2D packing problem

Jacob Eskin

20.12.2015

Contents

1 Introduction

In this project the Metropolis Monte Carlo, or MMC, algorithm is used to solve a 2 dimensional optimization problem motivated by the simulated annealing method. A set of rectangles is generated and randomly placed on a 2 dimensional grid after which the program minimizes the footprint of the set of by means of randomly moving them around and accepting the new configuration if it satisfies certain conditions. The program takes as input parameters only the number of rectangles to be generated, and as output it writes two text files for drawing the initial configuration and final configuration with Xgraph- program and prints out the initial footprint and the final footprint.

2 Methods

In physical simulated annealing problems the goal is usually to minimize the potential energy of the system; potential energy is described by a so called cost function. In this project instead of minimizing potential energy, the goal is to minimize the footprint, or total area of a square, occupied by a set of rectangles.This is done by generating configurations that obey the Maxwellian distribution

$$P(configuration)\propto exp(-f(configuration)/c),$$

where $f(configuration)$ is the footprint of the configuration of rectangles and c is a control parameter whose magnitude is related to the total number of possible configurations; when $c\rightarrow 0$, the above distribution approaches the minimum configuration, $P(x)\propto\delta(x-x_{min})$, where $x_{min}\equiv f(minimum)$.

Finding the minimum of the footprint is done by the following algorithm:

1. Choose an inital value for the control parameter

2. Generate the initial configuration where the rectangles do not overlap

3. Set the simulation step counter to its initial value

4. Generate a new configuration

5. Calculate the change in the footprint

6. Reject the new configuration if $\xi > exp(-f(newconfiguration)/c)$, where ξ is a random number from the interval $[1,0]$, or if the rectangles overlap

7. Increase the simulation step counter; if maximum steps are reached continue to 8., otherwise go back to step 4.

8. Decrease the control parameter, $c \to c\alpha$ where $0 < \alpha < 1$

9. If minimum value for c is reached, stop the algorithm, otherwise go back to step 3.

Initial configuration is generated by randomly assigning grid coordinates to a set of rectangles and new configuration is generated by randomly choosing a rectangle and eihter moving it one grid point or rotating it 90^o clockwise or swapping two rectangles. Random numbers are generated with the Mersenne twister.

3 Implementation of the methods

The program consists of two modules and a main program code. One of the modules, `mtfort90.f90`, has the subroutines for the Mersennse twister random number generator which is used to generate random numbers for the algorithm. The other module, `squares.f90`, defines the derived data type `square` for holding the necessary information of the rectangles as well three subroutines: generating the initial configuration, generating new configuration and calculating the footprint. The main program code, `packing.f90`, consists mainly of the steps of the above algorithm as well as writing the initial and final configurations into a text file for plotting. The module `squares.f90` and main program source code `packing.f90` where written by me, the module for Mersenne twister was downloaded from the course source code folder
http://www.courses.physics.helsinki.fi/fys/tilaII/files/mtfort90.f90. Compilation of the code is done by running a Makefile in the command window.

3.1 Module squares.f90

The `squares.f90` module is written as follows. In the beginning the derived type `square` is defined; it has four components, the dimensions of the rectangle and the x- and y- components of the position coordinate. It holds the configuration of the rectangles.

After this is written the subroutine `init_conf(n, sqrs)` which generates the initial configuration into the variable `sqrs(n)`. It randomly generates dimensions for all n rectangles and then generates position coordinates for the first rectangle, position coordinates being the coordinates of the lower left corner of the rectangle; this is how position is defined for all rectangles. After this position coordinates for the other rectangles are generated one by one making sure there is no overlap; in case of overlap new position coordinates are generated.

Next is written the subroutine `new_conf(n,sqrs_old,sqrs_new)`. This subroutine takes in the current configuration stored in the variable `sqrs_old` and creates a new configuration `sqrs_new` by one of the means described above.

First the method of creating the new configuration os chosen: a random integer between 1 and 3 is generated where 1 is moving a rectangle, 2 is rotating a rectangle and 3 is swapping two rectangles. In the case of 1 or 2, a random integer is again generated between 1 and n, and the rectangle `sqrs(n)` is then either moved or rotated, after which a routine for testing for overlap is performed. In the case of swapping two rectangles two random integers are generated between 1 and n, and these rectangles are swapped, after which again a routine to test for overlap is performed. In case of overlap, new operation and new square(s) are chosen randomly until one operation is succesfull.

Last is written the subroutine for calculating the footprint, `footprint(n,sqrs,fp)`. It goes through all rectangles in the configuration `sqrs(n)` and calculates the footprint by determining the what are the smalles and largest x- and y-coordinates the rectangles have.

3.2 Main program source code packing.f90

The main program begins by reading the number of rectangles of the system from the command line. After this it initiates the random number generator and calls the subroutine `init_conf()` in order to create the initial configuration. After this it calculates the intial footprint by calling `footprint()` and writes the initial configuration into a text file in a format from which the program Xgraph can draw it.

After all this begins the main part of the algorithm. The simulation consists of two nested do-loops that do not have an argument; the outer one holds the steps from 3 through 9, the inner holds the steps from 4 through 7. New configurations are generated by calling the subroutine `new_conf()` and are rejected or accepted by the criterion stated above. The inner loop is exited when maximum simulation steps are reached, the outer one whne the simulation is done due to the control parameter reaching its minumum value.

After the simulation is finished the final configuration is written on to a text file so that it can be drawn with Xgraph. Similarly the array that records the footprint every time the control parameter is changed is written on a file. Also the initial and final footprints are pritend out on to the command window.

4 Results

Values for the parameters used in running the simulation are the following:

Initial value for the control parameter $c_0 = 100$.
Minimum value for the control parameter $c_{min} = 0,00000000001$.
Value for the term that changes the control parameter $\alpha = 0,999$.
Maximum value for simulation steps was 1000000.

Also all rectangles had the dimensions of 2x2, 2x3 or 3x3.

These parameters were chosen after testing many different values and combinations. They gave a very good result in the final footstep while keeping the size of the simulation manageable. The simulation was ran with three different numbers for the rectangles, 10, 50 and 100. Below are the initial and final configurations and a plot that shows how the footprint changed after every 1000000 simulation steps, or in other words every time the control parameter was changed. The plots were created with the program Xgraph.

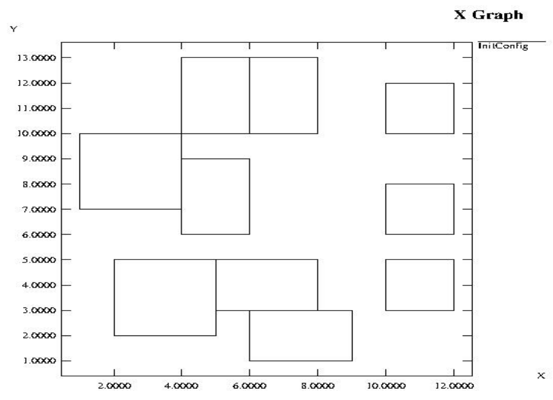

Figure 1: Initial configuration of 10 rectangles.

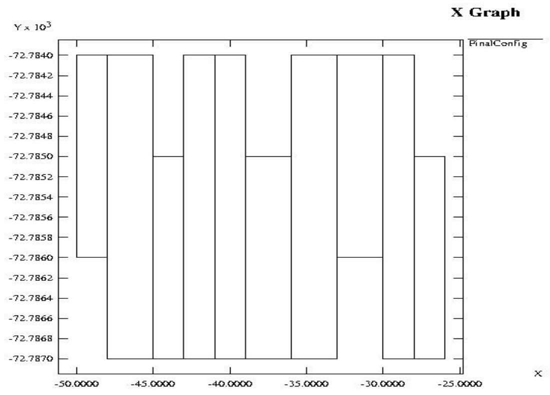

Figure 2: Final configuration of 10 rectangles.

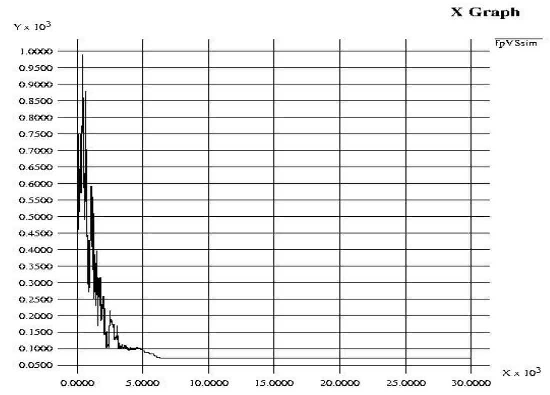

Figure 3: Y-axis is the footprint, X-axis is the simulation steps x 1000000.

Xgraph isn't the best graphical tool, as one can see the image in Figure 2 is squished from the sides and the x- and y- axis are out of proportion. But the important information on how the rectangles are situated next to each other neatly is conveyed. The following images are for 50 rectangles. As can be seen from the plot of footprint vs. simulated steps, finding the minimum happens a lot smoother in this case.

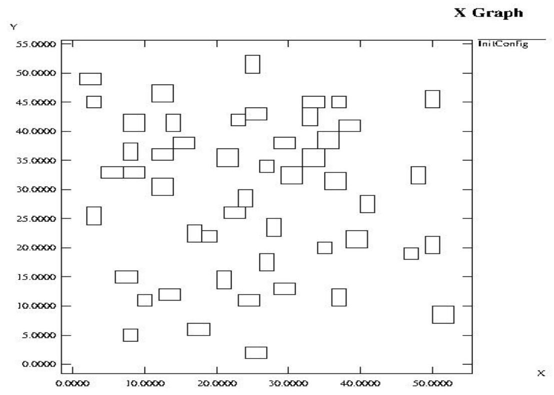

Figure 4: Initial configuration of 50 rectangles.

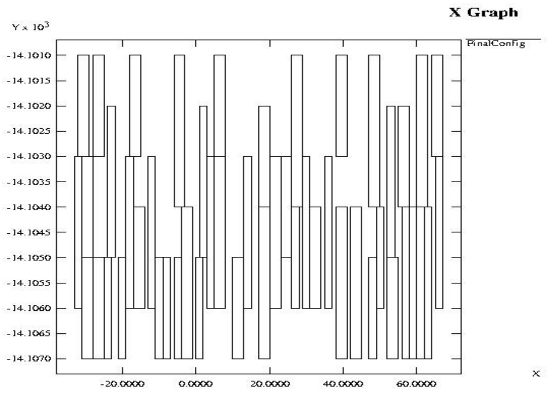

Figure 5: Final configuration of 50 rectangles.

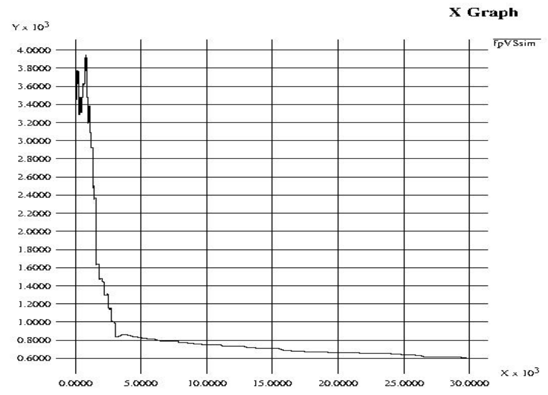

Figure 6: Y-axis is the footprint, X-axis is the simulation steps x 1000000.

And finally we have the same figures for 100 rectangels. Here the disproportion on of the axis in the figure for final configuration is most visible. Here in the last case we have the smoothest decent of the footprint to the minimum. Because we started with a positive and somewhat big c and α was so close to 1, it was possible for the footprint to grow at times and so the rectangles migrated quite alot, as can be seen by the negative axis values in the final configurations.

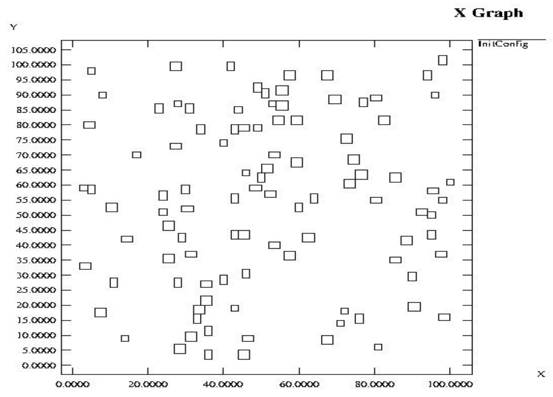

Figure 7: Initial configuration of 100 rectangles.

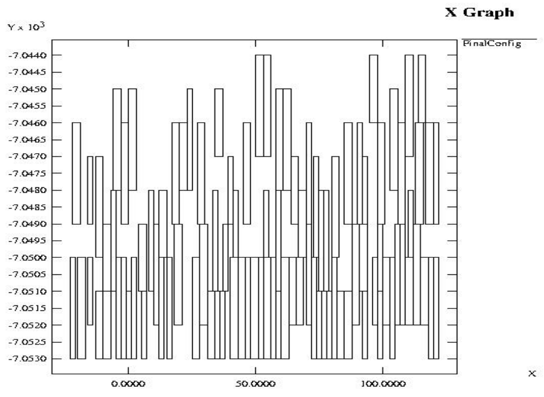

Figure 8: Final configuration of 100 rectangles.

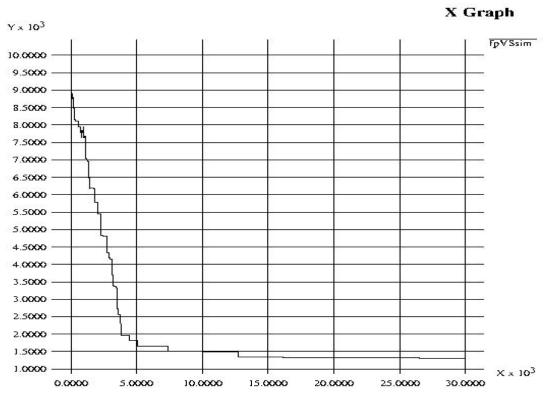

Figure 9: Y-axis is the footprint, X-axis is the simulation steps x 1000000.

5 Conclusion

This method seemed quite good in minimizing the footprint, then again our cost function was extremely simple. There are also extra steps that can be added to the algorithm in order to make the decent to the minimum non-linear; when the control parameter reaches a certain value, it is increased a little. This way for there is a bigger chance to find the global minimum for more complicated cost functions.

When it comes to the actual code, there might be some ways to shorten it a little. In the module `squares.f90` for example the segment in creating and changing the configuration that checks for overlap could have been written as a separate subroutine, but I deemed it unnecessary since it was not used in the main program code.