Greedy Algorithm: Optimizing Disk Allocation

By

James Thomas

Design and Analysis of Algorithms: CS627

Colorado Technical University

08/13/2023

Table of Contents

[Introduction 3](#_Toc142854739)

[Greedy Algorithm Pseudocode 4](#_Toc142854740)

[Optimality of Algorithm and Time Complexity 6](#_Toc142854741)

[Options of Brute Force Algorithm and Time Complexity 7](#_Toc142854742)

[References 9](#_Toc142854743)

# Introduction

In the world of multimedia and mobile technology, the challenge of transferring substantial media files from older storage disks to newer disks has become a critical and difficult task for our company. While the initial approach of transferring files randomly is working for now, I am convinced that there exists a more efficient and strategic method to optimize storage space utilization on the new disks. The prospect of achieving such efficiency has motivated us to explore the design of a greedy algorithm that not only streamlines the media transfer process but also minimizes wasted storage capacity across the array of new disks (Schuetz & Caflisch 2008). This endeavor is rooted in the optimization of problems, where the goal revolves around maximizing or minimizing a specific quantity within a set of constraints. In our case, the optimization challenge centers around effectively organizing a collection of “n” media files – each with varying sizes – onto a set of “m” disks, each possessing its storage capacity.

The primary objective here is to minimize the unused storage space on each disk, ensuring efficient resource utilization. This will reduce the overall number of disks required for the storage process. This would also contribute to cost savings across the board. While striving for optimal distribution, the focus will be placed on filling each disk as comprehensively as possible, leaving minimal unutilized space. The pinnacle of success would involve perfectly aligning each disk’s usage, leaving no wasted storage space. In the events where some disks remain unused, we stand to gain, as these surplus disks can be returned for a refund, adding a financial incentive to our optimization goals and milestones. Through the synergistic behaviors of intelligent algorithms, innovative thinking, and the drive to redefine efficiency, our organization is on the brink of transforming the way we manage the storage of multimedia content (Kuhnle, A. (2019).

# Greedy Algorithm Pseudocode

The provided code is written in a programming language that appears to be a pseudocode or a general form of language that can apply to many (Horowitz, et al.1997). The code describes a greedy algorithm for storing 'n' files onto 'm' disks based on file sizes 's' and disk storage capacities 't'. The goal is to place the largest files on the disks in a way that optimizes space usage.

Starting from the function “DiskTransfer”, we take the four parameters: “n” (number of files), “m”(number of disks), “s” (array of file sizes), and “t” (array of disks storage capacities). In sorting files by size, the algorithm starts by sorting the file sizes in a “non-decreasing order”. This is done using the “Heap Sort” algorithm(which is not completely shown in the pseudocode). The sorted file sizes are sorted in the “sortS” array (Sharma, et al. 2008). The Algorithm then creates the “Reverse Map” by associating each sorted file size with its original file size. This map will be used later to determine the original size of a file from its sorted size. After the mapping process, the code then enters the “greedy approach” through the assignment of files onto disks (Schuetz & Caflisch 2008). It iterates each file and attempts to assign the file to the lowest indexed disks that have sufficient storage space. After all, files have been assigned to disks, the algorithm returns the mapping that indicates which file is sorted on each disk. Here's the code with comments simplified for better understanding:

1.) //Stores n files onto "m" disks with file size "s" and disk storage "d"

2.) // This solution simply chooses the largest file to be sorted at each iteration

3.) //stores it on //the lowest indexed disk possible

4.)

5.) DiskTransfer (n,m,s,t)

6.)

7.) //Since we are using greedy approach, it'll make sense to sort out files by size

8.) //here we assume the SortingObject contains a sorted list

9.) SortingObject.sortedList and an

10.)

11.) SortingObject = HeapSort(s);

12.) // Get the sorted list of file sizes

13.) sortedS = SortingObject.sortedList;

14.)

15.) // Create map from sorted file sizes to Original sizes

16.) map\_from\_sortedS\_to s = sorted.reverseMap;

17.)

18.) //In a greedy fashion, assign each file to the lowest index possible

19.) for i := 1 to n do { //iterate through the files

20.) for j := 1 to m{ //iterate through the disks

21.) if sorted[i]<t[j]

22.)

23.) // Add the file to the disk and adjust the maximum storage

24.) map[map\_from\_sortedS\_s[i]] = j //add to the disk

25.) t[j]-sorted[i]; //adjust new maximum storage

26.)

27.) //Since we have sorted the file onto a disk

28.) //successfully, we can break from the inner "for" loop

29.) break;

30.) } end if

31.) } end for

32.) } end for

33.) Return map;

The code describes a greedy algorithm that sorts the files based on their sizes and then assigns each file to the disk with the lowest index that has enough storage capacity. It iterates through the files and disks, to check if the current file can fit into the surplus storage of a disk. If the files can fit the space, the file is assigned to that disk, and the storage capacity of the disk is adjusted. The process continues until all files are assigned. The goal of this algorithm is to efficiently distribute files among disks to minimize wasted space and make the best use of available storage. The code uses “Heap Sort” and greedy assignment to achieve this optimization. The algorithm sorts the files by size, then iterates through each file and assigns it to the lowest-indexed disk that can accommodate it. The greedy approach may not always produce an optimal result. The pseudocode provides a high-level understanding of the algorithm’s steps.

# Optimality of Algorithm and Time Complexity

Here we continue to review the optimality and time complexity of the algorithm. The algorithm's greedy approach involves sorting the files by size and then placing each file onto the disk with the lowest available storage that can accommodate it. This approach often leads to good solutions by maximizing space utilization (Kuhnle, A. 2019). Consider a scenario where the available disks have varying storage capacities, and there exists an arrangement of files that would fill in all disks, leaving no unused storage. The algorithm may not be able to achieve this optimal solution if the order of file placement does not align with the specific disk capacities (Schuetz & Caflisch 2008). Below in Table 1, we present an example, to reflect that though this is a good option, it is not the most optimal option or in other words, the perfect option.

Table 1.

|  |  |  |  |
| --- | --- | --- | --- |
| Greedy Algorithm | | Optimal Algorithm | |
| Disc Size (d) | File Size (f) | Disc Size (d) | File Size (f) |
| 7 | 5+2 | 7 | 2+2+3 |
| 3 | 3 | 3 | ~ |
| 5 | 2 | 5 | 5 |

In this example, a comparison between a greedy algorithm and the optimal algorithm or “Perfect algorithm” is shown with the output of results after calling the functions. On the left, we can see that the disc sizes of the greedy algorithm are the same as the sizes of the Optimal algorithm and in the same order. The array to sort in this example is: “5,3,2,2”. The greedy approach is to fill in the order of values that come next, as seen in Table 1. All the disks have been used, but with additional space left on the disk with the size of 5, whereas in comparison to the disks in the optimal algorithm, there is a disk unused, all the files were sorted into the proper disk and the disk with the size of 3 can be returned for a refund.

While analyzing the time complexity of the algorithm in the pseudocode, we consider the “HeapSort” time. Heap Sort takes O(n log n) time, so we must consider this when reviewing the time analysis for the algorithm as a whole (Sharma, et al. 2008). The two loops including the outer loop iterate over “n” files and the inner loop iterates over “m” disks. In the worst case, each file might need to be compared with all disks, therefore the time complexity of the loops is O(n • m). The dominant variable in the time complexity is the forming step (O(n log n)), which is followed by the loops (O(n • m)) (Sharma, et al. 2008). Therefore the time complexity of the algorithm is O(n log n + (n • m). The performance of the algorithm will depend on the specific file and disk configurations.

# Options of Brute Force Algorithm and Time Complexity

Regarding Brute Force algorithms, the brute force or exhaustive search method would not be nearly as optimal. The time complexity would be significantly higher compared to the greedy algorithm provided (Marx & Pilipczuk, 2015). Here we will review this time complexity and why it is not an optimal choice when compared to the greedy algorithm. In a brute-force approach, we would need to explore all possible combinations of file placements on disks to determine the arrangement that minimizes wasted space. In doing this, we would ultimately create openings for an error in wasted time. This involves trying every possible mapping of files to disks, which results in a time complexity of O(nm), where “n” is the number of files and “m” is the number of disks (Marx & Pilipczuk, 2015).

Each of the “n” files can be placed onto any of the “m” disks, leading to a total of “ nm ” possible combinations, which results in O(nm). The brute force approach also explores all possible ways to allocate files to disks which results in an exponential growth in time complexity as the number of files and disks increases. As the sizes of our variables “n” and “m” grows, the number of combinations grows exponentially. This could lead to impractical computational requirements which makes the brute force approach highly inefficient and could never be practical in use for the goals we are trying to accomplish. In comparison to the greedy algorithm, the time complexity is much different. The time complexity for the greedy algorithm is easier to manage than an exponential complexity like the brute force approach. While the greedy algorithm doesn’t guarantee an optimal solution in all cases, it offers a reasonable case of efficiency and finding good solutions for many scenarios, including our endeavor to file to disk allocation.

# References

Horowitz, E., Sahni, S., & Rajasekaran, S. (1997). *Computer algorithms C++: C++ and pseudocode versions*. Macmillan.

Kuhnle, A. (2019). Interlaced greedy algorithm for maximization of submodular functions in nearly linear time. Advances in neural information processing systems, 32.

Levitin, A. (2016). Introduction to the Design and Analysis of Algorithms (3rd ed.). Pearson Learning Solutions. <https://bookshelf.vitalsource.com/books/9781323417638>

Marx, D., & Pilipczuk, M. (2015, November). Optimal parameterized algorithms for planar facility location problems using Voronoi diagrams. In Algorithms-ESA 2015: 23rd Annual European Symposium, Patras, Greece, September 14-16, 2015, Proceedings (pp. 865-877). Berlin, Heidelberg: Springer Berlin Heidelberg.

Schuetz, P., & Caflisch, A. (2008). Efficient modularity optimization by multistep greedy algorithm and vertex mover refinement. Physical Review E, 77(4), 046112.

Sharma, V., Sandhu, P. S., Singh, S., & Saini, B. (2008). Analysis of modified heap sort algorithm on different environment. International Journal of Electrical and Computer Engineering, 2(6), 1143-1145.