An Experimental Evaluation of Approximation Algorithms for Bin Packing

By Tim Etchells, 250691282

CS 4445, Professor Solis-Oba

8 December, 2017

Section 1: 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 into a bin. We also assume that items cannot be split.

For example, given bins with $C = 1$, and a set of items S_i 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 solution of size 3 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 (including an optimal solution) *SOL* to a problem instance with a set of items S_i of size *n* is also lower-bounded by the total weight of the items divided by *C*, that is:

$$SOL \geq \left\lceil \frac{\sum_{\{i=1\}}^{n} w_i}{C} \right\rceil$$

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

where w_i 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 – so long as there is *something* in a bin, the entire bin must be included in the solution. Applying this formula to the simple instance above:

$$\frac{(0.5 + 0.4 + 0.1 + 0.1 + 0.6 + 0.8 + 0.5)}{1} = 3$$

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, 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 [1].

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 [2].

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 $\frac{SOL}{OPT} = 1$ 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.

Section 2: 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 $\frac{SOL}{OPT}$
Next Fit (NF)	$O(n)$	2.0 [3]
Next Fit Decreasing (NFD)	$O(n \cdot log(n))$	2.0 [Proven in discussion of NFD]
First Fit (FF)	$O(n^2)$*	1.7 [3]
First Fit Decreasing (FFD)	$O(n^2)$*	$\frac{11}{9}$ [3]
Worst Fit (WF)	$O(n \cdot log(n))$	2.0 [3]
Worst Fit Decreasing (WFD)	$O(n \cdot log(n))$	1.25 [3, 4]
Almost Worst Fit (AWF)	$O(n \cdot log(n))$	1.7 [3, 4]
Almost Worst Fit Decr. (AWFD)	$O(n \cdot log(n))$	$\frac{11}{9}$ [3, 4]
Best Fit Decreasing (BFD)	$O(n \cdot log(n))$	$\frac{11}{9}$ [3]
PTAS using Almost Worst Fit Decreasing (P-AWFD)	$O(n \cdot log(n))$	$(1 + \varepsilon) \cdot \mathrm{OPT} + 1$

Table 1: Summary of Time Complexities and Competitive Ratios of Bin-Packing Algorithms

Notes:

Throughout this report, $log(n)$ is shorthand for $log_2(n)$.

* First Fit and FFD can be implemented in $O(n \cdot log(n))$ time as well, but my own implementation does run in quadratic time. This was done to give a baseline to show how much faster the more efficient implementations run in practice on large inputs.

It's worth pointing out now that while the *Decreasing* versions of each algorithm generally perform better, 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) [3].

For the purposes of this report, the capacity C of all bins will always be 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 $O(n \cdot log(n))$ time, 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 S_i with weights w_i

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) 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, 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 $2^{[3]}$, 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 [4]:

$$S_i = \left\{\frac{1}{2}, \frac{1}{2n}\right\} \times \left\lceil\frac{n}{2}\right\rceil$$

Equation 2: *A bad input for Next Fit*

Explanation: A human can inspect this input and see that the $\frac{1}{2}$ items should clearly be put together, taking up a total of $\frac{n}{4}$ bins, and the small items can also be packed into their own bin, taking up a total of just $\frac{n}{2} x \frac{1}{2n} = \frac{1}{4}$ of a bin (so, 1 bin). This solution $\frac{n}{4} + 1$ is optimal. However, NF will instead create $\frac{n}{2}$ bins, each of which contains one $\frac{1}{2}$ and one $\frac{1}{2n}$ item. This is a competitive ratio of just under 2.

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

Next Fit Decreasing

Next Fit Decreasing is exactly the same as the NF algorithm, except the input S_i 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 $O(n \cdot log(n))$, which is the same as algorithms with better competitive ratios.

NFD cannot have a competitive ratio worse than NF, because the upper bound (2.0, Table 1) 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 S_i = {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

$S_{i\ (Sorted)}$ = {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 always a good 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 $O(n \cdot |bins|)$ 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 $O(n^2)$. 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 the 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[3], 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 analyzing.

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 $O(n^2)$ dominates $O(n \cdot log(n))$, 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 $\frac{11}{9} = 1.\overline{2}$ [3].

We construct an example demonstrating the strength of FFD over FF [4]:

$$S_i = \left\{\frac{1}{7} + \varepsilon\right\} \times \left\lceil\frac{n}{3}\right\rceil + \left\{\frac{1}{3} + \varepsilon\right\} \times \left\lceil\frac{n}{3}\right\rceil + \left\{\frac{1}{2} + \varepsilon\right\} \times \left\lceil\frac{n}{3}\right\rceil$$

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

Explanation: ε is some small term, so that only 6 instances of the $\frac{1}{7} + \varepsilon$ item can fit into a bin, 2 instances for $\frac{1}{3}$, and 1 instance for $\frac{1}{2}$. The input is composed evenly of items of these three sizes.

The subsets of the input must be ordered in this way (small to large), so that FF will greedily fill $\frac{n}{6}$ bins with the $\frac{1}{7}$ items, $\frac{n}{2}$ bins with the $\frac{1}{3}$ items, and n bins with the $\frac{1}{2}$ items, resulting in a solution of size $\frac{5n}{3}$. However, if the inputs are sorted non-increasingly so that the larger $\frac{1}{2}$ items are inserted into bins first, then the $\frac{1}{3}$ items, then the $\frac{1}{7}$ items, the algorithm will correctly create only *n* bins, each containing exactly one item of each size – an optimal solution. In this case, the competitive ratio of FF is $\frac{\frac{5n}{3}}{n} = \frac{5}{3} = 1.\overline{6}$, which FFD has competitive ratio 1.

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 demonstrates why Descending versions of bin-packing algorithms almost always produce better results.

An implementation of this exact instance of the bin packing problem will follow in the Section 3 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 balanced (e.g. AVL, Red/Black) 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. To achieve this, 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. If no such bin exists, it behaves the same as Worst Fit. 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))       if almost: lightest_bin_node =bin_weights.second_min()
O(log(n))       else: lightest_bin_node = bin_weights.min()
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 $O(n^2)$.

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 $O(n \cdot \log(n))$ 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[3], doing no better than Next Fit, and in worse time, while Almost Worst Fit achieves 1.7[3, 4], 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[3], fitting the criteria for a Decreasing *Any Fit* algorithm as defined by Johnson [4]. Similarly, Almost Worst Fit Decreasing has competitive ratio $\frac{11}{9}$[3], fitting the criteria for a Decreasing *Almost Any Fit* algorithm [4]. 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 $O(n \cdot log(n))$. It also has the same performance in terms of competitive ratio 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 $\frac{11}{9}$ [3]. This forms the lower bound for competitive ratio of *Any Fit* algorithms.

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 [5]. 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 ϵ). 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 (A)WF must change slightly to accept *(List S, boolean almost, **List packed**).*

```
     Algorithm P-AWFD(S,ϵ):
O(1)       large ← []
O(1)       small ← []
O(n)       for each item i in S:
O(1)            if i.weight > ϵ/2 :
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 ϵ. As shown by Theorem 4.7 in [5], the competitive ratio of this algorithm is $(1 + \varepsilon) \cdot \mathrm{OPT} + 1$, which is indeed much better than any of the others we have analyzed so far for small values of ε.

Summary of the Algorithms

So far, we have analyzed a number of slightly different approaches for bin packing. We see from Johnson's work that FFD, AWFD, and BFD can all be thought of as roughly equivalent in terms of competitive ratio, though the algorithm described here for FFD is worse in terms of time complexity. We can achieve a better competitive ratio by packing large items before small items, which is the principle applied most explicitly by P-AWFD, as well as by all the Decreasing algorithms described.

In the next section, we will run trials of these algorithms on large inputs, and see how their performance, both in terms of runtime as well as competitiveness of output, compares in practice.

Section 3: Experimental Evaluation of the Algorithms

All algorithms were implemented in Python, with the use of a AVL Binary Tree implementation found on GitHub[6] which I modified to add some methods I needed – *min()* and *second_min()* for Worst Fit and Almost Worst Fit respectively, *find_largest_lessthan(weight)* for Best Fit, as well as fixing some small bugs that occurred in my use case since my node "names" were integers instead of strings. The only requirement to run my program is Python 3.5.

First, I present my results of running most of the algorithms discussed on large inputs of size 100,000. Note that for First Fit and First Fit Descending, my implementation (as described above) runs in quadratic time which was too slow on an input of this size. So, experiments involving FF and FFD are run on input size 10,000 instead. My initial implementation of Best Fit also had this limitation due to my iterating over the bin list instead of using a BST to store and look up bin weights. It became obvious quickly that this was far too slow for large inputs, taking about ~900s (15 minutes) to solve one instance of the problem at n =100,000. By reducing the time complexity of Best Fit from $O(n^2)$ to $O(n \cdot log(n))$, the average runtime was cut down to ~7.5 seconds, a speedup by a factor of 120 times!

Items weights are on (0, 1], generated by Python's *random.random(),* and should be distributed uniformly over that domain. Bin capacity C is 1 as usual. The runtime reported is system clock time, not processor time, so it is subject to some variation based on CPU scheduling. This is not likely to be a large factor because my CPU was only under 15-20% load while running the program.

The competitive ratio reported is based on *OPT* as determined by Equation 1, lower-bounding *OPT* based on the total sum of item weights divided by C. In most cases, the optimal solution is actually larger than this (and the ratio therefore better), but it cannot be proven generally without solving for the optimal solution. If we could do this in polynomial time, we would not need to approximate.

What follows are the results from running each of the $O(n \cdot log(n))$ algorithms on the input described above on the same 128 instances of randomized inputs. P-AWFD and FF/D are omitted, and analyzed below.

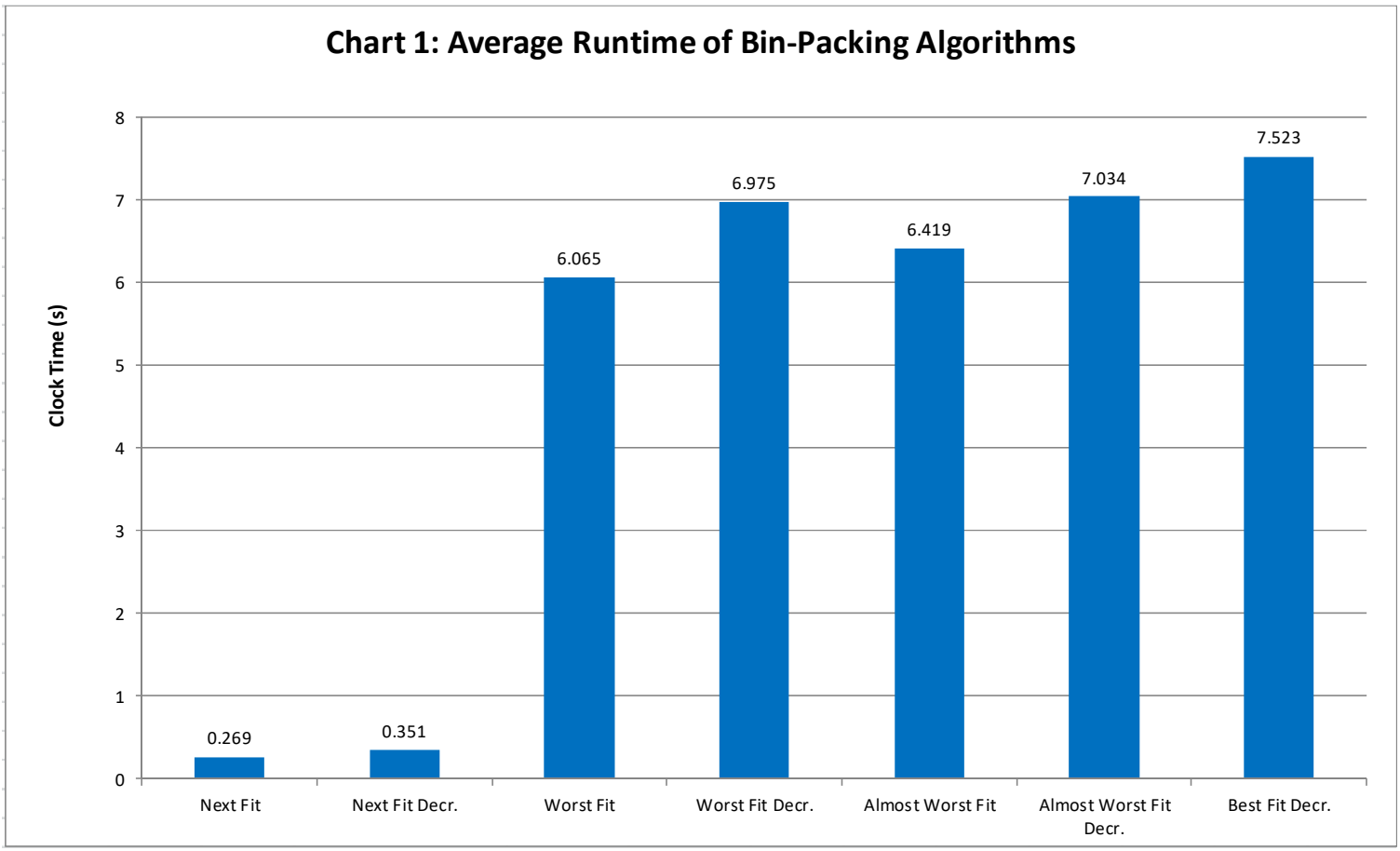
Chart 1: Average Runtime of Bin-Packing Algorithms
Clock Time (s)
8
7
6
5
4
3
2
1
0
0.269
0.351
6.065
6.975
6.419
7.034
7.523
Next Fit
Next Fit Decr.
Worst Fit
Worst Fit Decr.
Almost Worst Fit
Almost Worst Fit Decr.
Best Fit Decr.

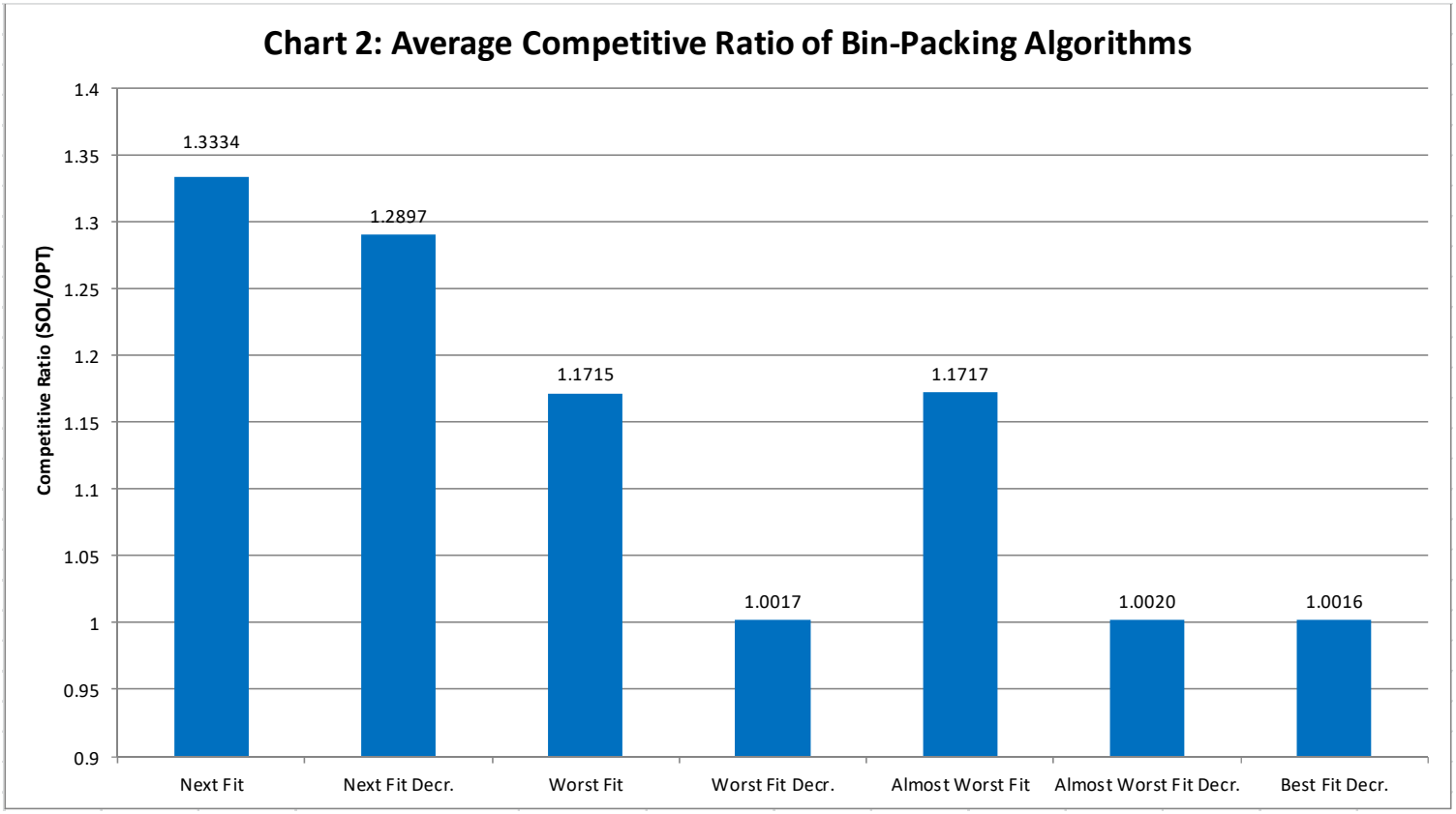
Chart 2: Average Competitive Ratio of Bin-Packing Algorithms
Competitive Ratio (SOL/OPT)
1.4
1.35
1.3
1.25
1.2
1.15
1.1
1.05
1
0.95
0.9
1.3334
1.2897
1.1715
1.0017
1.1717
1.0020
1.0016
Next Fit
Next Fit Decr.
Worst Fit
Worst Fit Decr.
Almost Worst Fit
Almost Worst Fit Decr.
Best Fit Decr.

There are no surprises in the data presented above. We see that, indeed, Next Fit produces the worst solutions, though still with a much better than worst-case competitive ratio around $\frac{4}{3}$, and terminates *much* faster than any of the other algorithms except NFD. The sorting step causes NFD to be quite a bit slower, with a runtime 30% longer, likely due to the overhead of sorting the input. However, notably better solutions are produced by NFD, with an improved competitive ratio of 1.29.

Worst fit and Almost Worst Fit have nearly identical results, as do their decreasing variations as well as Best Fit Decreasing. Again, this is in line with what we expect given Johnson's research – since we are not specifically targeting the worst case for any of these algorithms, they produce very good solutions, well below the upper-bound of $\frac{11}{9}$. The Decreasing algorithms especially produce excellent solutions: with optimal solutions around 50,000 bins, these algorithms use about 80-100 extra bins, a very good approximation. The actual runtime is also very comparable between the algorithms, with the sorting step adding some time in the decreasing scenarios and Best Fit being slightly slower, possibly due to its tree operations being somewhat more complex when compared to Worst Fit and AWF.

It should also be noted that the competitive ratios for the worst solutions over all the inputs produced by all these algorithms are very close to the average competitive ratios, different by a factor of just 1.004 at most. This means all these algorithms are producing these results consistently; without any outliers. Again, this can be attributed to the fact that the inputs are all very similar, and do not target any weaknesses of the algorithms.

Overall, the actual results for the algorithms that are theoretically the best (AWFD, BFD) are very good, and certainly seem more than "good enough" for real world use.

Experimental Evaluation of P-AWFD

When evaluating P-AWFD, we are interested in how the algorithm compares to those already presented (using the same metrics, time and competitive ratio), but we are also interested in the effect of changing the Epsilon parameter ε. From the algorithm's competitive ratio $(1 + \varepsilon) \cdot \mathrm{OPT} + 1$, we know that decreasing ε will result in a better competitive ratio. It seems intuitive that this should come at a cost – I hypothesize that a smaller ε parameter will lead to a longer runtime of the algorithm.

Below I present the average results of running P-AWFD on 128 randomized input instances of size 100,000 items, exactly as done in the previous section (though these are a different set of inputs). We are interested in investigating how changing ε affects the runtime and competitive ratio.

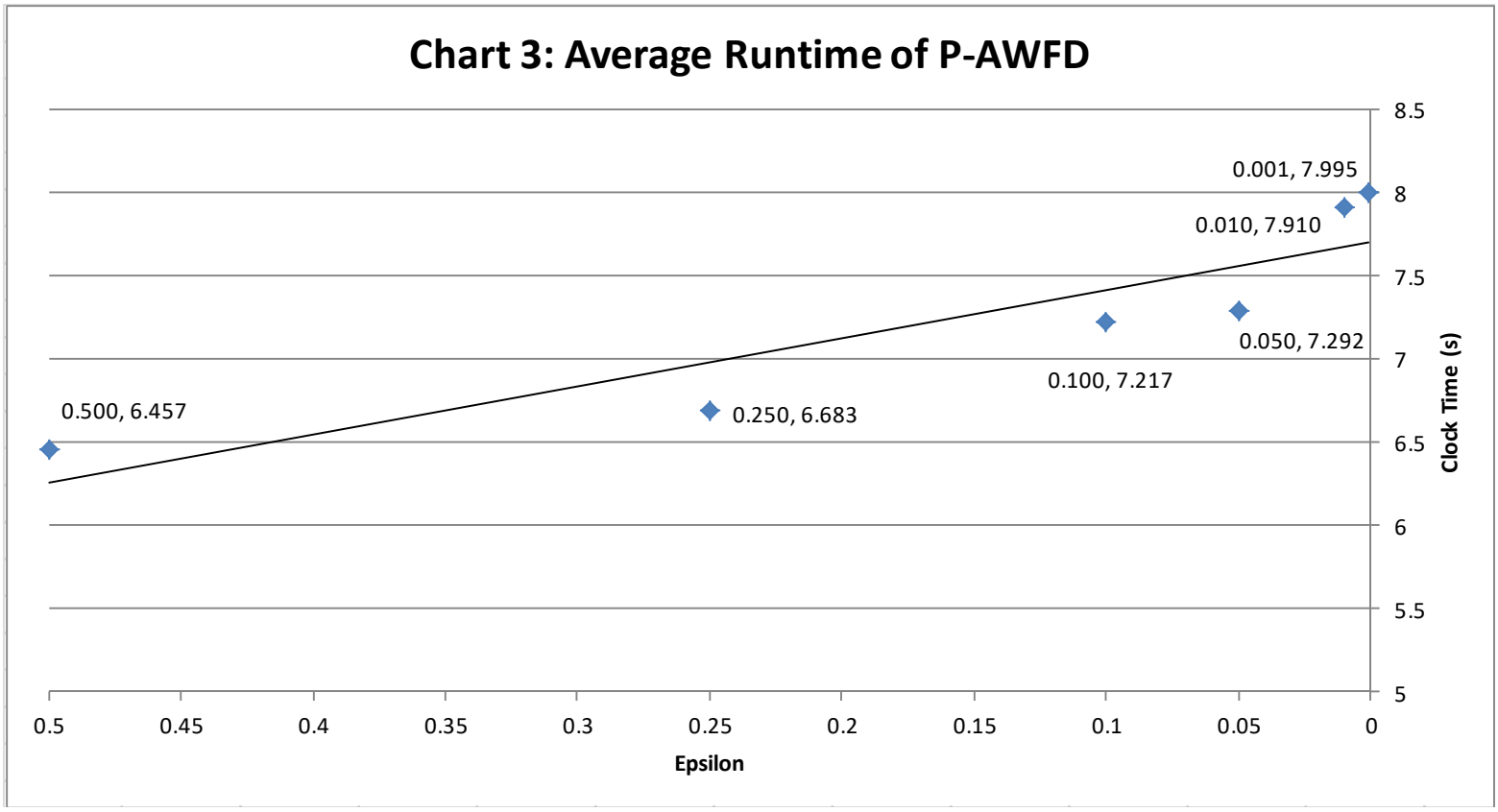
Chart 3: Average Runtime of P-AWFD
0.001, 7.995
0.010, 7.910
0.050, 7.292
0.100, 7.217
0.250, 6.683
0.500, 6.457
8.5
8
7.5
7
6.5
6
5.5
5
Clock Time (s)
0.5
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
Epsilon

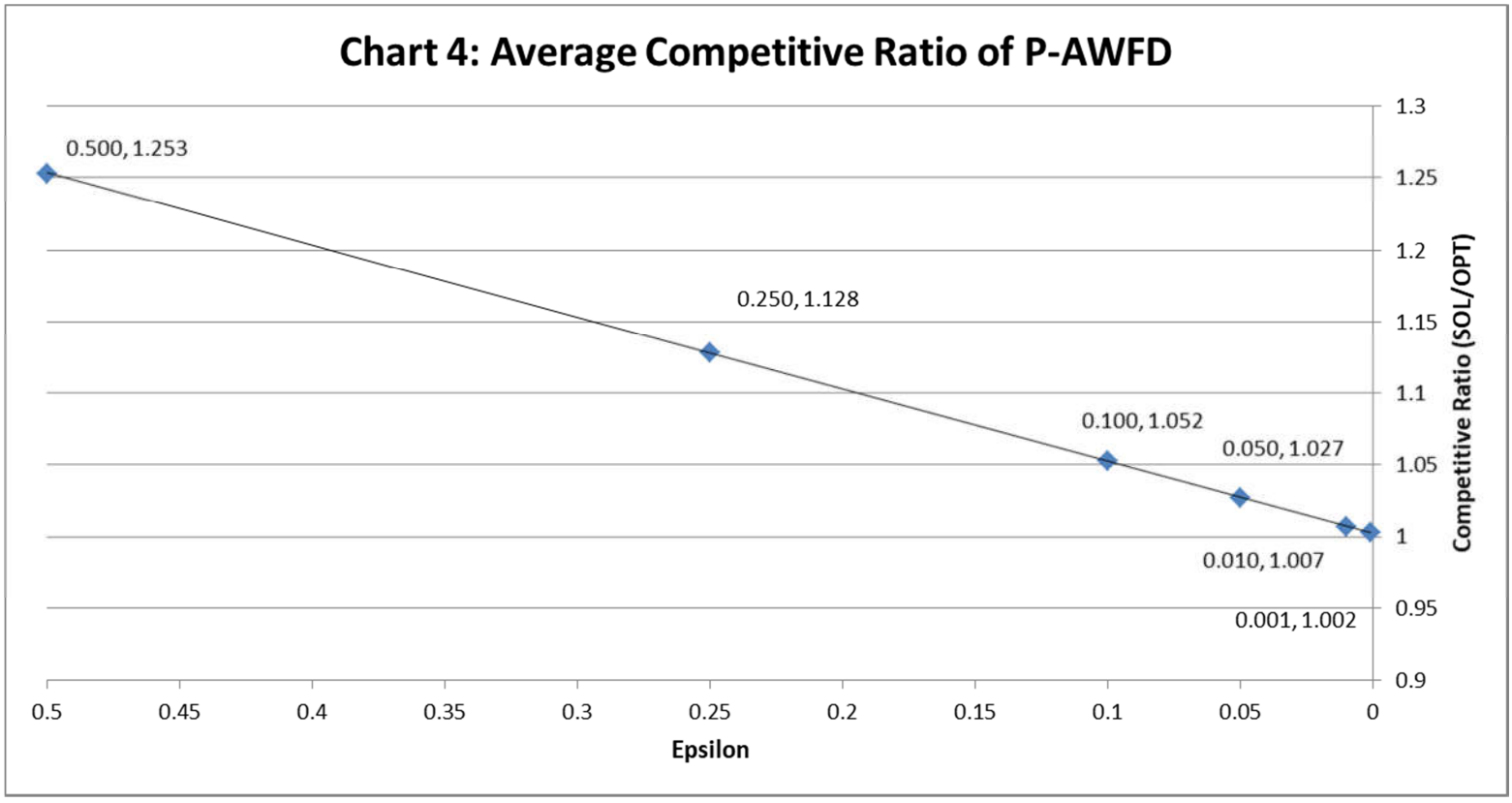
Chart 4: Average Competitive Ratio of P-AWFD
0.500, 1.253
0.250, 1.128
0.100, 1.052
0.050, 1.027
0.010, 1.007
0.001, 1.002
1.3
1.25
1.2
1.15
1.1
1.05
1
0.95
0.9
Competitive Ratio (SOL/OPT)
0.5
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
Epsilon

The charts above show two distinct trends: as ε decreases, so does the competitive ratio, but the algorithm's runtime increases. This confirms our hypothesis that making the ratio arbitrarily good does cost execution time.

A notable hiccup in Chart 4 is that when $\varepsilon = 0.001$, the competitive ratio is actually greater than 1 + ε. This is because as we approach smaller values of ε, the additive term (the + 1) of the competitive ratio $(1 + \varepsilon) \cdot \mathrm{OPT} + 1$ becomes more significant.

When observing all four charts at once, we see that P-AWFD is definitely an excellent algorithm, guaranteeing very good approximations without taking significantly longer than the simpler algorithms presented in charts 1 and 2. In practice, this certainly seems to be the best choice, and ε can be adjusted to fit whether speed or precision of solution is more important to the current application.

Finally, we will explore the bad inputs for Next Fit and First Fit as presented in Section 2, and see how this plays out in practice, as well as how the other algorithms perform on these manufactured inputs.

A Bad Input for Next Fit

Previously we presented an example of an input for which Next Fit performs very poorly. We provide this input to all of our algorithms to see how they perform in this extreme case. Recall Equation 2:

$$S_i = \left\{\frac{1}{2}, \frac{1}{2n}\right\} \times \left\lceil \frac{n}{2} \right\rceil$$

We run a similar experiment to above, but we want to involve First Fit, so we scale down the input size to *n = 10,000* so that FF can finish in a reasonable amount of time. We also perform only one trial, since we are only interested in the output from this one input. The runtimes are included in the table below, but our primary interest here is the solution produced – does the algorithm correctly handle this particular case?

Algorithm	Runtime (s)	SOL/OPT
Next Fit	0.015	1.9992
Next Fit Descending	0.012	1.0
First Fit	3.189	1.0
First Fit Descending	9.444	1.0
Worst Fit	0.502	1.9992
Worst Fit Descending	0.466	1.0
Best Fit Descending	0.509	1.0

Table 2: Performance of algorithms against an input rigged against Next Fit

Here we see that Next Fit (as expected) and Worst Fit are the only two to fall into the trap. Both will return $\frac{n}{2}$ bins, each of size $\frac{1}{2}+\frac{1}{2n}$, rather than $\frac{n}{4}+1$ bins, all full (this is detailed in the description of NF in Section 2). In this case, *OPT* is 2501 bins, while the *SOL* produced by these algorithms is 5000 bins. This matches the expect result exactly. The other algorithms all find the clearly optimal solution of 2501 bins.

Here we see why Worst Fit has a worse competitive ratio than the other algorithms presented, and why it is not preferred for online packing. Offline, it is still comparable to the others, since the descending version avoids this problem just like NFD.

An interesting property of the runtimes above is that both NFD and WFD run *faster* than their non-sorting counterparts (which are the same algorithms which fall into the input trap). FF succeeds at finding the optimal solution, so the descending algorithm is slower as usual. This surprising result can be explained by the descending versions not having to do any "thinking" to come up with the optimum – they simply pair up items of size $\frac{1}{2}$, then group the items of size $\frac{1}{2n}$ into one bin. The first bin tested for potential packing of *every* item is always the correct one, so there is no work to be done within the loop, and the algorithm can execute faster despite the sorting overhead.

A Bad Input for First Fit

We now try a similar experiment with a bad input for First Fit which was also presented in Section 2. Recall Equation 3:

$$S_i = \left\{\frac{1}{7}+\varepsilon\right\} \times \left\lceil\frac{n}{3}\right\rceil + \left\{\frac{1}{3}+\varepsilon\right\} \times \left\lceil\frac{n}{3}\right\rceil + \left\{\frac{1}{2}+\varepsilon\right\} \times \left\lceil\frac{n}{3}\right\rceil$$

Again running the same set of algorithms against a single *n = 10,000* set of items, the results are shown below:

Algorithm	Runtime (s)	SOL/OPT
Next Fit	0.016	1.702
Next Fit Descending	0.023	1.702
First Fit	9.295	1.702
First Fit Descending	8.402	1.021
Worst Fit	0.474	1.702
Worst Fit Descending	0.706	1.021
Best Fit Descending	0.742	1.021

Table 3: Performance of algorithms against an input rigged against First Fit

As with the previous experiment, we see that the algorithms fall into two classes – those which fall into the trap, grouping the items of the same size together (in the previous example, grouping like items was the right thing to do – in this example, it is wrong), and those which correctly group one item of each size into each bin. All of the non-descending algorithms fail to find the correct solution, which is not surprising since they will all greedily pack the $\frac{1}{7}$ items together as they are encountered, then does the same with the other two subsets. The worst failure is that of NFD, which sorts the input but still fails to find a good solution, since it seals each bin as it moves through each subset. However, all the other descending versions of the algorithm pack the items properly, with one of each item in each bin, finding the optimal solution.

The competitive ratio is not 1 despite the descending algorithms finding the optimal solution because the actual optimal (including the generalized optimal used in Johnson's proof) is larger than that given by Equation 1, which is how the optimal was determined for the experiments. This is important to note because in the bad cases for this example, we see that the competitive ratio 1.702 exceeds the theoretical maximum for most of these algorithms as shown in Table 1, 1.70. Again, this is because the actual optimal is larger than that used to calculate the upper bound on the competitive ratio.

We also see another instance of the descending algorithm being faster than the non-sorting one in the case of FF/D. The same logic as the previous example applies here: finding the optimal solution is "easy" once the input is sorted.

Conclusions

From implementing and experimentally evaluating these algorithms, we see that in practice, very good approximation ratios can be achieved, far better than the worst cases as discussed in Section 2, and usually significantly outperforming the competitive ratios as proven by Johnson.

This presents a strong case for not requiring exact solutions for the bin packing problem, since excellent results can be achieved in $O(n \cdot log(n))$ time, far faster than any deterministic algorithm producing an optimal solution can run. Any of BFD, WFD, AWFD, or best of all, P-AWFD can be selected to solve the problem very effectively, and very quickly.

References

[1] People.cs.umass.edu. (2006). Discussion #11 for CMPSCI 311, Fall 2006: Bin Packing. [online] Available at: https://people.cs.umass.edu/~barring/cs311/disc/11.html [Accessed 7 Dec. 2017].

[2] Korf, R. (2002). A New Algorithm for Optimal Bin Packing. AAAI Proceedings, 02, p.731.

Available at: http://www.aaai.org/Papers/AAAI/2002/AAAI02-110.pdf [Accessed 7 Dec. 2017].

[3] Johnson, D. (1974). Fast algorithms for bin packing. Journal of Computer and System Sciences, 8(3), pp.272-314. Most notably the summary on page 312.

[4] Hochbaum, D. (2003). Approximation algorithms for NP-hard problems. Boston, Mass. [u.a.]: PWS Publ., Chapter 2.

[5] Williamson, D. (2017). Lecture Notes on Approximation Algorithms. IBM Research Report. [online] Yorktown Heights, NY: TJ Watson Research Center, pp.26-29. Available at: http://www.csd.uwo.ca/courses/CS4445a/algorithms.pdf [Accessed 7 Dec. 2017].

[6] https://github.com/marehr/binary-tree/