Two-dimensional packing problem

Scientific Computing II

Tom Rindell
014605789

Introduction

In an optimization problem, one tries to find the best solution from a set of all possible solutions. This is achieved by asserting a value to each solution and finding a way to minimize or maximize the value, depending on the nature of the problem.

An example of this would be finding the highest point on a curve. After starting on a random point on the curve, one might choose to approach the problem by using hill climbing algorithm. This means choosing the next highest point and to repeat the process until a maximum is reached. However, if the curve has many peaks, the reached local maximum is unlikely to be a global maximum.

In this project, a technique called Simulated Annealing (SA) will be used to solve two-dimensional packing problem. By using probabilistic methods, SA avoids the problem of getting stuck on a local maximum (or minimum) and intends to find the global optimum. The problem is to pack a two dimensional set of rectangles, so that the area covered by rectangles (footprint) is minimized. The rectangles in the set must not overlap.

Methods

Two-dimensional packing problem will be solved using simulated annealing (SA) algorithm. The algorithm employs a degree of randomness to achieve the global optimum, without stopping at a local optimum. The SA algorithm is executed by following steps:

1. Initialize the control parameter $c \leftarrow c_0$
2. Initialize the system $x_i \leftarrow x_0$
3. Set $i \leftarrow 1$
4. Change the system configuration $x_i \leftarrow x_i + \delta x$
5. Calculate the change in the cost function: $\delta f = f(x_i + \delta x) + f(x_i)$
6. Generate a random number between 0 and 1: $\xi = [0, 1[$
7. If $\xi < e^{-\frac{\delta f}{c}}$, accept the new configuration: $x_{i+1} \leftarrow x_i + \delta x$
8. Set $i \leftarrow i + 1$. If $i \leq i_{max}$ go to step 4. Otherwise go to next step
9. Decrease the control parameters: $c \leftarrow \alpha c$, $0 < \alpha < 1$.
10. If $c < c_{min}$ stop. Otherwise set $i \leftarrow 1$ and go to step 3.

To generate random numbers, I will be using Mersenne twister (mtfort90.f90) module. The initial and final states of rectangles will be visualized by `xgraph`.

Implementation of the methods

Let us define a rectangle:

```
TYPE :: rect
   REAL :: w
   REAL :: h
   REAL :: r(2)
END type rect
```

Here w and h are width and height respectively, and r(2) is an array which contains the values of x and y coordinates. Even though the dimensions and the position of a rectangle are defined to be of `REAL` type, to make moving and manipulating rectangles easier, these values will be set as whole numbers.

```
TYPE(rect), DIMENSION(10) :: table
```

Before writing a function that sets the values for the 10 rectangles in the array, it may be useful to define another function, which checks whether two rectangles overlap. Function `testR(r1,r2)` takes two rectangles as an input and returns a boolean value `.FALSE.` if the rectangles overlap and `.TRUE.` if they don't. It is difficult to differentiate between tightly packed rectangles when using `xgraph`, so I defined `testR(r1,r2)` function in a way, in which it requires two rectangles to have a distance of at least single unit between them. The source code and `README` have instructions on how to remove this property.

(a) Allowed position (b) Not allowed position

Figure 1: testR(r1,r2) requires at least a single unit of distance

Next, let us define a function, which takes an integer `n` as an input and returns an array of size `n`, with rectangle variables initialized, without rectangles overlapping. After declaring variables, the function begins `makeRectangle: DO` loop, which initializes the variables for each rectangle one at a time. The values of a rectangle are chosen by random, within certain boundaries. The position

of a rectangle is measured from it's center. Width and height are defined to be even numbers as to avoid half-unit distances. After setting the values, an another `DO` loop checks that the new rectangle does not overlap with the ones defined earlier by using the `testR` function. If the new rectangle overlaps with another, the cycle restarts and the function finds new values for the rectangle.

In the SA algorithm of this project, the cost function is defined as the footprint of all rectangles. The aim is to minimize the footprint. Cost function `footp(x)` takes an array of rectangles as an input and returns the measured footprint as a `REAL` type value.

Finally, the SA algorithm will be implemented by creating `SA(x)` function, where x is an array of rectangles. Steps from 1 to 3 of SA algorithm are carried out by setting the initial values. Steps 4 to 10 are carried out inside `MMC: DO` loop.

In this problem, the allowed changes in the system configuration are movement, rotation (90 degrees), and swapping. One of these is chosen randomly by generating a number from 1 to 3. Then, a rectangle (two rectangles in case of swapping) to be transformed is chosen randomly. If the chosen configuration is movement, the direction of movement is also chosen randomly by generating a new random number from 1 to 4. Movement changes one of the rectangles coordinates by a single unit. Turning swaps the height and width values of a rectangle. Swapping swaps the coordinates of two rectangles.

After setting the new configuration, the algorithm has to check that the new state does not contain overlapping rectangles. Although this part would work in the same way after moving or rotating, however, after swapping it works little differently, so each transformation will have to have their own statements for checking. This is due to the fact that to check for overlapping, one has to only check that the transformed rectangle does not overlap with any other. But in case of swapping, there are two transformed rectangles.

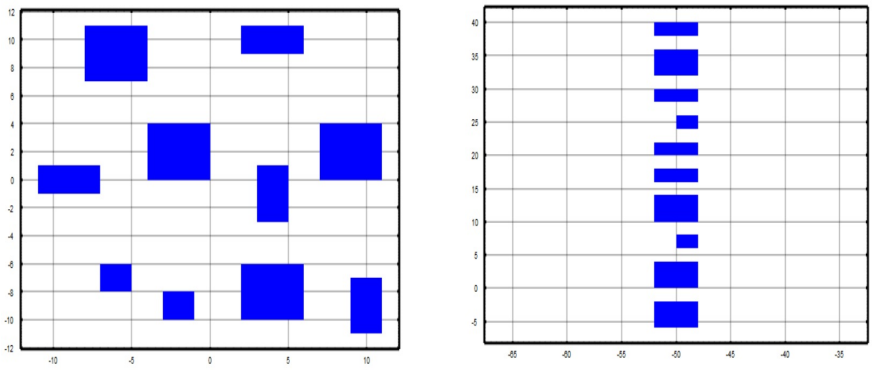
Steps from 5 to 10 are rather straightforward and are implemented at the end `MMC DO` loop

So far, all of the functionalities mentioned above are defined in `SAalgo` module. This module will be implemented in the main program. Subroutine for drawing a given set of rectangles is defined in the main program and functions by drawing lines connecting each angle of a rectangle. `xgraph` has a separate method for drawing rectangles, which I used to graph images in this report. These plots were made on Windows and I could not get the same method to work on Linux.

Running the program returns two `.dat` files: `initial.dat` and `result.dat`. These files can be graphed by using `xgraph` as is instucted in `ReadMe.txt`.

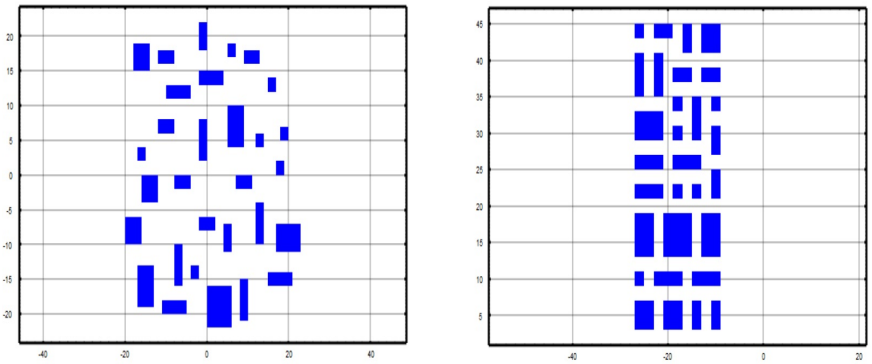
Results

By trying different values for c_0 and α I ended up setting c_0 to 1000 and α to 0.9999. I have attached two sets of problems solved by the SA algorithm. The first is run by generating 10 rectangles and the second by generating 30.



(a) Initial state (b) Final state

Figure 2: 10 rectangles



(a) Initial state (b) Final state

Figure 3: 30 rectangles

Algorithm can be changed to pack the rectangles without the gaps. Instructions on how to do this are written in `ReadMe.txt`. The program is written to generate 10 rectangles, but a user can increase this amount in source code. When changing the amount of rectangles to be generated, it is advised to also increase the area in which they will be generated in `initialize` function in `SAalgo.f90` module.

Conclusions

The algorithm seems to make noticeably fewer swaps, compared to moves or turns. This could imply that swapping two rectangles is more likely to cause overlapping than moving or rotating.

The SA algorithm seems to find optimal footprints quite well. The algorithm could be made more efficient by finding better initial value for the control parameter c and the cooling parameter α.