Homework 4

Genetic Algorithm

Advanced Algorithm

Supervisor:

Dr. Ziarati

Atefe Rajabi

40230563

Spring 2024

Introduction:

The 2D bin packing problem involves arranging a set of rectangular items into a finite number of bins with fixed dimensions in a way that minimizes the number of bins used or the unused space within each bin. This document describes the implementation of a solution to the 2D bin packing problem using a genetic algorithm, a heuristic search method inspired by natural selection.

Problem Definition:

The objective is to pack a set of rectangular items into the smallest number of bins possible. Each bin has fixed width and height, and each item has specific dimensions (width and height). Items cannot be rotated, and they must fit entirely within the bins without overlapping.

Genetic Algorithm:

A genetic algorithm (GA) is an optimization technique based on the principles of natural selection and genetics. It typically involves the following steps:

1. Initialization: Generate an initial population of solutions randomly.
2. Selection: Evaluate the fitness of each solution and select the fittest individuals for reproduction.
3. Crossover: Combine pairs of selected individuals to produce offspring.
4. Mutation: Introduce random changes to some individuals to maintain genetic diversity.
5. Replacement: Replace some of last generation individuals in the population with new offspring.
6. Termination: Repeat the process until a stopping criterion is met (e.g., a solution of sufficient quality is found or a maximum number of generations is reached).

Implementation Details:

1. Representation of Solutions: Solutions (chromosomes) are represented as permutations of the item list, indicating the order in which items are placed into bins. (Group-based representation[1])
2. Fitness Function: The fitness of a solution is evaluated based on the amount of unused space within the bins, also taking into account a penalty for solutions that are not feasible.
3. Selection Mechanism: The tournament selection method is used to choose individuals for reproduction, and a rate of 0.1 is applied for survival selection through elitism.
4. Crossover and Mutation: Crossover is performed by combining parts of two parent solutions, while mutation is applied by randomly altering an item's position in the bin or in the chromosome sequence.
5. Algorithm Parameters: Parameters such as population size, crossover rate, mutation rate, and number of generations are set based on experimental tuning.

[1] 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

Chromosome Representation:

In the implementation of genetic algorithms for 2D bin packing, the chromosome representation is crucial for effectively encoding solutions. The chosen method of representation is a group-based approach. This choice is based on several strategic benefits that align with the objectives of the genetic algorithm, particularly in terms of crossover and mutation processes.

Reasons for choosing Group-based Representation:

Facilitation of Crossover: The group-based representation is particularly well-suited for performing 2-point group-based crossover. This method involves exchanging segments between two parent chromosomes at two designated points, which can potentially include whole groups of items. Such an approach allows for the efficient combination of traits from different solutions, enhancing the diversity of the gene pool and potentially leading to more optimal packing configurations.

Diverse Mutation Strategies: This representation supports diverse mutation strategies. The mutations can be implemented within a single bin or between multiple bins. Mutation within a bin may involve rearranging the items in that bin to try to find a more compact layout, while mutation between bins might involve swapping items from one bin to another to improve overall packing efficiency.

Simplicity and Effectiveness: Despite its capability to support complex genetic operations, the group-based representation remains straightforward and intuitive. This simplicity is beneficial because it reduces the complexity of decoding the chromosome back into a bin packing solution. It allows the algorithm to more directly impact the packing process, making it easier to implement and understand.

Population Initialization:

Feasible solutions construction:

In the implementation of genetic algorithms for 2D bin packing, a key component is the generation of a feasible initial population. This process is crucial as it establishes the foundational gene pool from which the algorithm begins its optimization journey. The feasible population is generated using a randomized heuristic called feasible_random_packing, which uses an approach that will be explained further in the document.

Other approaches have also been implemented but not utilized, such as 'brute-force' and 'row-pack,' 'First Fit Packing' which is a semi-heuristic approach.

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.

Feasible Random Packing

The ‘feasible_random_packing’ method is designed to distribute a collection of items into bins in a randomized but feasible manner, aiming to pack all items without exceeding bin capacities.

1. Initialization

- Item Randomization: The items are shuffled to ensure randomness in the selection process, which helps in generating diverse initial solutions.

2. Packing Process

- Item Placement: The algorithm enters a loop that continues until all items are placed in bins or marked as leftovers. It processes items in batches.

- Batch Preparation: Depending on the availability of items and their distribution:

  - Items still in the map and leftover items are considered for the current batch.

  - The total area of these items is calculated to estimate how they can be distributed based on bin capacity.

- Bin and Item Distribution Calculation:

  - The required number of bins is estimated using the average area per item and the total bin capacity.

  - A normal distribution is used to randomize the number of bins, ensuring that the process introduces variability and avoids deterministic packing patterns.

  - Items for the current batch are then selected randomly from the available items.

3. Item Fitting

For each batch, a new bin is attempted:

  - Each item in the batch is attempted to be placed in the new bin. If it doesn’t fit, it’s moved to the leftover items.

  - If an item is placed successfully, it is also added to the list of successfully placed items and the bin's unoccupied area is updated.

  - The bin is only retained if at least one item fits in it, ensuring no empty bins are added to the solution.

The process repeats, re-attempting to place leftover items until all items are placed. At the end, the method checks if all items have been successfully packed. If any item remains unpacked, it throws an error indicating that not all items could be packed.

First Fit Packing

The provided function, ‘Last fit packing’ implements a variation of the bin packing problem with a heuristic approach that incorporates randomness and feasibility checks.

- Shuffles the list of items using a random number generator to introduce randomness into the packing order, potentially providing a diverse set of packing arrangements across different executions.

- Iterates through each item in the shuffled list.

- Attempts to place the item in the most recently created bin (`bins.back()`).

- If placement fails, the algorithm iterates through all existing bins (except the most recent one if it was just checked) to try and place the item in any bin that has sufficient unoccupied area.

- If the item does not fit in any existing bin, a new bin is created, and the item is placed in this new bin.

- If an item is successfully placed, the item is added to the bin, the list of placed items is updated, and the unoccupied area of the bin is adjusted accordingly.

- Each bin tracks its own feasibility status and available space to ensure that no bin is overfilled.

- At the end of the loop, if the number of successfully placed items does not match the total number of items provided, an error is thrown, indicating that not all items could be packed.

Heuristic Approach and Effectiveness

- First Fit Approach: The function essentially starts with a first-fit strategy but includes retries in previously unsuccessful bins. This could reduce the number of bins used compared to a strict first-fit approach that always moves to a new bin if the current one fails.

- Randomness: By shuffling the items before packing, the function tests different sequences in placing items, which can sometimes lead to more efficient packing due to the varied order of item sizes and shapes being packed. Randomness helps in exploring different packing configurations which might not be considered in a deterministic approach.

- Revisiting Bins: Before creating a new bin, the algorithm revisits all existing bins to check if the current item can fit. This ensures that any newly available space created by the configuration of previously packed items is effectively utilized.

Use Case and Optimization

- Large Number of Non-uniform Items: For instances with a large variety of item sizes and shapes, starting with a low number of bins and allowing for dynamic exploration of packing arrangements can be beneficial. The randomness and flexibility in bin selection help in efficiently utilizing bin space and potentially reducing the total number of bins required.

- Exploration and Exploitation: The approach balances between exploring new packing configurations (through randomness and checking multiple bins) and exploiting known configurations (by revisiting and filling up bins that are not yet full).

Infeasible Population Initialization

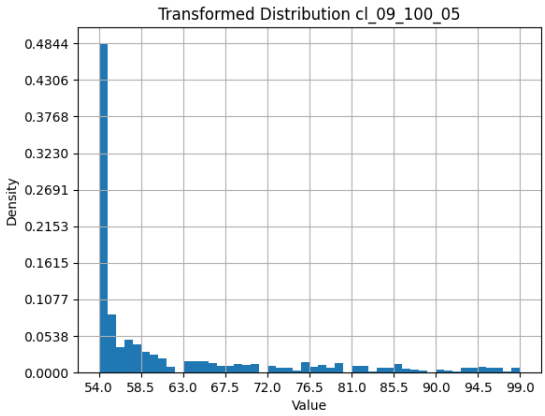
The generation of an infeasible initial population is another crucial aspect of population initialization in the genetic algorithm for 2D bin packing. This process intentionally creates chromosomes that are likely to violate some of the problem's constraints. The purpose of including infeasible solutions is to broaden the algorithm's exploration of the solution space, potentially leading to innovative solutions that might not be discovered through feasible-only populations.

1. Sampling the Number of Bins:

- The number of bins for each chromosome is determined by sampling from a skewed distribution. This is implemented using a power transformation (‘pow(real_dis(gen), 10)’) on a uniform distribution, which skews the number of bins towards a larger range beyond typical feasible solutions.

- This skewed sampling approach allows the algorithm to explore configurations with different numbers of bins than might be typically considered, including both over-packed and under-packed scenarios. This can lead to a greater understanding of the packing dynamics and potentially innovative packing strategies.

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.

2. Distributing Items Among Bins:

- After determining the number of bins, items are distributed among them. Initially, each bin is assigned one item to ensure all bins are utilized.

- Remaining items are distributed using a process that simulates natural variability and imbalance among the bins. This is achieved through a normal distribution based on the average remaining items per bin. The standard deviation is set to a third of the mean to introduce enough variability without extreme anomalies.

- This distribution method allows for some bins to be more densely packed than others, deliberately creating configurations that are likely to be infeasible under normal constraints. This step is crucial for testing the boundaries of feasible packing and provides a richer landscape for the genetic algorithm to explore.

3. Placing Items in Bins:

- For each bin, items are placed at random positions within the bin dimensions, taking care to ensure that the item does not exceed the bin's boundaries. This is achieved using uniform distributions to randomly select positions within the valid range.

- The random placement can lead to overlaps and inefficient use of space, further contributing to the infeasibility of the chromosome. This method challenges the subsequent stages of the algorithm to rearrange and optimize these initial placements through evolutionary operations like crossover and mutation.

4. Evaluating Infeasibility and Fitness:

- Unlike the feasible initialization, where only feasible chromosomes are retained, all generated configurations during the infeasible initialization are kept, regardless of their feasibility.

- Each chromosome undergoes a fitness calculation which measures the degree of infeasibility (e.g., how much items overlap and under-occupied). This metric can be useful for guiding evolutionary operations aimed at 'repairing' these infeasible solutions into feasible, optimized configurations.

Feasibility Check:

The goal is to ensure that no rectangles (items) within a bin overlap. The method uses a sweep-line algorithm combined with an interval tree to efficiently detect overlaps. (the Bentley-Ottmann algorithm [2])

Sweep line algorithm

The sweep-line algorithm is a computational geometry technique used to detect intersections in a set of intervals. It involves moving a vertical line (sweep line) from left to right across the plane, processing events where intervals (rectangles) start and end.

It works as follows:

Events Creation: For each rectangle in a bin, create two events: one for the left endpoint and one for the right endpoint.

Events Sorting: Sort all events based on their x-coordinate. If two events have the same x-coordinate, prioritize the right endpoint over the left endpoint to avoid considering touching intervals as overlapping.

Processing Events: Sweep a vertical line from left to right across the plane. Use an interval tree to keep track of active intervals (rectangles that intersect the sweep line). For each event: If it is a left endpoint, check if it overlaps with any active interval in the interval tree. If there is an overlap, report that the configuration is not feasible. Otherwise, insert the interval into the interval tree. If it is a right endpoint, remove the corresponding interval from the interval tree

Interval tree

An interval tree is a data structure that allows efficient querying of all intervals that overlap with a given interval. It supports operations to insert, delete, and search intervals, making it suitable for dynamically maintaining the set of active intervals intersected by the sweep line.

Insert: Insert a new interval into the tree, maintaining the tree's structure and updating the maximum endpoint value of each node.

Remove: Remove an interval from the tree, maintaining the tree's structure and updating the maximum endpoint value of each node.

Overlap Search: Check if a given interval overlaps with any interval currently in the tree.

[2] 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

Time Complexity:

The overall time complexity of the algorithm is determined by the sweep-line algorithm combined with interval tree operations:

Sort Events: O(nlogn), where n is the number of events (twice the number of rectangles).

Processing Events: For each event, we either insert or delete an interval and check for overlaps. Each of these operations on the interval tree takes O(logn) time.

where n is the number of rectangles and m is the number of bins:

Brute Force Approach: $O(mn^2)$

Sweep-Line + Interval Tree Approach: $O(m \cdot n\log n)$

Fitness Function:

These functions include fitness calculation, objective function calculation, and penalty calculation.[3]

‘calculate_fitness(penalty_factor)’

The fitness is influenced by both the objective value and a penalty for any constraints that are not satisfied.

- ‘penalty_factor’: (starting from 1.0) This is used to scale the penalty, allowing the algorithm to adjust the severity of the penalty over generations (e.g., increasing the penalty as the generations progress to discourage infeasible solutions).

Calculation

1. Objective Calculation: The ‘objective_function()’ is called to compute the objective value, which measures the efficiency of the bin packing.

2. Penalty Calculation: The ‘calculate_penalty()’ is computed to quantify how much the solution deviates from feasibility (e.g., bins overlapping or not fully utilized).

3. Fitness Calculation: Combines the objective and penalty:

- ‘alpha’ and ‘beta’ are constants set to weight the objective and the penalty, respectively.

- The formula for fitness is: ‘fitness_ = alpha * objective + beta * penalty * penalty_factor’.

- The penalty is scaled by the ‘penalty_factor’ and ‘beta’, emphasizing the penalty's effect on the overall fitness score.

‘objective_function’

Calculates the objective value based on the utilization efficiency of bins that are feasible.

1. Utilization Computation:

- Iterates over each bin in ‘groups_’.

- Skips bins that are not feasible.

- For each feasible bin, calculates the utilization ratio, which is the proportion of the bin area that is actually used.

- The utilization ratio is then raised to a power ‘z’ which is equal to 2.

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

2. Average Utilization:

- If there are feasible bins, the average utilization across all bins (not just the feasible ones) is computed, penalizing solutions where not all bins are used efficiently.

‘calculate_penalty’

Calculates the penalty for a chromosome based on the infeasibility of the bins.

1. Penalty Accumulation:

- Iterates over each bin.

- For bins that are not feasible, calls ‘calculate_bin_penalty()’ to compute the penalty for that specific bin based on how it violates the packing constraints.

‘calculate_bin_penalty’

Calculates the penalty for a single bin based on its packing inefficiencies.

1. Overlap Penalty:

- Computes the area of items that should be within the bin but are overlapping with the bin's boundaries.

- The penalty is higher if there is more overlap relative to the bin's capacity.

Parent Selection:

1. Selection of Candidates:

   The select_candidates() function randomly picks a set number of individuals (chromosomes) from the population. In this specific implementation, five unique individuals are selected. This randomness is achieved through the use of a uniform integer distribution that generates indices pointing to the population (or mating_pool_).

2. Evaluation and Ranking:

   Once the candidates are selected, they are evaluated based on a fitness criterion. This is apparent from the sorting mechanism in the parent_selection() function where candidates are sorted by their fitness values. The fitness function (assumed to be get_fitness()) quantitatively evaluates how good a solution represented by the chromosome is.

3. Selection of the Best:

   After sorting, the top individuals (the ones with the highest fitness) are chosen as parents. This ensures that the traits of the fittest individuals are more likely to be passed on to the next generation, promoting the improvement of the population over time.

   Why Tournament selection:

1. Efficiency:

   Tournament selection is computationally efficient and easy to implement. It does not require the fitness values of the entire population to be known before selection starts; only those of the selected candidates need to be compared.

2. Control over Selection Pressure:

   The size of the tournament (the number of candidates selected in each round) can be adjusted to control the selection pressure. A larger tournament size increases selection pressure as more individuals compete to be selected, generally leading to a quicker convergence towards better solutions.

3. Maintains Diversity:

   By randomly selecting candidates for each tournament, this method helps maintain genetic diversity within the population. This is crucial for avoiding premature convergence to suboptimal solutions and helps the algorithm explore a wider solution space.

Survival Selection:

Elitism

Elitism is a technique used in genetic algorithms to ensure that the fittest individuals of the current generation are automatically carried over to the next generation. This helps in retaining the best solutions found so far without risking their elimination due to selection, crossover, or mutation processes. When using elitism with a 0.1 ratio, it means that the top 10% of the population by fitness are guaranteed to survive to the next generation.

Functionality of Elitism with a 0.1 Ratio

1. Assessing Fitness:

- Each individual in the current population is evaluated based on a predefined fitness function that quantifies how well it solves the problem at hand.

2. Ranking Individuals:

- The entire population is ranked based on fitness scores. This ranking is crucial as it determines which individuals qualify as the 'elite'.

3. Selecting the Elite:

- The top 10% of individuals from the ranked list are identified. The exact number of elite individuals retained in the next generation depends on the size of the population. For example, in a population of 100, the top 10 individuals would be carried over.

4. Forming the Next Generation:

- These elite individuals are directly copied to the next generation. The rest of the next generation is filled up through other genetic operations such as crossover and mutation applied to the selected individuals from the current generation, possibly including some of the elites as well.

Why Elitism?

1. Preservation of Good Solutions:

- Elitism directly ensures that the genetic algorithm does not lose the best-found solutions due to stochastic errors in other genetic operations. This is particularly beneficial in landscapes where good solutions are rare.

2. Faster Convergence:

- By preserving the best individuals, elitism can accelerate the convergence of the genetic algorithm towards an optimal solution since subsequent generations are built on the proven good solutions.

3. Avoids Regression:

- Without elitism, there's a risk that the population's average fitness could increase (since we have minimizing problem) from one generation to the next if superior individuals are not selected for reproduction. Elitism prevents this by ensuring that top performers continue to contribute their genes to the pool.

4. Stability in Evolutionary Process:

- Elitism adds a level of stability to the evolutionary process by mitigating the effects of genetic drift (random changes in allele frequencies that occur in small populations) and maintaining a steady improvement in fitness levels.

5. Enhanced Performance in Multimodal Functions:

- In problems where multiple good solutions exist (multimodal functions), elitism helps in maintaining diversity among the top-tier solutions, allowing the algorithm to explore multiple peaks in the fitness landscape simultaneously.

Implementing elitism, especially at a relatively modest ratio like 0.1, strikes a balance between preserving excellent solutions and maintaining enough genetic diversity to explore new and potentially better solutions.

Crossover:

This genetic algorithm approach uses a group-based two-point crossover, which is a crucial operation in the evolution cycle that mixes the genetic information of two parent chromosomes to generate new offspring.

Implementation Details

Struct ‘Points’

Defines the boundaries of the crossover within the parent chromosomes. It includes the starting and ending indices for items and groups (bins) involved in the operation.

Function ‘get_points’

Randomly selects continuous segments (groups) of a chromosome for the crossover operation. It ensures that the segment size does not exceed the number of groups in the chromosome. Also, number of consecutive groups cannot be equal to the minimum of number of groups. Because, in that case, the crossover is useless, just swapping 2 parents.

Function ‘swap_groups’

Exchanges the selected groups between two parent chromosomes. This function is crucial for introducing genetic diversity and helps explore new regions of the solution space.

Function ‘update_frontiers’

Updates the "frontiers" of a chromosome, which are the indices marking the boundaries of the bins after the crossover.

Function ‘repair_offspring’

Ensures that the offspring chromosomes maintain valid and consistent bin configurations. It deals with issues such as duplicated or missed items or bins that may occur after crossover. This function is essential for maintaining the integrity of chromosome representations and enhancing the algorithm's performance by ensuring only feasible (only in terms of permutation of items) solutions are considered.

Crossover Process

Selection of Segments: Randomly determines the size of the segment (number of consecutive groups) and selects corresponding segments from both parent chromosomes.

Swap of Segments: Exchanges the selected segments between the two parent chromosomes to create two new offspring.

Repair of Offspring: Adjusts the offspring chromosomes to correct any inconsistencies or rule violations that may have arisen during the crossover.

Update of Frontiers: Recalculates the boundaries of each bin in the offspring chromosomes.

Why group-based two-point crossover will be helpful in a 2D bin packing problem?

Enhanced Search Diversity

Group-based crossover manipulates sets of items (or groups) rather than individual items. This method allows for larger and more impactful changes in each generation, which can help the algorithm escape local optima. By swapping whole groups between chromosomes, the offspring can explore new configurations that may not be reachable through single-item manipulations or simpler crossover operations.

Preservation of High-Quality Building Blocks

In genetic algorithms, the concept of building blocks refers to sets of genes (in this case, groups of items in bins) that work well together to contribute to good solutions. Group-based crossover has the potential to preserve these building blocks better than individual-based crossover approaches. Since whole groups are transferred between solutions, the structural integrity of these effective combinations is maintained, which can lead to a higher likelihood of producing fit offspring.

Reduced Fragmentation

In 2D bin packing, how items are grouped and arranged can significantly affect the packing density and the overall number of bins used. Group-based crossover reduces the chance of breaking effective item arrangements that have a high packing efficiency. By moving groups of items that are already packed together, the crossover respects the spatial relationships and packing patterns that have proven effective, potentially reducing fragmentation and the need for reorganization.

Effective Exploration of Solution Space

Group-based crossover allows the algorithm to sample a broader range of the solution space by combining features from two different parents in a more substantial manner. This capability is

crucial in complex problems like 2D bin packing, where the arrangement of items can dramatically influence the solution quality. Exploring a wide range of configurations increases the likelihood of finding more optimal solutions.

Compatibility with Problem Constraints

2D bin packing often involves various constraints, such as item orientation, bin size limits, and the non-overlapping requirement. Group-based crossovers are particularly adept at handling such constraints because they inherently ensure that groups of items that fit together (according to the constraints) are moved as a unit, thus preserving their feasibility in new contexts without violating constraints.

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:

This step is crucial for maintaining a high-quality pool of solutions in the population. The code features three main functions: ‘calculate_repair_probability’, ‘repair_chromosome’, and two specific repair strategies (‘repair_chromosome_semi_heuristic’ and ‘repair_chromosome_random’).

1. ‘calculate_repair_probability’

This function calculates the probability of attempting to repair an infeasible chromosome. The probability calculation is influenced by the condition specified by ‘stop_type’, which is based on generation count, time limit, or lack of improvement (convergence):

- Generation-based Probability: The probability increases as the current generation number approaches the maximum generation limit. This approach uses an exponential function to ensure that as more generations pass, the urgency or likelihood of repair increases and so on for others stopping criteria.

2. ‘repair_chromosome’

This function determines whether to attempt repairing a given chromosome based on its feasibility and a random chance compared to the calculated repair probability. If a chromosome is infeasible and the determined random chance falls below the calculated probability, the function calls the repair strategy (‘repair_chromosome_random’) to attempt making the chromosome feasible.

3. ‘repair_chromosome_semi_heuristic’

This strategy for repairing a chromosome uses a more structured approach:

- Item Reorganization: Each bin in the chromosome is unpacked, and items are reorganized. Items from bins and leftover items are sorted primarily by height and secondarily by width to optimize packing.

- Repacking: Items are repacked into new bins. The function attempts to place each item in a new bin and moves unplaceable items to a list of leftovers, which will be retried in subsequent bins.

- Iteration: The process continues with leftover items until all items are either packed or confirmed unplaceable. If at the end, some items remain unpacked, it raises an error indicating incomplete packing.

4. ‘repair_chromosome_random’

This function implements a random approach to repair:

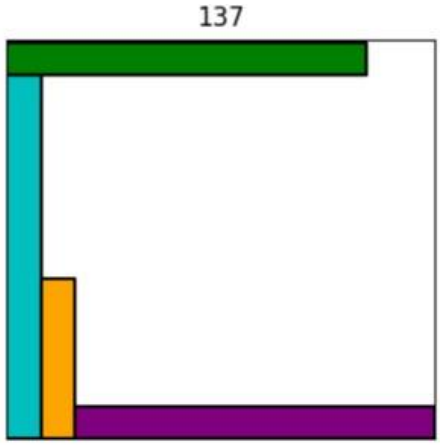
- Bin Iteration and Item Shuffling: It iterates through each bin, shuffles the items and leftovers, and attempts to repack them into new bins using a random placement method (‘place_item_feasible_randomly’).

- Random Placement and Feasibility Check: Each item is placed randomly within the bins. If an item fits, it is added; otherwise, it remains a leftover to be attempted again.

- Error Handling: Similar to the semi-heuristic approach, if items remain unpacked after all attempts, an error is raised.

Both repair strategies ensure that the chromosome maintains or reaches a feasible state, enhancing the genetic algorithm's ability to converge on effective solutions. However, both methods have drawbacks: the main concern for random repair is its time complexity, but for the semi-heuristic one, the problems are more serious, although it is time-efficient.

Since we just want to repair the chromosome but not change the bins configuration, we perform row-packing heuristic bin-based and this causes that problem of not letting the algorithm find optimized pack in a long run.



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.

Detecting Unoccupied Areas

1. Create Events for Each Rectangle:

Each rectangle generates two events: an "enter" event when the left edge is encountered by the sweep line, and an "exit" event when the sweep line encounters the right edge. Each event captures the x-coordinate of the edge, the event type, and the y-interval defined by the top and bottom edges of the rectangle.

2. Sort Events:

Events are sorted by the x-coordinate to facilitate a left-to-right sweep across the container.

3. Sweep Line Processing:

As events are processed:

- An "enter" event adds the rectangle's y-interval to a data structure (such as a 'Interval tree) that keeps track of which parts of the y-axis are currently occupied at the sweep line's position.

- An "exit" event removes the y-interval from the data structure, indicating that the rectangle no longer affects that section of the y-axis beyond the current x-coordinate.

4. Calculate Unoccupied Y-Intervals:

At each unique x position (before processing the next event), unoccupied y-intervals are calculated by iterating through the sorted y-values in the active intervals and identifying gaps where the cumulative count of overlapping rectangles drops to zero.

5. Store Unoccupied Spaces:

Each identified gap in the y-interval is recorded as an unoccupied space that extends from the last x-coordinate to the current one, thereby defining rectangular unoccupied spaces.

Rectangle Placement:

1. Merge Unoccupied Spaces:

Vertically aligned unoccupied spaces that are adjacent horizontally are merged to simplify the process of finding a fitting space for new rectangles.

2. Find a Fitting Space:

Each merged unoccupied space is evaluated to determine if it can accommodate a rectangle of specified width and height:

- The width of the space (difference between the x-values) is compared to the width of the rectangle.

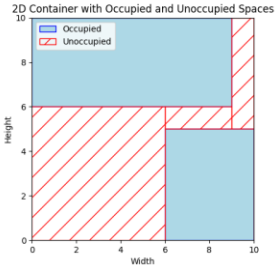
- The height of the space (difference between the y-values) is compared to the height of the rectangle.

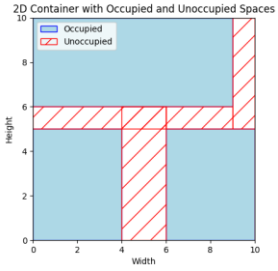
3. Place the Rectangle:

If a suitable space is found, the rectangle is positioned in the earliest fitting location:

- The position of the rectangle is updated accordingly.

- The unoccupied space is adjusted to reflect the placement of the new rectangle by potentially dividing the occupied space into smaller unoccupied areas.





The search process occurs through unoccupied spaces. These unoccupied spaces are then merged, helping to prevent the space from being divided into smaller pieces.

This methodology utilizes event sorting, sweep line processing, and dynamic interval management to efficiently manage and utilize space in a 2D container. It focuses on significant changes at rectangle edges and dynamically manages y-intervals to avoid the inefficiencies associated with brute force approaches, allowing for effective handling of larger and more complex scenarios.

Time Complexity:

The time complexity for detecting unoccupied spaces and placing a rectangle in the container is $O(n\log n+nk)$, where n is the number of rectangles, and k is the number of active intervals. This

complexity arises primarily from the event sorting and sweep line processing and O(mlogm) for placing a new rectangle, where m is typically much less than W×H because it represents merged spaces rather than all potential positions.

In contrast, a brute force approach for checking space availability and placing rectangles would have a time complexity of O(W×H×n), where W and H are the dimensions of the container. This approach is impractical for large containers or a large number of rectangles due to its computational inefficiency.

Mutation

‘feasible_insert’

This function attempts to insert an item into a bin in the unoccupied areas identified. It first calculates the unoccupied areas within the bin and then tries to place the item optimally within these spaces using the ‘place_rectangle’ function. If a valid placement is found (not having overlap with other items), the item is added to the bin, and its location is updated accordingly. The feasibility of the bin is checked before and after the insertion to ensure the bin's configuration remains valid. If the placement leads to an infeasible configuration, details are outputted for debugging.

‘randomly_insert’

This mutation function places an item randomly within the confines of the bin. It generates random x and y coordinates where the top-left corner of the item can be placed without exceeding the bin's boundaries. This method does not consider the overlap with other items or the bin's feasibility, making it a straightforward but potentially disruptive mutation strategy.

‘item_exceed_removal’

This function is designed to correct any items in a bin that exceed the bin's boundaries. It iterates over all items in a randomly selected bin and adjusts the positions of items that exceed the top or right boundaries of the bin. This is a corrective mutation aimed at maintaining the feasibility of the solution by ensuring all items fit within their designated bins.

‘item_overlap_removal’

This mutation function targets bins with overlapping items. For each bin that contains overlaps and is deemed infeasible, it randomly selects a pair of overlapping items and attempts to resolve the overlap by adjusting their positions. The method checks the direction of the overlap (horizontal or vertical) and moves one item to eliminate the overlap while trying to keep both items within the bin's boundaries.

‘random_reinsert_mutation’

This function selects an item from one bin and reinserts it into another bin, chosen at random. The item is removed from its original bin and added randomly into the new bin, potentially disrupting the layout of both bins. This mutation can help explore new bin configurations but may reduce the feasibility of the solution, especially if the item does not fit well in the new bin.

‘feasible_reinsert_mutation

Similar to the ‘random_reinsert_mutation’, this function also moves an item from one bin to another. However, it only performs the reinsertion if the target bin can feasibly accommodate the item without violating the packing constraints. This method attempts to improve the packing efficiency by exploring alternative placements that maintain or enhance feasibility.

‘random_between_swap_mutatio’

‘feasible_between_swap_mutation’

Both these functions swap items between two different bins. The ‘random_between_swap_mutation’ does this without regard to the feasibility of the new arrangement, while the ‘feasible_between_swap_mutation’ ensures that each item can fit into its new bin before performing the swap. These mutations are designed to explore different distributions of items across bins, potentially finding more efficient or feasible configurations.

‘Empty bin’:

This function randomly selects one bin from a chromosome, which represents a possible solution containing multiple bins. It then attempts to empty this selected bin by redistributing its items into other available bins in the chromosome. Items are moved only if they fit without violating packing constraints, as assessed by a feasibility check. The process involves attempting to insert each item from the selected bin into any other bin, using a shuffled order for the bins to ensure randomness. If all items from the selected bin are successfully moved to other bins, the now-empty bin is removed from the solution. This mutation operation can potentially lower the total number of bins used, thus improving the overall efficiency of the packing solution.

Each of these mutation functions introduces variations in the configuration of bins and items in a genetic algorithm, aiming to explore the solution space more thoroughly and escape local optima by adjusting item placements and bin assignments. The feasibility-focused functions add a layer of constraint satisfaction, enhancing the quality of solutions in terms of meeting the problem's requirements.

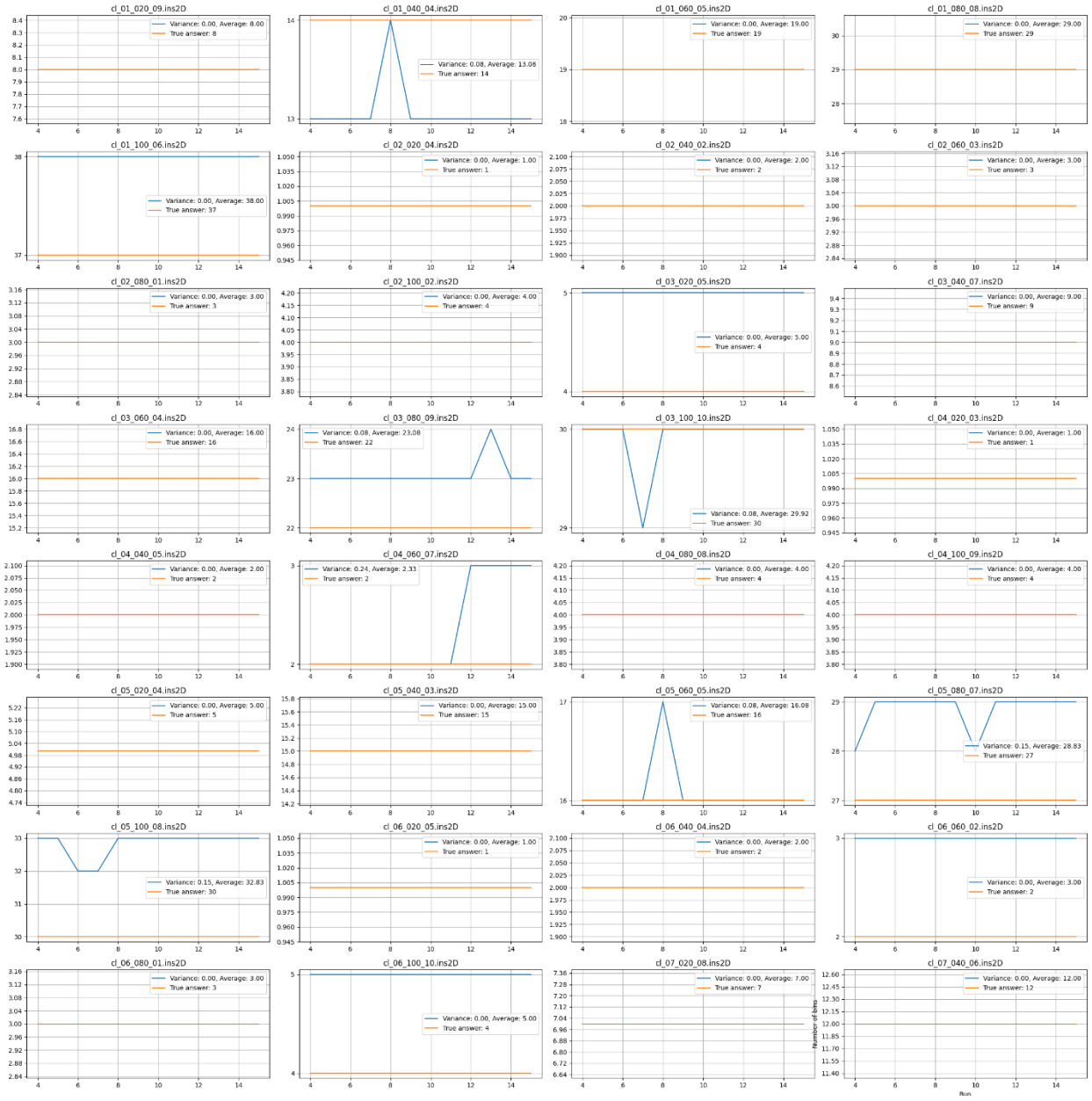
Local Search:

Hill climbing incorporated with feasible reinsertion, feasible swap tweak and empty bin tweak functions serves as an effective strategy in a memetic algorithm, particularly for optimizing solutions in bin packing problems. Since It is fast, simple and easy to implement. In this approach, the algorithm iteratively applies these mutations to enhance the quality of the population with a probability of 0.25 for selecting the chromosome for local search. The feasible reinsertion mutation method selectively moves items between bins, ensuring that each item fits properly within its new location without violating any constraints, thereby maintaining or improving the overall solution feasibility. Concurrently, the feasible swap mutation exchanges items between bins under the condition that the swap results in a valid configuration, enhancing the solution space exploration. By incorporating these strategies before the survival selection, the algorithm not only refines individual solutions but also ensures that the population as a whole is more adapted, significantly improving the quality of solutions retained for future generations. This method effectively combines local search techniques with evolutionary processes, optimizing both the structure and feasibility of solutions in a continuous improvement cycle. Because hill climbing is prone to getting stuck in local optima, a low number of iterations are set up to just slightly improve the solution.

Results:

instance_name	Min	Max
cl_01_020_09.ins2D	8	8
cl_01_040_04.ins2D	13	14
cl_01_060_05.ins2D	19	19
cl_01_080_08.ins2D	29	29
cl_01_100_06.ins2D	38	38
cl_02_020_04.ins2D	1	1
cl_02_040_02.ins2D	2	2
cl_02_060_03.ins2D	3	3
cl_02_080_01.ins2D	3	3
cl_02_100_02.ins2D	4	4
cl_03_020_05.ins2D	5	5
cl_03_040_07.ins2D	9	9
cl_03_060_04.ins2D	16	16
cl_03_080_09.ins2D	23	24
cl_03_100_10.ins2D	29	30
cl_04_020_03.ins2D	1	1
cl_04_040_05.ins2D	2	2
cl_04_060_07.ins2D	2	3
cl_04_080_08.ins2D	4	4
cl_04_100_09.ins2D	4	4
cl_05_020_04.ins2D	5	5
cl_05_040_03.ins2D	15	15
cl_05_060_05.ins2D	16	17
cl_05_080_07.ins2D	28	29
cl_05_100_08.ins2D	32	33
cl_06_020_05.ins2D	1	1
cl_06_040_04.ins2D	2	2
cl_06_060_02.ins2D	3	3
cl_06_080_01.ins2D	3	3
cl_06_100_10.ins2D	5	5
cl_07_020_08.ins2D	7	7
cl_07_040_06.ins2D	12	12
cl_07_060_05.ins2D	15	16
cl_07_080_08.ins2D	23	23
cl_07_100_10.ins2D	33	34
cl_08_020_05.ins2D	6	6
cl_08_040_03.ins2D	11	11
cl_08_060_05.ins2D	14	15
cl_08_080_07.ins2D	22	22
cl_08_100_03.ins2D	24	24
cl_09_020_04.ins2D	16	16
cl_09_040_10.ins2D	34	34
cl_09_060_07.ins2D	41	41
cl_09_080_04.ins2D	53	54
cl_09_100_05.ins2D	65	65
cl_10_020_08.ins2D	3	3
cl_10_040_03.ins2D	10	10
cl_10_060_05.ins2D	8	9
cl_10_080_02.ins2D	12	12
cl_10_100_04.ins2D	20	20

Variance:

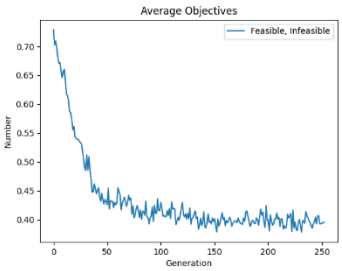

cl_01_020_09.ins2D
Variance: 0.00, Average: 8.00
True answer: 8
cl_01_040_04.ins2D
Variance: 0.08, Average: 13.08
True answer: 14
cl_01_060_05.ins2D
Variance: 0.00, Average: 19.00
True answer: 19
cl_01_080_08.ins2D
Variance: 0.00, Average: 29.00
True answer: 29
cl_01_100_06.ins2D
Variance: 0.00, Average: 38.00
True answer: 37
cl_02_020_04.ins2D
Variance: 0.00, Average: 1.00
True answer: 1
cl_02_040_02.ins2D
Variance: 0.00, Average: 2.00
True answer: 2
cl_02_060_03.ins2D
Variance: 0.00, Average: 3.00
True answer: 3
cl_02_080_01.ins2D
Variance: 0.00, Average: 3.00
True answer: 3
cl_02_100_02.ins2D
Variance: 0.00, Average: 4.00
True answer: 4
cl_03_020_05.ins2D
Variance: 0.00, Average: 5.00
True answer: 4
cl_03_040_07.ins2D
Variance: 0.00, Average: 9.00
True answer: 9
cl_03_060_04.ins2D
Variance: 0.00, Average: 16.00
True answer: 16
cl_03_080_09.ins2D
Variance: 0.08, Average: 23.08
True answer: 22
cl_03_100_10.ins2D
Variance: 0.08, Average: 29.92
True answer: 30
cl_04_020_03.ins2D
Variance: 0.00, Average: 1.00
True answer: 1
cl_04_040_05.ins2D
Variance: 0.00, Average: 2.00
True answer: 2
cl_04_060_07.ins2D
Variance: 0.24, Average: 2.33
True answer: 2
cl_04_080_08.ins2D
Variance: 0.00, Average: 4.00
True answer: 4
cl_04_100_09.ins2D
Variance: 0.00, Average: 4.00
True answer: 4
cl_05_020_04.ins2D
Variance: 0.00, Average: 5.00
True answer: 5
cl_05_040_03.ins2D
Variance: 0.00, Average: 15.00
True answer: 15
cl_05_060_05.ins2D
Variance: 0.08, Average: 16.08
True answer: 16
cl_05_080_07.ins2D
Variance: 0.15, Average: 28.83
True answer: 27
cl_05_100_08.ins2D
Variance: 0.15, Average: 32.83
True answer: 30
cl_06_020_05.ins2D
Variance: 0.00, Average: 1.00
True answer: 1
cl_06_040_04.ins2D
Variance: 0.00, Average: 2.00
True answer: 2
cl_06_060_02.ins2D
Variance: 0.00, Average: 3.00
True answer: 2
cl_06_080_01.ins2D
Variance: 0.00, Average: 3.00
True answer: 3
cl_06_100_10.ins2D
Variance: 0.00, Average: 5.00
True answer: 4
cl_07_020_08.ins2D
Variance: 0.00, Average: 7.00
True answer: 7
cl_07_040_06.ins2D
Variance: 0.00, Average: 12.00
True answer: 12
Number of bins
Run

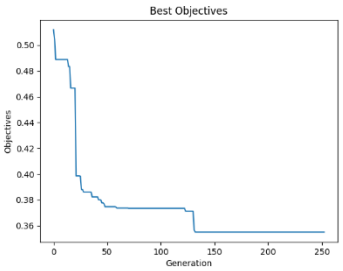
Plots:

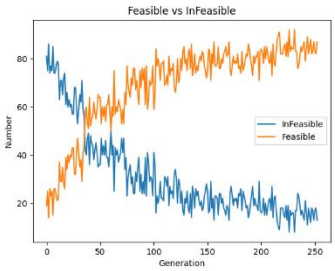
cl_1_020_09:

This instability in the average objective, despite an overall decrease, is due to an increase in the penalty function for infeasible solutions. Infeasible solutions are allowed in the population to encourage more exploration, but they are penalized to prevent the elimination of less optimal chromosomes.

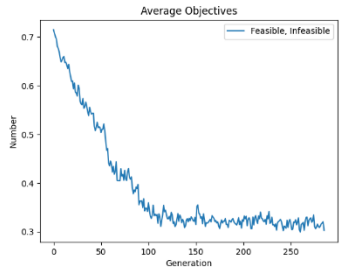
Due to the use of a repair mechanism with a dynamic probability, we can observe that the number of infeasible solutions decreases with passing generations. (third plot)

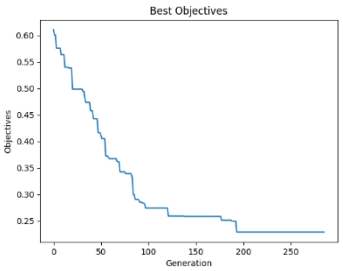


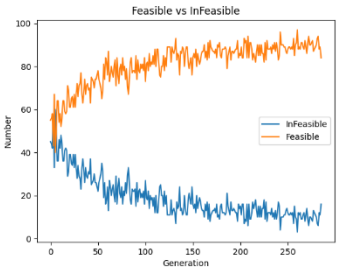




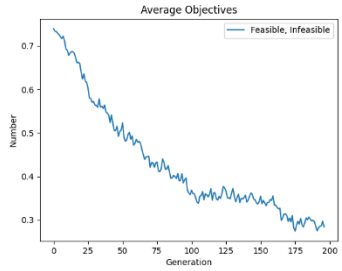
cl_01_040_04:

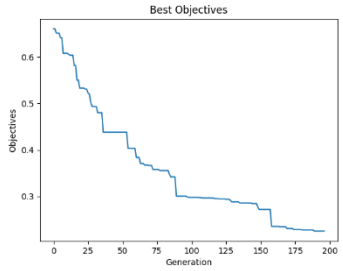


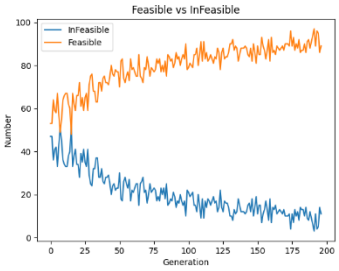




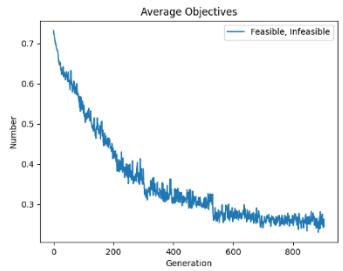
cl_01_060_05:

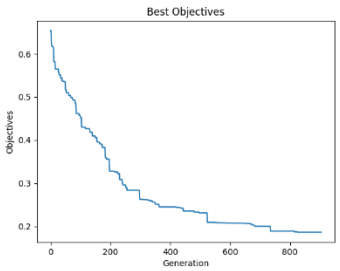


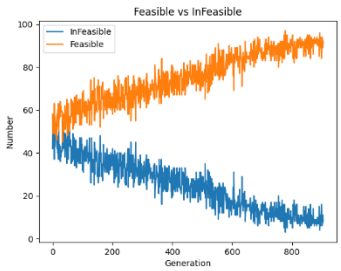




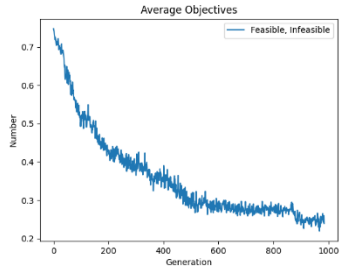
cl_01_080_08:

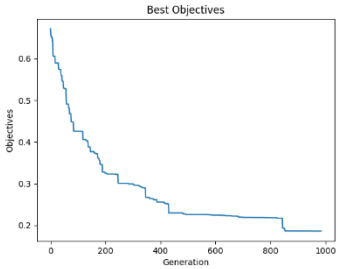


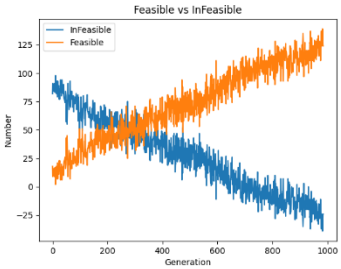




cl_01_100_06:

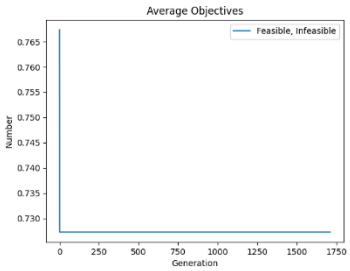


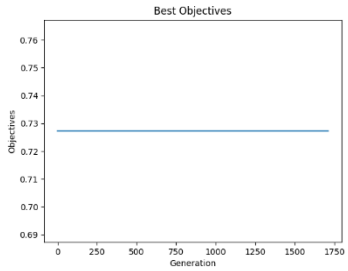


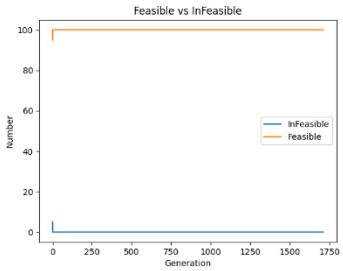


cl_02_020_04: (Easy instance set, all the items can be packed in a single bin)

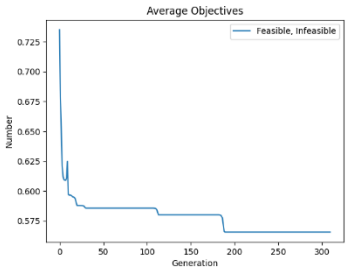
1750 generations for 10 second

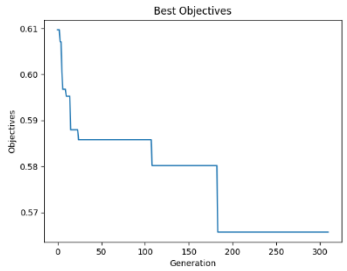


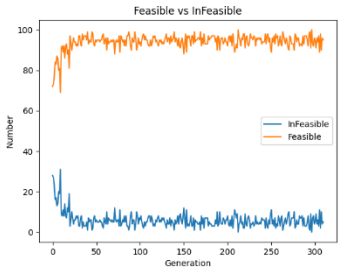




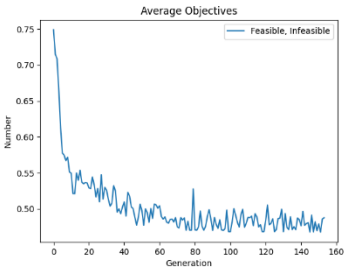
cl_02_040_02:

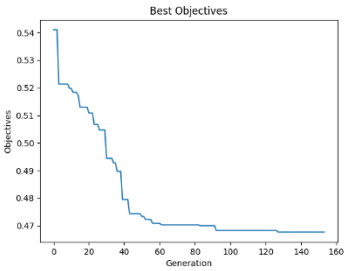


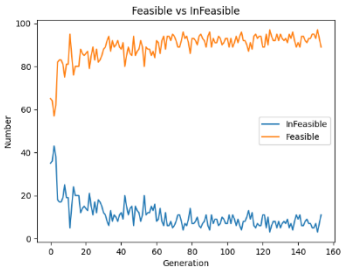




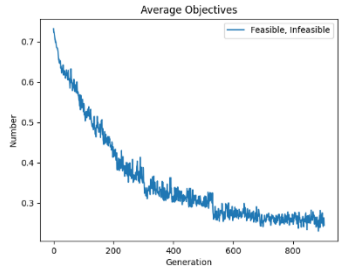
cl_02_060_03:

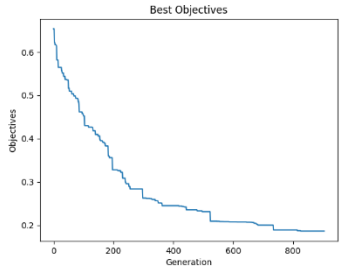


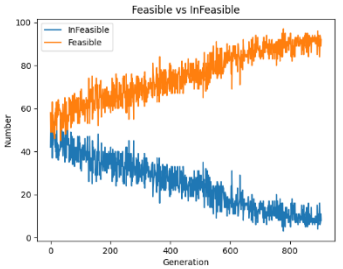




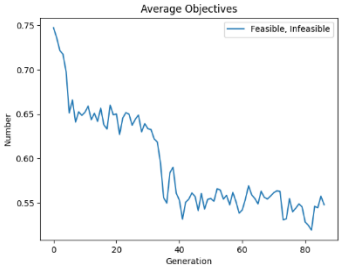
cl_02_080_01:

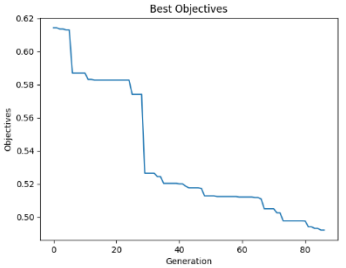


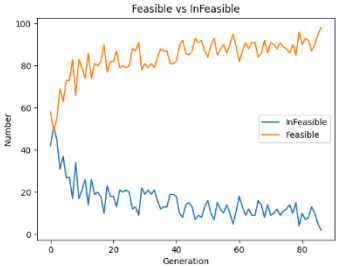




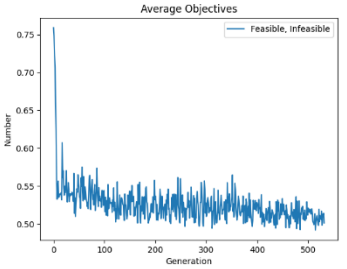
cl_02_100_02:

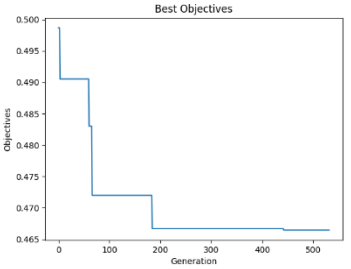


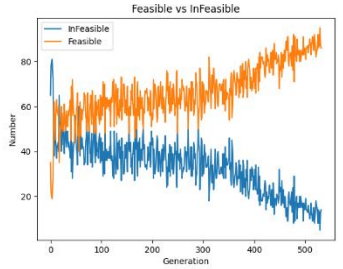




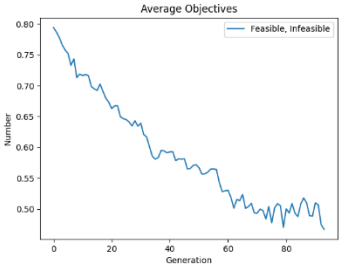
cl_03_020_05:

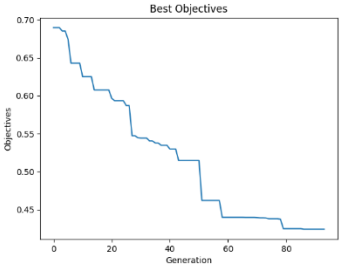


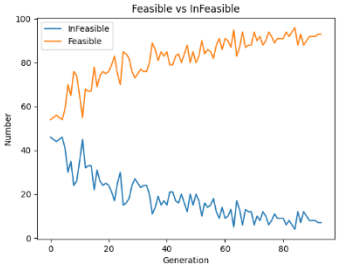




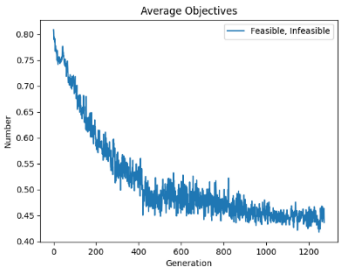
cl_03_040_07:

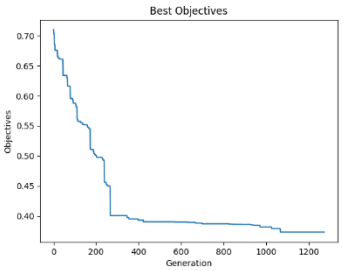


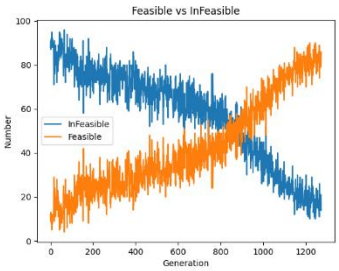




cl_03_060_04:

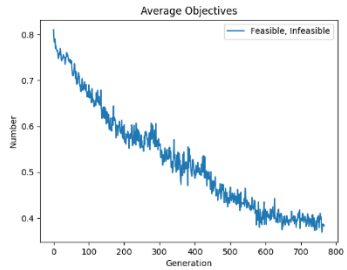


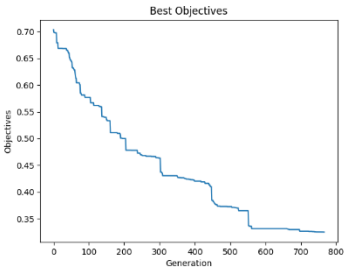


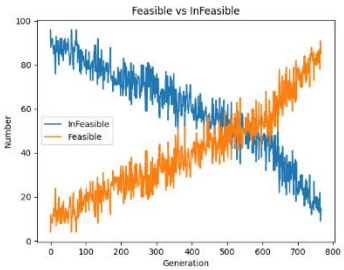


cl_03_080_09:

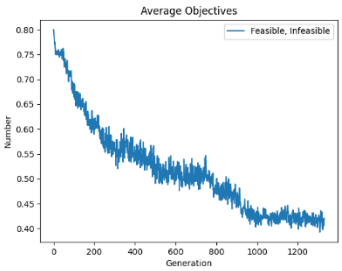
From the Average Objective plot, it is obvious that there is room for improvement by letting the algorithm continue to work.

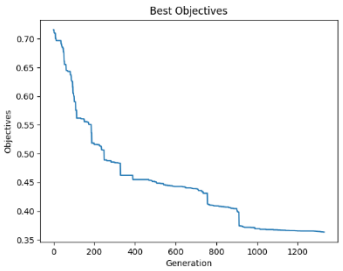


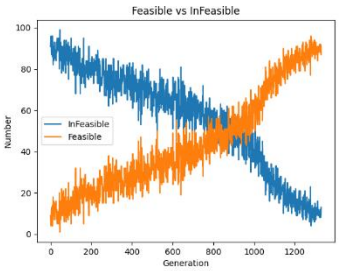




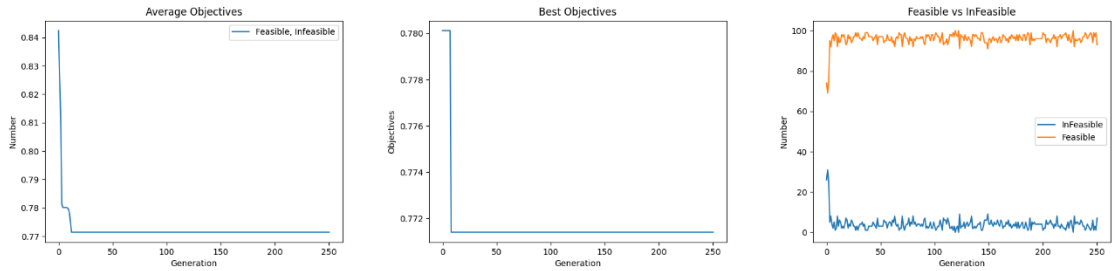
cl_03_100_10:



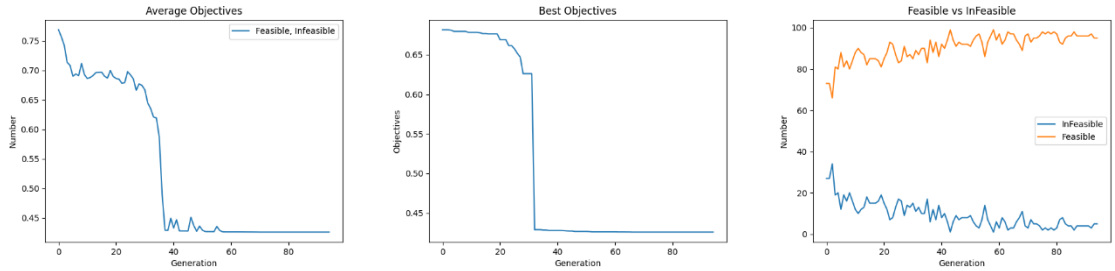




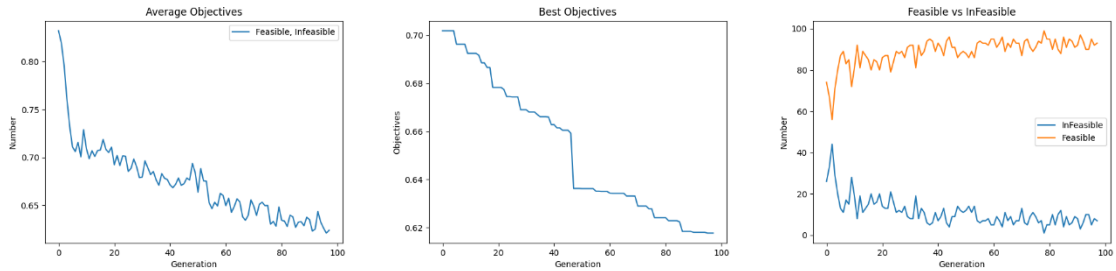
cl_04_020_03:



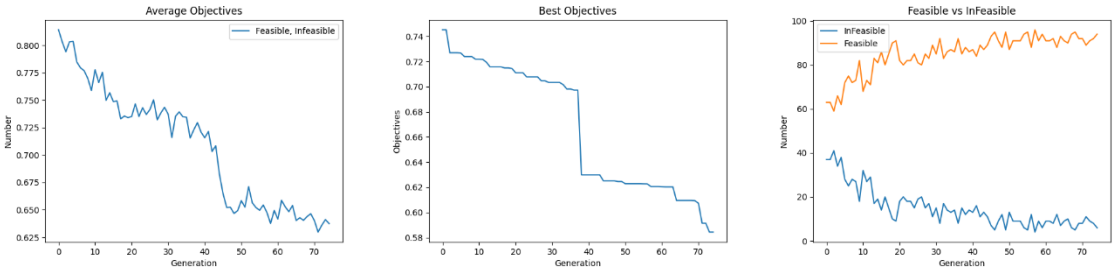
cl_04_040_05:



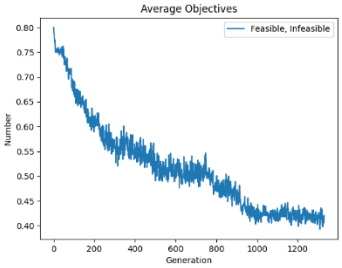
cl_04_060_07:

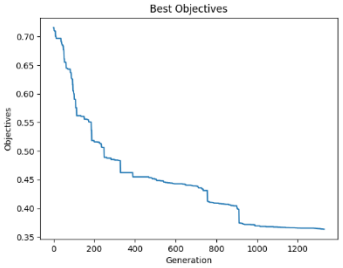


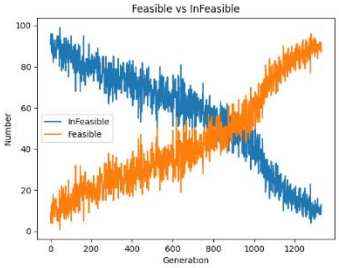
cl_04_080_08:



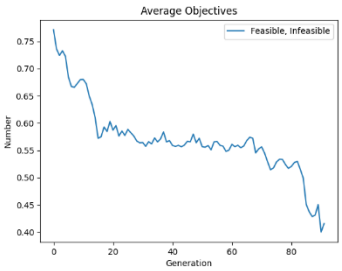
cl_04_100_09:

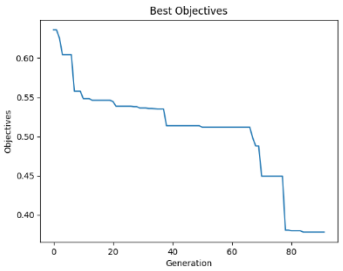


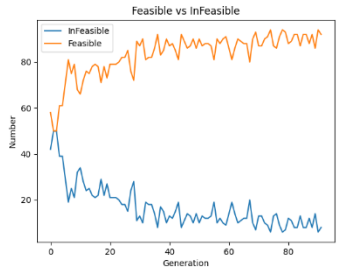




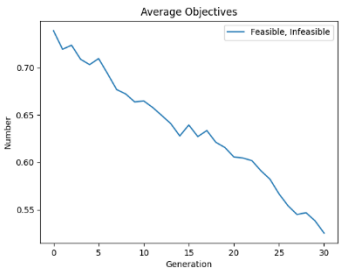
cl_05_020_04:

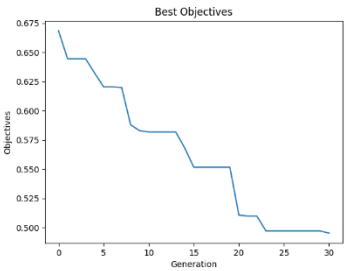


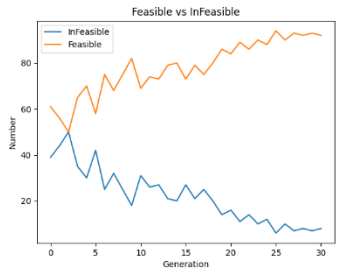




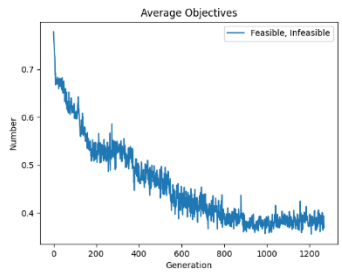
cl_05_040_03:

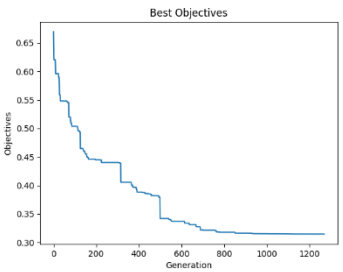


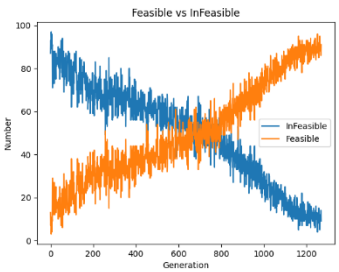




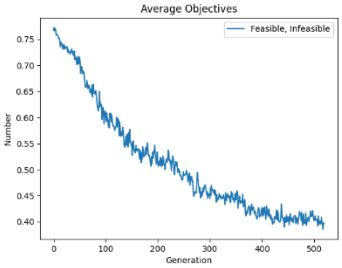
cl_05_060_05:

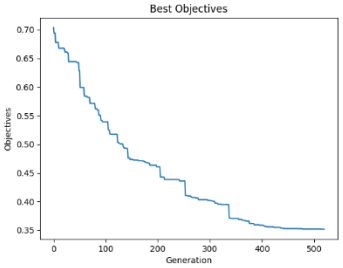


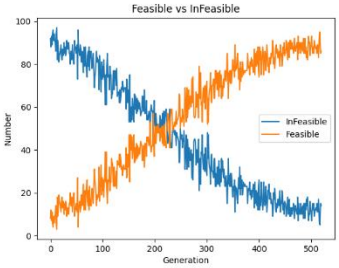




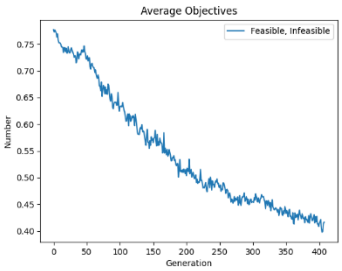
cl_05_080_07:

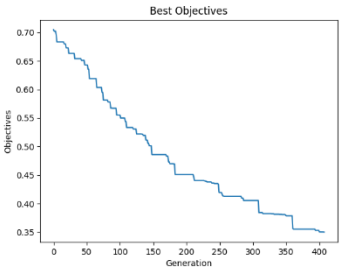


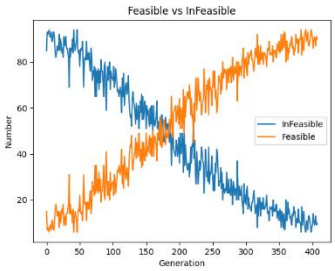




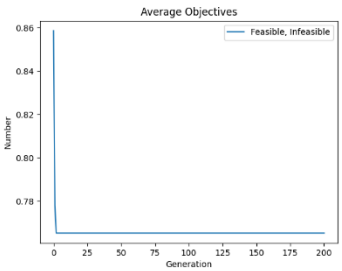
cl_05_100_08:

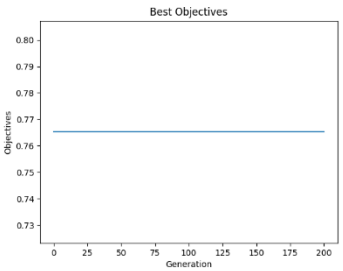


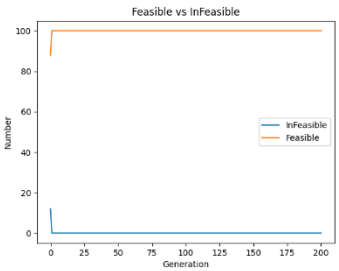




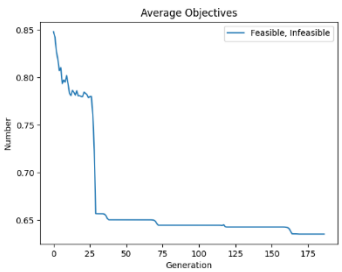
cl_06_020_05:

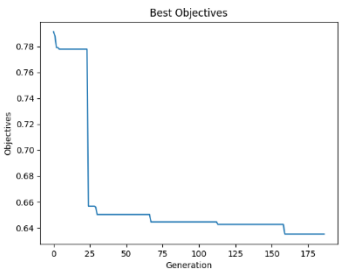


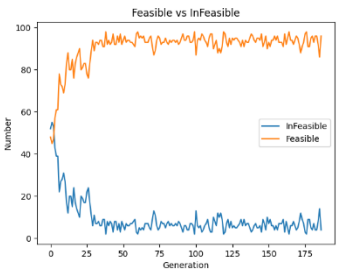




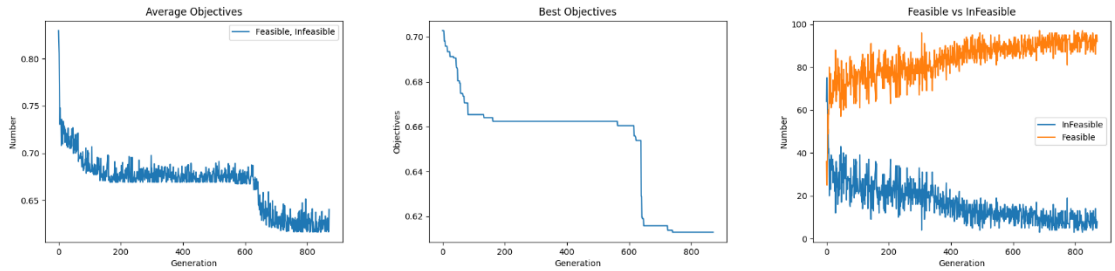
cl_06_040_04:



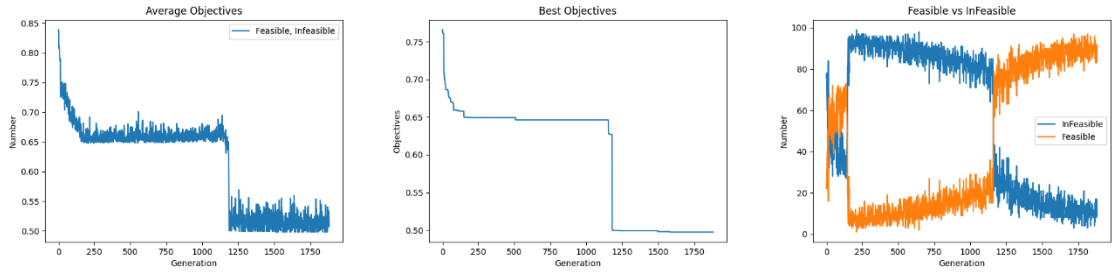




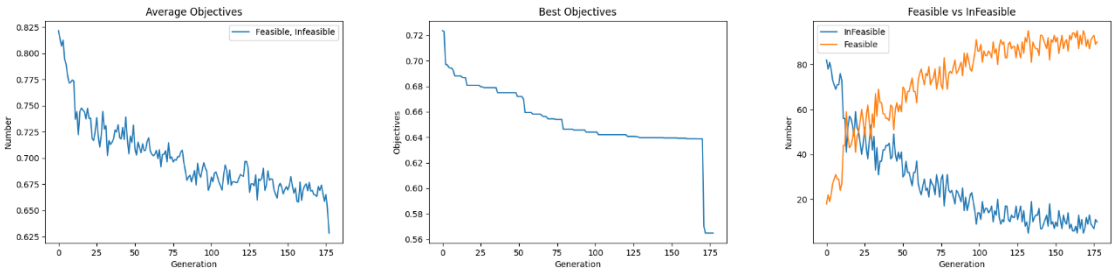
cl_06_060_02:



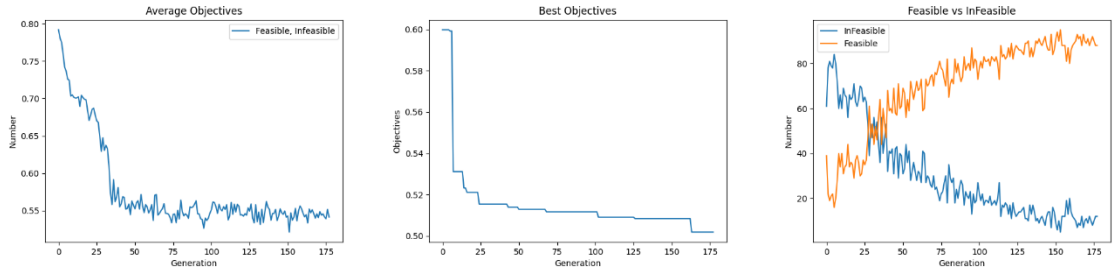
cl_06_080_01:



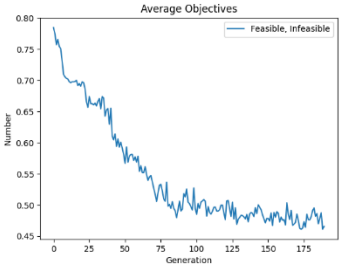
cl_06_100_10:

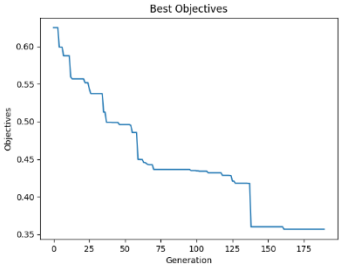


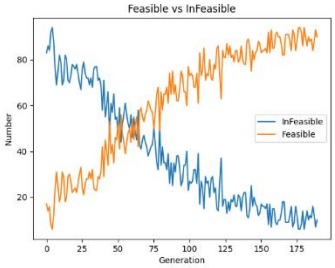
cl_07_020_08:



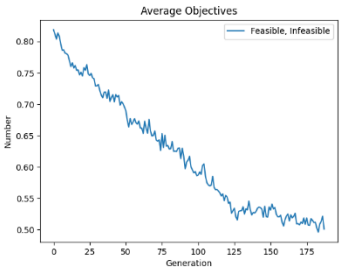
cl_07_040_06:

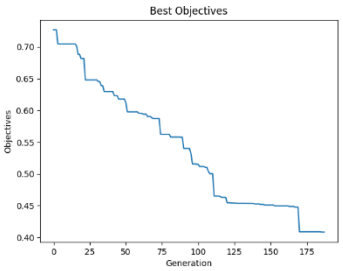


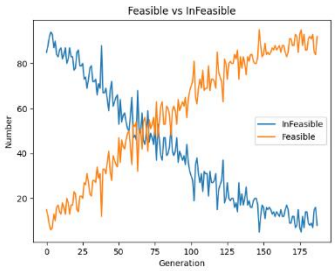




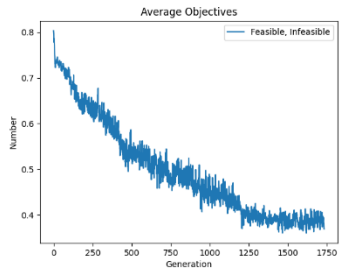
cl_07_060_05:

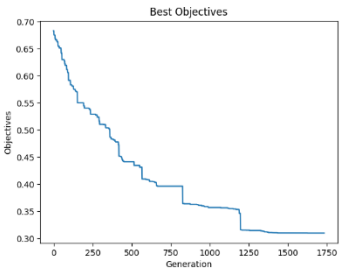


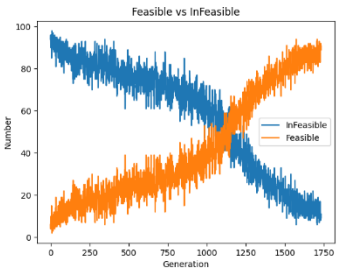




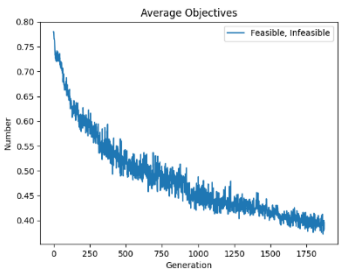
cl_07_080_08:

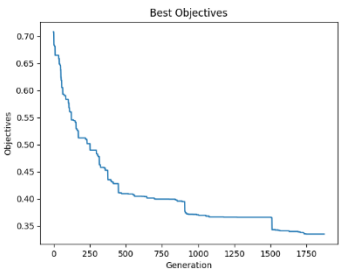


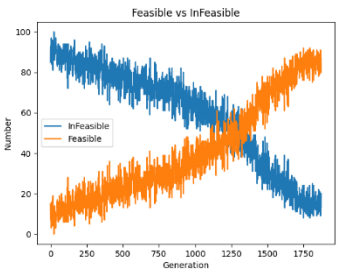




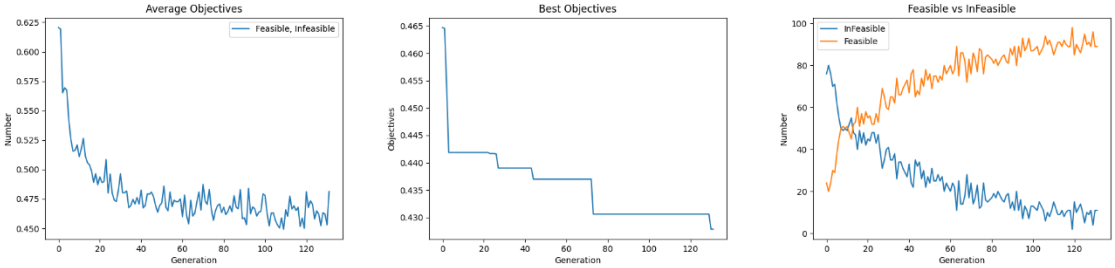
cl_07_100_10:



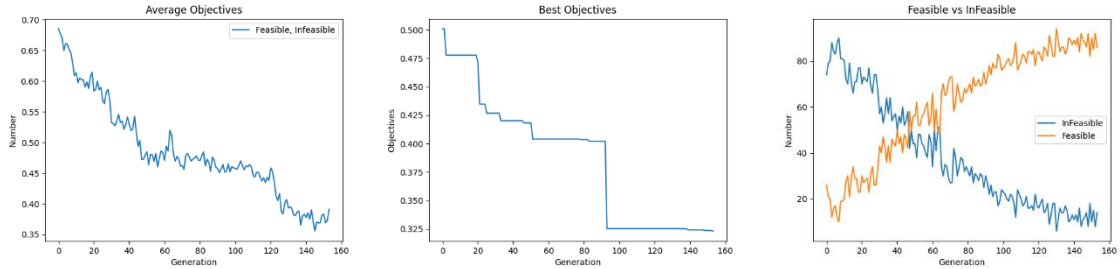




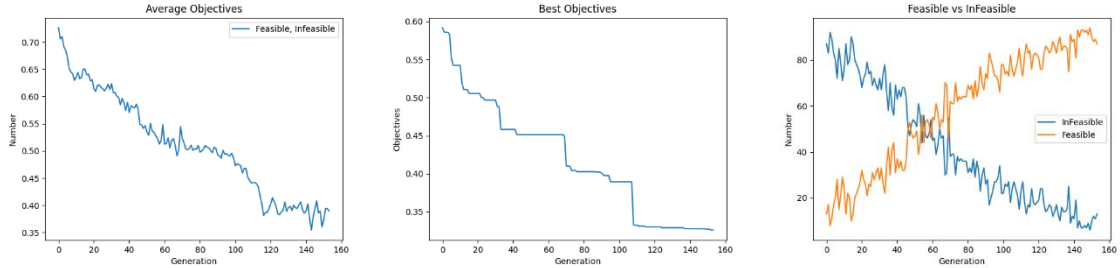
cl_08_020_05:



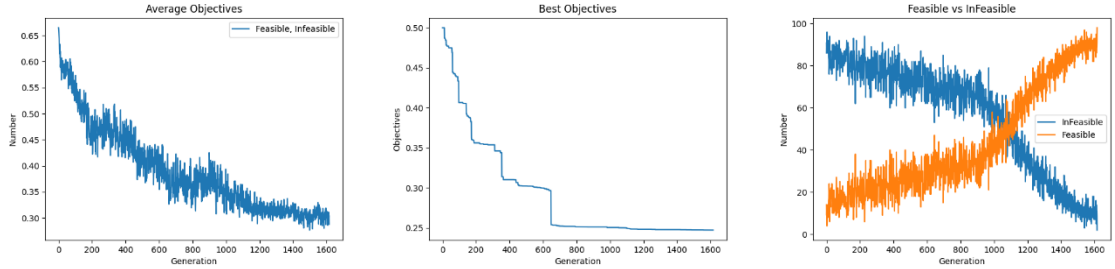
cl_08_040_03:



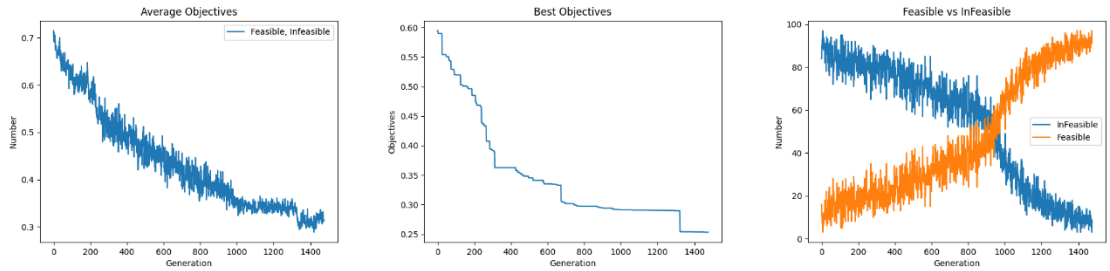
cl_08_060_05:



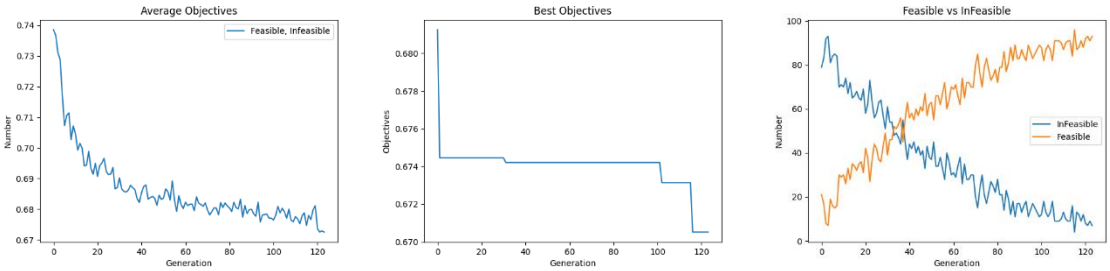
cl_08_080_07:



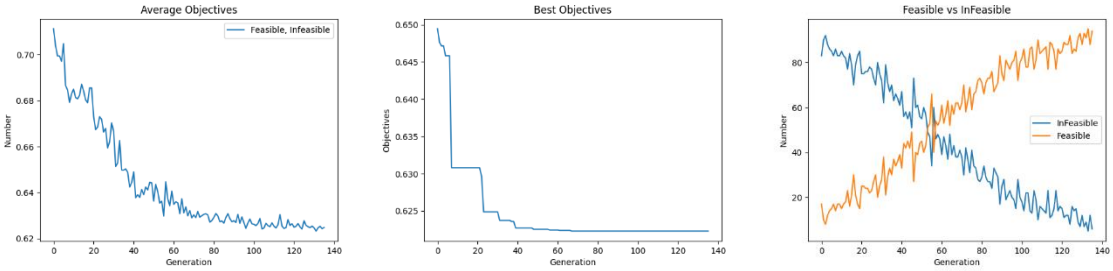
cl_08_100_03:



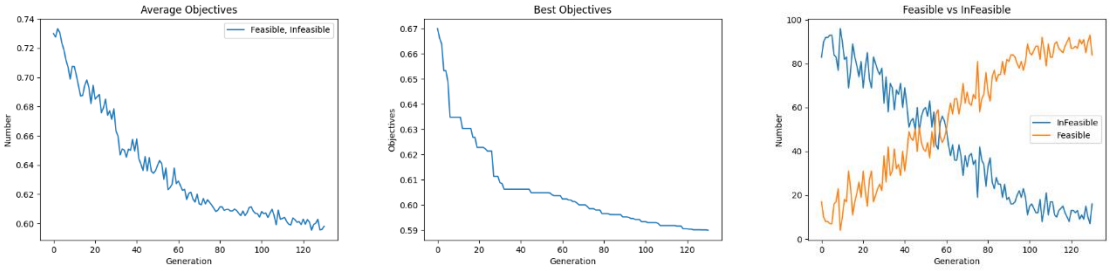
cl_09_020_04:



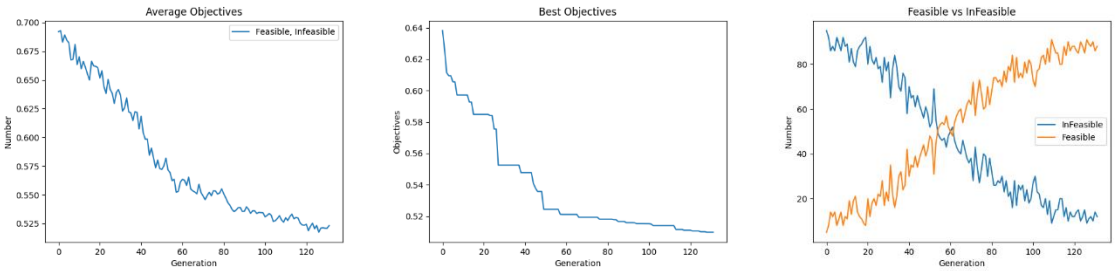
cl_09_40_10:



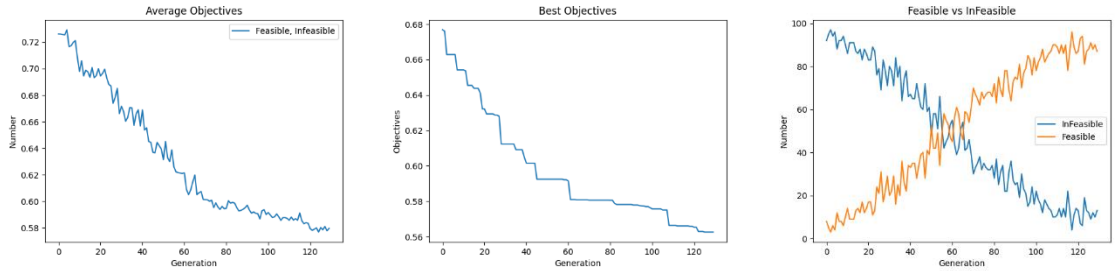
cl_09_060_07:



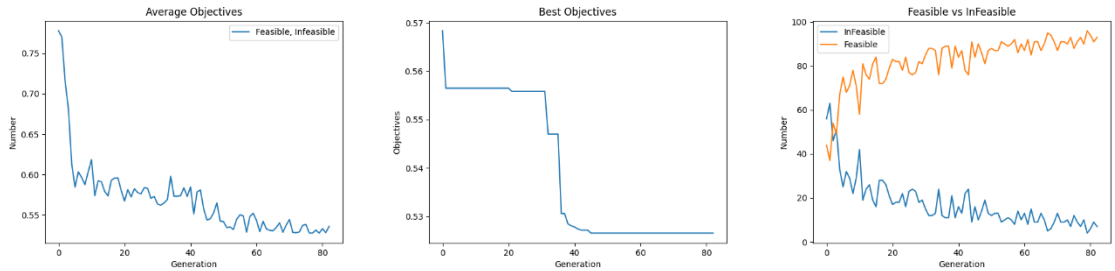
cl_09_080_04:



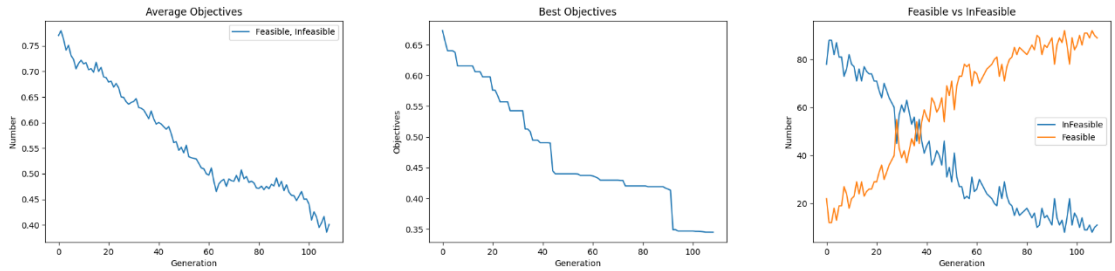
cl_09_100_05:



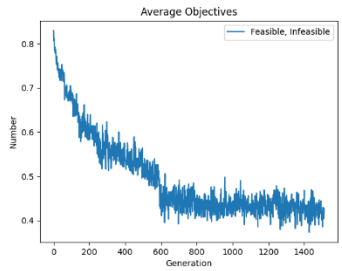
cl_10_020_08:

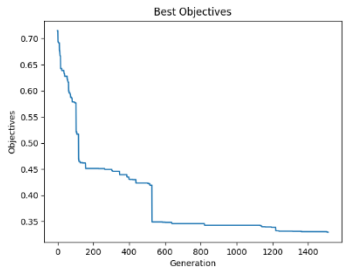


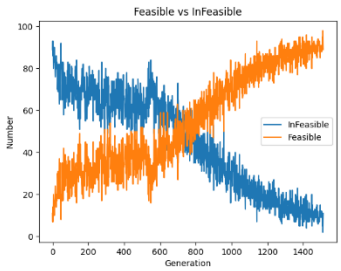
cl_10_040_03:



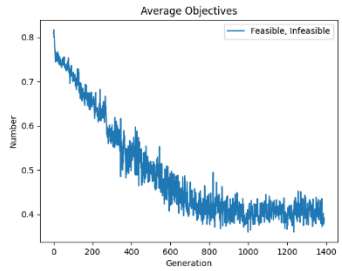
cl_10_060_05:

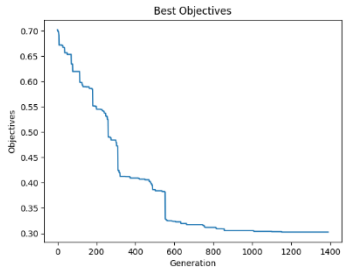


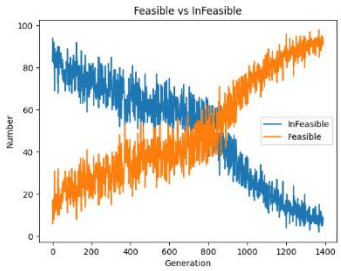




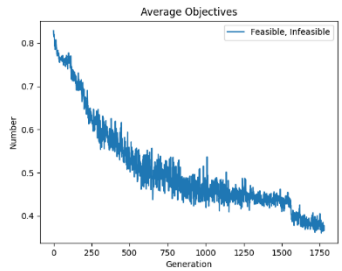
cl_10_080_02:

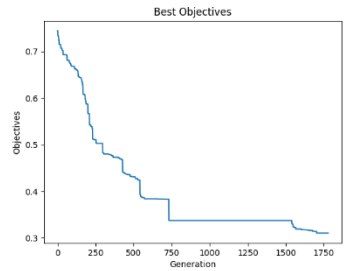


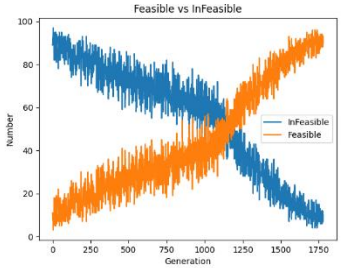




cl_10_100_04:







Discussion:

1. ‘mutation_probability’ (0.1)

- Exploration: A higher mutation rate increases the likelihood of introducing new genetic variations into the population, aiding in the exploration of new areas of the solution space; however, due to the stochastic nature of mutation, a high probability may prevent the algorithm from converging.

- Exploitation: While beneficial for exploration, excessive mutation can disrupt the refinement process of already good solutions, potentially hindering exploitation.

- Diversity Preservation: By creating new variants, mutation helps maintain genetic diversity within the population, preventing premature convergence.

2. ‘crossover_probability’ (0.9)

- Exploration: High crossover probability encourages the combination of different genetic materials from parent solutions, potentially leading to new and useful traits.

- Exploitation: Effective crossover exploits the good characteristics of parent solutions, potentially improving offspring quality.

- Diversity Preservation: While crossover can introduce diversity by combining traits, overly frequent crossover might lead to a loss of diversity if the same high-fitness individuals are repeatedly selected as parents.

4. ‘tournament_size’ (5)

- Exploration: Larger tournament sizes tend to select better individuals for reproduction, which might reduce exploration due to focusing on already successful solutions.

- Exploitation: Promotes exploitation by ensuring that stronger individuals have a higher chance of passing their genes to the next generation.

- Diversity Preservation: Smaller tournament sizes maintain diversity by giving less fit individuals a chance to be selected, whereas larger sizes might reduce diversity by favoring the fit ones excessively.

5. ‘population_size’ (100)

- Exploration: A larger population size provides a broader genetic base, allowing the algorithm to explore more diverse solutions.

- Exploitation: A larger population also means more competition among individuals, which can enhance the quality of solutions through natural selection.

- Diversity Preservation: More individuals mean more genetic diversity, which is crucial for avoiding local optima and ensuring robustness in the search process.

7. ‘elite_percentage’ (0.1)

- Exploration: Directly focuses on preserving high-fitness individuals rather than exploring new areas.

- Exploitation: Promotes exploitation by ensuring that the best solutions are carried over to subsequent generations unchanged.

- Diversity Preservation: While it ensures the survival of the fittest, too high an elitism percentage can reduce diversity by dominating the population with elite individuals.

8. ‘init_infeasibility’ (0.1)

- Exploration: Allows a portion of the initial population to be infeasible, potentially exploring riskier or unconventional areas of the solution space.

- Exploitation: Low initial focus on feasibility can delay the exploitation of optimal solutions as the algorithm may need to adjust infeasible solutions first.

- Diversity Preservation: Introducing infeasibility initially can enhance diversity by including a variety of solution types, not just those that are immediately feasible.

9. ‘Local Search’:

- Exploration: Local search can indirectly encourage exploration by optimizing individuals to levels that may lead them to explore different niches in the solution space.

- Exploitation: Local search primarily enhances exploitation by fine-tuning promising solutions to achieve superior local optima.

- Diversity Preservation: Although local search can reduce diversity by converging solutions, smart application only to select individuals can help maintain genetic diversity within the population.

Conclusion

The project on 2D bin packing that uses a memetic approach, combining a genetic algorithm with local search techniques such as hill climbing, represents a sophisticated strategy to optimize packing efficiency. The genetic foundation of this system employs a 2-point group-based crossover, which maintains diverse genetic contributions while also handling infeasible chromosomes through an innovative repair mechanism. This mechanism has a unique feature: the probability of repair increases with each generation, enabling gradual enhancement of solution feasibility without hastily discarding potentially valuable genetic material.

Incorporating infeasible solutions into the genetic algorithm for the 2D bin packing project, while it may initially slow the search process, plays a crucial role in enhancing the algorithm's overall effectiveness. This approach enables the exploration of new genetic material and assists in escaping from local optima, crucial for navigating complex solution landscapes.

Additionally, the fitness function is rigorously designed to incorporate an escalating penalty for infeasible solutions. This penalization strategy ensures that while the algorithm progressively favors feasible and optimally packed configurations, it remains robust by discouraging the selection of infeasible solutions, thus refining the overall search process.

The integration of hill climbing as a local search method enriches this genetic algorithm, transforming it into a memetic algorithm. This integration not only allows for a broad exploration of the solution space but also a focused exploitation of high-potential areas, thereby enhancing the precision in packing configurations.

This adaptive, dual-strategy approach significantly increases the efficiency and effectiveness of solving the complex problem of 2D bin packing. It leverages both the global search capabilities of the genetic algorithm and the local optimization power of hill climbing, ensuring a comprehensive exploration and exploitation that leads to high-quality, feasible packing solutions.