COMPARISON OF NEIGHBORHOOD STRUCTURES FOR THE RECTANGULAR TWO-DIMENSIONAL STRIP PACKING PROBLEM

Júlio-César-da-Silva-de-Oliveira[1] and Alexandre-Checoli-Choueiri[2]

[1]R. Elétrica - Jardim das Américas, Curitiba - PR, 82590-300
[2]R. Elétrica - Jardim das Américas, Curitiba - PR, 82590-300
E-mails: juliooliveira@ufpr.com.br / alexandrechecoli@ufpr.br

ABSTRACT. This paper addresses the Two-Dimensional Strip Packing Problem (2D-SPP) under specific constraints, employing local search techniques to explore three distinct neighborhood structures: **Switch**, **Switch/Swap**, and **Shift**. Extensive testing was conducted to evaluate the performance of these neighborhood structures across various instance sizes. Results were analyzed using tables, pie charts, and graphs to provide a comprehensive comparison. The findings indicate that the **Switch** structure consistently delivered the least favorable outcomes. In contrast, the **Switch/Swap** neighborhood excelled in medium-sized instances, while the **Shift** structure demonstrated superior performance for larger instances, highlighting its effectiveness in handling more complex scenarios.

Keywords: Strip-Packing Problem, Local Search, Heuristics, Two-dimensional cutting problems

1 Introduction

Efficient material utilization is crucial in manufacturing industries because every manufacturing process generates some amount of waste, as no transformation process can ever be 100 percent efficient. Therefore, it is in the best interest of manufacturing companies to minimize this waste whenever possible in order to maintain market competitiveness through lower production costs. Notable examples of waste generation in manufacturing settings include cutting situations, where material must be removed, reshaped and reprocessed to produce specific desired products as mentioned by Vanessa M. R. Bezerra and Santos (2020).

Cutting and packing problems are prevalent across various sectors of the economy, each with unique constraints and objectives, specific to its practical applications. Cutting problems commonly arise in industrial settings where minimizing waste is essential, such as when cutting raw materials like paper, adhesives, metal, glass, wood, or textiles, as corroborated by Verstichel et al. (2013). An optimized cutting strategy is vital in these industries because a significant portion of manufacturing costs comes from purchasing raw materials. Therefore, reducing waste during production can significantly increase profitability, especially when transforming materials that are expensive or have very costly reworking processes, like most metal alloys.

Metals, although capable of being cut directly into complex shapes, are more commonly initially cut into rectangles due to the ease of mass cutting, storage, and transportation that the shape provides. In the industrial context, optimization of any of these segments of the process is possible and important, but in terms of material waste reduction, the cutting problem is the most relevant. In most metal industries, the alloy is heat-treated in coil form, which is an efficient shape for oven insertion and heat control. Paper, adhesives, and textile feed-stock are also often stored in coils in an effort to streamline the transportation of large quantities of material. In Verstichel et al. (2013) the practical example is mainly about the paper sector but the restrictions are very similar.

As coils most commonly describe large quantities of materials, they can be considered as rectangles with "infinite" length as it takes numerous production orders to completely use up a coil. Therefore, the most sensible approach to this problem is to treat each material coil as a rectangle with fixed width and infinite height. Waste reduction would come from cutting the coil with minimal gaps between cuts (meaning they are as closely packed together as possible) and achieving the shortest possible maximum height that still accommodates all necessary cuts, as better illustrated in Oliveira et al. (2016).

This approach ties into the concept known in the literature as the strip packing problem. In this context, the "strip" represents the coil with "infinite" height, and the rectangles to be cut are the items to be packed into the strip. Modeling of cutting and packing problems are usually very similar and that is why most research approaches both simultaneously, as described by Wei et al. (2011).

In this paper, we address the strip packing problem using a local search approach, comparing the outcomes of three distinct neighborhood structures across various instance sizes.

This work is organized into five sections. Following the Introduction, Section 2 provides a review of current literature on cutting and packing problems. Section 3 discusses the concept of local search algorithms, outlining the approach taken to develop the proposed solution. Section 4 presents the results obtained from the implementation of the developed solution. Section 5 summarizes the findings and offers concluding remarks. Additionally, an Appendix is provided at the end, containing relevant supplementary results.

2 Literature Review

General cutting and packing problems are further differentiated by the practical constraints that are present for each specific application. In the textile, paper and metal industries the feed-stock are usually coils of material, and for this situation, the best fit would be to treat it as a Strip Packing Problem (SPP).

In this problem, there is a known width but infinite height for the rectangle that is meant to be cut (hence the strip), minimizing this infinite height when fitting all the items to be cut while also minimizing the gaps between them is the goal of strip packing algorithms.

The (SPP) can be categorized in varied ways based on fundamental cutting and packing constraints, as highlighted by Oliveira et al. (2016). These constraints include the input of the list of

items (online, meaning it is only known the information about the current item such as in scheduling problems; offline, meaning all rectangle information is known at all times), dimensions (1D, 2D, or 3D), shape, orthogonality (whether orthogonal or non-orthogonal), the allowance of rotation, and the type of cuts/packing (guillotinable or non-guillotinable). For two-dimensional problems, typical classifications include rectangular, circular/spherical, and irregular shapes.

There are several ways to go about solving the SPP, for example exact methods, that involves modeling the problem as an IP and using an enumerative method to solve it, like Branch and Bound. Although it provides the optimal solution, instances with more than 30 items becomes time-intractable, such as studied by Lesh et al. (2004), due to the combinatorial explosion of constraints. An alternative to exact-methods are then heuristics, that are able to find good quality solutions in reasonable computational times.

Ntene and van Vuuren (2009) propose a new level heuristic, after having done a deep exploration into the already known level algorithms. The authors developed a new method which they named as Size Alternating Stacks (SAS) and it is comprised of alternating between packing wide and narrow rectangles in an effort to leave less gaps leftover in each level. The skyline algorithm presented by Wei et al. (2011) is another example of a heuristic method, where the concept of a skyline is introduced in an attempt to better model and minimize the wasted space in the strip. A third example is the fast layer-based heuristic of Leung and Zhang (2011), the method improves upon the layer-based heuristic first proposed by Zhang et al. (2009) by ordering the rectangles by perimeter before placing them.

Another option for solving the 2D-SPP would be meta-heuristic algorithms. Meta-heuristics are heuristics in the sense that they do not guarantee optimal, but they are general algorithmic frameworks, capable of solving any combinatorial optimization problem. Hopper and Turton (2001) investigated the use of those algorithms in solving 2D packing problems, comparing the use of genetic algorithms (GA), simulated annealing (SA) and naive evolution (NE).

As a final approach in tackling the problem, a Hybrid among a all the others is possible. This hybridization of algorithms is explored and bench marked by Hopper and Turton (2001), Neuenfeldt Júnior et al. (2022) in a recent meta-study. The authors conclude the paper with a comparative table showing most solving methods used in recent researches: many of them choose the hybrid approach. Bench marks between meta-heuristic, heuristic and hybrid methods are presented in Riff et al. (2009), and Vanessa M. R. Bezerra and Santos (2020) makes a computational evaluation between the different methods.

In any meta-heuristic approach it is important to generate a quick initial solution. In the context of strip packing the Next-Fit Decreasing Height (NFDH) algorithm first proposed by Coffman (1980) and better explained and benchmarked by Ntene and van Vuuren (2009) can be utilized as a greedy first solution. This algorithmic rule was later evaluated as a method by other research papers such as Lodi et al. (2002) and Riff et al. (2009), and it is a common alternative for a first solution given that it is relatively simple to implement and does not demand much time or computation power to process.

3 Development

This paper focuses on strip packing problems characterized by the following characteristics:

1. Offline
2. Two dimensions (2D)
3. Only rectangular items
4. Orthogonal placement
5. Rotation able
6. Guillotinable cuts

The approach was made through a Hybrid model that uses the already presented Next-Fit Decreasing Height (NFDH) algorithm as a greedy solution and then improves upon it by using a local search algorithm with 3 different neighborhood structures to obtain contrasting results. All solutions are expressed by a list of rectangles that is then decoded by the NFDH algorithm to obtain a meaningful strip height. The utilization of an indirect representation of the solution with a simple list affords massive flexibility in the development of different neighborhoods and also very quick computation of results to iterate trough with the local search.

3.1 Local Search

Local search algorithms are optimization techniques that start with an initial solution and explore neighboring solutions in order to improve results. They work by iteratively adjusting the solution within its neighborhood, focusing on incremental changes that may improve the defined objective function, rather than exploring the entire solution space for the best possible alternative (a Global Optimum). This characteristic makes them particularly suitable for high-dimensional or complex problems, where evaluating every possible solution is impractical, such as the 2D-SPP.

The goal of local search algorithms, which are a type of meta-heuristic, is to find a solution that is not necessarily the best possible but is at least better than all its immediate neighbors (a Local Optimum). A limitation of local search algorithms is their tendency to converge on local optima, because they only explore the immediate neighborhood, they risk "getting stuck" in these sub-optimal points, unable to reach a potentially better solution elsewhere in the solution space. To address this, local search methods may often incorporate additional strategies, like perturbation or probabilistic steps, to escape local optima.

Algorithm 1 is a general template for local search algorithms:

3.2 The Next Fit Decreasing Height (NFDH) Decoder

The Next Fit Decreasing Height (NFDH) algorithm is a heuristic specific for two-dimensional bin or strip packing problems. The algorithm first sorts items by height in descending order, packing larger items before smaller ones to maximize space use and minimize big height gaps that would result in a taller overall cutting pattern, therefore wasting more material.

In NFDH, items are placed sequentially within a row until no more fit on the width, then, a new

Algorithm 1: *Generic Local Search*

```
1  s = InitialSolution()                              ▷ Generate Initial Solution
2  stop = False
3  while stop == false do
4      N = GenerateNeighbors(ℕ(s))
5      for n ∈ {1,...|N|} do
6          s′ = s_n                                   ▷ Select a Neighbor from s
7          if s′ is better than s then
8              s = s′                                 ▷ Replace s with s′
9              exit for
10         else
11             if n == |N| then
12                 stop == true                       ▷ Local optimum
13 return s
```

row is started directly above it. Each row's height is set to the tallest item in that row, maintaining compact packing with minimal wasted vertical space. This "next-fit" approach iterates until all items on the given list are packed.

Algorithm 2 is a Pseudo-code for the NFDH algorithm, where L is the list of items to be packed (each item has a *width* and *height*) and W the width of the strip:

Algorithm 2: *NFDH(L,W)*

```
1  foreach rectangle i ∈ L do
2      if i.width < i.height then
3          RotateRectangle (i)        ▷ Make width greater than height for every rectangle
4  SortDecreasingHeight(L)                            ▷ Sort list by decreasing height
5  while L ≠ ∅ do
6      width_remaining = W
7      foreach i ∈ L do
8          if i.width ≤ width_remaining then
9              width_remaining -= i.width
10             InsertInRow(i):                        ▷ Place rectangle in current row
11             Remove(i,L)                            ▷ Remove from list
12     RowHeigh = Max(Height)                         ▷ Set tallest from row as row height
13     TotalHeight = RowHeight + TotalHeight          ▷ Update total height
14     StartRow()                                     ▷ Move to a new row
15 return s                                           ▷ Return the packed solution
```

While simple and effective, NFDH may leave gaps between rows, as it does not rearrange items once placed, that will be the job of the meta-heuristic. For the purpose of this article it functions

as a very quick first solution and allows the flexibility to change the solution structure by simply rotating rectangles or rearranging the order of the list of items to be packed (when excluding the NFDH part that sorts items by height).

3.3 Initial Solution Generation

Initial solution generation is a critical step in optimization algorithms, and is specified in Algorithm 1 line 1 as the first part of local search. The quality of the starting solution can significantly impact both the efficiency and the effectiveness of the algorithm. A well-chosen initial solution provides a good starting point for the search process, allowing the algorithm to converge to high-quality results more quickly. Conversely, a poor initial solution may lead to longer convergence times or sub-optimal outcomes.

Several strategies can be employed to generate initial solutions:

- Random Initialization: Items are placed randomly within the strip, ensuring feasibility but with no regard for optimization. This method is quick and simple but often results in sub-optimal starting solutions.
- Greedy Heuristics: Items are sorted based on certain criteria, such as decreasing height or width, and then placed in sequence. This method leverages problem-specific knowledge to create a structured starting solution.

The choice of strategy depends on the problem requirements and computational constraints. For example, in high-dimensional or complex problems, a simple greedy heuristic may provide a good trade-off between computational cost and solution quality. On the other hand, more sophisticated methods, such as hybrid approaches combining randomness and heuristics, may yield better results for problems where initial solution quality heavily influences the overall performance.

In the case of this article, all neighborhoods utilize the unsorted NFDH Algorithm as a greedy heuristic, as they only try to improve upon the solution by trying to fill in gaps initially generated.

3.4 Neighborhoods

In this section, the 3 different neighborhood structures are explained. They differ in movement operator, but all output a modified list of rectangles to be packed by the NFDH (skipping sorting and initial rotations, only using the while in line 5 of Algorithm 2). The Max Iteration in line 2 of Algorithm 1 is controlled by increasing an integer every time a better solution is not found after making a change and resetting the integer to 0 every time the solution improves, setting, then, a number of iterations without improvement as the stoppage condition.

3.4.1 Switch neighborhood

The movement operator to change the ordered list in this case is the "switch", a rotation of any rectangle on the list is a switch, it was implemented in such a way so that a random item is rotated.

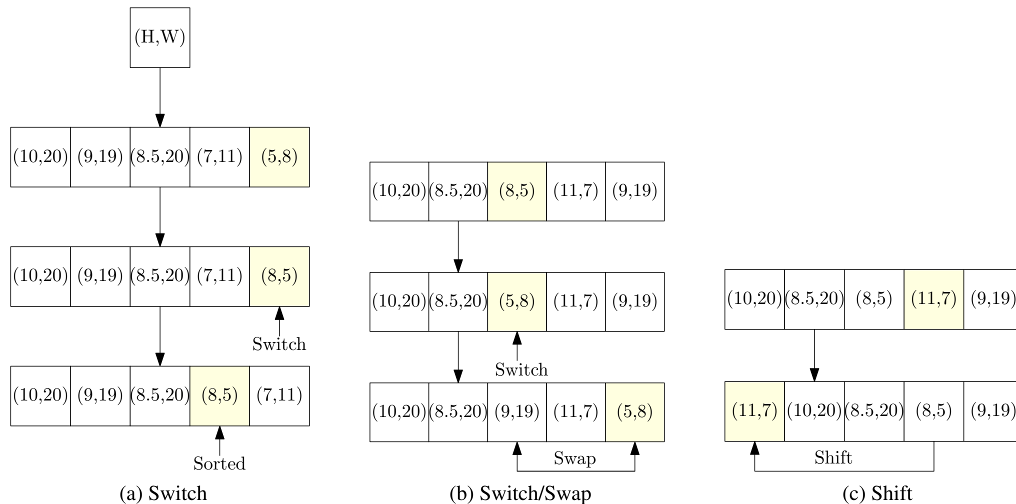
An example of one movement of this neighborhood is depicted in Figure 1 a. The list of items are represented with their height and width as (H,W). In that example, a switch is performed on the last item, and re-sorting the list positions it on the 3rd index (start at 0) instead of the last.

3.4.2 Switch/Swap neighborhood

The second neighborhood does not sort the list at any time. It simply calls functions that "switch" and "swaps" items, the switch is still a rotation but the swap is taking another random rectangle and changing its place on the list with the rotated one (hence, sorting would not make sense). There is more possibility for varied results based on the random item chosen each time the program is ran. An example is depicted in Figure 1 b. In it, a switch is performed on the item at index 2 and it is then swapped positions with the last item of list.

3.4.3 Shift neighborhood

The final neighborhood is based on the concept of "shifting" the list. Shifting is defined as randomly picking an item to be replaced to an also random location on the list, the difference from swapping is that the placement of the item makes an entire section of the list either move one position backwards or forwards. An example is depicted in Figure 1 c. In which a shift occurs on the item at index 3, it is then moved to the first position.



(a) Switch (b) Switch/Swap (c) Shift

Figure 1: Neighborhoods

3.5 Expressing results

To facilitate the analysis and debugging of code and results during development, a function was created that takes any solution represented as an array and generates a corresponding visual

representation in the form of a .png file. For example, the .png output files for the T5c instance are provided in the appendix, where the captions specify the code that was executed to generate each image.

4 Results

In this section we describe the results from testing the neighborhoods by applying the algorithms in the instances from Hopper and Turton (2001) found directly at: https://people.brunel.ac.uk/ mastjjb/jeb/orlib/files/strip3.xls. They generated 14 sets of guillotineable problems (from T1 to T7, and N1 to N7) each set with a fixed number of rectangles (from 17 to 199) and containing 5 instances (a to e) with a known optimum of 200 height and a fixed width of 200.

All local search algorithms were executed with the maximum number of iterations set to 1,000. For each instance, the searches were conducted five independent times, from start to finish, to account for variability in performance due to random factors. Only the best solution among these five runs was recorded as the final result, thereby minimizing the potential impact of "unlucky" random numbers hindering the progress of any single search. The algorithm was implemented using the Python programming language.

Table 1 displays the results for each of the instances described before. The **Size** column indicates the number of items to be packed. In order to see wether the local search were improving the solution, column **Greedy** shows the initial solution objective values. The 3 different local searches values are in columns **Shift**, **Switch/Swap** and **Shift**. Following, the **Gap** columns displays the distance of each solutions from the Optimal, and finally, column **Best algorithm** classify the best performing algorithm among the 3.

The information is better summarized in 2, which displays in a pie chart the amount of times each algorithm provided the best solution for an instance (lowest total height).

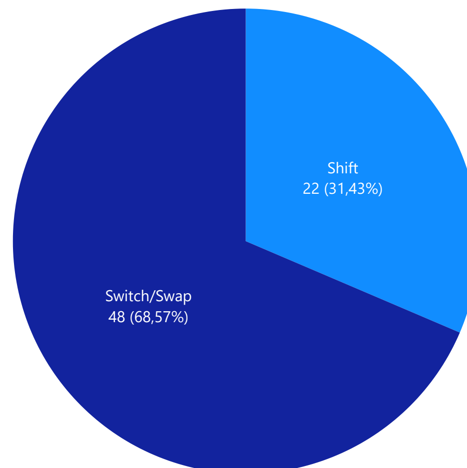
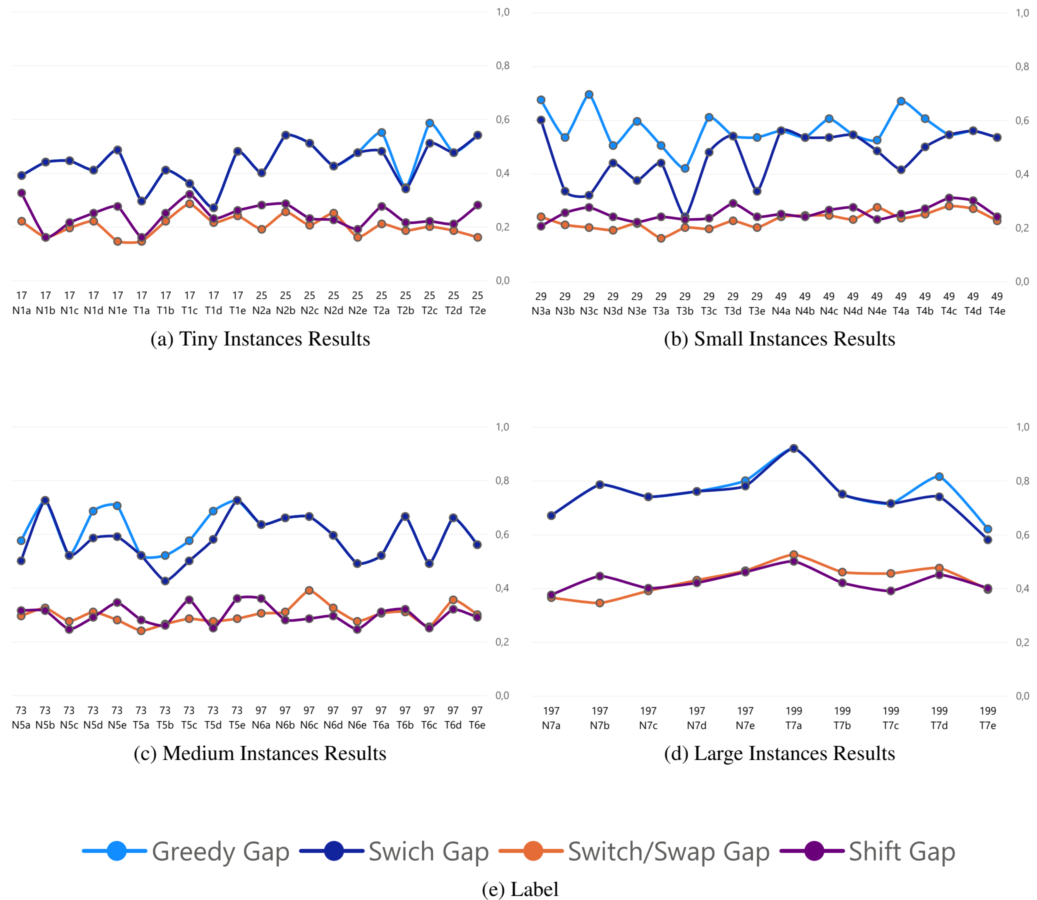


Figure 2: Illustration of Results

The results are presented in greater detail in Figure 3. These figures display the instance sizes (number of rectangles) and their corresponding names on the X-axis. The instances are categorized into tiny, small, medium, and large groups to enable a more refined analysis.

To enhance visibility, only the gaps (expressed as percentages above the optimal, represented in decimal form) are plotted. In these plots, lower points indicate better results, with smaller gaps reflecting closer approximations to the optimal solution.



(a) Tiny Instances Results

(b) Small Instances Results

(c) Medium Instances Results

(d) Large Instances Results

(e) Label

Figure 3: Comparison of Results for Different Instance Sizes

Examining Figure 2 in isolation, one might conclude that the Switch neighborhood structure is the least effective, as it never outperforms either Switch/Swap or Shift. Additionally, Switch/Swap might appear to be the best-performing structure, achieving superior results more than twice as often as Shift.

However, a closer examination of Figures 3a and 3b reveals that Switch/Swap consistently provides the best solutions for smaller instances. In contrast, Figures 3c and 3d indicate that Shift frequently outperforms the other methods for medium and large instances. This suggests that

Switch/Swap is the most suitable option for instances up to size 73, while Shift performs best for larger instances.

Overall, the analysis across all figures confirms the ineffectiveness of the Switch neighborhood structure, regardless of instance size.

5 Conclusion

In this article, we addressed the 2D Strip Packing Problem (2D-SPP), focusing on the constraints outlined at the beginning of Section 3. To solve this, we applied the concept of local search and designed three distinct neighborhood structures: **Switch**, **Switch/Swap**, and **Shift**. A series of tests were conducted to evaluate the performance of each structure, based on the characteristics described in Section 4. The results were presented in the form of a table, a pie chart, and several graphs, categorized by instance size.

From the analysis of the data, we concluded that the **Switch** neighborhood structure yielded the least favorable results. The **Switch/Swap** neighborhood performed better for medium-sized instances, while the **Shift** neighborhood structure proved to be the most effective for larger instances.

References

Hopper, E. and Turton, B. (2001). An empirical investigation of meta-heuristic and heuristic algorithms for a 2d packing problem. *European Journal of Operational Research*, 128(1):34–57.

Lesh, N., Marks, J., McMahon, A., and Mitzenmacher, M. (2004). Exhaustive approaches to 2d rectangular perfect packings. *Information Processing Letters*, 90(1):7–14.

Leung, S. C. and Zhang, D. (2011). A fast layer-based heuristic for non-guillotine strip packing. *Expert Systems with Applications*, 38(10):13032–13042.

Lodi, A., Martello, S., and Monaci, M. (2002). Two-dimensional packing problems: A survey. *European Journal of Operational Research*, 141(2):241–252.

Neuenfeldt Júnior, A., Silva, E., Francescatto, M., Rosa, C. B., and Siluk, J. (2022). The rectangular two-dimensional strip packing problem real-life practical constraints: A bibliometric overview. *Computers & Operations Research*, 137:105521.

Ntene, N. and van Vuuren, J. (2009). A survey and comparison of guillotine heuristics for the 2d oriented offline strip packing problem. *Discrete Optimization*, 6(2):174–188.

Oliveira, J. F., Neuenfeldt Júnior, A., Silva, E., and Carravilla, M. A. (2016). A survey on heuristics for the two-dimensional rectangular strip packing problem. *Pesquisa Operacional*, 36(2):197–226.

Riff, M. C., Bonnaire, X., and Neveu, B. (2009). A revision of recent approaches for two-dimensional strip-packing problems. *Engineering Applications of Artificial Intelligence*, 22(4):823–827.

Vanessa M. R. Bezerra, Aline A. S. Leao, J. F. O. and Santos, M. O. (2020). Models for the two-dimensional level strip packing problem – a review and a computational evaluation. *Journal of the Operational Research Society*, 71(4):606–627.

Verstichel, J., De Causmaecker, P., and Berghe, G. V. (2013). An improved best-fit heuristic for the orthogonal strip packing problem. *International Transactions in Operational Research*, 20(5):711–730.

Wei, L., Oon, W.-C., Zhu, W., and Lim, A. (2011). A skyline heuristic for the 2d rectangular packing and strip packing problems. *European Journal of Operational Research*, 215(2):337–346.

Zhang, D., Han, S., and Ye, W. (2009). A bricklaying heuristic algorithm for the orthogonal rectangular packing problem. *Chinese Journal of Computers*, 31:509–515.

Appendix

Table 1: Results

Instance	Size	Greedy	Switch	Switch/Swap	Shift	Switch Gap	Switch/Swap Gap	Shift Gap	Greedy Gap	Best	Best algorithm
T1a	17	259	259	229	232	0.295	0.145	0.16	0.295	0.145	Switch/Swap
T1b	17	282	282	244	250	0.41	0.22	0.25	0.41	0.22	Switch/Swap
T1c	17	272	272	257	264	0.36	0.285	0.32	0.36	0.285	Switch/Swap
T1d	17	254	254	243	246	0.27	0.215	0.23	0.27	0.215	Switch/Swap
T1e	17	296	296	248	252	0.48	0.24	0.26	0.48	0.24	Switch/Swap
T2a	25	310	296	242	255	0.48	0.21	0.275	0.55	0.21	Switch/Swap
T2b	25	269	268	237	243	0.34	0.185	0.215	0.345	0.185	Switch/Swap
T2c	25	317	302	240	244	0.51	0.2	0.22	0.585	0.2	Switch/Swap
T2d	25	295	295	237	242	0.475	0.185	0.21	0.475	0.185	Switch/Swap
T2e	25	308	308	232	256	0.54	0.16	0.28	0.54	0.16	Switch/Swap
T3a	29	301	288	232	248	0.44	0.16	0.24	0.505	0.16	Switch/Swap
T3b	29	284	248	240	246	0.24	0.2	0.23	0.42	0.2	Switch/Swap
T3c	29	322	296	239	247	0.48	0.195	0.235	0.61	0.195	Switch/Swap
T3d	29	308	308	245	258	0.54	0.225	0.29	0.54	0.225	Switch/Swap
T3e	29	307	267	240	248	0.335	0.2	0.24	0.535	0.2	Switch/Swap
T4a	49	334	283	247	250	0.415	0.235	0.25	0.67	0.235	Switch/Swap
T4b	49	321	300	250	254	0.5	0.25	0.27	0.605	0.25	Switch/Swap
T4c	49	309	309	256	262	0.545	0.28	0.31	0.545	0.28	Switch/Swap
T4d	49	312	312	254	260	0.56	0.27	0.3	0.56	0.27	Switch/Swap
T4e	49	307	307	245	248	0.535	0.225	0.24	0.535	0.225	Switch/Swap
T5a	73	304	304	248	256	0.52	0.24	0.28	0.52	0.24	Switch/Swap
T5b	73	304	285	253	252	0.425	0.265	0.26	0.52	0.26	Shift
T5c	73	315	300	257	271	0.5	0.285	0.355	0.575	0.285	Switch/Swap
T5d	73	337	316	255	250	0.58	0.275	0.25	0.685	0.25	Shift
T5e	73	345	345	257	272	0.725	0.285	0.36	0.725	0.285	Switch/Swap
T6a	97	304	304	261	262	0.52	0.305	0.31	0.52	0.305	Switch/Swap
T6b	97	333	333	262	264	0.665	0.31	0.32	0.665	0.31	Switch/Swap
T6c	97	298	298	251	250	0.49	0.255	0.25	0.49	0.25	Shift
T6d	97	332	332	271	264	0.66	0.355	0.32	0.66	0.32	Shift
T6e	97	312	312	260	258	0.56	0.3	0.29	0.56	0.29	Shift
T7a	199	384	384	305	300	0.92	0.525	0.5	0.92	0.5	Shift
T7b	199	350	350	292	284	0.75	0.46	0.42	0.75	0.42	Shift
T7c	199	343	343	291	278	0.715	0.455	0.39	0.715	0.39	Shift
T7d	199	363	348	295	290	0.74	0.475	0.45	0.815	0.45	Shift
T7e	199	324	316	279	280	0.58	0.395	0.4	0.62	0.395	Switch/Swap
N1a	17	278	278	244	265	0.39	0.22	0.325	0.39	0.22	Switch/Swap
N1b	17	288	288	232	232	0.44	0.16	0.16	0.44	0.16	Switch/Swap
N1c	17	289	289	239	243	0.445	0.195	0.215	0.445	0.195	Switch/Swap
N1d	17	282	282	244	250	0.41	0.22	0.25	0.41	0.22	Switch/Swap
N1e	17	297	297	229	255	0.485	0.145	0.275	0.485	0.145	Switch/Swap
N2a	25	280	280	238	256	0.4	0.19	0.28	0.4	0.19	Switch/Swap
N2b	25	308	308	251	257	0.54	0.255	0.285	0.54	0.255	Switch/Swap
N2c	25	302	302	241	246	0.51	0.205	0.23	0.51	0.205	Switch/Swap
N2d	25	285	285	250	245	0.425	0.25	0.225	0.425	0.225	Shift
N2e	25	295	295	232	238	0.475	0.16	0.19	0.475	0.16	Switch/Swap
N3a	29	335	320	248	241	0.6	0.24	0.205	0.675	0.205	Shift
N3b	29	307	267	242	251	0.335	0.21	0.255	0.535	0.21	Switch/Swap
N3c	29	339	264	240	255	0.32	0.2	0.275	0.695	0.2	Switch/Swap
N3d	29	301	288	238	248	0.44	0.19	0.24	0.505	0.19	Switch/Swap
N3e	29	319	275	243	244	0.375	0.215	0.22	0.595	0.215	Switch/Swap
N4a	49	312	312	248	250	0.56	0.24	0.25	0.56	0.24	Switch/Swap
N4b	49	307	307	249	248	0.535	0.245	0.24	0.535	0.24	Shift
N4c	49	321	307	249	253	0.535	0.245	0.265	0.605	0.245	Switch/Swap
N4d	49	309	309	246	255	0.545	0.23	0.275	0.545	0.23	Switch/Swap
N4e	49	305	297	255	246	0.485	0.275	0.23	0.525	0.23	Shift
N5a	73	315	300	259	263	0.5	0.295	0.315	0.575	0.295	Switch/Swap
N5b	73	345	345	265	263	0.725	0.325	0.315	0.725	0.315	Shift
N5c	73	304	304	255	249	0.52	0.275	0.245	0.52	0.245	Shift
N5d	73	337	317	262	258	0.585	0.31	0.29	0.685	0.29	Shift
N5e	73	341	318	256	269	0.59	0.28	0.345	0.705	0.28	Switch/Swap
N6a	97	327	327	261	272	0.635	0.305	0.36	0.635	0.305	Switch/Swap
N6b	97	332	332	262	256	0.66	0.31	0.28	0.66	0.28	Shift
N6c	97	333	333	278	257	0.665	0.39	0.285	0.665	0.285	Shift
N6d	97	319	319	265	259	0.595	0.325	0.295	0.595	0.295	Shift
N6e	97	298	298	255	249	0.49	0.275	0.245	0.49	0.245	Shift
N7a	197	334	334	273	275	0.67	0.365	0.375	0.67	0.365	Switch/Swap
N7b	197	357	357	269	289	0.785	0.345	0.445	0.785	0.345	Switch/Swap
N7c	197	348	348	278	280	0.74	0.39	0.4	0.74	0.39	Switch/Swap
N7d	197	352	352	286	284	0.76	0.43	0.42	0.76	0.42	Shift
N7e	197	360	356	293	292	0.78	0.465	0.46	0.8	0.46	Shift

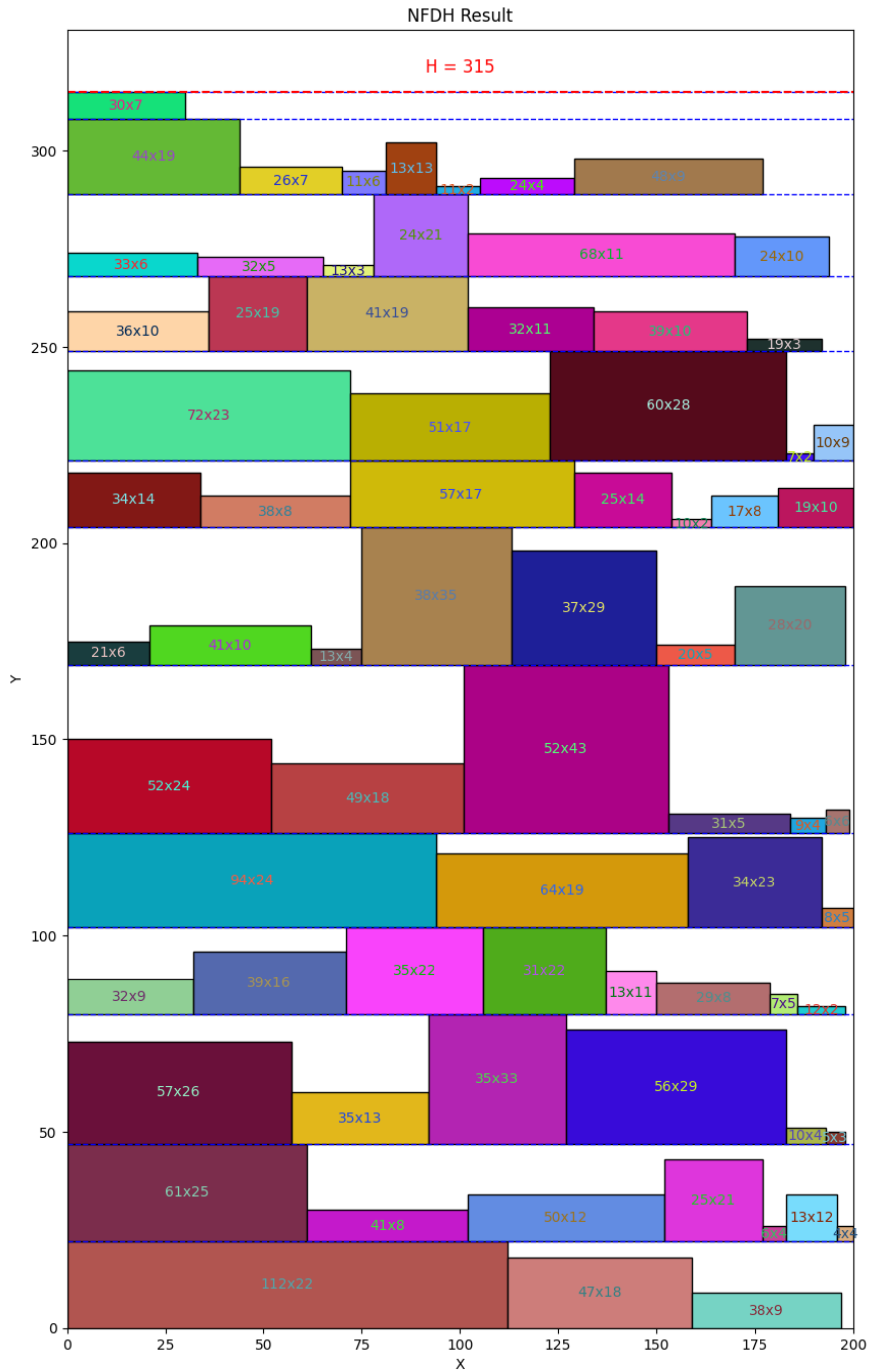


Figure 4: Greedy Algorithm Result

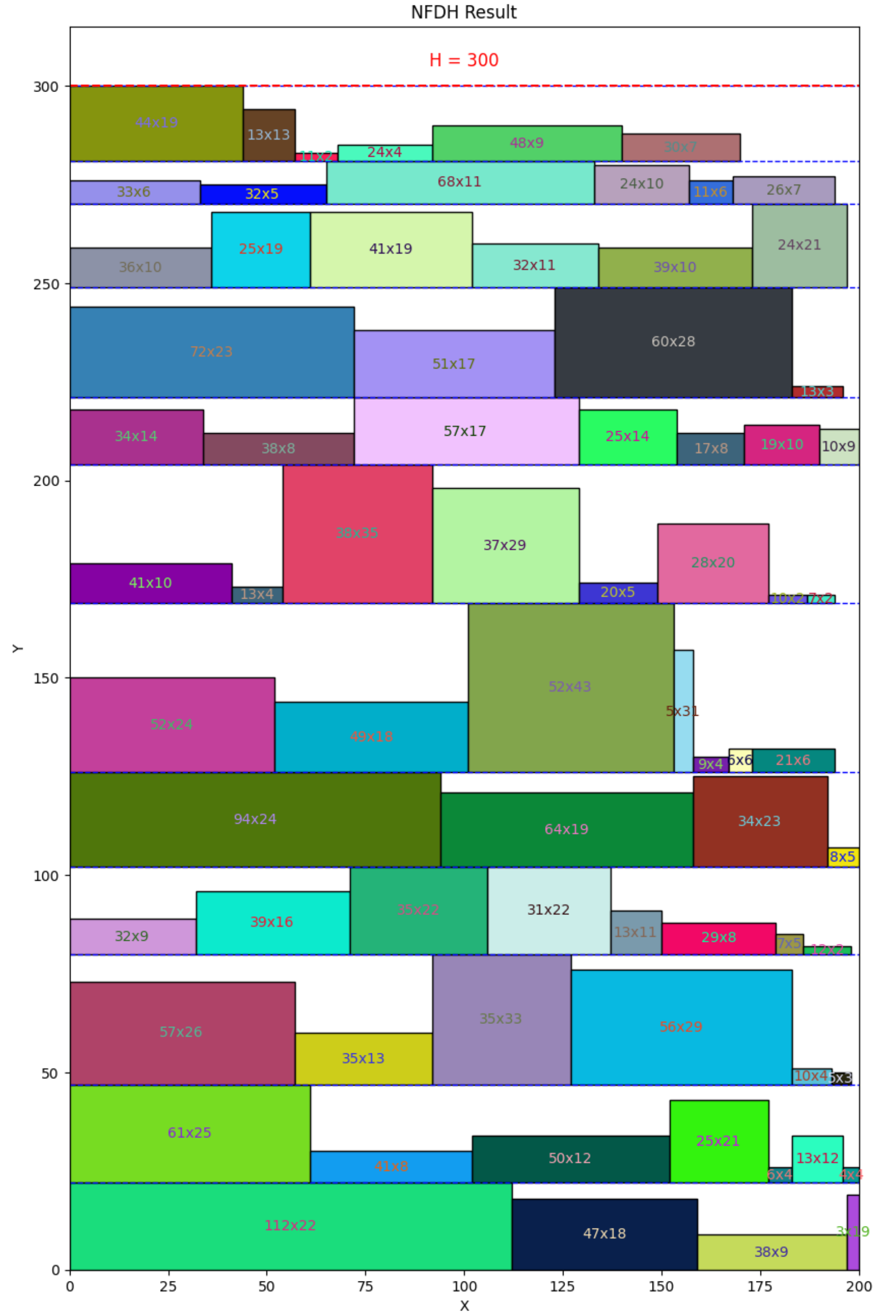


Figure 5: Switch Algorithm Result

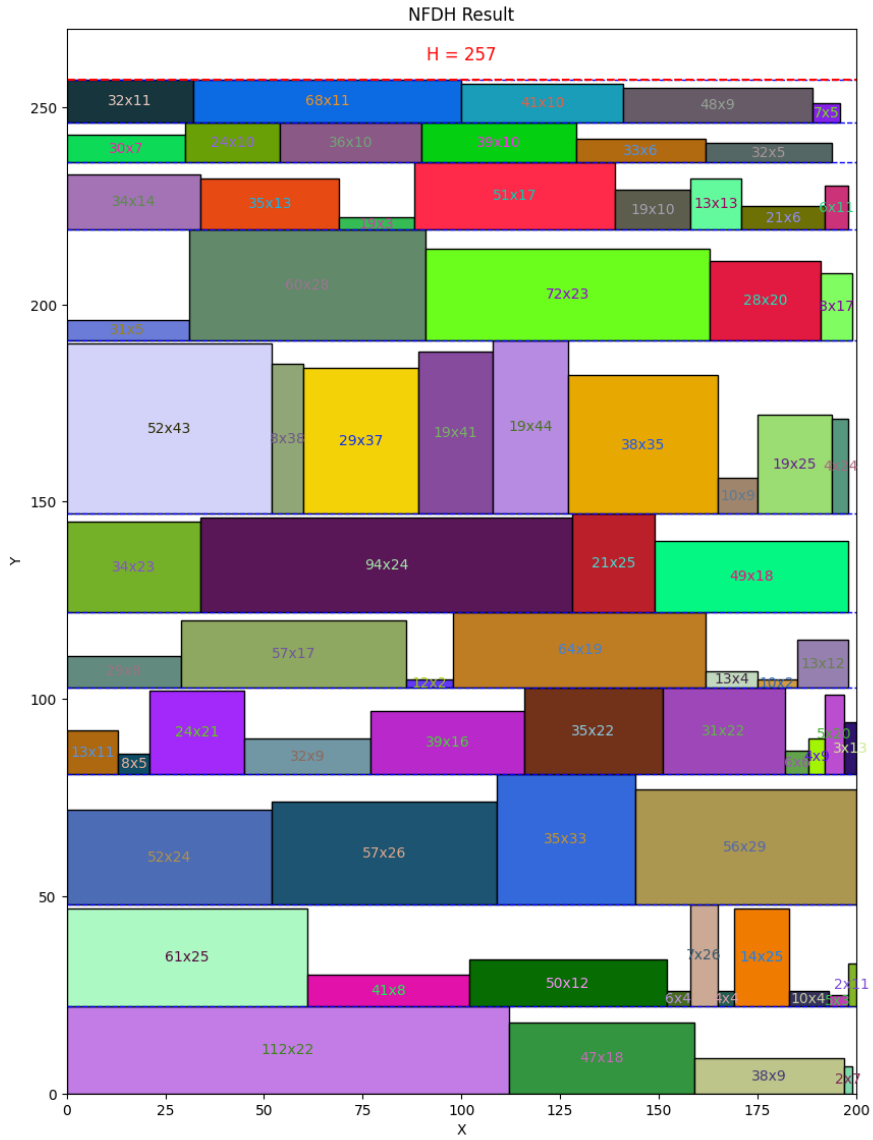


Figure 6: Switch/Swap Algorithm Result

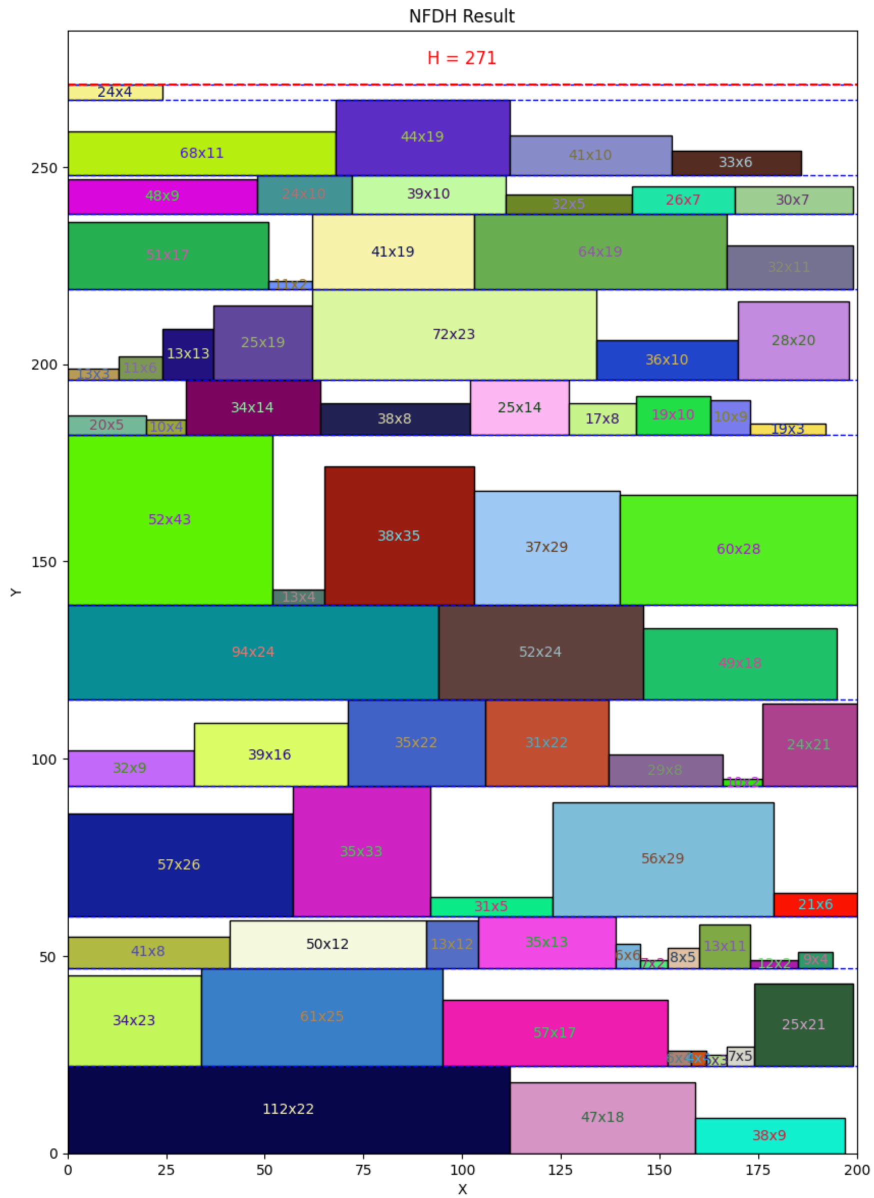


Figure 7: Shift Algorithm Result