Optimizing Profit in 3D Container Packing with Boxes and Pentominoes

Amir Mohseni, Maria Fanou, Zhili Yang Wu, Gilles Van der Spiegel, Sirbu Bogdan, Koung Myat Hein

Abstract **3**
Introduction **3**
Methods **4**
Description of sets and the knapsack containers: 4
Boxes 4
Pentominoes 4
Truck Container 4
Random Algorithm 5
Greedy Algorithm 5
Genetics Algorithm 5
Algorithm X 6
Implementation **7**
Databases 7
Algorithms 7
UI 8
Results **9**
Discussion **10**
Conclusion **10**
References **11**

Abstract

This research introduces a computer application to optimize cargo space packing in trucks. One of the main goals of most transportation companies is to optimize their available cargo space. The goal of this paper is to research the 3D knapsack problem and create a program to solve it with a user friendly interface. The program will then be used to solve 2 similar, but different problems. Firstly filling a cargo space with 3 cuboid parcels. For the second problem we will be filling the same cargo area with 3 different 3D pentominoes. We have made use of these 4 algorithms in the goal of finding the most optimal solution: Random, Greedy, Genetic and Algorithm X. The algorithms always come up with similar, but different solutions. Since we never found a 'perfect' solution we have concluded that there is no correct answer, but rather a multitude of solutions that approach the theoretical maximum score.

This research outlines an application to optimize the packing of parcels into a container. We developed ways to solve mathematical and programming problems, such as the knapsack and total cover problems. Using these solutions, we have developed algorithms to perform the solutions and display the result in a 3-dimensional render. The application includes multiple user interface elements to change the parameters of the algorithms. Settings such as the choice of algorithms (Random, Greedy, Genetic, and ALgorithm X), parcel type (Boxes, Pentominoes), values of parcels, operation type (maximize coverage, maximize score), and model slicing.

This document contains explanations on our algorithms, our implementations of the algorithms, our implementation of the application, the data we collected, and our analysis of said data.

Introduction

The knapsack problem is a combinatorial optimization problem where, given a set of non-divisible objects, each with a weight and a value, determine which items to include in the collection so that the total weight is less than or equal to a given limit and the total value is as large as possible [7].

Our primary goal is to develop a computer application to solve a three-dimensional knapsack problem, a challenge faced by logistics companies wanting to optimize their transportation operations. Since our application is aimed at three-dimensional problems, our metrics will be volume and value, instead of weight and value. Our application is designed to maximize the total value of transported parcels within our given cargo area. We will be covering 2 variations. The first one with 3 cuboid shaped parcels, each with a different volume and value. The second case with 3 pentomino-shaped parcels also with their own unique volume and value.

Our research aims to answer the following questions:

A) To what extent is it possible to achieve the full coverage of a 3D container with a set of different sized parcels?
B) What is the maximum value we can store in the grid?

The first question was solved with a mathematical approach, saving us valuable time instead of experimenting with algorithms and other approaches to reach this goal. The second question was handled with 4 different algorithms (Random Search, Greedy Search, Genetic Algorithm, Algorithm X). We then compared the results to come to a conclusion: Determine the most optimal / precise way of solving the knapsack problem.

Methods

Description of sets and the knapsack containers:

Boxes

For the first part of this research's problem, we used three different cuboid shapes: A, B, and C. Each with a different size and value. Specifically, their sizes were 1.0 x 1.0 x 2.0, 1.0 x 1.5 x 2.0, and 1.5 x 1.5 x 1.5. Their values were 3, 4 and 5, respectively. For practical reasons we doubled all size values, for both the grid and the parcels, so we ended up with parcels sized 2.0 x 2.0 x 4.0, 2.0 x 3.0 x 4.0 and 3.0 x 3.0 x 3.0. Each parcel is represented by a 3D array, with the indexes representing the dimensions of the parcel, a unique color and value, as well as every rotation of the piece.

Pentominoes

For the second part of the research, we had to fill the truck with pentominoes, more precisely the pentominoes shaped as **L, P, and T.** For better visualization of the pieces, we gave every piece a unique ID that has a unique color that helps us find any boundary bugs or blocks merging. The pentominoes are stored in a dataset where the first 3 values correspond to the 3D representation of each rotation, the next 2 values correspond to the value and the color of the piece. Each pentomino is made up of 5 cubes of dimensions 0.5 x 0.5 x 0.5, these values are also doubled resulting in cubes with dimensions 1.0 x 1.0 x 1.0.

Truck Container

The truck is represented by a 3D array 2.5 m wide, 16.5 m long, and 4 m tall. The truck was initially filled with 0s (representing empty spaces) and it gets updated every time a piece is placed. Same as with the cuboids and pentominoes, we also doubled the dimensions of the truck, rounding them up to 5.0 x 33.0 x 8.0.

Set	Shape	Value	Dimensions (m)
Cuboids	A	3	2.0 x .20 x 4.0
	B	4	2.0 x 3.0 x 4.0
	C	5	3.0 x 3.0 x 3.0
Pentominoes	L	3	5 cubes of 1.0 x 1.0 x 1.0
	P	4	

	T	5	

Table 1. Final values of dimensions and weights of the set of objects.

Random Algorithm

This algorithm randomly places each piece in the first empty space. It iterates over the grid's empty cells and tries to find a valid placement for the current piece. If it cannot find a valid placement, the algorithm stops and accepts the placements it made so far as the solution.

Greedy Algorithm

For the greedy algorithm, we used two different approaches. The first approach prioritizes the higher-value pieces. The pieces are sorted in descending order based on their value so that the most valuable ones are at the beginning of the list. The algorithm finds the first empty cell with the minimum coordinates (The coordinate that minimizes the tuple(x, y, z)) and iterates over the list of pieces, and finds the first piece that can be placed in the first empty space and places it in the grid. The algorithm stops when it cannot find any valid placement for any of the remaining pieces and returns the current grid. The second approach also finds the first empty cell but instead of prioritizing over the values of each piece, it iterates over their density, which means dividing their value over their volume.

Genetics Algorithm

The Genetic Algorithm is a meta-heuristic motivated by the evolution process and belongs to the large class of evolutionary algorithms in informatics and computational mathematics. These algorithms are frequently used to create high-quality solutions to optimize and search concerns by focusing on bio-inspired operators such as selection, convergence, or mutations [6]. The algorithm creates an initial population of genomes. Each genome is a random sequence and each element is a random piece and its rotation, picked from the databases. We run the algorithm for 200 generations by using the methods below to generate new generations:

- Fitness: The fitness of a sequence is calculated by placing each of the elements in the container by going from left to right and placing each piece in the first empty space that has the smallest coordinate value. To calculate the score, we assume that if we want to find the maximum coverage, we use the volume of the blocks for their values and if we want to find the highest score, we use the input values that are set to 3, 4, 5 by default for the pieces A, B, and C respectively.
- Crossover: We randomly find two parents from our current population with probability proportional to their fitness scores.
- Mutation: Each child would have a small probability of having one of its elements changed to another random number in the same range. We set the initial probability to 0.01
- Sorting: After creating each generation, we sort the population based on their fitness scores in descending order. Therefore, the first element would be the fittest genome in that population.

When the iteration ends, we use the fittest genome as our answer since it would get the highest score.

We also used another approach which uses two sequences for each genome. One for the order of the pieces and another one for the rotation of each of them. This would decrease the number of different sequences we can get which would help us use a higher population count and generation size. We use the same methods for placing the pieces. This approach helped us run the program using a bigger population size and generation size.

Algorithm X

Dancing links (DLX) is a method used to add and remove a node from a circular doubly linked list. Its primary advantage lies in its efficient implementation of backtracking algorithms, specifically Knuth's Algorithm X for solving the exact cover problem. Algorithm X is a recursive, nondeterministic, depth-first, backtracking algorithm that aims to find all possible solutions to the exact cover problem [2]. In scenarios where the search tree has numerous levels, Algorithm X begins to outperform alternative pentomino-placing procedures. By employing dancing links to execute the "natural" algorithm for exact cover problems, Algorithm X proves to be an effective approach for enumerating all solutions to such problems. Although it is nearly as fast as specialized algorithms tailored to solve specific problem classes, such as pentomino packing, in smaller cases, Algorithm X demonstrates even greater speed in larger cases due to its ordering heuristic. As computer processing power continues to advance, we are constantly tackling increasingly larger cases. Our implementation of dancing links allows us to input the database of Cuboids/Pentominoes and their corresponding values, resulting in a grid representation and a score. The algorithm's functionality can be succinctly summarized through a series of bullet points in our code. Generally, this algorithm is used to find exact solutions. However, changing it to accept semi-solutions will help us to find some estimations for maximizing the scores as well. Since, in some cases, the algorithm would run for a long time due to its exponential nature so, we had to put a time limit on the program for it to stop. This time limit is 30 seconds by default.

1. Class Structure:

Cargo Best Class:

Represents the main cargo space optimization algorithm.
Utilizes a modified version of Algorithm X.
Provides options for custom parcel amounts and values.
Implements time limits for algorithm execution.

N Class:

Defines polymorphic classes for the building blocks of the DLX data structure.
Includes Node, HNode (Header Node), and RNode (Row Node) classes.

DLX Fast Class:

Implements a highly optimized dancing links structure (DLXFast).
Represents constraints and possibilities of parcel placements in the cargo space.
Provides methods to add rows and initialize headers.

2. Algorithm Execution:

solvePacking Method:

Initiates the cargo space optimization algorithm.
Allows setting a time limit for algorithm execution.
Utilizes recursive backtracking to find a better solution.
Employs heuristics to efficiently choose the best constraint to fulfill.

3. DLX Fast Structure:
InitHeaders Method:
 Initializes header nodes for constraints and connects them to form a circular linked list.
 addRow Method (in DLXFast):
 Adds a row to the DLXFast structure representing a parcel placement.
 Connects nodes horizontally and vertically to the existing structure.
4. Node Classes (N Class):
Node:
 Represents a basic node in the DLX structure.
 Contains information about its value, neighbors, header node, and row node.
HNode (Header Node):
 Extends Node and represents a header node for a constraint.
 Keeps track of the number of possible rows that can fulfill the constraint.
RNode (Row Node):
 Extends Node and represents a row node, corresponding to a parcel placement.
 Stores information about the parcel, such as color, coordinates, space, and price.

Implementation

Databases

Our application implements two databases respective to the two parcel types: boxes and pentominoes. These databases allow for the retrieval of all variants of the parcels (i.e. mirrors and rotations).

We hard-coded a database containing all the parcels and their rotations, the number of each rotation varies for each set of parcels. Each item is represented by a 3D array in this database, which is also represented by integers of 0 and its ID (0 for empty spaces and piece ID of either 1, 2, or 3), and ordered sequentially with a corresponding number to identify it. This database is used in the implementation of each algorithm, having each calling items contained in this.

Algorithms

The algorithms retrieve all the variants of the parcels and attempt to fit them into the container. The algorithms are all capable of running in two modes: either in maximum coverage mode, where it will attempt to fill the container as completely as possible. Or in maximum value mode, where it will attempt to maximize the value of items in the container.

For the implementation of the Random search method, we used the Java Random method to select a piece from the data set and place it in the grid, following the procedure described in the method section. The algorithm makes use of the Random function to generate a random number and would call the corresponding item that is identified by the generated number.

For the Greedy algorithm, we first used Java Sort, to sort the values of the parcels in a descending order, and later their density. Then, we applied the method, which would place the parcel with the highest value the way it was explained previously.

For the Genetics one algorithm we used the following parameters: generation size, population size and the probability of mutation. The maximum generation and population size that we have set up is 50 and 5000 respectively, along with the 0.001 probability of mutation. The parameters for Genetics two were 150 for generation size and 200 for population size. We used Java built-in functions to perform the crossover of genomes based on the fitness that we have described earlier.

UI

Rendering

Once a solution is found by the algorithms, the renderer renders the result. The resulting container contains parcels separated by their distinct colors (red, green, and blue). The container can be manipulated by rotating and zooming in or out. The container can be further manipulated by the menu.

Menu

We have a control panel with a drop down menu to select which one of our six algorithms to use. Under that we have another drop down menu where the user can select to use boxes or pentominoes. Next there are three boxes to change each piece's respective value. Under that the user can decide to maximize coverage or maximize score.

Lastly we have 'slicing' controls at the bottom. Each slider changes the size of the rendered area on its respective axis (X, Y or Z), allowing the user to look inside of the model.

Experiments

In order to determine the efficiency of our algorithms and to answer our research questions, we have conducted the following experiments:

We ran each algorithm a number of times, using both the A, B and C parcels and then the L, P and T pentominoes. We tested with the maximum coverage model. Then we used the maximum score model. We recorded the results of each simulation, calculated the average score and compared them to evaluate their overall efficiency.

So, in the end, we ended up with the following number of simulations:

A) 60 times for the Random algorithm, using the cuboid and pentomino parcels, to find the maximum cover.
B) 60 times for the Random algorithm, using the cuboid and pentomino parcels with their respective values.
C) 1 time for the Greedy 1, Greedy 2, and Algorithm X. (Deterministic)
D) 90 times for the Genetic 1 algorithm, using the cuboid and pentomino parcels using the set of values {3, 4, and 5} and another time for finding the maximum cover.
E) 40 times for the Genetic 2 algorithm, using the cuboid and pentomino parcels using the set of values of {3, 4, and 5} and another time for finding the maximum cover.

Results

	Random	Greedy 1	Greedy 2	Genetic 1	Genetic 2	AlgoX
Boxes	70%	98%	98%	79%	94%	99%
Pentominoes	77%	84%	83%	62%	84%	100%

Table 2. Average volume occupied in the container.

	Random	Greedy 1	Greedy 2	Genetic 1	Genetic 2	AlgoX
Boxes	156	230	230	183	223	228
Pentominoes	806	995	1105	643	976	1188

Table 3. Average score obtained from the packing.

Discussion

With the algorithms set to maximize the occupied space in the container, Greedy 1, and Algorithm X provided the best coverage, almost achieving a complete fill. However, we could never fully fill the container with boxes. For pentominoes, we managed to fully fill the container with Algorithm X, but the rest of the algorithms provided a less than satisfactory result.

With the algorithms set to maximize the score, Greedy 1 and Greedy 2 managed to give the highest score at 230, far ahead of the other algorithms. For pentominoes, Greedy 2 and AlgoX gave the highest scores, with Greedy 1 and Genetic 2 coming close behind.

With these results, we learned that there is not a single best algorithm for all scenarios. Changing the parcel type (pentominoes, boxes), fill priority (maximize score, minimize empty space), would require different algorithms to achieve the best output.

Conclusion

Our exploration of the 3D Knapsack problem allowed us to find the best optimization for using cargo spaces. Implementing four algorithms: Random, Greedy, Genetics, and Algorithm X gave us a wide range of knowledge for the challenge of finding the best method to call for maximizing the value/space-filling results.

The outcome of our experiments has shown us that Greedy 1 and Algorithm X had the best performance in filling the cargo space with boxes but none of the algorithms managed full coverage. We were able to reach a perfect fill with pentominoes using Algorithm X!

Shifting the focus to maximizing the score brought Greedy algorithms to the forefront. Both Greedy 1 and Greedy 2 surpassed other approaches in both box and pentomino scenarios. Genetic Algorithm and Algorithm X also gave competitive results, demonstrating the effectiveness of these meta-heuristic approaches in the context of the 3D knapsack problem.

Importantly, our research got a spectrum of solutions approaching the theoretical maximum score, instead of having a "perfect one". This observation emphasizes the complex nature of the 3D Knapsack Problem, where different algorithms excel in different aspects.

From our experiments and data, we have concluded that it is not possible to fill the cargo space with the boxes A, B, and C. However, we are able to completely fill the cargo space with pentominoes L, P, and T using Algorithm X and maximum coverage mode. We have attached our mathematical proof that it is impossible to fill the cargo space with the boxes in the appendix.

Assigning points to the boxes, we are able to achieve a highest score for boxes A, B, and C of 230 points and 1188 points for pentominoes L, P, T.

Algorithm X yielded the best results across the board. In certain situations it gets beaten by our greedy approach for boxes but is by far the best in general. It managed to fill 99% and 100% of the cargo space with boxes and pentominoes respectively. Both Greedy algorithms barely got a higher score than Algorithm X with the boxes, averaging 230 points when Algorithm X averaged 228. With pentominoes, Algorithm X is by far the best, reaching 1188 points!

References

[1] Balodi, T. (n.d.). *A complete guide to solve knapsack problem using greedy method | Analytics Steps*. https://www.analyticssteps.com/blogs/complete-guide-solve-knapsack-problem-using-greedy-method

[2] Knuth, D. E. (2000). Dancing links. *arXiv (Cornell University)*. https://arxiv.org/pdf/cs/0011047

[3] *Dube, E. (2006). OPTIMIZING THREE-DIMENSIONAL BIN PACKING THROUGH SIMULATION.*

http://pdfs.semanticscholar.org/bb99/86af2f26f7726fcef1bc684eac8239c9b853.pdf

[4] Maarouf, W. F., Barbar, A., & Owayjan, M. (2008). A new heuristic algorithm for the 3D bin packing problem. In Springer eBooks (pp. 342–345). https://doi.org/10.1007/978-1-4020-8735-6_64

[5] Mohsin, R. R. (2022). Genetic Algorithm: A study survey. *Iraqi Journal of Science*. https://doi.org/10.24996/ijs.2022.63.3.27

[6] Tanweer Alam. Shamimul Qamar. Amit Dixit. Mohamed Benaida. " Genetic Algorithm: Reviews, Implementations, and Applications.", International Journal of Engineering Pedagogy (iJEP). 2020.

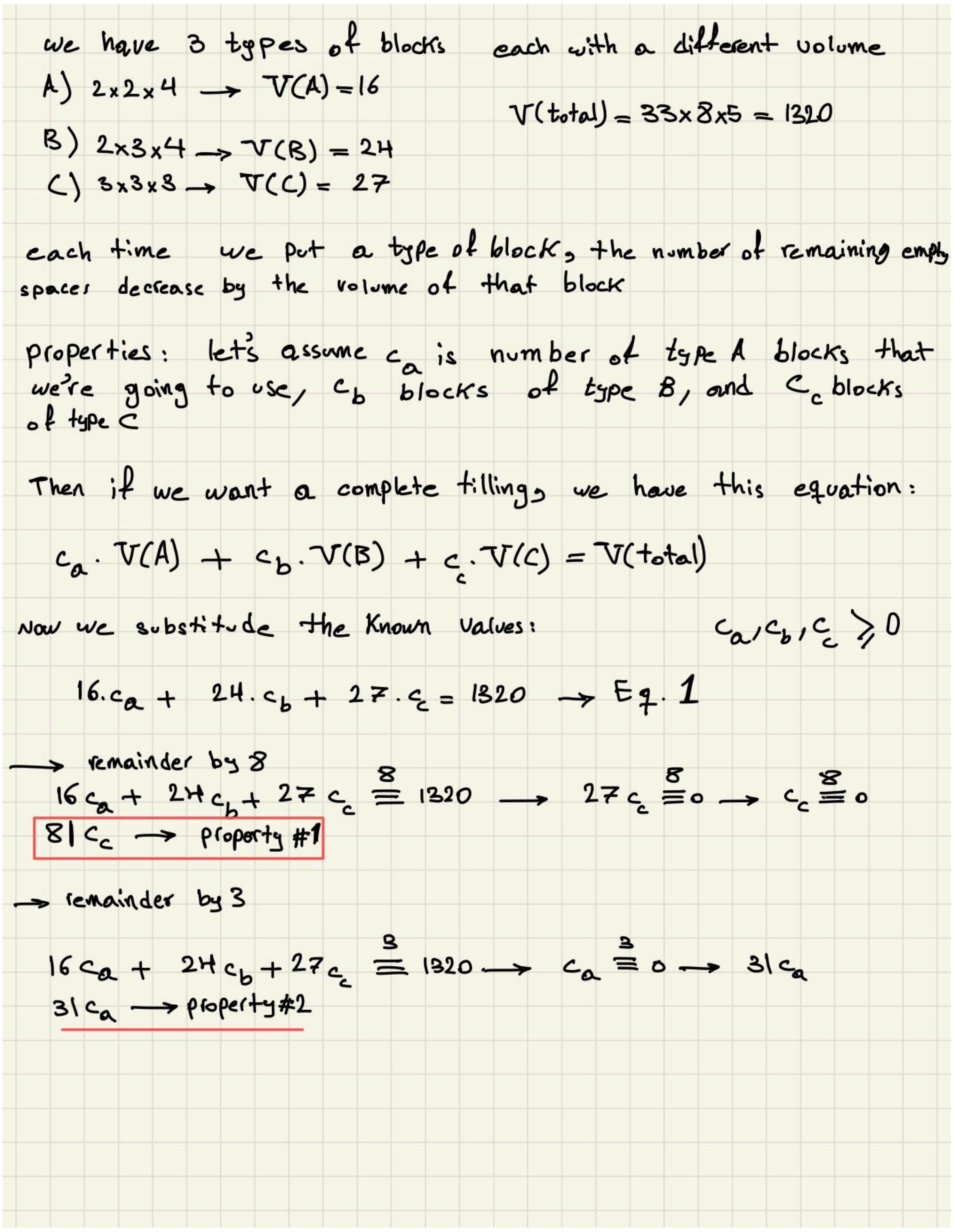
[7] Wikipedia contributors. (2024, January 2). Knapsack problem. Wikipedia. https://en.wikipedia.org/wiki/Knapsack_problem

Appendix

Spreadsheet for results of experimentation:
https://docs.google.com/spreadsheets/d/1O4j0K9UKrZAqcOeA6fXR7VfpnevTfG6JC7nRyljI43g/edit?usp=sharing

Proof for finding no solutions for filling the container with boxes of type A, B, or C:

Proof.pdf



we have 3 types of blocks each with a different volume

A) $2 \times 2 \times 4 \rightarrow V(A) = 16$

$V(\text{total}) = 33 \times 8 \times 5 = 1320$

B) $2 \times 3 \times 4 \rightarrow V(B) = 24$

C) $3 \times 3 \times 3 \rightarrow V(C) = 27$

each time we put a type of block, the number of remaining empty spaces decrease by the volume of that block

properties: let's assume c_a is number of type A blocks that we're going to use, c_b blocks of type B, and c_c blocks of type C

Then if we want a complete filling, we have this equation:

$$c_a \cdot V(A) + c_b \cdot V(B) + c_c \cdot V(C) = V(\text{total})$$

Now we substitude the Known values: $c_a, c_b, c_c \geq 0$

$$16 . c_a + 24 . c_b + 27 . c_c = 1320 \rightarrow Eq. 1$$

$\longrightarrow$ remainder by 8

$$16 c_a + 24 c_b + 27 c_c \overset{8}{\equiv} 1320 \longrightarrow 27 c_c \overset{8}{\equiv} 0 \longrightarrow c_c \overset{8}{\equiv} 0$$

$\boxed{8 \mid c_c \longrightarrow \text{property \#1}}$

$\longrightarrow$ remainder by 3

$$16 c_a + 24 c_b + 27 c_c \overset{3}{\equiv} 1320 \longrightarrow c_a \overset{3}{\equiv} 0 \longrightarrow 3 \mid c_a$$

$\underline{3 \mid c_a \longrightarrow \text{property\#2}}$

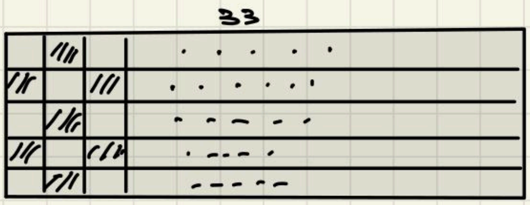
we can also find more limitations for blocks of type c:

we can have at most $\left\lfloor \frac{5}{3} \right\rfloor \times \left\lfloor \frac{8}{3} \right\rfloor \times \left\lfloor \frac{33}{3} \right\rfloor = 22$ blocks of type c. $\longrightarrow$ $c_c < 22$ property #3

Now using property #1 and #3 we can assume that:
$c_c \in \{0, 8, 16\}$

Now, we will prove that $c_c \geqslant 8$:

we can assume that our container is made of 8 slices of $33 \times 5 \times 1$. look at any of these slices as a 2D grid and color the cells like a checkerboard. Since 33×5 is odd, the number of black cells is not equal to the number of whites cells. Now, each block that we use to cover this grid uses two of its dimensions to cover a rectangle in this 2 dimensional grid.



The options for dimensions of the rectangles are:

$2 \times 2 \times 4$: $\{2 \times 2, 2 \times 4, 4 \times 2\}$

$2 \times 3 \times 4$: $\{2 \times 3, 2 \times 4, 3 \times 4, 3 \times 2, 4 \times 2, 4 \times 3\}$

$3 \times 3 \times 3$: $\{3 \times 3\}$

If you look at the sets, blocks of type A and B will produce even rectangles meaning the rectagles have equal number of white and black cells. So if we want to have different number of black and white cells, we have to use at least 1 block of type C $\longrightarrow$ $c_c \geqslant 1$ property #4

Now, using properties #3 and #4 we have:
$c_c \in \{8, 16\}$ property #5

we Know that for all of the 33×5 slices we need odd number of 3×3 blocks. Now this means that the slices of 33×8 and 5×8 s also contain these 3×3 s and since these slices are even, we need even number of 3×3 s for them.

33
5
at least 1

8 slices $\longrightarrow \left\lceil \frac{8}{3} \right\rceil$ 3×3×3 $\longrightarrow$ 3 different 3×3×3 needed and since

they also occupy a 3×3 in 33×8×1 and 8×5×1 slices, we need

in each slice that uses a 3×3.

This means that for every 5 8×5 we need at least 2 3×3 s. we can calculate a lower bound.

$\left\lceil \frac{33}{5} \right\rceil \times 2 = 6 \times 2 = 12 \longrightarrow$ the number of 3×3×3 s should be 16

Now we can calculate the possible combinations of (c_a, c_b, c_c) which only gives us 10 combinations. Now that we have limitation for the numbers we can use DLX to solve this problem. However, we didn't find an answer which shows there is no answer.