**The problem that we are trying to solve (problem statement):**

This project aims to schedule the courses of ECE department in Ohio State University automatically given the following constraints and information related to the offered courses:

**Basic constraints:**

1. The same instructor can’t teach 2 different courses simultaneously.
2. The same room can’t be scheduled for 2 different courses simultaneously.
3. The room scheduled for a specific course must have no less seats than the corresponding course capacity.

**Advanced constraints:**

1. For the courses with course number in the range of 3000~5999, the 3rd digit of the course number represents its study area (e.g. 0-Comm/DSP, 1-Emag, 2-Circuits, 3-SSEP, 4-Power, 5-Control, 6-Computer). The courses with the same study area can’t be scheduled in the same time slots.
2. For the courses with course number in the range of 5000~8999, the 3rd digit of the course number represents its study area (e.g. 0-Comm/DSP, 1-Emag, 2-Circuits, 3-SSEP, 4-Power, 5-Control, 6-Computer). The courses with the same study area can’t be scheduled in the same time slots.
3. Some instructors may have preference of time to teach their courses (e.g. Professor Lee may only have time to teach the course on 10AM-MWF).
4. No more than 2 large volume courses can be scheduled at the same time.
5. No more than 1 extra-large volume course can be scheduled at the same time.

**Information related to the offered courses:**

1. Course: the course is classified by the OSU course ID system, e.g. 5421. Usually the course ID is represented as 4-digit number, the first digit represents course difficulty, the third digit represent course field.
2. Class: a class is a real scheduling object we are dealing with, a course can have multiple classes, taught by different or same instructor.
3. Time Slot: all the available time a course can choose
4. Instructor: a class need an instructor to teach, sometimes we don't know who will be teaching the course, some time we know the instructor will be teaching multiple course, some time the instructor requires some preferred teaching time, we need to consider all the situation
5. Classroom: a classroom is a physical container of the class, we need to know the classroom capacity, classroom type (e.g. Lecture or Lab, does it include computers, etc.)

**The basic idea of this project:**

A course schedule should include the arrangement for all courses offered in one semester, and the arrangement of each class contains the related instructor to teach the course, the room for teaching the course, and the time to teach the course.

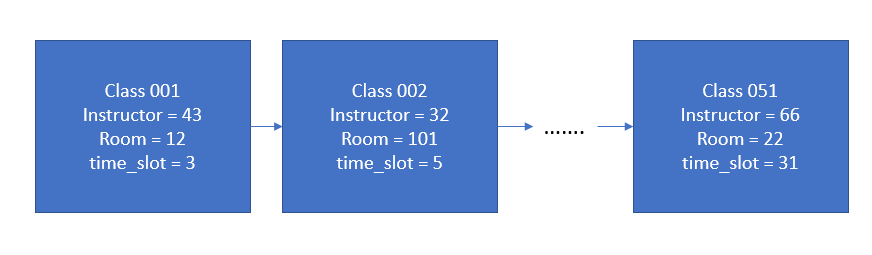


Figure 1 a single chromosome

Each course schedule is treated as a single chromosome in the genetic algorithm, and all the constraints (requirements) of the course schedule is coded to the fitness function.

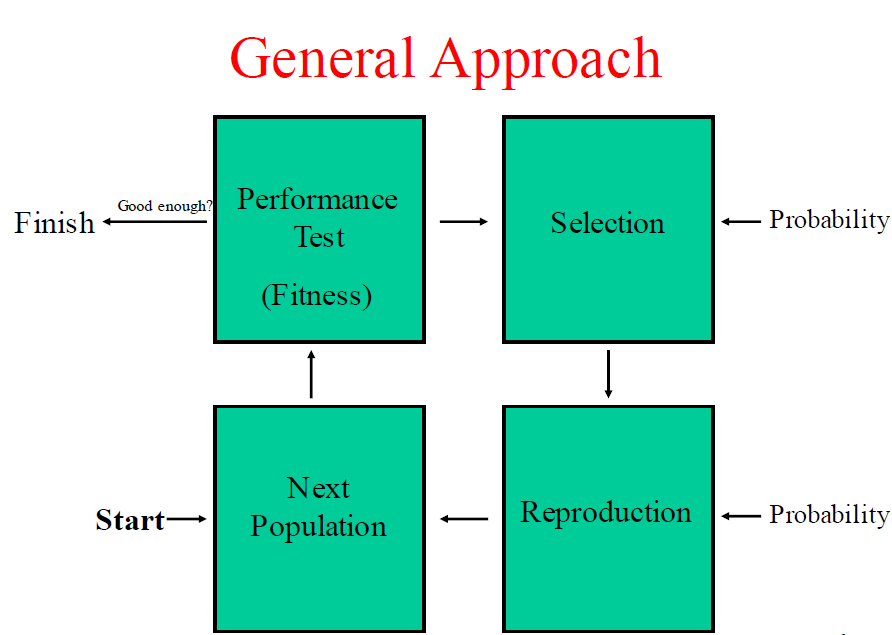


Figure 2

In the initial state, the program would randomly generate a group of chromosomes, each of the chromosome is a possible course schedule, then the fitness score of each chromosome is calculated through the fitness function in the performance test process. The fitness score is a numeric metric used to measure how well the current chromosome (solution of course schedule) satisfies the constraints, the higher the better (in our case, the fitness score ranges in [0,2], 2 is a “perfect” solution regarding the constraints).

The process after performance test is selection, it is a probability-based process. In this process, a roulette wheel selection method is used to choose parent chromosomes in the current generation to mate and generate new chromosomes (i.e. new possible solution of the course schedule) in the following reproduction process.

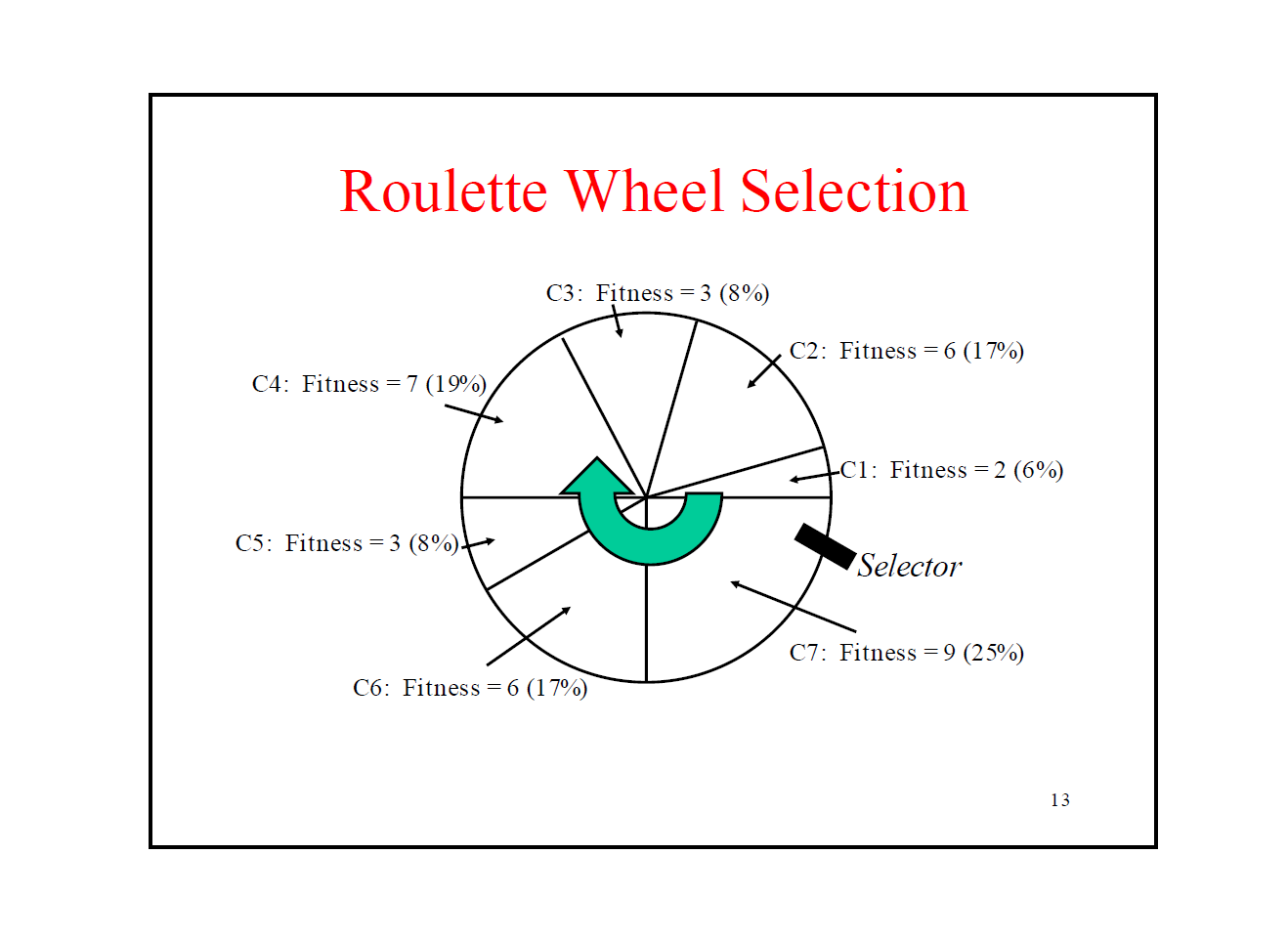


Figure 3 Roulette Wheel Selection

Fig 3 showed the Roulette wheel selection process, which is a probability-based selection. The chromosome with higher fitness score has higher odds to be selected as the parent chromosome to generate new chromosomes for the next generation.

After selection of the parent chromosomes, a mating process is applied to the parent chromosomes to generate new chromosomes.

In our approach, an adaptive mating strategy is used to generate new chromosomes. In the mating process, our program will randomly select some number of points in the chromosome for exchanging purpose. Each point in the chromosome represents a certain class with its arrangement. The number of exchanging points is based on the chromosome’s fitness score in the previous performance test (i.e. the higher fitness score of the parent chromosome, the less number of exchanging points is used, this is similar to adaptive gradient descent, the closer you get to the local optima, the smaller the iterative step becomes).

For example, if we are applying a 1-point mating process to the parent chromosomes, then the program would randomly choose a class (the cross over point), and the mating process will exchange either time\_slot or room of this class between the parent chromosomes to generate 2 new chromosomes as their children chromosomes for the next generation. The following figure is an example of 2-point mating process.

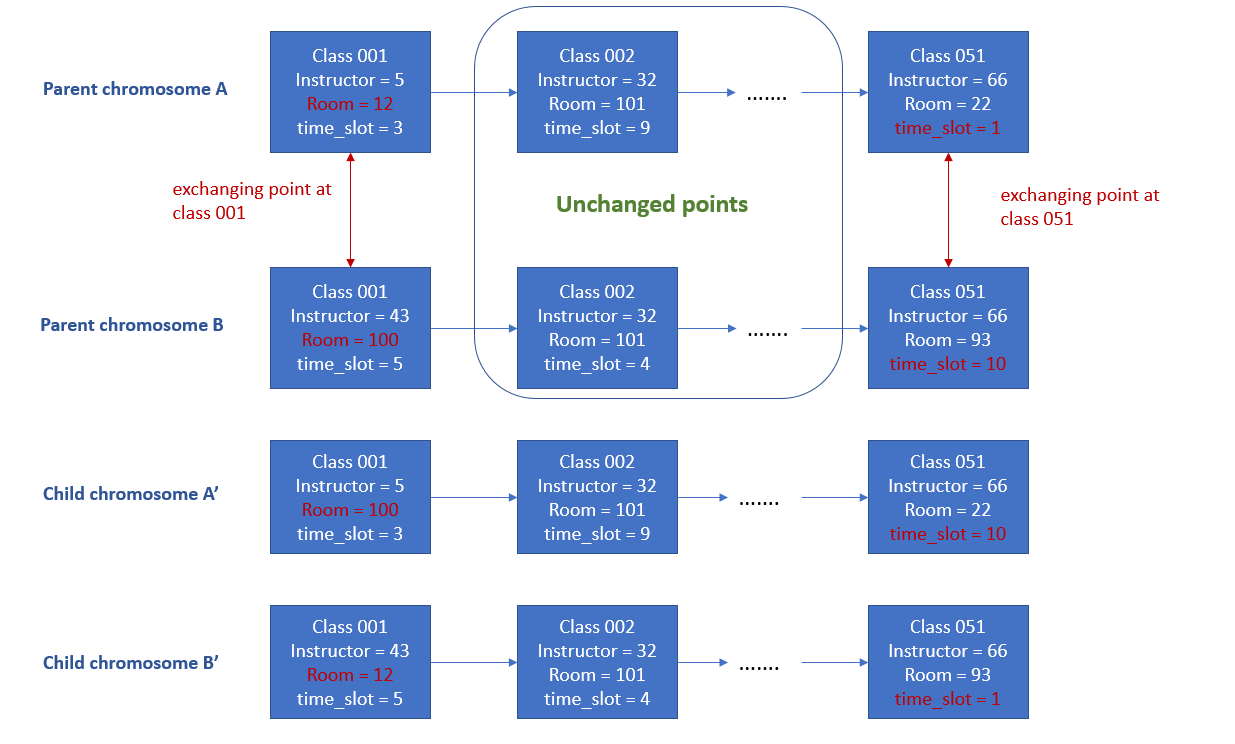


Figure 4. 2-point mating process

Fig 4 showed a 2-point mating process, the 1st exchanging point is class 001, and room arrangement is randomly chosen to be swapped between parent chromosomes, and the 2nd exchanging point is class 051, the time arrangement of this class is randomly chosen to be swapped. In this example, the length of chromosome is 51, it includes course schedule of 51 classes, and the 2 exchanging points is randomly chosen as the first and the last points.

After the reproduction process, the new generation of chromosomes are passed to performance test again and the whole process becomes iterative as shown in figure 2. Once the performance test found a “perfect” chromosome or the iteration has reached a user-defined maximum iteration times, the whole process will stop and output the best chromosome found so far.

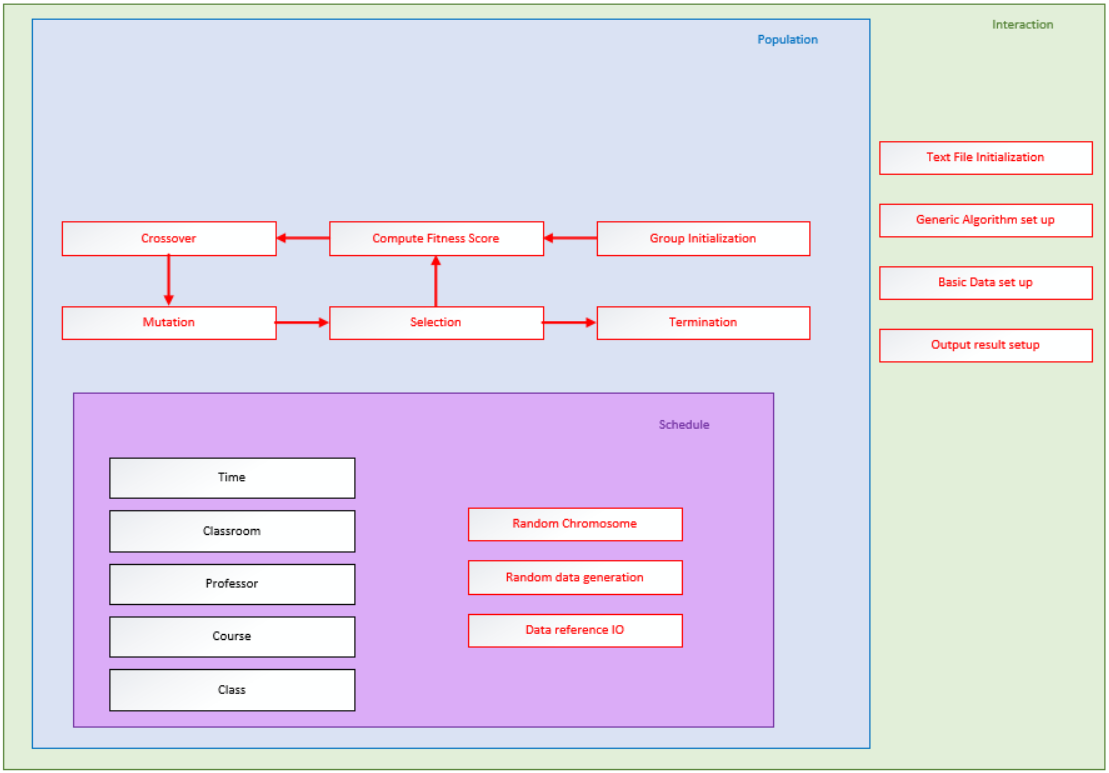
**The work flow of our program:**



**The current progress:**

We’ve implemented basic constraints (1), (2), and all advanced requirements.

**The basic framework of our C++ program:**



**The GitHub website:**

<https://github.com/geneticAlg/CourseSchedule_GA>

**Source Files:**

**reproduceFun.cpp**

1. **vector<vector<Population::\_case>> passDown(double perX, double perY, vector<vector<Population::\_case>> prevGene, vector<double>& prevFScore)**

The passDown() function is used to pass a part of chromosomes from the current generation to the next generation directly without mating or mutation. The chromosomes whose scores rank the top perX% and the chromosomes whose scores rank the last perY% of the current generation will be directly copied from current generation to the next generation.

The reason to “pass down” the best fit chromosomes is to remain the best solution of the problem found so far in the chromosome base. And the reason to keep the worst fit chromosomes is to keep some diversity in the chromosome base.

Input list:

1. perX

The percentage of best fit chromosomes to copy from current generation to the next generation.

1. perY

The percentage of worst fit chromosome to copy from the current generation to the next generation

1. prevGene

The chromosome-base of current generation. A vector that stores all the chromesomes of current generation.

1. prevFScore

The chromosomes’ fitness scores of current generation. A vector that stores all the fitness scores of the chromosomes in current generation. The index of chromosome in the chromosome-base prevGene is the same as the index of its fitness score in prevFScore.

Output

oldChromo, the best fit and worst fit chromosomes that needs to be copied to the next generation.

1. **int reproduceFun(Population \*p, int maxGeneration, double mutationRate, double fitScore\_goal,double perX, double perY)**

The reproduceFun function is the implementation of the iteration cycle showed in figure 2.

This function applies selection of parent chromosomes based on the fitness scores of current generation, and then called matingFun() function to generation new chromosomes from selected parent chromosomes. The newly generated chromosomes are than stored in a temporary chromosome-base newGene. After generation of enough new chromosomes for the next generation, the passDown() function is called to copied the best fitness perX% and worst fit perY% of the chromosomes to the new generation newGene. Finally, the newGene is passed to the chromosome-base of the next generation through method update\_chromosome\_base().

Input list:

1. p

A pointer which points to a Population class, the major class in the genetic algorithm program. This population object stores the requirement information of the course schedule as well as all the potential solutions (chromosomes) of the course schedule problem.

1. maxGeneration

maxGeneration is the variable that specifies the max iteration steps of iteration cycle showed in figure 2. It specifies a maximum number of generations for the reproducing process. This is one of the 2 stop criteria of the iterative while-loop inside reproduceFun().

1. mutationRate

This parameter specifies the mutation rate in the mating process for new chromosomes to mutate.

1. fitScore\_goal

This parameter specifies the goal of fitness score that the program is looking for. Once a chromosome’s fitness score reaches this goal, the reproducing process would stop and output this chromosome to further process to interpret it as a solution of the course schedule problem.

1. perX

The percentage of best fit chromosomes to copy from current generation to the next generation.

1. perY

The percentage of worst fit chromosome to copy from the current generation to the next generation

Output

slnIdx: the solution index, the index of the best fit chromosome in chromosome-base found by the program.

Population.cpp

1. vector<int> conv\_to\_unavi(vector<int>\_avi)
2. Population()
3. ~Population()
4. vector<Population::\_case> Population::get\_random\_chromosome()
5. vector<Population::\_case> Population::get\_random\_chromosome\_time(int\_i)
6. void Population::add\_professor(int i, string name, vector<int> prefer)
7. void Population::add\_course(int I, string name, string \_name\_professor, int \_default\_day, vector<int>\_avaliable, int fixt\_time, int fix\_room)
8. void Population::add\_class\_room(int i, int seats, int \_type)
9. void Population::add\_class(int \_course\_id, string course\_name, string professor\_name, int cap, int type\_num)
10. void Population::set\_time(int i)
11. void Population::print\_chromosome(vector<Population::\_case> \_chromosome)
12. double Population::get\_fitness\_score(void)

This function calculates the fitness score of a single chromosome (i.e. \_chromosome, the private member of class Population). Currently, there are 7 constraints coded in this function for calculating the fitness scores. These constraints are listed in the problem statement section, they are the first 2 basic constraints and all the advanced constraints.

In the get\_fitness\_score() function, the following variables are used to count the violations of the constraints:

1. multi\_c: count the number of violations of the basic constraint #1.
2. Ovrlap: count the number of violations of the basic constraint #2.
3. serious\_Overlap: count the number of violations of the advanced constraints #1 and #2.
4. prefer: count the number of violations of the advanced constraint #3.
5. roomSizeConstraints: count the number of violations of the advanced constraints #4 and #5.

The program counts these violations by scan through the chromosome and check for each type of violation.

The fitness score in this approach is set to 2 stages. In stage one, the fitness score ranges in (0,1]. In this stage, the performance test checks the violations of basic constraints which make the course schedule physically applicable. In stage two, the fitness score ranges from 1 to 2. In this stage, the course schedule is already physically applicable, but some advanced constraints are violated (e.g. the preferred teaching time of some instructors are not satisfied).

where fs stands for fitness score.

1. double Population::get\_fitness\_score\_parallel(void)

This function is also used to calculate the fitness score of variable \_chromosome. In this function, each constraint itself is coded into an independent function for later parallel computing optimization purpose.

1. int Population::studyAreaConflicts(void)

This function counts the number of violations of advanced constraints #1 and #2.

1. int Population::preferenceConflicts(void)

This function counts the number of violations of advanced constraints #3.

1. int Population::multiCourseConflict(void)

This function counts the number of violations of basic constraints #1.

1. int Population::roomOverlap(void)

This function counts the number of violations of basic constraints #2.

1. int Population::roomSizeOverlap(void)

This function counts the number of violations of advanced constraints #4 and #5.

1. int Population::roomSizeStudNumConflicts(void)

This function counts the number of violations of basic constraints #3. **(untested)**

1. void Population::generate\_init\_group(int i)
2. void Population::update\_group\_fitness()
3. void Population::mutateFun(vector<Population::\_case>& chromosome)

This function takes the reference of a chromosome as an input, and randomly select a “point” (arrangement of a class) in this chromosome and randomly choose to mutate the room or time\_slot arrangement of this class.

1. vector<vector<Population::\_case>> Population::matingFun(int id1, int id2, double \_mutationRate)

This function takes the indices of 2 parent chromosomes id1 and id2, and find these chromosomes in the chromosome-base, then randomly select 1 point location (a specific class) in the parent chromosomes and randomly exchange the room or time\_slot arrangement of the class in this selected point location. The newly generated 2 chromosomes are regarded as children chromosomes. And mutationRate specifies the probability of mutation of these 2 children chromosomes.

1. vector<vector<Population::\_case>> Population::matingFun(int id1, int id2, double \_mutationRate, double fScore)

This function is an overload function of the matingFun. Instead of randomly choose 1 point location for exchanging the arrangements. This function use fScore as a reference to generate an adaptive number of exchanging points. And randomly select this number of exchanging points for mating purpose. This is an acceleration strategy to help speed up the process of finding correction chromosomes. The lower fScore is, the higher number of exchanging points would be applied for mating process.

1. vector<vector<Population::\_case>> Population::matingFunAdapative(int id1, int id2, double \_mutationRate, double fScore)

This function is an update of function (24). It ensures that all the exchanging points that randomly selected for mating are different points, no duplicates exist using random\_shuffle.

**(untested)**

1. vector<vector<Population::\_case>> Population::pick\_random\_chromosom()
2. pair<int,int> Population::randSelect()

This function is the implementation of probability-based Roulette Wheel Selection process. It ensures that the chromosome whose fitness score is higher gets higher chance to be selected as the parent chromosome to reproduce new chromosomes for the next generation.

1. int Population::binarySearch(vector<int> section, int idx)

The function name is a mislead, this function is applying a linear search. The linear search tries to find out which section specified by variable section does idx fall into.

1. int Population::binarySearchNew(vector<int> section, int idx, int range)

This function is applying a binary search to fulfill the same function implemented by function (28).

1. void Population::print\_schedule(vector<Population::\_case> sch, int \_time\_s)
2. void Population::set\_prof\_preference(int prof\_id, vector<int> \_time)
3. unordered\_set<int> Population::get\_prof\_preference(int prof\_id)
4. void Population::printConflict(vector<Population::\_case>\_chromosome)

This function is used to output the violation of constraints in variable \_chromosome into text file.

1. void Population::printConflict()

This function is used to output the violation of constraints in variable \_chromosome into text file.

1. void one\_line\_initiation(string \_file\_name, int \_group\_size, int \_max\_gen, double \_mute\_rate, double \_standard, double \_save, double \_switch\_rate)
2. void Population::t\_ind\_conv()