Report: Two-dimensional packing problem

Otto Hytönen 015310439
28.11.22
SciComp2_2022

Introduction

Last summer I was working as an automation operator at a large fully automated warehouse. They had a deal with the automation manufacturer worth tens, maybe hundreds of millions of euros. During my orientation I was surprised to hear that the product the automation company sells is not really the hardware. They told me: ‘any idiot can manufacture conveyor belts and cranes’. No, the real product that differentiates them from the competition is the industry leading packing algorithm.

The problem of packing boxes of different sizes optimally according to various parameters and limitations is a surprisingly difficult one. In the real world the factors you have to account for are e.g. the footprint of the pallet, vertical stability, the compressibility of different packages and the variance in the size of the same product. In this assignment we are luckily limiting ourselves to two-dimensional packing and only minimizing the footprint of the packages.

Methods

As the instructions dictate, simulated annealing is used as the algorithm to solve the problem. Simulated annealing is a probabilistic method to find the optimal state of a system. The Simulated annealing algorithm given in the problem instructions uses the Metropolis Monte Carlo algorithm. It runs the MMC algorithm until a specified condition is met. Then it decreases the control parameters and runs MMC again. The algorithm loops like this until another specified condition is met.

The MMC algorithm used is the following: 1. Change the system configuration. New system configurations are found moving the boxes in different ways. 2. Calculate the change in the cost function. The cost function used in the algorithm is the area of the smallest rectangle containing all the boxes. 3. Generate a random number between n=[0,1). 4. Accept the new configuration if $n<\exp(-\delta f/c)$, in which δf is the change of cost function and c is a control parameter. 5. Loop continues from (1.) if the exit condition is not met.

Implementation of the methods

To run the program, all 6 files in the src folder must be compiled according to the instructions in the folder. Executing the program needs 6 command line inputs: c,a,n,filename1, filename2, filename3. C is real number, the control parameter. A is real number, the cooling parameter. N isinteger, the number of boxes. The last three are character strings. File 1 is the file to save positions for drawing the boxes before the algorithm, file 2 is to save the positions for drawing after the algorithm, file 3 is to save the footprint and iterations for

plotting. A comma is used as a data separator in file 3. The commands for drawing the boxes differ a bit from what was given in the problem instructions. I'm not sure why, maybe I used a different version of the xgraph software. The one I used was this: https://www.xgraph.org/.

The most foundational building block of the program is the derived type 'box'. A box is basically a size 4 real array. It contains four coordinates: x1, y1, x2, y2 These are the coordinates of the lower left corner and upper right corner of the box. Alternatively they can be thought as the defining values of each side of the box. There was an interesting choice on how to define the box. Another way would have been to define the center of the box and height and width of it. Both have their benefits.
I decided to use the coordinate format, because for me it was in most cases easier to conceptualize. The program also has a function to convert between these formats, because the other is more readable in some movement subroutines. The boxes are stored in an allocatable array *allbxs().* This array is accessible everywhere in the program. Another thing accessible everywhere is the box that defines the footprint of all boxes. It's called *boubox.*
Let's go through the program from the top level to the basic building blocks.

main.f90:

The main file of the program is almost exclusively used to call subprograms from the modules. The modules called here are named so clearly, you don't need to open the modules to get a top level understanding on how the code works.

The code there is very short and simple, but one could easily build much more complex implementations using the modules. The code here reads the command line, initializes the random number generator, sets the initial state of the boxes, writes the initial state to a file, runs the algorithm, writes the positions in another file and prints whatever information is wanted.

mrfort90.f90:

This is the random number generator given to us in the problem instructions. The module *mtmod* is used on three other modules. In the module *boxes* it is used to generate random sized boxes when setting the initial state of the system. In the module *movement* the module is used to get movement direction 1 or 2 and a gaussian random number is used to get the length of the move. In the module *algorithm* the random numbers are used to choose a random box and choose one of 3 moves to do to it.

algorithm.f90

This module has three subprograms. The first one just initializes a new random number generator. This subroutine has only two lines. This may seem useless, but there is a reason behind it. I think it's important to be able to initialize a new random number generator and new seed in the main program. I wanted the main program to be self explanatory, but the commands for the new generator were not. That is why I 'wrapped' them in this better named subprogram.
The subroutine *sa_algorithm* runs the actual simulated-annealing algorithm. It takes in the initial conditions of the algorithm. The parameters c and a are the control parameter and cooling parameter of the algorithm. The algorithm has 2 nested do loops. The inner one

iterates from 1 to scalar*n, in which n is the number of boxes. This ensures that the amount of moves per box stays the same regardless of the number of boxes. Inside the inner loop the program calls the move subroutine that does a random accepted move to a random box. Then it checks the accepting condition for a new move (the comparison to random between 0 and 1). If the condition is not met, the move is reverted with the *restore_position* subroutine. The outer loop runs a set amount of times. It writes the footprint of the boxes and amount of iterations to a file. Then it runs the inner loop and updates the control parameters.

The subroutine *move* generates a random accepted move to a random box or boxes in the case of a swap. The subroutines for different kinds of moves are in the module *movement.* These subroutines return false if a move is not accepted due to overlap. The *move* subroutine has a do loop that runs until a move is accepted. In the loop a random integer between 1 and 3 is generated. This decides which of the three possible movements is called. Then a random integer is used to choose a box or two boxes. Then the chosen move is tried to the chosen box. As stated earlier, the movement subroutines return false if move is not accepted due to overlap. The loop continues until an accepted move is found.

movement.f90

This module contains four subroutines. Three of these are for making the three different types of moves, the fourth one is helping with the *move_move* subroutine. The basic structure is the same between the three move subroutines. They all take in the index of the box (or boxes in the swap) and output a boolean that tells if the move overlaps with the existing boxes. Overlapping moves are not saved. The first thing they do is make a new box according to the move.

The new boxes are of course made differently in each subroutine. In the *move_rotate* subroutine two functions are used. *getxywh* returns the coordinates of the center of the box and the width and the height. That function is used to get the data of the chosen box before the move. Then the function *makebox* is used. That takes in the *xywh* format and creates a box. To get a rotated box, the makebox function is used with the old coordinates with swapped h and w.

In the case of the *move_swap* two new boxes are generated. The information of the boxes is gotten with the *getxywh* and then the w and h data is swapped between the coordinates and new boxes are made with the *makebox* function. The *move_move* subroutine uses the *generate_move_move* function to get the new moved box.

After the new boxes are made, the different move subroutines work in the same way. The subroutines use the *overlap* function to check if the new box(es) overlaps with existing boxes. If there is no overlap, the subroutines call the subroutine *remember_position()*. That saves the position(s) of the chosen box(es) before the move is saved in the list *allbxs().* Then the subroutines update the output boolean to true (the move succeeded). Lastly it calls the *opt_checkboundary* (opt=optimized) that updates the footprint box; *boubox*.

The last subprogram in this module is the *generate_move_move* whose purpose is to generate a new box for the *move_move* subroutine. *generate_move_move* takes in the index of the box to be moved. The function generates two random numbers integer (1,2) and

real, gaussian random. The integer decides the direction of the move, (1=x, 2=y). The length of the move is the gaussian random. This can be positive or negative.

boxes.f90

The module *constants* has the integer and real kind parameters. The module *boxes* has the definitions for the different boxes and connected values available everywhere in the program. It also contains different subprograms for the handling of the boxes. The function *makebox* takes in xywh data and outputs a box. The *setinitialstate* takes in the number of boxes. It allocates the array *allbxs().* Then it generates randomly sized boxes with maximum width and height of one. These are placed inside an imaginary grid with square size (1x1), each box in the center of a grid square. This way these boxes can't overlap in the beginning. The boxes are stored in the *allbxs()* array. After this the footprint box, *boubox* is calculated using the *checkboundary* subroutine. Lastly the indexes of the boxes that define the boundary are saved in the *boundaryindexes* list.

The subroutine *checkboundary* goes through the *allbxs* list to get min and max values of x and y to define the boubox. The subroutine *opt_checkboundary.* Is an optimized version of the previous subroutine. It takes in the index of the moved box. The index is needed to check if the moved box was defining the *boubox*. If the box was defining the *boubox,* the previous subroutine *checkboundary* is used. If the moved box was not defining the boundary, it only checks if the new place of the box grows the *boudbox*. Doing this eliminates the requirement of going through the whole list *allbxs()* unnecessarily when calculating the boundary. The runtime benefit for the whole program using the optimized boundary check was about 25%.

The logical function overlap takes in a box and an index. The index, called ignore, is from the box that was moved. Due to the order of operations in the *move_* subroutines, when the *overlap* function is called there exists the old box and the new moved box. The overlap function must ignore the old box. The overlap function goes through the list *allbxs()* and checks if the box overlaps with any of them. The overlap is tested with a long if statement. This function takes the most runtime in the whole program. It could be optimized in many ways. More about these in the conclusions.

The function *getxywh* takes in a box, returns a 4 item real array with the xywh data of the box. The subroutine *remember_position* is used to save a position of box(es) before making a move. It takes in 2 box indexes. If only one box is moved, the other index given must be 0. It saves these indexes in a list *rememberindex* and saves the boxes and boubox in a array of three boxes, called *rememberpos.*

The subroutine *restore_position* restores the remembered position in case the move is not accepted. It checks the list *rememberindex*. If the value 2 is 0, it only restores 1 box, otherwise 2 boxes. It uses the saved indexes to change the corresponding box(es) in the *allbxs* to the box(es) stored in the array *rememberpos*. It also restores the boubox.

The real function *area* takes in a box object and returns the area of that box. This is mostly used for calculating the change of the cost function, meaning the area of the *boubox*. The last function *boxesarea* uses the *area* function and a do loop to calculate the total area of the

boxes. This can be used to evaluate the end result of the algorithm by comparing ‘empty’ area to the *boubox* area.

Inout.f90

The inout module contains all subprograms related to reading and writing. The subroutine *read_cmd_line* reads in the command line arguments. It gives an error message, if the number of arguments is wrong. The subroutine *delete_and_openfile* takes in a filename and the i/o unit. I decided that, instead of giving an error message when a file by that name already exists, the program deletes the file and makes a new file with the same name. This was done to streamline the testing of the code.

The module *closefile* takes in the i/o unit and closes it. This seems useless. Only reason is to streamline other modules and make them more readable.

The subroutine *writefootprint* takes in the i/o unit and amount of iterations and writes to that i/o channel the amount of iterations and area of the *boubox*. This is used inside the algorithm. This subroutine doesnt have the open and close commands because the file needs to be accessed repeatedly during the run of the *sa_algorithm*, so it makes more sense to just keep it open during that.

The function *normalize* is used when writing the values of the box coordinates for drawing. It takes in a box and changes its coordinates so that the lower left corner of *boubox* is printed at the coordinate (0,0). The subroutine *writeboxcoord* takes in the i/o unit and a box
It writes the coordinates in a format readable by the xgraph application. This implementation draws the outlines of the boxes. The function *var_writeboxcoord* differs from the former only in that the drawn boxes are solid color (not just outline) and are of random colors. It is not used at the moment, but it is better for visulizing a large number of boxes. (200+)

The last subroutine *writepositions* takes in a filename and unit. It is used to write all boxes and boundbox in a form readable to the xgraph. It calls the *delete_and_openfile* subroutine. First it writes the line thickness to the file. Then it goes through *allbxs* and uses the *writeboxcoord* subroutine and *normalize* function to write the coordinates of each box. Then it writes a command to change the line color. Lastly it writes the coordinates of *boubox* and calls the *closefile* subroutine to close the file.

Results

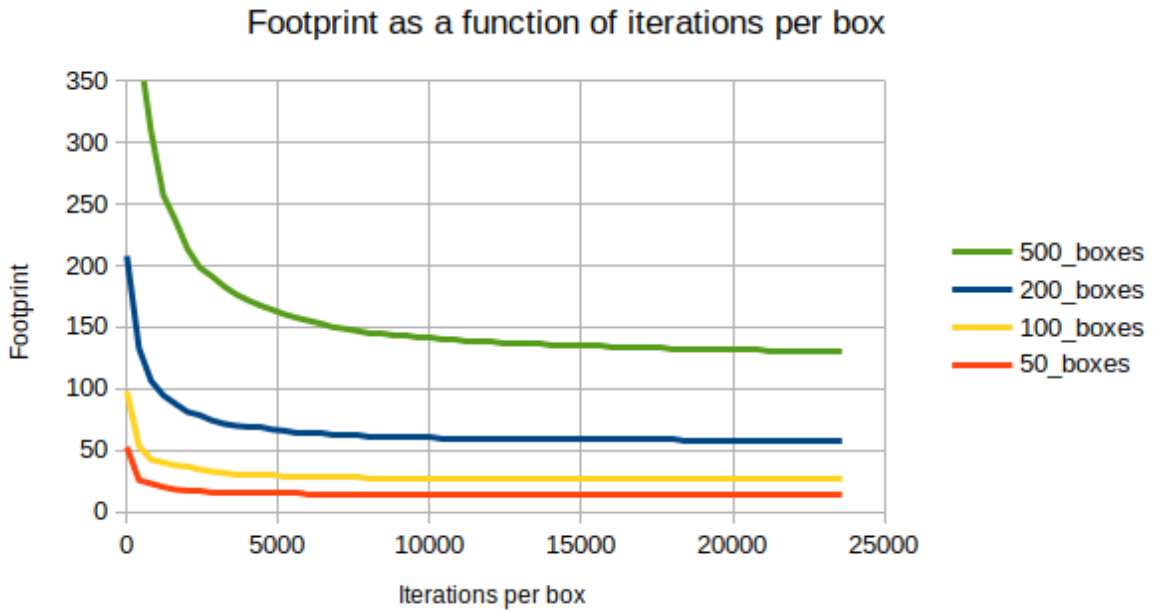
The choice of the parameters is the key to getting good results from a sa-algorithm, or so I at least thought. In practice with this program the end result is not highly dependent on the variables, only the iteration count and time to reach the end result varies. I found that a good starting value for c was 0.1. If it was bigger, the footprint would increase unnecessarily and it would take more iterations to reach the final state. If it was way too small the program wouldn’t accept any radical changes to the initial state.

The cooling parameter also didn’t seem to matter much. I settled on 0.8 because it reached consistently good results. The cooling parameter is a compromise between best results

(near to 1) and faster code (nearer to 0). Last parameters to choose were i and j, the iteration count for each c value and the iteration count of the outer loop decreasing c. For i I chose 400*n in which n is the amount of rectangles. Good value for j was 40 With 40 iterations the final value of c was in the order of 10e-7.

All these parameters are a choice between performance and accuracy. Accuracy can be measured as a ratio of empty space inside the footprint to the area of the footprint. I preferred the accuracy and chose variables so that the final configuration looks perfect to the eye. The drawback of this is that the amount of boxes that can be calculated in a reasonable time is quite small. Hundred boxes takes about 4 seconds and the empty space percentage is only about 6%. With different parameters I can get that down to 1 second, but empty space is about 10% of the footprint.

In the graph (pic 1) we see that the decrease in area is first really fast, but then gets almost stable. One could argue that for a small number of boxes the simulation runs far past the point of diminishing returns. It is interesting that a larger number of boxes also needs more iterations per box to reach the ideal state.



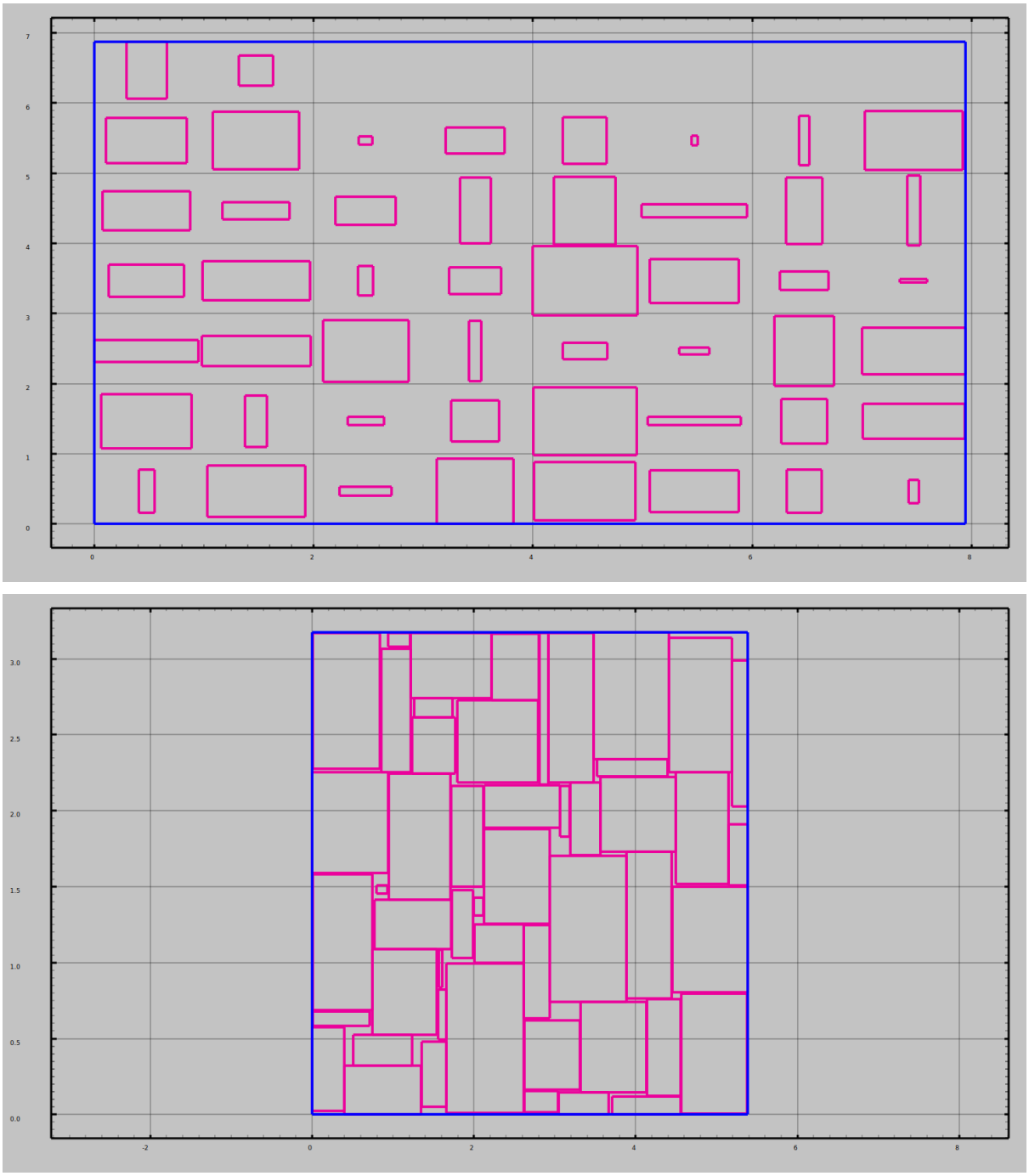
Pic 1: Footprint as a function of iterations per box. Larger number of boxes also needs a larger number of iterations per box to reach the point of diminishing returns.

Conclusion

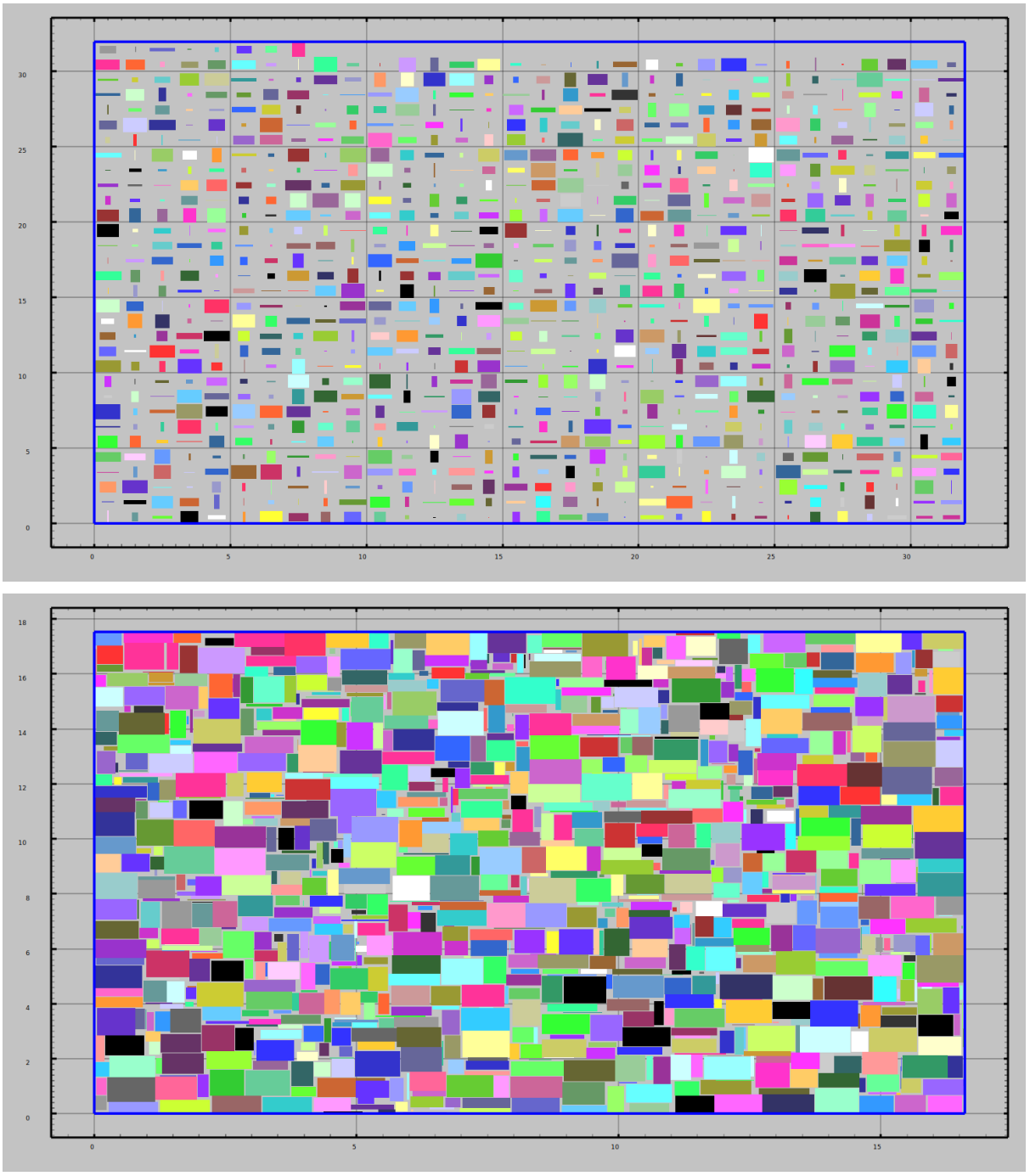
The program works as required. However it is not very well optimized. In real use cases one should thoroughly consider where the point of diminishing returns lies. How perfectly do the boxes need to be packed? As for actual optimization; the runtime of the code rises exponentially when the number of boxes is increased. This is mainly due to the fact that the

function for checking overlap is not optimized. It goes through the list of all boxes every time a move is made. That is very inefficient.

I came up with two ideas to optimize the overlap function. The first one: divide the footprint into smaller areas and keep a list of which boxes overlap which area. When a move is made, it is enough to check overlap with the boxes in the smaller area. Second way to check overlap more efficiently would be to sort the list of all boxes by the x coordinate. When a move is made you would have to only check the couple of boxes whose x coordinate overlaps with the box. This would require an efficient way to move the single box in the sorted list from one index to another. I don't know if that is possible.



Pic 2: 50 boxes initial and final configuration. Runtime 1.8s Empty space percentage 4.4%



Pic 3: 1000 boxes initial and final configuration. Runtime 6 min 11s. Empty space percentage 11%