Design a Genetic Algorithm to solve TSP

Object-oriented Programming report

Nguyen Phuc Truong An - 20200005 - an.npt200005@sis.hust.edu.vn, Hoang Van An - 20204864 - an.hv204864@sis.hust.edu.vn, Nguyen The An - 20204865 - an.nt204865@sis.hust.edu.vn, Nguyen Truong Truong An - 20204866 - an.ntt204866@sis.hust.edu.vn

SOICT, Hanoi University of Science and Technology.

July, 2022

Abstract

A **genetic algorithm(GA)** is a search heuristic that is influenced by Charles Darwin's theory of natural evolution. The process of natural selection is reflected in **GA**, where the most fit individuals are chosen for reproduction in order to give rise to the following generation's children. In this mini-project, we are asked to solve the *Travelling Salesman Problem(TSP)* by implementing Genetic Algorithm alongside with applying OOP techniques and Java Programming.

1. Introduction

Travelling Salesman Problem (TSP) is one of the most famous and interesting problems in the field Computer Science, the problem has been shown to be NP-hard. **Genetic Algorithm** is a *metaheuristic* inspired by the process of natural selection that belongs to the larger class of evolutionary algorithms **(EA)**. Genetic algorithms simulate the process of natural selection which means those species who can adapt to changes in their environment are able to survive and reproduce and go to next generation. In this project, we will implement a **Genetic Algorithm (GA)** to determine the (near-)optimal route.

1.1. Problem Description

The problem statement is simple, given a graph **G** = (**V, E**), where **V** is the set of nodes, **E** is the set of edges, there is additional information about distances between nodes. The goal of this problem is to output the shortest route starting from V_{start} and ending at V_{start}, such that each node is visited once.

1.2. Methodology

In this report, we will go through some operators of the **GA** algorithm such as *mutation*, *crossover* and *selection*. We also give some context and explanation of the OOP techniques applied in the implement procedure. Finally, a GUI will be carried out to illustrate how the algorithm find the optimal route.

2. Use case diagram

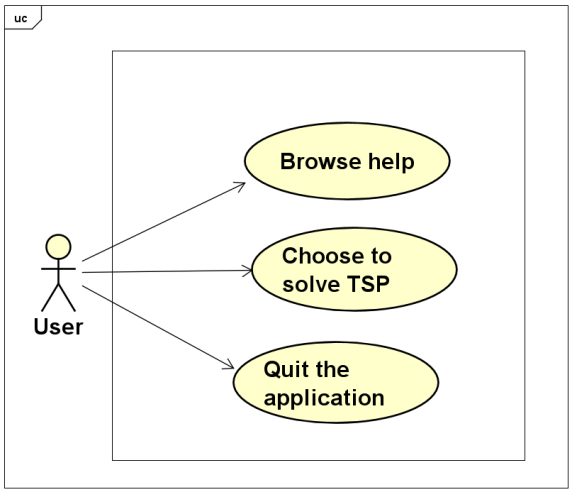

Figure 2: Use case diagram

Our application has three main use cases:

- **Browse help**: Use when the user wants to see the instructions of the application or information about Genetic Algorithm. The user need to click the button **Help**.
- **Choose to solve TSP**: Use when the user wants the system to run the algorithm. This use case requires the user to input parameters (or use default parameters), then the user can click button **Start**. When the application is running, the user can click button **Stop** to cancel the current run. When the TSP is solved, user can click button

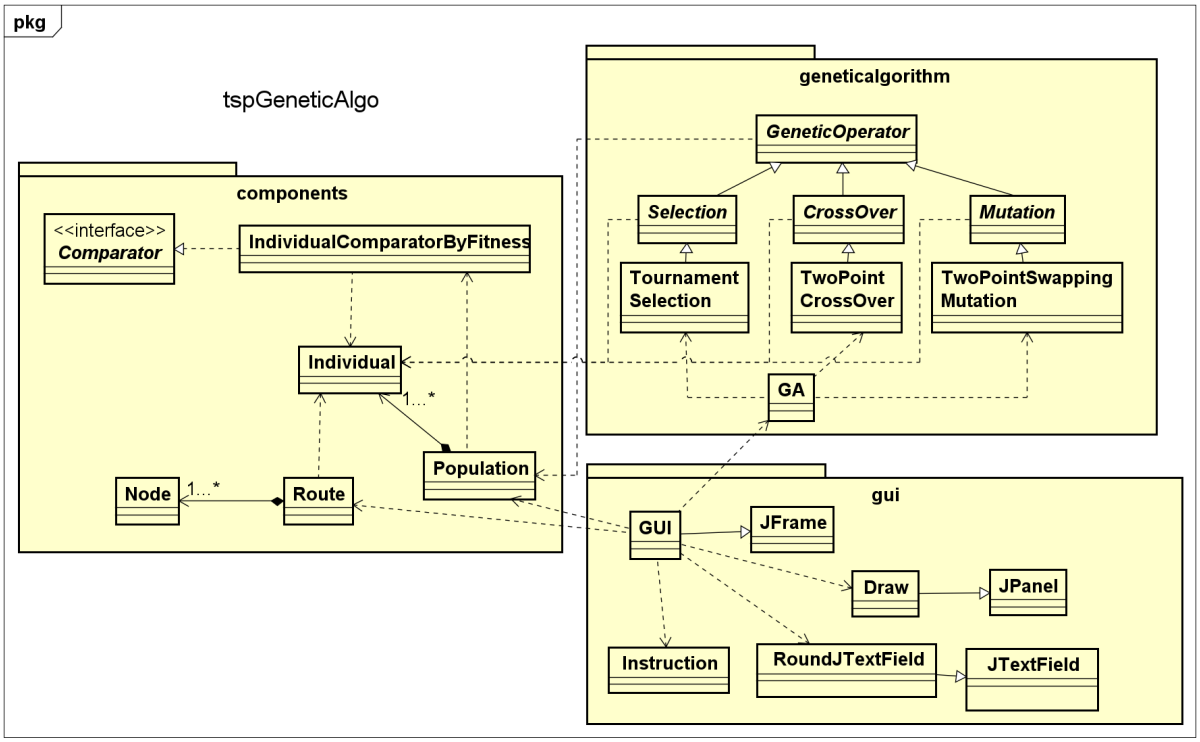

Figure 1: General class diagram

Print route to print out the best route found by the algorithm.

- **Quit the application**: Use when the user wants to quit the application. The user need to click the button **Quit**. After that, a confirmation pane will pop up and asks whether the user really wants to quit the application. If they click button "Yes", the application will close.

3. General class diagram

There are three packages: components, geneticalgorithm and tsp:

*3.1. The **components** package*

The components package contains the entities that is needed in the algorithm: node, individual, route, population and a comparator class.

- A node is a city in TSP problem.
- An individual is an array of integer represents an order of nodes, considered as a solution to the problem.
- A route is a combination of nodes in the order of an individual.
- A population is a combination of individuals.

*3.2. The **geneticalgorithm** package*

The geneticalgorithm package, as its name suggests, contains our algorithm. The genetic algorithm has three genetic operators: selection, cross over and mutation.

- **Selection** is the method to choose the "best" individual to participate in crossover process. There are many selection strategy, and in this project, we use the strategy called **Tournament Selection**. Each individual in the population will be selected at least once in this process and become the first parent (called father). Then, we choose a random of n individuals from population (tournament). The second parent (called mother) is the individual in the tournament that has highest fitness (represents the best solution). The father and mother will then go to crossover step to find the children. Another selection method that is used is elitism. During a generation, some individuals that have highest fitness will be passed to the next generation without having to participate in crossover and mutation.
- **Cross over** is the process of taking two parent individuals (chromosomes) and producing a child individual from them. The child solution will replace the father in the

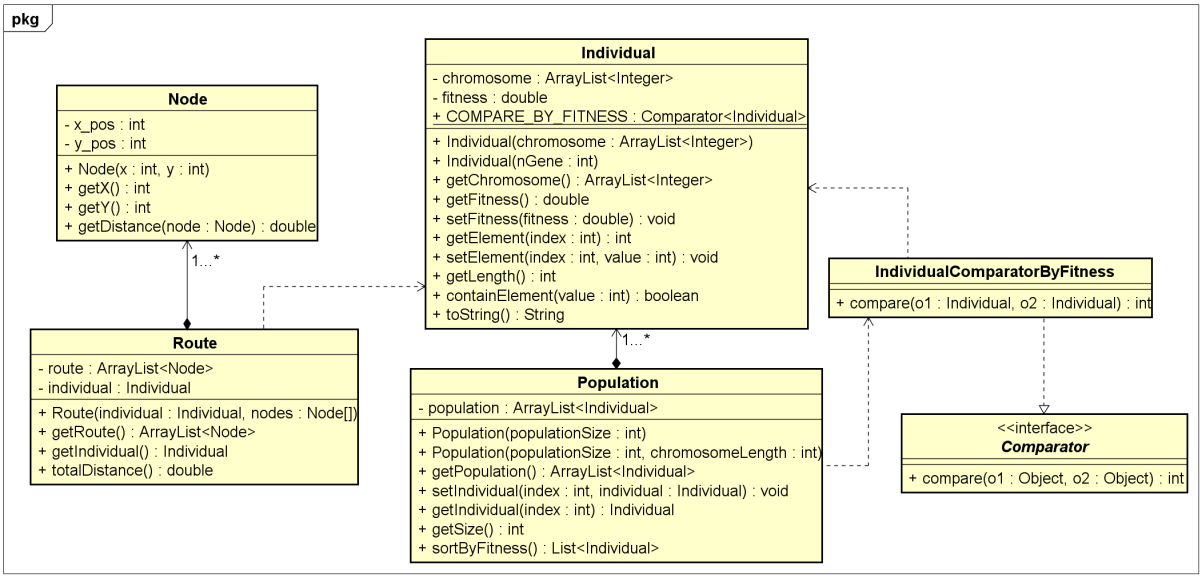

Figure 3: class diagram for *components* package

population. Our strategy for crossover is **Two Point Cross Over**: randomly select a start and end gen in the father chromosome and put them in child chromosome. Then for every gens that not appear in child chromosome, we will select them in the order of mother chromosome.

- **Mutation** is the process that randomly change some individuals in some way. It encourages genetic diversity among individuals and help prevent the genetic algorithm converging to a local minimum by stopping the individuals becoming too close to one another. Our strategy for mutation is **Two Point Swapping Mutation** randomly select two gens in a individual and then swap them to make a new individual.

*3.3. The **gui** package*

This is the package for Graphical User Interface.

- GUI is the main class to run the whole project.
- Draw is the class that draws nodes and lines, with their changes over generations
- Instruction is the class that contains the text for instructions.
- RoundJTextField is the class for adjusting the border shape of JTextField. The built-in JTextField is in shape in rectangle and RondJTextField is in shape of round corner rectangle.

4. Detailed class diagram for each package

4.1. components package

4.1.1. node

Node represents the city in TSP.

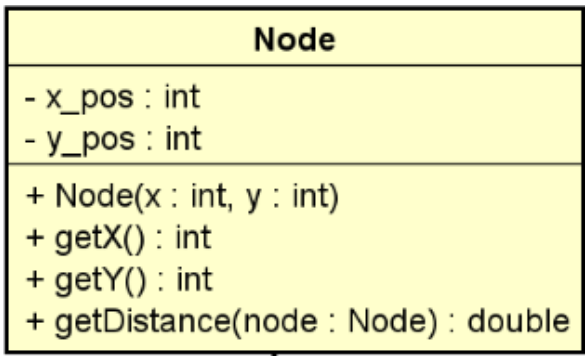

Figure 4: Node class diagram

- Attributes:
  - `x_pos and y_pos` : locate the city in 2D space
- Methods:
  - Node(x: int, y: int): constructor, take in the value of x and y.
  - getX(), getY(): getter for x and y.
  - getDistance(node: Node): calculate and return the Eucidean distance between the current Node and the parameter Node .

4.1.2. individual

Individual is an entity that represents an acceptable solution to the problem. The Route class and IndivudialComparatorByFitness class is depends on this class.

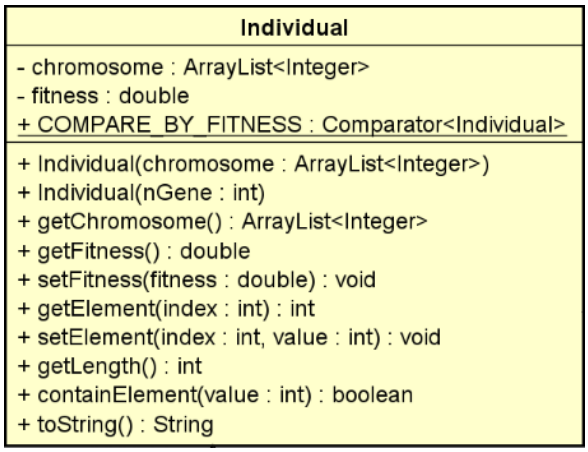

Figure 5: Individual class diagram

- Attributes:
  - chromosome: represent the individual (solution) by an array of genes.
  - fitness: Fitness Score, shows the ability of an individual to "compete". Individuals with better fitness score could produce better offsprings.
  - *COMPARE_BY_FITNESS*: IndividualComparatorByFitness() object.
- Methods:
  - Individual(ArrayList <Integer> chromosome): constructor, take in 1 parameter chromosome
  - Individual(int nGene): constructor, has 1 integer parameter nGene and add number from 0 to nGene to the Individual's chromosome
  - getChromosome(): getter for chromosome
  - getFitness() and setFitness(double fitness): getter and setter for fitness.
  - getElement(int index) and setElement(int index, int value): getter and setter for a specific gene in the chromosome.
  - getLength(): return the length of the chromosome.
  - containElement(int value): a boolean method return true if the parameter value is in the chromosome and false otherwise.
  - toString(): return the current chromosome.

4.1.3. comparator

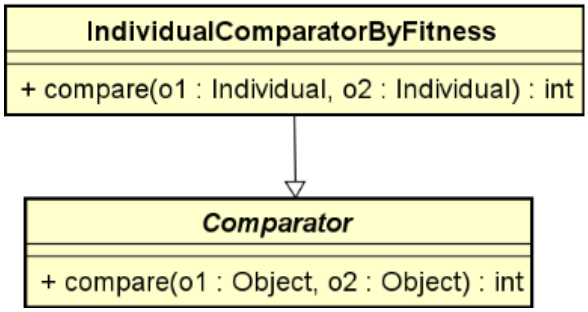

Figure 6: IndividualComparatorByFitness and Comparator class

The IndividualComparatorByFitness class implements the java interface Comparator which is used for sorting Java objects and it override the method compare to compare the fitness score of 2 Individuals.

4.1.4. route

Route is a composition of Node, and depends on the Individual. It is a list of cities in the order of an acceptable solution founded by the algorithm.

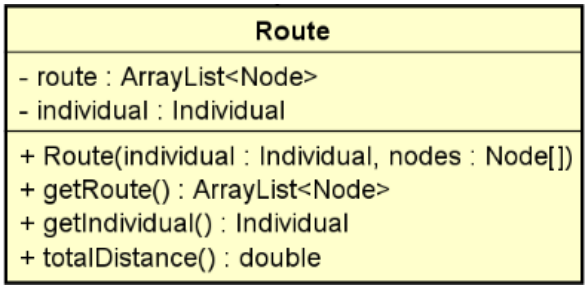

Figure 7: Route class

- Attributes:
  - route: a list of Nodes.
  - **fitness**: Fitness Score, shows the ability of an individual to "compete". Individuals with better fitness score could produce better offsprings.
- Methods:
  - Route (Individual individual, Node[] nodes): constructor, take in 2 parameters individual and list of nodes.
  - getRoute() and getIndividual(): getters for route and individual.

– getChromosome(): getter for chromosome

4.1.5. population

Population is a composition of Individuals. It represents a set of solutions to the problem. From a population, we will perform genetic operators with the goal of making a new generation with better individuals (solutions).

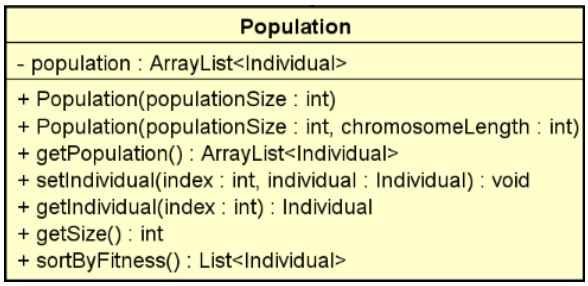

Figure 8: Population class

- Attributes:
  - population: a list of Individuals
- Methods:
  - Population(int populationSize): constructor, create a Population contains n individuals (n = populationSize)
  - getRoute() and getIndividual(): getters for route and individual.
  - getPopulation(): getter for the population
  - setIndividual(int index, Individual individual) and getIndividual(int index): getter and setter for the population
  - getSize(): get the size of the population
  - sortByFitness(): use the java interface Collection and to sort the population by Individual's fitness

4.2. geneticalgorithm package

The IndividualComparatorByFitness class implements the java interface Comparator which is used for sorting Java objects and it override the method compare to compare the fitness score of 2 Individuals.

4.2.1. ***GeneticOperator*** *class*

This is the super class for all operators in GA (Selection, CrossOver, Mutation). It is defined as an abstract class.

- Attributes: *(protected)* population: the population that we are working with
- Methods: All the methods are public
  - GeneticOperator(): constructor, take in a population
  - (abstract) execute(): the execution of the operator on the population
  - getPopulation(): to get the population after execution of the operator

4.2.2. ***Selection*** *class*

This is a child class of GeneticOperator, also a super class of all selection strategies. It is defined as an abstract class.

- Attributes: All attributes are protected
  - elitism: the number of best individuals (has highest fitness) that are passed to next generation without participating in crossover or mutation
  - father: the first parent in crossover process
  - mother: the second parent in crossover process
- Methods: All the methods are public
  - Selection(): constructor, take in a population and an integer represents number of elitism
  - *(abstract)* execute(): the execution of Selection
  - getFather()/getMother(): getters for the parents

4.2.3. ***CrossOver*** *class*

This is a child class of GeneticOperator, also a super class of all cross over strategies. It is defined as an abstract class.

- Attributes: All attributes are protected
  - crossOverRate: the chance that cross over will happen for two parents.
  - father: the first parent in crossover process
  - mother: the second parent in crossover process

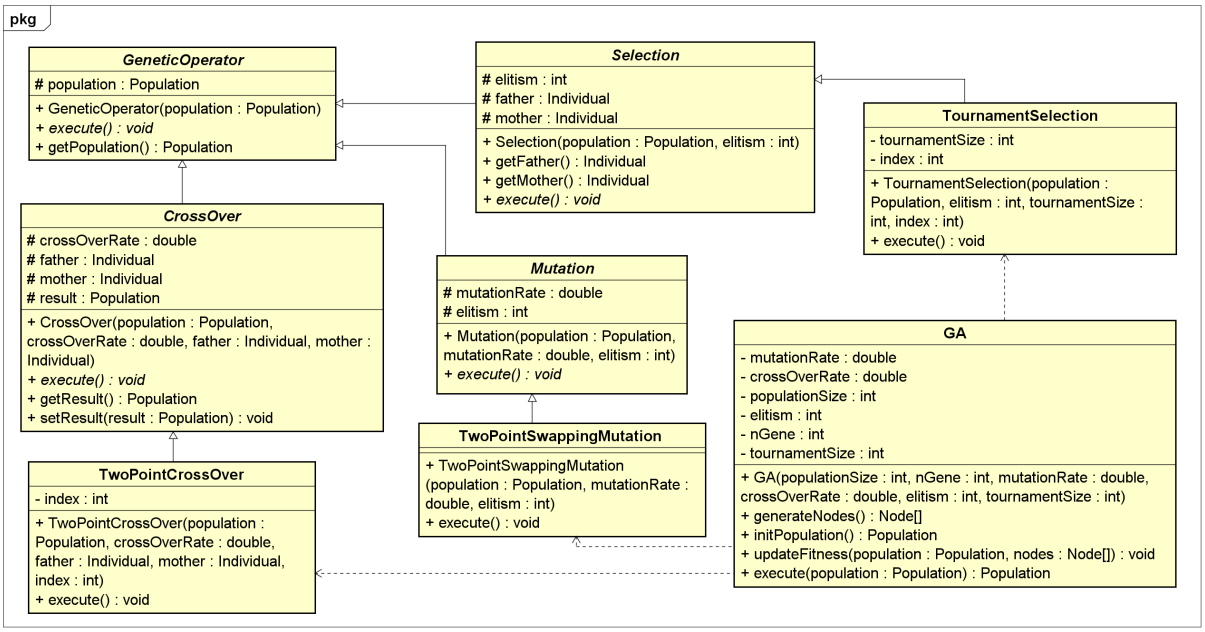

Figure 9: class diagram for geneticalgorithm package

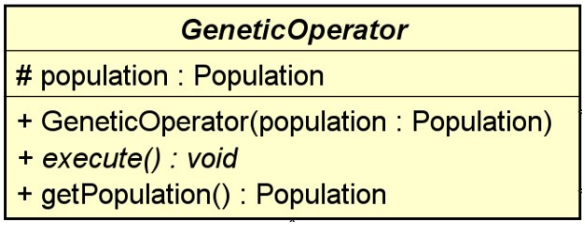

Figure 10: GeneticOperator class diagram

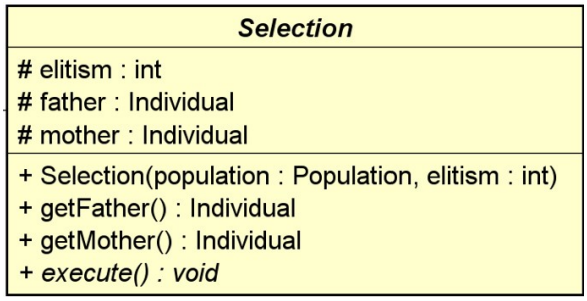

Figure 11: Selection class diagram

  - result: temporary of new population after a cross over

- Methods: All the methods are public
  - CrossOver(): constructor, take in a population, cross over rate and two parents
  - *(abstract)* execute(): the execution of Cross over
  - getResult() and setResult(): getter and setter for result.

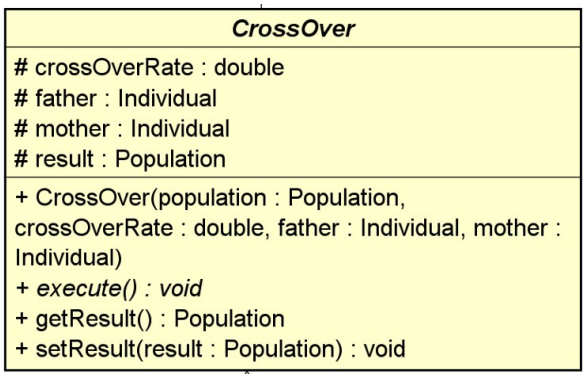

Figure 12: CrossOver class diagram

4.2.4. ***Mutation*** *class*

This is a child class of GeneticOperator, also a super class of all mutation strategies. It is defined as an abstract class.

- Attributes: All attributes are protected
  - mutationRate: the chance that mutation will happen for an individual
  - elitism: the number of best individuals (has highest fitness) that are passed to next generation without participating in

- Methods: All the methods are public
  - Mutation(): constructor, take in a population, mutation rate and elitism.

– *(abstract)* execute(): the execution of mutation

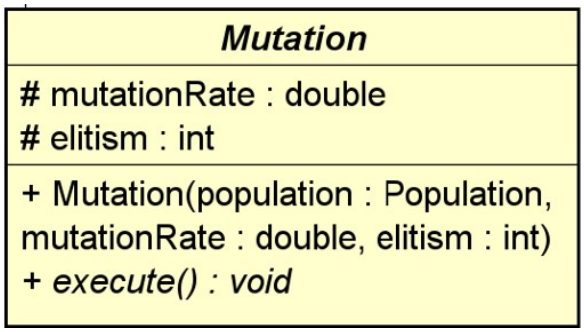

Figure 13: Mutation class diagram

4.2.5. ***TournamentSelection*** *class*
This is a child class of Selection, represents a selection strategy.

- Attributes: All attributes are *private*
  - tournamentSize: the size of tournament.
  - index: current index of the father in population
- Methods: All the methods are public
  - TournamentSelection(): constructor, take in a population, elitism, tournament size and index
  - execute(): overriding method, execution of Tournament selection strategy

Figure 14: TournamentSelection class diagram

4.2.6. ***TwoPointCrossOver*** *class*
This is a child class of CrossOver, represents a cross over strategy.

- Attributes: *(private)* index: current index of the father in population.
- Methods: All the methods are public
  - TwoPointCrossOver(): constructor, take in a population, cross over rate, two parents and index
  - execute(): overriding method, execution of Two point cross over strategy

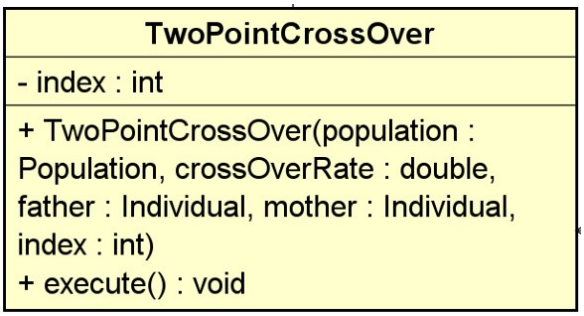

Figure 15: TwoPointCrossOver class diagram

4.2.7. ***TwoPointSwappingMutation*** *class*
This is a child class of Mutation, represents a mutation strategy.

- Methods: All the methods are public
  - TwoPointSwappingMutation(): constructor, take in a population, cross over rate, two parents and index
  - execute(): overriding method, execution of Two point swapping mutation strategy.

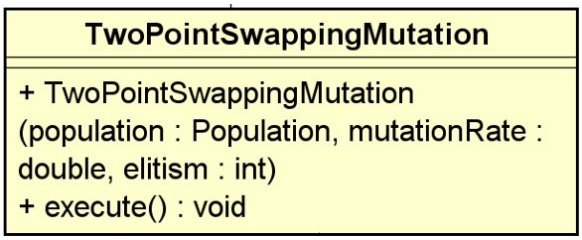

Figure 16: TwoPointSwappingMutation class diagram

4.2.8. ***GA*** *class*
The genetic algorithm class, is the combination of three genetic operators.

- Attributes: All attributes are private
  - mutationRate, crossOverRate, elitism, tournamentSize: attributes from Tournament Selection, Two Point CrossOver, Two Point Swapping Mutation process
  - populationSize: the size of population

  - nGene: number of genes in one individual (number of cities in TSP)

- Methods: All the methods are public
  - GA: constructor
  - generateNodes: randomly generate nodes (cities) and return an array of nodes
  - initPopulation: initialize a starting population. All individuals are the same, and will change after generations
  - updateFitness: used after each generation to update fitness for all individuals in the population
  - execute: take in a population and perform the algorithm (selection, cross over, mutation). After that, return the new population.

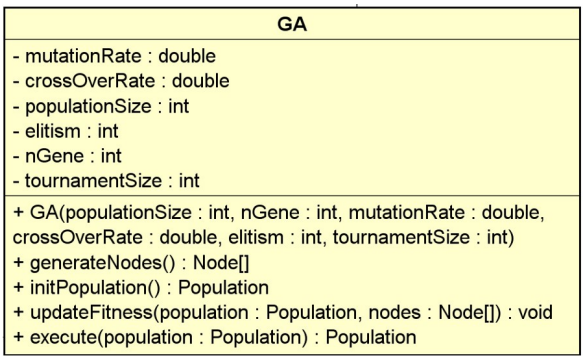

Figure 17: GA class diagram

4.3. gui package

4.3.1. ***GUI*** *class*

This class inherits from class javax.swing.JFrame.

- Attributes: All attributes are private
  - route, prevRoute: to store and update the best route found in the generation
  - populationSizeText, numGensText, numNodesText, numSurvivalText, rCrossOverText, rMutationText, sTournamentText: JTextField to adjust the parameters in the genetic algorithm.
  - lblGenerations, lblbestDistance: showing the current generation and the shortest path obtain during running time.
  - btnPrint: the button that print out the nodes' information and the best found route whenever clicked. This button only appear after activating the app.

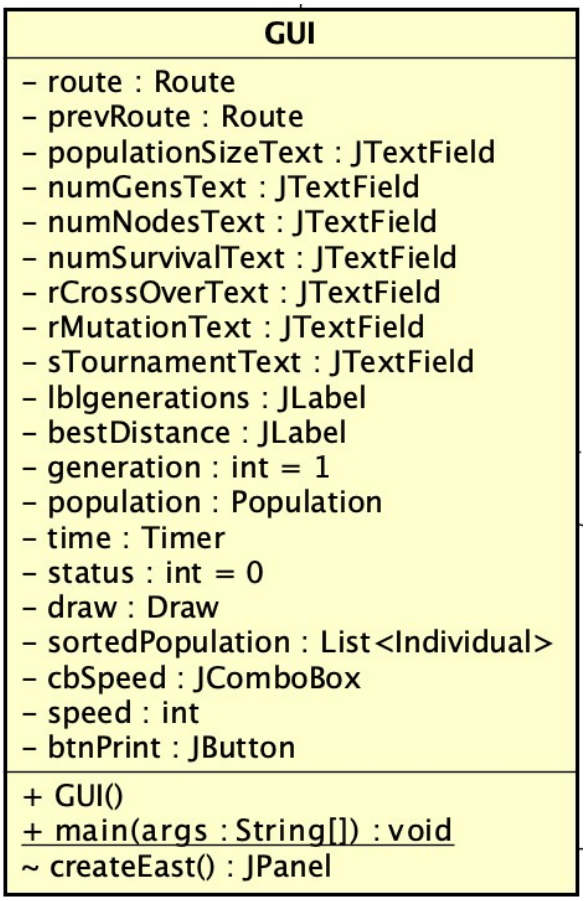

Figure 18: GUI class diagram

  - cbSpeed: Combo box that contain options for speed.
  - speed: adjusting the speed that the application would find a new solution and demonstrate in the center of the frame.
  - time, status: control the application whether to run or stop.
  - population, generation: population at a specific generation.
  - sortedPopulation: List of individuals that were sorted by fitness.
  - draw: Panel draws nodes and demonstrates the process of finding the best route throw all nodes.

- Methods:
  - GUI: constructor, creating the Frame for the application and add main panels to the frame
  - createEast: default access modifier, create Western panel (or control board) which includes all text-fields, buttons

and setting to run and adjust the application.

- main: run the application.

4.3.2. ***ButtonListener*** *class*

Inner class of GUI class, to set action or function for some buttons whenever users choose those.

Figure 19: ButtonListener class diagram

- Methods:
  - actionPerformed: overriden method from interface ActionListener, set the action corresponding to a specific ActionEvent.

4.3.3. ***Draw*** *class*

This class inherits from class javax.swing.JPanel, draw the panel with nodes and line of the route inside.

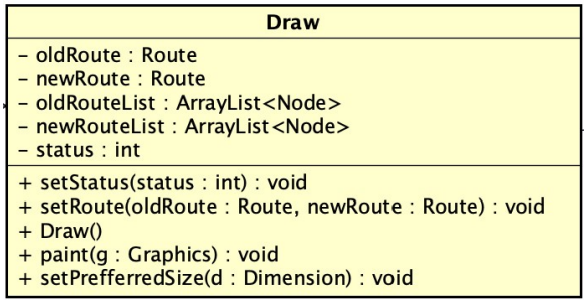

Figure 20: Draw class diagram

- Attributes: All attributes are private
  - oldRoute, oldRouteList: previous best route
  - newRoute, newRouteList: updated best route (lower total distance)
  - status: control when the paint method excute to draw.
- Methods: All the methods are public
  - Draw: constructor.
  - setRoute: set new value for oldRoute and newRoute
  - paint: method overriding, draw the cities and the line to connect all cities including old route and new route.

4.3.4. ***RoundJTextField*** *class*

This class inherits from class javax.swing.JTextField, creating a text field in shape of round corner rectangle.

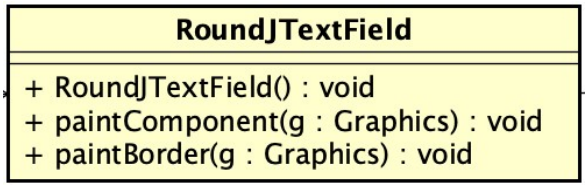

Figure 21: RoundJTextField class diagram

- Methods: except for the constructor, all methods have protected access-modifier.
  - RoundJTextField: constructor, use super constructor with 2 arguments text and columns
  - paintComponent: method overriding, fills the specified rounded corner rectangle with a specific color (white in this case).
  - paintBorder: method overriding, paints the component's border with a specific color (white in this case).

4.3.5. ***Instruction*** *class*

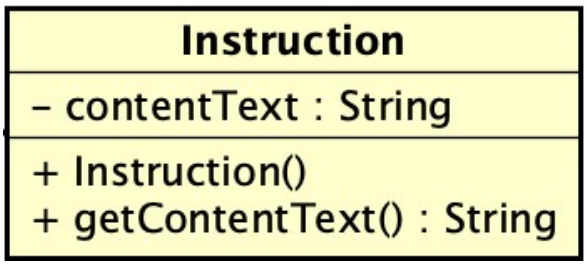

Figure 22: Instruction class diagram

- Attribute: private final
  - contentText: contain content for guiding users to use the application, a string in HTML (Hypertext Markup Language).
- Methods:
  - getContentText: public method return the contentText

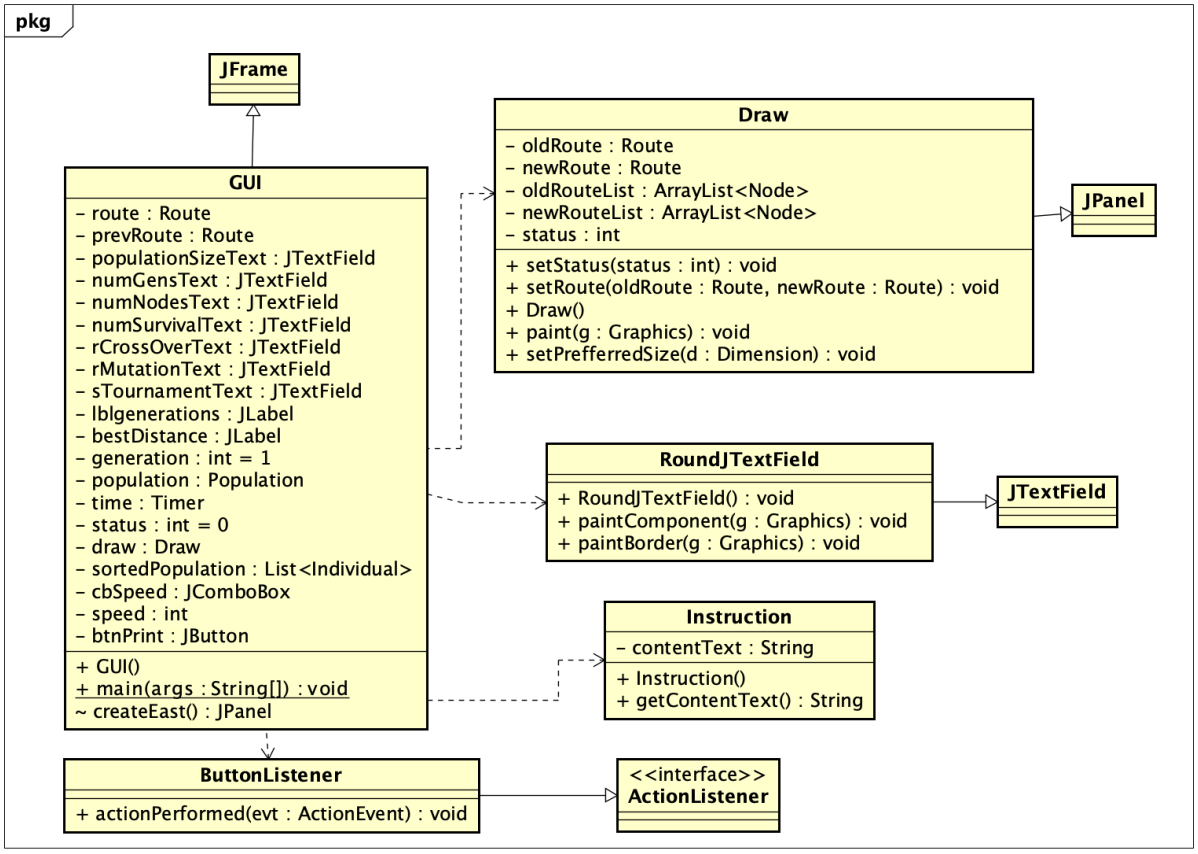

Figure 23: class diagram for *gui* package

5. Explanation of OOP techniques

In our project, we have used all Object-oriented Programming techniques: *Abstraction, Encapsulation, Inheritance, Polymorphism.*

1. **Encapsulation**:
   Encapsulaltion is used everywhere in the class building process. Attributes and methods are encapsulated in a class. All attributes are defined as *private* or *protected* so they can only be accessed from methods in the class (private attributes) or child classes (protected attributes). Other objects that want to access to the private data must perform via public functions provided in the class. This will help to improve security of our code. Classes with the same logic relations are grouped into a package (*components, geneticalgorithm, gui*). Package is considered as a directory, a place to organize classes in order to locate them easily.
2. **Abstraction**:
   Alongside with encapsulation, abstraction is also used everywhere in our project. All entities with same properties are represented in terms of a class. By using abstraction techniques, we can extract the most important aspects of an entity while suppressing or ignoring less important details.
3. **Inheritance**:
   In GA, there are three steps: selection, cross over and mutation. They are called genetic operators. So we first create an abstract class genetic operator with a protected attribute population and public abstract method execute. Then we create three child class of it: selection, crossover and mutation. These child class will inherit the attribute population and method execute. For each genetic operator, there are many strategies to do. Example: we can do tournament selection or Boltzmann selection. So we will create child classes for each genetic operators, represents the strategies. Each strategy will perform in different ways, so their classes will have their own private attributes, but they all have same inherited method execute.
   The inheritance technique here can increase reusability, When a class inherits or derives another class, it can access all the function-

ality of the class it inherits from. Inheritance also helps to limit code redundancy and supports code extensibility. In the future works, if we want to add more strategy for selection, cross over or mutation, we just need to create a new class that inherit the corresponding super class.

4. **Polymorphism**:
   First use of polymorphism in this project is in method *execute* defined in class *GeneticOperator* and overrided by its child classes. Each genetic operator and each strategy of a genetic operator will have different ways to *execute*. By using polymorphism, we dont need to create a unique method for each class to run the algorithm, but only one method *execute*.
   Second use of polymorphism in this project is in method *compare* from class *IndividualComparatorByFitness*, which overrides from interface *Comparator*. In this project, we choose fitness as a scale to compare individuals, but there can be another scale like reward in **Reward-based selection**. In future works, if we want to implement this selection strategy, we can have another class *IndividualComparatorByReward* and implements it to *Population* class without having to change the compare methods.
   Last use of polymorphism in this project is in package *gui* with 2 classes inherits and overrides methods of javax.swing package, which is Draw and RoundJTextField. Class Draw is the child class of JPanel and it override method paint to draw the process of finding the best route through cities. Class RoundJTextField is the child class of JTextField and it override methods paintComponent and paintBorder to create a text field in shape of round border rectangle.

Overall, by using OOP techniques, we can improve reusability, security, limit redundancy in code, reduce cost for maintenance and it will be easier to extend the program.

6. Member assigments

1. **Hoang Van An**:
   - *components* package (20%):
     - Design package
     - toString method
   - *Construct class diagram* (30%)
   - *geneticalgorithm* package (30%)
     - UpdateFitness method
     - Try different strategies for algorithms
   - *gui* package (60%)
     - Create Help button with guidelines for the application.
     - Create Print route button.
     - Improve *GUI* class (ButtonListener, color, size)
     - Improve *Draw* class (color, size, experiment the proper scale to fit the frame)
     - Create class *RoundJTextField, Instructions*.
   - *presentation slide* (90%)
   - *report* (20%)
2. **Nguyen Truong Truong An**:
   - *Construct use case diagram* (40%)
   - *components* package (10%)
   - *presentation slide* (10%)
   - *report* (20%)
3. **Nguyen The An**:
   - *Construct class diagram* (60%)
   - *Construct use case diagram* (60%)
   - *components* package (70%):
     - Design package
     - Create classes *Node, Individual, Route, Population*
   - *geneticalgorithm* package (70%)
     - Design package
     - *GeneticOperator* class
     - *Selection, CrossOVer, Mutation* class
     - *TournamentSelection, TwoPointCrossOver, TwoPointSwappingMutation* class
   - *gui* package (40%)
     - Create *Draw* class
     - Create initial *GUI* class
     - Implement Timer in *GUI*
   - *report* (50%)
4. **Nguyen Phuc Truong An**:
   - *Construct class diagram* (10%)
   - *components* package (10%): comparator method
   - *geneticalgorithm* package (10%): Handle Exception
   - *report* (10%) : Conclusion

7. Conclusion

Genetic algorithm (GA) is an effective heuristic to solve Travelling Salesman Problem (TSP), which could generate high-quality solutions within a short period of time. In this project, we have implemented Genetic Algorithm to solve TSP with some application of Object oriented programming (OOP) techniques. For future work, the program could be developed with more strategies for selection, cross-over, mutation, and other algorithms besides GA to get a complete application for solving TSP. With the current system that were built based on OOP techniques, it will not take time to extend and implement these ideas, while ensuring its reusability and security.

References

[1] Fang Y., Li J. (2010) *A Review of Tournament Selection in Genetic Programming*, Advances in Computation and Intelligence - 5th International Symposium, China.

[2] Umbarkar1 A. J., Sheth P.D. (2015) *Crossover operators in genetic algorithms: A review*, ICTACT Journal on Soft Computing.