Structure de données et algorithmes

Projet 2: bin packing

25 février 2025

The objective of the project is to implement, theoretically analyze, and empirically compare different approximate algorithms to solve the bin packing problem. Implementing these different algorithms will allow you to implement different data structures covered in the course (priority queues and binary search trees).

Bin packing

The SEGI is calling you to solve the following algorithmic problem. The Rector has asked them to make a complete backup of the university's data but would obviously like to do so at a lowest cost. The university's data is distributed across N files of strictly positive integer sizes denoted $s_i \in \mathbb{Z}_0^+$, with $1 \leq i \leq N$. To simplify purchasing, the SEGI plans to only order disks of size B. To facilitate archiving, we also want to store each file entirely on a single disk. We will therefore assume that $s_i \leq B$ for all i. To minimize costs, SEGI asks you to determine the minimum number of disks needed to store all the files under these constraints and how to distribute the files between the disks to achieve this minimum number. You already know, of course, that this number will be between $\lceil \sum_i s_i/B \rceil$ and N.

After some research, you realize that this algorithmic problem is the well-known bin packing problem [1] and by reading about it, you learn that it is an extremely complex problem and that unfortunately no exact, efficient solution exists. You therefore resort to approximate solutions based on greedy heuristics. Your idea is to implement them efficiently (N is potentially very large) and see which one will yield the best solution on the university's data.

Most of these heuristics are based on the same greedy approach. Files are processed sequentially. For each one, the most suitable disk to place the file is searched for among the open disks (i.e., disks that have already received a file) according to a specific strategy detailed below. If no disk can contain the file, a new disk is created on which the file is placed, and this disk is added to the list of open disks.

The following four disk selection strategies seem most relevant to you:

- **Next-Fit**: Keep only one disk open and fill it with the files scanned in order as long as possible. As soon as a file doesn't fit on the current disk, move to a new disk.
- **Worst-Fit**: Place the current file on the least full disk of those open.
- **Best-Fit**: Place the current file on the most full disk that can still contain the file.
- **First-Fit**: Place the current file on the first disk, in their opening order, that can contain the file.

[1] https://en.wikipedia.org/wiki/Bin_packing_problem

For best performance, you will process files in descending size order in your various implementations. You read that processing small files at the end allows you to fill in the gaps left by large files.

Data Structures

Since the university's files number several hundred thousand, your implementation must be as efficient as possible to allow SEGI to obtain a solution in a reasonable amount of time. This means that for certain strategies, you will need to use the most appropriate data structures. Some ideas are given below for each strategy.

Next-Fit. This is the simplest approach. No special structure is required to implement it efficiently.

Worst-Fit. This solution requires accessing for each new file on disk among the open ones that is the least full. It must also be possible to update the empty disk space. This seems to fit very well with the specifications of a priority queue.

Best-Fit. Here, we must be able to access the disk with the lowest free space that can receive the file. This seems possible if you store the open disks in a binary search tree using the size of the free disk space as the key (see figure 1 and question 3 of the theoretical analysis). To ensure efficient code, however, it is necessary to use a balanced tree implementation.

First-Fit. This is the most complicated approach to implement efficiently. It is necessary to be able to find the first open disk that can contain the file. Therefore, there are two criteria to consider: the size of the empty space on the disk and the time at which the disk was opened. One possible approach is to use a binary search tree using the disk index as the key in the order it was opened. By adding the maximum capacity of the disks in the subtree of which this node is the root, to each node of this tree, it is possible to quickly find the target disk (see figure 2 and question 4 of the theoretical analysis).

Implémentation

We provide you with the following files:

- `File.h/File.c`: a simple data type to represent a file, characterized by a name and a size (of type `size_t`)
- `Disk.h/Disk.c`: a simple data type to represent a disk, characterized by its capacity, a free space size, a used space size, and a list of files.
- `LinkedList.h/LinkedList.c`: a list implementation, which will be used to provide data to your functions and retrieve the results. The implementation also contains a list sort function, which will be useful for sorting files in descending order.
- `main.c`: a main file that allows you to read an input file containing a list of files in csv format to test your implementations. It is also possible to create a list of random files for larger-scale testing.
- `Makefile`: Allows you to compile all files and test them.

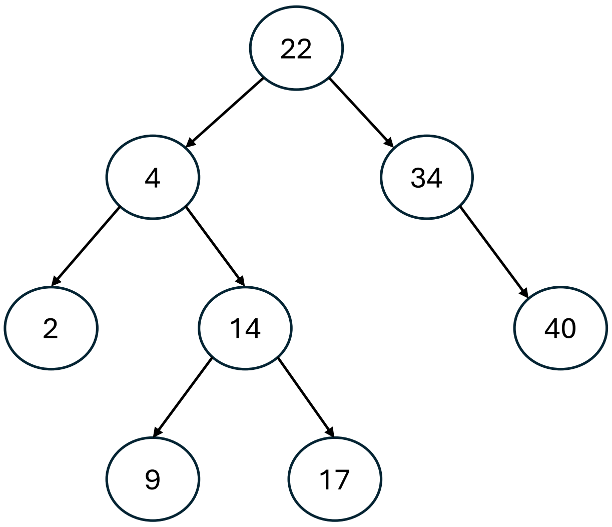

Figure 1: A binary search tree whose keys represent the space remaining on 8 open disks. With the Best-Fit strategy, a file of size 12, for example, would have to be stored on disk size 14, a file of size 17 on disk size 17, and a file of size 41 would lead to the creation of a new disk.

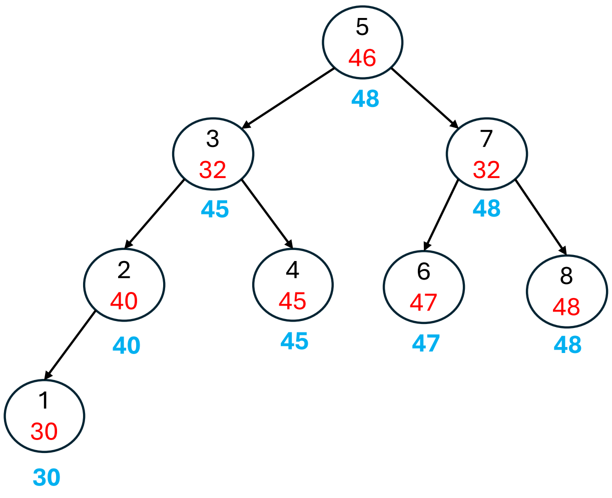

Figure 2: A binary search tree to implement the First-Fit approach. The key, in black, is the disk creation index, the value in red is the free disk space, and the value in blue is the maximum disk space among all nodes in the subtree whose node is the root. With the First-Fit strategy, a file of size 32 must be stored on disk 2 and a file of size 44 on disk 4. This representation allows the target disk to be found by traversing a single branch in the tree.

You must implement the following four files:

`BP_nextfit.c`, `BP_worstfit.c`, `BP_bestfit.c`, `BP_firstfit.c`.

Each of these files must implement the solution corresponding to the bin packing problem through the single function `binpacking` declared in the header file `BP.h`, which takes as input a disk size (B above), a list of files, and a list of disks, initially empty. The function must return the number of disks found by the algorithm and place these disks, along with their files, into the list given as the third argument. A solution `BP_naive.c` is provided as an example, which simply creates a new disk for each file by taking them in the initial order.

To implement `BP_worsfit.c`, you must rely on a priority queue that you will have to implement yourself in the file `PQ.c` according to the interface defined in `PQ.h`. To implement `BP_bestfit.c` and `BP_firstfit.c`, you can rely on an existing implementation of a generic binary search tree, one of these two:

- `avl_bf.c/avl_bf.h`, obtained here https://github.com/xieqing/avl-tree/.
- `treap.c/treap.h`, obtained here https://github.com/matthewclegg/treap.

The first is an implementation of AVL as seen in the course. The second is an implementation of what is called a Treap or a randomized tree[2]. Unlike AVL, this approach does not guarantee $\log N$ in the worst case, but only on average. However, this implementation provides sufficient performance for our application and is somewhat simpler than AVL. You can modify these implementations as you wish for your implementations of the files `BP_bestfit.c` and `BP_firstfit.c`. Your modified tree code should be provided in the files `BST_bestfit.c/.h` and `BST_firstfit.c/.h`, respectively, which are compiled by the provided `Makefile`.

Suggestions: For `BP_bestfit.c`, simply use the tree to store the disks and add a function to the interface to find the appropriate disk (this is the subject of the third theoretical question below). For `BP_firstfit.c`, two pieces of information must be added to the tree nodes: the free disk space at the node and the maximum disk space among all disks in the subtree, including this node and the root. It is then possible to find the appropriate disk efficiently (see the fourth theoretical question). The main difficulty, however, is keeping the disk space information up to date during insertion, taking rotations into account. Note that for `BP_firstfit.c`, you don't normally need to delete disks from the tree (if a disk's capacity is 0, it will never be chosen). You can therefore remove this function from the code and therefore don't need to modify it to take into account the information about disk spaces.

Analyse théorique

In your report, answer the following questions:

1. Give a rough idea of the complexity of a solution to the problem by exhaustive search (as a function of N).
2. Prove, by giving a counterexample, that at least two of the four proposed greedy solutions are not optimal.
3. Give the pseudo-code of a function `Tree-Search-BF(T, k)` finding, in a binary search tree represented as in the lecture, the smallest key of the tree greater than or equal to k. Analyze its worst-case time complexity as a function of N.
4. Give the pseudo-code of a function `Tree-Search-FF(T, size)` finding, in a binary search tree as represented in figure 2, the node corresponding to the first disk that can contain a file of size `size`. Let `x` be a node. `x.key`, `x.space`, `x.maxspace` contain, respectively, the disk number, the free space on this disk, and the maximum space among the disks in the subtree rooted by `x`. Analyze its worst-case time complexity as a function of N.

[2]https://en.wikipedia.org/wiki/Treap

5. Analyze the worst-case time complexity of the four algorithms as a function of the number of files N, taking into account (and specifying in the report) your implementation choices.

Empirical Analysis

In your report, answer the following questions:

1. Using the provided file `main.c`, plot the waste generated by each solution as a function of N on a graph. Waste is defined as the difference between the sum of the disk sizes used and the total file size. You can set the disk size to 1,000,000. Average curves for at least 5 experiments. Briefly discuss the differences between the methods.

2. For the same experiments, also plot the average computation times as a function of N. Discuss the agreement between these curves and the theoretical analysis in the previous section.

Submission and Deadline

The project must be completed either alone or in pairs by Sunday, May 11, 2025, at 11:59 PM. The project must be submitted via Gradescope (http://gradescope.com, course code 42V2RV).

The following files must be submitted:

1. your report (maximum 6 pages) in PDF format and named report.pdf. Be brief but precise and carefully follow the numbering of the (sub-)questions,

2. the four files `BP_nextfit.c`, `BP_worstfit.c`, `BP_bestfit.c`, `BP_firstfit.c`,

3. The implementation of the priority queue, `PQ.c`,

4. the two tree implementations `BST_bestfit.c/.h` and `BST_firstfit.c/.h`.

Please respect file extensions and file names, including upper/lower case. Do not include any additional files.

A project not submitted on time will receive an overall grade of zero.

Submitting the project assumes that you have read the plagiarism rules listed on Ecampus. In the event of proven plagiarism, the group will be assigned a grade of zero for the entire course.

The marking criteria are specified on Ecampus.

Bon travail !