Solving the Bin Packing problem

Iova Ioan Alexandru and Alexe Mihail

No Institute Given

1 Introduction

The Bin Packing Problem (BPP) is a well-known combinatorial optimization challenge where a set of items, each with a specific size, must be placed into a minimum number of bins of fixed capacity. The aim is to ensure that the total size of items in any bin does not exceed its capacity. As an NP-hard problem, Bin Packing has a significant theoretical interest and presents practical challenges in finding efficient solutions.

1.1 Problem Description

This work focuses on solving the Bin Packing Problem efficiently. Formally, given n items with sizes $s_1, s_2, \ldots, s_n$ and a bin capacity C, the objective is to minimize the number of bins required to store all items without exceeding the capacity of any bin. Due to its computational complexity, exact solutions are impractical for large inputs, and heuristic or approximation methods are often employed. There also exists a decision variant of this problem, where the aim is to decide whether the items can fit into a specified number of bins.

1.2 Practical Applications

The Bin Packing Problem has practical relevance across many domains:

- In **logistics**, it helps optimize the packing of goods into containers or trucks, reducing costs.
- In **computing**, it is used for resource allocation, such as assigning virtual machines to physical servers.
- In **manufacturing**, it applies to cutting raw materials efficiently, minimizing waste.
- In **memory management**, it assists in efficiently allocating tasks to memory blocks.

These diverse applications underline the importance of developing effective strategies for solving the Bin Packing Problem.

2 Proof of the Bin Packing problem belonging in the NP-Hard class

A well known NP-Complete problem is the Partition problem which asks whether a certain Set $S = x_1, x_2, x_3, ..., x_n$ can be partitioned into S_1 and S_2 such that $\sum_{x \in S_1} x_i = \sum_{x \in S_2} x_j$.

Should there exist a polynomial-time algorithm that determines whether a set S of integers can be fit into n containers of a given size, a function could be built that takes, as input, a Set, calculates the sum of its values and determines, in polynomial time, whether the given values could fit into just two containers whos sizes are half the sum of all the items.

2.1 Function which solves the Bin Packing problem

$$f(S,\ binSize,\ binCount) = \begin{cases} true: \ \ items\ fit\ into\ the\ containers \\ false: otherwise \end{cases} \tag{1}$$

This function can easily be used to create the function g by setting the container size to half the sum of all elements and the number of containers to 2.

2.2 Function which solves the Partition problem

$$g(S) = \begin{cases} true: \ \ items\ can\ be\ partitioned \\ false: otherwise \end{cases} \tag{2}$$

As the Partition problem is easier than the Bin Packing problem so that solving the latter in polynomial time would also solve the former, something which has not yet been achieved, the Bin Packing problem must be NP Hard.

3 Algorithms

3.1 Next fit

This algorithm is the most simple and the fastest one out of all the ones presented here, however its accuracy is quite lacking. The way it works is simple:

1. To start, create the first bin
2. For each element, if there is space left in the bin, add it to the bin
3. If there isn't enough space in the bin, start a new bin

As far as the time complexity is concerned, this algorithm can be proven to be $\theta(n)$, as a constant amount of operations are computed at each step (the if statement to check whether there is enough space for a new item and then either putting the item into the existing bin or creating a new one).

The space complexity is also ideal, being in fact $\theta(1)$. There is need for only two variables, one to store the number of bins used, and one to store the capacity of the current bin. Whenever a new bin is created, the variable containing the capacity of the current bin can just be updated.

When it comes to accuracy, it is guaranteed that the number of bins given by this algorithm is no more than double that of the optimal solution. The reasoning is as follows:

1. the number of bins is at least $\frac{\Sigma_{x \in S} x_i}{C}$, where C is the capacity of a bin.
2. Two adjacent bins are guaranteed to be, together, filled to at least C, otherwise 2 bins would not have been used, as the items in the second would have fit in the first
3. Therefore, in the worst case, all bins would be filled to $\frac{C}{2}$
4. The number of bins used in the worst case would then be $\frac{\Sigma_{x \in S} x_i}{\frac{C}{2}}$, which is twice the amount of the lower bound of the optimal solution.

3.2 First fit

This algorithm gives more accurate results, focusing on checking the bins until one is found that can accommodate the current item. It is slightly more complex than the last one:

1. To start, create the first bin
2. For each element, check all bins until one with enough space left for it is found
3. If none of them can hold the current item, create a new bin for it

Normally, this algorithm's complexity depends a lot on the input, as, for example, if the bin capacity is much greater than the size of the items, this algorithms would be functionally identical to Next fit. In the worst case, only one item can fit in each bin and so all bins are checked at every step. For the first item, there would be one bin checked, for the second, the first bin, and then a new bin would be created. In fact, for item x_n, if all items have their own bin, $n-1$ bins would have to be checked. The total number of checks would be a Gaussian sum from 1 to n, that is, $\frac{n(n+1)}{2}$, which means that the algorithm belongs to the complexity class $O(n^2)$.

The worst case for the amount of space used is when each item needs a separate bin, as they all have to be kept in memory. As the needed memory grows linearly with the number of items in the worst case, it can be said that the spatial complexity is $O(n)$.

As for its accuracy, it has been proven that this algorithm uses at most 1.7 times the amount of bins that the optimal solution uses [2]

3.3 Best fit

. The final approximation algorithms that will be discussed is the best fit algorithm. As the name suggests, at each iteration, it tries to find the bin where the current item would both fit and leave as little space left as possible. concrete steps are outlined here:

1. To start, create the first bin
2. For each element, check all bins and memorize the one with the least space left where the element would fit
3. If none of them can hold the current item, create a new bin for it

The time and space complexities are identical to the first fit, which can clearly be seen when considering the worst-case input: where each item needs its own bin. In this situation, every bin is checked every time, leading to the same Gaussian sum discussed in section 3.2 for the time complexity which is also $O(n^2)$, and to the same n bin variables that lead to the $O(n)$ space complexity.

The similarities do not stop, as it's accuracy is identical to that of first fit, the number of bins used being at worst 1.7 times that of the optimal solution [3].

It is, however, worth noting that on average best-fit does produce better results, but at the cost of a longer computation time. This is due to the fact that First fit doesn't always check each bin which speeds it up at the cost of ignoring potentially optimal item placements.

3.4 Improving First fit and Best fit

There are ways of optimizing these two algorithms in terms of both accuracy and time complexity. A great and cheap way to improve the resulting output is to sort the set of items in non-increasing order so that the biggest values always com first. Because sorting has a time complexity $O(nlogn)$, doing it before the $O(n^2)$ algorithm would not impact asymptotic performance. This has been proven to lead to solutions that are only 1.5 times greater than the optimal solution in the worst case. This was proven in 1994 [4]. Moreover, no polynomial-time algorithm can have a better performance unless $P = NP$. The proof was found in [1] and is as follows:

1. Let there be these two functions, the first one returning the minimum number of bins needed to fit the items of the set S, and the second one solving the partition problem.

$$f(S,\ binSize) = minimum\ bin\ count \tag{3}$$

$$g(S) = \begin{cases} true: & f(S, \frac{\Sigma_{x \in S} x_i}{2}) = 2 \\ false: & f(S, \frac{\Sigma_{x \in S} x_i}{2}) > 2 \end{cases} \tag{4}$$

2. If the answer is 2, the g function returns true in polynomial time.
3. If the answer is false, the function returns false in polynomial time.
4. Because the return value of the function f is necessarily an integer, and the amount of bins returned must be less than 1.5 times the optimal amount, if the real answer to the problem is 2, the function f could only return integers that are strictly smaller than $2 * 1.5$, but greater or equal to 2. In other words, the only option is 2
5. The Partition problem would be solved in polynomial time, which would mean that $P = NP$

Another improvement would be the utilization of special data structures for the bins, such as self balancing binary trees. These would greatly optimize the asymptotic complexity by facilitating the search for the right bin for each item, turning what was on average a $O(n)$ operation into a $O(logn)$ one. This improvement guarantees a $O(nlogn)$ asymptotic performance.

3.5 Backtracking using the Branch and Bound strategy

The algorithm chosen to find the exact solution to the bin packing problem takes the following steps:

1. Sort the items in decreasing order
2. Use the Best Fit Decreasing algorithm to calculate an upper limit to the number of bins needed to store the given items
3. Use a Depth-First Search strategy to explore the decision tree recursively
4. At the beginning of each recursion step, calculate the lower bound of the number of bins still needed given the items left unassigned to any bin and the contents of the bins. If it is greater than the current best bin count, prune this part of the decision tree
5. After pruning, if the current decision node shows promise (its lower bound is smaller than the current best candidate for the optimal solution), run the best fit decreasing algorithm on the items and bins to try to find a better solution in order to improve pruning
6. For each element, try to insert it into the first open bin. If it fits, use recursion to check the decision subtree where the current item is placed in the selected bin
7. After checking the subtree, or if the item doesn't fit in the current bin, continue to the next bin, and eventually create a new bin entirely for the given item

This slow but exhaustive algorithm has a space complexity of $O(n)$ fo the bins, as there is at worst one bin for each item. When it comes to the recursive calls and the size of the stack, because of the depth-first-search strategy, the stack will at any point b smaller or equal to the size of a stack frame multiplied by the

height of the decision tree. Th height of the decision tree is equal to the number of items because the only time a decision is made is when an item is assigned to a bin. It follows that the stack size is also $O(n)$.

As far as time complexity is concerned, the worst-case scenario would be exponential. Should each item fit into any of the open k bins, there would be up to k^n configurations for all items. If there can be one bin for each item, again a worst-case scenario, then $k = n$, and the number of configurations would be n^n, which would make the time complexity $O(n^n)$. In practice however, effective pruning would greatly improve the time taken by the algorithm to solve the given problem while remaining exponential.

3.6 Advantages and disadvantages of the presented algorithms

Each one of the presented algorithm has strengths and weaknesses. To better see them, they have been compiled in a table where OPT is the optimal number of bins. For the First fit and Best fit algorithms, their variants where the input is first sorted in decreasing order and the bins are kept in a special data structure are used.

Algorithm	Time Complexity	Space Complexity	Packing Efficiency
Next fit	$O(n)$	$O(1)$	$2 * OPT$
First fit decreasing	$O(nlogn)$	$O(n)$	$1.5 * OPT$
Best fit decreasing	$O(nlogn)$	$O(n)$	$1.5 * OPT$
Branch and bound	$O(n^n)$	$O(n)$	OPT

Table 1. Algorithm comparison

Essentially, for small amounts of items backtracking is feasible. As the input size grows, First fit decreasing and Best fit decreasing should be utilized to reach suboptimal but still manageable solutions depending on whether the user prefers a slight increase in speed or reliability. Next fit should only be considered if there exist memory concerns, as its space complexity is $O(1)$.

Acknowledgments. Ceva de pus la final daca mai avem chef

4 Evaluation

4.1 Description of the test set construction used for validation

The process of creating the test set for validating the bin packing algorithms involves several steps. The objective was to construct a diverse set of test cases under consistent conditions, enabling a thorough evaluation of the algorithm's performance. The test set was split into two parts, one testing the accuracy of the algorithms and one testing their speed.

The accuracy test set was constructed as follows:

– **Parameter setup:**
  - Bin capacity was set to `500`.
  - The number of items per test case was fixed to `45`.
  - Item weights were randomly generated as integers between `1` and `500`.
– **Input generation:**
  - For each test case, random weights were generated based on the defined parameters.
  - The input data, including the number of items, bin capacity, and item weights, was saved in a file within the `input/` folder.
– **Reference solution generation:**
  - The internal tool `mtp` was used to compute the optimal bin packing solution for each test case.
  - The output of the tool was saved as a reference solution in the `ref/` folder.
– **Test set size:**
  - A total of `1000` test cases were generated.

The speed test set was constructed in the same way as the accuracy one, with a few key differences:

– **Parameter setup:**
  - The number of items per test case was fixed to `10000`.
– **Input generation:**
  - The input data, including the number of items, bin capacity, and item weights, was saved in a file within the `second_input/` folder.
– **Reference solution generation:**
  - There was no reference generated, as the only the execution time was measured.

This structured approach ensures the test sets cover diverse scenarios under uniform constraints, enabling robust validation of the bin packing algorithms. The separation of input data and reference solutions facilitates easy comparison between the algorithm's output and the optimal solutions.

4.2 System specifications on which the tests were run

The tests were run on two different systems, each with its own set of specifications. The script `run_tests.sh` was executed to evaluate four different algorithms (`first_fit`, `best_fit`, `next_fit`, and `mtp`) on a series of input files. The results of each run were saved in separate output directories. Below are the specifications and execution times for both systems.

System 1 The first system used for testing had the following specifications:

- Operating System: Ubuntu 24.04.1 LTS
- Processor: Intel Core i7-7820HQ @ 2.90GHz
- Memory: 15GB RAM (2.6GB used, 12GB free, 12GB available)
- Swap: 4GB Swap (0B used, 4GB free)

The testing process involved running the executables `first_fit`, `best_fit`, `next_fit`, and `mtp` on each input file from the `input/` directory. For each executable, the script ran it with each input file, saving the results in the corresponding output directory.

Execution times for System 1 without echo messages:

- `real`: 4m33.749s
- `user`: 4m24.409s
- `sys`: 0m3.095s

System 2 The second system had the following specifications:

- Operating System: Ubuntu 24.04.1 LTS
- Processor: Intel Core i5-12500H @ 4.5GHz (12 cores, 2 threads per core)
- Memory: 15GB RAM (5.8GB used, 4.6GB free, 9.5GB available)
- Swap: 4GB Swap (0B used, 4GB free)

System 2 was equipped with a more powerful CPU compared to System 1. It has 12 cores, each supporting 2 threads, with a maximum frequency of 4.5GHz. The swap space on this system was sufficient for the test runs, with no swap usage during the tests.

Execution times for System 2 with output redirection:

- `real`: 5m41.972s
- `user`: 5m25.554s
- `sys`: 0m16.601s

Execution times for System 2 without output redirection:

- `real`: 7m0.669s
- `user`: 6m35.186s
- `sys`: 0m25.385s

The execution times on each system provide insights into the performance differences between the two machines when running the same algorithms on identical input files. The detailed results are discussed in the following sections.

`run_tests.sh` executes the algorithms in a loop, where each input file in the `input/` directory is processed sequentially by all four executables. For each test, an output file is generated in the corresponding directory within `out/`, with the results of the bin packing algorithm's execution.

4.3 Illustration, using graphs and tables, of the evaluation results on the test set

The evaluation results on the test set are presented below in the form of tables and a graph, which compare the performance of different bin packing algorithms and show how the execution times vary across the test cases.

Execution Times and Test Set Parameters Table 2 presents the execution times for the four algorithms (`first_fit`, `best_fit`, `next_fit`, and `mtp`) on both systems. The table includes the times for running all the accuracy test cases, both with and without output redirection in the terminal. The algorithms are being run in sequence, their respective completion times adding up to the table values, most of the final value being made up by the `mtp` algorithm.

System	With Output Redirection	Without Output Redirection
System 1	4m33.749s	5m28.868s
System 2	5m41.972s	7m0.669s

Table 2. Execution times across two systems. Test parameters include the number of items and bin capacity.

Table 3 presents the execution times for the three polynomial-time algorithms (`first_fit`, `best_fit`, and `next_fit`). The `mtp` algorithm was not included, as its completion time was great even in the efficiency tests which only contained 45. With 10000 values per test, the completion time would have skyrocketed.

Algorithm	Best Fit	First Fit	Next Fit
Time	51.755s	1m7.159s	45.196s

Table 3. Execution times across two systems. Test parameters include the number of items and bin capacity.

Algorithm Performance Comparison The performance of each algorithm, in terms of the percentage difference from the reference solution, is illustrated in Figure 1. The plot shows the accuracy of the `first_fit`, `best_fit`, `next_fit`, and `mtp` algorithms for all test cases. The percentage difference is calculated as the deviation from the reference bin packing solution.

Miss Percentage for Each Algorithm In addition to the overall performance, we also present the miss percentage for each algorithm. The miss percentage

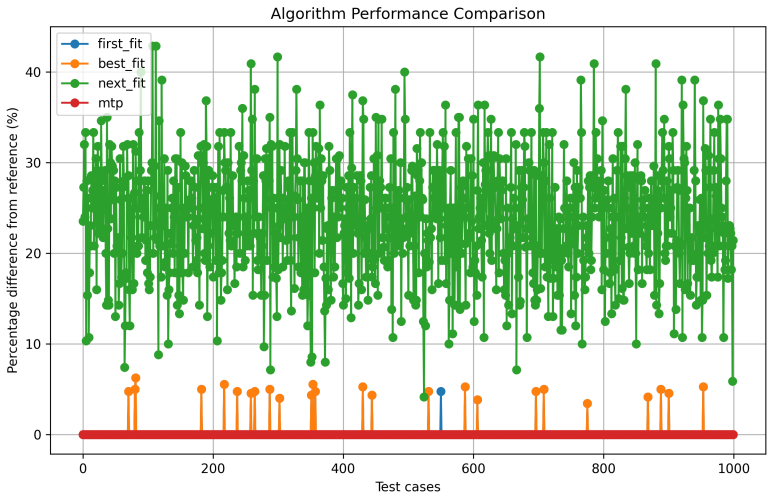

Fig. 1. Comparison of algorithm performance based on percentage difference from the reference solution.

represents the proportion of test cases where an algorithm used more bins than the reference solution. The following results summarize the miss percentage for each algorithm:

- `first_fit`: 2.6%
- `best_fit`: 2.5%
- `next_fit`: 100.0%

These results highlight the differences in the algorithms' effectiveness, with `next_fit` performing less optimally compared to `first_fit` and `best_fit`, as evidenced by a much higher miss percentage.

Best, Worst, and Median Performances The best, worst, and median performance values for each algorithm, based on the percentage difference from the reference solution, are summarized below:

Strategy	Best (%)	Worst (%)	Median (%)
`first_fit`	0.00	6.25	0.00
`best_fit`	0.00	6.25	0.00
`next_fit`	4.17	42.86	24.00

These performance metrics are calculated from the percentage difference between the algorithm's result and the reference solution. Lower values indicate

better performance, with `mtp` (the optimal solution) achieving a perfect match with the reference in all test cases.

4.4 Brief interpretation of the values obtained from tests

The combination of execution times, miss percentages, and best/worst/median performance provides a comprehensive view of the strengths and weaknesses of each algorithm. This section will offer a deeper interpretation of the results based on these metrics.

Execution Times: The execution times of the algorithms on both systems show distinct differences in efficiency. The `first_fit`, `best_fit`, and `next_fit` algorithms had similar execution times, the clear winner however being the last one. A surprising result is seeing First Fit behind Best Fit in the time category. This is, however, explained by the fact that the algorithm used for Best Fir included self-balancing binary trees, while the one used for First Fit did not. Because of this fact, the asymptotic efficiency of Best Fit was $O(nlogn)$, and not $O(n^2)$ as it was for First fit.

Miss Percentages: The `miss percentage` indicates how many times the algorithm used more bins than the reference solution, which points to its efficiency in terms of space utilization.

- `first_fit` and `best_fit` both exhibited the similar miss percentages (of **2.5%** and **2.6%** respectively). This indicates that these algorithms are fairly accurate, performing reasonably well in finding bin packing solutions that are close to the optimal solution in most cases. Between the two, Best fit is slightly more accurate.
- `next_fit`, however, had a **miss percentage of around 100%**. This suggests that this algorithm is highly inaccurate, as it fails to generate acceptable solutions in nearly all cases. The fact that it uses an excessive number of bins in such a large proportion of the instances indicates that the implementation of this algorithm is far from optimal and may not be suitable for solving bin packing problems effectively compared to the other algorithms.

The higher miss percentage of `next_fit` can be attributed to its strategy, which is simpler and might not always make the most optimal decision in selecting the next bin. This leads to more wasted space and higher miss percentages compared to `first_fit` and `best_fit`, which consider more factors before placing an item in a bin.

Best, Worst, and Median Performance: The best, worst, and median performance values provide additional insight into the reliability and stability of the algorithms' solutions across multiple test cases. The `percentage difference` between the results of the algorithms and the reference solution is used to compute these values.

- For `first_fit` and `best_fit`, the best and worst performances were quite close, with the median also reflecting good stability. Both algorithms performed with a **best performance** of **0.00%**, indicating that in some cases they were able to perfectly match the reference solution. Their **worst performances** were relatively low as well, suggesting that even in the worst cases, the discrepancy from the reference was not substantial.
- `next_fit` exhibited more variability in its performance. While its **best performance** was **4.17%**, which is better than the **worst performance** of **42.86%**, the wide gap between the best and worst values indicates that the algorithm struggles to consistently find near-optimal solutions. The **median performance** of **24.00%** confirms this, showing that on average, the algorithm is less reliable and efficient than the others.
- The `mtp` algorithm demonstrated the highest consistency, with all performance values being identical (**0.00%**), indicating a perfect alignment with the reference solution in every case. This outcome is expected, as the reference tests were originally designed to produce this exact result.

General Interpretation: In summary, the `first_fit` and `best_fit` algorithms demonstrated solid performance in both time efficiency and accuracy, making them reliable choices for bin packing tasks. These two algorithms were able to achieve lower miss percentages and generally stayed close to the reference solution in terms of performance.

On the other hand, `next_fit` performed extremely poorly in the accuracy tests. Its higher miss percentage and lower consistency in performance suggest that this algorithm might not be suitable for solving larger or more complex bin packing problems, where an efficient bin allocation is essential.

This analysis indicates that while there is no one-size-fits-all solution, choosing between `first_fit` or `best_fit` depends on the specific problem requirements—whether you prioritize speed, accuracy, or both. It also shows how special data structures can be used to greatly improve the efficiency of algorithms.

5 Conclusions

Based on the analysis, the problem could be approached as follows in practice: for small datasets, the Backtracking algorithm using the Branch and Bound strategy can provide an exact solution. However, for larger datasets, suboptimal solutions like First Fit Decreasing and Best Fit Decreasing are preferred due to their faster execution time, albeit with a slight loss in accuracy. The choice of solution depends on the user's preference between time and accuracy.

6 Bibliography

1. Bin packing and cutting stock problems: Mathematical models and exact algorithms Maxence Delorme, Manuel Iori, Silvano Martello (2016)

2. G. Dósa and J. Sgall. First Fit bin packing: A tight analysis. In Leibniz International Proceedings in Informatics, LIPIcs, volume 20, pages 538–549, 2013.
3. G. Dósa and J. Sgall. Optimal analysis of Best Fit bin packing. In Automata, Languages, and Programming, pages 429–441. Springer, 2014.
4. D. Simchi-Levi. New worst-case results for the bin-packing problem. Naval Research Logistics, 41:579, 1994