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 genetic algorithm, which has been implemented.

Problem Statement

Objective:

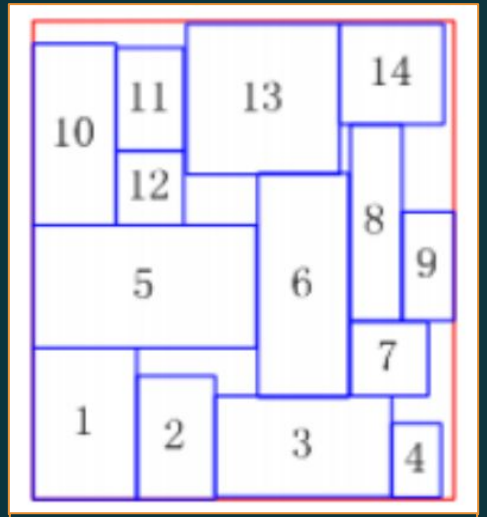
To place a number of blocks on a board (both with known dimensions).

Constraints:

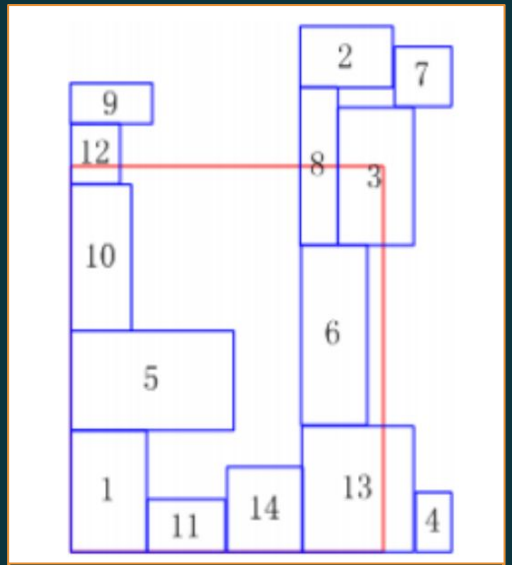
1. The blocks must not overlap
2. Entire block must lie on the board.

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

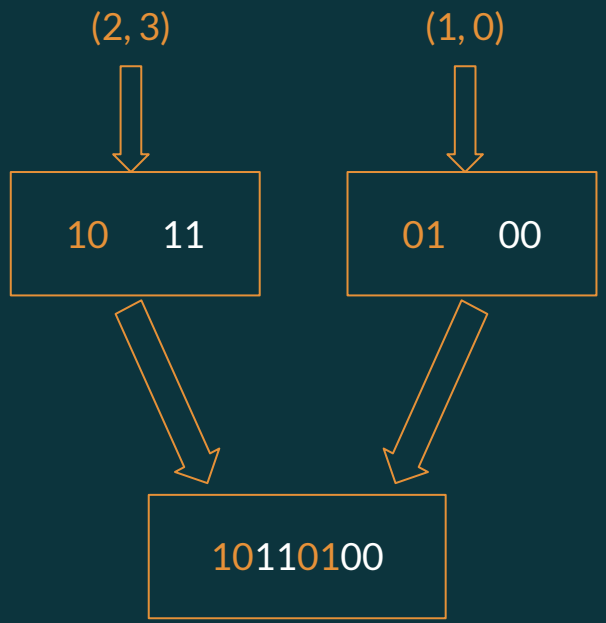
The floorplanning problem belongs to the class of combinatorial optimisation problem. Perhaps the most famous problems in this class are the Travelling Salesman Problem, and the Knapsack Problem.

Other popular methods to solve this category of problems are:

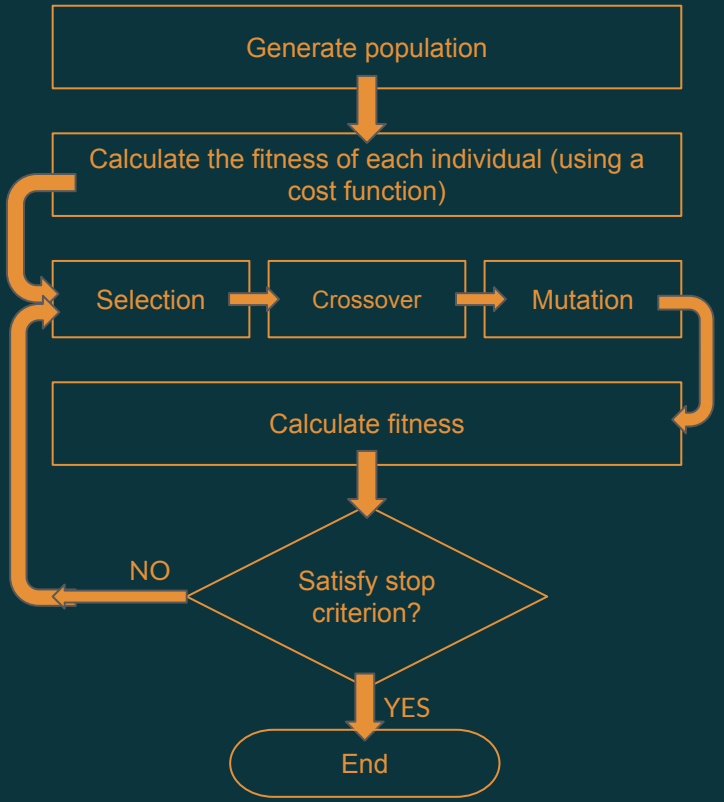
1. Branch and bound search
2. Simulated annealing

Step 1: Encode data into chromosome

The x and y coordinates of the top-left corner of every block were encoded into a chromosome. Consider two blocks of coordinates (2, 3) and (1, 0):



Step 2: Apply genetic algorithm



The blocks are placed on a board represented by a 2D array (or a square matrix) 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)

How is 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 (empty space), 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 with the total area of the board. So, all the valid solutions will have cost < Area of the board

Total Cost = Cost due to empty space + (Area of the board) * (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.

To parallelise, we 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

Observations (Slide 1)

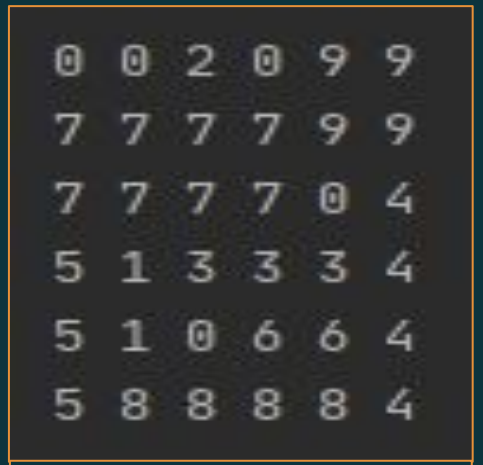
Time Taken	1 Processor	2 Processors	3 Processors	4 Processors
For 100 iterations	1.78s	0.94s	0.64s	0.50s
For 1000 iterations	16.96s	8.77s	5.91s	4.50s
For 10000 iterations	174.08s	86.43s	58.16s	43.79s

Observations (Slide 2)

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

Analysis & Result

- Tournament selection and one-point crossover led to the fastest convergence.
- Speedup approached linearity as the amount of parallelisation (or the number of processors) increased.
- Valid solutions were found quickly for 3 when the unoccupied space is 3 or more. However, increasingly more number of iterations were required as the empty space approached 0.



A valid solution for a 6x6 board

Thank You