AMSI SUMMER RESEARCH SCHOLARSHIPS 2025–26

Get a taste for Research this Summer

Optimisation Techniques to Multi-Component Bin Cutting and Packing Problems

My Huy Lim

Supervised by Dr. Sergey Polyakovskiy

Deakin University

Jane Street®

AMSI AUSTRALIAN MATHEMATICAL SCIENCES INSTITUTE

Abstract

Cutting and packing problems are fundamental optimization challenges with widespread industrial applications in manufacturing, logistics and production planning. In many real-world settings, these problems arise as multi-component optimization tasks, where several interdependent sub-problems must be solved simultaneously, significantly increasing computational complexity and limiting the effectiveness of traditional optimization methods.

This research focuses on the optimization of multi-component bin cutting and packing problems, aiming to address their compounded structure and conflicting constraints. To tackle these challenges, the study investigates hybrid and decomposition-based optimization techniques that combine multiple algorithmic strategies to efficiently explore the solution space.

The proposed approaches decompose the overall problem into manageable components while coordinating their interactions through advanced search and diversification strategies. Experimental results on representative problem instances demonstrate improved solution quality and scalability compared to conventional methods.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Importance of Multi-Component Bin Cutting and Packing Problems

Multi-component bin cutting and packing problems appear across a wide range of industries including manufacturing, logistics, warehousing, construction, and eCommerce. Companies involved in packaging, shipment consolidation, production planning and material cutting (e.g., steel, glass, textiles, wood, and paper industries) rely heavily on cutting and packing optimization to reduce material waste, improve resource usage, and lower overall operational costs. Cutting and packing problems are widely studied due to their many real-world applications and the high complexity of their industrial forms [5].

Unlike traditional cutting and packing problems that focus solely on finding a feasible arrangement of items within a single bin or stock sheet, multi-component cutting and packing problems integrate multiple sub-problems into one compounded optimization task. These problems may combine decisions such as item selection, packing feasibility, sequencing, scheduling, and downstream logistics, creating strong interdependence between components. As a result, complexity grows rapidly due to conflicting objectives, constraints, and decision layers, making them significantly more challenging than simpler single-component formulations.

Due to the high complexity of multi-component cutting and packing problems, manual planning becomes impractical in real-world scenarios and can lead to inefficient outcomes such as excessive waste, increased production time, and higher transportation costs. While traditional optimization approaches such as exact methods may only be effective for small-scale cases, modern solution techniques such as decomposition methods, hybrid metaheuristics, and advanced search frameworks are more promising for large and realistic industrial instances. In particular, decomposition-based hybrid optimization methods have been highlighted as effective for handling the structure and compounded complexity of challenging combinatorial optimization problems [7]. Therefore, advanced optimization techniques are crucial for developing scalable, high-quality and industry-ready solutions that reduce cost, improve efficiency, and enhance sustainability through waste minimization and improved resource consumption.

2 Literature Review

Multi-component cutting, packing, and scheduling problems with earliness–tardiness objectives represent a class of highly complex combinatorial optimisation problems. These problems integrate layout feasibility with time-sensitive scheduling decisions and are generally NP-hard, making them difficult to solve using exact optimisation methods for realistic instance sizes.

Polyakovskiy et al. [6] studied the Just-in-Time Batch Scheduling problem with weighted earliness–tardiness penalties and demonstrated that exact solvers such as CPLEX are limited in scalability when batching and scheduling decisions are combined. Their work highlights the need for heuristic and metaheuristic approaches when addressing integrated packing–scheduling problems.

A comprehensive survey by Desale et al. [1] reviewed major heuristic and metaheuristic frameworks, including Genetic Algorithms, Simulated Annealing, Tabu Search, and Variable Neighbourhood Search. The authors emphasised the importance of memory mechanisms, acceptance criteria, and diversification strategies in achieving competitive performance. Their findings support the adoption of hybrid and domain-specific metaheuristics for large-scale optimisation problems.

Memory-guided local search techniques, particularly Tabu Search, have been shown to improve robustness in complex search landscapes. Holger and Thomas [3] demonstrated how tabu lists, aspiration criteria, and adaptive tenures enable effective escape from local optima. Such mechanisms are especially relevant for cutting and packing problems, where poor configurations can trap simple local search procedures.

Variable Neighbourhood Search (VNS) provides a systematic framework for balancing intensification and diversification by dynamically changing neighbourhood structures. Hansen et al. [2] showed that VNS and its variants consistently deliver strong results for scheduling, routing, and packing problems. The flexibility of VNS makes it suitable for multi-component optimisation, where multiple interacting decision dimensions must be explored.

Hybrid approaches combining constructive heuristics with metaheuristics have received increasing attention. Hu et al. [4] proposed a hybrid of Simulated Annealing and a minimum horizontal line heuristic for pallet loading, achieving substantial improvements over classical methods. Their work demonstrates that constructive heuristics can provide high-quality initial solutions, while metaheuristics enable global refinement.

For scheduling problems with tardiness objectives, Wang et al. [9] developed an iterated greedy algorithm integrating heuristic seeding, destruction–reconstruction phases, and acceptance rules. Their approach achieved strong performance on large instances and highlights the effectiveness of hybrid perturbation and re-optimisation mechanisms in time-sensitive environments.

Robustness and premature convergence remain important challenges in population-based metaheuristics. Sultana et al. [8] introduced an adaptive multi-restart exploration framework for Genetic Algorithms based on solution clustering. Their method improved search diversity and stability by identifying promising regions of the search space and focusing computational effort accordingly.

2.1 Research Gaps

Despite significant advances, several limitations remain in the current literature. First, exact optimisation methods remain unsuitable for large integrated packing–scheduling problems. Second, relatively few approaches jointly optimise bin packing, batching, and tardiness scheduling within a unified framework. Third, many metaheuristic methods exhibit sensitivity to parameter settings and stochastic variability, requiring improved mechanisms for adaptive control and robustness.

2.2 Positioning of This Research

Motivated by these gaps, this research develops a hybrid construct–improve–diversify framework that integrates fitness-based decomposition, exact LP-based evaluation, and coordinated diversification strategies. By combining structured construction with controlled stochastic exploration, the proposed approach aims to improve both solution quality and robustness for multi-component cutting and packing problems with earliness–tardiness penalties.

3 Methodology

This research adopts a quantitative computational methodology based on iterative algorithm development and experimental evaluation. The study follows a systematic cycle of literature review, framework design, implementation, and performance analysis. Prior research on hybrid metaheuristics, decomposition methods, and scheduling with earliness–tardiness objectives was reviewed to inform the proposed approach, with regular supervision guiding methodological decisions.

A hybrid *construct–improve–diversify* optimization framework was designed and implemented in C. The framework integrates fitness-based decomposition, biased randomized construction, exact LP-based evaluation using Gurobi, structured local improvement, and multiple coordinated diversification operators. Emphasis was placed on modular design, reproducibility, and traceability through explicit index management, controlled random seeding, and comprehensive logging.

The framework was evaluated using benchmark instances of 20, 40, and 60 items provided by the supervisor, representing small, medium, and larger problem scales. For each configuration, multiple independent runs were conducted using fixed parameter settings and different random seeds. This experimental design enabled systematic analysis of solution quality, stability, and stochastic behavior.

Performance was measured using objective value (total weighted earliness–tardiness penalty), number of bins, convergence time, and runtime distribution between construction and diversification phases. All experimental data were recorded in structured CSV logs and analyzed on both per-run and aggregated levels. Results were compared across parameter settings and against published research benchmarks to assess effectiveness and efficiency.

Ethical considerations were minimal, as only synthetic benchmark datasets were used. Strong emphasis was placed on reproducibility and transparency to ensure that all results can be independently validated.

4 Problem Definition and Evaluation Model

4.1 Problem Description

The problem involves a set of identical rectangular bins $B = \{1, \ldots, m\}$ and a set of rectangular items $I = \{1, \ldots, n\}$ to be packed into the bins.

Each item $i \in I$ is defined by its length l_i, width w_i, due date d_i, desired completion time c_i, early penalty ϵ_i, and late penalty τ_i.

Each bin $b \in B$ is characterized by its length L and width W, which together define the constraint on the capacity of the bin. All items assigned to a bin b share a common completion time C_b. The processing time of a bin b, denoted by p_b, is a linear function of the number of items assigned to the bin, expressed as $f(I_b)$, where $I_b \in I$ is the set of items packed in bin b.

The early and late penalties measure how early or late the completion time of bin b, C_b, is relative to the due date d_i of each item assigned to b.

The objective of the problem is to determine a feasible packing of items into bins and a feasible

schedule of the bins that minimizes the total weighted earliness–tardiness penalty, defined as

$$\min \sum_{i=1}^{n} (\epsilon_i E_i + \tau_i T_i)\,,$$

where $E_i = \max\{0, d_i - c_b\}$ and $T_i = \max\{0, c_b - d_i\}$ when item $i \in I_b$

4.2 Item, Bin, and Schedule Representation

In the implementation, a candidate solution is represented solely by an assignment vector

$$x = (x_1, \ldots, x_n),$$

where $x_i = b$ indicates that item i is assigned to bin b. The set of items assigned to each bin is derived directly from this vector, and the total number of bins used is computed from the assignment.

4.3 Solution Evaluation Model

Candidate solutions are evaluated using a linear programming (LP) model implemented in `Gurobi_LP.LP`. The heuristic constructs a fixed item-to-bin assignment x, while the LP optimizes timing-related variables to compute the corresponding objective value.

Given an instance p, the number of used bins m, and an assignment vector x, the evaluation model determines: (i) bin completion times C_b, (ii) item-level earliness and tardiness values (E_i, T_i), and (iii) the resulting objective value.

The LP minimizes the total weighted earliness–tardiness penalty for the fixed assignment:

$$\min \sum_{i=1}^{n} (\epsilon_i E_i + \tau_i T_i)\,.$$

If the LP cannot be solved to optimality, a large sentinel value (`int.MaxValue`) is returned to indicate evaluation failure.

4.4 Decision Variables

For a given assignment x, the quality of the solution is assessed solely through its objective value. The LP optimizes bin completion times C_b and item-level earliness and tardiness values (E_i, T_i), and these variables determine the resulting objective value. The heuristic does not modify timing variables directly; it operates only on the assignment x, while the LP provides an exact evaluation of the associated scheduling cost.

5 Core Components

5.1 Fitness Band Component

5.1.1 Fitness Definition

The initial step involves identifying the most troublesome group of items to be sorted into bins first, as they contribute the most to the earliness–lateness penalty score. A fitness value is computed for each item as follows.

Given a problem instance p, for each item i, define its raw fitness value as

$$f_i = \frac{w_i}{\max(\epsilon_i, \tau_i)}, \tag{1}$$

where w_i denotes the item weight (or size measure used by the instance), and ϵ_i and τ_i are the earliness and tardiness penalty weights.

Let $f_{\min} = \min_{i \in I} f_i$ and $f_{\max} = \max_{i \in I} f_i$. The normalized fitness value of item i is computed as

$$\hat{f}_i = \begin{cases} \dfrac{f_i - f_{\min}}{f_{\max} - f_{\min}}, & \text{if } f_{\max} > f_{\min}, \\ 0.5, & \text{otherwise.} \end{cases}$$

5.1.2 Fitness Normalization

The normalization above scales raw fitness values into $[0, 1]$ to enable consistent thresholding across instances.

5.1.3 Fitness Band Partitioning

Once a fitness score is calculated for each item, a threshold is chosen to split the normalized fitness range $[0, 1]$ into three categories: small, medium, and large. Threshold values, 0.1 and 0.2 were used as threshold for different instance sizes. Small thresholds were used considering a large number of items as troublesome to better construct the initial assignment; if larger values were used such as 0.5, this would make the algorithm disregard too many items as troublesome and the initial assignment would be more of a random initialization.

5.1.4 Rationale for Medium-Fitness Priority

With how the formula in (1) is defined, small fitness means that the item's penalty is expensive compared to each unit of its weight (i.e., small weight, large penalty), large means that the item's penalty is cheap compared to each unit of its weight (i.e., large weight, small penalty), and medium means that the weight and penalty are proportional to each other (i.e., (large weight, large penalty)

or (small weight, small penalty)). This observation suggests that small fitness items can easily be inserted into remaining space of bins due to their small weight, and large fitness items can be inserted into existing partially filled assignments or isolated into their own bins with less impact on the total earliness-tardiness penalty. On the other hand, medium-fitness items contain the most troublesome group of items: (large weight, large penalty) items are hard to move around in a partially filled assignment due to their size, and any movement may dramatically affect the assignment's penalty. Therefore, the framework prioritizes filling medium-fitness items first, then small and large items afterwards.

Let I be the full set of items in the original instance p. Using the fitness-band partitioning defined in Section 5.1.3, let $I_{\text{med}} \subseteq I$ denote the subset of medium-fitness items.

5.2 Biased Randomization Components

5.2.1 Biased Index Selection

Several stages of the framework require selecting an item from a list that has already been ranked by desirability (e.g., decreasing normalised fitness or a combined score). To avoid purely greedy behaviour while still favouring high-ranked candidates, selection is performed using a biased randomised index rule. Conceptually, the method assigns higher probability to small ranks and then samples one position at random; the selected item is the element at that position in the ranked list.

In the implementation, this mechanism is encapsulated in `BiasHelper.BiasedPickIndex(·)` and is reused for (i) selecting the initial medium-item subset for the two-bin seed, (ii) filling newly created bins during the medium-bin construction loop, and (iii) choosing non-medium items for incremental insertion.

5.2.2 Biased Two-Bin Seed Generation

To initialize construction on the medium-fitness subset, a small set of medium items is sampled using biased randomization. Items in I_{med} are first ranked by decreasing normalized fitness (higher means more "troublesome" under the fitness definition). A subset $S \subseteq I_{\text{med}}$ is then obtained by repeatedly applying the biased index selection rule to this ranked list.

The sampled items are restricted to two bins and are accompanied by a bin-map $\beta : S \to \{0, 1\}$ that assigns each selected item to one of two initial bins. This produces a two-bin seed assignment that is subsequently improved by OPT_ALL, with a feasibility fallback to the seeded assignment if capacity constraints are violated.

5.3 Sorting and Index Management Components

This component provides the supporting mechanisms required to maintain a consistent and reproducible correspondence between item identities in the original problem instance and their positions in dynamically constructed restricted subproblems. Since the framework repeatedly constructs, expands, and perturbs subproblems throughout solution construction and diversification, careful index management is essential to ensure correctness of assignments, feasibility checks, and objective evaluation.

Two core mechanisms are employed: stable due-date sorting and explicit index mapping between the original instance and each restricted subproblem.

5.3.1 Stable Due-Date Sorting

Whenever a subset of items is selected for subproblem construction or expansion, the items are ordered using a stable due-date sorting operator. Given a problem instance p and a list of original item indices $S \subseteq I$, the operator SORTBYDUESTABLE(p, S) returns an ordered list $\pi = (i_1, \ldots, i_{|S|})$ such that

$$d_{i_1} \leq d_{i_2} \leq \cdots \leq d_{i_{|S|}},$$

while preserving the relative order of any two items with equal due dates.

This deterministic stability property is critical for reproducibility: repeated runs with identical inputs produce identical orderings, avoiding tie-induced noise during seed construction and subproblem rebuilding. The resulting Earliest Due Date (EDD) ordering provides a strong constructive sequence that reduces the risk of large tardiness penalties in early solution phases.

Algorithm 1: SORTBYDUESTABLE(p, S)

Input:
p: problem instance with due dates $d[i]$
S: list of original item indices

Output:
π: list of indices sorted by non-decreasing due date, stable on ties

Attach original order position to each index in S;
Sort indices by increasing $d[i]$;
Break ties using the original order positions;
Return the ordered list π;

5.3.2 Index Mapping Between Original and Subproblems

Restricted subproblems are constructed by selecting subsets of items from the original instance and re-indexing them from 1 to the subproblem size. Let $\pi = (i_1, \ldots, i_k)$ denote the ordered list of original item indices used to build a subproblem p'. A bijective mapping

$$\mu : i_j \mapsto j \quad \text{for } j = 1, \ldots, k$$

is constructed to translate between original item indices and subproblem positions.

This mapping is required whenever:

- an existing assignment vector is transferred to a rebuilt subproblem,
- new items are appended during subproblem expansion,
- assignments are merged after diversification or perturbation steps.

In the implementation, this mapping is realised explicitly as a dictionary from original item indices to their positions in the ordered subproblem index list.

Algorithm 2: BUILDINDEXMAP(π)

Input:
$\pi = (i_1, \ldots, i_k)$: ordered list of original item indices

Output:
μ: mapping from original index to subproblem position

Initialize empty map μ;
for $j = 1$ *to* k **do**
  Set $\mu[i_j] \leftarrow j$;
return μ;

5.3.3 Assignment Remapping During Subproblem Expansion

When a subproblem is rebuilt after adding new items, the ordering of the index list may change due to stable due-date sorting. Therefore, any existing assignment vector must be remapped to align with the new index ordering.

Let x be the assignment vector aligned with the previous index list π_{old}, and let π_{new} denote the updated ordered index list. Using the index map μ, assignments are transferred as follows.

Algorithm 3: REMAPASSIGNMENT($\pi_{\text{old}}, \pi_{\text{new}}, x$)

Input:
π_{old}: previous ordered index list
π_{new}: updated ordered index list
x: assignment vector aligned with π_{old}

Output:
x': remapped assignment vector aligned with π_{new}

Build index map μ from π_{new};
Initialize x' of length $|\pi_{\text{new}}|$;
for *each position* j *in* π_{old} **do**
  Let $i = \pi_{\text{old}}[j]$;
  Set $x'[\mu(i)] \leftarrow x[j]$;
Assign bins to newly added items as required;
return x';

5.4 Subproblem Construction Component

This component constructs *restricted subproblems* by selecting a subset of items from the original instance and re-indexing them into a compact instance that can be evaluated and improved by the MIP/LP evaluator and OPT_ALL. In the implementation, subproblem construction is used (i) to create the initial two-bin medium instance, and (ii) repeatedly during expansion when new items are appended (during medium-bin creation and during non-medium insertion), where the subproblem is rebuilt on an updated index set.

5.4.1 Restricted Subproblem Generation

Two-bin medium initialization (subproblem + seed solution). Given the ordered index list $\pi = (i_1, \ldots, i_k)$, we first construct a restricted instance containing only these medium-fitness items,

$$p_{\text{2med}} = \textsc{BuildSubProblem}(p, \pi),$$

where items are copied from the original instance and re-indexed to $\{1, \ldots, k\}$. We then initialize a two-bin seed assignment using the bin-map β,

$$x^{(0)} = \textsc{SeedAssignment}(\beta, \pi),$$

where each $x_k^{(0)} \in \{0, 1\}$ specifies whether item i_k is placed in bin 0 or bin 1. By construction, the initial seed uses at most two bins, i.e., $B(x^{(0)}) \leq 2$.

Algorithm 4: $\textsc{BuildSubProblem}(p, \pi)$

Input:
p: original problem instance containing all items
π: ordered list of selected original item indices

Output:
p': restricted subproblem containing only items in π, re-indexed to $1, \ldots, |\pi|$

Create an empty problem instance p';;
Copy the bin parameters (e.g., capacity) from p into p';;
for $k \leftarrow 1$ **to** $|\pi|$ **do**
    $i \leftarrow \pi[k]$;;
    Copy item i (all attributes) from p into p' as item k;;
return p';;

Algorithm 5: $\textsc{SeedAssignment}(\beta, \pi)$

Input:
β: bin-map assigning each selected item to bin 0 or 1
π: ordered list of selected item indices

Output:
$x^{(0)}$: initial two-bin assignment vector for items in π

Initialize an assignment vector $x^{(0)}$ of length $|\pi|$;
For each position $k = 1, \ldots, |\pi|$:
    Let i_k be the k-th item in π;
    Set $x_k^{(0)} \leftarrow \beta(i_k)$;
Return $x^{(0)}$;

5.4.2 Preservation of Item Attributes

Subproblem construction is *lossless* with respect to item data: for each selected item $i \in \pi$, its geometric attributes (length/width or weight), due date d_i, and penalty parameters (ϵ_i, τ_i) are copied verbatim from the original instance. The only modification is the index system: items are renumbered to $\{1, \ldots, |\pi|\}$ to produce a dense representation for the optimization routines. Likewise, bin capacity is preserved exactly, so feasibility checking and objective evaluation remain consistent with the original instance.

When the subproblem is rebuilt after adding new items, the ordering of the updated index list may change due to stable due-date sorting. Therefore, any existing assignment vector must be remapped to the new index order using the procedure described in Section 5.3. In the implementation, this remapping uses an original-index-to-position map (e.g., `idxToPos`) constructed over the updated ordered list.

5.4.3 Feasibility Safeguards

Feasibility safeguard. If $x^{(1)}$ violates bin capacity constraints, it is rejected and the method falls back to the seeded solution:

$$x^\star = \begin{cases} x^{(1)}, & \text{if } x^{(1)} \text{ is feasible,} \\ x^{(0)}, & \text{otherwise.} \end{cases}$$

The resulting assignment $x^\star$ and its objective value serve as the initial two-bin medium solution that is subsequently expanded by the bin creation procedure.

5.5 Local Improvement Component: OPT_ALL

5.5.1 Exact Evaluation and Local Improvement

Exact evaluation and local improvement. Assignments are evaluated using a Gurobi-based linear programming (LP) model to compute the corresponding objective value and timing-related quantities which returns bin completion times and item-level earliness and tardiness values consistent with the evaluation model.

A local improvement procedure OPT_ALL$(\cdot)$ is then applied to refine the seeded assignments.

5.6 Bin and Feasibility Helper Components

5.6.1 Bin Counting

Given an assignment vector x, the number of used bins is computed as $B(x) = |\{x_i \mid i = 1, \ldots, n\}|$. This value is passed to the LP evaluator as m and is also used for reporting.

5.6.2 Capacity Feasibility Checking

Capacity feasibility is checked by summing item weights within each bin induced by x. If any bin exceeds the capacity, the assignment is rejected and the algorithm reverts to the last feasible assignment (e.g., the pre-OPT_ALL or pre-perturbation solution).

5.6.3 Bin Index Compression

After perturbation moves (e.g., bin removal or swaps), bin labels may become non-contiguous. The implementation compresses bin ids by remapping the set of used labels to $\{0, \ldots, m-1\}$ while preserving the assignment structure. This ensures the LP evaluator receives a consistent bin index set.

5.7 Diversification Strategy Components

After an initial feasible solution has been constructed and locally improved using OPT_ALL, the framework enters a diversification phase designed to escape local optima and explore structurally different regions of the solution space. Diversification is achieved by applying a collection of stochastic perturbation strategies that deliberately disrupt the current assignment while preserving feasibility or restoring it through subsequent re-optimization.

Each diversification iteration proceeds by selecting a starting solution from the solution pool, applying exactly one diversification strategy, rebuilding the corresponding subproblem, and re-optimising the perturbed assignment using OPT_ALL. Only solutions that improve upon the worst pool entry are retained.

The implemented diversification strategies can be grouped into five main categories, described below.

5.7.1 Item Relocation Strategies

Item relocation strategies aim to reduce the objective value by identifying high-penalty items and relocating them to more favourable bins. These moves are guided by earliness–tardiness penalties and are therefore more informed than purely random perturbations.

Worst-item relocation. The primary relocation strategy identifies the *worst bin* (i.e., the bin with the highest aggregated penalty contribution) and then selects the *worst item* within that bin. The selected item is tentatively relocated to a small sample of alternative bins that can accommodate it without violating capacity constraints. Each sampled relocation is evaluated using the LP evaluator, and the first improving move is accepted.

This strategy directly targets the dominant contributors to the weighted earliness–tardiness objective and provides a focused yet stochastic diversification step.

Algorithm 6: RELOCATEWORSTITEMSAMPLED

Input: problem p, assignment x, LP timing values (C, E, T), objective z, random generator `rng`

```
Identify worst bin b* based on penalty contribution;
Identify worst item i* ∈ b*;
Determine feasible destination bins for i*;
Randomly sample up to k feasible bins;
for each sampled bin b do
    Move i* to bin b;
    Evaluate new assignment using LP;
    if objective improves then
        return improved assignment;
return original assignment;
```

5.7.2 Item Reinsertion Strategies

Item reinsertion strategies perform controlled random disruption by removing an item from its current bin and reinserting it elsewhere. Unlike relocation strategies, reinsertion does not explicitly target the worst item, thereby introducing additional randomness into the search.

Single-item reinsertion. A random bin is selected, followed by a random item from that bin. The item is removed and reinserted into a randomly ordered list of alternative bins that can accommodate it. If no existing bin can host the item without violating capacity constraints, a new bin is created.

This operator encourages exploration of new bin configurations while retaining overall feasibility.

Algorithm 7: REINSERTONEITEM

Input: problem p, assignment x, random generator `rng`

```
Select a random non-empty bin b;
Select a random item i ∈ b;
Remove i from its bin;
Shuffle remaining bins;
for each candidate bin b' do
    if capacity allows then
        Assign i to b';
        return updated assignment;
Create a new bin and assign i;
return updated assignment;
```

5.7.3 Bin-Level Perturbation Strategies

Bin-level strategies introduce larger structural changes by manipulating entire bins rather than individual items. These operators are effective at escaping deep local minima where small item-level moves are insufficient.

Bin swapping. Two distinct bins are selected at random, and all items assigned to these bins are swapped. This operation preserves bin loads but alters the temporal order and penalty structure

induced by bin completion times.

Bin removal and shuffle reinsertion. A randomly selected bin is removed entirely, and its items are collected. Additionally, an equal number of randomly selected items from other bins are temporarily removed. All removed items are then reinserted one by one using a greedy capacity-based reinsertion rule.

This aggressive perturbation reshapes both bin composition and bin count, promoting exploration of substantially different packing and scheduling configurations.

Algorithm 8: REMOVEBINANDSHUFFLEREINSERT

Input: problem p, assignment x, random generator `rng`

Select a random bin b and remove it;
Collect items from b;
Randomly remove the same number of additional items from other bins;
Reinsert all removed items greedily into feasible bins, creating new bins if needed;
return updated assignment;

5.7.4 Penalty-Based Removal Strategies

Penalty-based removal strategies focus on medium-fitness items with disproportionately large contributions to the objective function. These items are difficult to place optimally during construction and are therefore strong candidates for targeted disruption.

Medium-item penalty-weight removal. Items in the medium-fitness band are ranked according to their penalty density, defined as weighted earliness–tardiness penalty per unit weight. A biased randomised selection rule is then used to remove a small fraction (approximately 10% of the total instance size) of the highest-ranked medium items. Removed items are subsequently reinserted using a capacity-feasible heuristic.

This strategy balances intensification and diversification by focusing on high-impact items while still incorporating randomness in the removal process.

Algorithm 9: REMOVEHIGHPENALTYMEDIUMITEMS

Input: problem p, assignment x, medium item set I_{med}, random generator `rng`

Evaluate assignment using LP;
Compute penalty density for each $i \in I_{\text{med}}$;
Rank items by decreasing penalty density;
Remove a biased sample of top-ranked items;
Reinsert removed items using capacity-feasible insertion;
return updated assignment;

5.7.5 Constraint Programming-Based Strategies

In addition to heuristic perturbations, the framework incorporates three constraint programming (CP) based diversification strategies provided as part of the project specification. These strategies embed a limited CP model to re-optimize selected subsets of the solution while preserving the surrounding structure.

CP13. This strategy probabilistically selects a single high-penalty item and removes it from its current bin. A CP model then determines whether the item should be inserted into an existing bin or placed in a newly created bin, subject to capacity and scheduling constraints, with the objective bounded by the current solution value.

CP30. This strategy selects a small contiguous block of bins, extracts all items within that block, and reassigns them jointly using a CP model while fixing the preceding and succeeding bins. This enables coordinated reassignment of multiple items with flexible bin completion times inside the block.

CP50. This strategy attempts to merge two randomly selected bins if their combined load does not exceed capacity. If feasible, all items are merged into a single bin and the schedule is re-optimized. This operator primarily targets bin count reduction and structural simplification.

All CP-based strategies return a modified assignment that is guaranteed to be capacity-feasible if accepted. The resulting assignment is subsequently passed to OPT_ALL for full local optimization before pool evaluation.

5.8 Solution Pool Management

This component maintains a bounded *solution pool* during the post-construction diversification phase. The pool stores a small set of candidate solutions (assignments) together with run-time metadata, enabling the diversification loop to (i) restart from previously found good solutions, and (ii) keep only the best candidates seen so far under a fixed memory budget.

In the implementation (in `Test.cs`), the pool size is a user parameter `poolSize` (default 1), and the pool is represented as a list of tuple records:

$$\text{poolEntries} = \{(\text{iter}, \text{strategy}, \text{startTime}, \text{timeSec}, \text{startAss}, \text{startObj}, \text{startBins}, \text{solAss}, \text{solObj}, \text{solBins})\}.$$

Each record stores both a *starting* assignment snapshot (startAss) and the resulting *solution* assignment (solAss) after applying one diversification strategy and re-optimising with OPT_ALL. The pool is seeded with the best solution obtained at the end of the construction phase (prior to diversification), so there is always at least one candidate available to start from.

5.8.1 Pool Initialisation

At the start of diversification, the current assignment produced by the construction pipeline is cloned and inserted as the initial pool entry (iter $= 0$, strategy $= 0$). This ensures the pool is never empty and provides a baseline candidate for subsequent diversification runs.

Algorithm 10: INITSOLUTIONPOOL$(x_0, z_0, b_0, t_{\text{sec}})$

Input: current assignment x_0; objective z_0; bins used b_0; elapsed seconds t_{sec}
Output: pool list $\mathcal{P}$ (list of entries)

$\mathcal{P} \leftarrow$ empty list
Insert into $\mathcal{P}$ the entry $(0, 0, \text{runStart}, t_{\text{sec}}, \text{clone}(x_0), z_0, b_0, \text{clone}(x_0), z_0, b_0)$
return $\mathcal{P}$

5.8.2 Starting-Solution Selection

On each diversification iteration, a starting solution is selected from the pool using a helper routine PICKWEIGHTEDINDEX($\mathcal{P}$, rng, favorLowObjective $=$ true), and the candidate assignment is cloned from the chosen entry's solAss. The exact weighting scheme is encapsulated inside `ItemsHelper.PickWeightedIndex` and therefore is not expanded here; the observable behaviour in `Test.cs` is that the selection routine is invoked with favorLowObjective set to true.

5.8.3 Pool Update Rule (Bounded Capacity)

Let poolSize denote the maximum allowed number of entries. After generating a new candidate solution $\hat{x}$ (post-OPT_ALL) with objective $\hat{z}$, the pool update proceeds as follows:

- If $|\mathcal{P}| <$ poolSize, the new entry is appended.
- Otherwise, identify the index of the *worst* pool entry (largest solObj). Replace that worst entry *only if* the new objective is strictly better, i.e., $\hat{z} < z_{\text{worst}}$.

No entries are explicitly deleted via list removal; when full, the pool evolves only by *overwriting* the current worst entry under strict improvement. As a result, the pool acts as a bounded "elite set" ranked purely by objective value (solObj), with ties not triggering replacement.

Algorithm 11: UPDATESOLUTIONPOOL($\mathcal{P}$, poolSize, e_{new})

Input: pool list $\mathcal{P}$; capacity poolSize; new entry e_{new} with objective e_{new}.solObj
Output: updated pool list $\mathcal{P}$

if $|\mathcal{P}| <$ *poolSize* **then**
  append e_{new} to $\mathcal{P}$
  return $\mathcal{P}$
worstIdx $\leftarrow \arg\max_{j \in \{1,\ldots,|\mathcal{P}|\}} \mathcal{P}[j]$.solObj
if e_{new}.*solObj* $< \mathcal{P}$[*worstIdx*].*solObj* **then**
  $\mathcal{P}$[worstIdx] $\leftarrow e_{\text{new}}$
return $\mathcal{P}$

5.8.4 Best-Solution Extraction

After each pool update, the current best solution is extracted as the entry with minimum solObj and stored as the incumbent best assignment. At the end of the diversification time budget, the final output of the diversification phase is the best pool entry found.

Algorithm 12: GETBESTFROMPOOL($\mathcal{P}$)

Input: pool list $\mathcal{P}$
Output: best entry $e^\star$

$e^\star \leftarrow \arg\min_{e \in \mathcal{P}} e$.solObj
return $e^\star$

5.8.5 Overview of Solution Pool

The implemented pool logic provides a simple and deterministic capacity control mechanism: it preserves up to poolSize candidate solutions and only replaces the current worst objective value when a strictly better solution is produced. The pool is also used as a restart reservoir by selecting starting points via PICKWEIGHTEDINDEX (called with favorLowObjective set to true), enabling the diversification loop to revisit and perturb good solutions rather than always restarting from the single current incumbent.

5.9 Logging and Traceability Component

5.9.1 Log Structure

The implementation includes an explicit logging subsystem to support offline analysis, experimental reporting, and traceability of algorithmic decisions. Logging is implemented in `Test.cs` and `CsvHelper.cs`, and each execution produces a dedicated comma-separated values (CSV) log file.

A timestamped log file named `run_YYYYMMDD_HHMMSS.csv` is created at program start and stored under a fixed root directory. The file is initialized immediately with a predefined header structure to ensure that partial progress is preserved even if the run terminates early. During execution, the log file is repeatedly overwritten with updated content to reflect the current state of the run.

The CSV file is organised into three logical blocks:

1. **Run-level summary.** Records the total elapsed wall-clock time since the start of the run.
2. **Initial-solution snapshot.** Stores the assignment produced before diversification, together with its objective value, number of bins, and the time required to generate it.
3. **Diversification records.** Contains one row per solution-pool entry, capturing the outcome of each diversification iteration.

5.9.2 Recorded Metrics

For each diversification iteration, the following metrics are recorded as a single CSV row:

- iteration index;
- elapsed runtime in seconds since parameter initialisation;
- identifier of the diversification strategy applied;
- wall-clock timestamp at which the iteration started;
- starting assignment vector and its objective value and bin count;
- resulting assignment vector after re-optimisation;
- resulting objective value and number of bins used.

Assignment vectors are stored as space-separated integer sequences, where each entry denotes the bin index assigned to the corresponding item in the current subproblem ordering. Objective values correspond to the total weighted earliness-tardiness penalty computed by the LP evaluation model.

In addition to per-iteration records, the log also stores the pre-diversification solution and the cumulative runtime, enabling separation of construction cost from diversification cost during post-run analysis.

5.9.3 Reproducibility Considerations

Reproducibility is supported through explicit recording of all sources of algorithmic non-determinism and timing. Each run logs the exact assignment vectors, objective values, diversification strategies, and elapsed times, allowing the full search trajectory to be reconstructed offline.

All stochastic decisions are driven by a single pseudo-random number generator initialized from a user-specified or automatically generated seed. Because the same seed governs all biased selection, perturbation, and diversification operations, re-running the algorithm with identical inputs and parameters reproduces the same sequence of logged solutions.

5.10 Random Seeding and Time-Budget Control

This section describes how the implementation controls stochastic behaviour through explicit random seeding and enforces a user-defined total runtime through a structured time-budget mechanism. Both mechanisms are implemented directly in `Test.cs` and serve two primary purposes: ensuring reproducibility for experimental evaluation, and enabling practical deployment under explicit computational time constraints.

5.10.1 Random Seed Initialisation and Usage

Initialisation. At program startup, the user is prompted to provide a random seed. If the input parses to a valid integer, that value is used directly; otherwise (empty or invalid input), the seed defaults to `Environment.TickCount`. The chosen seed is printed to the console and used to initialise a single pseudo-random number generator:

$$\texttt{rng} \leftarrow \texttt{new Random(seed).}$$

This single `rng` instance is passed to all stochastic components, ensuring that the entire run is governed by one consistent random stream.

Why explicit seeding matters. The algorithm relies on biased and randomized decisions during both solution construction and diversification. Without a fixed seed, repeated runs on the same instance may follow different random trajectories and yield different solutions. Allowing a user-specified seed enables exact reproducibility, which is essential for scientific reporting and controlled experimentation. At the same time, allowing an automatic seed supports exploratory or production runs where solution diversity is desirable.

Where the seed is used. In the implementation, `rng` influences all stochastic logic, including:

- biased selection of medium-fitness items for the initial two-bin seed;
- biased item selection during iterative medium-bin construction;
- biased selection of non-medium items during incremental insertion;
- weighted selection of starting solutions from the solution pool;
- random choice of diversification strategies and their internal perturbation operators (e.g., swaps, reinsertion, bin removal).

Algorithm 13: INITRANDOMSEED()

Output: random generator `rng`; seed value `seed`

Prompt user for seed input
if *input parses to integer* `seed` **then**
  use parsed value
else
  `seed` ← `Environment.TickCount`
Print chosen seed
`rng` ← `new Random(seed)`
return `rng`, `seed`

5.10.2 Total Runtime Input and Budget Allocation

User-defined total runtime. The program prompts the user to specify a desired *total runtime in seconds*. If the input parses to a positive real value, it is converted into a `TimeSpan`:

`targetTotalRuntime` ← `TimeSpan.FromSeconds(runtimeSeconds).`

Otherwise, the target runtime is set to `TimeSpan.Zero`, indicating that no explicit time budget is enforced.

Rationale for a total runtime budget. In industrial and operational settings, optimization is typically executed under strict time limits (e.g., overnight planning windows, limited compute allocations, or real-time decision-making). By accepting a total runtime constraint rather than separate limits for individual phases, the implementation mirrors real-world usage: it always produces a feasible solution as quickly as possible, and then invests any remaining time into solution improvement.

5.10.3 Budget Split Between Construction and Diversification

The implementation treats the user-specified runtime as a *global* budget for the entire algorithm. A stopwatch (`programStart`) is started at the beginning of execution and remains active throughout the complete construction pipeline, including:

- two-bin medium seed generation,

- iterative medium-bin creation,
- non-medium item insertion,
- and the final OPT_ALL improvement.

Immediately before entering the diversification phase, the elapsed time is recorded as:

$$\texttt{preDiversificationDuration} \leftarrow \texttt{programStart.Elapsed}.$$

This value represents the wall-clock time required to produce the initial solution *without diversification*. The stopwatch is then stopped, and the remaining time (if any) is allocated to diversification:

$$\texttt{diversificationDuration} \leftarrow \texttt{targetTotalRuntime} - \texttt{preDiversificationDuration}.$$

If this subtraction yields a negative value, the diversification budget is clamped to zero and the diversification phase is skipped entirely.

Algorithm 14: COMPUTEDIVERSIFICATIONBUDGET

Input: total runtime `targetTotalRuntime`; stopwatch `programStart`
Output: `preDiversificationDuration`, `diversificationDuration`

```
preDiversificationDuration ← programStart.Elapsed
Stop programStart
diversificationDuration ← TimeSpan.Zero
if targetTotalRuntime > TimeSpan.Zero then
    diversificationDuration ← targetTotalRuntime −
     preDiversificationDuration
    if diversificationDuration < TimeSpan.Zero then
        diversificationDuration ← TimeSpan.Zero
return preDiversificationDuration, diversificationDuration
```

5.10.4 Enforcing the Diversification Budget

If a positive diversification budget is available,
a dedicated stopwatch (`diversificationStopwatch`) is started and diversification iterations are executed while its elapsed time remains below the allocated budget:

$$\texttt{diversificationStopwatch.Elapsed} < \texttt{diversificationDuration}.$$

If no time remains, the diversification phase is skipped and the constructed solution is returned immediately.

Algorithm 15: RUNDIVERSIFICATIONWHILETIMEREMAINS

Input: `diversificationDuration`

if *diversificationDuration* $\leq$ *TimeSpan.Zero* **then**
  Print "No time budget left for diversification; skipping"
  return

Start `diversificationStopwatch`

while *diversificationStopwatch.Elapsed* $<$ *diversificationDuration* **do**
  Perform one diversification iteration

Logging clock. A separate stopwatch (`autoRunClock`) is started immediately after user parameters are read. This clock is used exclusively for timestamping log entries (e.g., solution-pool records) and does not influence runtime control or budget enforcement.

6 Proposed Optimization Framework

6.1 Framework Overview

The proposed optimization framework follows a structured *construct–improve–diversify* paradigm designed to address the multi-component nature of the bin cutting and packing problem.

The construction phase prioritizes medium-fitness items, which represent the most challenging trade-offs between size and penalty sensitivity. A small medium-only seed is first constructed using biased randomization and local optimization, after which the remaining medium items are incorporated through iterative bin creation and re-optimization. This produces a stable medium-fitness backbone that captures the dominant penalty structure of the instance.

Once all medium-fitness items are placed, non-medium items are inserted incrementally. These items are selected using a composite score that balances penalty magnitude and asymmetry, favoring insertions that are both impactful and robust to scheduling changes. Periodic re-optimization ensures that insertion decisions remain consistent with the global objective.

After a complete feasible assignment is obtained, the framework enters a diversification phase. Here, stochastic perturbation strategies are applied to solutions drawn from a bounded elite pool, followed by exact local improvement. The time spent in diversification is controlled by a global runtime budget, ensuring that solution construction is always prioritized while allowing additional exploration when time permits.

Together, these phases integrate biased randomization, exact evaluation, and structured diversification into a coherent optimization framework capable of producing high-quality solutions under practical computational constraints.

6.2 Initialization Phase

The initialization phase prepares the data structures and initial restricted instance used by the construction pipeline. First, the framework computes item fitness values and partitions items into fitness bands as described in Section 5.1. The medium-fitness set I_{med} is then sorted using the stable due-date ordering (Section 5.3.1) to ensure deterministic subproblem construction.

Next, a small seed subset $S \subseteq I_{\text{med}}$ is sampled using biased randomization (Section 5.2) and mapped into two initial bins, producing a two-bin seed assignment. The seed assignment is embedded into a restricted subproblem built using the index-mapping and subproblem-construction mechanisms described in Section 5.3 and Section 5.4.

The output of initialization is a feasible (or safeguarded) two-bin medium-only assignment that serves as the starting point for the medium-fitness expansion phase.

6.3 Medium-Fitness Item Construction Phase

6.3.1 Two-Bin Biased Seed Construction

This step instantiates the first feasible structure of the solution on the medium-fitness subset. A biased sample of medium items is selected and assigned to two bins using the bin-map mechanism described in Section 5.2.2. The resulting two-bin assignment is improved using OPT_ALL (Section 5.5) and validated using the feasibility safeguards in Section 5.4.3. The best feasible assignment becomes the initial medium-only backbone used for subsequent expansion.

6.3.2 Iterative Bin Creation and Expansion of Medium-Fitness Items

After the initial two-bin medium-fitness solution is constructed and locally improved, there typically remain medium-fitness items that have not yet been assigned. These remaining items are incorporated through an iterative bin creation procedure that progressively expands the medium-only assignment.

Let I_{med} denote the full set of medium-fitness items and let π_{med} be the ordered index list of medium items already included in the current restricted subproblem. At each iteration, a new bin is created and filled by selecting items from the remaining set $I_{\text{med}} \setminus \pi_{\text{med}}$.

Items are first ranked by decreasing normalized fitness ($\hat{f}_i$), so that more troublesome medium items are considered earlier. Rather than selecting items greedily, the framework applies the biased index selection mechanism (Section 5.2.1) to this ranking. This introduces controlled randomness while maintaining a strong bias towards high-impact items.

Items are added to the new bin as long as capacity allows. To prevent excessive sampling once the remaining capacity becomes small, the filling procedure is terminated after a small number of consecutive failed attempts or when no remaining item can fit. The filled bin is then merged with the existing medium assignment.

After each bin is added, the restricted subproblem is rebuilt using stable due-date sorting, the current assignment is remapped to the new index order, and the expanded solution is re-optimised using OPT_ALL. This process repeats until all medium-fitness items are assigned.

Algorithm 16: EXPANDMEDIUMFITNESSITEMS

Input: instance p; medium set I_{med}; current assignment x; index list π; RNG `rng`

while $\pi \neq I_{med}$ **do**
- $R \leftarrow I_{\text{med}} \setminus \pi$
- Rank R by decreasing normalized fitness
- Initialize empty bin B
- **while** *capacity allows and attempts remain* **do**
  - pick index via BIASEDPICKINDEX
  - **if** *item fits* **then**
    - add item to B
- $\pi \leftarrow \pi \cup B$
- rebuild subproblem and remap assignment
- apply OPT_ALL

return updated assignment x

6.4 Non-Medium Item Insertion Phase

6.4.1 Item Ranking and Selection

After all medium-fitness items are placed, the framework inserts the remaining items from the small- and large-fitness bands. These items are generally easier to accommodate due to either small size or low penalty sensitivity, and are therefore inserted into the partially filled assignment produced by the medium-fitness construction phase.

To guide insertion, each non-medium item i is assigned a composite score that balances penalty magnitude and penalty asymmetry. First, define the normalized penalty magnitude

$$\widehat{p}_i = \text{norm}(\max(\epsilon_i, \tau_i)),$$

and the normalized penalty asymmetry

$$\widehat{\delta}_i = \text{norm}\left(\frac{|\epsilon_i - \tau_i|}{\max(\epsilon_i, \tau_i)}\right),$$

where $\text{norm}(\cdot)$ denotes min–max normalization over all non-medium items.

The final non-medium score is computed as

$$s_i = \alpha\widehat{p}_i + (1-\alpha)(1-\widehat{\delta}_i), \tag{2}$$

with $\alpha \in [0, 1]$ (set to 0.9 in the implementation).

The first term in (2) prioritizes items with high penalty impact, while the second term favors items whose earliness and tardiness penalties are relatively balanced. Such items tend to behave more stably when their completion times change, as they do not strongly prefer being early or late.

As a result, biased selection based on s_i promotes non-medium items that can be inserted with a lower risk of causing large, unexpected penalty spikes. This “less surprise” effect makes the incremental insertion process more robust, even though it remains stochastic.

6.4.2 Incremental Insertion and Re-Optimization

Non-medium items are inserted incrementally in small batches. In each insertion round, a subset of items is selected using biased index selection applied to the ranking induced by s_i. Each selected item is inserted using a capacity-feasible first-fit rule: the item is assigned to the earliest bin that can accommodate it, or to a new bin if necessary.

After a batch of items is inserted, the restricted subproblem is rebuilt and the assignment is re-optimized using OPT_ALL. This batching strategy reduces computational overhead while allowing the LP-based evaluation model to correct suboptimal placement decisions introduced during insertion.

Algorithm 17: INSERTNONMEDIUMITEMS

Input: instance p; current assignment x; non-medium set I_{non}; parameter α; RNG `rng`

Compute scores s_i for all $i \in I_{\text{non}}$
Rank items by decreasing s_i
while *uninserted items remain* **do**
    select batch using BIASEDPICKINDEX
    insert each item using first-fit (open new bin if needed)
    rebuild subproblem and remap assignment
    apply OPT_ALL
return updated assignment x

6.5 Diversification Search Phase

The diversification phase attempts to escape local optima after a complete feasible assignment has been constructed. The remaining runtime available for diversification is computed using the global time-budget mechanism described in Section 5.10. If no budget remains, the framework skips diversification and reports the constructed solution.

Otherwise, the framework initializes a bounded solution pool (Section 5.8) and iteratively applies one diversification strategy per iteration (Section 5.7), followed by OPT_ALL re-optimization. The pool update rule retains only solutions that improve upon the current worst pool entry.

6.5.1 Time Budget Allocation

Let T denote the user-defined total runtime. The framework measures the time spent in construction and assigns the remaining time to diversification, as described in Section 5.10.3. Diversification iterations are executed while the elapsed diversification time remains below the allocated budget (Section 5.10.4).

7 Results and Discussion

7.1 Parameter Definitions and Notation

Tables 1 and **??** report experimental results using abbreviated parameter names to improve readability. The meaning of each column and configuration option is summarised below.

Instance. Number of items in the problem instance (problem size). Values such as 60 and 100 indicate instances containing 60 and 100 items, respectively. Every instance is solved repeatedly using identical parameter settings but different random seeds. This allows the stochastic behaviour of the algorithm to be evaluated, with per-run results recorded and aggregated statistics reported.

Thr (Fitness Threshold). Threshold used for partitioning normalized fitness values into small, medium, and large bands.

OPT_ALL (Bin Handling Policy). Variant of the OPT_ALL local improvement procedure:

- **Preserve**: empty bins are retained (bin indices are preserved);
- **Remove**: empty bins are removed and indices are compressed.

Band Split (Fitness Band Policy). Fitness measure used for band partitioning:

- **W/maxPen**: weight divided by maximum penalty, $\frac{w_i}{\max(\epsilon_i,\tau_i)}$;
- **MinPen/W**: minimum penalty divided by weight, $\frac{\min(\epsilon_i,\tau_i)}{w_i}$.

Batch (Non-medium Insertion Batch Size). Maximum number of non-medium items inserted in each incremental insertion round before re-optimization.

FailLim (Medium Bin Fill Failure Limit). Maximum number of consecutive failed insertion attempts allowed when filling a new medium-fitness bin before terminating the filling process.

Div (Diversification Enabled). Indicates whether the diversification phase is executed:

- **On**: diversification strategies are applied;
- **Off**: diversification phase is skipped.

TotalTime. User-specified total runtime budget (in seconds) for the entire algorithm.

NonDiv. Elapsed runtime (in seconds) spent in the construction and local improvement phases before diversification.

DivTime. Elapsed runtime (in seconds) allocated to the diversification phase.

TimeObj. Time stamp when the first assignment with best objective value in a run was found.

Obj (Objective Value). Final weighted earliness–tardiness penalty computed by the LP evaluation model. Best objective value (i.e. lowest objective value) in a run is recorded in the table.

Bins. Number of bins used in the final assignment.

Seed. Random seed used to initialise the pseudo-random number generator for the run.

7.2 Research best objectives

20 items: (1h) 18658 — (30mins) 18658 — (10mins) 18658
40 items: (1h) 60378 — (30mins) 60398 — (10mins) 60398
60 items: (1h) 113872 — (30mins) 114432 — (10mins) 116846

7.3 Per-run table

Instance	Thr	OPT_ALL	Batch	FailLim	TotalTime	NonDiv	DivTime	Obj	Bins	Seed
20	0.1	Remove	4	3	268	11	257	18658	8	174016
20	0.1	Remove	4	3	604	11	594	18858	8	174513
20	0.1	Remove	4	3	464	48	416	18658	8	180104
20	0.1	Remove	4	3	380	12	462	18658	8	180915
20	0.2	Remove	4	3	240	9	231	18658	8	212613
20	0.2	Remove	4	3	616	45	571	18858	8	213032
20	0.2	Remove	4	3	551	11	540	18658	8	214331
20	0.2	Remove	4	3	124	14	110	18658	8	185822
40	0.1	Remove	4	3	874	31	843	60643	10	181618
40	0.1	Remove	4	3	614	28	586	60398	11	183146
40	0.1	Remove	4	3	765	31	734	60643	10	184345
40	0.1	Remove	4	3	706	26	680	61013	12	185822
40	0.2	Remove	4	3	840	26	814	60398	11	215534
40	0.2	Remove	4	3	470	35	435	60398	11	221018
40	0.2	Remove	4	3	86	24	62	60378	10	221814
40	0.2	Remove	4	3	576	26	550	61231	10	221946
60	0.1	Remove	4	3	749	350	399	129981	21	122622
60	0.1	Remove	4	3	918	325	593	126645	21	124121
60	0.1	Remove	4	3	1049	521	728	118896	24	125652
60	0.1	Remove	4	3	1044	582	462	124586	21	154547

Table 1: Per-run results showing parameter settings and outcomes.

7.4 Discussion

Overall, the experimental results demonstrate that the improved framework developed in Trimester 3 consistently outperforms the earlier Trimester 1 approach. The previous framework relied on random greedy construction, simple hill-climbing local search, and limited diversification. In contrast, the current framework integrates fitness-based item classification, structured subproblem construction, exact LP-based evaluation, and multiple coordinated diversification operators. These enhancements enable the algorithm to generate higher-quality initial solutions and to escape local optima more effectively, resulting in more stable and competitive performance across problem sizes.

7.4.1 Fitness Threshold Experiments

The impact of the fitness threshold was examined using values of 0.1 and 0.2. For both 20- and 40-item instances, the results indicate that the choice of threshold has only a minor influence on the final objective values. In most cases, both settings were able to reach the research best objectives with similar consistency.

For the 60-item instances, a threshold of 0.1 was selected for subsequent experiments. This lower threshold classifies a larger proportion of items as medium fitness, thereby encouraging the construction procedure to consider more potentially troublesome items during the early stages. This leads to more informed initial assignments and provides a stronger foundation for subsequent improvement.

7.4.2 Effect of Fitness Banding

The fitness banding mechanism plays a key role in guiding the construction process. By separating items into small, medium, and large fitness categories, the algorithm prioritizes difficult items during initial bin formation and incremental expansion.

The results show that this structured handling of items contributes to the consistent attainment of high-quality solutions for smaller and medium-sized instances. Compared to the earlier framework, which treated all items more uniformly, the current approach benefits from a more targeted and systematic construction strategy.

7.4.3 Effect of Diversification

Diversification has a clear and positive impact on solution quality. In all tested configurations, the best objective value obtained after diversification is lower than that of the pre-diversification solution. This confirms that the diversification phase successfully enables the search to escape local optima.

Furthermore, the use of multiple diversification strategies allows the algorithm to explore the solution space from different perspectives. Operators such as item reinsertion, bin swapping, and penalty-based removal modify the assignment in complementary ways. This coordinated approach maintains diversity and prevents premature convergence more effectively than relying on a single perturbation mechanism.
The results indicate that combining several diversification methods leads to more reliable long-term improvement and better overall robustness.

7.5 Runtime Versus Solution Quality

For the 20-item instances, the algorithm consistently reached the research best objective values reported in the literature for 10-minute, 30-minute, and 1-hour time limits. In particular, the best value of 18 658 was frequently achieved in under 10 minutes, and in one run in slightly over one minute. This demonstrates the strong potential of the framework to rapidly identify near- optimal solutions.

For the 60-item instances, the algorithm has not yet matched the published 10-minute best objective value, remaining approximately 6 000 units above the reference benchmark. However, it consistently produces high-quality solutions within limited runtime, indicating good robustness and stability. The

remaining performance gap suggests that further refinement of diversification intensity or using methods that require less memory usage and computational time is needed.

For the 40-item instances, the algorithm consistently reached the research best objective values reported for the 10-minute and 30-minute time limits (approximately 60 398) within relatively short runtimes. In several runs, these values were obtained well before the allocated time budget was exhausted, demonstrating rapid convergence to high-quality solutions.

Moreover, in some runs the algorithm approached or matched the reported 1-hour best objective value (60 378) within only a few minutes. This indicates that the framework is capable of discovering very strong solutions at an early stage of the search. However, this behaviour is not yet fully consistent across all runs. While some executions rapidly converge to near-optimal solutions, others require substantially longer exploration before achieving comparable quality.

This variability suggests that the current framework exhibits a degree of instability in its long-term search dynamics. Nevertheless, the frequent early discovery of high-quality solutions highlights the strong potential of the proposed approach, and indicates that further improvements in diversification control and adaptive parameter tuning may enhance its reliability.

Overall, the framework demonstrates strong performance in rapidly generating competitive solutions and maintaining consistent quality across repeated runs. Its main strengths lie in its effective initial construction, powerful local improvement, and coordinated diversification. While performance on larger instances remains improvable, the results indicate substantial progress over earlier approaches and provide a solid foundation for further enhancement.

8 Conclusion and Future Work

8.1 Summary of Contributions

This research investigated hybrid and decomposition-based optimization techniques for multi-component bin cutting and packing problems with earliness–tardiness penalties. A structured *construct–improve–diversify* framework was developed to address the compounded nature of the problem and its strong interdependence between packing and scheduling decisions.

The proposed framework integrates several key components, including fitness-based item classification, biased randomized construction, stable subproblem management, exact LP-based evaluation, coordinated local improvement, and a bounded elite solution pool. By prioritizing medium-fitness items during construction and combining multiple diversification strategies, the algorithm is able to generate high-quality initial solutions and systematically explore the solution space.

Extensive computational experiments demonstrate that the proposed approach consistently outperforms the earlier Trimester 1 framework. In particular, the new framework achieves research best objective values for small and medium-sized instances within short runtimes, and produces robust and competitive solutions for larger instances. These results confirm the effectiveness of combining structured construction with controlled stochastic exploration for multi-component optimization problems.

Overall, this work provides a practical and extensible optimization framework that improves both solution quality and stability, and contributes a detailed implementation and experimental analysis to the study of hybrid approaches for complex cutting and packing problems.

8.2 Limitations

Despite its strong performance, the proposed framework exhibits several limitations. First, the algorithm relies on a number of user-defined parameters, including fitness thresholds, batch sizes, diversification intensity, and pool capacity. Although reasonable values were identified experimentally, these parameters are currently fixed and may not be optimal for all instance types.

The computational cost of the algorithm seems to be high. As instance size increases, evaluation overhead becomes a more significant bottleneck, reducing the time available for exploratory search.

The stochastic nature of the algorithm leads to variability between runs. Although reproducibility is ensured through explicit random seeding, some executions converge rapidly to near-optimal solutions while others require longer exploration. This instability suggests that the balance between intensification and diversification is not yet fully optimized.

8.3 Future Research Directions

Additional or hybrid diversification operators could be developed. Combining heuristic perturbations with more advanced constraint programming or large neighbourhood search techniques may enable deeper escapes from local optima, particularly for larger instances.
In summary, this research establishes a strong foundation for hybrid optimization of multi-component cutting and packing problems. With continued refinement and extension, the proposed framework has significant potential to serve as an effective and scalable tool for complex industrial optimization tasks.

References

[1] S. Desale, A. Rasool, and S. Andhale, *Heuristic and meta-heuristic algorithms and their relevance to the real world: A survey*, International Journal of Computer Science and Information Technologies, 6 (2015), pp. 1005–1010.

[2] P. Hansen, N. Mladenović, and J. A. M. Pérez, *Variable neighborhood search: Basics and variants*, EURO Journal on Computational Optimization, 4 (2016), pp. 1–30.

[3] H. H. Hoos and T. Stützle, *Stochastic Local Search: Foundations and Applications*, Morgan Kaufmann, 2005.

[4] Y. Hu, L. Zhang, and J. Wang, *Combination of simulated annealing algorithm and minimum horizontal line algorithm to solve two-dimensional pallet loading problem*, in Proceedings of the Winter Simulation Conference, 2022, pp. 1–12.

[5] M. Iori, S. Martello, and M. Monaci, *Exact solution techniques for two-dimensional cutting and packing problems: A survey*, European Journal of Operational Research, (2021).

[6] S. Polyakovskiy, V. A. Strusevich, and Y. N. Sotskov, *Just-in-time batch scheduling subject to batch size*, in Proceedings of the Genetic and Evolutionary Computation Conference, 2020, pp. 1377–1385.

[7] G. R. Raidl, *Decomposition based hybrid metaheuristics*, European Journal of Operational Research, (2015).

[8] S. Sultana, M. M. Rahman, and M. M. Hassan, *Avoiding premature convergence to local optima with adaptive exploration for genetic algorithms*, Applied Soft Computing, (2025).

[9] Z. Wang, X. Li, and W. Liu, *Minimizing total tardiness in the distributed flowshop group scheduling problem with an iterated greedy algorithm*, IEEE Conference on Control, Decision and Information Technologies, (2022), pp. 1–6.