Implementation of the Offline Best Fit Decreasing algorithm for DVD burning

Zoltán Csáti

August 23, 2015

1 Introduction

Besides external HDDs and cloud storage, optical storage facilities (CDs, DVDs, etc.) are still used for data backup. It is often of interest how to place data on them so that the minimum number of disks are required. Manual distribution for many files are cumbersome, therefore some kind of automation is welcome. The problem can be traced back to a more general task.

Problem Statement. *Given a list $L = (a_1, a_2, \cdots, a_n)$ of real numbers in $(0, C]$, place the elements of L into a minimum number M of bins so that no bin contains numbers whose sum exceeds C [1].*

There are two versions of the problem, the offline and the online one bin-packing. In the former, all the elements of L are known a-priori, while in the online version, one element is given at a time.
Because of its practical importance in many aspects of life (memory allocation, prepaging, logistics; just to name a few), researchers constructed several algorithms for it. The classic types are Next-Fit (NF), First-Fit (FF) and Best-Fit (BF). Note that these are heuristic methods, which do not necessarily provide an optimal solution.

Algorithm 1 (Next-Fit). *The first item a_1 is put into the first bin B_1. Items $2, \cdots, n$ are then considered by increasing indices: each item is assigned to the current bin if it fits; otherwise it is assigned to a new bin, which becomes the current one [3].*

Algorithm 2 (First-Fit). *Let the bins be indexed as B_1, B_2, $\cdots$, with each initially filled to level zero. The numbers a_1, a_2, $\cdots$, a_n will be placed in that order. To place a_i, find the nearest j such that B_j is filled to level $\beta \leq C - a_i$, and place a_i in B_j. B_j is now filled to level $\beta + a_i$ [1].*

Algorithm 3 (Best-Fit). *Let the bins be indexed as B_1, B_2, $\cdots$, with each initially filled to level zero. The numbers a_1, a_2, $\cdots$, a_n will be placed in that order. To place a_i, find the nearest j such that B_j is filled to level $\beta \leq C - a_i$ and beta is as large as possible, and place a_i in B_j. B_j is now filled to level $\beta + a_i$ [1].*

Algorithm 4 (Next-Fit Decreasing). *Arrange $L = (a_1, a_2, \cdots, a_n)$ into non-increasing order and apply Algorithm 1 to the derived list [3].*

Algorithm 5 (First-Fit Decreasing). *Arrange $L = (a_1, a_2, \cdots, a_n)$ into non-increasing order and apply Algorithm 2 to the derived list [1].*

Algorithm 6 (Best-Fit Decreasing). *Arrange $L = (a_1, a_2, \cdots, a_n)$ into non-increasing order and apply Algorithm 3 to the derived list [1].*

Apart from approximate algorithms, exact bin-packing algorithms exist, see e.g. [2]. Let us denote by B_m the required number of bins for a specific method and denote by B_{opt} the optimum number of bins. The upper bound r is called the worst-case performance ratio and is defined as

$$\frac{B_m}{B_{\text{opt}}} \leq r.$$

The asymptotic worst-case performance ratio r^∞ is more commonly used. It is defined as the limit of r as the number of items tend to infinity. Table 1 is taken from [3] and shows the time complexity and the asymptotic worst-case performance of the described algorithms.

Algorithm	Time complexity	r^∞
NF	$\mathcal{O}(n)$	2
FF	$\mathcal{O}(n \log n)$	17/10
BF	$\mathcal{O}(n \log n)$	17/10
NFD	$\mathcal{O}(n \log n)$	1.691...
FFD	$\mathcal{O}(n \log n)$	11/9
BFD	$\mathcal{O}(n \log n)$	11/9

Table 1: Complexity of approximate bin-packing algorithms

Remark 1. *As we are going to use BFD, we note that bounds exist for it (see [1]):*

$$\frac{11}{9} B_{opt} - 2 \leq B_m \leq \frac{11}{9} B_{opt} + 4.$$

The same applies to FFD.

2 Implementation

2.1 Implementation details

For various reasons (speed, open source, portability), the program was written in the C programming language. The program was created on a computer with an Intel Core i5-2500K processor and 8GB RAM running 64bit Windows 7. We built it with Microsoft Visual Studio 2013, therefore the executable is

Windows-only and requires the 64 bit Visual C++ Redistributable Package for Visual Studio 2013[1]. However, you can also compile the source file `DO.c` with another C compiler, like `tcc`[2].

Value	Cause
-1	The input file location is missing
-2	The output and input file locations are missing
-3	Unknown option
-4	Input file must contain directory or drive
-5	Could not open the file for reading
-6	Could not open the file for writing

Table 2: Return values and their causes

2.2 Program structure

The program can be divided into four logically different sections.

The first one consists of the description, includes the header files, declares the functions used and declares the two fundamental structures, `bin` and `item`.

Function `main` begins with the declaration and definition of local variables, then continues with the handling of command-line arguments. To process them, we applied a similar manner seen in the `.c` file of PTGui []. After it, we deal with the case if no output location is given: this time the output will be written to the same folder where the input is. In the next few lines of our code, the number of input items are counted with `nlines` and is loaded to an array of `bin` structures with `loadData`. In BFD, letter D stands for decreasing, so the structure array, representing the items, are sorted in descending order by `sortDescend`. Then the elements that do not fit into the bins (i.e. item size greater than C) are excluded. With this knowledge, ... number of bins are allocated. These steps are referred to as the preprocessing stage and comprise the second part.

The third part realizes the algorithm itself. The idea is the following. We loop through the items we want to distribute and in each loop the following is done. According to the BF algorithm, the next item must be put into the bin with the least free space. The free space in a particular bin is the difference of the bin capacity and the current load level. It is determined by the `freeSpace` function. Next, we put the current item into all bins virtually and the resulting free space is calculated by the difference of the previous free space and the item size. If this value is negative, it indicates that the current item does not fit into that bin. Otherwise, the less the new free space is, the better it is. So the minimum is to be found. However, the negative values cause that the specific case would be found when it does not fit. A safe way to exclude bins with negative free space values is to give them a larger value than the bin capacity. Therefore, the `processNegativeValues` function provides the array

[1]Download: http://www.microsoft.com/en-us/download/details.aspx?id=40784
[2]Tiny C Compiler, http://bellard.org/tcc/

of free space gained by placing the current item to all the bins and switching the negative numbers to $C + 1$. After it, the `minimum` function finds the index of the minimum value we get when the specific item is placed in the right bin. Finally, we put the item to the location of the chosen bin determined by the `nextIndex` field of the `bin` structure. This, and the `usedSpace` field (tracking the load level of the bin) are updated. Basically, these are performed in each iteration.

When we have processed all the items, the structures are ready to be written to the output file with `saveData`. Function `main` ends with the release of the dynamically allocated variables.

After function `main`, the applied functions are defined.

Remark 2. *Memory consumption. As mentioned, the program takes the worst case into account. It means that much more space is allocated than it needs. However, for the typical application (even burning 100 DVDs), this is not a memory concern on today's computers. Moreover, it maintains simple program structure.*

Remark 3. *Equal bin sizes. Up to the current release disk size is constant. In most cases it is not a problem since most people want to write all their data on one specific type of disk (e.g. DVD5).*

Remark 4. *Language. For maximum portability, we insisted on C89.*

3 Usage

The program is a command-line tool. It takes at least one input argument (besides the program name): the location of the input file. However, additional options can be given using *flags* as shown in Table 3.

Flag	Description
`-a`	Appends the output file instead of overwriting it if it exists
`-d`	Specify the bin capacity (default: 4700)
`-h`	Displays help on the screen
`-o`	Specify the output file location. `-o` must be followed by the location
`-v`	Displays program version on the screen

Table 3: Optional parameters for the program

The input file must contain exactly two columns; the first column consists of the file name, the second one stands for the size. The space holding in a row can be arbitrary number of spaces or tabulators. One possible sample is shown in Table 3.

Film1	3719
Film2	1943
Film3	5114
Film4	645.7

Table 4: Sample input

The general syntax to call the program is

```
DO -option1 -option2 ... -optionN inputFile
```

Now, let us take some examples.

Example 1: Suppose that we want to write four films (their sizes are given in MB) seen in Figure 3 onto DVD5 disks with capacity of 4.7 GB. We immediately see that Film3 does not fit on a DVD5 disk. To be optimal, Film1 and Film4 are placed on one disk because that way the least free space remains. It is a trivial example but for many files the program prevails over humans. We save the data of Table 3 as `films.txt` to drive `D` and invoke the program as

```
DO -d 4700 D:\films.txt
```

The program returns with a warning printed into the command prompt that Film3 does not fit on the disks with capacity set as 4700 with the `-d` flag. It also shows that the result of the classification is saved into `D:\output.txt`, the default location which is in the same folder as the input file.

Example 2: Let's stick to the first example, but now, write the films on DVD9 disks and save the results into another directory. We are also curious about the program version. So

```
DO -v -d 9400 -o D:\Programs\result D:\films.txt
```

The order of the options is arbitrary but the input file must be given in the last place.

Example 3: Assume, we would like to add one other film to our compilation: Film5; 2104.05 MB. In order output.txt is not overwritten, we have two options: use the `-o` flag to give a different file name or use `-a` to append the existing file. The first approach is described in Example 2, here we take the latter strategy:

```
DO -a D:\films.txt
```

Note that the disk size is omitted since the default value is 4700.

4 Conclusion

A lightweight, easy to use file distributor was created which can be used for other 1D offline bin-packing problems. The program is amply commented and due to its modular structure, it can be easily modified and augmented. One possible further extension would be the creation of a GUI which could make the preprocessing stage even faster and more user-friendly. Thanks to the documentation, the applied concepts can also be used for pedagogical purposes.

5 Code

References

[1] D.S. Johnson, A. Demers, J.D. Ullman, M.R. Garey, and R.L. Graham. Worst-Case Performance Bounds for Simple One-Dimensional Packing Algorithms. *SIAM J. Comput.*, 3(4):299–325, 1974.

[2] R. E. Korf. A New Algorithm for Optimal Bin Packing. In *Eighteenth National Conference on Artificial Intelligence*, pages 731–736, 2002.

[3] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*, volume 25 of *Wiley Series in Discrete Mathematics and Optimization*. Wiley, first edition, 1990.