Two-dimensional packing problem

Final Project of Scientific Computing II

Nyyti Ojanen
015850229

16.12.2022

1 Introduction

Imagine having a bunch of packages of rectangular shapes. Your job is to rent a moving van and relocate these packages to a new address. To save resources and space, you want to rent the smallest possible van that can still fit all the packages inside. Now all you need to do is find the smallest possible cuboidal container that minimizes the waste space.

Figure 1: Illustration of a moving van. Now all the black spots are empty spaces between the packages [1]

Let's study two-dimensional version of this problem. Instead of packing three-dimensional packages you need to organize rectangles inside rectangular container. This is called a two-dimensional packing problem. It has the same idea as packing the moving van, you need to minimize the waste space. [2]

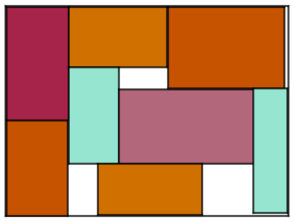

Figure 2: Rectangular container filled with rectangles of different sizes and colors. White spaces are empty. [1]

There are multiple approximation methods to solve this problem. In this project the problem is solved using the simulated annealing (SA). It is one of the common applications of Monte Carlo methods used to solve optimization problems.

2 Methods

2.1 Simulated Annealing

If you are familiar with the physical process of annealing, you may know that you can decrease the defects of a material, if you first increase its temperature and then slowly cool it down.

This is the idea behind simulated annealing. We generate new configurations of a system and at the same time decrease the temperature. The goal is to find the most optimal configuration, the global minimum of costfunction.

The new configurations of the system need to follow the Maxwellian distribution.

$$p(x) \propto exp\left(\frac{-\Delta f(x)}{c}\right) \tag{1}$$

Here, Δ f(x) is the change in the costfunction, x is the configuration and c is the control parameter (other known as temperature)

This is done by using the Metropolis-Hastings algorithm. In each iteration we randomly choose i_max configurations from the set of all possible configurations. The probability isn't distributed evenly within the set, but it is instead distributed exponentially as shown in the formula (1).

When we iterate over the Metropolis-Hastings algorithm while decreasing the temperature, we get closer and closer to the global minimum. The algorithm avoids getting trapped in local minima because it samples from a probability density instead of always following the cost gradient. The more the temperature decreases the more picky the algorithm gets and in the end it settles to the global minimum.

Because the simulated annealing is an approximation method we might not get exact results. They are approximations. [1]

In the two-dimensional packing problem the goal is to pack the rectangles as tightly as possible. In other words the area of the rectangular container needs to be minimized. It makes sense to choose the costfunction as the area of the rectangular container. The configurations are different versions of the order of rectangles. Parameters c and i_max can be chosen to correspond to the desired accuracy of the approximation.

3 Implementation of methods

The fundamental idea of the program is to create a two dimensional packing problem and solve it. The first step is to create a bunch of random rectangles that does not share any common area. The rectangles are widely spread around the grid area. The second step is to find a new arrangement for the rectangles that minimizes the waste space by running the beginning layout through the simulated annealing algorithm.

There are three ways to move a rectangle. One is to move the rectangle from place A to a randomly chosen place B. Second is to swap places of two rectangles and third is to rotate the rectangle by 90 degrees. The rectangles aren't allowed to overlap after any of the moves.

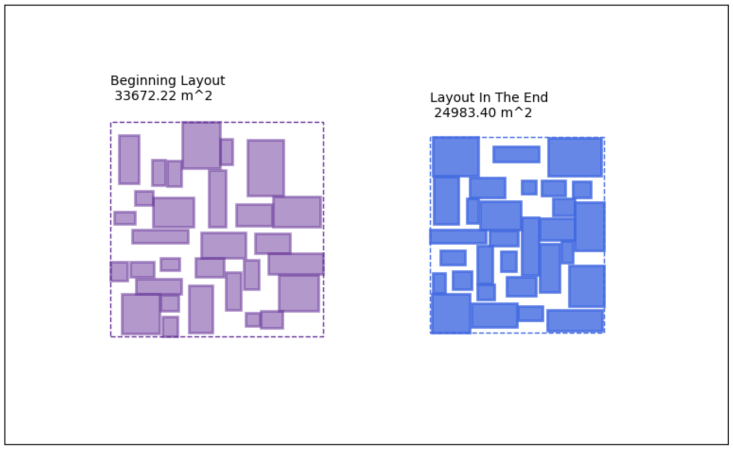

Figure 3: On the left side we can see the beginning configuration of rectangles and on the right side the resulting layout. [1]

3.1 The architecture of the program

The program is divided into nine modules, main program and python program. The main blocks of the code are the modules global_constants, create_rectangles and simulated_annealing and of course the main program and python program. Rest of the modules are created to help build these blocks.

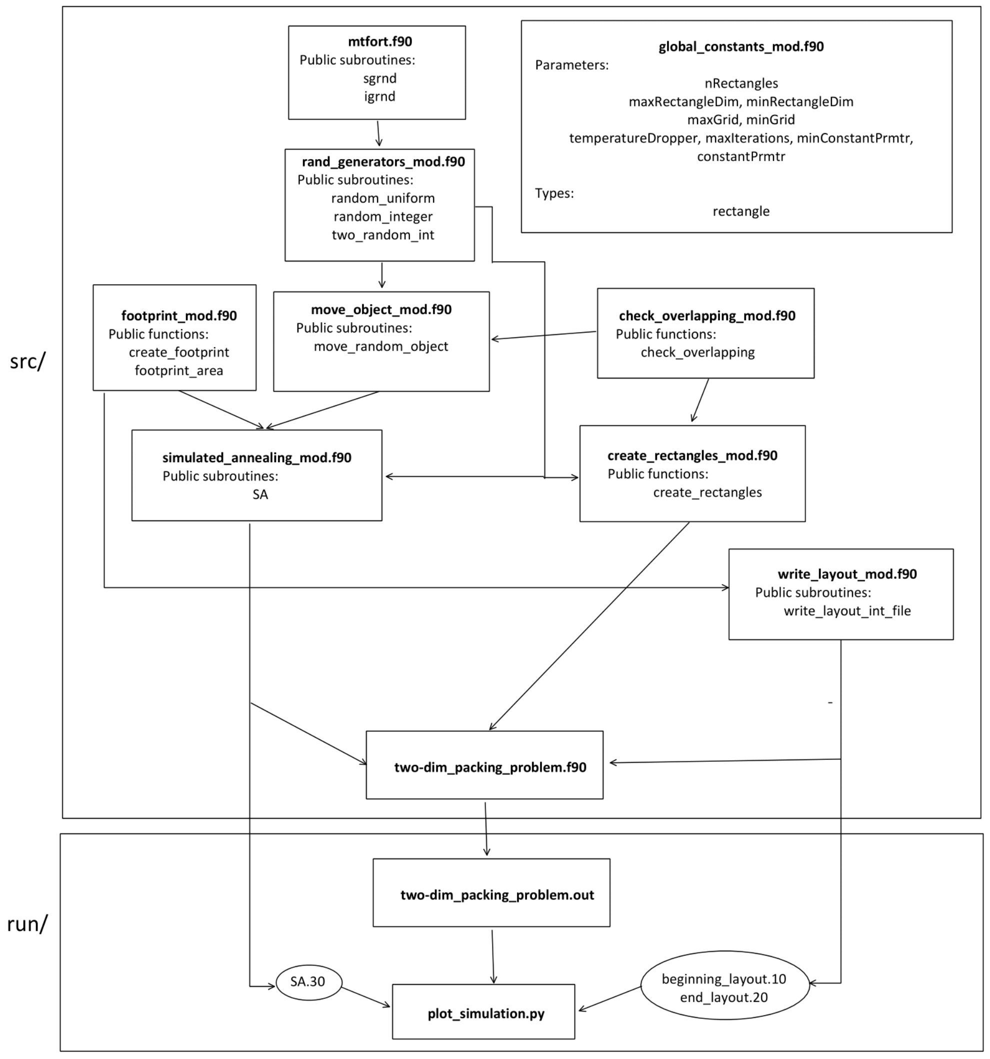

Figure 4: The architecture of the program. The diagram shows which modules uses which and what are the main modules that the program uses. Global constants are used in every module. Inside the src file, the squares are modules and programs. Inside the run file, circles are data files and the squares are executable programs. [1]

Modules:

global_constants_mod.f90 is a module that includes instructions to type rectangle and all of the parameters which values often needs changing before running the program. In this module you can control the features and number of rectangles and change the parameters of the simulated annealing.

mtfort.f90 is a module by Antti Kuronen. It includes many subroutines and functions that generate radom numbers based on different distributions. The program uses the subroutine sgrnd and function igrnd. Sgrnd is a subroutine that returns a random seed. Based on that seed the function igrnd returns a random integer from a uniformly distributed interval (a,b).

rand_generators_mod.f90 is a module that includes functions random_uniform, random_integer and two_random_int. Function random_uniform returns a random real number from the uniformly distributed interval (a,b). Function random_integer combines the subroutine sgrnd and function igrnd to create a random integer. Its soul purpose is to simplify the code when creating random integers. Function two_random_int returns two random integers that are not equal.

check_overlapping_mod.f90-module includes function does_rects_overlap that checks wether two rectangles share common area. If they do, it returns logical value true. Otherwise it returns logical value false. Function does_any_rects_overlap uses the function does_rects_overlap to do the same inspection for a given array and rectangle. If rectangle overlaps with any of the rectangles in the array, it returns logical value true. These two functions are made to simplify the function check_overlapping. check_overlapping takes in the r:th rectangle of an array and the array itself. It checks whether the rectangle overlaps with any of the other rectangles in the array.

create_rectangles_mod.f90-module includes function create_rectangle that creates a rectangle with randomly chosen position, width and length that are within the boundaries chosen in the module global_constants. Function create_rectangles creates the wanted amount of rectangles that does not share any common are and stores them in an array. As you may guess this function uses the function check_overlapping. It is used to create the beginning setup for the rectangles.

move_object_mod.f90 - module includes subroutines move, swap_places and rotate to perform any of the three moves on the rectangle. It also includes a subroutine move_randomly that chooses with equal chances one of these three subroutines. The same way the subroutine move_random_object chooses two random rectangles from given array and performs subroutine move_randomly on them. After the move is made the subroutine checks whether the rectangles are now overlapping. If they are, the move is rejected and it tries again with a new pair of rectangles. When a move that can be accepted is found, it returns the changed array.

footprint_mod.f90-module includes function create_footprint that creates the smallest rectangle in which the rectangles in a given array can be stored in when the places of the rectangles remains unchanged. It also includes function footprint_area that creates a footprint with the function create_footprint and calculates its area.

write_layout_mod.f90-module contains a subroutine write_layout that takes in an array that contains a layout of rectangles. It also takes in the file information where the layout is due to store. It creates the footprint of the layout with function create_footprint and writes it to the first row of the file. After that it simply writes each rectangle of the array to it's own row of the file.

simulated_annealing_mod.f90- module includes subroutine SA that performs the simulated annealing algorithm to a given set of rectangles. Every time the temperature is dropped, the program writes the current temperature and area of the footprint in file SA.30. It returns the smallest configuration it can find with the parameters chosen in the module global_constants.

Main program:

two_dim_packing_problem_mod.f90 is the main program where the two-dimensional packing problem is created and solved. First it creates the beginning layout of rectangles and stores it in file beginning_layout.10. After that it calls the subroutine SA to find the smallest configuration for these rectangles and writes the result in file end_layout.20.After all of the layout information are saved the program request the python program to run. The program calculates the CPU-time that is burned during computations and prints it to terminal after the python program has been completed.

Plotting with python:

plot_simulation is a python program that reads in the layout information from the files beginning_layout.10, end_layout.20 and SA.30. It plots the beginning- and ending-layout of the rectangles and also the area of the footprint as a function of constant parameter.

3.2 Compilation

I've created a Makefile to simplify the compilation of the program. When you're in the folder finalproject the compilation and executing can be done in the following way in the terminal:

somepath/finalproject make
somepath/finalproject ./run/two_dim_packing_problem.out

4 Results

In all configurations the width and height of the rectangles are values between (10,50).

I ran the program with eight rectangles and the initial temperature of 100. In each iteration I tried 24 different configurations and dropped the temperature with factor 0.995. The minimum temperature was set to 0.01. You can see the the results in figures 5 and 6 down below.

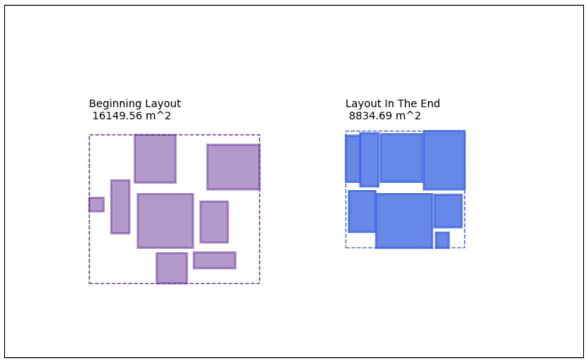

Figure 5: On the left side we can see the beginning configuration of rectangles and on the right side the resulting layout. [1]

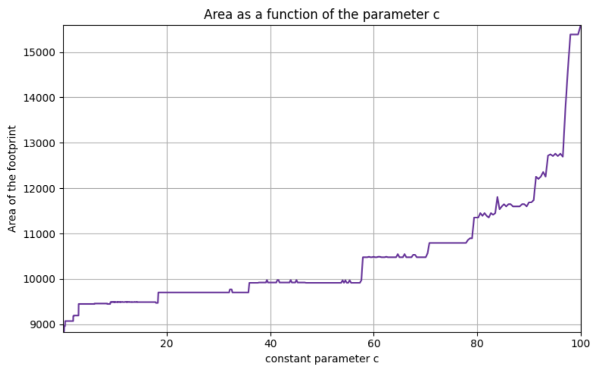

Figure 6: How the costfunction (Area) drops when the temperature (control parameter) is dropped [1]

From the figures we can see that the area dropped from 16 149.55 m^2 to 8834,69 m^2. The area dropped 45% percent from the original area.

I ran the program with forty rectangles and the initial temperature of 100. In each iteration I tried 120 different configurations and dropped the temperature with factor 0.995. The minimum temperature was set to 0.01. You can see the the results in figures 7 and 8 down below.

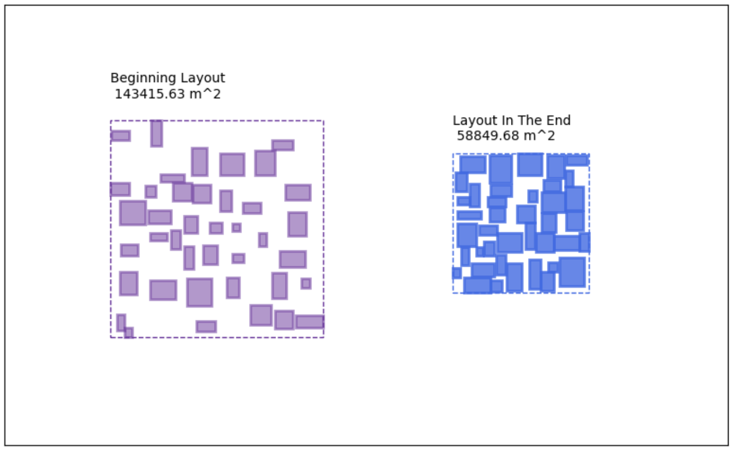

Figure 7: On the left side we can see the beginning configuration of rectangles and on the right side the resulting layout. [1]

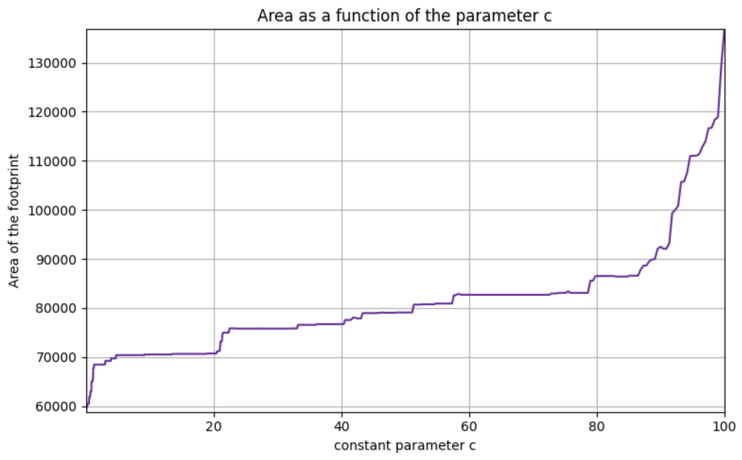

Figure 8: How the costfunction (Area) drops when the temperature (control parameter) is dropped [1]

From the figures we can see that the area dropped from 136 779.55 m^2 to 58 849,68 m^2. The area dropped 57% percent from the original area.

I ran the program with hundred rectangles and the initial temperature of 100. In each iteration I tried 100 different configurations and dropped the temperature with factor 0.995. The minimum temperature was set to 0.01. You can see the the results in figures 9 and 10 down below.

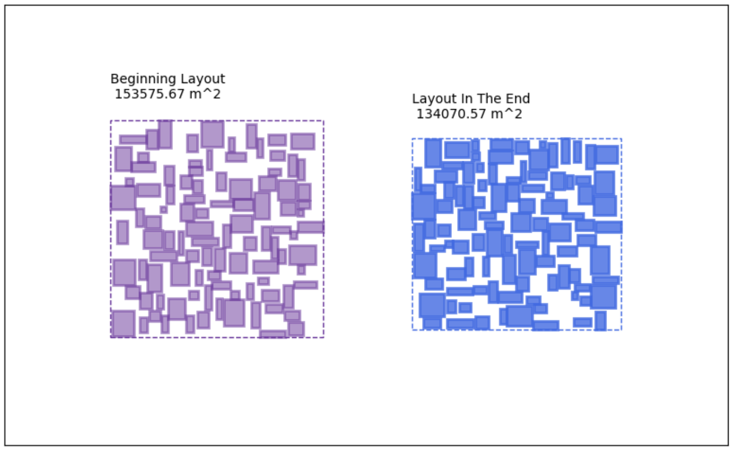

Figure 9: On the left side we can see the beginning configuration of rectangles and on the right side the resulting layout. [1]

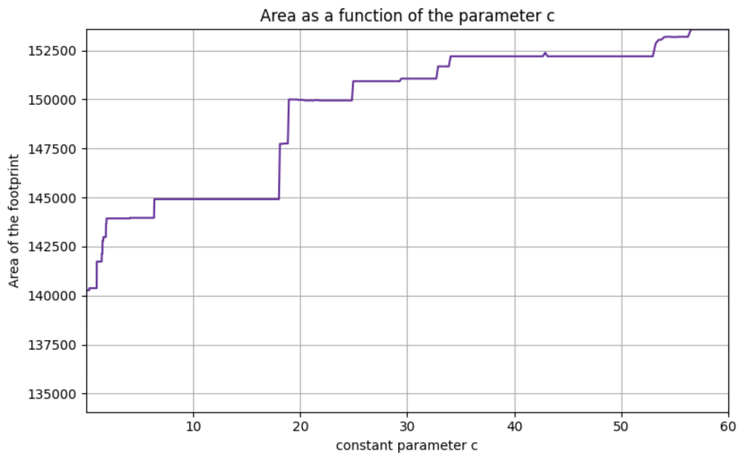

Figure 10: How the costfunction (Area) drops when the temperature (control parameter) is dropped [1]

From the figures we can see that the area dropped from 153 575,67 m^2 to 134 070,57 m^2. The area dropped only 13% percent from the original area.

5 Conclusions

One thing is clear, increasing the number of new configuration trials corresponds with good accuracy on the results. This can be done in multiple ways: by raising the beginning value for constant parameter or i_max, by dropping the minimum value for the temperature or by decreasing the temperature in a slower rhythm. Unfortunately this usually also means huge computation time. For example when I ran the program with the same parameters as in figure 9, the computation took almost 50 minutes. It can also be seen that if the computation time remains unchanged and we add more rectangles, the accuracy of the result gets worse and more waste space is left between the rectangles.

I think it's fair to say that my program could use some improving. It is fairly slow if we considerate that the problem that needs to be solved is a very common one.

One way to optimize the program is to get rid of unnecessary loops and if possible, pack more actions in one loop. Although there is a downside for doing this. It makes the code more unpleasant to read and understand. Its also good to minimize the if con

Another way to optimize the program is to clear all the avoidable copying of variables. Many functions in the program takes in an array that it copies to a dummy variable. This is something that takes a lot of computation time. Especially if it's done many times and the array has huge dimensions.

Then there is the parallel computing. It means executing programs computations and processes simultaneously. It's very convenient especially for program that has processes that are totally separate from each other.

6 Sources

illustration
[1] Nyyti Ojanen

materials

[1] MathWorks Help Center 2022, What Is Simulated Annealing?
https://se.mathworks.com/help/gads/what-is-simulated-annealing.html

[2] Andrea Lodi,Silvano Martello Michele Monaci, 1.9.2002, Two-dimensional packing problems: A survey, European Journal of Operational Research, 241-252