

A Solution to the Floorplanning Problem Using Genetic Algorithm

(and Speedup Through Multiprocessing)

Sarthak Choudhary (17085066)
Avinash Singh (17095022)

Introduction

- This problem was part of the paper 'High Performance Computing for Cyber Physical Social Systems by Using Evolutionary Multi-Objective Optimization Algorithm'.
- Most of the problems in CPSSs are multi-objective optimization problems, and there is no general approach to solve such problems.
- The paper presented a general approach in the form of evolutionary algorithm, which we've implemented.

Problem Statement

Objective:

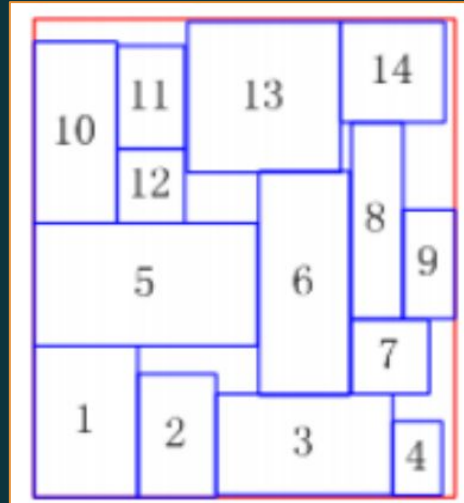
To place a number of blocks on a board
(both with known dimensions).

Constraints:

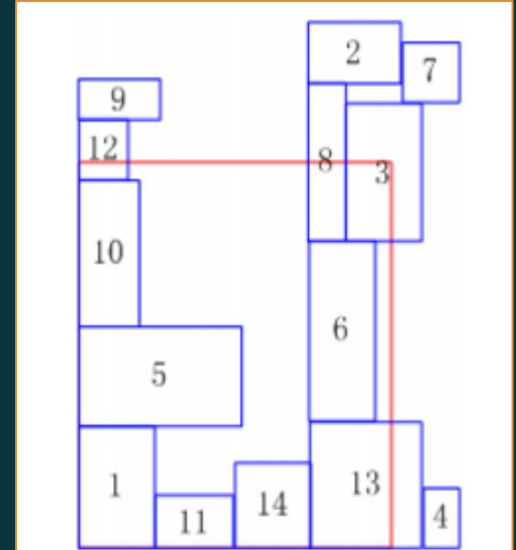
1. The blocks must not overlap
2. Entire block must lie on the board.

To Minimize:

After the blocks have been placed, the
amount of empty space must be as low as
possible.



A valid solution



An invalid solution

Other Methods

The floorplanning problem belongs to the class of combinatorial optimisation problem. Perhaps the most famous problems in this class are the Travelling Salesman Problem, and the Knapsack Problem.

Other popular methods to solve this category of problems are:

1. Branch and bound search
2. Simulated annealing

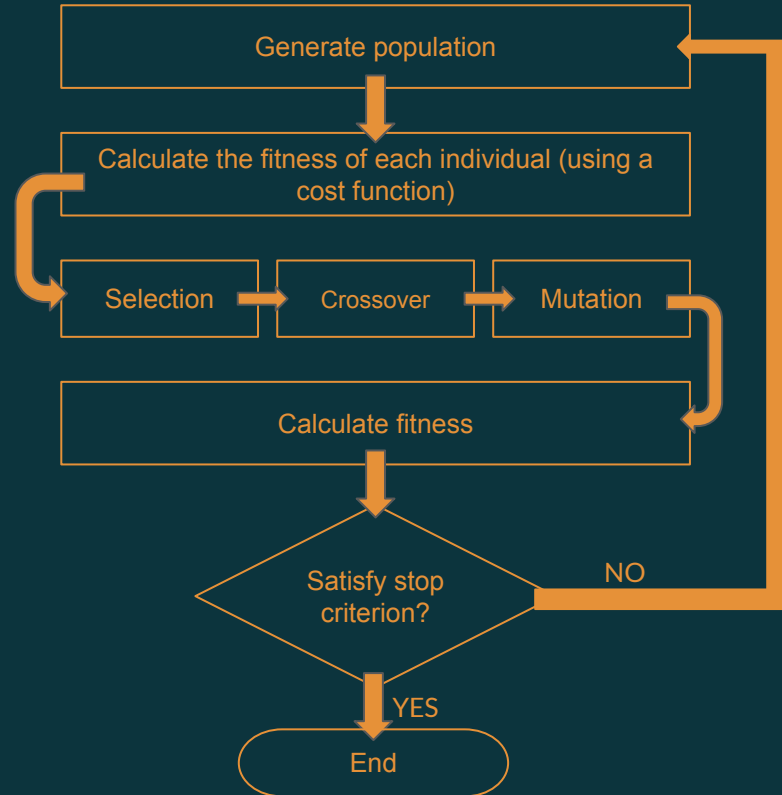
Proposed Solution: Genetic Algorithm

Step 1: Encode data into chromosome

The x and y coordinates of the top-left corner of every block were encoded into a chromosome.

For example: if there are two blocks with x and y positions (2, 3) and (1, 3), the resultant chromosome will be: '10' + '11' + '01' + 11 = 10110111

Step 2: Apply genetic algorithm



Proposed Solution (Slide 2)

The entire board on which the blocks are to be placed is represented using a square matrix (or a 2D array) of 0s and 1s, where 0s mean empty space and 1s mean occupied space.

How are the constraints taken care of?

1. **Overlap:** We iterate over each block and mark the coordinates occupied by it in the board array (by changing the the value to 1. If the value was already 1 (due to a previously placed block) we add to the total cost.
2. **Block lies outside the board:** If the coordinates occupied by the block exceed the dimensions of the board array, we add to the total cost.

```
[[0, 0, 0, 0, 0]
 [0, 0, 0, 0, 0]
 [0, 0, 0, 0, 0]
 [0, 0, 0, 0, 0]
 [0, 0, 0, 0, 0]]
```

Representation of a 5x5 board (initially empty)

Proposed Solution (Slide 3)

How is the fitness calculated?

After adding the costs due to the given constraints, we iterate over each element in the block array to count the number of 0s, and add this count to the total cost.

We also want to penalize constraint violations more severely, so we multiply their additions to the cost the total area of the board. So, all the valid solutions will have cost < Area of the board

$$\text{Total Cost} = \text{Cost due to empty space} + (\text{Area of the board}) * (\text{Cost due to constraint violations})$$

Parallelising

We run the genetic algorithm in a for loop for a given number of times. Each iteration of this for loop performs selection, crossover, and mutation to generate the fittest chromosome.

If we run the program normally, the entire for loop will run on a single processor (or a single core). To parallelise, we've split this into multiple cores. For example, a 1000 iteration for loop will take 500 iteration each on 2 processors, or 250 iterations each on 4 processors.

How did we implement it in python?

We used the multiprocessing module from python's built-in library:

1. First, we created a pool object, which takes in the number of processors as an argument.

```
pool = multiprocessing.Pool(NUMBER_OF_PROCESSORS)
```

2. Then we call the object's map method using two arguments: the function (which we want to parallelize) and an iterable, say an array as arguments. It then passes each element of the array as argument to the given function and runs each of these iterations separately on the specified number of processors.

```
pool.map(genetic_algorithm, [NUMBER_OF_ITERATIONS // NUMBER_OF_PROCESSORS] * NUMBER_OF_PROCESSORS)
```


Demo

Analysis

- Comparison between different selection methods

Best Cost	Tournament Selection	Ranking Selection	Roulette Selection
After 1000 iterations	5 (valid solution)	100+ (invalid)	800+ (invalid)

0	0	2	0	9	9
7	7	7	7	9	9
7	7	7	7	0	4
5	1	3	3	3	4
5	1	0	6	6	4
5	8	8	8	8	4
A solution for a 6x6 board					

- Comparison between sequential and parallel computing

Time Taken	1 Processor	2 Processors	3 Processors	4 Processors
For 100 iterations	1.78s	0.94s	0.64s	0.50s
For 1000 iterations	16.96s	8.77s	5.91s	4.50s
For 10000 iterations	174.08s	86.43s	58.16s	43.79s

Speedup	1 Processor	2 Processors	3 Processors	4 Processors
For 100 iterations	1	1.89	2.78	3.56
For 1000 iterations	1	1.93	2.86	3.76
For 10000 iterations	1	2.01	2.99	3.97

Observations & Result

- Tournament selection performed the best and lead to fast convergence. However, as the complexity of the problem increases (by reducing the amount of empty space via adding more blocks), the algorithm required much more iterations to find a feasible solution.
- Speedup approached linearity as the amount of parallelisation (or the number of processors) increased.

Thank You