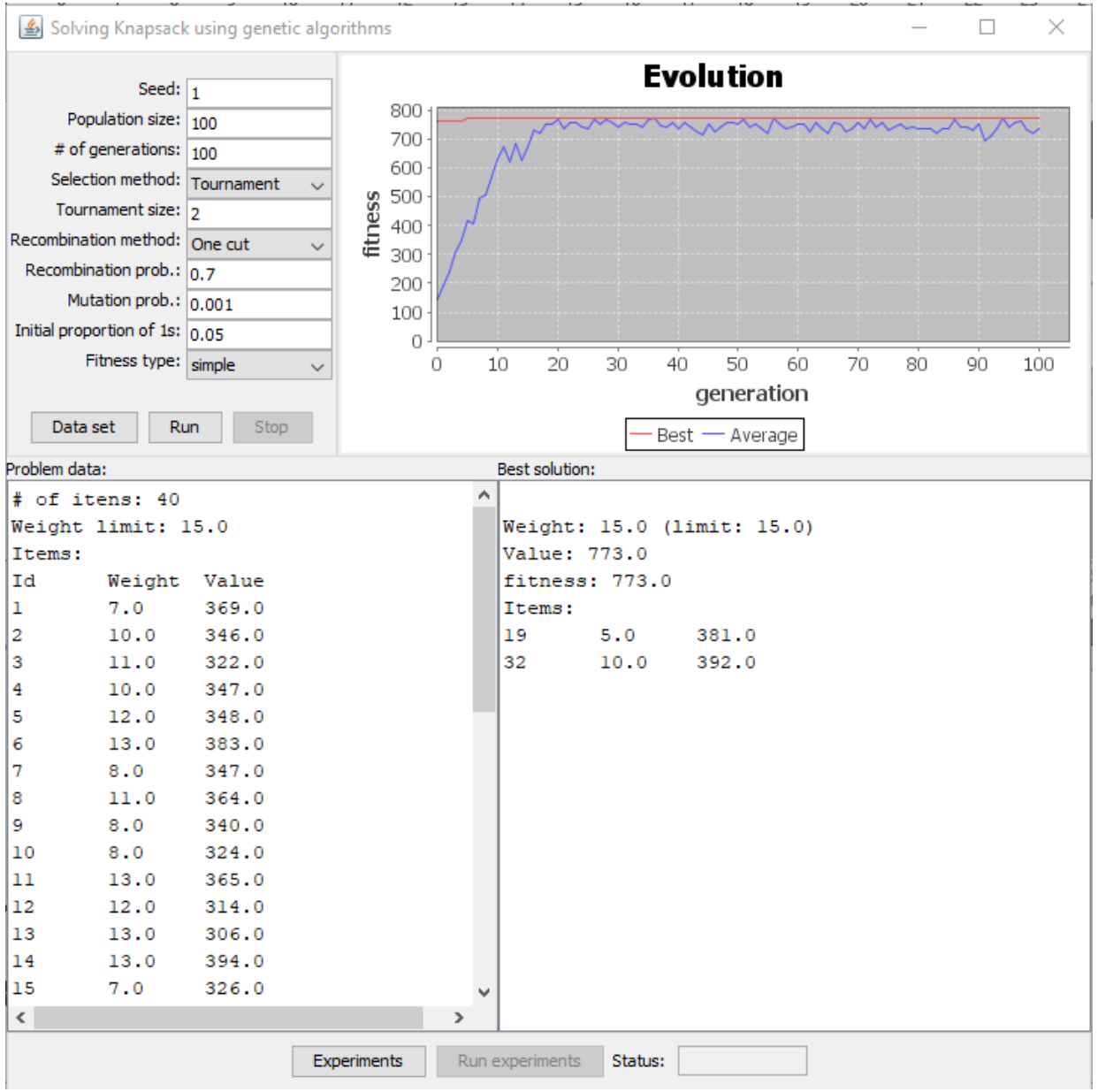
# Artificial Intelligence

## Genetic Algorithms, worksheet 3 exercises

Practical classes guide

# The program GUI

- After completing the practical sheet, this will be the program's GUI
- It has 5 panels:
    1. Up left: where the user can choose the problem dataset, define the algorithm parameters and run it
    2. Down left: where dataset data is presented
    3. Up right: chart that presents the evolution of the average fitness of the population and the fitness of the best inidividual found so far
    4. Down left: where the best found solution is shown
    5. Bottom: panel that allows the user to run several experiments on backround



# Project overview

- As in previous projects, the genetic algorithms project is designed so that there is a clear separation between the Model and the View (GUI)
- As always, we will be working on the model only
- Also, the genetic algorithm is designed so that minimum changes should be made from problem to problem:
  - For example, the genetic algorithm is generic and you don't need to change it once implemented; this is the same for selection methods

# Packages

- The project is composed of the following packages:
  - `experiments`: this package contains classes that allows the user to run several experiments on backround; We will not be working on these classes
  - `ga`: this package contains classes needed to run an instance of a genetic algorithm; it also contains subpackages `geneticOperators` and `selectionMethods` that contain classes representing genetic operators and selection methods, respectively
  - `gui`: this package contains the classes related to the GUI
  - `knapsack`: this package contains classes specific to the knapsack problem
  - `statistics`: this package contains classes able to compute and save statistics when several experiments are run on backround
  - `utils`: this package contains classes for file managing and mathematical operations
- At least for now, we will be working on the classes belonging to packages `ga` and `knapsack` only

# The GeneticAlgorithm class

- The `GeneticAlgorithm` class is the core of the whole project

```
public class GeneticAlgorithm<I extends Individual, P extends Problem<I>> {
```

- Using generics, it works on
  - some type of individual I descending from the abstract class `Individual`
  - some type of problem P implementing the `Problem` interface
- The `Individual` class represents a generic individual: it has a `problem` and a `fitness` value and methods `computeFitness()`, `getNumGenes()` and `swapGenes()` that all non-abstract subclasses should implement*
- The `Problem` interface defines the `getNewIndividual()` method that should be implemented in order to generate new random individuals

*Please, notice that the `Individual` class doesn't define the individual's genotype; this is because individuals representation is strongly dependent on the problem to be solved; so, the genotype must be defined on subclasses

# The GeneticAlgorithm class

- The `GeneticAlgorithm` class has the following attributes:
  - `random`: it should be used to generate random numbers during the algorithm's execution; it is static so that it can be easily accessed by other classes (e.g., selection methods and genetic operators)
  - `populationSize`: the number of individuals in the population
  - `maxGenerations`: the number of generations (iterations) after which the algorithm stops
  - `population`: the population of individuals (see slide 8)
  - `selection`: the selection method to be used (see slide 9)
  - `recombination`: the recombination method to be used (see slides 10 and 11)
  - `mutation`: mutation method to be used (see slide 10 and 12)
  - `t`: the current iteration
  - `stopped`: flag that indicates if the algorithm has been stopped be the user
  - `bestInRun`: the best individual found so far during the execution of the algorithm

# The GeneticAlgorithm class

- The `GeneticAlgorithm` class has the following methods (besides the constructor):
  - `public I run(P problem)`: it receives a problem to be solved, runs the algorithm and returns the best individual found
  - `private void computeBestInRun(I bestInGen)`: it receives the best individual of the current generation and, if it is better than the current best individual of the run, it becomes the best individual of the run; this method should be called in the `run()` method
  - Other auxiliary and self-explanatory methods

# The Population class

- The population class represents the population of individuals:
- It has the following attributes
  - `individuals`: the individuals of the population
  - `best`: the best individual of the population
- It has the following constructors:
  - `public Population(int size)`: it builds an empty population with the given size; it should be used when constructing population P'(t) in the selection method
  - `public Population(int size, P problem)`: it builds a population with the given size; individuals are generated using the `getNewIndividual()` method from the problem instance; please, notice that the generation of individuals is stronlgy problem dependent; it should be used to create the initial population

# The SelectionMethod class

- The abstract `SelectionMethod` class represents a selection method
- It has attribute `popSize`, which represents the population size
- It has the `abstract Population<I, P> run(Population<I, P> original)` method
  - This method should receive the current population – P(t) – and apply the selection method to build the P'(t) intermediary population
- We will work with two selection methods: roulette wheel and tournament; The first is already implemented in the `RouletteWheel` class (please, analyze the implementation); the second one should be later implemented by you on the `Tournament` class

# The GeneticOperator class

- The abstract `GeneticOperator` class represents a genetic operator
- It has the `probability` attribute, which represents the operator application probability

# The Recombination class

- The abstract `Recombination` class represents a recombination method
- It has the following methods:
  - `public void run(Population<I, P> population)`: it receives population P'(t), resulting from the selection method application; it iterates the population, two individuals at a time; for each pair of individuals, it generates a random number in the interval [0, 1] and it applies the operator to the two individuals if the random number value is smaller than the operator's ocurrence probability (see the `GeneticOperator` class)
  - `public abstract void recombine(I ind1, I ind2)`: this method should be implemented by non-abstract subclasses; it applies the operator to the two input individuals
- We will work with three recombination methods: one cut recombination (`RecombinationOneCut` class, already implemented), two cut recombination (`RecombinationTwoCuts` class, already implemented) and uniform recombination (`RecombinationUniform` class, to be implemented by you)

# The Mutation class

- The abstract `Mutation` class represents a mutation method
- It has the following methods:
  - `public void run(Population<I, P> population)`: it receives population P''(t) after recombination; it iterates the population, one individual at a time; for each individual, it calls the mutate method
  - `public abstract void mutate(I ind1, I ind2)`: this method should be implemented by non-abstract subclasses; it applies the operator to the input individual
- We will work with the classic binary mutation method (`MutationBinary` class, to be implemented by you)

# Exercise 1a

- Please, implement the run() method in the GeneticAlgorithm class (see the algorithm to be implemented in the next slide)
- Some hints:
  - Call the `fireGenerationEnded()` method as follows after the initial population evaluation and in the end of each iteration so that the GUI is refreshed

```
fireGenerationEnded(new GAEvent(this));
```

  - In the while condition, take into consideration the `stopped` flag besides the t value
  - The population returned by the selection method should be assigned to a auxiliary population; genetic operators will operate over this auxiliary population; this means that this population is used both as P'(t) and P''(t)
  - The create_new_population(P(t), P''(t)) step consists simply in assigning the auxiliary population to the population attribute
  - Don't forget to compute the best individual in the run by calling method computeBestInRun() after the evaluation of the population in the cycle (this is different in the evaluation of the first population; how different?)
  - Call the `fireRunEnded()` method before returning the best individual found so that the GUI is refreshed

```
fireRunEnded(new GAEvent(this));
```

# evaluate(P(*t*))

```
t = 0
create_initial_population(P(t))
evaluate(P(t))
while stop condition not met
    P'(t) = select(P(t))
    P''(t) = apply_genetic_operators(P'(t))
    P(t + 1) = create_next_population(P(t), P''(t)))
    evaluate(P(t + 1))
    t = t + 1
return best individual found
```

# Exercise 1b

- Please, implement the `tournament()` method in the `Tournament` class
- This method simulates one tournament of T elements (T is represented by the `size` variable); it returns a clone of the individual that wins the tournament
- The `run()` method of this class is already implemented; it calls the `tournament()` method `popSize` times; each time, it chooses one more individual and adds it to the population of selected individuals (P'(t))
- Hints:
  - Use the `random` attribute from the `GeneticAlgorithm` class to generate random numbers
  - Use the `compareTo()` method of the I class to compare two individuals

# Exercise 1c

- Please, implement the `recombine()` method in the `RecombinationUniform` class
- This method iterates over the genes of the individuals; for each index, it generates a random boolean value; if it is true, the two corresponding genes are exchanged
- Hints:
  - Use the `random` attribute from the `GeneticAlgorithm` class to generate random boolean values
  - Use the `swapGenes()` method from class I to swap genes between individuals

# Exercise 1d

- Please, implement the `mutate()` method in the `MutationBinary` class
- This method iterates over the individual's genes; for each gene, it generates a random value in the [0, 1] interval; if it is smaller then the mutation probability value, the gene value is flipped
- Hints:
  - Use the `random` attribute from the `GeneticAlgorithm` class to generate random values
  - Use the `getGene()` and `setGene()` methods from class I to flip the gene value

# The Item class (knapsack package)

- The Item class represents an item of the knapsack problem
- Each item has a name, a weight and a value

# The Knaspck class

- The Knapsack class represents a knapsack problem
- It has the following attributes:
  - `items`: an array with all the items
  - `maximumWeight`: the maximum weight we can put in the sack
  - `prob1s`: the probability of generating a 1 valued gene when creating new individuals in the getNewIndividual() method
  - `fitnessType`: value 0 indicates that simple fitness (with no penalty) will be computed; value 1 indicates that the fitness with penalty will be computed
  - `maxVP`: it represents the maxVP value (see slides on the knapsack problem)

# The Knapsack class (continued)

- It has the following methods:
  - `public KnapsackIndividual getNewIndividual()`: it creates a new individual; it is used to create the individuals of the initial population
  - Self-explanatory auxiliary methods

# The BitVectorIndividual class

- The `BitVectorIndividual` class (package `ga`) represents a binary individual; it is implemented because the binary representation is a common one
- For example, our `KnapsackIndividual` (see next slides) extends this class because we will use a binary representation in the knapsack problem
- As attribute, it has an array of boolean values; we will consider that the true value corresponds to 1 and false to 0
- It has two constructors:
  - `public BitVectorIndividual(P problem, int size, double prob1s)`: it is used to build the individuals of the initial population; the probability of a gene being 1 is given by the `prob1s` argument
  - `public BitVectorIndividual(BitVectorIndividual<P, I> original)`: this is the copy constructor which allows us to build a copy of another individual
- The methods are self-explanatory

# The KnapsackIndividual class

- The KnapsackIndividual class represents an individual to be used by the genetic algorithm to solve the knapsack problem
- It has the following attributes:
  - `value`: the sum of the value of the items to be put in the sack (the items whose corresponding gene has value true)
  - `weight`: the sum of the weight of the items to be put in the sack (the items whose corresponding gene has value true)

# The KnapsackIndividual class (cont.)

- It has the following methods:
  - `public double computeFitness()`: it computes the individual's fitness; it should be later implemented by you
  - public int compareTo(KnapsackIndividual i): it returns 0 if the individual has the same fitness as i, 1 if its fitness is greater than i's fitness and -1, otherwise
  - public KnapsackIndividual clone(): returns a copy of the individual

# Exercise 1e

- Please, implement the `computeFitness()` method in the `KnapsackIndividual` class
- Hints:
  - The method should start by computing the sum of all weights and all values
  - It computes the simple fitness or the fitness with penalty, depending on the value returned by the `getFitnessType()` method from the `Knapsack` problem
  - For the fitness with penalty, use the `getMaxVP()` method from the `Knapsack` problem

# Exercise 2

- You should perform tests that allow you to study the following aspects:
  - The performance of the algorithm when the roulette wheel selection method is used
  - The performance of the algorithm when the tournament selection method is used
  - The performance of the algorithm when one cut recombination is used
  - The performance of the algorithm when two cuts recombination is used
  - The performance of the algorithm when uniform recombination is used
  - The effect of varying the tournament size
  - The effect of varying the population size
  - The effect of varying the probabilities of the genetic operators
- You should perform at least 30 independent experiments with each algorithm configuration so that the obtained results have some statistical meaning

# Exercise 2 (cont.)

- In order to solve this exercise, you should first create a text file describing the experiments to be done
- You may find already three examples of these files in the project root directory
  - config_selection.txt
  - config_tournament_size.txt
  - config_tournament_size_and_one_cut_recombination.txt
- In the next slide you find a description of the structure of this type of files

# Exercise 2 (cont.) – file example

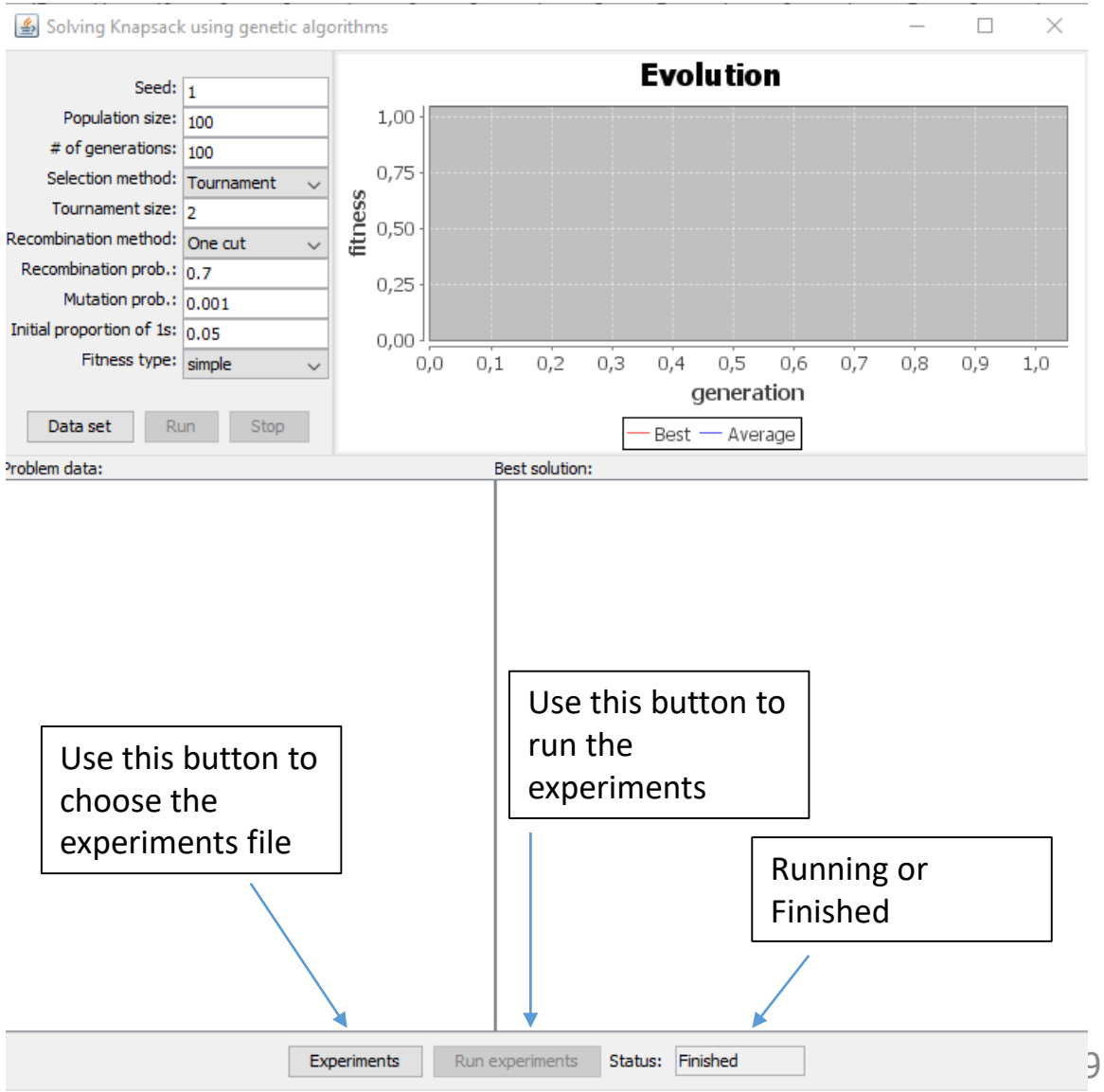
```
Runs: 50
Population_size: 100
Max_generations: 100
//--------------------------------
Selection: roulette_wheel, tournament
Tournament_size: 2, 3, 4
//--------------------------------
Recombination: one_cut
Recombination_probability: 0.7
//--------------------------------
Mutation: binary
Mutation_probability: 0.025
//--------------------------------
Probability_of_1s: 0.05
Fitness_type: 0
//--------------------------------
Problem_file: ./DataSets/DataSet1.txt
//--------------------------------
Statistic: BestIndividual
Statistic: BestAverage
```

- Each file consists in a set of attributes and its value(s)
- The program will run experiments with all possible combinations of values
- In this example, each experiment (that is, each possible combination of values) will be run 50 times
- The “Tournament size” attribute will be used only when experiments with the tournament selection method are performed
- In this example, the dataset ./DataSets/DataSet1.txt will be used
- For each experiment, the following statistics are computed:
  - BestIndividual: an excel file with the fitness of the best individual found in each experiment is created; a text file describing the best individual found in each experiment is also created
  - BestAverage: an excel file with the average and standard deviation of the best individuals found in each experiment (for example, in this example, the fitness of the best individual found in each of the 50 runs done in each experiment is taken and then averaged

# Exercise 2 (cont.) – valid values

Runs: one positive integer value (you can use more than one value but it doesn't makes sense)
Population_size: one or more positive integer value separated by a comma
Max_generations: one or more positive integer value separated by a comma
//--------------------------------
Selection: roulette_wheel, tournament
Tournament_size: one or more positive integer value separated by a comma
//--------------------------------
Recombination: one_cut, two_cuts, uniform
Recombination_probability: one or more values in the [0, 1] interval separated by a comma
//--------------------------------
Mutation: binary
Mutation_probability: one or more values in the [0, 1] interval separated by a comma
//--------------------------------
Probability_of_1s: one or more values in the [0, 1] interval separated by a comma
Fitness_type: 0, 1
//--------------------------------
Problem_file: ./DataSets/DataSet1.txt or some other file describing the dataset (see examples in the DataSets directory)
//--------------------------------
Statistic: BestIndividual
Statistic: BestAverage

# Exercise 2 (cont.)



# Exercise 2 (cont.)

- After finishing the experiments, the result files are saved in the root project directory
- Example of the statistic_average_fitness.xls file for the config_tournament_size.txt file:

| A | B | C | D | E | F | G | H | I | J |
|---|---|---|---|---|---|---|---|---|---|
| Population size: | Max generations: | Selection: | Recombination: | Recombination prob.: | Mutation: | Mutation prob.: | | Average: | StdDev: |
| 100 | 100 | Tournament(2) | One cut recombination (0.7) | 0.7 | Binary mutation (0.025) | 0.025 | | 874.38 | 151.6995570197883 |
| 100 | 100 | Tournament(4) | One cut recombination (0.7) | 0.7 | Binary mutation (0.025) | 0.025 | | 1146.98 | 4.319675913769459 |
| 100 | 100 | Tournament(6) | One cut recombination (0.7) | 0.7 | Binary mutation (0.025) | 0.025 | | 1136.64 | 50.79163710690964 |
| 100 | 100 | Tournament(8) | One cut recombination (0.7) | 0.7 | Binary mutation (0.025) | 0.025 | | 1093.8 | 124.57527844640765 |
| 100 | 100 | Tournament(10) | One cut recombination (0.7) | 0.7 | Binary mutation (0.025) | 0.025 | | 1091.28 | 123.71629480387782 |

- Based on these files, you can use charts to derive your conclusions
- Although the program allows you to define more than one value for more than one attribute, we advise you to define one base configuration common to all your experiments and define several values for just one attribute each time (the one whose effect you want to study); it is easier to manage the results this way

Enjoy ☺