GIIE Course: 546U4030 SOFT COMPUTING METHODS AND APPLICATIONS

Assignment 08 (2020)
Preliminary Design of a Generic GA Solver

Since the type of a gene in encoding a chromosome might be integer, 0-1 binary value, and real number, the generic GA solver can be designed as a template class that takes a type parameter for specifying data type of the gene value. Normally, we set this type parameter with variable name T. Therefore, the class name of the generic GA solver might be `GeneticAlgorithm<T>.` Although a 2-dimensional rectangular matrix can be defined for the set of chromosomes, a list of one-dimensional chromosome array is preferred for evaluation of an individual chromosome. Therefore a jagged 2-D array is defined for the chromosomes; i.e.,

```
T[][] chromosomes;
```

To record the objective values and fitness values of all of the chromosomes, double arrays

`double[] objectiveValues; double[] fitnessValues;` should be defined. In addition, to record so far the best solution and objective, define

```
T[] soFarTheBestSolution; double soFarTheBestObjectiveValue;
```

Note that the constructor of the `GeneticAlgorithm<T>` class should ask for the number of genes, optimization type, and the delegate that can be called to evaluate a solution of type `T[]` and return its objective value. To efficiently use memory in GA evolutional computation, required data structure should be effectively designed. This assignment asks you to design a generic GA solver implementing required generic data and utility functions. Sepecific calculations will therefore be overridden in the derived classes.

During the genetic evolution, there exists three sets of chromosomes, the population (parents), crossovered children, and mutated children. Three integer numbers are therefore defined to record numbers of chromosomes in these sets. Let `populationSize`, `numberOfCrossoveredChildren`, and `numberOfMutatedChildrenbe` be these number. The `populationSize` is specified by the user and its value is thereafter fixed throughout the evolution while `numberOfCrossoveredChildren` and `numberOfMutatedChildrenbe` are calculated and changed in run-time.

Moreover, we prefer to allocate a large chunk of memory to accommodate these sets of chromosomes as a group and not to re-allocate memory for (newing) them during the run-time. Therefore, only one set of chromosomes is allocated once, whose size is 3 times `populationSize`. The first `populationSize` chromosomes are for the population. Then the following `numberOfCrossoveredChildren` chromosomes are crossovered offspring and finally, they are followed by `numberOfMutatedChildrenbe` mutated children. Similarly, `double` type arrays for `objectiveValues` and `fitnessValues` are allocated with a size of 3 times `populationSize`, corresponding to the aggregated chromosomes.

In the selection operation we need to pick `populationSize` better chromosomes from these three sets and overwrite the first set to constitute the population of the next generation. In order not to obscure the

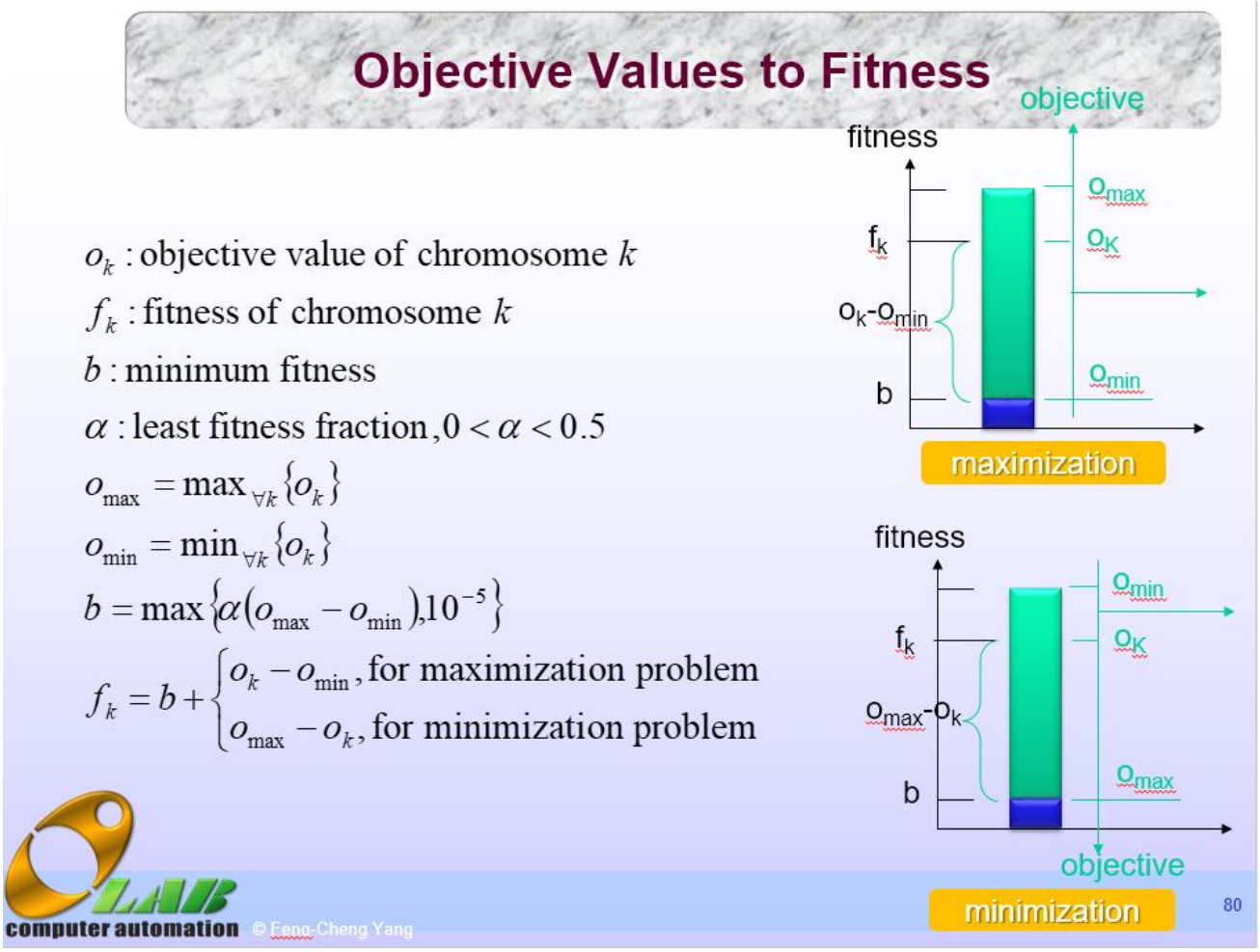
candidates in the first set during the overwriting and for easier implementation, the selected chromosomes as well as their objective values will be assigned to another chunk of memory. When the selection is completed, they are gene-by-gene cloned back to overwrite the original population (the first set). Therefore, a 1-D array of 1-D chromosome array (a 2-D jagged array) is defined for storing the selected chromosomes, whose size is `populationSize`; i.e.,

```
T[][] selectedChromosomes;
```

In addition, a double array `double[] selectedObjectives;` is defined to store the objective values of the selected chromosome.

In your GA solver, the optimization goal is known to be either minimization or maximization. Prepare a function to transfer objective values to fitness. Note that the fitness of chromosomes will serve as the surviving probabilities in genetic selection. Which means the values must be positive and the worst chromosome should receive a minimal amount of positive probability. Fitness values are transformed from objective values, suppose o_k is the objective value of chromosome *k* and f_k is the fitness of chromosome *k*. (1) Derive your transformation equation to calculate f_k from o_k and then (2) implement this function:

```
void setFitnessFromObjectives();
```



In GA, *deterministic* selection is to choose the best (based on fitness) `populationSize` chromosomes from

the parent and offspring. On the other hand, *stochastic* selection generates a random number to perform each probability-proportional selection, where the fitness values are the selection probabilities of chromosomes. Note that a chromosome might be selected multiple times in stochastic selection. To record the indices of the selected chromosomes, we prepare an integer array, namely `indices`, with the same size of the objective and fitness arrays. The indices of the selected `populationSize` chromosomes will be placed on the first `populationSize` locations of the `indices` array. Your mission is to (3) implement functions:

```
void DeterministicSelection();
```
and

```
void StochasticSelection();
```

which conduct genetic selections to write indices of selected chromosomes to the first `populationSize` position of the `indices` array. After that (4) implement the

```
void PerformSelectionOperation();
```

function that calls either function DeterministicSelection() or StochasticSelection() to determine selected indices. Referring to the selected indices, gene-wise copy the genes of the selected chromosomes to `selectedChromosomes` array. Note that, their objective values are also copied to `selectedObjectives.` Finally, these selected gene values and objectives are then gene-by-gene copied back to the first `populationSize` parent chromosomes and objectives in `chromosomes` and `objectiveValues.`

In this assignment, create a .NET Framework library project to host the related classes of GA. Then add a WinForm application project to have a UI to test these implemented functions. Note that in this development stage, you can set all the data fields of the GA solver `public` for easier value assignments.

From the user's perspective (main form) three public functions (methods) of the `GeneticAlgorithm <T>` should provide: `Reset(); RunOneIteration(); RunToEnd(). Reset()`is called to allocate necessary amount of memory for those data related to GA parameters, such as population size. Then virtual initialization operation `initializePopulation`() is conducted to randomly assign values to the variables. In `RunOneIteration()`function, genetic operations `performCrossoverOperation`(), `performMutationOperation`(), `performSelectionOperation` (), and `updateSoFarTheBestObjectiveAndSolution`() are sequentially executed. In `updateSoFarTheBestObjectiveAndSolution()` you need to call `computeObjectiveValues`(), `setFitnessFromObjectives`(). Define public properties of a GA solver that can be displayed on a PropertyGrid UI control and their values can be modified by the user.

Several termination conditions can be implemented to stop the solution evolution; for example, limit of iterations (generations), limit of objective function calls, limit of CPU execution time, etc. Define properties IterationCount, IterationObjectiveAverage, IterationBestObjective for progress display.

Star implementing a binary encoded GA solver that inherits `GeneticAlgorithm <byte>`; e.g., `BinaryGA : GeneticAlgorithm< byte >` Take the advantages of the polymorphism mechanism of O-O techniques to implement virtual and override functions to complete the functionalities of the derived class. We

will later use the binary GA to solve the job assignment problem with a model similar to the mathematical programming method done in Excel.

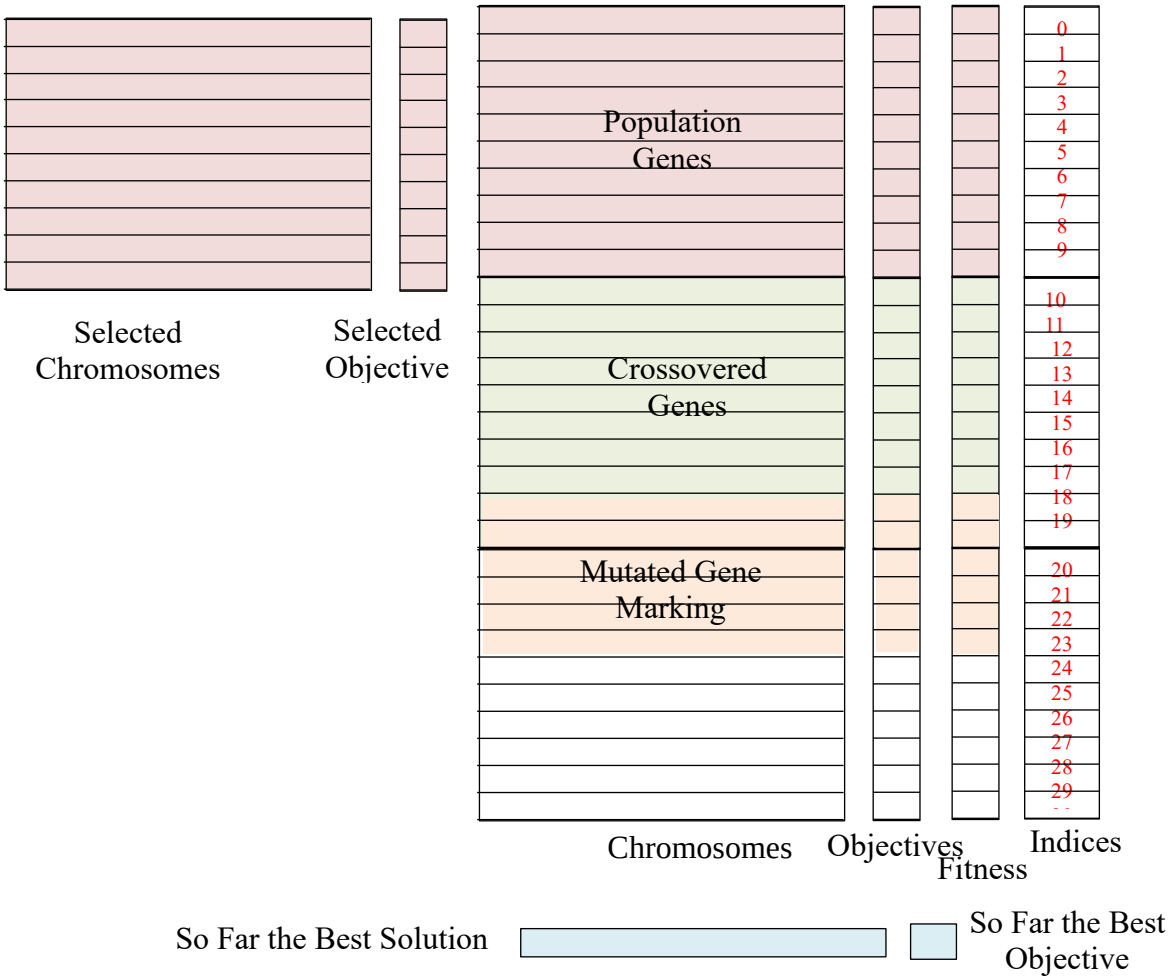


Fig. A. Suggested Data Structure for Implementation

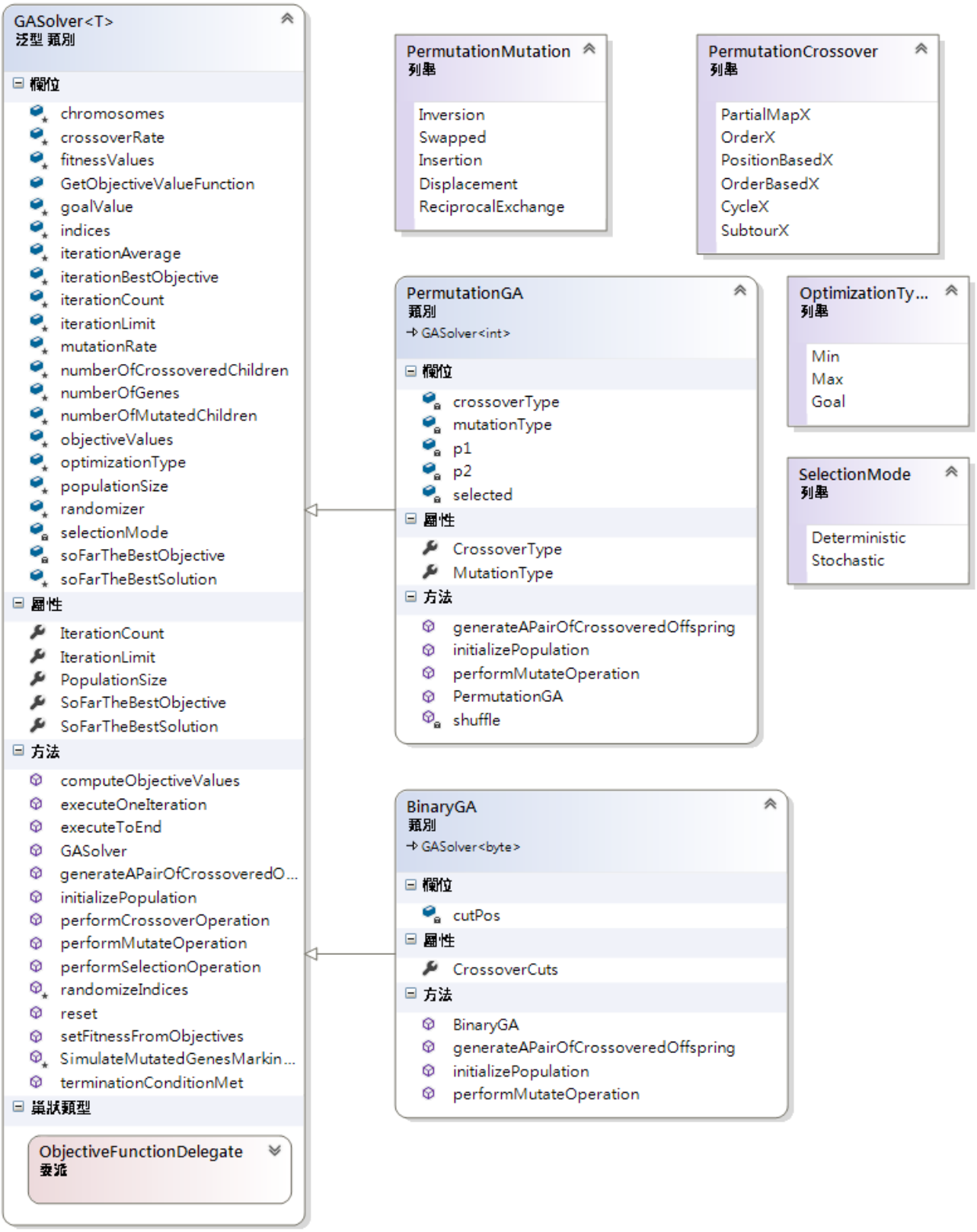


Fig. B. Class Design (reference only)