Implementation of Algorithm to solve Two-Dimensional Bin Packing Problem

Team 4

Fernando Alcalá Casas - 1942695

Introduction

This document presents a heuristic solution to the bin packing problem, more specifically to its two-dimensional variation. For this we deal with the description of the problem and its context of complexity, we deal with the description of the heuristics, the logic and the dialectical process that allows us to reach a result for the instance. The algorithm is also addressed through pseudocode, to follow the workflow of the algorithm and how it arrives at a result..

Problem description

The two-dimensional bin packing problem is an optimization problem, in which items of different sizes must be stored in bins, these bins can also be of different sizes or of a fixed size. The goal of the problem is to optimize the number of containers used. Use the fewest number of bins to store as many items as possible without overlapping.

2D-BPP is not only used for the storage of items in containers, but also to determine the most optimal way to cut the surface of an item on a material. In essence the problem is the same, place a number of items on a surface to optimize space used.

The problem is classified as NP-hard, and NP-complete, the latter indicating that the optimal solution of the problem can be computed in polynomial time and that solution can be checked by brute force.

Heuristic description

The heuristics proposed in this document allow the organization of the items given by the problem in the given containers. For this we start by ordering the items given by the problem instance, this ordering leaves us with a list of items ordered from highest to lowest. In the same way, but without ordering, we store the containers given by the instance in a list of objects.

From there we go through the list of items to order them one by one in the containers: In this process, we will go to the first container of the list of containers, and we will check the available spaces and store their coordinates in a list. In case the sum of the available spaces is less than or equal to the area of the item, we start looping through the list of coordinates, and for each one we calculate the available coordinates that would be needed to store the item and save those calculated coordinates. We loop through the list of computed coordinates and if each of them is available, we store the element.

In case the element cannot be stored, we go to the next container and repeat the insertion process.

Once the item is inserted, we repeat the process of finding space for it in the list of containers from the first, since the list of items is ordered in descending order, the probability that an item can be stored in a container with little space increases as we advance, in this way the waste of space is reduced.

Algorithm

Input:

Li = List Items.

Lb =List bins.

Output: Lis = List Items Sorted

Initialization

Li « Instance

Lb « Instance

Sort items of Li

Create bins list where each one is a bidimensional char matrix = Lb_char_matrix

Foreach bin in Lb

Initialize bin char matrix in zeros in function of bin

Push bin char matrix to Lb_char_matrix

End for

For i in length of Li

Inserted = Boolean variable that indicates if the item has been inserted

Inserted = false

Foreach bin in Lb

If Inserted == true

Break the foreach of the bins in Lb

End if

Look for available positions in the bin matrix of chars and save it in a list

If available positions are less than item area

Continue to the next bin.

End if

Foreach available position in List of available positions

If Inserted == true

break

End if

Calculate needed positions available to insert the item

If all the needed positions are available

Insert item

Inserted == true

End if

End For

End for

End for

Computational Results:

After running the implementation of the program in the different instances, we obtained favorable results in most of the instances.

Given the way in which the problem was approached, the increase in computational time depends on two factors, the number of items and containers, and the size of each element, given that the elements are treated on character arrays, the increase in the size of the elements implies an n^2 increase in the insertion time of each element.

First, we need to talk about the instances we're working with. Instances can be retrieved from people.brunel.ac.uk. In the particular case of the two-dimensional bin packing problem instances are retrieved from: binpacktwo.xls (live.com)

The resource has three categories of problems, which contain items and containers of different sizes. Each category has five quiz problems.

According to the description of the problems provided by the instance, the test problems were built without a known solution, but with the item dimensions sufficient for each one to be accommodated within the containers. The dimensions of each item were created randomly.

Description of the computer equipment:

» 8th Generation Intel i7 Processor at 1.8 – 2.0 GHz

»12GB of DDR4 RAM at 2400Mhz

» HDD Mechanical Hard Drive

So, is time to talk about resutls. In the table at T1 we can find by columns the name of the evaluated instance, the number of items that make it up, the area of the item with the largest area, the number of containers in the instance, the number of items inserted by the algorithm, the bins used by the algorithm, the time it took to execute the algorithm and a ΔBins that tells us the difference between the bins given by the instance, and the bins used by the algorithm, here, negative values of ΔBins mean that we use fewer bins than those obtained in the instance.

Instance	Items of the instance	Largest item area	Bins of the instance	Items inserted by the algorithm	Used Bins by the algorithm	Time (s) of execution	ΔBins
M1a	100	81	10	100	9	2.98	-1
M1b	100	72	16	100	10	2.81	-6
M1c	100	72	16	100	9	2.77	-7
M2a	100	784	18	100	11	57.41	-7
M2b	100	784	18	100	12	69.11	-6
M2c	100	700	18	100	12	68.22	-6
M3a	150	841	20	150	13	150.98	-7
M3b	150	754	20	150	13	164.57	-7
M3c	150	700	20	150	13	113.60	-7
Ng1	10	21	10	10	4	0.05512	-6
Ng2	17	27	10	17	4	0.08089	-6
Ng3	21	32	10	21	4	0.10326	-6

Results table T1

Well, time to talk about the results. We can see that the increase in the execution time of the algorithm is given by two factors, the number of items, and the size of the elements and containers. To observe this, we record the size of the largest item and thus give us an idea of the workload to which we submit the algorithm.

In all the instances on which we tested the algorithm, we managed to insert 100% of the items given by the instance, and in the same way, for all the tested instances, the algorithm used a smaller number of containers than those given by the instance.

Conclusions

Developing a solution for this problem results in the need to find a way to address the problem and represent the elements treated in the instances through computational entities and develop a logic that allows us to perform the calculations through these entities (in this case arrays). necessary to verify the availability of the available spaces in the containers, the way to address the problem affects the complexity of the code and the processing times of each instance, adding an extra factor of computational complexity that does not depend just of the elements processed , but also an inherent sub-step of the solution process, as in this case it is the traversal of arrays treating each memory address as a potential space for the item.

How to improve it, the algorithm could add a data processing process, which seeks to lower the order of magnitude of the size of the elements and containers, within what is allowed by the discrete nature of the grid on which we work.

References

http://people.brunel.ac.uk/~mastjjb/jeb/orlib/files/

https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0229358#:~:text=Introduction-,The two-dimensional bin packing problem (2D-BPP),i and height hi.