**An Experimental Evaluation of Approximation**

**Algorithms for Bin Packing**

**By Tim Etchells, 250691282**

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.

**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)* | 2.0 [4] |
| Next Fit Decreasing (NFD) | *O(n ⋅ log(n))* | 2.0 |
| First Fit (FF) | *O(n2)\** | 1.7 [4] |
| First Fit Decreasing (FFD) | *O(n2)\** | [4] |
| Worst Fit (WF) | *O(n ⋅ log(n))* | 2.0 [4] |
| Worst Fit Decreasing (WFD) | *O(n ⋅ log(n))* | 1.25 [4, 5] |
| Almost Worst Fit (AWF) | *O(n ⋅ log(n))* | 1.7 [4, 5] |
| Almost Worst Fit Decr. (AWFD) | *O(n ⋅ log(n))* | [4, 5] |
| Best Fit Decreasing (BFD) | *O(n ⋅ log(n))* | [4] |
| PTAS using Almost Worst Fit Decreasing (P-AWFD) | *O(n ⋅ log(n))* | (1 + )⋅ OPT + 1 |

**Notes on the Analysis Section:**

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

\* 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. Most of this work was done by (Johnson, 74)[4].

For the purposes of this report, the capacity *C* of all bins will alwaysbe 1, and the weights of items will be from the domain (0, 1]. This can be assumed without loss of generality, since any set of inputs as well as *C* for an instance of the problem can be divided by the instance’s *C* (which is also, by the definition of the problem, no less than the heaviest item) in order to scale all weights down to (0, 1].

**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[6], 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 bin represented by a list of items 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. Note that in (Johnson, 74) [4] this algorithm is referred to as Next-1 fit.

**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. We present an example to show the worst case of this weakness [5]:

***Equation 2:*** *A bad input for Next Fit*

**Explanation:** A human can inspect this input and see that the ½ items should clearly be put together, taking up a total of bins, and the small items can also be packed into their own bins,

However, this is the price of the linear runtime. In summary, Next Fit sacrifices optimization in favour of simplicity and speed.

**Next Fit Decreasing**

Next Fit Decreasing is exactly the same as the NF algorithm, except the input *Si* is sorted by weight in non-increasing order at the very beginning. As mentioned above, NFD is not usually 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. Generally, NFD cannot do worse than NF, because the bound (2.0) for NF holds in the case where the input is already sorted in decreasing order. If given the bad input for NF shown above, NFD will indeed find an optimal solution. However, there are cases in which sorting will make Next Fit perform worse, a caveat that does not apply to the other decreasing versions of bin-packing algorithms (since they do not “seal” bins).

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

*Si (Sorted)* = {0.6, 0.6, 0.4, 0.4}:

**Bin 1: {0.6}**

**Bin 2: {0.6} 🡨** Bin 1 is sealed

**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 when compared to NF.

**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. This is a much more effective approach than that of Next Fit – since bins are never sealed, if there is room for an item in a bin, that room will always be found.

**Algorithm FirstFit(S):**

O(1) bins 🡨 []

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

O(1) packed 🡨 false

**O(|bins|) for each** Bin b **in** bins

O(1) **if** (i fits in b):

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

**break** inner

O(1) **if not** packed

O(1) b = new Bin()

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

O(1) bins.push(b)

O(1) **return** bins

By iterating over (up to) all bins, we can produce a much more efficient solution in terms of competitive ratio, but this comes at a cost. The double for loop results in a time complexity of *O(n ⋅ |bins|)* where |bins| is the number of bins currently in the list of bins. We can bound this by *n*, since we know the number of bins will never exceed the number of items since an item cannot weigh more than the bins’ capacity. So, the time complexity of this algorithm is *O(n2)*. It can be sped up by placing the bins into a data structure more efficient for searching, such as a binary tree or heap, but this algorithm given is how it was presented in class. The more efficient implementation is used for some of the other algorithms that will be presented.

The competitive ratio of FF was proven in class to be at least 2. However, Johnson proved that the competitive ratio of First Fit is 1.7[4], which is the same bound given to Best Fit and any bin packing algorithm under his category *Almost Any Fit* algorithms. This is the same proof that will be applied to most of the non-decreasing algorithms I am looking at.

**First Fit Decreasing**

Again, FFD is simply FF plus a sort of the input into non-increasing order at the very beginning of the algorithm. Recall also that this version of the algorithm is only available offline. In this case, sorting does not affect the time complexity of the algorithm since *O(n2)* dominates *O(n ⋅ log(n)),* though it should be noted that in practice the sort step does of course slow down the algorithm. In this case, sorting the input actually has a significant effect on the competitive ratio, improving it from 1.7 to [4].

We construct an example demonstrating the strength of FFD over FF[5] :

*Si* =

***Equation 3:*** *A bad input for First Fit, but a good one for FFD*

**Explanation:** is some small term, so that only 6 instances of the item can fit into a bin, 2 instances for , and 1 instance for . The input is composed evenly of these three sizes.

The subsets of the input must be ordered in this way (small to large), so that FF will greedily fill bins with the items, bins with the items, and bins with the items, resulting in a solution of size *.* However, if the inputs are sorted non-increasingly so that the larger items are inserted into bins first, then the items, then the items, the algorithm will correctly create only *n* bins, each containing exactly one item of each size – an optimal solution.

In this case, FFD far outperforms FF, and we see the problem with packing smaller items first – we can be left with a series of large items that don’t fit into any of the existing bins, but also can’t be combined with each other. The small items could be combined with them, but they are already packed into nearly-full bins with other small items. This example clearly demonstrates why Descending versions of bin-packing algorithms produce better results.

An implementation of this exact instance of the bin packing problem will follow in the experimental section of this report.

**Worst Fit and Almost Worst Fit**

The Worst Fit algorithm takes a slightly different approach – instead of picking the first bin that has room, we select the bin which has the *most* room available. This is easily doable in *O(log(n))* time by maintaining a binary search tree of bins sorted by weight. We can then look up the minimum-weight bin and get its index in the list of bins (the bin index must also be stored in the tree as the node’s payload).

The Almost Worst Fit algorithm is nearly identical – but it selects the bin with the *second* most room available, instead of the *most* room. This is also achievable in *O(log(n))* time since we can simply look up the minimum bin and then get its parent. Only one line differs between the two algorithms, so we present them together, with a boolean parameter *almost* which indicates which version of the algorithm to run.

**Algorithm WorstFit(S, almost):**

O(1) bins 🡨 []

O(1) bin\_weights = BinaryTree()

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

O(1) packed 🡨 false

**O(log(n))** lightest\_bin\_node = bin\_weights.min()

O(1) **if** almost:

O(1) lightest\_bin\_node = lightest\_bin\_node.parent

O(1) **if** lightest\_bin\_node **is not** null:

O(1) light\_b = bins[lightest\_bin\_node.bin\_index]

O(1) **if** (i fits in light\_b):

O(1) add i to light\_b and update light\_b.weight

O(1) packed 🡨 true

O(1) **if not** packed:

O(1) b = new Bin()

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

O(1) bins.push(b)

O(log(n)) bin\_weights.insert(b.weight, b.index)

O(1) **else**:

O(log(n)) bin\_weights.remove(lightest\_bin\_node)

O(log(n)) bin\_weights.insert(light\_b.weight, light\_b.index)

O(1) **return** bins

Mostly due to the more complex data structure, this algorithm is quite a bit more verbose than the previous ones. However, there is a notable improvement in time complexity compared to First Fit – the tree allows logarithmic time for looking up the bin we are interested in. We could achieve the same with a simpler implementation lacking the tree, but then the time complexity would be *O(n2).*

The time complexity is dominated by the *n* loops, each performing at least one *min()* call to the tree (as well as one insertion, and either 0 or 1 removals). The tree operations each take *O(log(n))* time, so by doing at least one per loop the algorithm runs in *O(n ⋅ log(n))* time, a noticeable improvement from the naïve First Fit algorithm above.

In terms of the competitive ratios of these algorithms, there is surprisingly a significant difference depending on whether the worst or almost-worst bin is selected. The latter algorithm falls into Johnson’s category of *Almost Any Fit* (AAF)algorithms, while the former does not. So, the competitive ratio of Worst Fit is 2.0[4], doing no better than Next Fit, and in worse time, while Almost Worst Fit achieves 1.7[4, 5], like the other AAF algorithms (so far, First Fit is the only we have discussed).

**Worst Fit Decreasing and Almost Worst Fit Decreasing**

By now we know what the decreasing version of an algorithm entails – we sort the input non-increasingly before doing anything else, and we can only do this offline. In this case, the effect is very similar to that discussed for First Fit above. The competitive ratio for Worst Fit Decreasing is 1.25[4], fitting the criteria for a Decreasing *Any Fit* algorithm as defined by Johnson [5]. Similarly, Almost Worst Fit Decreasing has competitive ratio [4], fitting the criteria for a Decreasing *Almost Any Fit* algorithm [5]. This is considerably better than the non-decreasing versions of each algorithm.

**Best Fit Decreasing**

We discuss Best Fit Decreasing as a slightly different approach to Worst Fit Decreasing. The algorithm is the same, but instead of selecting the bin with minimum weight (or second-minimum weight), we select the one which, after adding item *i*, will have the weight closest to the capacity. That is, the bin which will be “most full” after adding the item, rather than “most empty”. So, we select the bin *b* such that *b.weight + i.weight* is as close as possible to capacity *C*. In order to do this, we implement a method in our weight binary tree *find\_largest\_lessthan(weight)* which runs in *O(log(n))* time (since it’s just a modified *find)*, and finds the value in the tree which is as large as possible, but not larger than the parameter *weight.* Then, we can do the following:

optimal\_weight = *C* – i.weight

best\_bin\_node = bin\_weights.find\_largest\_lessthan(optimal\_weight)

By replacing the lines which select the minimum weight from the bin-weight-tree in (Almost) Worst Fit with these two lines, we get a Best Fit algorithm. Since the *find\_largest\_lessthan* lookup also takes *O(log(n))* time, this new algorithm has the same time complexity as Worst Fit at *O(n ⋅ log(n))*. It also has the same performance in terms of quality of solutions as First Fit and Almost Worst Fit – Johnson uses it as one of his primary examples, and shows that it has a competitive ratio of [4].

**Polynomial Time Approximation Scheme (PTAS) using Almost-Worst Fit Decreasing**

I have also implemented a simple version of the PTAS as created by de la Vega and Leuker, and described in [7]. As noted in the cited report, this is not a proper PTAS since it introduces an additive factor, but we will refer to it as such for simplicity.

First, we split items into “large” and “small” (as determined by some input ). Then, we apply **any** of the *Almost Any Fit* packing algorithms discussed above to the large set, and then do the same with the small set. In my case, I used AWFD, but one could just as easily use FF/D, BF/D, or AWF.

In order to support this, we must also assume our AWFD algorithm can accept a set of bins to start from, so that we can pass it the bin packing containing only the large items. This is trivial, since instead of initializing our *bins* variable to an empty list at the beginning of the algorithm, we can just initialize it to the given set of bins. This means our method signature for WF must change slightly to accept *(List S, boolean almost,* ***List packed****).*

**Algorithm PTAS-AWFD(S,):**

O(1) large 🡨 []

O(1) small 🡨 []

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

O(1) **if** i.weight >  **:**

O(1)large.push(i)

O(1) **else:**

O(1) small.push(i)

**O(n⋅log(n))** large\_packed = **WorstFit**(large, true, [])

O(n⋅log(n)) **return** **WorstFit**(small, true, large\_packed)

This algorithm runs in the same time complexity as the other descending packing algorithms, since its time complexity is dominated by the call to the packing algorithm. However, algorithms using this approach can be proven to have a better approximation ratio dependant on As shown by Theorem 4.7 in [7], the competitive ratio of this algorithm is (1 + )⋅ OPT + 1, which is indeed much better than any of the others we have analyzed so far.

**Summary of the Algorithms**

So far, we have analyzed a number of slightly different approaches

**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] <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>

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

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

[7] <http://www.csd.uwo.ca/courses/CS4445a/algorithms.pdf>