Comparative Analysis of Metaheuristic Algorithms for the Bin Packing Problem

A Final Report

Supervisor:

Dr. Ziarati

Atefe Rajabi

40230563

Summer 2024

Index

Introduction

The bin packing problem is a classic optimization problem with significant practical applications, from logistics and manufacturing to resource allocation in computing. It involves efficiently packing a set of items of varying sizes into a finite number of bins, minimizing the number of bins used. This problem, known to be NP-hard, has been extensively studied and tackled using various algorithms, including both exact and heuristic methods.

In this report, we explore and compare several metaheuristic algorithms for solving the bin packing problem, focusing on both single-solution based approaches and population-based approaches. The algorithms evaluated include Simulated Annealing (SA), Tabu Search (TS), Greedy Randomized Adaptive Search Procedure (GRASP), Genetic Algorithms (GA), Ant Colony Optimization (ACO), and Particle Swarm Optimization (PSO). Each of these algorithms brings unique strategies for exploring the solution space and optimizing bin packing configurations.

Problem Definition

The bin packing problem can be formally defined as follows: given a set of items, each with a specific size, and a set of bins, each with a fixed capacity, the goal is to pack all the items into the minimum number of bins without exceeding the capacity of any bin. Mathematically, if we denote the set of items $I = \{i_1, i_2, \dots, i_n\}$ and the capacity of each bin as C, the objective is to minimize the number of bins B such that:

$$\sum_{i \in b_j} s_i \leq C \quad \forall b_j \in B$$

where s_i is the size of item i, and b_j is the set of items packed in bin j.

The primary objective of this report is to compare the performance of these metaheuristic algorithms in solving the bin packing problem. We aim to evaluate their efficiency in terms of solution quality and computational time, providing insights into their suitability for different types of bin packing instances.

By understanding the strengths and weaknesses of each algorithm, we can better select and tailor optimization strategies for practical applications where efficient bin packing is crucial.

Execution Environment

The algorithms were executed on a system with the following specifications:

Processor: Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz

Memory: 8 GB RAM

This setup ensures that the computational resources are consistent across all algorithm executions, providing a fair comparison of their performance in terms of both solution quality and computational time. The screenshot of the task manager shows the CPU and memory usage during the execution of the algorithms, confirming the utilization of the available resources.

The source code for single-solution based algorithms (Simulated Annealing, Tabu Search, and GRASP) and population-based algorithms (GA, PSO, and ACO) was meticulously developed and tested to ensure it efficiently handles the challenges of the bin packing problem. This source code is provided on GitHub.

Neighborhood Definition

The Neighborhood function is crucial in optimization processes like simulated annealing because it directly influences the exploration and exploitation mechanisms within the solution space. Here's a breakdown of each tweak provided by the Neighborhood function and why such a function is essential:

1. Single Item Move: This Tweak attempts to move an item from one bin to another that has sufficient capacity. The item and bin are selected randomly.

2. Swap: This Tweak attempts to swap two items between two different bins. The swap only occurs if both bins have sufficient capacity to accommodate the swapped items. This could help in optimizing space usage across bins.

3. Merge Bins: This function tries to merge the contents of one bin into another if there's enough capacity. This could reduce the total number of bins used, thereby optimizing the solution.

4. Item Reinsertion: Insert an item into a new bin.

Adaptive Selection Mechanism[1]

The adaptive selection mechanism uses rewards and penalties to guide the selection of tweaks during the optimization process. The method is designed to prefer tweaks that have historically led to improvements and to avoid those that frequently result in worse solutions. Here's how it works:

1. Initialization and Updating of Probabilities:

   - Each tweak type is assigned a probability of being selected, which is initially set equally across all tweaks.

   - After each application of a tweak, rewards and penalties are updated based on the tweak's effectiveness. Rewards are given for improvements in the solution, and penalties are given for non-improvements or worsening of the solution.

2. Reassign Probabilities:

   - The probabilities of selecting each tweak are recalculated based on the recent history of rewards and penalties.

   - This is done by computing a value for each tweak using the formula:

$$Value = \frac{rewards}{rewards+penalties}$$

   - These values are then normalized to ensure they sum to one, forming a new set of probabilities that guide the future selection of tweaks.

   - If a probability is zero (indicating consistent failure of a tweak), it is adjusted to a small positive value to ensure that no tweak is ever completely excluded, allowing for exploration.

3. Selection of Tweak:

   - When a tweak is to be selected, a random value is generated and used to select a tweak probabilistically based on the cumulative distribution of the tweak probabilities.

   - This selection is inherently adaptive as it favors tweaks that have been more successful but still allows for exploration of less successful tweaks.

[1] Inspired by "Xue, Y., Zhu, H., Liang, J., & Słowik, A. (2021). Adaptive crossover operator based multi-objective binary genetic algorithm for feature selection in classification. *Knowledge-Based Systems, 227*, 107218. https://doi.org/10.1016/j.knosys.2021.107218

Tweak Selection Probability Trends Over Iterations:

iteration	swap	Single item move	Merge bins	Item reinsertion
1	0.25	0.25	0.25	0.25
2	0.201054	0.265645	0.529983	0.00331763
3	0.201268	0.26581	0.529602	0.0033202
4	0.201326	0.266098	0.529252	0.00332431
20	0.220326	0.290568	0.484864	0.00424135
21	0.220561	0.290677	0.484508	0.00425474
98	0.283306	0.29889	0.408853	0.00895172
99	0.333303	0.348661	0.309069	0.008967
100	0.333295	0.348718	0.309007	0.008980

From the table of tweak selection probabilities, it is evident that in the early stages, where a random initialization often results in a higher number of bins, the 'Merge bins' tweak is more successful compared to 'Single item move' or 'Swap'. However, as the solution configuration becomes more compact over time, merging bins becomes less efficient due to limited space for successful merges. Instead, the combination of 'Swap' and 'Single item move' becomes more effective at reducing the objective function value of the solution. This demonstrates not only the effectiveness of the adaptive tweak strategy but also the logical basis for employing this combination of tweaks as the search progresses.

The use of adaptive tweaks in metaheuristic algorithms, particularly for complex optimization problems like bin packing, brings additional strategic depth through the varied nature of the tweaks themselves. Each tweak—whether it's splitting bins, merging them, or simply moving items between them—offers unique ways to explore and exploit the solution space.

1. Diverse Exploration Strategies

- Each tweak type inherently manipulates the solution structure in a different way, providing varied methods of exploration:

- Example: Splitting Bins: A tweak like splitting a bin can temporarily worsen the current solution by increasing the number of bins used. However, this act creates new configurations by redistributing items across more bins, potentially uncovering new and better arrangements that were not possible in a fewer-bin setup. This can be particularly useful in scenarios where bins may be overfilled or where the distribution of item sizes makes single-bin solutions suboptimal.

2. Potential for Superior Long-Term Solutions

- The initial degradation in solution quality can lead to superior solutions in the long term:

- Indirect Benefits: While the immediate effect of some tweaks might seem detrimental (e.g., increasing the number of bins), the rearrangement can facilitate further beneficial tweaks that were not previously feasible, leading to a better overall solution. For instance, splitting bins might allow subsequent tweaks to more efficiently pack items, reducing the total number of bins eventually needed.

Single Solution Base

Single solution-based metaheuristic algorithms focus on iteratively improving a single candidate solution to an optimization problem. Unlike population-based algorithms, which manage multiple solutions simultaneously, single solution-based approaches refine one solution at a time. This method is beneficial for exploring deep and complex search spaces thoroughly using one evolving solution pathway.

Here are some prominent single solution-based metaheuristic algorithms:

1. Simulated Annealing (SA): Mimics the physical process of heating and then gradually cooling a material, starting with a random solution and making random changes. It can occasionally accept worse solutions to escape local optima.

2. Tabu Search (TS): Uses a memory structure called a tabu list to avoid revisiting previous solutions, moving from one solution to another while avoiding the repetition of certain moves.

3. Greedy Randomized Adaptive Search Procedures (GRASP): Alternates between constructing a randomized solution and applying local search to refine this solution. The construction phase is greedy and adaptive, making it capable of exploring diverse areas of the search space.

Solution Construction

The initial solution for the bin packing problem is generated through a randomized approach that strategically distributes items across multiple bins.

1. Identify Unplaced Items
2. Randomize Item Order
3. Sequential Bin Allocation: Begin placing items into bins one by one. For each new bin:
   - Attempt to add items individually.
   - Continue adding items until the bin reaches its capacity or a random decision—determined by sampling from a Bernoulli distribution—indicates to stop further additions. This process effectively mimics a coin flip, providing a 50% chance of continuing or stopping after each item is placed.
4. Evaluate Configuration

This method not only leverages randomness for diverse initial configurations but also incorporates a mechanism to break the placement sequence, adding an additional layer of complexity and variability to the solution space exploration.

Objective Function

The objective function proposed in the paper[2] focuses on optimizing the placement of items in bins by considering not only the number of bins used but also the amount of unused space within these bins. Specifically, to avoid stagnation and encourage the algorithm towards finding optimal solutions, the objective function incorporates the amount of unused space alongside the count of bins.

Minimization Objective Function: This function aims to minimize the inverse of the sum of the filled capacity ratios of all bins.

$$1 - \sum(fill_k/C)^z/N$$

Mathematically, it is represented as minimizing, where $fill_k$ is the filled capacity of bin k, C is the fixed bin capacity, z is a constant used to define the equilibrium of the filled bin, and N represents the number of bins in the solution.

These objective function prioritizes solutions that use fewer bins and minimize the unused space within those bins, aiming for efficient bin utilization. The inclusion of z, typically set to 2, helps balance the emphasis on bin count versus the filled capacity, guiding the optimization towards more desirable solutions.

[2] Munien, C., & Ezugwu, A. E. (2021). Metaheuristic algorithms for one-dimensional bin-packing problems: A survey of recent advances and applications. *Journal of Intelligent Systems*, 30, 636-663. https://doi.org/10.1515/jisys-2020-0117

Simulated Annealing

1. Initialization:
   The algorithm initializes by setting up an instance of the bin packing problem with specific parameters such as initial temperature, cooling schedule, stopping conditions, etc. This includes loading the initial bin packing configuration.
2. Calculation of Initial Temperature:
   The initial temperature is calculated to ensure that the algorithm starts with a high probability of accepting worse solutions, facilitating exploration of the solution space. This temperature is determined based on the maximum energy difference (or cost difference) observed when moving from best solution to the worst one within the context of the problem.

   Specifically, the formula used is:

$$\text{Initial Temperature} = -\frac{c_{\max}}{\log(\text{acceptance rate})}$$

   Here, 'c_max' is the maximum change in the objective function value that could result from a single tweak to the configuration at the start of the process. The logarithm of the acceptance rate (a predefined parameter indicating how likely the algorithm is to initially accept worse solutions) modifies this value to set an appropriate starting temperature.

   Starting from a calculated initial temperature in Simulated Annealing facilitates a controlled exploration of the solution space, preventing excessive early exploration that could lead to inefficient use of resources, while still allowing the algorithm to avoid premature convergence to suboptimal solutions. This approach ensures a smooth transition from accepting a broad range of solutions to fine-tuning around promising areas, enhancing the algorithm's efficiency and robustness throughout the optimization process.

3. Temperature Calculation:
   Depending on the selected cooling schedule (linear, logarithmic, adaptive, or square root), the algorithm calculates the temperature for each iteration. This temperature influences the probability of accepting suboptimal solutions, thereby aiding in the exploration of the solution space. The adaptive cooling schedule is particularly effective because it includes a reheating mechanism that helps the algorithm escape from local optima after several iterations of no improvement.
4. Solution Assessment:
   At each iteration, the current bin packing configuration's objective function (the unused space) is calculated. This serves as a measure of the solution quality.
5. Neighborhood Search:
   The algorithm explores the neighborhood of the current solution by applying small changes, known as tweaks, to the bin configuration. The specific tweak to be applied is selected based on a mechanism outlined in the " Neighborhood Definition" section of the current document.
6. Acceptance of New States:
   A new state (solution) generated from a tweak is evaluated. If this new state improves the objective function, it is accepted as the new current solution. If the new state is worse, it might still be accepted with a probability that depends on the current temperature and the extent of solution deterioration. This probability decreases as the temperature lowers, reducing the chance of accepting worse solutions as the algorithm proceeds.

7. Cooling:
   The temperature is reduced according to the cooling schedule after a set of iterations.
8. Equilibrium Condition:
   The algorithm checks if a state of equilibrium is reached within an iteration, which can involve no further improvements or changes being accepted for a set number of tries. This controls the inner loop and helps in deciding when to proceed to the next temperature level.
9. Stopping Conditions:
   The algorithm terminates based on specified stopping criteria which could be a time limit, a maximum number of evaluations, reaching a target solution quality, or a convergence condition where no improvements are observed over a set number of iterations.

Flowchart:

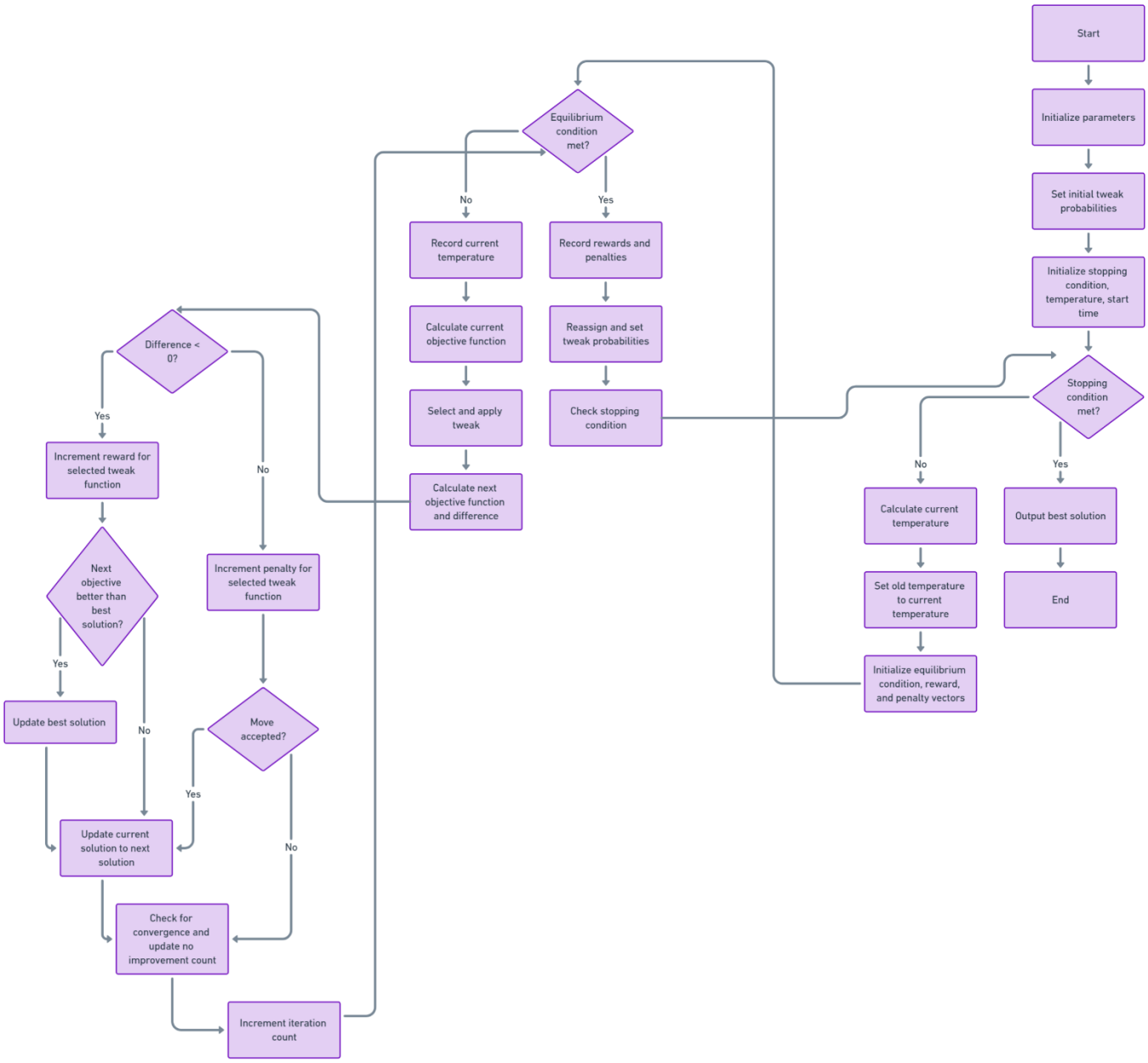
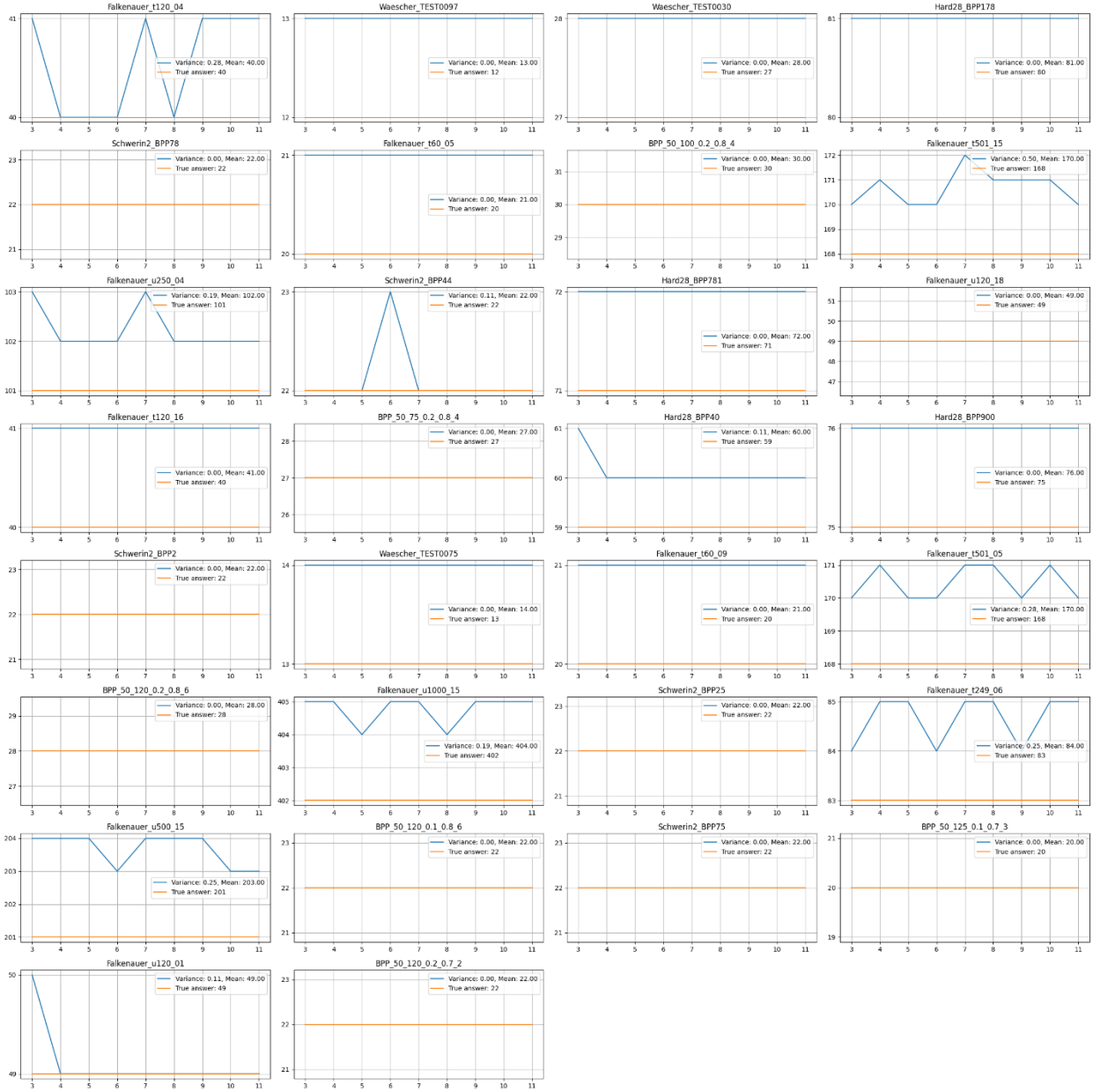


Figure 1: Simulated Annealing Flowchart

Results:

Instance Set	Optimal Solution	Min	Max	Variance	Mean	Gap	Time
Falkenauer_t120_04	40	40	41	0.28	40	0	80
Waescher_TEST0097	12	13	13	0	13	1	76
Waescher_TEST0030	27	28	28	0	28	1	47
Hard28_BPP178	80	81	81	0	81	1	80
Schwerin2_BPP78	22	22	22	0	22	0	0
Falkenauer_t60_05	20	21	21	0	21	1	54
BPP_50_100_0.2_0.8_4	30	30	30	0	30	0	0
Falkenauer_t501_15	168	170	172	0.5	170	2	115
Falkenauer_u250_04	101	102	103	0.19	102	1	99
Schwerin2_BPP44	22	22	23	0.11	22	0	0
Hard28_BPP781	71	72	72	0	72	1	105
Falkenauer_u120_18	49	49	49	0	49	0	2
Falkenauer_t120_16	40	41	41	0	41	1	47
BPP_50_75_0.2_0.8_4	27	27	27	0	27	0	0
Hard28_BPP40	59	60	61	0.11	60	1	117
Hard28_BPP900	75	76	76	0	76	1	100
Schwerin2_BPP2	22	22	22	0	22	0	0
Waescher_TEST0075	13	14	14	0	14	1	107
Falkenauer_t60_09	20	21	21	0	21	1	52
Falkenauer_t501_05	168	170	171	0.28	170	2	119
BPP_50_120_0.2_0.8_6	28	28	28	0	28	0	0
Falkenauer_u1000_15	402	404	405	0.19	404	2	1189
Schwerin2_BPP25	22	22	22	0	22	0	0
Falkenauer_t249_06	83	84	85	0.25	84	1	40
Falkenauer_u500_15	201	203	204	0.25	203	2	117
BPP_50_120_0.1_0.8_6	22	22	22	0	22	0	0
Schwerin2_BPP75	22	22	22	0	22	0	0
BPP_50_125_0.1_0.7_3	20	20	20	0	20	0	0
Falkenauer_u120_01	49	49	50	0.11	49	0	1
BPP_50_120_0.2_0.7_2	22	22	22	0	22	0	0

Plot of different Runs



Sample plot for showing implemented algorithm performance

\Falkenauer_u1000_15

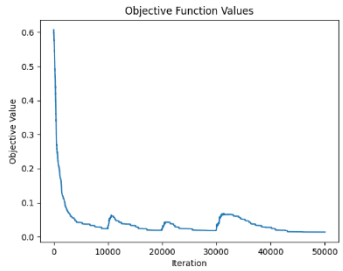


Figure 1

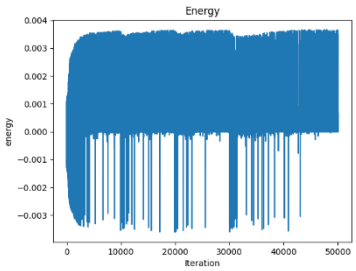


Figure 2

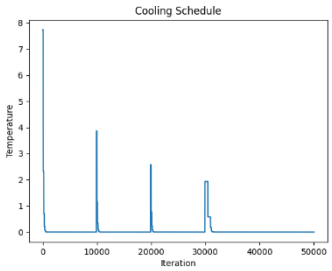


Figure 3

Figure 1 shows the decline and fluctuations in objective function values during a simulated annealing process, highlighting initial improvements and subsequent variations due to reheating, which helps the algorithm escape local minima by accepting worse solutions temporarily to explore new possibilities.

Figure 2 illustrates the energy changes over iterations, with stable positive energy levels and sharp downward spikes indicating acceptance of lower-energy solutions. This pattern supports deep exploration of the solution space and escape from local optima, facilitated by the Metropolis criterion.

Figure 3 depicts the temperature management in simulated annealing, with significant drops and plateaus indicating intermittent reheating. This pattern occurs after no improvements, allowing the algorithm to accept worse solutions for a time, thereby aiding in escaping local minima and balancing exploration with exploitation for optimal results.

Tabu Search

1. Initial Solution Assessment:
   The initial configuration's objective function, which is the unused space, is evaluated. This serves as the benchmark for subsequent solutions.

2. Solution Space Exploration:
   The algorithm explores the solution space by iteratively tweaking the current bin configuration, with a predefined number of tweaks.

3. Acceptance condition:
   For each tweak, the new configuration is assessed. First and foremost, if the proposed solution is the best found so far ('is_best'), it is immediately accepted, regardless of any other considerations. This ensures that the search always moves towards potentially optimal solutions when they are discovered. If the solution is not the best, the algorithm then checks whether it is not on the tabu list ('!next_in_tabuList'). This primary check is crucial to avoid revisiting recent configurations that didn't yield improvements. If the proposed move is not tabu, the algorithm will accept it if either the current solution is tabu ('current_in_tabuList') or if the move leads to a better solution ('is_better').

4. Aspiration Criteria:
   This criterion allows the algorithm to override the tabu status of a move if the cost of the new configuration is within a small threshold (epsilon) of the best solution.

5. Tabu List Updating:
   Moves that result in a new configuration are potentially added to the tabu list to prevent the algorithm from cycling back to recently explored solutions. These changes, made by tweaking the previous solution, are then added to the tabu list. (Details on the tabu list configuration will be elaborated later.)

6. Dynamic Tenure Adjustment:
   The tabu tenure is dynamically adjusted based on the state of the search, particularly if the algorithm is stuck in a local optimum. If the count of iterations without improvement exceeds the threshold, the tabu list will be expanded. This expansion increases the duration for which changes are tabued, thereby forcing the algorithm to explore more broadly.

7. Stopping Conditions:
   The algorithm terminates based on specified criteria, which could include a maximum number of iterations or duration of time, a convergence condition where no significant improvements are observed over a set number of iterations, or achieving a target solution quality.

Structure of the Tabu List

The "Tabu List" in the Tabu Search algorithm functions as a short-term memory system, recording recent modifications to the solution to avoid revisiting previous configurations. This mechanism is essential for promoting exploration and preventing the algorithm from cycling back to familiar territories.

The tabu list is structured to efficiently manage and track changes made during the search process:

- Storage: It is designed to quickly access, insert, and remove elements, ensuring that recent changes are always accounted for and older, less relevant changes are discarded once the list reaches its capacity. This is achieved by keeping a record of each modification along with the sequence in which they were made.

- Order Management: A specific component of the tabu list tracks the order in which changes are added. This component ensures that the first entry made (oldest) is the first to be removed when the list needs to make space for new entries, adhering to a First-In-First-Out (FIFO) policy.

The operations of the tabu list are fundamental to its role in the search algorithm:

- Insertion: New modifications to the solution are added to the tabu list. If the list has reached its capacity, the oldest modification is removed to make room for the new one.

- Removal: The oldest modifications are removed in FIFO order. This operation not only deletes the entry from the list but also ensures that any associated records are cleaned up to prevent unnecessary memory use.

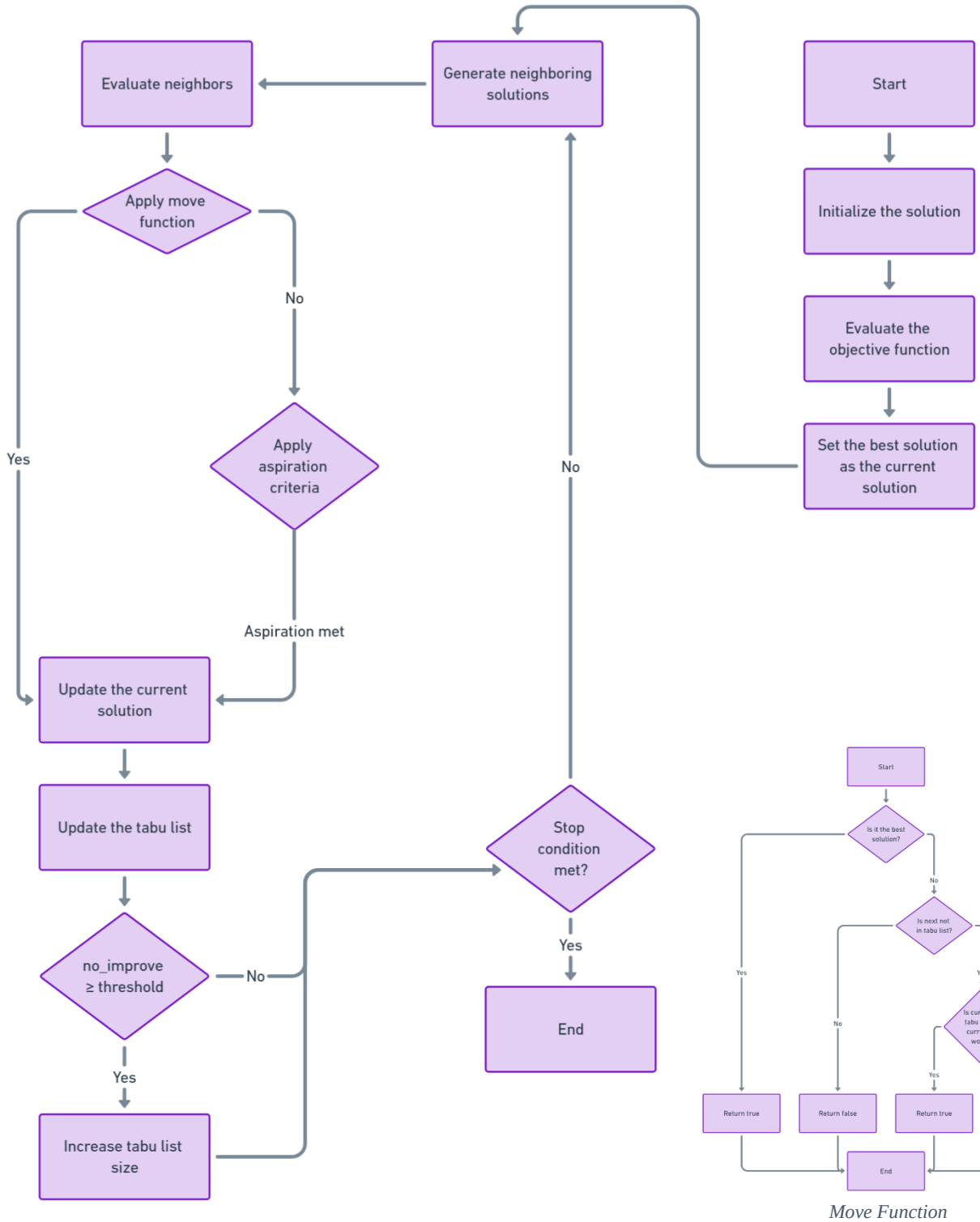
- Query: The tabu list allows for querying to check if a specific change is currently considered tabu, helping decide whether a move should be avoided.

What is a tabu move?

When considering a new move, the algorithm checks the tabu list to see if the proposed destination bin for the item is currently listed as a recent destination for that item. If the destination bin is on the list, the move is considered "tabu."

- Example: If an item was recently moved to Bin 5, and the algorithm is considering a move that would again place the item into Bin 5, this move would be checked against the tabu list. If Bin 5 is still within the tabu tenure for that item, the move is prohibited.

Flowchart

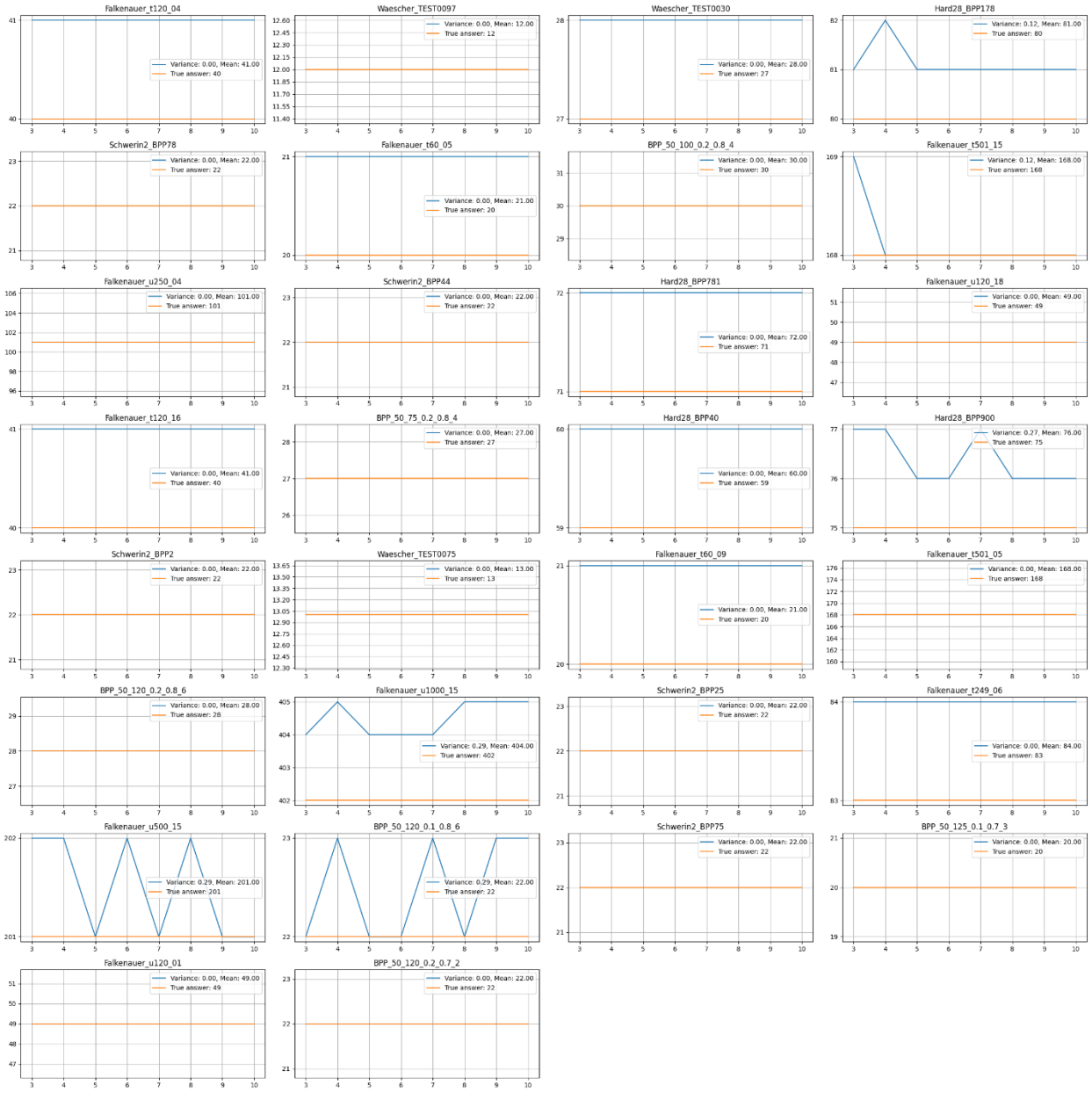


Move Function

Results:

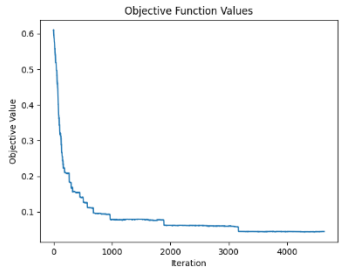
Instance Set	Optimal Solution	Min	Max	Variance	Mean	Gap	Time
Falkenauer_t120_04	40	41	41	0	41	1	1
Waescher_TEST0097	12	12	12	0	12	0	0
Waescher_TEST0030	27	28	28	0	28	1	1
Hard28_BPP178	80	81	82	0.12	81	1	14
Schwerin2_BPP78	22	22	22	0	22	0	0
Falkenauer_t60_05	20	21	21	0	21	1	0
BPP_50_100_0.2_0.8_4	30	30	30	0	30	0	0
Falkenauer_t501_15	168	168	169	0.12	168	0	23
Falkenauer_u250_04	101	101	101	0	101	0	4
Schwerin2_BPP44	22	22	22	0	22	0	0
Hard28_BPP781	71	72	72	0	72	1	27
Falkenauer_u120_18	49	49	49	0	49	0	0
Falkenauer_t120_16	40	41	41	0	41	1	2
BPP_50_75_0.2_0.8_4	27	27	27	0	27	0	0
Hard28_BPP40	59	60	60	0	60	1	13
Hard28_BPP900	75	76	77	0.27	76	1	26
Schwerin2_BPP2	22	22	22	0	22	0	0
Waescher_TEST0075	13	13	13	0	13	0	1
Falkenauer_t60_09	20	21	21	0	21	1	0
Falkenauer_t501_05	168	168	168	0	168	0	13
BPP_50_120_0.2_0.8_6	28	28	28	0	28	0	0
Falkenauer_u1000_15	402	404	405	0.29	404	2	116
Schwerin2_BPP25	22	22	22	0	22	0	0
Falkenauer_t249_06	83	84	84	0	84	1	34
Falkenauer_u500_15	201	201	202	0.29	201	0	37
BPP_50_120_0.1_0.8_6	22	22	23	0.29	22	0	0
Schwerin2_BPP75	22	22	22	0	22	0	0
BPP_50_125_0.1_0.7_3	20	20	20	0	20	0	0
Falkenauer_u120_01	49	49	49	0	49	0	0
BPP_50_120_0.2_0.7_2	22	22	22	0	22	0	0

Plot of Different Runs:



Sample plot for showing implemented algorithm performance

\Falkenauer_u250_04



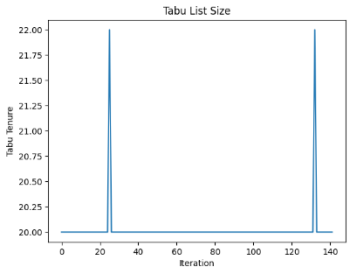


Figure 1 Figure 2

There are two major decreases in the objective function value, occurring at iterations 1900 and 3100, after a prolonged period of being stuck in local optima. The decreases are attributed to the detection of local optima and subsequent expansion of the tabu list size. Note that the iteration axis in Figure 2's plot does not correspond to the iteration numbers in Figure 1, due to the inner loop of the Tabu Search algorithm.

GRASP

1. Construction Phase:

- Begins with a set of items that need to be packed into bins.

- Iterates over items and places each into a suitable bin based on a Restricted Candidate List (RCL). The size of the RCL is calculated dynamically based on alpha, affecting how many bins are considered during placement.

- Alpha affects how selective the RCL is — a lower alpha results in a greedier approach, while a higher alpha allows more randomness. After several iterations with no improvement in the best objective, alpha is increased to explore more of the search space.

- The 'First Fit' heuristic is utilized for generating candidate bins during the construction phase. Compared to the 'Best Fit' heuristic, 'First Fit' is much faster and less greedy. By combining this with item shuffling in each construction phase, additional randomness is introduced, enhancing the exploration of the search space.

- New bins are created if no existing bins can accommodate the current item.

- A candidate bin from the RCL is selected randomly to place the current item, promoting diversity in bin selection.

3. Local Search Application:

- Once all items are initially placed, a local search algorithm is applied to improve the solution.

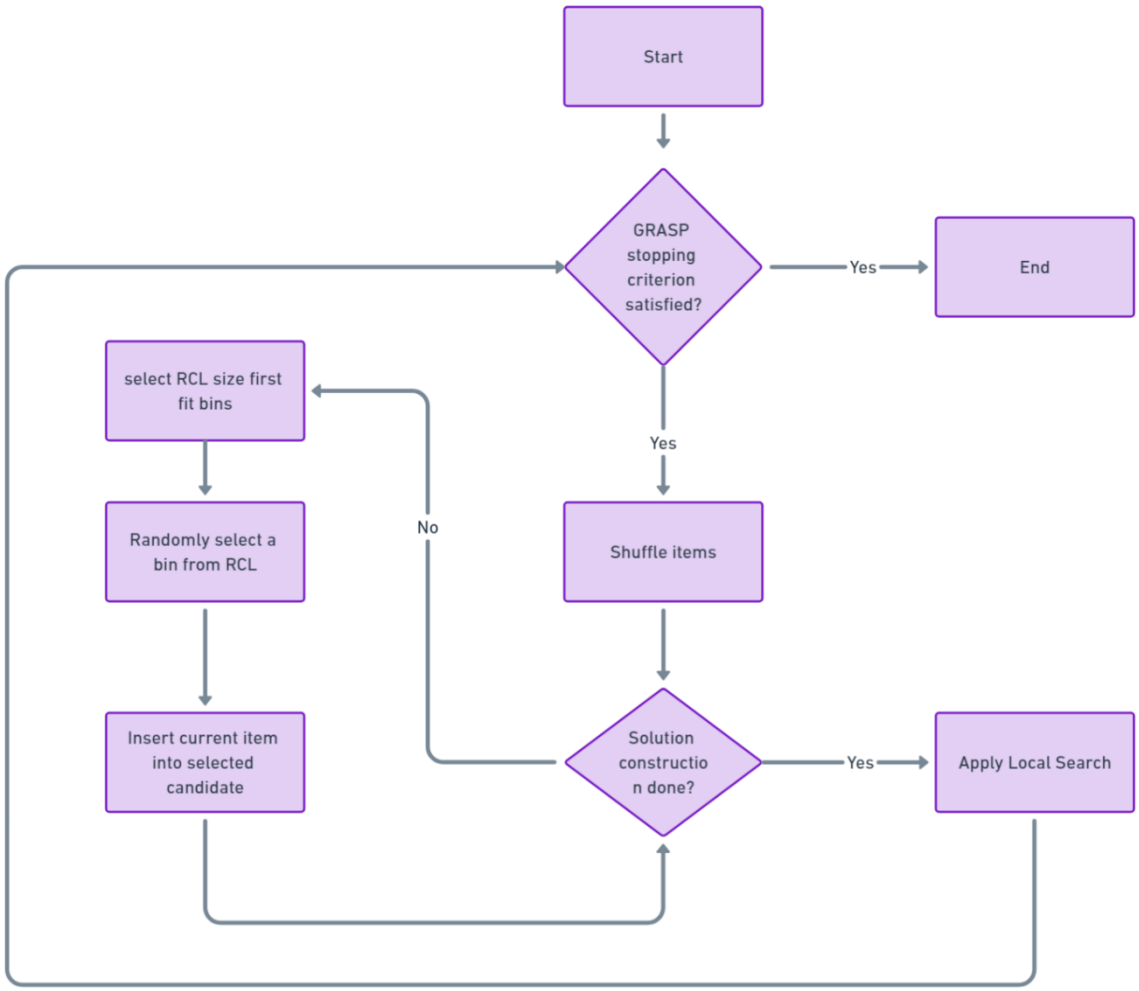
- The choice of local search method—Hill Climbing, Simulated Annealing, or Tabu Search—affects how the solution is refined. Although Hill Climbing has been chosen for testing instance sets, Tabu Search and then Simulated Annealing have been found to outperform it.

4. Adaptation of Parameters:

- Parameters like alpha are adjusted dynamically based on the progress of the search, particularly in response to the number of iterations without improvement.

- This step aims to dynamically balance exploration and exploitation based on the search history. If there's no improvement in the best solution for several iterations, alpha is adjusted to explore alternative solutions.

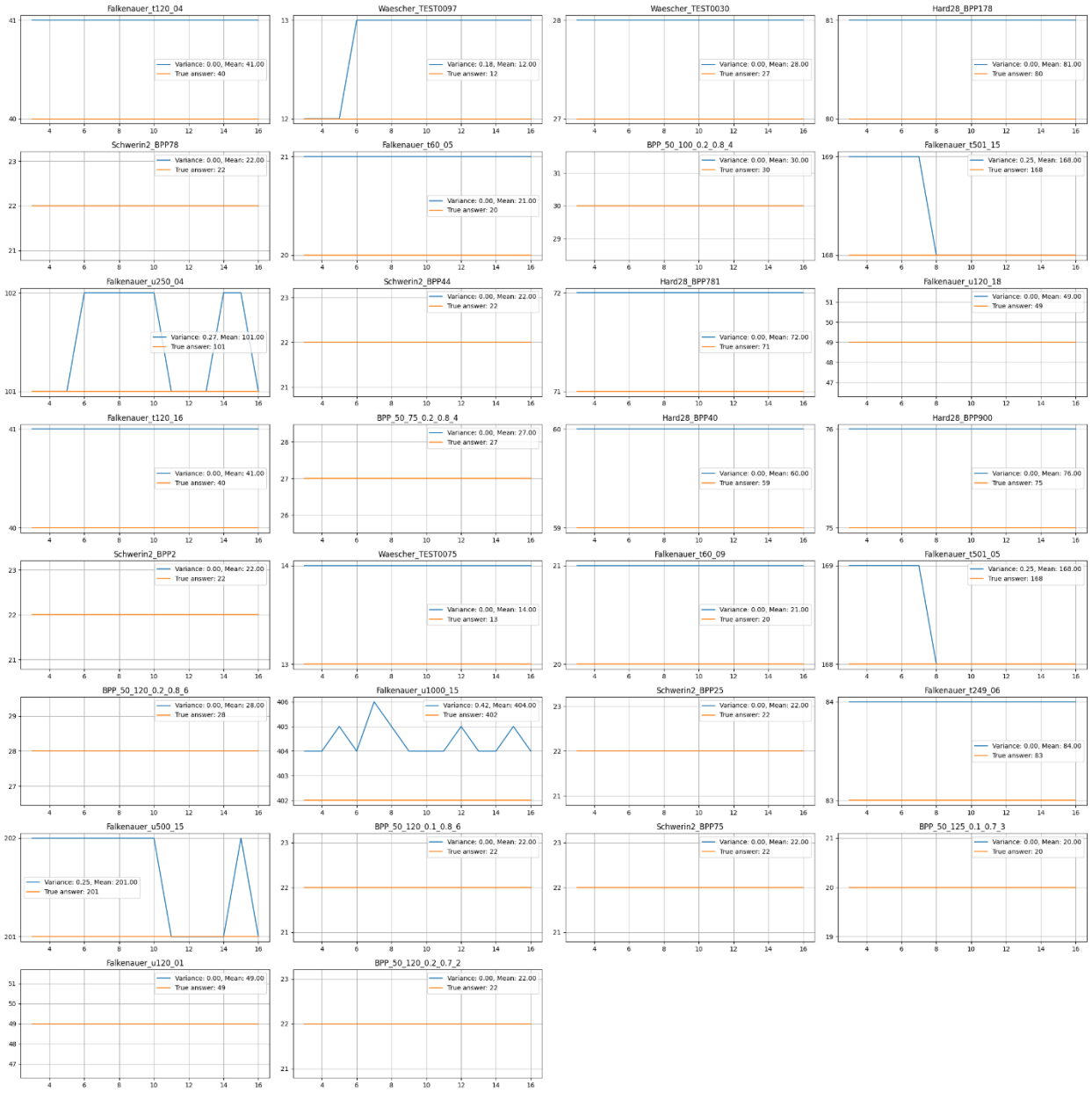
Flowchart


Start
GRASP stopping criterion satisfied?
Yes
End
Yes
select RCL size first fit bins
Randomly select a bin from RCL
No
Shuffle items
Insert current item into selected candidate
Solution constructio n done?
Yes
Apply Local Search

Results

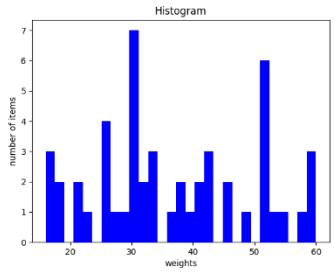
Instance Set	Optimal Solution	Min	Max	Variance	Mean	Gap	Time
Falkenauer_t120_04	40	41	41	0	41	1	300
Waescher_TEST0097	12	12	13	0.18	12	0	83
Waescher_TEST0030	27	28	28	0	28	1	300
Hard28_BPP178	80	81	81	0	81	1	63
Schwerin2_BPP78	22	22	22	0	22	0	0
Falkenauer_t60_05	20	21	21	0	21	1	204
BPP_50_100_0.2_0.8_4	30	30	30	0	30	0	0
Falkenauer_t501_15	168	168	169	0.25	168	0	10
Falkenauer_u250_04	101	101	102	0.27	101	0	81
Schwerin2_BPP44	22	22	22	0	22	0	0
Hard28_BPP781	71	72	72	0	72	1	61
Falkenauer_u120_18	49	49	49	0	49	0	0
Falkenauer_t120_16	40	41	41	0	41	1	250
BPP_50_75_0.2_0.8_4	27	27	27	0	27	0	0
Hard28_BPP40	59	60	60	0	60	1	96
Hard28_BPP900	75	76	76	0	76	1	120
Schwerin2_BPP2	22	22	22	0	22	0	0
Waescher_TEST0075	13	14	14	0	14	1	287
Falkenauer_t60_09	20	21	21	0	21	1	300
Falkenauer_t501_05	168	168	169	0.25	168	0	4
BPP_50_120_0.2_0.8_6	28	28	28	0	28	0	0
Falkenauer_u1000_15	402	404	406	0.42	404	2	295
Schwerin2_BPP25	22	22	22	0	22	0	0
Falkenauer_t249_06	83	84	84	0	84	1	67
Falkenauer_u500_15	201	201	202	0.25	201	0	11
BPP_50_120_0.1_0.8_6	22	22	22	0	22	0	0
Schwerin2_BPP75	22	22	22	0	22	0	0
BPP_50_125_0.1_0.7_3	20	20	20	0	20	0	20
Falkenauer_u120_01	49	49	49	0	49	0	0
BPP_50_120_0.2_0.7_2	22	22	22	0	22	0	0

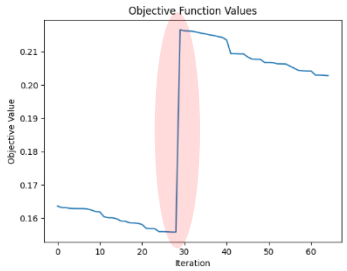
Plot of Different Runs: (Hill Climbing as Local Search)



Sample plot for showing implemented algorithm performance

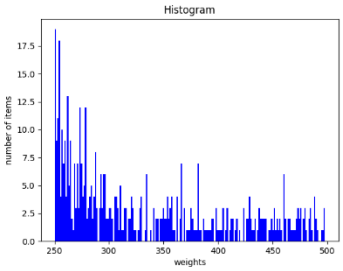
\BPP_50_75_0.2_0.8_4

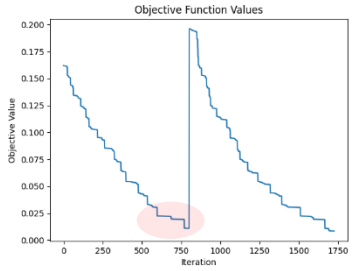




The increase in the objective value indicates that the GRASP algorithm initiated a new start. However, the subsequent decrease in other parts of the objective value suggests improvements

\Falkenauer_t501_05





Based on the decreasing trend observed in the plot, it is clear that the local search phase required more time to converge to an optimal value for this complex set of instances, as indicated by the data distribution.

For such a problem, where item weights can vary across different instance sets, observing the data distribution can be beneficial. Based on these observations, one can implement different greedy functions to achieve better-quality initial solutions. Then, by using the RCL and adjusting the alpha parameter, the algorithm can introduce randomness, thus leveraging the benefits of a purely random approach.

Population Base:

Population-based metaheuristics are a category of optimization algorithms that use a population of solutions to explore and exploit the search space, aiming to find the best solution to a problem. These algorithms are inspired by biological, physical, or social phenomena, and they typically involve mechanisms like selection, reproduction, mutation, and collaboration among individuals in the population. Genetic Algorithms (GA), Ant Colony Optimization (ACO), and Particle Swarm Optimization (PSO) are examples of population-based algorithms.

Each of these algorithms has its strengths and is suitable for different types of optimization problems. Their effectiveness often depends on the nature of the problem, the quality of the implementation, and the tuning of their parameters.

Feasibility Check:

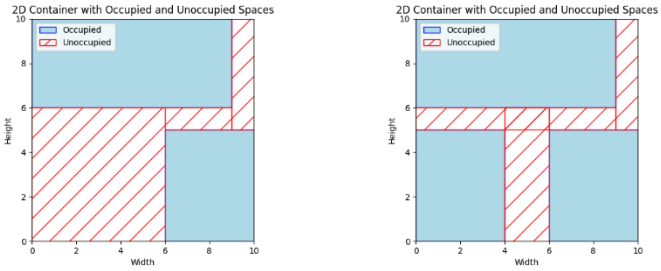
The feasibility check for ensuring that no rectangles overlap within a bin utilizes a sweep-line algorithm combined with an interval tree, specifically leveraging the Bentley-Ottmann algorithm[3]. The sweep-line algorithm detects intersections by moving a vertical line across the plane from left to right, processing events where rectangles start and end. It involves three main steps: creating events for each rectangle's endpoints, sorting these events by x-coordinate, and processing the events while maintaining active intervals in an interval tree. The interval tree efficiently manages these active intervals, allowing quick insertion, deletion, and overlap checking operations. The overall time complexity of this approach is significantly improved compared to a brute-force method, with sorting events taking O(nlogn) and processing each event in O(log n) time. Thus, the sweep-line combined with the interval tree results in a time complexity of O(mnlog n), where n is the number of rectangles and m is the number of bins, compared to the O(mn^2) complexity of a brute-force approach.

[3] Smid, M. (Year). *Computing intersections in a set of line segments: The Bentley-Ottmann algorithm.* Retrieved from https://people.scs.carleton.ca/~michiel/lecturenotes/ALGGEOM/bentley-ottmann.pdf

Rectangle Placement:

The Rectangle placements module manages the placement of items within bins, specifically focusing on optimizing how space is utilized. Each bin maintains a list of its unoccupied spaces. When an item is to be placed, the module searches for the first available unoccupied space within the bin that is large enough to accommodate the item. The placement strategy involves positioning the item in the leftmost corner of this chosen unoccupied space.

If the item does not fully occupy the selected space, the remaining area of the unoccupied space is split into two new spaces. This splitting is guided by the placement of the item in the corner, which naturally divides the remaining area into two distinct regions based on the orientation of the item and its dimensions relative to the dimensions of the original space. The search process occurs through unoccupied spaces. These unoccupied spaces are then merged, helping to prevent the space from being divided into smaller pieces.[4]



Before placing an item, the module first verifies that the area of the selected unoccupied space is greater than or equal to the area of the item. If a suitable space is found, the item is placed; if not, the module employs a strategy to reorganize the unoccupied spaces to potentially create a suitable area. This reorganization involves sweeping through the spaces either from right to left or from top to bottom. The choice between these strategies depends on the item's dimensions: a right-to-left sweep is used for items that are wider than they are tall (wide items), whereas a top-to-bottom sweep is used for items that are taller than they are wide (tall items).

If, after reorganizing the spaces through the sweep, no suitable space can be found to fit the item, the module returns false, indicating that the item cannot be placed in the current configuration of the bin. This systematic approach to item placement and space management allows for the efficient use of space within bins, although it is constrained by the physical dimensions and the initial arrangement of the unoccupied spaces.

[4] Nguyen, T.-H., & Nguyen, X.-T. (2023). Space Splitting and Merging Technique for Online 3-D Bin Packing. *Mathematics, 11*(1912), 1-16. https://doi.org/10.3390/math11081912

Right-to-Left Sweep Line Method for Space Optimization

The right-to-left sweep line method is an algorithm designed to efficiently manage and reorganize unoccupied spaces within a two-dimensional area, specifically to enhance the accommodation of tall items. This method employs a dynamic and systematic approach to analyze and partition available spaces based on their dimensions and positional relationships.

- Spatial Analysis: As the sweep line progresses from the rightmost edge to the leftmost edge of the container, it encounters and processes the boundaries of existing unoccupied spaces.

- Event-Driven Manipulation: Each space is associated with two events—'start' at the right boundary and 'end' at the left boundary. The ordering of these events is crucial, as it determines when spaces are activated or deactivated for consideration.

- Space Partitioning: When the sweep line reaches the 'end' of a space without aligning with its left boundary, the space may be split. This splitting is based on the current position of the sweep line, effectively dividing the space into two segments: one to the right of the line (which remains active and will be used for merging) and one to the left (which recursively undergoes further processing).

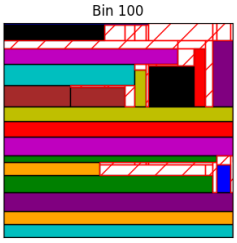
Advantages of Vertical Merging:

- Enhanced Vertical Space Utilization: By partitioning spaces as described, the algorithm optimally prepares them for vertical merging. This is particularly beneficial for the placement of tall items, as it creates taller, continuous vertical spaces free of horizontal interruptions.

- Improved Packing Density: This method reduces the fragmentation of space, allowing taller items to be placed more efficiently. By minimizing wasted space through strategic vertical merging, the overall packing density and organization of the area are significantly enhanced.

Top-To-Bottom Sweep Line Method for Space Optimization

The top-to-bottom sweep line method mirrors the functionality of the right-to-left sweep line approach but is tailored for optimizing space utilization for wide items. In this method, the sweep line moves from the top to the bottom of the container, systematically managing the horizontal expanses of unoccupied spaces. As the line encounters the upper and lower boundaries of these spaces, it triggers events—'start' at the upper boundary and 'end' at the lower boundary. This event-driven process ensures that spaces are considered active when the line enters them and are re-evaluated for splitting as the line exits. If a space's lower boundary does not align with the sweep line's current position, it may be split horizontally, creating separate upper and lower sections. This horizontal partitioning is key to preparing spaces for subsequent horizontal merging, which is particularly advantageous for placing wider items. By consolidating fragmented horizontal spaces into larger continuous areas, this method enhances the efficient utilization of space, enabling better placement of wide items and improving overall packing density within the container.



By using this approach, items can be placed anywhere within the bin without using any heuristics.

Objective Function:

The objective_function calculates the overall fitness of a list of container bins. For each container (bin), it accumulates the fitness value obtained from calling get_fitness on that container. The fitness of each container is calculated as the square of the ratio of the filled area (total_area - available_area) to the total area of the container (width * height). This squaring emphasizes differences in packing efficiency. The objective function then returns the inverse of the average fitness across all containers, calculated as 1 - (bins_fitness / len(containers)). This means that a higher average fitness (i.e., more space filled in each container) results in a lower objective value, aiming to minimize this value to improve packing efficiency.

$$\text{Objective} = 1 - \frac{1}{n}\sum_{i=1}^{n}\left(\frac{\text{total\_area}_i - \text{available\_area}_i}{\text{total\_area}_i}\right)^2$$

However, based on experiments with this objective function, having fewer bins (with a difference of at most one) does not necessarily mean a lower objective value. To address this issue, we prioritize the number of bins as the main objective of the problem. Therefore, a solution with fewer bins is considered better than one with a lower objective value but more bins.

Genetic Algorithm:

Representation:

In the implementation of genetic algorithms for 2D bin packing, the chosen group-based[5] chromosome representation is pivotal for enhancing the effectiveness of crossover and mutation processes. This representation facilitates efficient 2-point group-based crossover by allowing the exchange of whole item groups between parent chromosomes, thereby enriching genetic diversity and improving packing solutions. It also supports versatile mutation strategies, either within a single bin for a more compact layout or between bins to enhance overall packing efficiency. Despite its capacity for complex genetic operations, the group-based approach maintains simplicity and intuitiveness, simplifying the algorithm's implementation and making the translation from chromosome to packing solution more straightforward.

4 – 7	9	2 – 3	5 – 1 – 8	6

This is a chromosome, where each index indicates a bin, and the elements in each cell are item ids. Each item object has id and coordinates indicating the x and y positions of the item's bottom-left corner within the bin area, which evolve during the search process.

[5] Falkenauer, E. (1994). A new representation and operators for genetic algorithms applied to grouping problems. Evolutionary Computation, 2(2), 123-144. https://doi.org/10.1162/evco.1994.2.2.123

Population initialization:

‘feasible_random_packing’ function:

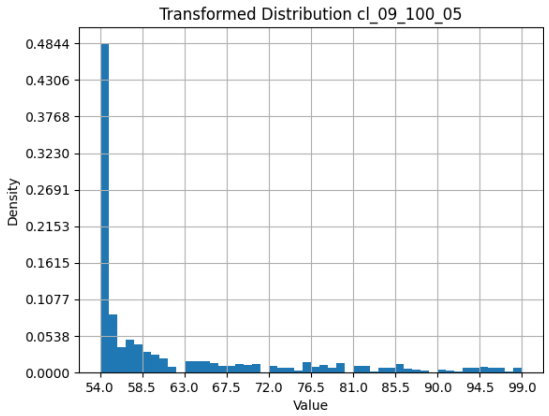
Feasible random packing in genetic algorithms offers a robust and efficient solution to the 2D bin packing problem by leveraging the randomness in item selection and placement. This approach ensures diversity in the solutions, enabling the algorithm to explore a wide range of configurations and avoid local optima. Diversity in a population-based algorithm like this is crucial because it prevents the algorithm from converging prematurely on suboptimal solutions, thereby enhancing its ability to find more optimal or near-optimal solutions across various runs. It also helps maintain a healthy variety of genetic material in the population, which is essential for effective crossover and mutation processes, key mechanisms in genetic algorithms that drive evolution towards better solutions.

‘infeasible_packing’ function:

In the genetic algorithm for 2D bin packing, generating an infeasible initial population intentionally includes chromosomes that violate constraints to widen the exploration of the solution space, potentially uncovering innovative solutions. This process involves:

1. Sampling the Number of Bins: Bins are sampled from a skewed distribution to explore configurations that vary significantly from typical solutions, such as over-packed scenarios.

For example, for instance set “” we have this distribution:



This ensures not only that we have a low number of bins to guarantee infeasibility, but also that we don't have too many bins. The minimum number is not necessarily achievable since it represents the minimum bins required to accommodate items with relaxed constraints. Furthermore, the maximum number of bins in this distribution equals the total number of items.

This approach allows for some chromosomes to have a low number of bins due to overpacking, with the hope that they will be able to spread good genetic material.

2. Distributing Items Among Bins: Items are initially distributed to ensure each bin has at least one item, with the rest allocated using a normal distribution to create natural variability and deliberate imbalances.

3. Placing Items in Bins: Items are randomly placed within bins (coordinates of bottom left corners are initialized randomly), often leading to overlaps and inefficient space usage, which intentionally enhances infeasibility.

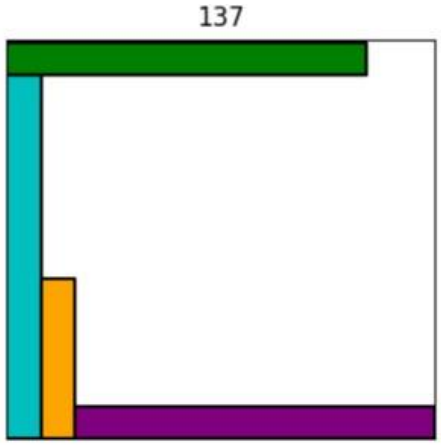
4. Evaluating Infeasibility and Fitness: All configurations, regardless of feasibility, are retained for fitness evaluation, which assesses the degree of infeasibility and guides evolutionary operations to optimize these initially flawed solutions.

A portion (like 95%) of the population is initialized with feasible chromosomes, and another portion with infeasible chromosomes.

For exploring additional initialization approaches, other methods have been tested, such as the randomized row packing heuristic and the first fit heuristic. However, these were not able to diversify the population effectively, which may lead to premature convergence.

For example, Randomized Row Packing Heuristic**:**

The row_packing function is designed to place items into bins in a manner that may vary depending on the order of items due to its randomized nature. This randomization is critical as it prevents the algorithm from repeatedly placing the same items together, thereby increasing the diversity of the population. Items distributed among bins are sorted primarily by height, which facilitates packing into rows within each bin. This method prioritizes filling bins row by row, a common heuristic in packing problems to maximize space utilization. Although there are some important drawbacks to this heuristic, which will be elaborated upon later, its strength lies in its speed. After sorting, items are sequentially placed into the current bin until no more items can fit without violating bin constraints. If an item does not fit, it is set aside for potential placement in a new bin. However, this approach does not have good long-term performance.



This is a poor genotype that has spread in the population due to the nature of the row-packing heuristic. However, other heuristics may work better at the bin level, but they are not fast as row packing.

Selection

Parent Selection:

Tournament selection is a method used in genetic algorithms that involves randomly selecting a subset of individuals from a population to compete in a 'tournament,' where the individual with the highest fitness wins and is chosen for reproduction. This process is repeated to populate the next generation. The size of the tournament, typically small (e.g., 2 or 3), can be adjusted to control the selection pressure: larger tournaments increase pressure and can accelerate convergence, but they may also risk premature convergence by reducing genetic diversity. The method is effective because it is simple, flexible, and scales well with parallel processing, making it particularly suitable for complex optimization problems. However, its stochastic nature means that it doesn't always guarantee the selection of the best individuals. This is advantageous as every chromosome, even the least fit, has a chance to be selected, which may include unique and valuable genetic material.

Survival Selection:

Elitism is a survival selection strategy used in genetic algorithms to ensure that the best individuals, or "elites," from a population are carried over to the next generation without undergoing crossover or mutation. Typically, only a small percentage, denoted as x%, of the population is selected as elites. This method directly preserves the top performers, ensuring that the genetic material of the fittest individuals is not lost during the generational transition. The remaining portion of the population is replaced through standard selection methods involving crossover and mutation of other individuals. The benefit of elitism is that it provides a steady improvement in fitness over generations by continuously building on the best available solutions. This approach reduces the risk of losing highly fit solutions due to random selection processes and helps maintain steady progress towards the optimization goal.

Mutation:

Main operator for evolving (items position within a bin)

1. Placement and Removal Mutations:

- ‘feasible_insert’: Attempts to place an item into a bin's unoccupied area optimally. It checks for overlaps and maintains the bin's feasibility.

- ‘randomly_insert’: Places an item at random coordinates within a bin, not accounting for overlaps or feasibility, which may introduce disruptions.

- ‘item_overlap_removal’: This function addresses items that overlap with others by adjusting their positions to ensure all items fit within their designated bins without overlapping.

2. Reinsertion and Swapping Mutations:

- ‘random_reinsert_mutation’ and ‘feasible_reinsert_mutation’: Move an item from one bin to another random bin, with the latter ensuring the move maintains bin feasibility. These mutations operate at the chromosome level, potentially calling the ‘random_insert’ or ‘feasible_insert’ functions to place an item in the destination bin, thus acting like utility functions that handle item placement within the context of a broader chromosome structure.

- ‘random_between_swap_mutation’ and ‘feasible_between_swap_mutation’: Swap items between two bins. The feasible version checks that each item fits in its new location before executing the swap, using the capabilities of the insert functions to validate placements.

3. Bin Optimization Mutation:

- ‘Empty bin’: Attempts to redistribute all items from a randomly selected bin into other bins, aiming to potentially remove the empty bin and reduce the total number of bins used.

These mutation functions are crucial for enhancing the genetic algorithm’s ability to explore the solution space thoroughly. They adjust item placements and bin assignments to escape local optima, with some focusing on maintaining or improving feasibility to ensure high-quality solutions, and some providing new genetic material.

Crossover:

- Group-Based Approach: Instead of swapping individual items between parent chromosomes, this method swaps whole groups of items (bins). This helps preserve effective item combinations (building blocks) and reduces the chances of breaking efficient packing patterns.

- Implementation Steps:

1. Select Segments: Randomly pick segments of bins from each parent chromosome.

2. Swap Segments: Exchange these segments between the two parents to create new offspring, mixing their genetic material.

3. Repair Offspring: Adjust the new chromosomes to fix any issues, such as duplicated or missing items, ensuring they are valid. Missing items are placed randomly within the bin, with their bottom-left corner coordinates added randomly. This means that although the parents were feasible, there is a high chance their offsprings might become infeasible, so a repair mechanism and a penalty to the objective were introduced to handle this issue.

How it performs:

Number of consecutive groups to be selected: 1

Parent1_point1: 5, parent1_point2: 5

7	1	9	3 – 8	6	5 – 4	2

parent2_point1: 7, parent2_point2: 8

7	9	6	3	5 – 1 – 8	4 – 2

Offspring1 before repair:

7	1	9	3 – 8	4 – 2	5 – 4	2	8	1

Offspring2 before repair:

7	9	6	3	5 – 1 – 8	6

Offspring1 after repair to keep permutation:

Missed items will be inserted in places outside the swap range to preserve the inherited bins, and duplicated items will be eliminated from areas outside the swap range.

7	1	9	3 – 8	4 – 2	5 – 6

Offspring2 after repair to keep permutation:

4 – 7	9	2 – 3	5 – 1 – 8	6

Repair mechanism:

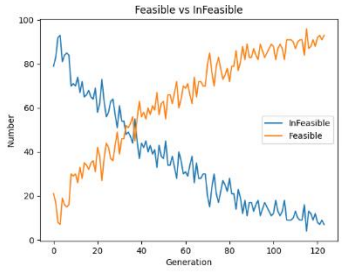
The ‘calculate_repair_probability’ function assesses the likelihood of initiating a repair based on factors like the number of generations passed, time constraints. This probability is dynamically adjusted using an exponential function to increase the urgency of repairs as more time passes or as the algorithm nears its generation limit.

$$P = 1 - e^{\left(\frac{-2 \times \text{current\_generation}}{\text{max\_generation}} \right)}$$

On the other hand, the ‘repair_chromosome’ function decides whether to actually proceed with repairing a specific chromosome, factoring in its feasibility and a randomly generated number. If a chromosome is found to be infeasible and this random number is less than the previously calculated repair probability, the function triggers a specific repair strategy, such as ‘repair_chromosome’, to make the chromosome feasible again.

The repair_chromosome function systematically addresses the reorganization of infeasible bins within a chromosome in a genetic algorithm for 2D bin packing. It processes each bin identified as infeasible by attempting to repack its items into a newly created bin using a feasible rectangle placement strategy. Items that don't fit are marked as leftovers. At the end of iterating through all bins, the function continues by trying to place these leftover items into previous bins or additional new bins, repeating this cycle until all items are successfully placed or confirmed as unplaceable. This ensures each chromosome evolves towards maximum feasibility by optimally utilizing bin space and reducing the number of infeasible bins.

Together, these functions ensure the algorithm efficiently identifies and corrects suboptimal solutions, maintaining the overall quality of the population.



Due to the use of a repair mechanism with a dynamic probability, we can observe that the number of infeasible solutions decreases with passing generations.

Constraint Handling:

Constraint: No two items within a bin may overlap or extend beyond each other's boundaries.

The functions for evaluating bin packing solutions include calculating fitness, which integrates an objective function for efficiency and a penalty for constraints violations[6]. The 'calculate_fitness(penalty_factor)' function determines the overall fitness by weighing the objective value, which assesses bin utilization efficiency, and a penalty that measures deviation from feasibility, such as item overlaps. The penalty's impact on fitness is adjusted using constants 'alpha' and 'beta', and scaled further by the 'penalty_factor', which increases over generations to discourage infeasible solutions.

The 'objective_function' calculates the average utilization of only feasible bins, emphasizing efficient space use, while the 'calculate_penalty' function accumulates penalties for each non-feasible bin based on specific packing inefficiencies, such as items overlapping the bin boundaries. These components work together to guide the algorithm towards more efficient and feasible bin packing configurations.

Alpha = 1 and beta = 1.5, penalty factor increases over generations to discourage infeasible solutions.

m is number of feasible bins.

$$\text{Objective} = 1 - \left(\frac{\sum_{i=1}^{m} \text{Utilization}_i}{m \times \text{Area}_{\text{total}}} \right)^z$$

$$\text{Penalty}_{\text{bin},i} = \frac{\text{Area}_{\text{items},i} - (\text{Area}_{\text{total},i} - \text{Area}_{\text{unoccupied},i})}{\text{Area}_{\text{total},i}}$$

$$\text{Total Penalty} = \frac{\sum \text{Penalty}_{\text{bin},i}}{\text{Number of Infeasible Bins}}$$

$$\text{Fitness} = \alpha \times \text{Objective} + \beta \times \text{Penalty Factor} \times \text{Total Penalty}$$

[6] Ponce-Pérez, A., Pérez-Garcia, A., & Ayala-Ramirez, V. (n.d.). Bin-packing using genetic algorithms. Universidad de Guanajuato FIMEE, Salamanca, Gto., Mexico.

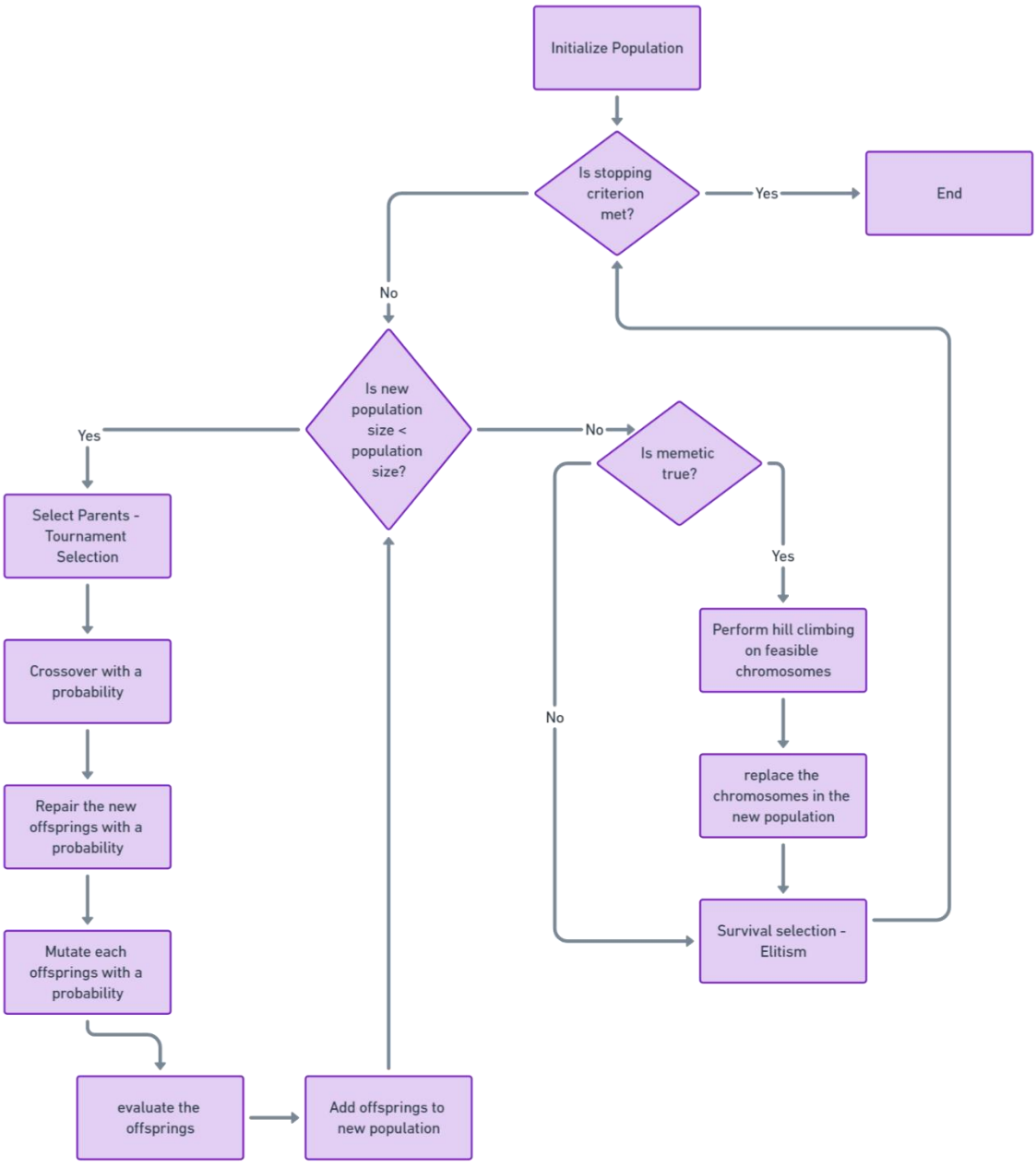
Memetic Algorithm:

The described function employs a local search strategy as part of a memetic algorithm, which combines the global search capabilities of a genetic algorithm with the precision of local optimization techniques like hill climbing. This is particularly effective in solving complex bin packing problems. In this setup, each feasible chromosome from the new population undergoes hill climbing optimization, with a 75% probability. This selective approach helps balance exploration and exploitation, ensuring that not every solution is intensely optimized every time, which helps to maintain genetic diversity within the population.

The hill climbing method used includes specialized mutations such as feasible reinsertion, feasible swap, and empty bin tweaks. These operations are critical as they allow the algorithm to fine-tune each solution by making small, valid adjustments that do not breach the problem's constraints. Feasible reinsertion moves items between bins while maintaining the integrity of the solution, ensuring that items fit without causing overlaps. Feasible swap similarly allows for the exchange of items between bins under the condition that the swap maintains a valid configuration, thereby increasing the diversity of solution exploration.

These tweaks are strategically implemented to refine solutions incrementally, improving the overall quality of the population before it undergoes survival selection. This methodology not only enhances individual solutions but also ensures the population evolves to become more adept at solving the problem in successive generations. The inclusion of hill climbing, characterized by its simplicity and effectiveness, allows for quick enhancements within local neighborhoods of solutions, albeit with controlled iterations to prevent the algorithm from getting trapped in local optima. This integrative approach effectively marries evolutionary strategies with targeted local optimizations, promoting a continuous cycle of improvement in both the structure and feasibility of solutions.

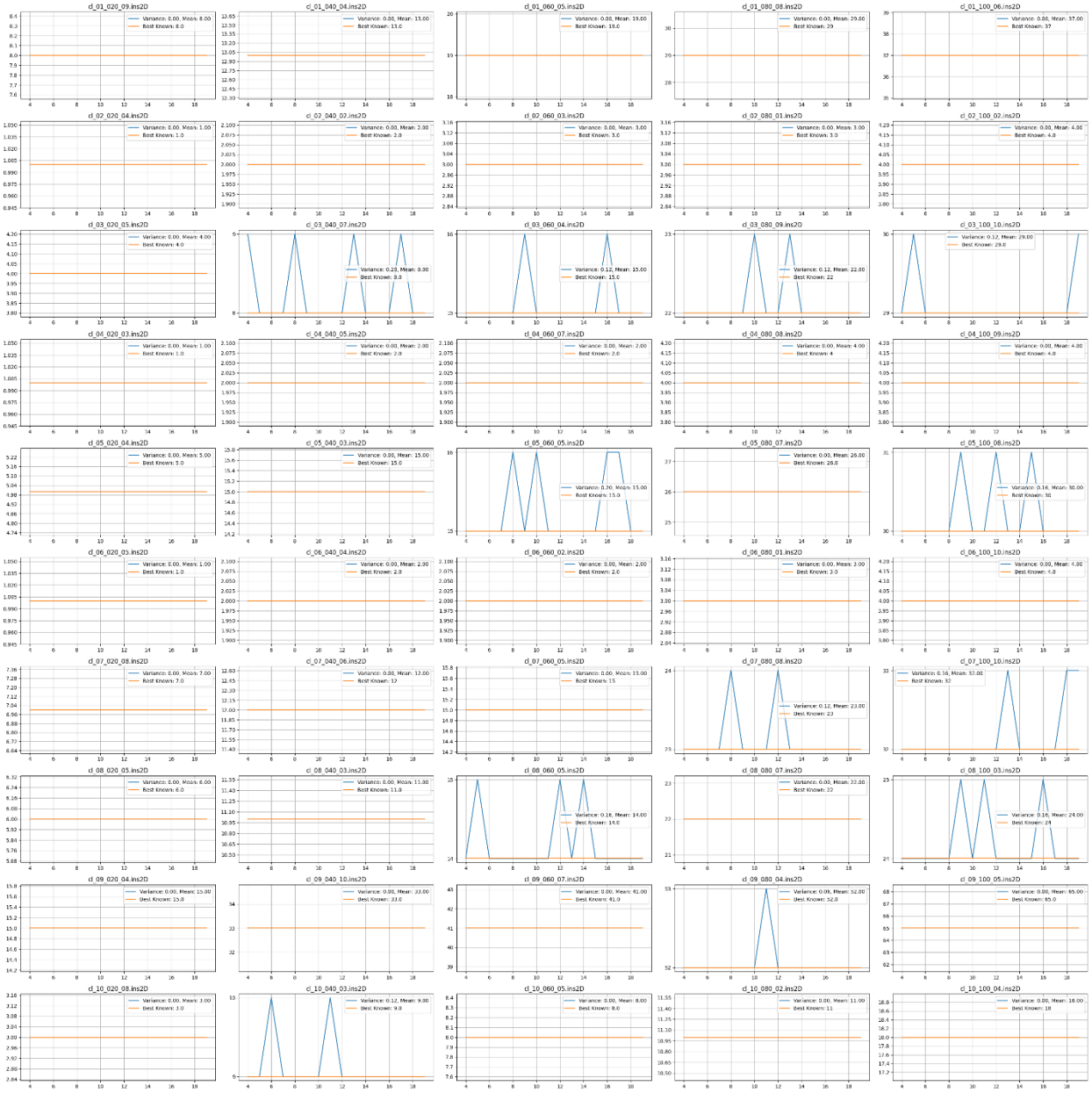
Flowchart:


Initialize Population
Is stopping criterion met?
Yes
End
No
Is new population size < population size?
Yes
No
Is memetic true?
Yes
No
Select Parents - Tournament Selection
Crossover with a probability
Repair the new offsprings with a probability
Mutate each offsprings with a probability
Perform hill climbing on feasible chromosomes
replace the chromosomes in the new population
Survival selection - Elitism
evaluate the offsprings
Add offsprings to new population

Results

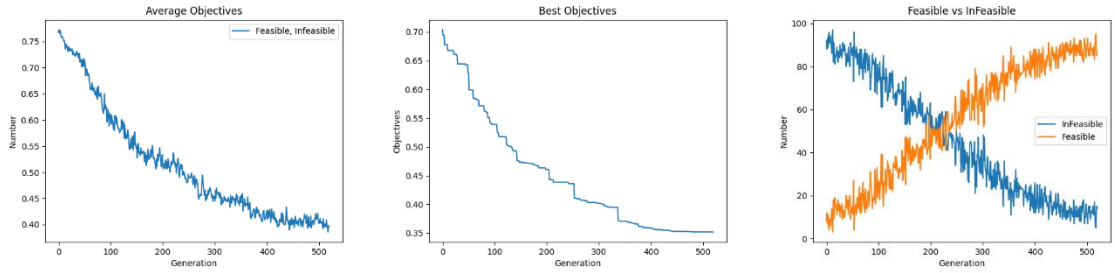
Instance Set	Best Known Solution	Min	Max	Variance	Mean	Gap	Time
cl_01_020_09.ins2D	8	8	8	0	8	0	0
cl_01_040_04.ins2D	13	13	13	0	13	0	16
cl_01_060_05.ins2D	19	19	19	0	19	0	2
cl_01_080_08.ins2D	29	29	29	0	29	0	12
cl_01_100_06.ins2D	37	37	37	0	37	0	12
cl_02_020_04.ins2D	1	1	1	0	1	0	0
cl_02_040_02.ins2D	2	2	2	0	2	0	0
cl_02_060_03.ins2D	3	3	3	0	3	0	0
cl_02_080_01.ins2D	3	3	3	0	3	0	12
cl_02_100_02.ins2D	4	4	4	0	4	0	11
cl_03_020_05.ins2D	4	4	4	0	4	0	0
cl_03_040_07.ins2D	8	8	9	0.2	8	0	4
cl_03_060_04.ins2D	15	15	16	0.12	15	0	21
cl_03_080_09.ins2D	22	22	23	0.12	22	0	40
cl_03_100_10.ins2D	29	29	30	0.12	29	0	72
cl_04_020_03.ins2D	1	1	1	0	1	0	0
cl_04_040_05.ins2D	2	2	2	0	2	0	0
cl_04_060_07.ins2D	2	2	2	0	2	0	100
cl_04_080_08.ins2D	4	4	4	0	4	0	1
cl_04_100_09.ins2D	4	4	4	0	4	0	3
cl_05_020_04.ins2D	5	5	5	0	5	0	0
cl_05_040_03.ins2D	15	15	15	0	15	0	4
cl_05_060_05.ins2D	15	15	16	0.2	15	0	8
cl_05_080_07.ins2D	26	26	26	0	26	0	41
cl_05_100_08.ins2D	30	30	31	0.16	30	0	33
cl_06_020_05.ins2D	1	1	1	0	1	0	0
cl_06_040_04.ins2D	2	2	2	0	2	0	0
cl_06_060_02.ins2D	2	2	2	0	2	0	0
cl_06_080_01.ins2D	3	3	3	0	3	0	13
cl_06_100_10.ins2D	4	4	4	0	4	0	0
cl_07_020_08.ins2D	7	7	7	0	7	0	0
cl_07_040_06.ins2D	12	12	12	0	12	0	6
cl_07_060_05.ins2D	15	15	15	0	15	0	109
cl_07_080_08.ins2D	23	23	24	0.12	23	0	216
cl_07_100_10.ins2D	32	32	33	0.16	32	0	72
cl_08_020_05.ins2D	6	6	6	0	6	0	0
cl_08_040_03.ins2D	11	11	11	0	11	0	0
cl_08_060_05.ins2D	14	14	15	0.16	14	0	1
cl_08_080_07.ins2D	22	22	22	0	22	0	4
cl_08_100_03.ins2D	24	24	25	0.16	24	0	17
cl_09_020_04.ins2D	15	15	15	0	15	0	0
cl_09_040_10.ins2D	33	33	33	0	33	0	0
cl_09_060_07.ins2D	41	41	41	0	41	0	0
cl_09_080_04.ins2D	52	52	53	0.06	52	0	1
cl_09_100_05.ins2D	65	65	65	0	65	0	4
cl_10_020_08.ins2D	3	3	3	0	3	0	0
cl_10_040_03.ins2D	9	9	10	0.12	9	0	0
cl_10_060_05.ins2D	8	8	8	0	8	0	4
cl_10_080_02.ins2D	11	11	11	0	11	0	27
cl_10_100_04.ins2D	18	18	18	0	18	0	33

Plot for different Runs



Sample plot for algorithm performance

\cl_05_080_07:



The three plots provide a comprehensive view of the genetic algorithm's performance on a 2D bin packing problem over 500 generations.

The first plot, which shows the average objectives of all chromosomes, indicates a steady decrease in average objective values, demonstrating consistent improvement in packing efficiency, despite including both feasible and infeasible solutions. This suggests that even as infeasible solutions are increasingly penalized, overall solution quality continues to improve.

The second plot focuses on the best objectives and reveals a smooth, steady decrease, highlighting the algorithm's ability to continuously find and refine the best solutions.

Finally, the third plot contrasts the number of feasible versus infeasible chromosomes, showing that although the algorithm starts with more infeasible solutions (attributable to the nature of crossover rather than the initial population, as initially 95% of the chromosomes are feasible), it successfully evolves to produce more feasible than infeasible solutions by the halfway mark. This underlines the algorithm's effectiveness in navigating the solution space towards feasibility.

In the initial stages of the search, which need to be more explorative, allowing a higher number of infeasible solutions can be beneficial. This approach helps the algorithm reach unvisited parts of the search space, which is crucial for finding the global optima.

Ant Colony

Representation:

In the ACO algorithm, each ant is represented by a permutation of items. Due to the vast number of potential positions for items within each bin, the graph becomes excessively large. Consequently, the position of each item does not evolve within this algorithm.

In bin packing problems, the Ant Colony Optimization (ACO) algorithm faces significant challenges due to the exponential growth in potential item positions within bins, leading to an excessively large graph. This vast graph complicates the algorithm's core operations: updating pheromones and computing state transitions become computationally burdensome due to the sheer number of nodes and edges. Specifically, the time complexity associated with recalculating pheromone levels across millions of potential pathways and determining probabilities for transitioning between these numerous states makes real-time or efficient processing impractical. Additionally, the large size of the state space impedes the ants' ability to sufficiently explore and reinforce the most promising paths, severely hindering the algorithm's convergence and effectiveness. Using a permutation of items as the representation for each ant in Ant Colony Optimization (ACO) simplifies the approach. This method allows for optimizing the placement of items without directly addressing each possible position, thereby managing the combinatorial explosion of the state space.

4	7	9	2	3	5	1	8	6

In the context of graph representation for solving combinatorial optimization problems such as Bin Packing, each item in the problem is represented as a node within the graph. The edges between these nodes are weighted with pheromone levels, which play a critical role in the Ant Colony Optimization (ACO) algorithm. This pheromone acts as a form of indirect communication among the ants (agents in the ACO algorithm), which guides their search towards more promising regions of the solution space. Specifically, the pheromone levels on the edges increase as more ants follow that path, indicating a potentially better solution. Over time, this results in a higher probability of the path being chosen by other ants, thereby reinforcing successful solutions and gradually leading to the optimal or near-optimal solutions to this NP-hard problem.

Solution Construction:

1. LF_construction:
   The purpose of this process is to construct a packing solution using the LF strategy, where each item is placed in the most recently utilized container that can accommodate it, or a new container is initiated if it cannot. The process begins by initializing a list with one empty container and copying the list of rectangles to be placed. The first item is placed in the initial container, setting it as the current item. Subsequently, items are iteratively selected based on the 'ACS_state_transition' function, which uses pheromone trails to determine the sequence of placement. Each selected item is attempted to be placed in the last used container; if it doesn't fit, a new container is created for it. This procedure continues until all items are placed, adhering to the LF strategy by using the most recent container whenever possible.

2. FF_construction:
   The "First Fit" approach constructs a packing solution by starting with a single container and iteratively placing each subsequent item into the first available container that has sufficient space to accommodate it. If an item cannot be placed in any existing container due to space constraints, a new container is initiated. This method, although straightforward and initially using a minimal number of bins compared to LF strategy, often leads to local optima because it does not necessarily place connected items (as defined by state transitions in algorithms like Ant Colony Optimization) in the same bin. This misalignment with ACO's state transition definition, where the placement of one item influences the placement of the next connected item in the same bin, can hinder the overall efficiency and effectiveness of the search process.

   Furthermore, the First Fit strategy, while simple and quick to implement, is not always time-efficient compared to more sophisticated heuristics that can offer better optimization by allowing the algorithm to explore a wider range of configurations. Unlike LF strategy, which can provide the search algorithm with greater flexibility to find optimal configurations by potentially reusing later containers and thus exploring more diverse solutions, First Fit restricts this exploration. This limitation often results in suboptimal packing configurations that do not fully utilize the container space, thereby preventing the algorithm from effectively exploring and exploiting the solution space, aligning poorly with the algorithmic definitions that aim to enhance both explorative and exploitative capabilities in heuristic searches.

Heuristic information:

This parameter shows how well a rectangle fits within a container by considering the remaining available area in the container after placing the rectangle. Specifically, ‘eta’ is calculated as one plus the square of the ratio of the used area (after placing the rectangle) to the total area of the container. This ratio is determined by subtracting the area of the rectangle from the available area in the container to find the remaining space, and then comparing this remaining space to the total container area.

The heuristic value (‘eta’) effectively measures the "goodness" of placing the rectangle in the container, with higher values indicating a better fit (i.e., less wasted space). If the rectangle doesn't fit (resulting in negative remaining space), a very small heuristic value (‘1e-20’) is returned, strongly discouraging the choice of that rectangle.

$$R = Area_{reamining} = Area_{available} - Area_{rectangle}$$

$$R \geq 0$$

$$\eta(rectangle, container) = 1 + \left(\frac{Area_{total} - R}{Area_{total}} \right)^2$$

$$R < 0$$

$$\eta(rectangle, container) = 10^{-20}$$

Global Update Pheromones:

Updating pheromones globally in an Ant Colony System trails in two steps:

1. Evaporation
2. Reinforcement

During evaporation, all pheromone trails are reduced by a factor of (1 - ρ), where ρ is the evaporation rate, to simulate natural decay and prevent the algorithm from converging prematurely on local optima. If a pheromone value drops below a minimum threshold, it's set to this minimum to maintain influence in the trail system. For reinforcement, the function increases the pheromone levels for pairs of rectangles placed in the same container in the provided solution, proportionate to the inverse of the objective function's value, scaled by ρ and adjusted for existing pheromone persistence. This update strategy encourages the algorithm to explore configurations that improve packing efficiency, and the global update after all ants complete their solutions ensures that the most successful patterns have a stronger influence on subsequent iterations.

Local Update Pheromones:

There are two local updates:

1. The local pheromone update occurs after selecting a rectangle for placement (state transition). This update adjusts the pheromone trail using the formula:

$$\tau_{ij} = (1 - \rho) \cdot \tau_{ij} + \rho \cdot \tau_{initial}$$

   Here, τ_{ij} is the pheromone between the current item i and the chosen rectangle j, and ρ is the evaporation rate. This rule reduces the existing pheromone to simulate evaporation and adds a new constant amount to encourage exploration of alternative paths, thus helping to prevent premature convergence on suboptimal solutions.

2. In case the chosen rectangle does not fit in the current bin despite the bin having enough available space to accommodate the rectangle, the algorithm discourages this edge's pheromone.

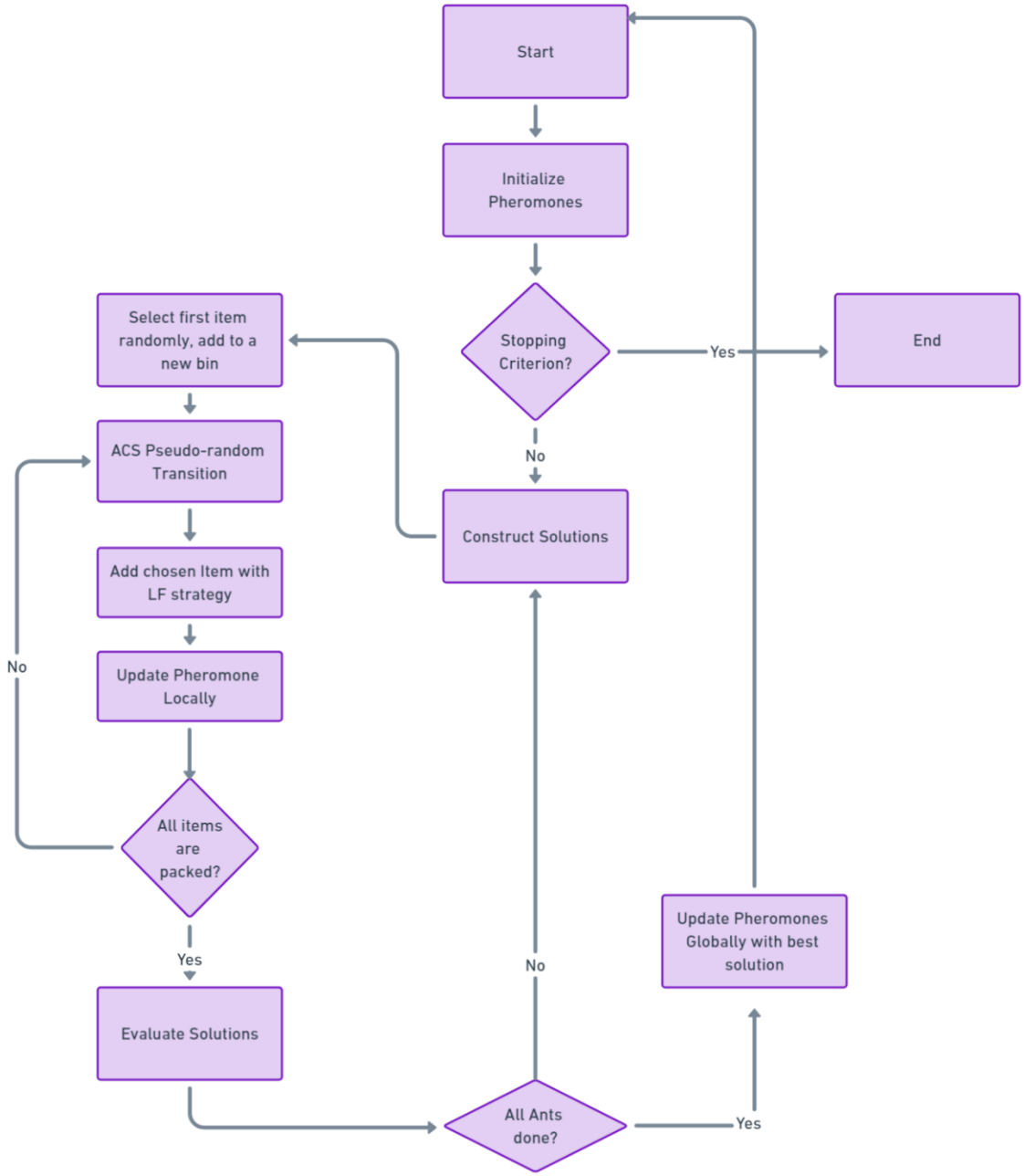
State Transition:

The Ant Colony System (ACS) employs a state transition rule known as the pseudo-random proportional rule, which skillfully balances exploration and exploitation through a defined probability threshold. This strategic decision-making process is pivotal in directing the ants' choices for their next steps in the solution construction phase. Key to this system is the prohibition of repetitive item usage—once an item is used, it becomes a tabu move, ensuring that no item is selected more than once.

$$j = \begin{cases} q \leq q_0 \\ \arg \max_{k \in \text{allowed}} \{\tau_{ik}^{\alpha} \cdot \eta_{ik}^{\beta}\} \end{cases}$$

$$P(i,j) = \begin{cases} q > q_0 \\ \dfrac{\tau_{ij}^{\alpha} \cdot \eta_{ij}^{\beta}}{\sum_{k \in \text{allowed}} \tau_{ik}^{\alpha} \cdot \eta_{ik}^{\beta}} \end{cases}$$

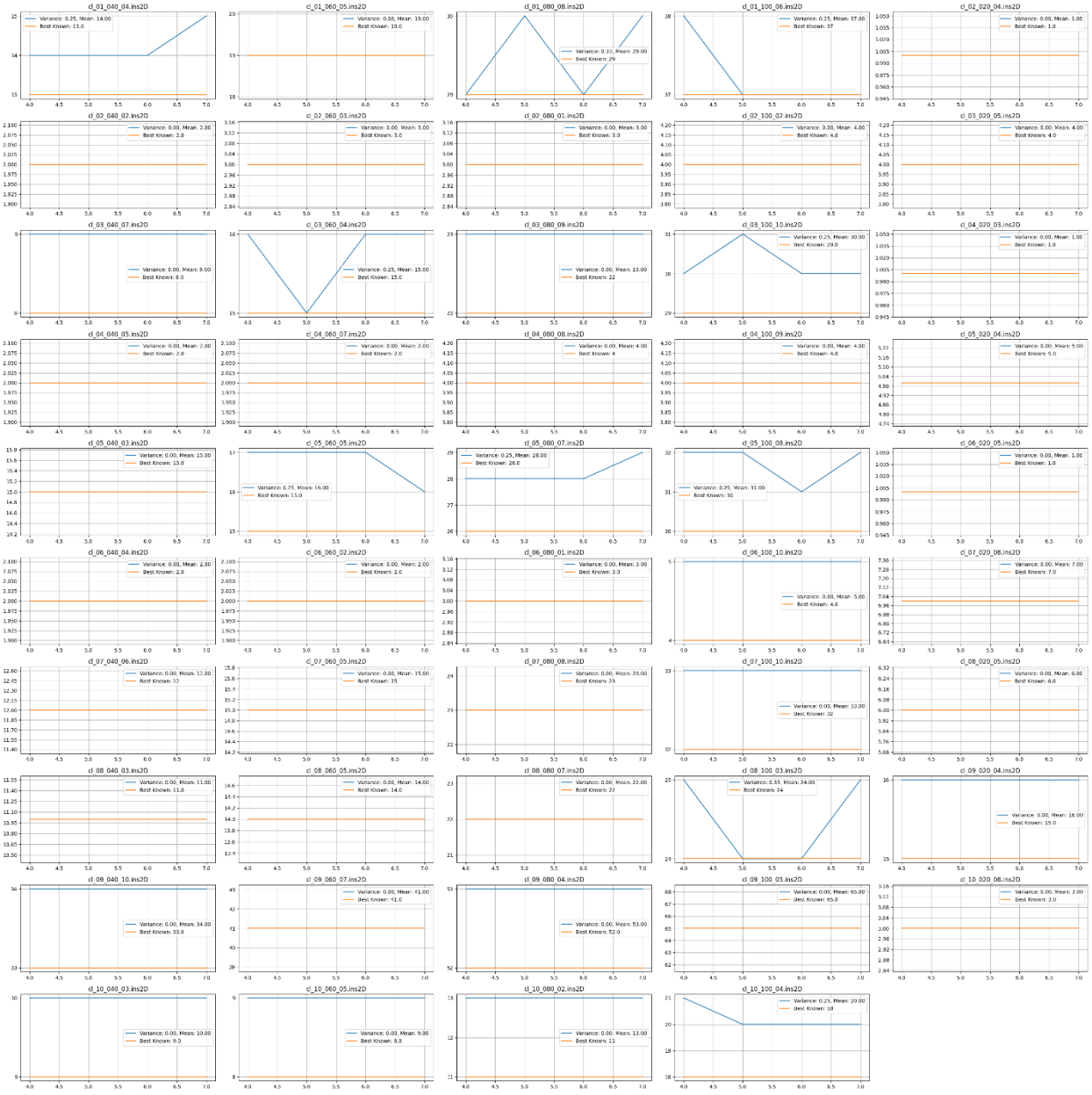
Flowchart


Start
Initialize Pheromones
Stopping Criterion?
Yes
End
No
Construct Solutions
Select first item randomly, add to a new bin
ACS Pseudo-random Transition
Add chosen Item with LF strategy
Update Pheromone Locally
All items are packed?
No
Yes
Evaluate Solutions
All Ants done?
No
Yes
Update Pheromones Globally with best solution

Results

Instance Set	Best Known Solution	Min	Max	Variance	Mean	Gap	Time
cl_01_040_04.ins2D	13	14	15	0.25	14	1	223
cl_01_060_05.ins2D	19	19	19	0	19	0	0
cl_01_080_08.ins2D	29	29	30	0.33	29	0	279
cl_01_100_06.ins2D	37	37	38	0.25	37	0	130
cl_02_020_04.ins2D	1	1	1	0	1	0	0
cl_02_040_02.ins2D	2	2	2	0	2	0	0
cl_02_060_03.ins2D	3	3	3	0	3	0	0
cl_02_080_01.ins2D	3	3	3	0	3	0	0
cl_02_100_02.ins2D	4	4	4	0	4	0	32
cl_03_020_05.ins2D	4	4	4	0	4	0	264
cl_03_040_07.ins2D	8	9	9	0	9	1	226
cl_03_060_04.ins2D	15	15	16	0.25	15	0	84
cl_03_080_09.ins2D	22	23	23	0	23	1	276
cl_03_100_10.ins2D	29	30	31	0.25	30	1	269
cl_04_020_03.ins2D	1	1	1	0	1	0	0
cl_04_040_05.ins2D	2	2	2	0	2	0	0
cl_04_060_07.ins2D	2	2	2	0	2	0	0
cl_04_080_08.ins2D	4	4	4	0	4	0	1
cl_04_100_09.ins2D	4	4	4	0	4	0	24
cl_05_020_04.ins2D	5	5	5	0	5	0	1
cl_05_040_03.ins2D	15	15	15	0	15	0	0
cl_05_060_05.ins2D	15	16	17	0.25	16	1	261
cl_05_080_07.ins2D	26	28	29	0.25	28	2	177
cl_05_100_08.ins2D	30	31	32	0.25	31	1	185
cl_06_020_05.ins2D	1	1	1	0	1	0	0
cl_06_040_04.ins2D	2	2	2	0	2	0	0
cl_06_060_02.ins2D	2	2	2	0	2	0	0
cl_06_080_01.ins2D	3	3	3	0	3	0	0
cl_06_100_10.ins2D	4	5	5	0	5	1	111
cl_07_020_08.ins2D	7	7	7	0	7	0	0
cl_07_040_06.ins2D	12	12	12	0	12	0	50
cl_07_060_05.ins2D	15	15	15	0	15	0	292
cl_07_080_08.ins2D	23	23	23	0	23	0	158
cl_07_100_10.ins2D	32	33	33	0	33	1	279
cl_08_020_05.ins2D	6	6	6	0	6	0	0
cl_08_040_03.ins2D	11	11	11	0	11	0	0
cl_08_060_05.ins2D	14	14	14	0	14	0	244
cl_08_080_07.ins2D	22	22	22	0	22	0	297
cl_08_100_03.ins2D	24	24	25	0.33	24	0	184
cl_09_020_04.ins2D	15	16	16	0	16	1	0
cl_09_040_10.ins2D	33	34	34	0	34	1	1
cl_09_060_07.ins2D	41	41	41	0	41	0	0
cl_09_080_04.ins2D	52	53	53	0	53	1	50
cl_09_100_05.ins2D	65	65	65	0	65	0	0
cl_10_020_08.ins2D	3	3	3	0	3	0	0
cl_10_040_03.ins2D	9	10	10	0	10	1	221
cl_10_060_05.ins2D	8	9	9	0	9	1	74
cl_10_080_02.ins2D	11	13	13	0	13	2	16
cl_10_100_04.ins2D	18	20	21	0.25	20	2	230

Plot for different Runs



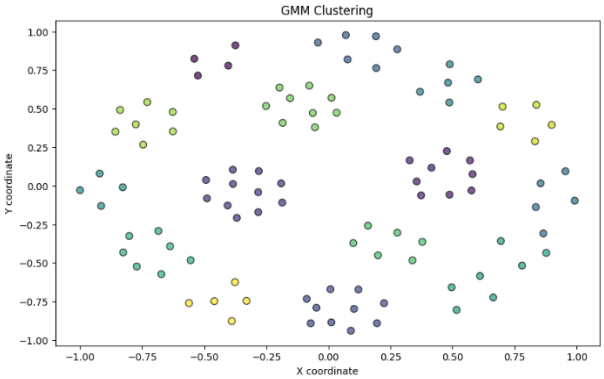
Sample plot for algorithm performance:

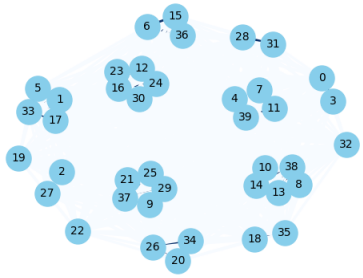
An increase in the number of ants and a decrease in the ACS probability (using the pseudo-random technique in the ACO algorithm) indicate that there has been no improvement for several consecutive iterations. Therefore, by increasing the number of ants and decreasing the probability, the algorithm tries to explore more of the search space.

There is a minimum threshold for ACS probability, which is set at 0.7.

Pheromone Graph and Clustering:

After applying the ACO algorithm to the problem and plotting the graph, each edge is shown with its pheromone value. The larger the pheromone value, the stronger the edge. Additionally, the distance between the nodes connected by each edge is reduced. Each cluster on the graph represents a bin, which demonstrates the effectiveness of the ACO algorithm in solving the problem by adjusting the pheromone trail and using heuristic information. This suggests that a clustering approach could be used after some iterations to explore different configurations, because some edges with strong connections (higher pheromone levels) might not be in the same bin. Using clustering could help capitalize on these kinds of configurations





Particle Swarm

This approach combines Particle Swarm Optimization (PSO) with a 2D bin packing algorithm, incorporating concepts from genetic algorithms[7] and island-based models[8]. In this method, PSO is applied to explore the solution space of the bin packing problem. The key concept of PSO involves particles, each representing a potential solution to the bin packing problem, moving through the solution space. Their movement is influenced by both their personal best position (pbest) and the global best position (gbest) found by the swarm.

Island Model with Migration

The Island Model enhances the PSO framework by dividing the swarm of particles into several subgroups (islands), each evolving independently. This structure helps maintain diversity within the solution space and reduces the risk of premature convergence on suboptimal solutions. Periodically, selected solutions (particles) migrate between islands, facilitating information sharing and improving the search process across the swarm.

Genetic Recombination

A distinctive feature of this approach is the application of genetic recombination, which involves creating a new particle (child) from three parents: pbest, gbest, and the current particle. This process is regulated by parameters c1 (cognitive coefficient), c2 (social coefficient), and w (inertia weight), which dictate the probability of selecting features (e.g., bin selections) from each parent. This recombination simulates the movement of particles and the updating of their positions in the solution space, along with social communication and self-communication of PSO.

Solution Evaluation and Repair

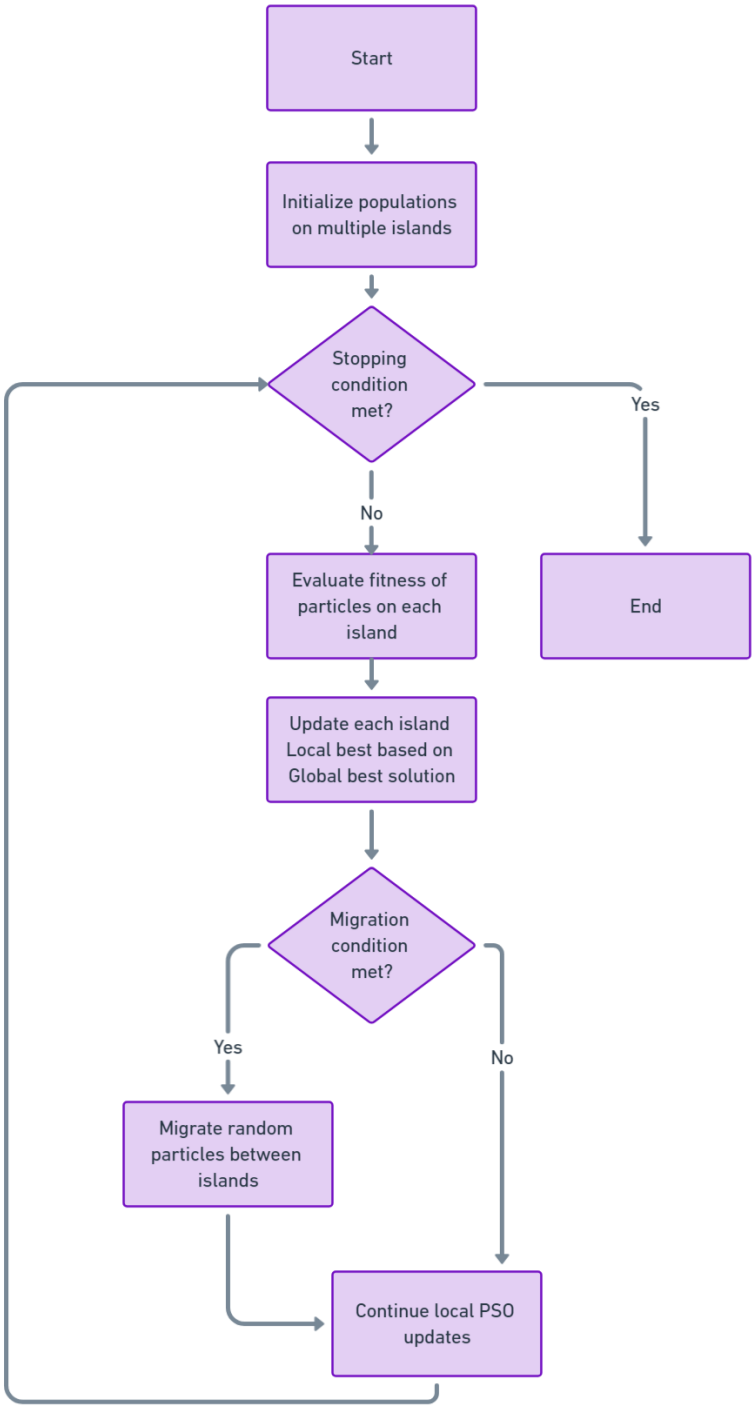
Once a new child solution is generated through recombination, its feasibility is assessed by checking for duplicate items and ensuring all items are appropriately packed. Any issues, such as duplicates or missing items, are addressed in a repair phase, where adjustments are made to conform to the constraints of the bin packing problem. The positions of items within the bins are inherited from the parent whose features were chosen, maintaining consistency in item placement and potentially reducing the need for extensive rearrangement. In this approach, the coordinates x, y (representing the bottom-left corner of each item) also evolve. These coordinates are inherited from three parents. If a duplicate is detected or an item is missing, these specific instances are addressed by using the Place Rectangle module to find appropriate placements for the items within the bins. When placing an item, if the module determines there isn't enough space in the current bin, the item may be reinserted into a different bin or a new bin may be created. This repositioning leads to an implicit evolution of the x, y coordinates. By ensuring items are properly placed and fit within the available bins, this method maintains that all solutions are feasible, thus preventing any infeasible solutions.

---

[7] Borna, K., & Khezri, R. (2015). A combination of genetic algorithm and particle swarm optimization method for solving traveling salesman problem. *Cogent Mathematics*, *2*(1), 1048581. doi: 10.1080/23311835.2015.1048581

[8] Abadlia, H., Smairi, N., & Ghedira, K. (2017). Particle Swarm Optimization based on Island Models. *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO'17)*, Berlin, Germany. doi:10.1145/3067695.3076068.

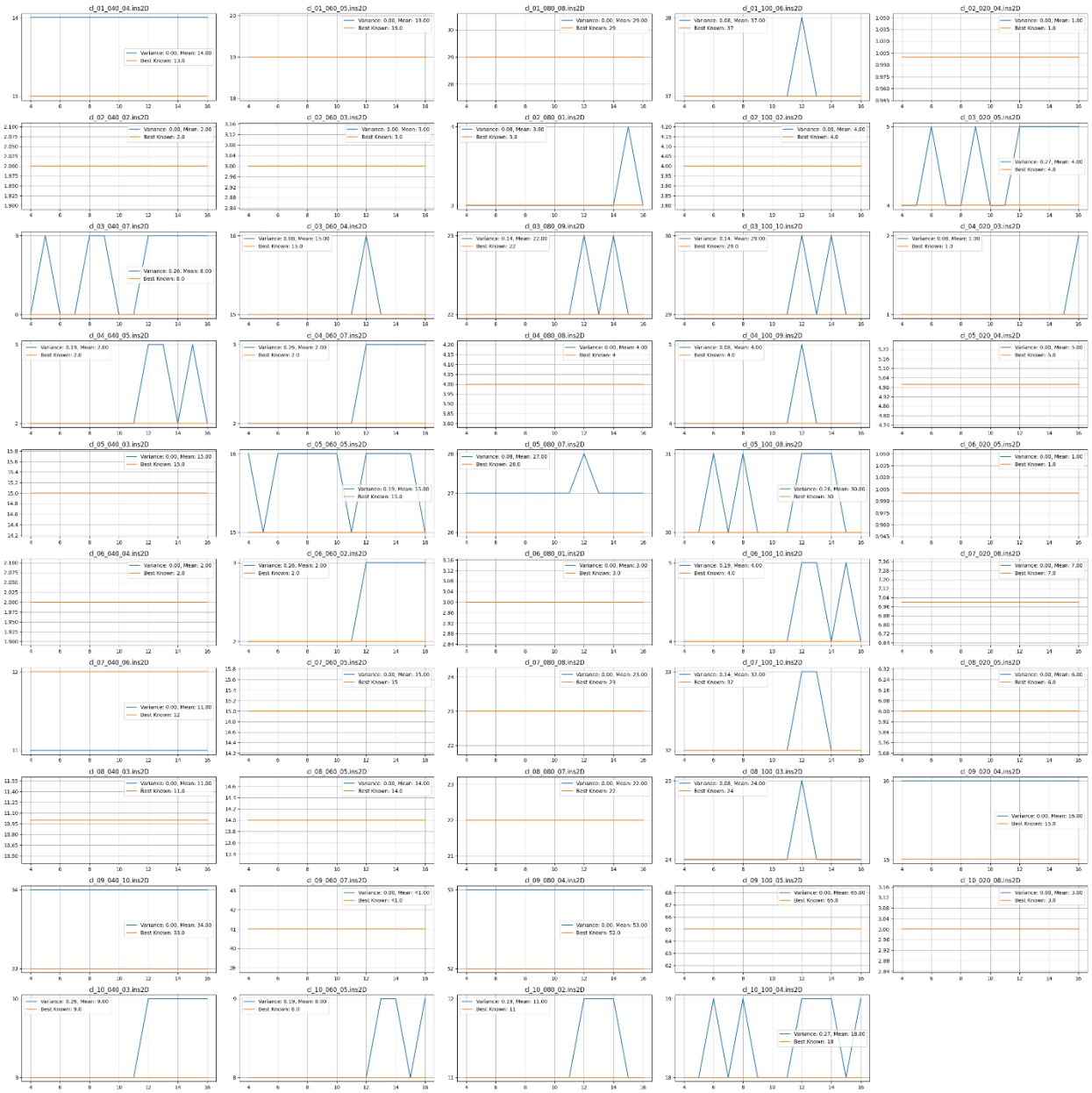
Flowchart:


Start
Initialize populations on multiple islands
Stopping condition met?
Yes
No
End
Evaluate fitness of particles on each island
Update each island Local best based on Global best solution
Migration condition met?
Yes
No
Migrate random particles between islands
Continue local PSO updates

Results

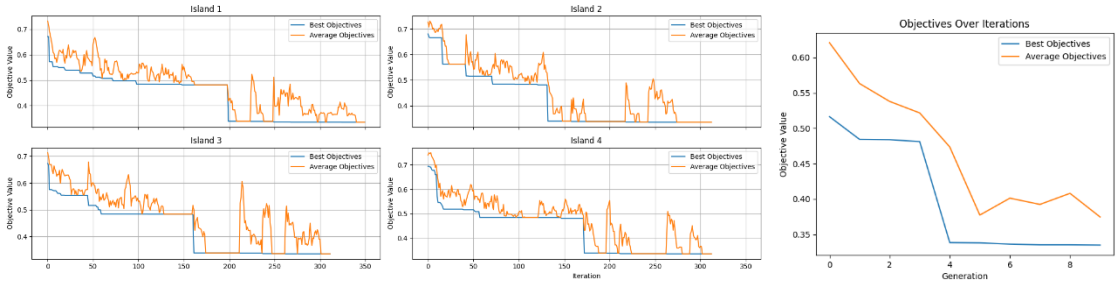
Instance Set	Best Known Solution	Min	Max	Variance	Mean	Gap	Time
cl_01_040_04.ins2D	13	14	14	0	14	1	4
cl_01_060_05.ins2D	19	19	19	0	19	0	0
cl_01_080_08.ins2D	29	29	29	0	29	0	286
cl_01_100_06.ins2D	37	37	38	0.08	37	0	300
cl_02_020_04.ins2D	1	1	1	0	1	0	0
cl_02_040_02.ins2D	2	2	2	0	2	0	0
cl_02_060_03.ins2D	3	3	3	0	3	0	0
cl_02_080_01.ins2D	3	3	4	0.08	3	0	1
cl_02_100_02.ins2D	4	4	4	0	4	0	2
cl_03_020_05.ins2D	4	4	5	0.27	4	0	0
cl_03_040_07.ins2D	8	8	9	0.26	8	0	1
cl_03_060_04.ins2D	15	15	16	0.08	15	0	7
cl_03_080_09.ins2D	22	22	23	0.14	22	0	300
cl_03_100_10.ins2D	29	29	30	0.14	29	0	6
cl_04_020_03.ins2D	1	1	2	0.08	1	0	0
cl_04_040_05.ins2D	2	2	3	0.19	2	0	0
cl_04_060_07.ins2D	2	2	3	0.26	2	0	0
cl_04_080_08.ins2D	4	4	4	0	4	0	7
cl_04_100_09.ins2D	4	4	5	0.08	4	0	2
cl_05_020_04.ins2D	5	5	5	0	5	0	0
cl_05_040_03.ins2D	15	15	15	0	15	0	0
cl_05_060_05.ins2D	15	15	16	0.19	15	0	27
cl_05_080_07.ins2D	26	27	28	0.08	27	1	266
cl_05_100_08.ins2D	30	30	31	0.26	30	0	297
cl_06_020_05.ins2D	1	1	1	0	1	0	0
cl_06_040_04.ins2D	2	2	2	0	2	0	0
cl_06_060_02.ins2D	2	2	3	0.26	2	0	2
cl_06_080_01.ins2D	3	3	3	0	3	0	1
cl_06_100_10.ins2D	4	4	5	0.19	4	0	5
cl_07_020_08.ins2D	7	7	7	0	7	0	0
cl_07_040_06.ins2D	12	11	11	0	11	-1	6
cl_07_060_05.ins2D	15	15	15	0	15	0	83
cl_07_080_08.ins2D	23	23	23	0	23	0	296
cl_07_100_10.ins2D	32	32	33	0.14	32	0	248
cl_08_020_05.ins2D	6	6	6	0	6	0	0
cl_08_040_03.ins2D	11	11	11	0	11	0	0
cl_08_060_05.ins2D	14	14	14	0	14	0	2
cl_08_080_07.ins2D	22	22	22	0	22	0	288
cl_08_100_03.ins2D	24	24	25	0.08	24	0	301
cl_09_020_04.ins2D	15	16	16	0	16	1	0
cl_09_040_10.ins2D	33	34	34	0	34	1	6
cl_09_060_07.ins2D	41	41	41	0	41	0	0
cl_09_080_04.ins2D	52	53	53	0	53	1	280
cl_09_100_05.ins2D	65	65	65	0	65	0	0
cl_10_020_08.ins2D	3	3	3	0	3	0	0
cl_10_040_03.ins2D	9	9	10	0.26	9	0	1
cl_10_060_05.ins2D	8	8	9	0.19	8	0	6
cl_10_080_02.ins2D	11	11	12	0.19	11	0	26
cl_10_100_04.ins2D	18	18	19	0.27	18	0	101

Plot for different Runs



Sample plot for algorithm performance:

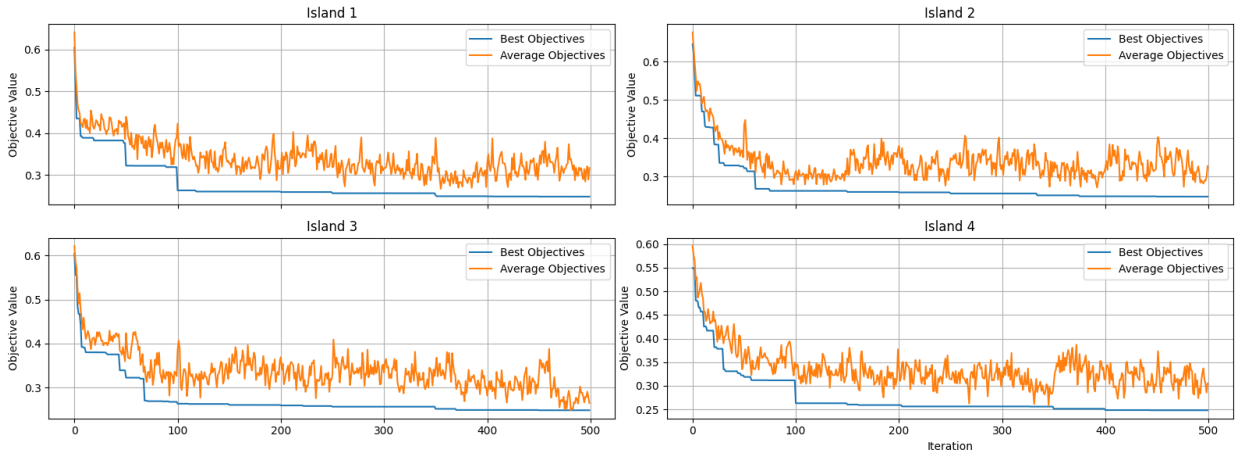
cl_06_100_10:



The provided plots demonstrate the IslandPSO algorithm's effectiveness in managing fast convergence and avoiding local optima on an easy instance set. Key observations include a steep initial improvement in objective values, indicating rapid solution discovery. The migration operator plays a crucial role, periodically injecting new genetic material into each island, which is evident from sudden improvements or stability in objective values at specific intervals. This continuous introduction of diversity prevents the premature convergence typical of traditional PSO and sustains exploration across generations. Each island shows unique trends, suggesting varied exploration and exploitation strategies or different efficacies in assimilating new genetic material. Overall, the strategic use of migration ensures a balance between exploration and exploitation, maintaining genetic diversity and enhancing the algorithm's overall performance by keeping the population dynamic and responsive to new opportunities.

The second plot illustrates overall improvements in the IslandPSO algorithm, showcasing both the enhancement of the global best solution and the average objectives across particles over time. This indicates not only effective individual performance but also a consistent uplift in the collective output of the swarm. The periodic injection of new genetic material through the migration operator is key to this success, ensuring diversity and preventing the fast convergence issues common in traditional PSO. By maintaining a dynamic and explorative environment across the islands, the algorithm successfully balances exploration with exploitation, leading to progressive and sustained improvements in performance.

Analysis of Objective Value Trends Across Islands in IslandPSO



The provided plots depict the evolution of best and average objective values across four islands in an IslandPSO setup over 500 iterations, with migrations occurring every 50 iterations. Key observations include:

1. Initial Improvement: Each island demonstrates a rapid improvement in objective values initially, signaling efficient exploration of the solution space at the outset.

2. Impact of Migration: A notable enhancement in best objective values around iteration 100, particularly influenced by Island 2, indicates successful migration and sharing of superior solutions, boosting performance across the islands except for Island 2 which already had a superior solution.

3. Continued Exploration and Stabilization: After iteration 100, the objective values show fluctuations, reflecting persistent exploration. By iteration 300, these values start to converge, suggesting a stabilization in solution quality with diminishing returns on new discoveries.

4. Island-Specific Dynamics: Island 1 and Island 3 exhibit similar trends of steady improvement and stabilization. Island 2 stands out as a strong performer, particularly influencing other islands' performance around iteration 100. Island 4 demonstrates a more consistent and gradual improvement, indicating a steady exploration and exploitation strategy.

Overall, the migration strategy effectively promotes global search efficacy by distributing high-quality solutions, enhancing local search capabilities, and helping overcome local optima. This dynamic facilitates robust exploration and optimization, proving particularly effective for complex challenges like bin packing. The strategic timing and frequency of migrations are crucial, highlighting their role in preventing stagnation and enhancing the algorithm's overall performance.

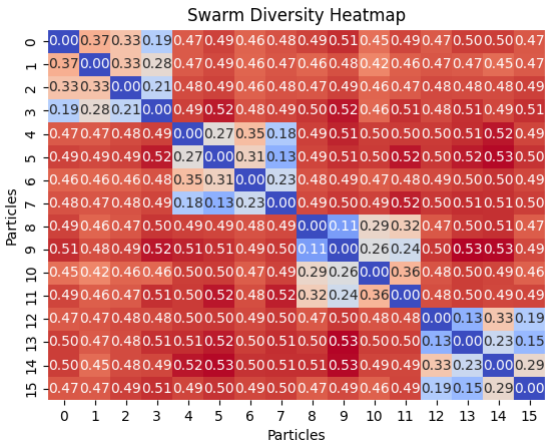


Figure 2 no migration scenarion

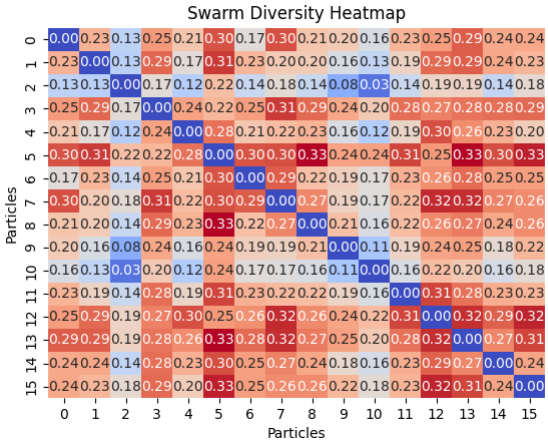
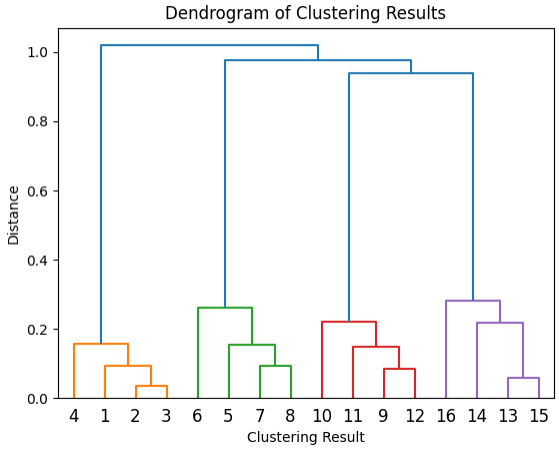


Figure 3 migration scenario

The heatmaps depict the effects of migration policies in an IslandPSO setup on swarm diversity during optimization tasks, like bin packing. In the no migration scenario, diversity (calculated by NMI) within islands is low, showing convergence towards similar solutions due to the lack of external influences, resulting in a suboptimal 21 bins outcome. Conversely, the scenario with frequent migration every 2 out of 10 iterations exhibits uniform diversity across islands, promoting effective solution integration and balanced exploration-exploitation. This setup not only prevents stagnation at local optima but also improves the packing result to 19 bins, demonstrating that regular genetic mixing enhances optimization performance by maintaining a healthy diversity level and encouraging comprehensive exploration of the solution space.

Potential Enhancement through Strategic Particle Regrouping Based on Clustering Analysis

In swarm optimization, clustering results visualized through a dendrogram can be used to enhance the swarm's exploration capabilities by regrouping particles into new sub-swarms based on their similarities. By identifying distinct clusters and systematically reassigning one particle from each cluster to form diverse sub-swarms, this strategy promotes diversity, improves the balance between exploration and exploitation, and prevents premature convergence. This regrouping can be triggered during periods of stagnation or lack of improvement, dynamically adapting to the evolving solution space and potentially leading to more effective problem-solving in complex environments like 2D bin packing.



Comparison

Solution Quality:

ARPD in metaheuristic algorithm analysis measures the deviation of solutions obtained from the algorithm from a known optimal or benchmark solution, expressed as a percentage. This metric helps assess the algorithm's performance consistency and accuracy across multiple runs or problem instances.

$$\text{ARPD} = \frac{100}{n} \sum_{i=1}^{n} \left(\frac{|f_i - f^*|}{|f^*|} \right)$$

- 0% ARPD indicates perfect performance, where the algorithm always finds the optimal or best-known solution for every instance.

- Greater than 0% indicates some degree of deviation from the optimal solutions. The higher the ARPD, the greater the deviation.

Time-Quality:

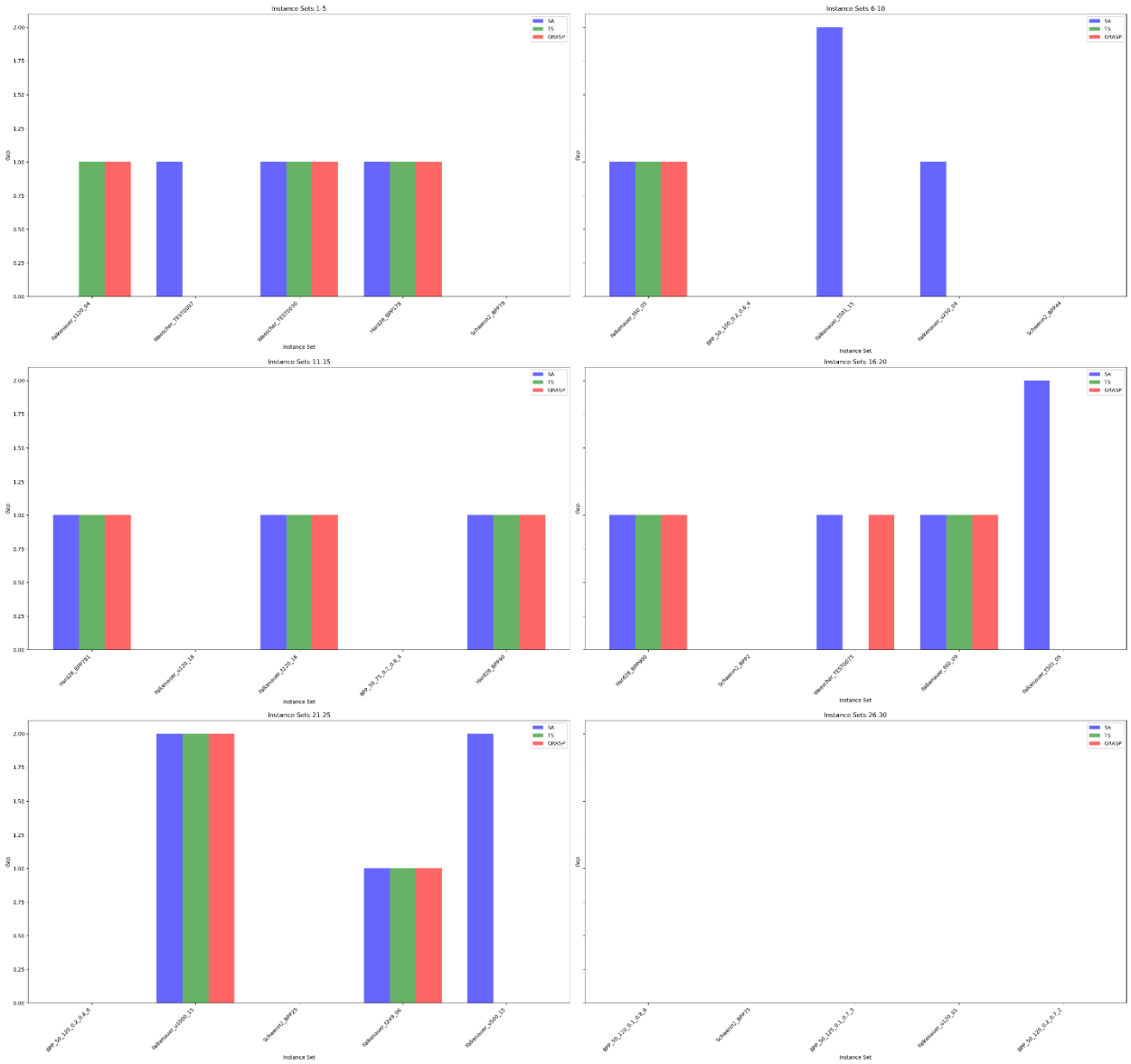
This metric is a weighted sum of the logarithmic transformations of solution quality and computational time, with both weights, ‘alpha’ and ‘beta’, set to 0.5, indicating equal importance to both factors. Specifically, logarithm of one plus the value is applied to both the mean solution quality ("Mean") and the computational time ("Time"). This transformation helps in normalizing the data, making it more comparable and stable across different datasets. The resulting "tm_metric" serves as a composite indicator of both the effectiveness (solution quality) and efficiency (computational time) of the algorithms, providing a single value that facilitates easier comparison of their overall performance.

Borda count voting method: A method having a rank o_k is given o_k points. Given the m experiments, each method sums its points o_k to compute its total score. Then, the methods are classified according to their scores.

Single Solution base algorithms:

Solution Quality Comparison:

Gap Comparison Across Algorithms:



Calculated based on ARPD:

1. SA: 0.015
2. TS: 0.009
3. GRASP: 0.011

Evaluation of Time – Quality with Borda Count Voting method:

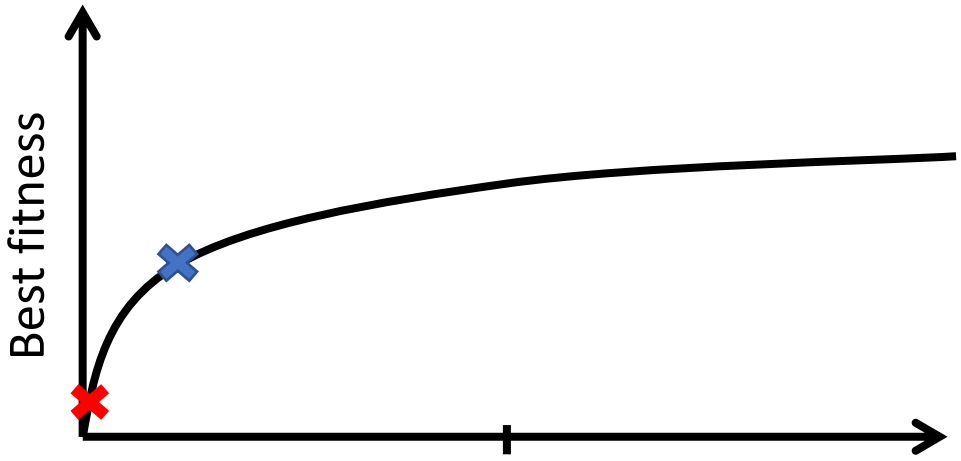
Instance Set	SA	TS	GRASP
Falkenauer_t120_04	2	1	3
Waescher_TEST0097	2	1	3
Waescher_TEST0030	2	1	3
Hard28_BPP178	3	1	2
Schwerin2_BPP78	1	1	1
Falkenauer_t60_05	2	1	3
BPP_50_100_0.2_0.8_4	1	1	1
Falkenauer_t501_15	3	2	1
Falkenauer_u250_04	3	1	2
Schwerin2_BPP44	1	1	1
Hard28_BPP781	3	1	2
Falkenauer_u120_18	3	1	1
Falkenauer_t120_16	2	1	3
BPP_50_75_0.2_0.8_4	1	1	1
Hard28_BPP40	3	1	2
Hard28_BPP900	2	1	3
Schwerin2_BPP2	1	1	1
Waescher_TEST0075	2	1	3
Falkenauer_t60_09	2	1	3
Falkenauer_t501_05	3	2	1
BPP_50_120_0.2_0.8_6	1	1	1
Falkenauer_u1000_15	2	1	3
Schwerin2_BPP25	1	1	1
Falkenauer_t249_06	2	1	3
Falkenauer_u500_15	3	2	1
BPP_50_120_0.1_0.8_6	1	1	1
Schwerin2_BPP75	1	1	1
BPP_50_125_0.1_0.7_3	1	1	3
Falkenauer_u120_01	3	1	1
BPP_50_120_0.2_0.7_2	1	1	1
Sum	58	33	56

1. Tabu: 33
2. SA: 58
3. GRASP: 56

Discussion:

	ARPD	Time-quality vote
Simulated Annealing	0.015	58
Tabu Search	0.009	33
GRASP	0.011	56

This result demonstrates that the TS algorithm excels in terms of both time efficiency and solution quality, primarily due to its additional memory that prevents cyclic actions and compels the search procedure to cover a broader area of the search space. While GRASP was expected to surpass other algorithms owing to its initial construction of high-quality solutions, the metaheuristic algorithms' plots suggest that achieving a good starting point in the search process does not require excessive time (blue point time minus red point time). Therefore, it is more advantageous to employ an algorithm that ensures ample exploration of the search space.

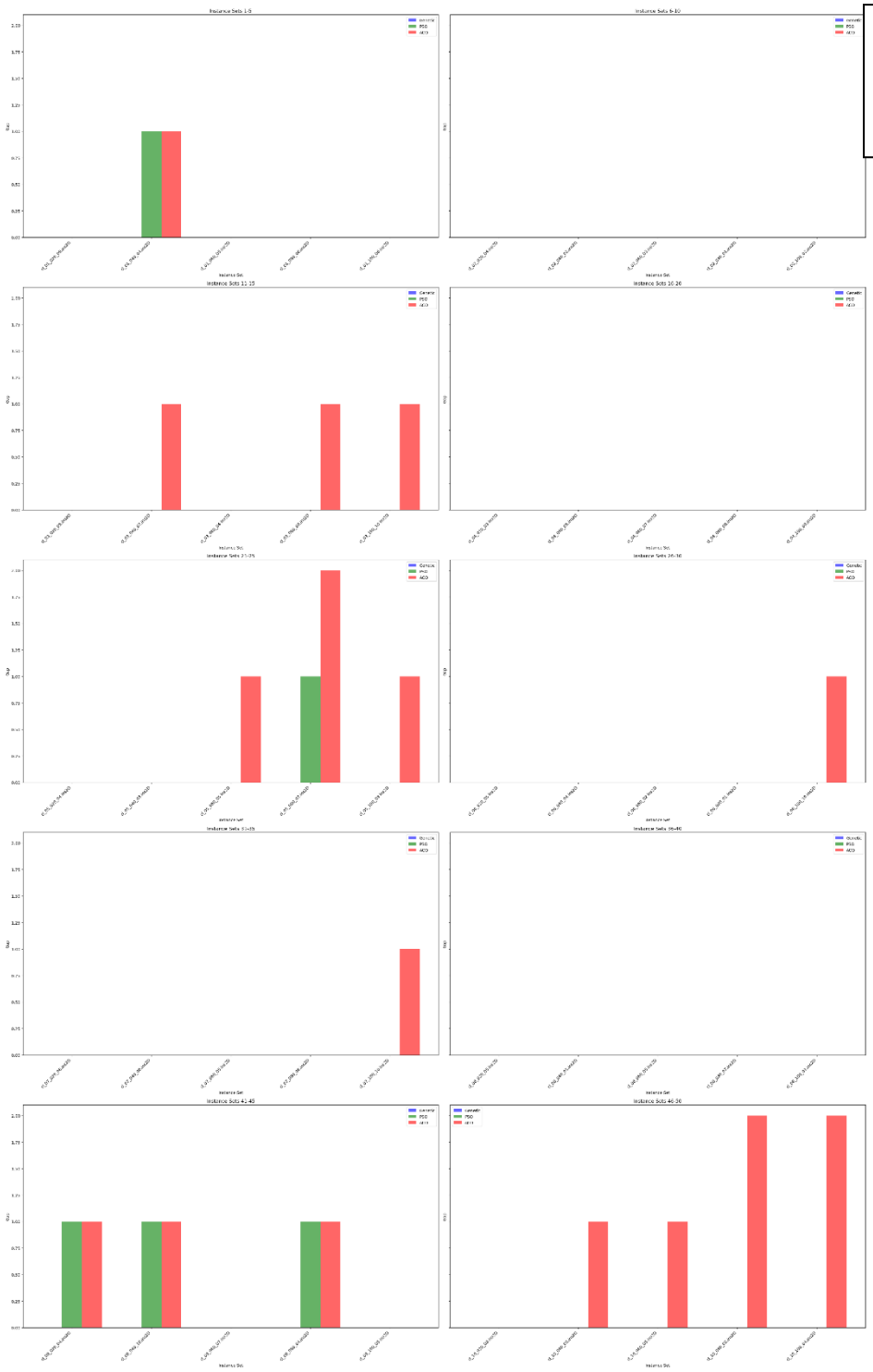


SA algorithm although has enough exploration due to its reheating process, but may take cyclic actions and increase run time, however, may in long run be able to find optimal solution.

Population-based:

Solution Quality Comparison:

Gap Comparison Across Algorithms:



Calculated based on ARPD:

1. Genetic: 0.001
2. PSO: 0.005
3. ACO: 0.028

Evaluation of Time – Quality with Borda Count Voting method:

Instance Set	Genetic	PSO	ACO
cl_01_020_09.ins2D	1	1	1
cl_01_040_04.ins2D	2	1	3
cl_01_060_05.ins2D	3	1	1
cl_01_080_08.ins2D	1	3	2
cl_01_100_06.ins2D	1	3	2
cl_02_020_04.ins2D	1	1	1
cl_02_040_02.ins2D	1	1	1
cl_02_060_03.ins2D	1	1	1
cl_02_080_01.ins2D	3	2	1
cl_02_100_02.ins2D	2	1	3
cl_03_020_05.ins2D	1	1	3
cl_03_040_07.ins2D	2	1	3
cl_03_060_04.ins2D	2	1	3
cl_03_080_09.ins2D	1	3	2
cl_03_100_10.ins2D	2	1	3
cl_04_020_03.ins2D	1	1	1
cl_04_040_05.ins2D	1	1	1
cl_04_060_07.ins2D	3	1	1
cl_04_080_08.ins2D	1	3	1
cl_04_100_09.ins2D	2	1	3
cl_05_020_04.ins2D	1	1	3
cl_05_040_03.ins2D	3	1	1
cl_05_060_05.ins2D	1	2	3
cl_05_080_07.ins2D	1	3	2
cl_05_100_08.ins2D	1	3	2
cl_06_020_05.ins2D	1	1	1
cl_06_040_04.ins2D	1	1	1
cl_06_060_02.ins2D	1	3	1
cl_06_080_01.ins2D	3	2	1
cl_06_100_10.ins2D	1	2	3
cl_07_020_08.ins2D	1	1	1
cl_07_040_06.ins2D	2	1	3
cl_07_060_05.ins2D	2	1	3
cl_07_080_08.ins2D	2	3	1
cl_07_100_10.ins2D	1	2	3
cl_08_020_05.ins2D	1	1	1
cl_08_040_03.ins2D	1	1	1
cl_08_060_05.ins2D	1	2	3
cl_08_080_07.ins2D	1	2	3
cl_08_100_03.ins2D	1	3	2
cl_09_020_04.ins2D	1	2	2
cl_09_040_10.ins2D	1	3	2
cl_09_060_07.ins2D	1	1	1
cl_09_080_04.ins2D	1	3	2
cl_09_100_05.ins2D	3	1	1
cl_10_020_08.ins2D	1	1	1
cl_10_040_03.ins2D	1	2	3
cl_10_060_05.ins2D	1	2	3
cl_10_080_02.ins2D	3	2	1
cl_10_100_04.ins2D	1	2	3
Sum	73	84	95

Evaluation Based on Borda:

1. Genetic 73
2. PSO 84
3. ACO 95

Discussion:

	ARPD	Time-quality vote
PSO	0.005	84
Genetic Algorithm	0.001	73
ACO	0.028	95

In the implemented genetic algorithm, the exploration of infeasible search spaces is allowed, which is not the case with other algorithms such as PSO (Particle Swarm Optimization) and ACO (Ant Colony Optimization); these are restricted to feasible search spaces only. One reason the genetic algorithm outperforms PSO and ACO is that exploring infeasible solutions can introduce unique and valuable genetic material. Additionally, the use of a penalty function and a probabilistic repairing mechanism effectively manages infeasible solutions, leading the genetic algorithm to rank first. Moreover, the genetic algorithm features diverse operators, including adaptive mutations that allow for large and small steps under different conditions, an option not available in the other algorithms.

Traditional PSO tends to converge quickly to local solutions. However, Island PSO maintains diversity within the population. PSO, ranking second, although implemented PSO incorporates genetic recombination, PSO's inherent continuity poses limitations when applied to discrete problems, such as bin packing.

In the case of ACO, relaxing constraints on the evolution of positions (denoted as x, y) has resulted in the algorithm getting stuck in local optima, thereby preventing it from finding better solutions. Also, for ACO, it was harder to tune the hyperparameters, so ACO for this problem is not robust enough compared to Genetic or PSO algorithm, for example heuristic information directly influences the search procedure. ACO algorithm is better to be used for problems with Graph base nature.

Conclusion:

When addressing the bin packing problem, various algorithms demonstrate distinct strengths and weaknesses depending on their approach to exploration and optimization.

1. Population-Based Algorithms:

- Pros: Their ability to explore multiple solutions simultaneously makes them less likely to get trapped in local optima, enhancing their performance on complex problems over the long term.

- Cons: They often exhibit slower initial performance due to size of population compared to single solution-Based algorithms.

2. Genetic Algorithm (GA):

- Strengths: GA stands out due to its ability to explore infeasible search spaces, introducing valuable genetic material through a penalty function and probabilistic repairing mechanism. Adaptive mutations also enable flexible adjustments under varying conditions.

- Performance: GA ranks first, outperforming PSO and ACO due to its robust handling of infeasible solutions and diverse operators.

3. Particle Swarm Optimization (PSO):

- Strengths: Island PSO maintains population diversity, which helps avoid quick convergence to local solutions.

- Limitations: Traditional PSO converges quickly to local solutions, and its continuous nature poses challenges for discrete problems like bin packing. Despite incorporating genetic recombination, PSO ranks second.

4. Ant Colony Optimization (ACO):

- Strengths: Suitable for problems with a graph-based nature.

- Limitations: Relaxing constraints on position evolution leads to local optima entrapment. ACO's sensitivity to hyperparameters and the direct influence of heuristic information on the search process make it less robust compared to GA and PSO for bin packing. Also, it works better on Graph nature problems like TSP.

5. Tabu Search (TS):

- Strengths: TS excels in time efficiency and solution quality due to its additional memory, which prevents cyclic actions and broadens the search space coverage.

- Performance: TS is highly effective for bin packing, outperforming other single solution algorithms by balancing time efficiency and solution quality.

6. Greedy Randomized Adaptive Search Procedure (GRASP):

- Expected to perform well due to the initial construction of high-quality solutions.

- Limitations: The metaheuristic plots indicate that the time required to achieve a good starting point is not excessive, with no good randomization it is highly prone to being stuck in local optima. Ample exploration of the search space is more advantageous.

7. Simulated Annealing (SA):

- Strengths: Sufficient exploration due to its reheating process.

- Limitations: Prone to cyclic actions, which can increase runtime. However, in the long run, SA may find optimal solutions.

For the bin packing problem, algorithms that emphasize broad exploration and diverse solution handling, such as the Genetic Algorithm and Tabu Search, demonstrate superior performance. While GA benefits from its ability to manage infeasible solutions and adaptive mutations, TS excels through its efficient use of memory to prevent cycles. On the other hand, algorithms like PSO and ACO, despite certain strengths, face challenges with discrete problems and parameter tuning, respectively. In summary, a balanced approach that incorporates robust exploration and diverse strategies tends to yield better long-term results in bin packing problems.

Reference:

1. Xue, Y., Zhu, H., Liang, J., & Słowik, A. (2021). Adaptive crossover operator based multi-objective binary genetic algorithm for feature selection in classification. *Knowledge-Based Systems, 227*, 107218. https://doi.org/10.1016/j.knosys.2021.107218

2. Munien, C., & Ezugwu, A. E. (2021). Metaheuristic algorithms for one-dimensional bin-packing problems: A survey of recent advances and applications. *Journal of Intelligent Systems*, 30, 636-663. https://doi.org/10.1515/jisys-2020-0117

3. Smid, M. (Year). *Computing intersections in a set of line segments: The Bentley-Ottmann algorithm*. Retrieved from https://people.scs.carleton.ca/~michiel/lecturenotes/ALGGEOM/bentley-ottmann.pdf

4. Nguyen, T.-H., & Nguyen, X.-T. (2023). Space Splitting and Merging Technique for Online 3-D Bin Packing. *Mathematics*, *11*(1912), 1-16. https://doi.org/10.3390/math11081912
5. Falkenauer, E. (1994). A new representation and operators for genetic algorithms applied to grouping problems. Evolutionary Computation, 2(2), 123-144. https://doi.org/10.1162/evco.1994.2.2.123

6. Ponce-Pérez, A., Pérez-Garcia, A., & Ayala-Ramirez, V. (n.d.). Bin-packing using genetic algorithms. Universidad de Guanajuato FIMEE, Salamanca, Gto., Mexico.

7. Borna, K., & Khezri, R. (2015). A combination of genetic algorithm and particle swarm optimization method for solving traveling salesman problem. *Cogent Mathematics*, *2*(1), 1048581. doi: 10.1080/23311835.2015.1048581

8. Abadlia, H., Smairi, N., & Ghedira, K. (2017). Particle Swarm Optimization based on Island Models. *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO'17)*, Berlin, Germany. doi:10.1145/3067695.3076068.