Homework 2

Tabu Search

Advanced Algorithm

Supervisor:

Dr. Ziarati

Atefe Rajabi

40230563

Spring 2024

Index

Abstract:

This document explores the application of the Tabu Search algorithm to the one-dimensional bin packing problem (1D-BPP), a classical combinatorial optimization challenge. The bin packing problem involves arranging items into the fewest number of bins without exceeding the capacity of any bin. Due to its NP-hard nature, finding an exact solution grows increasingly infeasible with the size of the dataset, making heuristic approaches like Tabu Search a valuable alternative. This study utilizes a variety of Tabu Search modifications, such as dynamic tabu tenure and strategic neighborhood generation, to enhance the search process. Parameters are carefully tuned to balance between exploration and exploitation, aiming to reduce the total number of bins used while optimizing the packing arrangement.

Introduction:

The one-dimensional bin packing problem (1D-BPP) is a well-known NP-hard problem that poses significant challenges in fields ranging from logistics to data storage. The objective is to minimize the number of bins used to store a set number of items, each with a specific size, without exceeding the capacity of any bin. Traditional methods often fall short, especially as problem instances increase in complexity and size, necessitating more sophisticated approaches like the Tabu Search heuristic.

Tabu Search, a metaheuristic algorithm, is renowned for its capability to overcome local optima and explore a broader solution space through mechanisms that forbid returning to recently visited solutions. This document details the implementation of the Tabu Search algorithm tailored to the bin packing problem, emphasizing the adaptability and robustness of this method in solving complex optimization problems. The project explores various components of the algorithm including initial solution generation, neighborhood structures, aspiration criteria, and dynamic tabu tenure adjustments.

Bin Packing problem:

The 1D bin packing problem (1D-BPP) is a classical combinatorial optimization problem that deals with efficiently packing a finite set of items into bins. Each item has a weight (or size), and each bin has a maximum capacity that cannot be exceeded. The goal is to minimize the number of bins required to pack all the items. This problem is an example of an NP-hard problem, meaning it is computationally challenging to find an exact solution, especially as the size of the problem grows.

Here are the key components of the 1D bin packing problem:

1. Items: Each item has a specific size or weight.

2. Bins: All bins have the same maximum capacity, and no bin can hold items exceeding this capacity.

3. Objective: The primary objective is to minimize the number of bins used to pack all the items.

This problem has practical applications in various industries, such as logistics for optimizing space in cargo containers, in computing for data storage optimization, and in manufacturing for minimizing material waste.

The challenge lies in how to arrange the items in the bins in such a way that the number of bins is minimized, considering the capacity limit of each bin. Various algorithms, especially heuristic and metaheuristic algorithms, are employed to find near-optimal solutions to this problem due to its complexity and the impracticality of solving it exactly for large instances.

Solution Feasibility:

If both checks are passed—no bins are overloaded and the total weight of items in the bins matches the total weight of all items—the function returns ‘true’, confirming that the solution is feasible. This means:

- All items are accounted for without duplication.

- No bin exceeds its capacity.

This feasibility check is crucial for ensuring that any proposed solution to the bin packing problem adheres to the fundamental constraints of the problem, thereby confirming that the solution is viable and correct.

Tabu Search:

Tabu search is a metaheuristic search algorithm used to solve complex optimization problems, including the bin packing problem. It extends the capability of local search methods by using memory structures that describe the visited solutions or user-defined rules to escape local optima and explore the solution space more thoroughly.

Tabu search can be particularly effective for the bin packing problem.

1. Initial Solution:

- Start with an initial solution, which can be generated randomly (In our implementation it has been generated randomly). This solution is the current solution and also the best solution found so far.

2. Neighborhood Generation:

- Generate a "neighborhood" of solutions around the current solution. This involves making small changes to the arrangement of objects in bins. For example, moving an item from one bin to another or swapping items between bins.

3. Choosing the Next Solution:

- From the neighborhood, select a candidate solution that has the least number of bins used, which might not necessarily be better than the current global best. This step incorporates a crucial element of tabu search: some moves (like revisiting a recent configuration) are marked as "tabu" (forbidden) unless they satisfy certain criteria (such as yielding a new best solution), which helps in avoiding cycles and getting stuck in local optima.

4. Tabu List:

- Maintain a tabu list that records certain attributes of the recent moves (e.g., recently moved or swapped items) to prevent the algorithm from revisiting them.

5. Aspiration Criteria:

- Sometimes, a move that is marked as tabu can be allowed if it meets an aspiration level, typically if it results in a solution better than any previously found solution.

6. Termination Conditions:

- The algorithm terminates when a stopping criterion is met, which could be a time limit, a maximum number of iterations, reaching a convergence threshold or evaluation count threshold.

7. Update Records:

- Throughout the process, the best solution is updated whenever a better solution is found. This record-keeping helps in guiding the search and in implementing the aspiration criteria.

Parameters:

Each parameter plays a specific role in influencing the behavior and performance of the algorithm. Let’s go through each parameter and understand its significance:

1. ‘z’: Z is a constant used to define the equilibrium of the filled bin.

2. ‘max_iteration’:

- This is the maximum number of iterations the algorithm will execute before stopping. It acts as a hard limit on the total number of algorithmic steps (including all inner loops and procedures) that will be performed, preventing the algorithm from running indefinitely.

3. ‘no_improvement_threshold’:

- This parameter specifies the number of consecutive iterations during which no improvement is found in the solution before the algorithm stops. It's used to prevent unnecessary computation if the solution quality is not improving.

4. ‘stopType’:

5. ‘convergence_threshold’:

- This is the threshold used to determine when the algorithm has sufficiently converged to a solution.

6. ‘time_limit’:

- The maximum time (in seconds) that the algorithm should run. If this time limit is reached, the algorithm stops.

7. ‘tabuSize’:

- The size of the tabu list, which is the number of moves or solutions that are kept in memory as "forbidden". This prevents the algorithm from cycling back to these states, helping to escape local optima and encouraging exploration of new areas in the solution space. The effects of tabu size on exploration and exploitation will be discussed further.

8. ‘tweaksCount’:

- The number of tweaks or modifications made to generate the neighborhood of solutions in each iteration. This can dictate the breadth of search and has a direct impact on the exploration capabilities of the algorithm. The effects of tweaks count on exploration and exploitation will be discussed further.

9. ‘aspiration_probability’:

- The probability that an aspiration criterion will override the tabu status of a move. An aspiration criterion typically allows a tabu move if it results in a solution that is better than any solution previously encountered.

Parameters value:

{

"z": 2,

"max_iteration": 300000, // preventing infinitely running

"no_improvement_threshold": 20,

"stopType": 2, // convergence stopping condition

"convergence_threshold": 0.1, // is dynamically set within the code based on item size of each instance set

"time_limit": 10,

"tabuSize":20, // will modified based on searching improvement

"tweaksCount": 50,

"aspiration_probability" : 0.1 // the function is written but not used.

}

Objective Function:

The objective function proposed in the paper[1] focuses on optimizing the placement of items in bins by considering not only the number of bins used but also the amount of unused space within these bins. Specifically, to avoid stagnation and encourage the algorithm towards finding optimal solutions, the objective function incorporates the amount of unused space alongside the count of bins.

Minimization Objective Function: This function aims to minimize the inverse of the sum of the filled capacity ratios of all bins. Mathematically, it is represented as minimizing

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

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

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

The constant z in the objective functions of the optimization problem is a significant parameter that influences how the solution's quality is assessed in terms of both bin utilization and the number of bins used. Here's a detailed explanation:

Role of z

The constant z plays a crucial role in balancing these two aspects. Specifically, it adjusts how much emphasis is placed on the filled capacity of the bins relative to their total capacity.

Mathematical Interpretation

The exponent z modifies the impact of the ratio $fill_k/C$ on the objective function:

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

- When $z > 1$, the function places increasing emphasis on bins that are closer to being full. This means that as bins get filled closer to their capacity, they contribute more significantly to the objective function, encouraging solutions that efficiently utilize bin space.

- When $z = 1$, the function simplifies to a linear relationship with the fill ratio. In this case, each percentage point increase in bin utilization is valued equally, regardless of how full the bin is.

- When $z < 1$, the function would, hypothetically, place less emphasis on bins being fully utilized.

Algorithm:

The provided project implements a Tabu Search algorithm for the bin packing problem, aiming to minimize the number of bins required to store items. At the outset, the algorithm initializes with a random bin packing configuration and computes its objective function, which has been represented before. Throughout its execution, the algorithm iteratively explores the neighborhood of the current solution by applying various "tweaks," which are minor adjustments to the bin configuration. Each tweak is evaluated, and its impact on the bin packing configuration is assessed through the objective function.

Within each iteration, the algorithm maintains a dynamic list of tabu moves—recently made changes that should not be immediately reversed—to prevent cycling back to inferior solutions. This list adjusts its size based on the tenure calculation, which is influenced by the improvement trends in the objective function values. If a new configuration yields a better objective value and is not prohibited by the tabu status, it becomes the current solution, and its details are recorded. The algorithm employs a system of rewards and penalties to adjust the probabilities of selecting future tweaks, encouraging exploration of more promising configurations. It terminates based on several conditions: time elapsed, number of iterations, or convergence, where convergence is assessed by the lack of significant improvement over a set number of iterations. This approach allows the algorithm to efficiently explore the solution space, balancing between intensifying the search around promising areas and diversifying to explore new possibilities, thereby enhancing the likelihood of finding an optimal or near-optimal solution.

Neighborhood:

Tweak Functions

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

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

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

Adaptive Selection Mechanism[2]

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

1. Initialization and Updating of Probabilities:

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

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

2. Reassign Probabilities:

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

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

- This is done by computing a value for each tweak using the formula: $Value = \frac{rewards}{rewards+penalties}$ - These values are then normalized to ensure they sum to one, forming a new set of probabilities that guide the future selection of tweaks.

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

3. Selection of Tweak:

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

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

Tabu List:

In tabu search, the "tabu list" is a sort of short-term memory that keeps track of the recent moves (changes made to the solution) so that the algorithm does not repeat them. This helps prevent the algorithm from cycling back to previously visited solutions and encourages exploration of new areas in the solution space.

The provided C++ code defines a template-based class ‘TabuList’ that implements a tabu list, an essential component in Tabu Search algorithms used to avoid cycles and ensure the search explores new areas by temporarily forbidding certain moves. This data structure is designed to store and manage tabu moves, which are represented as tuples of transitions (‘FromType’ to ‘ToType’) associated with specific items (‘KeyType’).

Structure and Operations

1. Storage Mechanism:

- The class utilizes an unordered map (‘std::unordered_map’) where each key (an item identifier of type ‘KeyType’) maps to a queue of changes (transitions from ‘FromType’ to ‘ToType’). This setup allows for efficient lookup, insertion, and removal of tabu states associated with specific items.

- Additionally, a list (‘std::list’) named ‘orderList’ maintains the order of insertion for these tabu entries. Each entry in this list contains both the key and the change, ensuring that the oldest entries can be removed in FIFO (First-In-First-Out) order once the list exceeds the predefined maximum size (‘maxTabuSize’).

2. Functional Operations:

- Insertion (‘put’): Adds a new tabu move to the structure. If the tabu list is full (i.e., its size equals ‘maxTabuSize’), the oldest entry is removed using the ‘pop’ operation before the new entry is added. This ensures the tabu list does not exceed its capacity limit, maintaining the most recent tabu moves.

- Removal (‘pop’): Removes and returns the oldest tabu entry from the list. This operation also handles the removal of this entry from the corresponding queue in the map, and if the queue becomes empty after this operation, the key is erased from the map to prevent memory waste.

- Query (‘get’): Retrieves all changes associated with a given item key, returning them as a vector of tuples. This is useful for checking if a specific move is currently tabu.

- Display (‘display’): Outputs the current state of the tabu list to the console, showing all tabu moves stored along with their item identifiers. This is helpful for debugging or monitoring the algorithm’s progress.

The ‘TabuList’ class is a critical utility for managing tabu statuses within Tabu Search, enabling the algorithm to systematically forbid and later permit moves as their tabu tenure expires. This management helps prevent the algorithm from getting stuck in local optima and encourages exploration of the search space by revisiting previously excluded solutions only after sufficient time has passed.

Tabu Tenure:

Tabu tenure is a critical concept in the Tabu Search algorithm. It refers to the duration (number of iterations) for which certain moves or solutions are classified as "tabu" or forbidden. The primary aim of establishing a tabu tenure is to prevent the search procedure from cycling back to previously visited solutions, thereby encouraging the exploration of new regions in the solution space and helping the algorithm to escape from local optima.

The tabu tenure is a balancing act. Too short a tenure might not sufficiently discourage cycling, leading the search to revisit previous solutions. Too long a tenure can overly restrict the search, preventing the algorithm from thoroughly exploring the vicinity of promising solutions.

Dynamic Tenure:

Normally, the size of the tabu list (how many moves are remembered as "tabu") is fixed before the search starts and doesn't change. However, a fixed size might not be ideal because as the search progresses, the suitability of that size might change depending on how the search landscape looks.

The ‘calculate_tenure_based_on_quality’ function in the ‘TabuSearch’ class is a straightforward implementation of dynamic tenure, where the tabu tenure is adjusted based on the search's progress in improving the objective value. The function aims to modify the tenure (i.e., the length of time moves are kept in the tabu list) based on the lack of improvement in the solution's quality over successive iterations.

1. Checking for Stagnation:

- The function first checks if the ‘no_improvement_count’ (a counter tracking the number of consecutive iterations without an improvement in the objective function) has reached or exceeded a predefined ‘no_improvement_threshold’. This threshold is a critical parameter that determines how tolerant the algorithm is towards non-improving moves before it decides to intensify the search constraints.

2. Adjusting the Tabu Tenure:

- If the ‘no_improvement_count’ is greater than or equal to ‘no_improvement_threshold’, it indicates that the search has been stagnant for a considerable number of iterations without finding a better solution.

- In response to this stagnation, the function increases the ‘tabuSize’ by 10% (‘tabuSize * 1.1’). This adjustment is made to intensify the search's exploration by preventing the search from cycling back to recently explored and unfruitful areas, hence encouraging the exploration of new, potentially more promising areas of the solution space.

Implications and Effects

- Enhanced Exploration: By increasing the tenure during periods of no improvement, the algorithm effectively broadens the search area, potentially escaping suboptimal local minima.

- Dynamic Response: This method allows the tabu search to dynamically respond to the search's progress. If improvements are hard to come by, it becomes stricter in its tabu conditions, which is a flexible approach compared to static tabu tenure.

- Prevention of Cycling: A larger tabu list (in terms of tenure) helps prevent the algorithm from cycling back to previously considered solutions, which is crucial for the effectiveness of the tabu search, especially in complex search landscapes.

The ‘calculate_tenure_based_on_quality’ function is an implementation of dynamic tabu tenure that adjusts the constraints imposed by the tabu list based on the search's current performance. This approach helps maintain a good balance between exploration and exploitation by modifying the memory of the search (tabu list size) in reaction to observed search dynamics, specifically the occurrence of stagnation.

Aspiration Criteria:

Stopping Condition:

The ‘stopping_condition’ function in the Tabu Search algorithm defines multiple criteria for when the search should terminate, based on user-configurable parameters. The method employs a switch statement to handle different types of stopping conditions: time limits, iteration limits, and convergence.

1. Time Limit: The ‘stopBy_time’ method checks if the current runtime has exceeded a predefined duration (‘loop_duration’).

2. Iteration Limit: The ‘stopBy_iteration’ method compares the current number of iterations (‘current_iteration’) against a maximum allowed value. This criterion prevents the search from continuing indefinitely and is useful when the search space is large or when diminishing returns are observed in solution improvement as iterations increase.

3. Convergence: The ‘stopBy_Convergence’ method employs a more nuanced approach, stopping the search if no significant improvement is detected over a series of iterations. Specifically, it checks if the count of iterations without improvement (‘count’) has reached a threshold of 25, indicating potential convergence to a local optimum. This method helps in avoiding unnecessary computations once the solutions are no longer meaningfully improving.

Libraries:

In the implementation of the Tabu search algorithm for the bin packing problem, several external libraries have been utilized to enhance functionality and streamline various operations. Matplotlibcpp is employed for plotting and visualization, enabling a graphical representation of the algorithm's performance over different iterations and configurations. This visualization aids in understanding the optimization trends and the effectiveness of different parameter settings. The Json library is used to handle configuration files, allowing for flexible and easily adjustable parameters that can be modified without altering the core codebase. This capability is crucial for experimenting with different algorithm settings to determine optimal configurations. Lastly, Libxl is integrated for its capabilities in reading from and writing to Excel files, facilitating easy data manipulation and storage. This is particularly useful for documenting the results in a structured format that is suitable for analysis and presentation, thereby supporting extensive evaluation and reporting of the algorithm's outcomes. These libraries collectively contribute to a robust development environment that supports efficient data handling, result visualization, and configuration management, crucial for comprehensive optimization studies.

Results:

All the plots, visualization, outputs are available in the project folder.

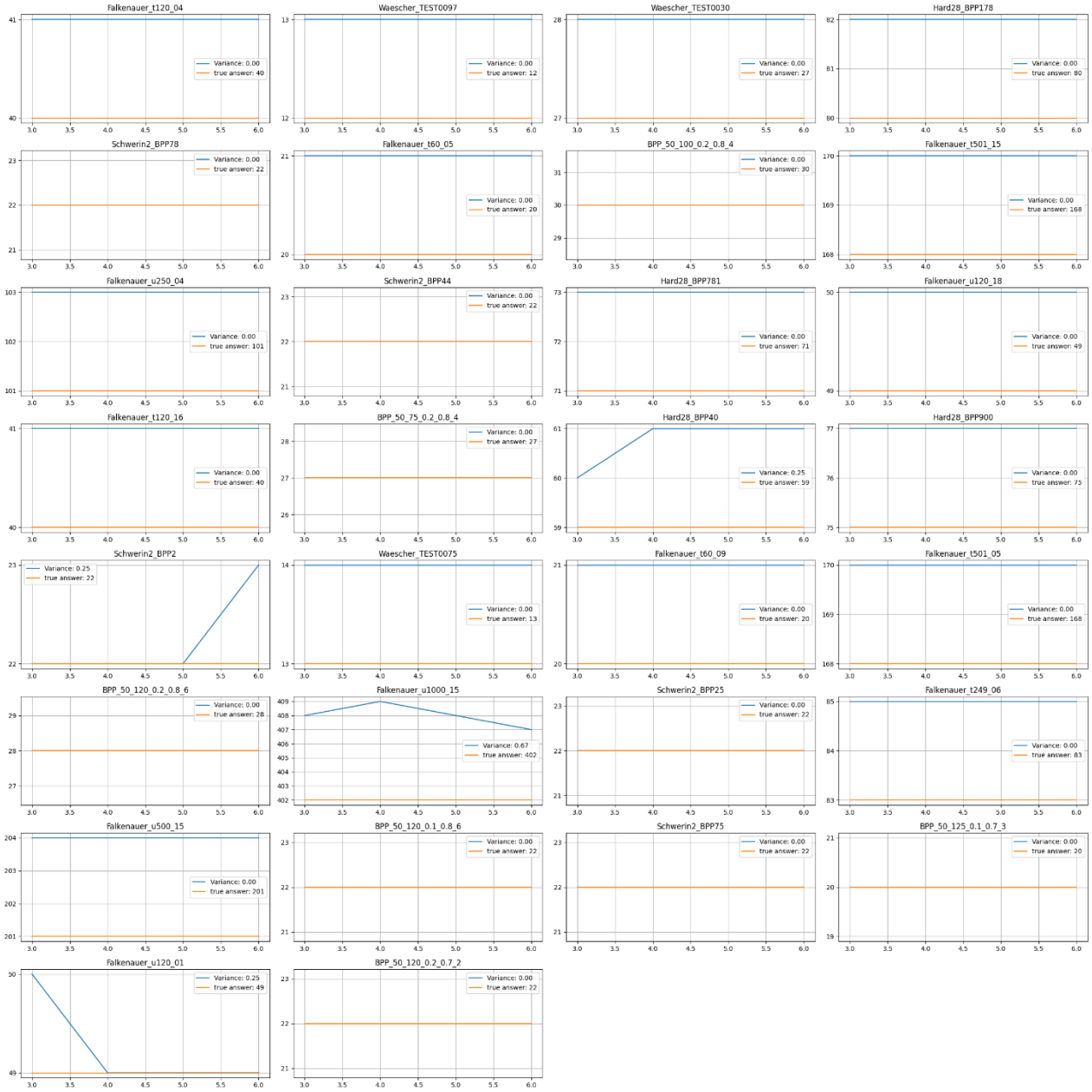
These results are derived from 20 algorithm execution.

file_name	Optimum solution	Best solution	Worst solution	Average time
Falkenauer_t120_04	40	41	41	
Waescher_TEST0097	12	12	12	
Waescher_TEST0030	27	28	28	
Hard28_BPP178	80	81	82	
Schwerin2_BPP78	22	22	22	
Falkenauer_t60_05	20	21	21	
BPP_50_100_0.2_0.8_4	30	30	30	
Falkenauer_t501_15	168	168	169	
Falkenauer_u250_04	101	101	102	
Schwerin2_BPP44	22	22	22	
Hard28_BPP781	71	72	73	
Falkenauer_u120_18	49	49	49	
Falkenauer_t120_16	40	41	41	
BPP_50_75_0.2_0.8_4	27	27	27	
Hard28_BPP40	59	60	61	
Hard28_BPP900	75	76	78	
Schwerin2_BPP2	22	22	22	
Waescher_TEST0075	13	13	14	
Falkenauer_t60_09	20	21	21	
Falkenauer_t501_05	168	168	169	
BPP_50_120_0.2_0.8_6	28	28	29	
Falkenauer_u1000_15	402	404	405	
Schwerin2_BPP25	22	22	22	
Falkenauer_t249_06	83	84	84	
Falkenauer_u500_15	201	201	202	
BPP_50_120_0.1_0.8_6	22	22	22	
Schwerin2_BPP75	22	22	22	
BPP_50_125_0.1_0.7_3	20	20	20	
Falkenauer_u120_01	49	49	49	
BPP_50_120_0.2_0.7_2	22	22	22	

Variance:

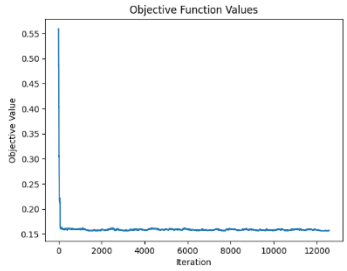
Orange line: Optimum solution

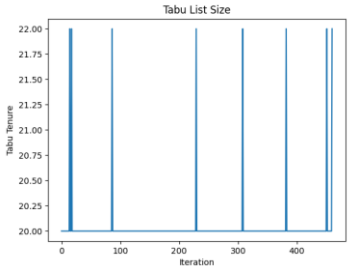
Blue line: Solution with Tabu Search



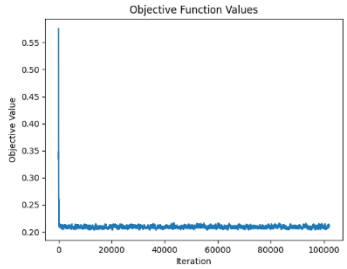
Plots:

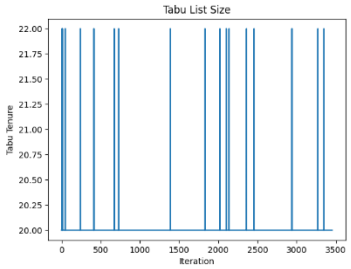
\BPP_50_75_0.2_0.8_4



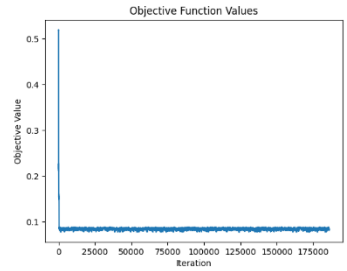


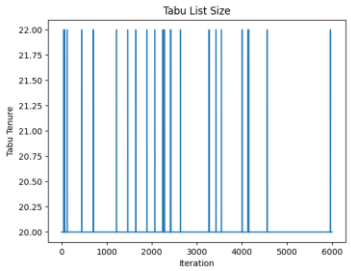
\BPP_50_100_0.2_0.8_4



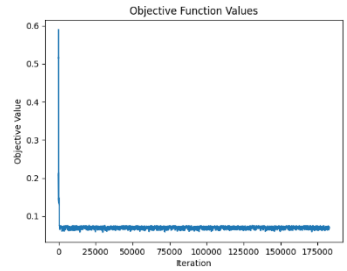


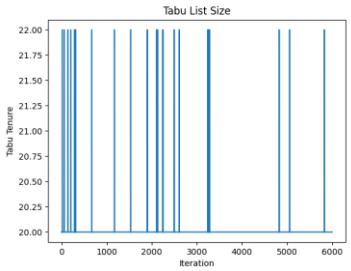
\BPP_50_120_0.1_0.8_6



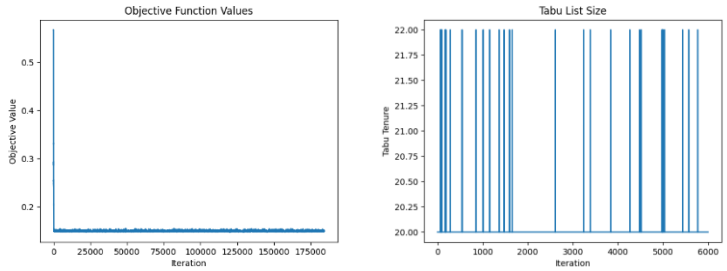


\BPP_50_120_0.2_0.7_2

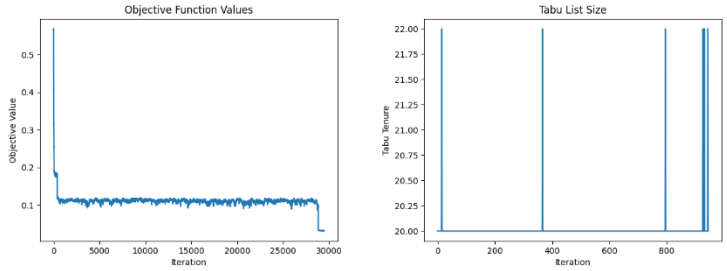




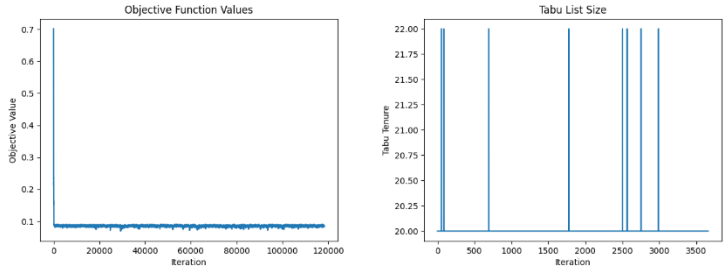
\BPP_50_120_0.2_0.8_6



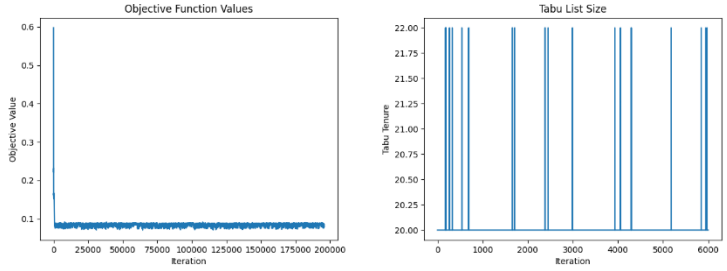
\BPP_50_125_0.1_0.7_3



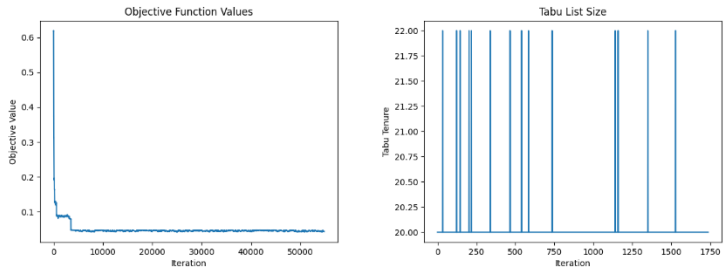
\Falkenauer_t60_05



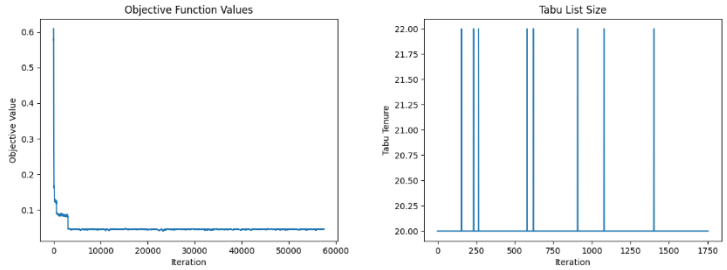
\Falkenauer_t60_09



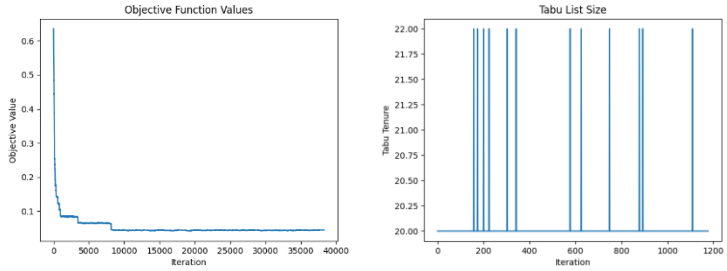
\Falkenauer_t120_04



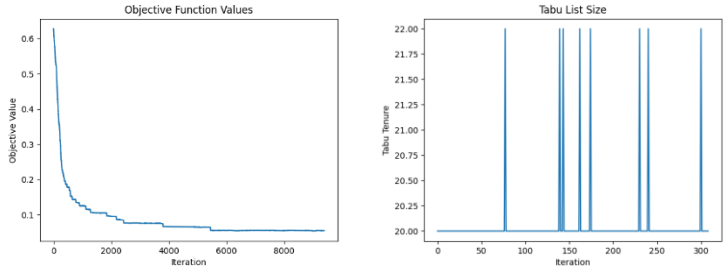
\Falkenauer_t120_16



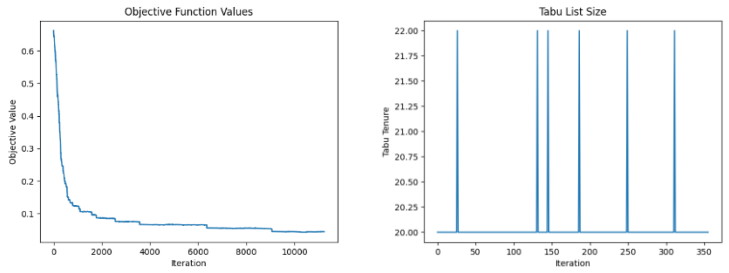
\Falkenauer_t249_06



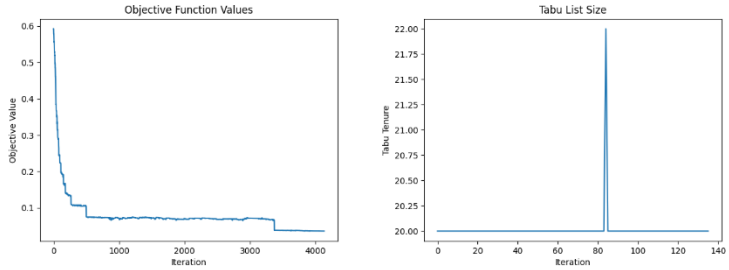
\Falkenauer_t501_05



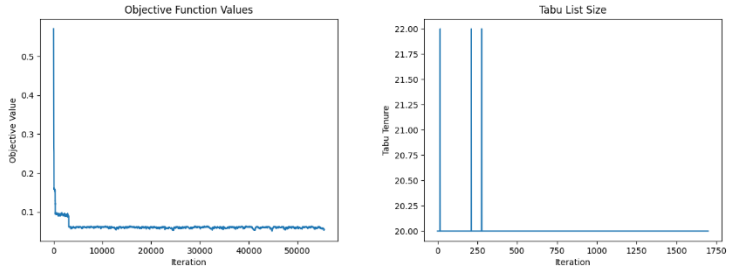
\Falkenauer_t501_15



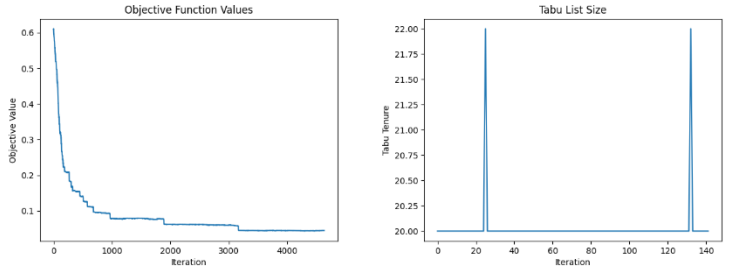
\Falkenauer_u120_01



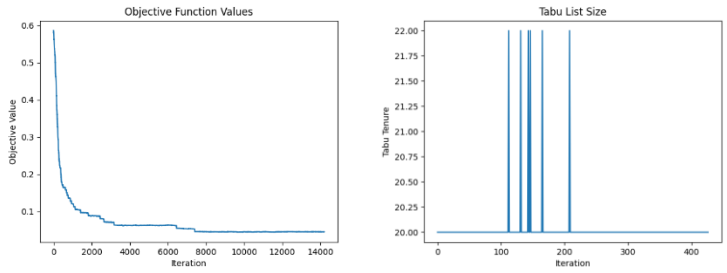
\Falkenauer_u120_18



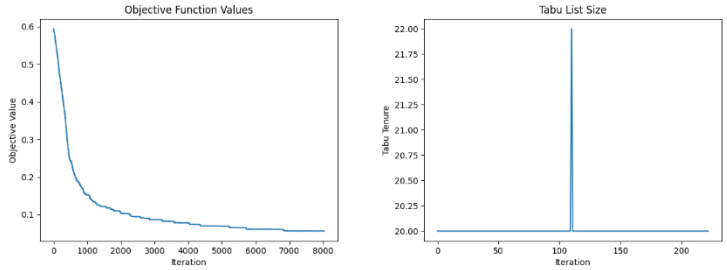
\Falkenauer_u250_04



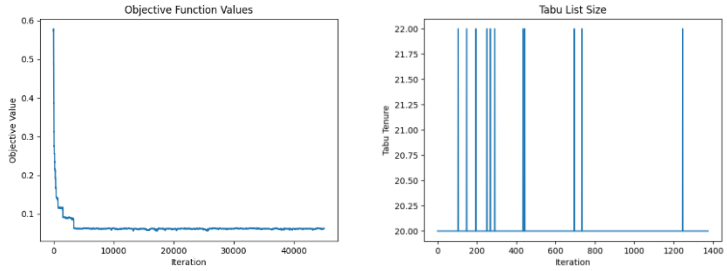
\Falkenauer_u500_15



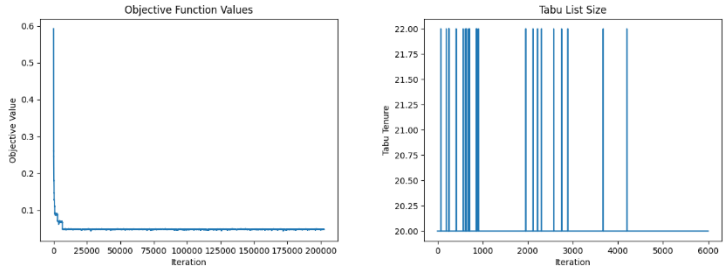
\Falkenauer_u1000_15



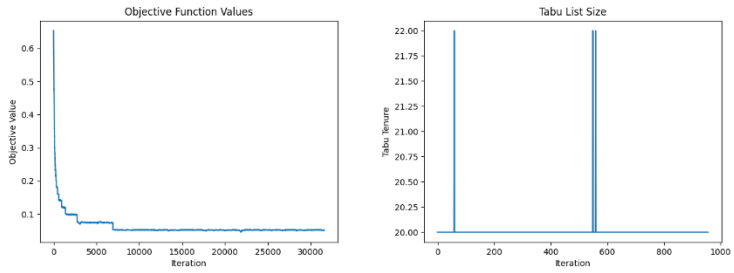
\Hard28_BPP40



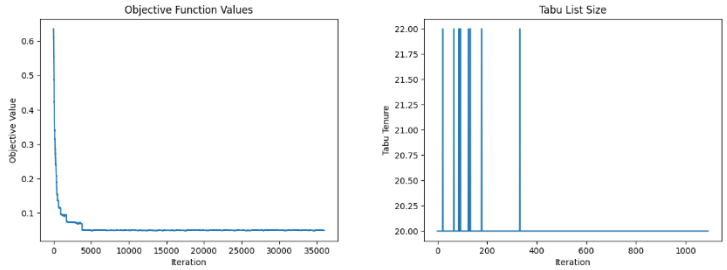
\Hard28_BPP178



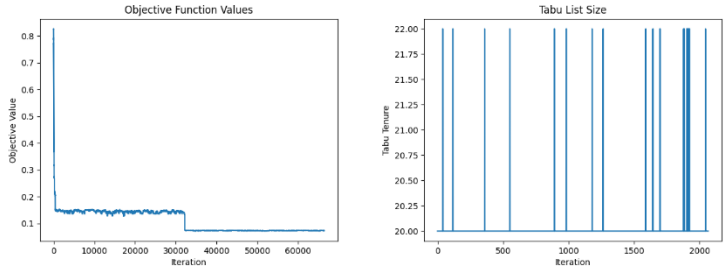
\Hard28_BPP781



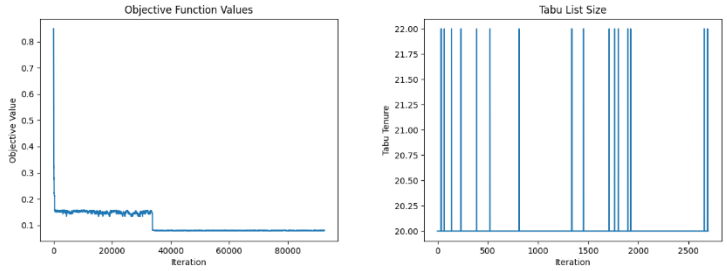
\Hard28_BPP900



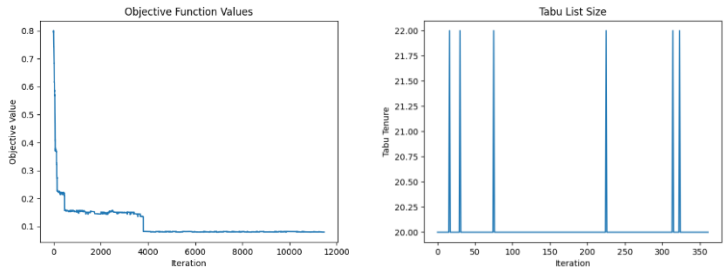
\Schwerin2_BPP2



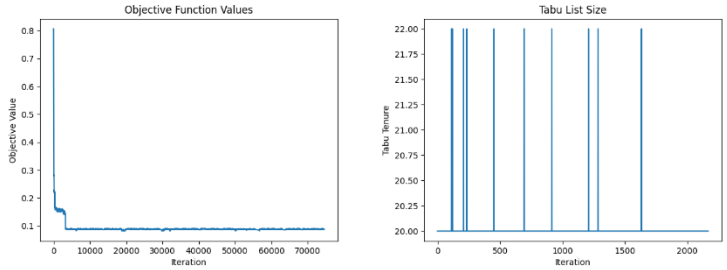
\Schwerin2_BPP25



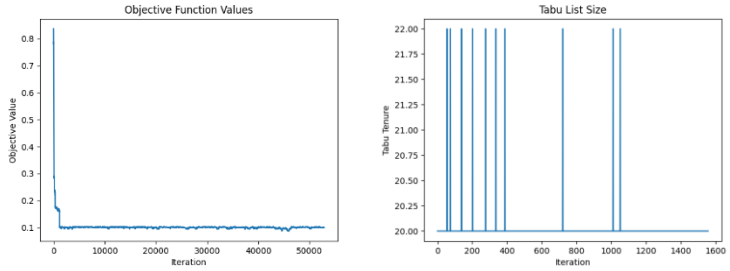
\Schwerin2_BPP44



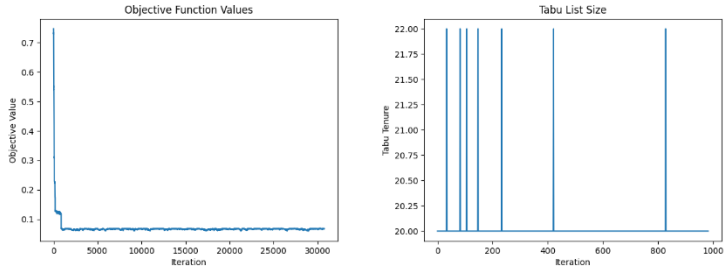
\Schwerin2_BPP75



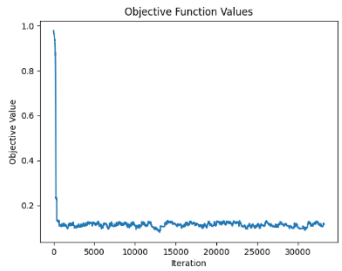
\Schwerin2_BPP78

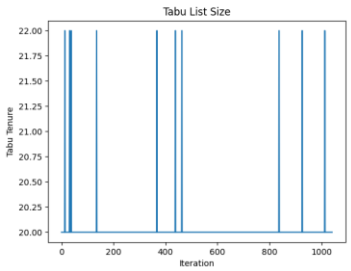


\Waescher_TEST0030

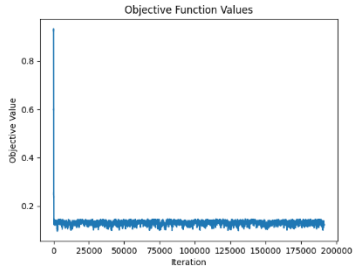


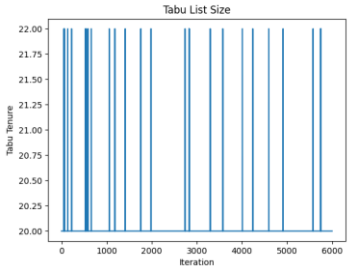
\Waescher_TEST0075





\Waescher_TEST0097





Discussion:

1. Tabu List Size:

A larger tabu list in Tabu Search essentially forces the algorithm to search unvisited or less frequently visited regions of the search space. Since the algorithm is barred from making moves that are on the tabu list, it must look for other, potentially unexplored moves that are not forbidden. This pushes the search into newer areas, which can be beneficial in finding global optima that are hidden away from local optima clusters.

When dealing with more complex optimization problems, it is generally beneficial to favor more exploration. This approach helps to mitigate the risks of prematurely converging on local optima and ensures a more thorough investigation of the solution space, which is particularly important in landscapes that are rugged or have many local optima. For more complex problems, a larger Tabu list is used at the beginning of the search process to aim for more exploration.

2. Convergence Threshold:

In simpler problems, where it is easier to locate and verify the global optimum, we might choose a larger threshold for minimal improvements, thereby stopping the search earlier and avoiding unnecessary computations. Also in these problems, we can afford more aggressive stopping criteria because the risk of missing out on significantly better solutions in unexplored areas is lower. This might include fewer iterations without improvement or a smaller change in solution values between iterations before stopping.

3. Adaptive Tweak Selection:

Early Stages: Merging Bins

- More Empty Space: In the initial stages of a bin packing algorithm, we often start with many bins, some of which are sparsely filled.

- Merging Bins: It is easier to find and merge bins early on because there are more candidates for merging, and the flexibility in space allows for straightforward combinations without violating bin capacity constraints. Merging bins can significantly reduce the total number of bins used, helping to quickly approach a more optimal solution.

Later Stages: Refining Bin Content

- Near Optimal Number of Bins: As the algorithm progresses and the total number of bins approaches the optimum (or a near-optimal solution), the focus shifts from merely reducing the number of bins to optimizing how items are packed within them.

- Swap Tweak: Utilizing a swap-based approach (swapping items between bins) can help refine the arrangement by improving space utilization and potentially allowing further bin reductions. Swaps can fine-tune the packing configuration without necessarily increasing the bin count, making it a valuable tactic in the final stages of optimization.

4. Generating More Neighborhood Solutions:

1. Increased Exploration: By generating a larger number of neighboring solutions (tweaks) at each step, the algorithm effectively increases its ability to explore the solution space. This broad exploration is useful for discovering various feasible solutions that might not be immediately apparent from a more narrowly focused search.

2. Escaping Local Optima: With more neighbors considered, the probability of finding a pathway out of local optima increases. This is because there are more opportunities to find a move that leads to a better solution, potentially bypassing the local traps that hinder finding the global optimum.

3. Enhanced Diversification: Generating more neighbor solutions contributes to the diversification aspect of Tabu Search. It prevents the search from getting too focused on a small region of the solution space, thereby promoting a more thorough investigation of unexplored areas.

- Computational Cost: More neighbors mean more solutions to evaluate, which can significantly increase computational time and resource usage. This factor needs to be balanced based on the available computational resources and the problem's complexity.

- Exploitation Needs: At certain stages of the search, particularly as the algorithm converges towards promising regions of the solution space, it may be beneficial to reduce the number of neighbors considered to focus more on exploiting these areas. This involves making smaller, more precise tweaks to fine-tune the solutions.

- Start with Wide Exploration: Initially, use a larger set of neighbors to ensure broad coverage of the solution space, which is especially useful for complex or large-scale optimization problems.

- Narrow Down Gradually: As the search progresses and certain promising regions are identified, narrow down the focus by reducing the number of neighbors. This shift helps in exploiting these regions to extract finely tuned solutions.