Unbounded rucksack problem in the context of 2 dimensional space

Piotr Pilis

Karol Kłosowski

Łukasz Niedziałek

Table of content

Problem description

The rucksack problem (or knapsack problem) is a problem in combinatorial optimization.

Given a set of blocks, each with a size and value, determine blocks to include in a board so that they fit inside given board and the total value is as large as possible.

Algorithms for this problem can be classified into two groups:

1. one-phase algorithms directly pack the items into the finite size bins
2. two-phase algorithms start by packing the items into a single strip, i.e., a bin having finite width and infinite height. In the second phase, the strip solution is used to construct a packing into finite bins.

Most of the algorithms are of greedy[1] or heuristic type[2].

Most of the approaches are shelf algorithms, i.e., the bin/strip packing is obtained by placing the items, from left to right, in rows forming levels (shelves). The first shelf is the bottom of the bin/strip, and subsequent shelves are produced by the horizontal line coinciding with the top of the tallest item packed on the shelf below.

In our case we are interested in exact[3] algorithm, so we won't consider greedy, heuristic (and metaheuristic) algorithms.

Unfortunately rucksack problem is NP-complete, thus there is no known algorithm both correct and fast (in polynomial-time) in all cases; so our goal was to develop a brute force algorithm solving this problem and add to it some improvements (keeping exactness) to speed up this process.

---

[1] **Greedy algorithm** - the one that looks for optimal solution of some local part of the problem which we are currently solving, which may not be the best option in the long run.

[2] **Heuristic algorithm** - the one that can be described as follows: "usually works in the real conditions but we can't prove it".

[3] **Exact algorithm** - the one that always gives correct answer

Solution description

Base algorithm idea

General idea of this solution is to go through each combination of subsets of blocks and check if given subset can be fitted into a board. If it can and its total value is bigger than currently held, then mark it as the biggest and go to another combination.

Pseudocode

```
VAR currently_best_value = NULL

FOR EACH subset OF blocks {
     IF (subset CAN_BE_FITTED_INSIDE board) {
          IF (subset.total_value > currently_best_value) {
               currently_best_value = total_value
          }
     }
}

RETURN currently_best_value
```

How we generate each subset

In order to generate each subset we have to create each possible combination of blocks. We do that by calculating the total number of possible combinations (2^n). Then we iterate from 0 to that number, which produces all possible (different and not doubled) integers in needed range. Then for each iteration we look at binary representation of produced number and for each ‘1’ we add corresponding block, from lost of blocks, to new combination.

Pseudocode

```
VAL possibleSubsets = 2.POWER(blocks.length)
VAR counter = 0

WHILE (counter < possibleSubsets) {
      VAL subset = LIST<INT>()
      VAL binaryCounter = counter.TO_BINARY()

      VAR j = 0
      WHILE (j < blocks.length) {
            IF (binaryCounter[j] == 1){
                  VAL correspondingBlock = blocks[j]
                  subset.ADD(correspondingBlock)
            }

            j++
      }

      counter++

      // new subset is ready and we can do something with it
}
```

How we generate permutations

It is not mentioned in ‘Base algorithm idea’, but we also have to generate permutations of given combinations (‘why’ will be further explained in proof of correctness).

In order to generate permutations we use recursive function, which takes 3 input parameters: LIST (to be permuted), INT (starting index **S**), INT (ending index **E**). Function terminates its node if **S** is equal to **E**. Main mechanism works by looping from **S** till **E** and for each iteration we swap **S**th object with **E**th one, call this function with **S** increased by one (make recursion), and swap those objects back.

Pseudocode

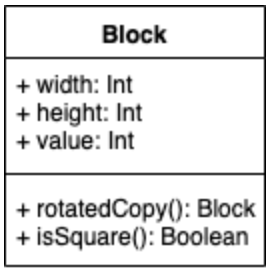
```
FUNCTION permute (LIST<T> list, INT s, INT e) {
      IF (s == e) {
            // permutation is ready (stored in list variable passed
            // to function)
      } ELSE {
            LIST<T> copy = list

            SWAP copy[s] WITH copy[e]
            permute(copy, s + 1, e)
            SWAP copy[s] WITH copy[e]
      }
}
```

How we check if given subset fits inside a board

This is the most complicated part of our solution and it will be better to show class diagrams (of used model classes) first.

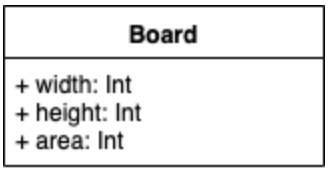
Block

Block class represents a single block that we try to place on board.



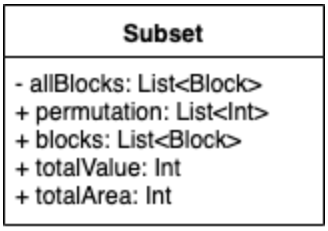
Board

Board class represents a board, but in the sense that it stores its dimensions only.



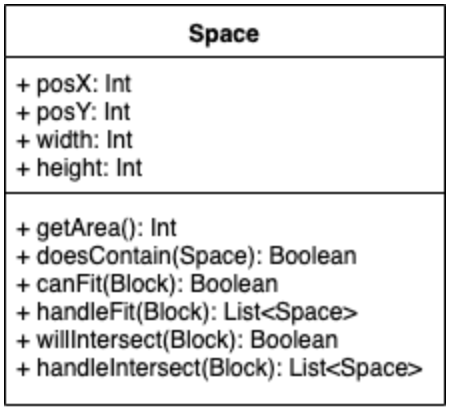
Subset

Subset class represents a given subset of blocks. It receives all blocks and a given list of blocks that it should include. Then it ‘creates itself’ because while coping needed blocks it also calculates its total value and area.



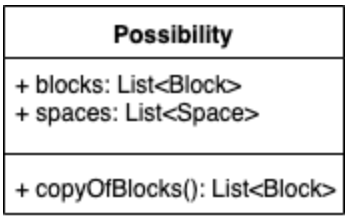
Space

Space class is used to represent an empty space inside a board. It has some logic related to checking whether some block will intersect with it and returning new spaces after such intersection, checking whether some block will fit inside it and returning new spaces after placing it inside it, and also checking whether it fully contains some other space.



Possibility

Possibility class represents some possible situation of checking whether given subset can fit board bounds.



Checking logic

Checking for each subset is made in separate class that can manipulate its objects and only produces information (boolean) whether given subset can be placed inside a board.

There is a small initialisation phase, where we create first Space, that is equal to the whole board, and create a stack of Possibilities with first possibility consisting of all blocks and first Space.

Main logic consists of looping (inside while loop) until stack of possibilities is empty. In each iteration we pop a possibility (**PP**, check whether we have more available blocks in **P** (if not, it means that given board can store all blocks and we terminate earlier), we check if **P** contains more available spaces (if not, it means that there is no more place for remaining blocks, so they cannot be fitted and we terminate earlier) and also most important part in which we place a block. We pop a block (**B**) from **P** and try to place it, with and without rotation, in each of **P**'s spaces. If **B** can be placed somewhere, then we generate new spaces for such placing and push a new possibility consisting of those places and **P**'s blocks.

Pseudocode

```
VAR possibilities = LIST<Possibility>()
VAL initSpaces = LIST<Space>()
VAL firstSpace = SPACE(posX = 0, posY = 0, board.width, board.height)
initSpaces.ADD(firstSpace)
VAL firstPossibility = Possibility(subset.blocks, initSpaces)
possibilities.ADD(firstPossibility)


VAR can_fit_all_blocks = false

WHILE (possibilities.length > 0) {
	VAL currentPossibility = possibilities.POP()

	IF (currentPossibility.blocks.length == 0) {
		can_fit_all_blocks = true
		BREAK
	}

	IF (currentPossibility.spaces.length == 0) {
		BREAK
	}

	VAL currentBlock = currentPossibility.blocks.POP()
	VAR can_fit_block = false

	FOR (i = 0; i < currentPossibility.spaces.length; i++) {
		VAL currentSpace = currentPossibility.spaces[i]

		IF (currentSpace.canFit(currentBlock)) {
			VAL nextSpaces = LIST<Space>

			FOR (j = 0; i < currentPossibility.spaces.length; j++) {
				IF (j == i) CONTINUE

				VAL otherSpace = currentPossibility.spaces[j]
				IF (otherSpace.willIntersect(currentBlock)) {
					VAL newSpaces = otherSpace.handleIntersect(currentBlock)
					nextSpaces.ADD(newSpaces)
				}
			}

			VAL newPossibility = Possibility(currentPossibility.blocks, nextSpaces)
			possibilities.ADD(newPossibility)
		}

		// THE SAME FOR ROTATED BLOCK
```

```
        }
}

RETURN can_fit_all_blocks
```

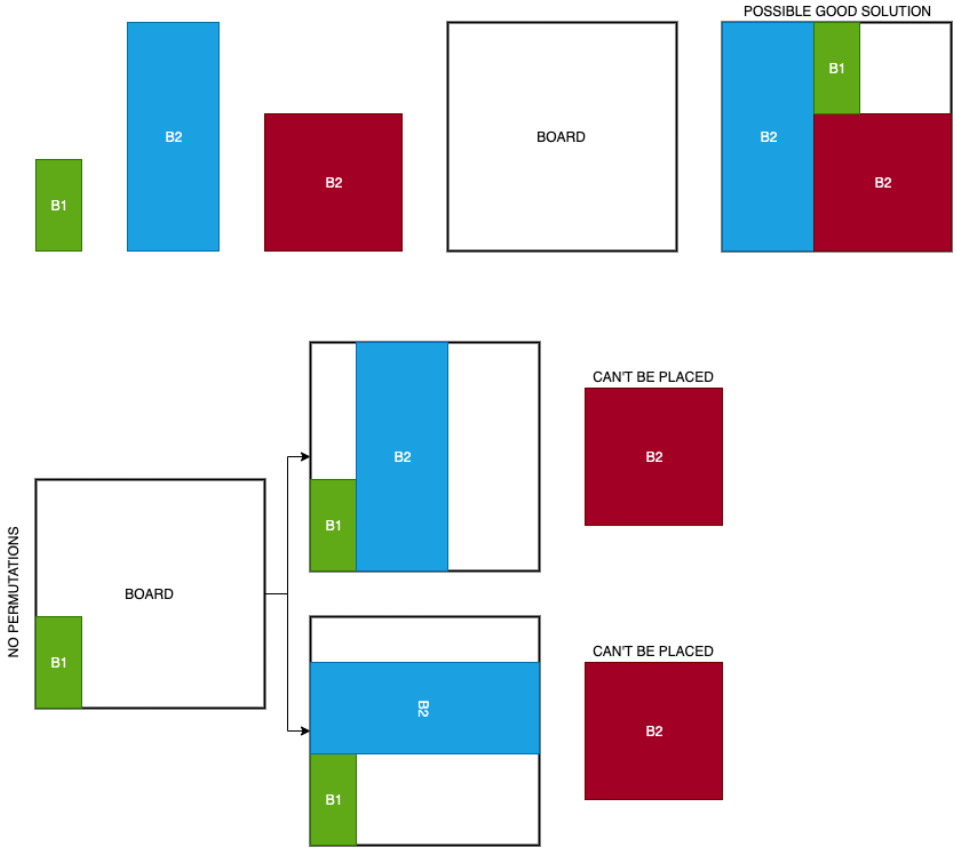
Proof of correctness

Do we consider every single solution to this problem?

In our solution we consider every single combination of blocks, so we go through each possible solution.

Do we handle each possibility correctly?

Our method of placing blocks, one after another, inside a board, in the bottom left corner of some available Space, may not produce proper result in some cases, e.g.



As we can see, this algorithm may not lead to a proper solution. But adding permutations of blocks solves this problem, so yes - we handle each possibility correctly.

Computational complexity

n - number of blocks
w - width of a board
h - height of a board

Worst case

Generating all subsets:

We have a loop going through each possible subset and there are $\mathbf{2^n}$ of them.
For each number we **n** to go through each bit of its binary representation.

So it combines to $O(\mathbf{n2^n})$

Generating all permutations:

For each subset we have to generate all possible permutations and there are **n!** of them.

So it is $O(\mathbf{n!})$, and gives us $O(\mathbf{n2^nn!})$ so far.

Trying to place blocks:

In the worst case we have to place as many blocks as there are fields inside a board, so that gives us **(w*h)!** possibilities.

So it is $O(\mathbf{w*h!})$, and in total we have $O(\mathbf{n2^nn!(w*h)!})$

Best case

We generate all subsets so that's $O(\mathbf{n2^n})$ and at the first try we get a block with size equal to board's size and a value greater than all other blocks, so that gives us $O(\mathbf{1})$.

In total $O(\mathbf{n2^n})$.

Improvements

Threads

We have added option to choose on how many threads the application should perform computations. It has added some more places in which we have to check certain properties optimizing solution in other way, e.g. whether given subset has greater total value than current best, but overall performance is much better.

There are cases then single thread finds best solution and skips other possibilities, while for multiple threads same situation takes longer, because they all have to terminate their calculations, which may take some time, and program cannot be terminated in that time. But still we can always run computations on one thread.

Ignoring too big blocks

We ignore blocks from the data file that are too big to fit given board.

Used memory

We have implemented this program in such a way that it does not have to store all combinations or permutations in memory. This removes some possibilities of improvements related to sorting those lists, but gives us possibility to handle huge numbers of input data (without it program would crash due to memory overflow caused by number of combinations above n = 20, where n is a number of all blocks).

Skipping not needed permutations

Going through all possible permutations generates an enormous number of additional computations, while problem described in ‘Proof of correctness’ occurs very rarely. So we firstly go through each combination without its permutations, and then check again if some permutation of any combination can lead to bigger total value. (There was no such case in approximately 50 different datasets we have tried)

Skipping subset

We skip checking subsets that may not be fitted inside a given board, their total area is bigger than board’s, or can not produce better total value than the current one.

Spaces

Storing available spaces inside a board (for a given Possibility) significantly removes number of possible placings of some block. In the worst (theoretical) case we have to check every single

field of board, but in practice it gives as huge speed advantage. We also check if some places are fully contained by other places and remove them is such a situation (unfortunately we do not eliminate all of such places, but only most of them).

Examples

Different width / height ratios

Board size: 10 x 10
Total area of set: 154
Total value of set: 154 (blocks' values proportional to their sizes)

case number	1	2	3
blocks (width, height, value)	5x (2, 2, 4) 4x (3, 3, 9) 3x (4, 4, 16) 2x (5, 5, 25)	4x (2, 3, 6) 3x (2, 4, 8) 3x (4, 3, 12) 2x (3, 5, 15) 2x (3, 6, 18) 2x (1, 2, 2)	2x (1, 10, 10) 2x (1, 9, 9) 3x (1, 8, 8) 3x (1, 7, 7) 3x (1, 6, 6) 4x (1, 5, 5) 3x (1, 4, 4) 3x (1, 3, 3) 4x (1, 2, 2)
number of blocks	14	18	27
time needed (s)	81.992	564.312	125.436
value achieved	100	100	100

Different values

First example

Board size: 9 x 9

case number	1	2	3
blocks (width, height, value)	4x (1, 2, 2) 4x (3, 3, 9) 4x (2, 5, 10) 4x (3, 4, 12)	4x (1, 2, 12) 4x (3, 3, 10) 4x (2, 5, 9) 4x (3, 4, 2)	4x (1, 2, 1) 4x (3, 3, 1) 4x (2, 5, 1) 4x (3, 4, 1)
time needed (s)	9.895	3.424	8,872

Second example

Board size: 10 x 10

case number	1	2	3
blocks (width, height, value)	2x (2, 2, 2) 5x (2, 3, 6) 4x (2, 4, 8) 3x (2, 5, 10)	2x (2, 2, 10) 5x (2, 3, 8) 4x (2, 4, 6) 3x (2, 5, 2)	2x (2, 2, 1) 5x (2, 3, 1) 4x (2, 4, 1) 3x (2, 5, 1)
time needed (s)	13.858	12.931	13.340

Different numbers of blocks

Board: 10 x 10
Total area of set: 154
Total value of set: 154 (blocks' values proportional to their sizes)

case number	1	2	3	4
blocks (width, height, value)	2x (2, 2, 4) 3x (2, 3, 6) 3x (2, 3, 8) 2x (2, 5, 10) 2x (2, 6, 12) 2x (2, 7, 14) 2x (2, 8, 16)	2x (2, 2, 4) 5x (2, 3, 6) 4x (2, 3, 8) 3x (2, 5, 10) 2x (2, 6, 12) 1x (2, 7, 14) 1x (2, 8, 16)	6x (2, 2, 4) 6x (2, 3, 6) 4x (2, 3, 8) 2x (2, 5, 10) 1x (2, 6, 12) 1x (2, 7, 14) 1x (2, 8, 16)	2x (2, 2, 4) 5x (2, 3, 6) 4x (2, 3, 8) 3x (2, 5, 10)
number of blocks	16	18	21	14
time needed	1.783	3.996	4.568	13.858
value achieved	100	100	100	100

Other interesting cases

Difference between single and multiple threads

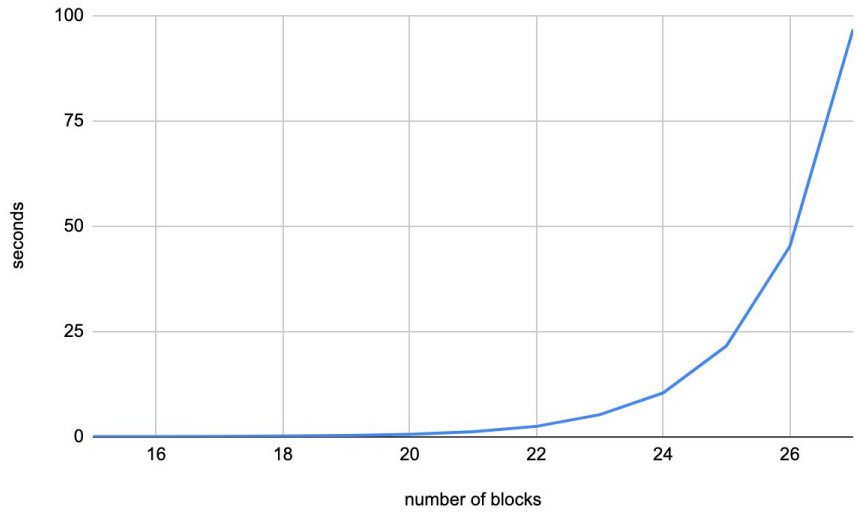
Board size: 12 x 6
Number of blocks: 22

Avg. time for 5 threads: ~ 32.2s
Best result: 47
Img:

Avg. time for 1 thread: ~ 4.2s
Best result: 47
Img:

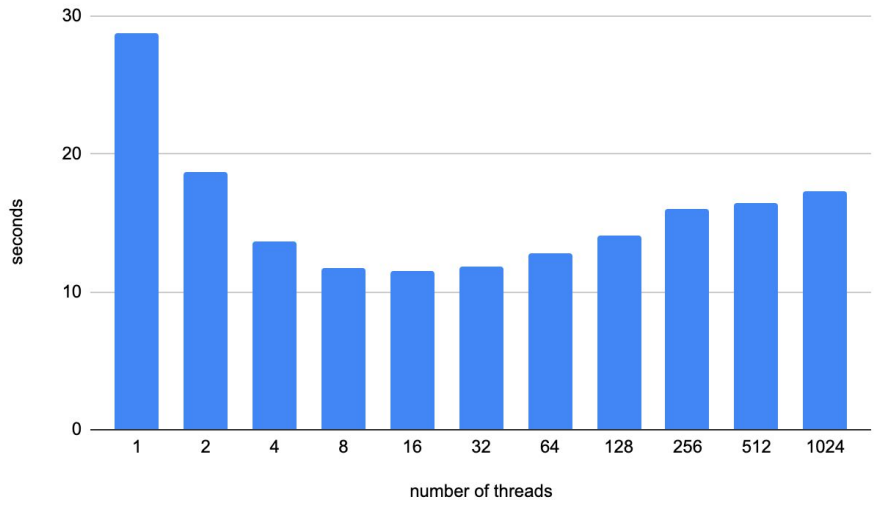
Relation between number of blocks and time of computations

Board size: 7x7
All blocks: w = 3, h = 1, v = 1
Always 4 threads



Relation between number of threads and time of computations

Board size: 10x10
10 blocks: 3x(5, 5, 5), 3x(4, 4, 4), 2x(3, 3, 3), 2x(2, 2, 2)
It is a case in which our program checks permutations



Conclusions

1. The hardest sets of input data are those in which blocks vary as much as possible (their sizes) and value is proportional to their sizes (‘Different width / height ratios’).

2. It may happen that set with smaller number of blocks will take much more time to be checked, but in general time grows with the number of blocks (‘Different numbers of blocks’).

3. Sets where blocks’ values are proportional to their sizes are the hardest to be checked, then there are those where for all blocks there is one (or similar) value, and the easiest are those where smaller blocks have bigger values (‘Different values’).

4. In practice it is hard to tell whether given example (a pair of a set and board) can be checked faster or slower. Small change (like one additional block, one different value, or board’s size different by one) can produce significant change, even the order of blocks in input data may change time of computations.

5. Sets that demand checking permutations need much more time to be checked, but there is not much of them.

6. It is hard to come up with a data set for which computations will take approximately a minute, because small changes in sets can produce huge differences and in most cases computations take up to 40 seconds or much more.

7. In case of examples where best example is at the very beginning of checking, one thread may end computations earlier than multiple.

8. In practice it is much better to use some greedy or heuristic algorithm instead of exact one.