**An Experimental Evaluation of Approximation**

**Algorithms for Bin Packing**

By **Tim Etchells**

CS 4445, Professor Solis-Oba

8 December, 2017

**Introduction to the Problem**

The Bin Packing Problem is a combinatorial optimization problem with a wide variety of practical applications. The problem is to assign a series of items to a minimal number of bins, such that the total weight of all items in a bin does not exceed some predefined per-bin capacity *C*. The individual weight of an item cannot exceed *C,* else it would be impossible to pack that item. We also assume that items cannot be split.

For example, given bins with *C = 1,* and a set of items *Si* with weights {0.5, 0.4, 0.1, 0.1, 0.6, 0.8, 0.5}. Then, a human can come up with an optimal packing easily enough:

**Bin 1:** {0.5, 0.5}

**Bin 2:** {0.4, 0.6}

**Bin 3:** {0.1, 0.1, 0.8}

In each bin, the weights sum to exactly 1, so each of these bins is completely full. The number of bins used by any solution *SOL* to a problem instance with a set of items *Si* of size *n* is also lower-bounded by the total weight of the items divided by *C*, that is:

***Equation 1****: Lower bound for an optimal solution*

where *wi*is the weight of item *i*. No solution can do better than this, since all items must be packed, and no bin can hold more than *C* worth of items. The ceiling function also must be applied because we cannot have a fractional bin. Applying this formula to the simple instance above:

we see that any solution must use at least 3 bins, so our solution is indeed optimal.

Returning to the practical implications of the problem, the obvious application is when packing literal bins e.g. on a ship to be transported overseas. The ship can only hold so many bins, so we want to maximize the number of items the ship can carry by minimizing the number of bins used.

Algorithms for this problem can also be applied to backing up data: perhaps the goal is to copy the contents of hundreds of CD-ROMs of varying fullness (weight) onto a minimal number of empty DVDs (bins) without splitting the contents of one CD across multiple DVDs.

A slightly different application of these algorithms is the Cutting Stock Problem [1], which seeks an optimal way of cutting smaller sheets of paper of varying size from a minimum number of larger sheets. In this, the bins are the large sheets, and the items are the smaller sheets to be cut.

Given these applications, the desire for an efficient algorithm for bin packing is understandable. However, the problem is NP-Hard. Due to the exponential growth of the number of possible solutions as the number of items increases, it is prohibitively computationally expensive to iterate over all possible solutions in order to pick the best one. The related decision problem asking whether a set of items fits into *k* bins is NP-Complete, by reduction to the Number Partitioning Problem [2].

There do exist algorithms to seek optimal solutions to the bin packing problem, but such algorithms are generally discussed with input sizes on the order of dozens or hundreds due to their combinatorial complexity [3].

I will focus on polynomial-time approximation algorithms on much larger inputs (usually *n* = 100,000). We evaluate the effectiveness of such approximation algorithms by their worst-case Competitive Ratio, comparing the number of bins used in the algorithm’s solution (denoted *SOL*) to the lower-bound of the problem instance’s optimal solution (*OPT*). A competitive ratio implies a perfect solution – the algorithm has found the optimal solution – so we aim to reach a competitive ratio as close as possible to 1. A competitive ratio is proven by upper-bounding all values of *SOL* produced by an algorithm relative to *OPT*, combined with lower-bounding the values of *OPT* for all possible inputs by methods such as Equation 1. Time complexity is of course also a factor in an algorithm’s effectiveness, so I will evaluate both asymptotic performance of each algorithm as well as average actual performance over a number of test runs.

In this report, a series of well-established approximation algorithms will be presented and analyzed, both for their asymptotic time complexity as well as the relative performance of their competitive ratios. Next, I will present my implementation of a subset of these algorithms, and compare their practical performance from running them on hundreds of problem instances. Finally, I will probably do some other stuff, but I don’t know what yet.

**Analysis of Approximation Algorithms**

The algorithms I have implemented and will be analyzing can be summarized as follows:

|  |  |  |
| --- | --- | --- |
| **Algorithm Name** | **Time Complexity** | **Upper Bound of Competitive Ratio** |
| Next Fit (NF) | *O(n)* | 2 [4] |
| Next Fit Decreasing (NFD) | *O(n ⋅ log(n))* | 2 **\*** |
| First Fit (FF) | *O(n2)\*\** | 1.7 [5] |
| First Fit Decreasing (FFD) | *O(n2)\*\** | [5] |
| Worst Fit (WF) | *O(n ⋅ log(n))* | 1.7 [5] |
| Worst Fit Decreasing (WFD) | *O(n ⋅ log(n))* | 1.25 [5] |
| Almost Worst Fit (AWF) | *O(n ⋅ log(n))* | 1.7 [5] |
| Almost Worst Fit Decr. (AWFD) | *O(n ⋅ log(n))* | 1.25 [5] |
| Best Fit Decreasing (BFD) | *O(n ⋅ log(n))* | [5] |
| PTAS using Almost Worst Fit Decreasing (P-AWFD) | *O(n ⋅ log(n))* | TODO |

**Notes:**

For the purposes of this report, *log(n)* is shorthand for *log2(n).*

**\*** Next Fit Decreasing cannot do worse than Next Fit, because the bound for NF holds in the case where the input is already sorted in decreasing order. But it also does not necessarily do better –in fact in many cases, sorting will make Next Fit perform worse. This is detailed on the next page.

\*\* First Fit and FFD can be implemented in *O(n ⋅ log(n))* time as well, but my own implementation does run in quadratic time.

It’s worth pointing out now that while the *Decreasing* versions of each algorithm generally perform better (excepting NFD as mentioned above), they are not available in the online version of the problem, since you cannot sort the input if you do not yet have access to the whole thing. So, both sorted and unsorted versions are worth discussing, since the former can be applied online and is faster due to skipping the sorting step (usually not affecting the asymptotic complexity) but the latter will generally produce better solutions.

Proper proofs of both the time complexities and competitive ratios of these algorithms will be given as the individual algorithms are presented.

**The Algorithms**

I present the algorithms in pseudocode for readability, translated from my Python implementations with some detail abstracted away. The sort implementation used by the Python language is *TimSort*, which runs in *O(n ⋅ log(n))* time[7], so this is the time complexity I will use for sort calls.

All algorithms have the same input and output, except for the additional Epsilon parameter for P-AWFD:

**Input:** List of *n* items *Si* with weights *wi*

**Output:** List of packed bins, each element a list of items (the item’s index in Si) packed into that bin.

**Next Fit**

The Next Fit algorithm is dead simple, runs in linear time, and has the advantage that a bin can be marked as “sealed” well before the algorithm terminates – in case, for example, some other algorithm is waiting on the output of the bin packing. The basic principle is to try and add each item to the most recently created bin. If it fits, add it, else create a new bin and put it in there, making the new bin the “current” bin.

**Algorithm NextFit(S):**

O(1) bins 🡨 []

O(1) b 🡨 **new** Bin()

O(1) bins.push(b)

**O(n)** **for each** item i in items:

O(1) **if not** (i fits in b):

O(1) b = **new** Bin()

O(1) bins.push(b)

O(1) add i to b and update b.weight

O(1) **return** bins

The key to NF is noting that only one bin – the current bin – is tracked at a time. The older bins can be considered to be sealed, and will not be touched again. This simplicity is the strength of NF, since we never have to iterate over past bins, the algorithm runs in linear time.

The slowest step here is clearly the loop (since this is the non-decreasing version of the algorithm, we skip the sorting step), which iterates over each item exactly once, and performs only constant-time operations within its body. So, we will have exactly *n* loops, and the algorithm runs in *O(n)* time.

NF is proven to have a competitive ratio of 2[4], which is notably worse than those of the other algorithms presented. By ignoring past bins, we sacrifice potential optimizations where smaller items in the later parts of the input could have fit into bins that have already been sealed. However, this is the price of the linear runtime. In summary, Next Fit sacrifices optimization in favour of simplicity and speed.

**Next Fit Descending**

Next Fit Descending is exactly the same as the NF algorithm, except the input *Si* is sorted at the very beginning. As mentioned above, NFD should not be preferred over NF even for offline applications. This is partially due to the fact that the sorting step increases the time complexity to *O(n ⋅ log(n)),* which is the same as algorithms with better competitive ratios. Even more important is the fact that for some inputs, the solution found by NFD can actually be worse than that found by NF, a restriction that does not apply to the other descending versions of bin-packing algorithms.

For example, an input *Si* = {0.6, 0.4, 0.6, 0.4} will produce the following solution using NF:

**Bin 1: {0.6, 0.4}**

**Bin 2: {0.6, 0.4}**

which is clearly optimal. But, NFD will fail to see this solution, and will seal the first bin before either of the 0.4 items is reached, since the input to the main algorithm will instead be

{0.6, 0.6, 0.4, 0.4}:

**Bin 1: {0.6}**

**Bin 2: {0.6}**

**Bin 3: {0.4, 0.4}**

which is a worse solution than that produced by Next Fit. So, due to its weakness as per this example in addition to its worse time complexity, NFD is not a very useful approach.

**First Fit**

The First Fit algorithm iterates over items, and for each item, iterates over all bins to check if each bin has room for the current item.

**References**

**[1] https://www-sciencedirect-com.proxy1.lib.uwo.ca/science/article/pii/S0377221716302491**

[2] <https://people.cs.umass.edu/~barring/cs311/disc/11.html>

[3] <http://www.aaai.org/Papers/AAAI/2002/AAAI02-110.pdf>

[4] <http://ac.informatik.uni-freiburg.de/lak_teaching/ws11_12/combopt/notes/bin_packing.pdf> page 57

[5] <https://ac-els-cdn-com.proxy1.lib.uwo.ca/S0022000074800267/1-s2.0-S0022000074800267-main.pdf?_tid=3dd899aa-dbae-11e7-8fd0-00000aacb35f&acdnat=1512692886_09d1151089edcea67876b92b29268835>

[6] <http://www.labri.fr/perso/eyraud/pmwiki/uploads/Main/BinPackingSurvey.pdf>

[7] <https://mail.python.org/pipermail/python-dev/2002-July/026837.html>