Project Phase One Report

## The Problem

The problem that I solved in this project is to implement a distributed course timetable schedular to solve class scheduling problem using genetic algorithm. Class scheduling problem is given a set of courses that must be scheduled into a fixed number of rooms and timeslots. Only one course can be in a room at a time, and the room must be able to accommodate the number of students registered in the course. There cannot be any time or room conflicts, and all students must have a seat. To ensure there are no conflicts, a genetic algorithm (GA) must be employed. This is because this is a NP-Hard problem and cannot be solved by using a deterministic algorithm in polynomial time, and a GA would work well.

There are two types of constraints, hard constraints which must be met and soft constraints which are desirable scenarios but are non-essential. The optimal solutions are satisfying all the hard constraints. The object is to find an optimal solution (meet all hard constraints) and maximize the amount of soft constraints that are met. Constraints established so far are listed below:

Hard Constraints:

* One course per room in a given time slot
* Number of students less or equal to room capacity

Soft Constraints:

* Time preference is met

## Language

In my implementation, I used Golang as the language. The reason I chose to use Golang is that I found some of the code from labs are extremely useful for implementing communication between workers and master through RPC.

The program is similar to lab2, genetic algorithm data types are defined in src/ts/data.go, genetic algorithm functions are defined in src/ts/geneticAlgo.go.

## Methodology

Genetic algorithm is inspired by natural evolution. A generation is a collection of chromosomes, a chromosome is a potential solution to the problem, each chromosome consists of multiple genes, genes are parts of a solution.

In this project, a gene has course id, the room id that is assigned to the course, and the time slot id that is assigned to the course. A chromosome has the number of courses of genes, as each gene represents which room and time slot a unique course is being assigned to.

To evaluate how good a chromosome is, a fitness value f is used. The fitness function to get f is f = (1/(21 \* # of hcv + # of scv))^ 2, hcv is hard constraint violations, scv is soft constraint violations. The lower the value of f, the better the performance. This fitness function ensures a chromosome with less hard constraint violations has a lower fitness value than a chromosome with more hard constraint violations, so that it can be more likely to be chosen as parents for generating chromosome for next generation.

Each generation has 20 chromosomes, each execution has 3000 generations. The initial generation is randomly generated by master node. For each subsequent generation, the master node will copy the individual with lowest value of f from previous generation to the next generation, then the master node will assign workers to generate a new chromosome for next generation. To create a new chromosome for next generation, the worker will pick two parents from previous generation, the lower value f is, the more likely to be chosen as a parent.

After choosing two parents, crossover will be performed. Crossover is performed by picking a random midpoint in a chromosome, the genes before the midpoint will be inherited from parent one, and the rest genes will be inherited from parent two.

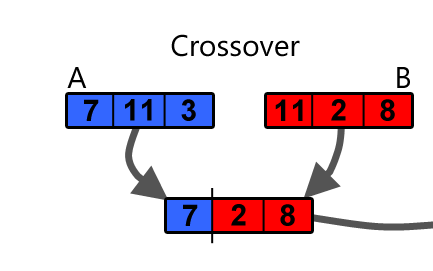


Figure 1 Crossover - <https://becominghuman.ai/understanding-genetic-algorithms-a-use-case-in-organizational-field-2087c30fb61e>

An evolve function is then implemented, with the purpose of attempting to remove room capacity violations by assigning chromosome with room capacity violation to a room that has a higher room capacity.

Then, there is a small chance of mutation. Mutation is just replacing a chromosome with a randomly created chromosome.

Crossover step and mutation step increase chromosome diversity to avoid stuck in local optimization.

The algorithm and data flow are as follows:

1. Master - read data and write into lists
2. Master – random generate initial population, and set it to be current generation
3. Master – copy the best chrom to next generation
4. Master – pass data lists and current generation to workers
5. Workers – Create new chromosome for next generation
   1. Pick parents
   2. Crossover
   3. Evolve
   4. Small chance of mutate
6. Workers – return new chromosome to master and ask for new job
7. Master – Once receive enough chrom for a generation, set new current generation go to step 3
8. Repeat 3000 times (Most likely do not need that many generations to reach global optimization)

## Issues

The major issue that I have encountered is that the program does not always generate a solution that has 0 hard constraint violation, so sometimes a manual restart is needed in order to generate an optimal solution with 0 hcv. This is very weird because when I implement this in a non-distributed way using JavaScript or Python, it always gets a solution with 0 hard constraint. As this is not a showstopper, so I did not invest too much of my time to investigate, but I make this to be one of the objectives for phase two.

## Phase two

The program is already a distributed application, so as for now, the focuses of phase two will be investigating why the program does not always reach an optimal solution like implementing in other programing languages like Python or JavaScript, and analysis the how much faster to generate 3000 generations comparing to a single system. The plan for analyzing is to record the time that is needed to generate 3000 generations with different numbers of workers. We can use only one worker to simulate a single system, instead of implementing a different application in Golang.