Basic Analysis of Bin-Packing Heuristics
Bastian Rieck

1. Results for the test problems

The benchmarks have been performed using an Intel Celeron M 1.5 GHz. The results are not too surprising: Obviously, the `Next-Fit` heuristic is fastest because only 1 bin has to be managed. However, due to the efficient data structure (a priority queue) that has been used for the `Max-Rest` heuristic, this heuristic will generally be almost as fast as `Next-Fit`. Furthermore, the implementation of the `Best-Fit` heuristic has a worst-case running time of $\mathcal{O}(Kn)$, where K is the maximum weight. Thus, the slowest algorithms are `First-Fit` and First-Fit-Decreasing.

Detailed results can be studied in the table below. In each set, the best solutions are marked using the dagger symbol "†". The timing is not accurate for the small running times. This is due to the `CLOCKS_PER_SEC` macro that has been used for the benchmarks. Hence, in some cases, the same running times will appear. This means that the running times differ by a very small amount (measured in "raw" CPU cycles). A value of 0 signifies that the measurement is outside the notable range.

The algorithms have been compiled with the `-O3` optimizations of the `g++` compiler. All heuristics have been abbreviated to fit in the table. Thus, `MR` is `Max-Rest`, for example. The +-signs after the algorithm names specify whether an optimized version of the algorithm has been used. For implementation details, refer to table 2 on page 6.

Table 1: Results for the test problems

Problem set	Algorithm	Bins	Time in s
`bp1.dat`	`MR+`	628	0
	`FF++`	564	0
	`FFD++`	545	0
	`NF`	711	0
	`NFD+`	686	0
	`BF++`	553†	0
`bp2.dat`	`MR+`	6131	0†
	`FF++`	5420	0†
	`FFD++`	5321†	0†
	`NF`	6986	0†
	`NFD+`	6719	0†
	`BF++`	5377	0†
`bp3.dat`	`MR+`	16637	0†
	`FF++`	16637	0.0071825
	`FFD++`	10000†	0†
	`NF`	16637	0†
	`NFD+`	16637	0†
	`BF++`	16637	0†

Table 1: Results for the test problems

Problem set	Algorithm	Bins	Time in s
bp4.dat	MR+	29258	0.0078125
	FF++	25454	0.015625
	FFD++	25157†	0.0078125
	NF	33397	0.0078125
	NFD+	32174	0†
	BF++	25303	0†
bp5.dat	MR+	32524	0†
	FF++	30155	0.0078125
	FFD++	30111†	0†
	NF	36623	0.0078125
	NFD+	37048	0†
	BF++	30152	0†
bp6.dat	MR+	55566	0.0234375
	FF++	50021	0.0078125
	FFD++	49951†	0.0078125
	NF	63078	0.0078125
	NFD+	59852	0†
	BF++	50021	0.0078125
bp7.dat	MR+	38242	0.015625
	FF++	36863†	0.0078125
	FFD++	39276	0.0071825
	NF	42082	0†
	NFD+	47937	0†
	BF++	36866	0.0703125
bp8.dat	MR+	82804	0.0234375
	FF++	79746	0.015625
	FFD++	79130†	0.015625
	NF	93618	0.0078125†
	NFD+	104096	0.0078125†
	BF++	79731	0.015625
bp9.dat	MR+	293319	0.078125†
	FF++	252577	0.234375
	FFD++	251164†	0.125
	NF	333852	0.0078125†
	NFD+	322643	0.015625
	BF++	251568	0.0390625
bp10.dat	MR+	588478	0.171875
	FF++	506844	0.296875
	FFD++	504927†	0.179688
	NF	669926	0.015625†
	NFD+	645906	0.0390625
	BF++	505643	0.046875
bp11.dat	MR+	2929609	0.882812
	FF++	2510868	8.42188
	FFD++	2502387†	4.86719
	NF	3333928	0.09375†

Table 1: Results for the test problems

Problem set	Algorithm	Bins	Time in s
	`NFD+`	3225064	0.234375
	`BF++`	2504284	1.3125

2. Worst-case running time

2.1. **First-Fit.** In pseudo-code, we have the following algorithm:

Algorithm 1 First-Fit

```
1: for All objects i = 1, 2, ..., n do
2:    for All bins j = 1, 2, ... do
3:       if Object i fits in bin j then
4:          Pack object i in bin j.
5:          Break the loop and pack the next object.
6:       end if
7:    end for
8:    if Object i did not fit in any available bin then
9:       Create new bin and pack object i.
10:   end if
11: end for
```

In the worst-case, a new bin has to be opened each time a new object is inserted. Thus, there are $1, 2, 3, \ldots, n-1$ executions of the inner loop, which yields an asympotical factor of $\mathcal{O}(n^2)$.

2.2. **First-Fit-Decreasing.** Since all weights of the objects are known *prior* to running any algorithm, `Counting Sort` is the best choice for sorting the objects.

Algorithm 2 First-Fit-Decreasing

```
1: Sort objects in decreasing order using Counting Sort.
2: Apply First-Fit to the sorted list of objects.
```

Since `Counting Sort` has a complexity of $\mathcal{O}(n+k)$, where k is the largest weight, the algorithm is obviously dominated by the running time of `First-Fit`, which yields a factor of $\mathcal{O}(n^2)$.

2.3. **Max-Rest.** In pseudo-code, the following algorithm is used:

Algorithm 3 Max-Rest

```
1: for All objects i = 1, 2, ..., n do
2:    Determine k = min{i | c_i = min_{j=1}^{j=m} c_j}, the index of the bin with maximum
      remaining capacity.
3:    if Object i fits in bin k then
4:       Pack object i in bin k.
5:    else
6:       Create new bin and pack object i.
7:    end if
8: end for
```

If a naive algorithm is used, determining the bin with maximum remaining capacity yields a factor of $\mathcal{O}(n)$. Thus, the worst-case running-time of the algorithm is $\mathcal{O}(n^2)$.

A more detailed analysis shows that the bin can be determined by using a *priority queue* (i.e. a heap). In this case, the bin can be determined in constant time. Packing the object (either in a new bin or in an existing one) then requires adding or updating an element of the heap, which can be done in $\mathcal{O}(\log n)$. Hence, the improved version of the algorithm has a worst-case running-time of $\mathcal{O}(n \log n)$:

Algorithm 4 Max-Rest-Priority-Queue

```
1: for All objects i = 1, 2, ..., n do
2:    if Object i fits in top-most bin of the priority queue then
3:       Remove top-most bin from queue.
4:       Add object i to this bin.
5:       Push updated bin to queue.
6:    else
7:       Create new bin and pack object i.
8:    end if
9: end for
```

2.4. **Next-Fit.** In pseudo-code, my implementation proceeds as follows:

Algorithm 5 Next-Fit

```
1: for All objects i = 1, 2, ..., n do
2:    if Object i fits in current bin then
3:       Pack object i in current bin.
4:    else
5:       Create new bin, make it the current bin, and pack object i.
6:    end if
7: end for
```

Since packing an object can be done in constant time, the algorithm is dominated by the loop, which has a running-time of $\Theta(n)$.

2.5. **Next-Fit-Decreasing.** The algorithm is straightforward:

Algorithm 6 Next-Fit-Decreasing

```
1: Sort objects in decreasing order using Counting Sort.
2: Apply Next-Fit to the sorted list of objects.
```

Since `Next-Fit` has a running time of $\Theta(n)$, the dominating factor is the `Counting Sort` algorithm, which has a running time of $\mathcal{O}(n + k)$, where k is the maximum weight of the problem.

2.6. **Best-Fit.** The algorithm works like this:

Algorithm 7 Best-Fit

```
1: for All objects i = 1, 2, ..., n do
2:   for All bins j = 1, 2, ... do
3:     if Object i fits in bin j then
4:       Calulate remaining capacity after the object has been added.
5:     end if
6:   end for
7:   Pack object i in bin j, where j is the bin with minimum remaining capacity
     after adding the object (i.e. the object "fits best").
8:   If no such bin exists, open a new one and add the object.
9: end for
```

Since *all bins* are examined in each step, the algorithm has an obvious running-time of $\mathcal{O}(n^2)$. The running time can be decreased by using a heap in order to determine the best bin:

Algorithm 8 Best-Fit-Heap

```
1: for All objects i = 1, 2, ..., n do
2:   Perform a Breadth-First-Search in the heap to determine the best bin.
3:   if Best bin has been found then
4:     Pack object i in this bin.
5:     Restore the heap property.
6:   else
7:     Open a new bin and add the object.
8:   end if
9: end for
```

Using a heap decreases the running time slightly. An even more rapid implementation of this algorithm will be outlined later on.

3. Speedup for choosing the right bin

Algorithms `First-Fit-Decreasing` and `Next-Fit-Decreasing` greatly benefit from changing the sorting algorithm to `Counting Sort`, thus yielding a factor of $\mathcal{O}(n)$ for sorting instead of the usual $\mathcal{O}(n \log n)$ for Heapsort.

All algorithms benefit from the following idea: Since the minimum weight w is known, a bin can be closed after adding an object if the remaining capacity of the bin is less than the "limit capacity" $c_l = K - w$ (where K is the total capacity of the bin).

If not mentioned otherwise, optimized versions of all algorithms have been implemented. Figures 1 and 2 show the running times of all algorithms for problems `bp5` and `bp8`, respectively. The algorithms depicted in the figures are explained in table 2.

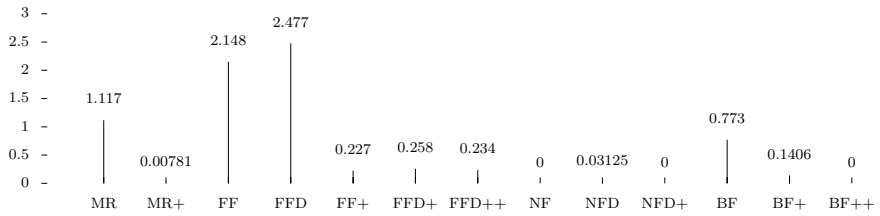
3.1. **First-Fit & First-Fit-Decreasing.** The right bin can be determined very quickly if the algorithm uses a *map* that stores the index j of the bin which contains an object of weight w. Since the bins are filled in increasing order of their indices, all bins with index $j\prime < j$ can be skipped when adding an object of weight $w\prime \geq w$. Asymptotically, the algorithm is still in $\mathcal{O}(n^2)$.

A sample implementation has been supplied with the source code. See function `first_fit_map` for more details. In pseudo-code, we have:

TABLE 2. Naming schemes used in figures 1 and 2

MR	`Max-Rest`
MR+	`Max-Rest` using a *priority queue*
FF	`First-Fit`
FF+	`First-Fit` using the `vector` container from the STL. Thus, "almost full" bins can be removed without having to re-order the array.
FF++	`First-Fit` using the `map` container from the STL. Thus, a lookup table is created that determines the proper bin more rapidly.
FFD	`First-Fit-Decreasing` using the `Heapsort` algorithm.
FFD+	`First-Fit-Decreasing` using **FF+** and `Counting Sort`.
FFD++	`First-Fit-Decreasing` using **FF++** and `Counting Sort`.
NF	`Next-Fit`
NFD	`Next-Fit-Decreasing`
NFD+	`Next-Fit-Decreasing` using `Counting Sort`.
BF	`Best-Fit`

FIGURE 1. Running times in seconds for problem `bp5`. Note that a running time of $0s$ simply means that only very few CPU cycles have been used (in this case, the `clock()` function will not be able to take accurate measurings).

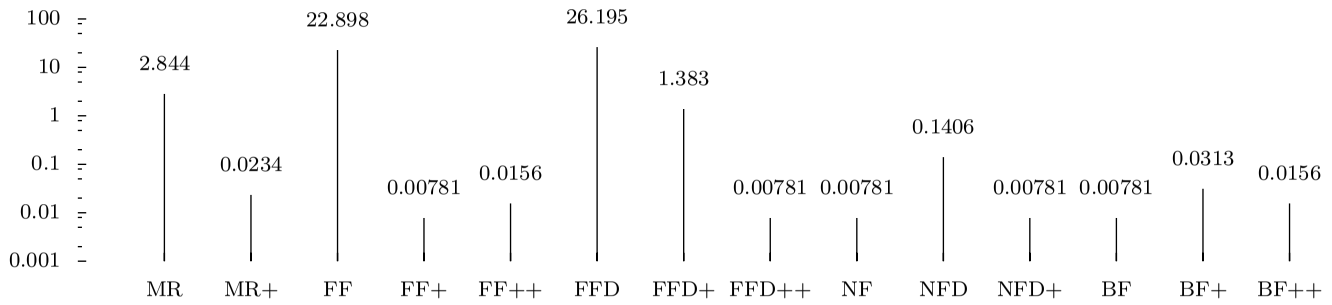


Algorithm 9 First-Fit-Lookup

```
for All objects i = 1, 2, ..., n do
    Lookup the last bin j that was used to pack an object of this size. If no such
    bin exists, set j = 1.
    for All remaining bins j, j + 1, ... do
        if Object i fits in current bin then
            Pack object i in current bin.
            Save the current index in the lookup table.
            Break the loop and pack the next object.
        end if
    end for
    if Object i did not fit in any available bin then
        Create new bin and pack object i.
    end if
end for
```

3.2. **Max-Rest.** This algorithm greatly benefits from a *priority queue* (see the running time analysis): When the heuristic needs to decide where to place an object, the bin with the highest priority (i.e. the *maximum* remaining capacity)

FIGURE 2. Running times for problem `bp8`. Note that a logarithmic scale has been used because the running times differ greatly.



would be chosen from the queue. If the object does not fit, a new bin needs to be created. If it does fit, the bin's capacity is updated and it is placed back in the priority queue.

A sample implementation has been supplied with the source code. See function `max_rest_pq` for more details.

3.3. **Best-Fit.** The `Best-Fit` algorithm greatly benefits from using a *lookup table* for the bin sizes. Index i of this table stores the number of bins with remaining capacity i. When a new object is added, the lookup table is queried for increasing remaining capacities. Thus, the first match will be the best one. The algorithm can be formalized as follows:

Algorithm 10 Best-Fit-Lookup-Table

1: Initialize lookup table t with $t_K = n$ (there are n bins with remaining capacity K, hence no object has been packed yet).
2: **for** All objects $i = 1, 2, \ldots, n$ **do**
3: Let w_i be the current object size.
4: Search a suitable bin by querying the lookup table for bins of remaining capacity $w_i, w_i + 1, \ldots$ until a bin has been found at index t_l.
5: $t_l - -$, because there is one bin less with remaining capacity l.
6: $t_{l-w_i} + +$, because the number of bins with remaining capacity $l - w_i$ has been increased by one.
7: **end for**
8: Compute $\sum_{i=0}^{i=K-1} t_i$ in order to determine the number of bins that has to be used.

Since the remaining capacity of a bin is at most K, a suitable bin can always be determined in $\mathcal{O}(K)$. Because of the outer loop, the algorithm has a running time of $\mathcal{O}(nK)$.

The obvious disadvantage of this algorithm is that the object positions are not stored in the lookup table. This can be solved by using a table of queues, for example, which contain pointers to the "real" bins. In the last two steps of the outer loop, the elements would be removed or added to the queues, respectively. Since this can be done in constant time, the running time of the algorithm is *not* changed.

3.4. **Next-Fit and Next-Fit-Decreasing.** Since only one bin is managed at a time, these algorithms cannot be optimized any further (apart from using `Counting Sort` for `Next-Fit-Decreasing`).

4. Worst-case results

Claim 4.1. $m_{\mathrm{FF}} < \frac{17}{10} \cdot m_{\mathrm{OPT}} + 2$

Proof. Already shown in lecture. □

Claim 4.2. $m_{\mathrm{FFD}} < \frac{11}{9} \cdot m_{\mathrm{OPT}} + 4$

Proof. Already shown in lecture. □

Claim 4.3. $m_{\mathrm{NF}} \leq 2 \cdot m_{\mathrm{OPT}}$

Proof. For two subsequent bins, $c_i + c_{i+1} > 1$ holds. If it were otherwise, no new bin would have been opened. Hence, the solution of the heuristic is at most twice as big as the optimum solution. The bound is *tight* for a problem that contains n objects with weights $\frac{1}{2}$ and n objects with weights $\frac{1}{2n}$. If the objects arrive in

an alternating fashion, i.e. $\frac{1}{2}, \frac{1}{2n}, \frac{1}{2}, \ldots,$ `Next-Fit` will create $2n$ bins, whereas the optimal solution consists of $n + 1$ bins. $\square$