COS 314 Assignment 1

Tayla Orsmond u21467456

Report on the effectiveness of Iterative Local Search and Tabu Search for the One-Dimensional Bin Packing Problem (1BPP)

# Introduction

The one-dimensional bin packing problem (1BPP) involves packing items of variable size into bins of a fixed capacity. There are many variations of bin-packing, but in this instance the items and bins are one-dimensional, so there is no need to consider the item shape or different stacking configurations within a bin. The goal of the problem is to use the fewest bins possible.

This report considers two algorithms, Iterative Local Search (ILS) and Tabu Search (TS), in their effectiveness in solving 1BPP for various datasets. The algorithms each make use of specific heuristics which are discussed below.

# Heuristics

The initial consideration of this problem involved reflecting on how a human would solve the problem given a set of physical bins and items. The following method resulted:

1. Order items into descending order
2. Pack the largest items into the bins first, attempting to fit as many items in the bins as possible wherever they will fit best (before adding a bin).
3. After all the items are packed, take the least-filled bin, and attempt to repack the (hopefully smallest) items in that bin into other bins.

After some research, it was clear that the steps above correspond to methods and heuristics that have been well-researched and developed by many others [1][2]. Thus, with these steps as a starting point, several heuristics were considered and selected to solve this problem.

## Best-Fit (BF) Heuristic:

This constructive heuristic corresponds to step 1 and 2 above and is generally considered very strong in this type of problem [1]. Every item is packed into the bin that would best fit that item, i.e., the bin that would result in the least space left after inserting this item. If no bin can fit the item, a new bin is added, and the item inserted into that bin [1].

Ordering the items in descending order is not guaranteed to produce an optimal solution using this BF heuristic (known as BF-Descending) [1]. However, it was decided that the data would be ordered in descending order to hopefully improve the performance of other heuristics in the algorithm.

## Attempt to Empty a Bin:

This heuristic involves selecting a bin and attempting to repack all the items in that bin into the other bins, with the goal of decreasing the number of bins used [2]. This corresponds to step 3 above.

## Attempt to Swap Two Items:

This heuristic takes one random item from two different bins and attempts to swap them if possible and if the latter bin is better filled, i.e., if the item from the first bin is larger than the item from the second bin but can still fit in the second bin [2]. This is done in addition to the three steps above and assists when the other heuristics fail to produce a better solution.

This is designed to move smaller items out of bins and into a selected bin in the hopes that repacking the items again from a bin will result in a better fit in all bins.

## Bin Selection:

This heuristic is combined with the two above to guide the algorithm.

A selected bin can be:

* **The least-filled bin** (i.e., the bin with the least number of items): This works best when attempting to empty a bin, as the fewer the number of items in the bin, the more likely the bin can be emptied entirely.
* **A random bin in the bottom-half of the dataset**: Because the data is sorted in descending order, it may be likely that the smaller items will end up in the bottom half of the bins, thus, it is likely that these smaller items will fit in other bins, even if the selected bin cannot be emptied entirely.
* **A random bin**: Select a random bin in order diversify the search space. There may not be a least-filled bin to select from (in which case it’s important the same bin is not continuously selected).

# The heuristic is intended to exploit the tendency of least-filled bins to contain smaller items that can be easily emptied, while maintaining diversity in the search space. At first, the algorithm only focused on the least-filled bin. However, this approach was modified to include random bins when it was found that the algorithm could not always identify a least-filled bin, and would repeatedly select the first bin.

# Environment (Specs)

This assignment was completed on a Dell Inspiron 7490 laptop with the following specs:

**Processor**: Intel(R) Core (TM) i7-10510U CPU @ 1.80GHz 2.30 GHz

**RAM**: 16.0 GB (15.8 GB usable)

**Development Environment**: Visual Studio Code, JDK version 17.0.6

# Iterated Local Search

The Iterated Local Search (ILS) algorithm begins by constructing an initial set of bins using the BF-Descending heuristic and sets this as its initial best solution. It then iterates through two perturbation methods for the remainder of the algorithm, generating, searching, and evaluating neighbours until it arrives at a termination condition (discussed later).

At each step, if a change is made that either results in a fewer number of bins, or the better packing of a bin, the algorithm updates its best solution to reflect this change and sets the “repeatOverall” variable to be true, indicating that a move towards an optimal solution has been made and the algorithm should iterate again.

The first heuristic that is used is the Bin Selection heuristic. A random number is rolled [P1] to pick whether the bin to be used for the next two heuristics should be the least-filled bin or a random bin. If the random number is above a certain threshold, it picks the least-filled bin. The threshold is skewed to favour picking the least-filled bin. If it is not, the algorithm decides whether to be biased towards the bottom half of the bins or not by rolling another random number [P2].

From there, the ILS algorithm swaps between two perturbation methods in a loop, changing to the next method when it can no longer improve the solution using the first.

The first heuristic used for perturbation is Attempt to Empty a Bin. The algorithm perturbs by attempting to repack the items from the selected bin into the other bins. It then locally searches the new solution to determine whether the new solution is better than the current best solution – whether there are a fewer number of bins or whether the selected bin is emptier. It repeats this process if an improvement is made (Greedy Hill Climbing).

The second heuristic used is Attempt to Swap Two Items. A neighbour solution is created by attempting to swap an item from the selected bin with one from a random bin if possible and if the random bin is better filled after the swap. The new solution is then locally searched as before, and this process repeats if a swap results in a better fit for the random bin (Greedy Hill Climbing).

Once it can no longer perturb using the second heuristic, the algorithm will repeat with a different bin (using the Bin Selection heuristic again) if an improvement was made using either heuristic.

In order that the algorithm does not get stuck in local optimum, a random number is rolled [P3] to force the algorithm to continue and possibly accept a worse solution. If the random number is above a certain threshold, the algorithm iterates again. If it is not, the algorithm accepts the current solution as the best solution and terminates.

# Tabu Search

The Tabu Search (TS) algorithm makes use of the same heuristics in the same order as ILS in order that the algorithms be comparable. It begins by constructing an initial best solution using the BF-Descending heuristic, then begins with the Bin Selection, then moving on to the Attempt to Empty a Bin and Attempt to Swap Two Items heuristics.

The difference between the algorithms is that TS does not perform a local search. Instead, TS considers if the perturbed solution is already in its list of previously visited solutions and is thus “Taboo” and cannot be visited again, in which case the algorithm will not consider the solution and instead move to the next perturbation.

If the solution is not in the Taboo list, then the algorithm considers the solution and updates its current best solution if the new solution is better. The Taboo list is finite, controlled by the “tabuListLength” [P4] parameter. If the Taboo list length becomes too large, the first item from the list is removed as this is the least likely to be encountered again and likely the least optimal solution in the list.

Once it can no longer produce a better solution that is not in the Taboo list, the algorithm terminates. Like ILS, the algorithm is prevented from getting stuck in a local optimum by rolling a random number [P3]. If the number is lower than a certain threshold, the algorithm accepts the current solution as the best solution and terminates.

# Parameters

Each algorithm was given the exact same parameters in order that they remain comparable. These parameters needed to be fine-tuned throughout the process to improve each algorithm’s performance, especially because these algorithms rely heavily on stochastic methods.

This was done offline, by starting with initial values for each parameter and slowly adjusting the parameters incrementally until there was no longer an improvement in the algorithm. The values were both increased and decreased to attempt to improve performance. “Improvement” was considered as an average enhancement in both algorithms across all the datasets over multiple runs.

Below is a list of parameters and their initial and final values after fine-tuning:

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Initial Value** | **Final Value** |
| Capacity of the bins | Second line of the PI text file | Second line of the PI text file |
| [P1] LF Bin Selection Threshold | > 0.4 | > 0.4 |
| [P2] Lower-half bias | < 0.6 | < 0.6 |
| [P3] Forced Continue Threshold | > 0.4 | > 0.1 |
| [P4] tabuListLength | The number of items in the file | The number of items in the file |
| **Time (ILS & TS respective)** | **~5s and ~4s** | **~8s and ~4s** |

The algorithms’ performance changed slightly by tweaking these values. Changing [P2] and [P1] had no effect, and thus simply allowing the algorithms to run for longer, [P3], was enough to improve performance. The improvement slowed between threshold > 0.2 and > 0.1. Although there was an improvement in the number of optimal solutions produced in each dataset for both algorithms, the biggest change was in the execution time of ILS, which increased significantly, whereas TS stayed roughly the same.

# Results

**The tables here show a run that represents the rough average across runs.**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **ILS** | | | **TS** | | |
| **Dataset** | **Opt** | **Opt-1** | **Sum** | **Opt** | **Opt-1** | **Sum** |
| Falkenauer\_T | 0 | 0 | 80 | 0 | 0 | 80 |
| Falkenauer\_U | 8 | 24 | 80 | 6 | 26 | 80 |
| Hard28 | 7 | 20 | 28 | 5 | 23 | 28 |
| Scholl\_1 | 550 | 111 | 720 | 547 | 114 | 720 |
| Scholl\_2 | 245 | 123 | 480 | 236 | 127 | 480 |
| Scholl\_3 | 0 | 0 | 10 | 0 | 0 | 10 |
| Schwerin\_1 | 1 | 95 | 100 | 0 | 100 | 100 |
| Schwerin\_2 | 3 | 92 | 100 | 0 | 93 | 100 |
| Waescher | 12 | 5 | 17 | 2 | 15 | 17 |

***Table 1****: Table showing the number of problem instances, for each of the data set categories, which were solved to optimality or near-optimality (one bin from the optimum), for ILS and Tabu Search* ***after*** *the stochastic parameters were fine-tuned.*

|  |  |  |
| --- | --- | --- |
| **Dataset** | **ILS** | **TS** |
| Falkenauer\_T | 0.09 | 0.003 |
| Falkenauer\_U | 0.011 | 0.010 |
| Hard28 | 0.003 | 0.001 |
| Scholl\_1 | 0.002 | 0.002 |
| Scholl\_2 | 0.002 | 0.001 |
| Scholl\_3 | 0.158 | 0.005 |
| Schwerin\_1 | 0.000 | 0.000 |
| Schwerin\_2 | 0.001 | 0.000 |
| Waescher | 0.003 | 0.000 |
| **Total** | **0.27** | **0.022** |

***Total Time for ILS: 6.717s Total Time for Tabu: 3.576s Table 2****: Table showing the average runtimes (in seconds) averaged across each dataset as well as all problem instances for ILS and Tabu Search* ***after*** *the stochastic parameters were fine-tuned.*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **ILS** | | | **TS** | | |
| **Dataset** | **Opt** | **Opt-1** | **Sum** | **Opt** | **Opt-1** | **Sum** |
| Falkenauer\_T | 0 | 0 | 80 | 0 | 0 | 80 |
| Falkenauer\_U | 7 | 25 | 80 | 6 | 26 | 80 |
| Hard28 | 6 | 21 | 28 | 5 | 23 | 28 |
| Scholl\_1 | 549 | 112 | 720 | 547 | 114 | 720 |
| Scholl\_2 | 240 | 128 | 480 | 236 | 127 | 480 |
| Scholl\_3 | 0 | 0 | 10 | 0 | 0 | 10 |
| Schwerin\_1 | 0 | 100 | 100 | 0 | 100 | 100 |
| Schwerin\_2 | 3 | 91 | 100 | 0 | 93 | 100 |
| Waescher | 4 | 13 | 17 | 2 | 15 | 17 |

***Table 3****: The average performance of each algorithm* ***before*** *the stochastic parameters were fine-tuned.*

In analysing the results, it’s clear that each algorithm has different strengths and weaknesses that should be considered when selecting which algorithm to use in a particular scenario.

Overall, as seen in the “Opt” columns of Table 1 above, ILS attains a slightly greater number of optimal solutions than TS in most instances over the datasets. Particularly in difficult datasets such as Schwerin\_2 or Waescher, ILS may sometimes achieve optimal solutions where TS fails.

As seen in the “Opt-1” columns of Table 1, ILS and TS achieve a similar number of near-optimal solutions for most datasets, whereby TS achieves slightly more, since ILS produces greater number of optima (since they both produce near identical numbers of sub-optimal solutions).

Thus, ILS is more suited to scenarios where accuracy is considered a high priority and sub-optimal solutions should be minimised. ILS may perform better in situations of greater difficulty, where the algorithms are able to run for longer. This can be seen by comparing Table 1 and Table 3, where the accuracy of the solutions produced by ILS slightly increased when the algorithm performed a greater number of iterations.

However, as can be seen in Table 2, the average execution time for each algorithm differs significantly. In this regard, TS outperforms ILS in near every dataset and overall, the algorithm is faster.

In the Parameters section above, it can be seen from the table that TS’s execution time, on average, remained significantly lower than ILS including when the number of iterations increased. This means that TS may be more scalable, however this cannot be certain due to the stochastic nature of the termination condition.

Thus, TS may be better suited in situations where speed is more important than optimal solutions, as TS achieves a similar number of near optimal solutions as ILS, and so may perform just as well at a greater speed than ILS.

# Conclusion

In general, Iterative Local Search (ILS) achieves a greater number of optimal solutions per dataset than Tabu Search (TS), but TS is significantly faster. Both algorithms have unique strengths and weaknesses that should be considered when selecting an algorithm to use for a particular situation. In situations where the problem requires more accuracy, ILS outperforms TS. However, in situations where speed is a higher priority than achieving (near-)optima, TS is the better choice of algorithm.

# References

1. Munien, C. and Ezugwu, A. (2021) Metaheuristic algorithms for one-dimensional bin-packing problems: A survey of recent advances and applications. *Journal of Intelligent Systems*, Vol. 30 (Issue 1), pp. 636-663. Available at: <https://doi.org/10.1515/jisys-2020-0117> (Accessed: March 20, 2023).
2. Hyde, M. *et al.* (2011) *A HyFlex Module for the One-Dimensional Bin Packing Problem*. rep. Nottingham, UK: School of Computer Science, University of Nottingham, Jubilee Campus, pp. 1–5. Available at: <https://www.inf.ufpr.br/aurora/disciplinas/topicosia2/problemas/BinPackingHyFlex.pdf> (Accessed: March 20, 2023).