.trays[self.tray_num
    ].target_weight)*100/(self.trays[self.tray_num].target_weight))
            print(f"Overfill % = {self.wastage}")
            print(f"Average overfill % = {numpy.mean(self.wastage)}")
            yield env.execute(self.c_out.send(self.trays[self.tray_num]))
            self.trays[self.tray_num] = Tray(env, 1000)
            self.trays[self.tray_num].weight = 0
```

Listing 7: Custom Algorithm 1 (Variant 2)

```
class Batcher(Resource):
    def __init__(self, env, num_of_trays):
        super().__init__(env)
        self.wastage = []
        self.num_of_trays = num_of_trays
        self.trays = [Tray(env, 1000) for i in range(num_of_trays)]
        self.tray_weights = []
        self.count = 0

    @process
    def product_process(self, env: Environment, product: Product):
        self.tray_num = random.randint(0,self.num_of_trays-1)
        self.shuffle = 100
        while self.shuffle > 0:
            if self.trays[self.tray_num].weight + product.weight > 0.92*self.trays[self.tray_num].target_weight and self.trays[self.tray_num].weight + product.weight < self.trays[self.tray_num].target_weight:
                self.tray_num = random.randint(0,self.num_of_trays-1)
                self.shuffle -= 1
            else:
                self.shuffle = 0

        self.trays[self.tray_num].weight += product.weight
        self.trays[self.tray_num].xs += [product]

        if self.trays[self.tray_num].weight > self.trays[self.tray_num].target_weight:
            self.wastage.append((self.trays[self.tray_num].weight - self.trays[self.tray_num].target_weight) * 100 / (self.trays[self.tray_num].target_weight))
            print(f"Overfill % = {self.wastage}")
            print(f"Average overfill % = {numpy.mean(self.wastage)}")
            yield env.execute(self.c_out.send(self.trays[self.tray_num]))
            self.trays[self.tray_num] = Tray(env, 1000)
            self.trays[self.tray_num].weight = 0
```

Listing 8: Custom Algorithm 2

```
class Batcher(Resource):
    def __init__(self, env, num_of_trays):
        super().__init__(env)
        self.wastage = []
        self.num_of_trays = num_of_trays
        self.trays = [Tray(env, 1000) for i in range(num_of_trays)]
        self.tray_weights = []

    @process
    def product_process(self, env: Environment, product: Product):
        for i in range(self.num_of_trays):
            self.tray_weights.append(self.trays[i].weight)
        for i in range(self.num_of_trays):
            if ((self.trays[i].target_weight < self.tray_weights[i] + product.weight) and (self.tray_weights[i] + product.weight < 1.05*(self.trays[i].target_weight))):
                self.tray_num = i
            else:
                self.tray_num = random.randint(0, self.num_of_trays-1)

        self.tray_weights = []

        self.trays[self.tray_num].weight += product.weight
        self.trays[self.tray_num].xs += [product]


        if self.trays[self.tray_num].weight > self.trays[self.tray_num].target_weight:
```

```python
            self.wastage.append((self.trays[self.tray_num].weight - self.trays[self.tray_num
    ].target_weight)*100/(self.trays[self.tray_num].target_weight))
            print(f"Overfill % = {self.wastage}")
            print(f"Average overfill % = {numpy.mean(self.wastage)}")
            yield env.execute(self.c_out.send(self.trays[self.tray_num]))
            self.trays[self.tray_num] = Tray(env, 1000)
            self.trays[self.tray_num].weight = 0
```

Listing 9: Custom Algorithm 3 (Variant 1)

```python
class Batcher(Resource):
    def __init__(self, env, num_of_trays):
        super().__init__(env)
        self.wastage = []
        self.num_of_trays = num_of_trays
        self.trays = [Tray(env, 1000) for i in range(num_of_trays)]
        self.tray_weights = []

    @process
    def product_process(self, env: Environment, product: Product):
        for i in range(self.num_of_trays):
            self.tray_weights.append(self.trays[i].weight)
        for i in range(self.num_of_trays):
            if ((self.trays[i].target_weight < self.tray_weights[i] + product.weight) and (
    self.tray_weights[i] + product.weight < 1.05*(self.trays[i].target_weight))):
                self.tray_num = i
            else:
                self.min = min(self.tray_weights)
                for x in range(self.num_of_trays):
                    if self.trays[x].weight == self.min:
                        self.tray_num = x

        self.tray_weights = []

        self.trays[self.tray_num].weight += product.weight
        self.trays[self.tray_num].xs += [product]


        if self.trays[self.tray_num].weight > self.trays[self.tray_num].target_weight:
            self.wastage.append((self.trays[self.tray_num].weight - self.trays[self.tray_num
    ].target_weight)*100/(self.trays[self.tray_num].target_weight))
            print(f"Overfill % = {self.wastage}")
            print(f"Average overfill % = {numpy.mean(self.wastage)}")
            yield env.execute(self.c_out.send(self.trays[self.tray_num]))
            self.trays[self.tray_num] = Tray(env, 1000)
            self.trays[self.tray_num].weight = 0
```

Listing 10: Custom Algorithm 3 (Variant 2)

```python
class Batcher(Resource):
    def __init__(self, env, num_of_trays):
        super().__init__(env)
        self.wastage = []
        self.num_of_trays = num_of_trays
        self.trays = [Tray(env, 1000) for i in range(num_of_trays)]
        self.tray_weights = []
        self.count = 0

    @process
    def product_process(self, env: Environment, product: Product):
        for i in range(self.num_of_trays):
            self.tray_weights.append(self.trays[i].weight)
        self.min = min(self.tray_weights)
        self.tray_weights = []
        for x in range(self.num_of_trays):
            if self.trays[x].weight == self.min:
                self.tray_num = x


        self.trays[self.tray_num].weight += product.weight
        self.trays[self.tray_num].xs += [product]

        if self.trays[self.tray_num].weight > self.trays[self.tray_num].target_weight:
            if self.count <= 15:
                self.wastage.append((self.trays[self.tray_num].weight - self.trays[self.
    tray_num].target_weight)*100/(self.trays[self.tray_num].target_weight))
                print(f"Overfill % = {self.wastage}")
                positive = filter(lambda x: x > 0, self.wastage)
                print(f"Average overfill % = {numpy.mean(list(positive))}")
                yield env.execute(self.c_out.send(self.trays[self.tray_num]))
                self.trays[self.tray_num] = Tray(env, 1000)
                self.trays[self.tray_num].weight = 0
                self.count += 1
            else:
                for x in range(self.num_of_trays):
                    if self.trays[x].weight != 0:
                        self.wastage.append((self.trays[x].weight - self.trays[x].
    target_weight) * 100 / (self.trays[x].target_weight))
                        print(f"Overfill % = {self.wastage}")
                        positive = filter(lambda x: x>0,self.wastage)
                        print(f"Average overfill % = {numpy.mean(list(positive))}")
                        yield env.execute(self.c_out.send(self.trays[x]))
                        self.trays[x] = Tray(env, 1000)
                        self.trays[x].weight = 0
                self.count = 0
```

Listing 11: Custom Algorithm 4