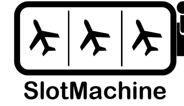
Ref. Ares(2022)311643 - 15/01/2022

D4.2 Specification of Evolutionary Algorithm

Deliverable ID:	**D4.2**
Dissemination Level:	**PU**
Project Acronym:	**SlotMachine**
Grant:	**890456**
Call:	**H2020-SESAR-2019-2**
Topic:	**SESAR-ER4-27-2019 Future ATM Architecture**
Consortium Coordinator:	**Frequentis**
Edition Date:	**15 January 2022**
Edition:	**01.00.01**
Template Edition:	**02.00.03**

Founding Members



SlotMachine

A PRIVACY-PRESERVING MARKETPLACE FOR SLOT MANAGEMENT

This Deliverable is part of a project that has received funding from the SESAR Joint Undertaking under grant agreement No 890456 under European Union’s Horizon 2020 research and innovation programme.

Abstract

The SlotMachine system employs an evolutionary algorithm in conjunction with multiparty computation to optimize flight lists in a privacy-preserving way. This document describes the Heuristic Optimizer component of the SlotMachine system, which realizes the evolutionary algorithm for finding solutions to the SlotMachine flight prioritization problem. The Heuristic Optimizer provides an extendable framework allowing to plug in different implementations of evolutionary algorithms for optimization of flight lists; this document describes a configurable genetic algorithm implementation. Experiments conducted with generated synthetic data for different scenarios serve to evaluate the implementation of the Heuristic Optimizer.

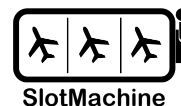


Table of Contents

Abstract 2
1 Introduction 6
1.1 Purpose of the document 6
1.2 Scope 6
1.3 Intended readership 6
1.4 Background 6
1.5 Structure of the document and relation to other deliverables 6
2 Heuristic Optimizer: Overview 8
2.1 SlotMachine Flight Prioritization Problem 8
2.2 Evolutionary Optimization Algorithm 9
2.3 Heuristic Optimizer and Privacy Engine 11
3 Framework 12
3.1 Structure 12
3.2 Creating and Initializing an Optimization 14
3.3 Running an Optimization 15
4 Genetic Algorithm 16
4.1 Parameters 16
4.2 Implementation 19
4.2.1 Structure 19
4.2.2 Creating and Initializing an Optimization 21
4.2.3 Running an Optimization 22
5 Fitness Function 24
6 Evaluation 26
6.1 Relation to Requirements 26
6.2 Experiments 30
6.2.1 Setup 30
6.2.2 Results 32
7 Conclusions 45
8 References 46
Appendix A Third-Party Libraries 48

List of Tables

Table 1. Parameters for the genetic algorithm 17

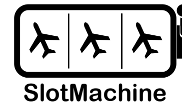


Table 2. Requirements from D2.1 and their implications on the design of the Heuristic Optimizer ... 26

Table 3. Cases used in experiments ... 31

Table 4. Genetic algorithm configurations used in experiments ... 31

Table 5. List of third-party libraries ... 48

List of Figures

Figure 1. The SlotMachine flight prioritization problem as an unbalanced assignment problem ... 8

Figure 2. Principle of the iterative optimization process of a genetic algorithm ... 10

Figure 3. Principle of the iterative optimization algorithm with ranked population and fitness estimation ... 11

Figure 4. UML class diagram for optimization runs in the Heuristic Optimizer component ... 13

Figure 5. UML sequence diagram of the creation and initialization of an optimization run ... 14

Figure 6. UML sequence diagram for starting an optimization run and retrieving the results ... 15

Figure 7. UML class diagram for initiating and running optimizations using the Jenetics framework . 20

Figure 8. UML sequence diagram of the creation of an optimization run using the Jenetics framework ... 21

Figure 9. UML sequence diagram of an optimization run using the Jenetics framework ... 22

Figure 10. UML sequence diagram of batch evaluation of a population of flight lists ... 23

Figure 11. Distribution of fitness values for different fitness estimators ... 25

Figure 12. Average/best fitness over five runs of various genetic algorithm configurations using no fitness estimator, i.e., evaluation through computation of absolute fitness. ... 33

Figure 13. Average/best fitness over five runs of various genetic algorithm configurations using a linear fitness estimator when maximum fitness of a population and a ranking of individuals within the population is known. ... 34

Figure 14. Average/best fitness over five runs of various genetic algorithm configurations using a logarithmic fitness estimator when maximum fitness of a population and a ranking of individuals within the population is known. ... 35

Figure 15. Average/best fitness over five runs of various genetic algorithm configurations using a sigmoid fitness estimator when maximum fitness of a population and a ranking of individuals within the population is known. ... 36

Figure 16. Evaluated fitness distribution in different generations of J1 applied to Case 9 using no estimator ... 37

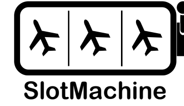


Figure 17. Evaluated fitness distribution in different generations of J1 applied to Case 20 using no estimator 38

Figure 18. Estimated and evaluated fitness distributions in different generations of J1 applied to Case 9 using linear estimator 39

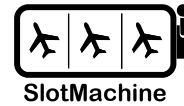
Figure 19. Estimated and evaluated fitness distributions in different generations of J1 applied to Case 20 using linear estimator 40

Figure 20. Estimated and evaluated fitness distributions in different generations of J1 applied to Case 9 using logarithmic estimator 41

Figure 21. Estimated and evaluated fitness distributions in different generations of J1 applied to Case 20 using logarithmic estimator 42

Figure 22. Estimated and evaluated fitness distributions in different generations of J1 applied to Case 9 using sigmoid estimator 43

Figure 23. Estimated and evaluated fitness distributions in different generations of J1 applied to Case 20 using sigmoid estimator 44



1 Introduction

1.1 Purpose of the document

The purpose of this document is to specify the Heuristic Optimizer component of the SlotMachine system, which is responsible for optimizing flight lists given the airspace users' preferences. To this end, the Heuristic Optimizer employs an evolutionary algorithm. This document describes, on the one hand, the general framework of the Heuristic Optimizer component, which allows to plug in different implementations of evolutionary algorithms. On the other hand, this document also describes a concrete implementation of a genetic algorithm that can be used to optimize flight lists.

1.2 Scope

This document describes an extensible implementation of a framework for running optimizations of flight lists in the SlotMachine project. The implementation will serve as the basis for further development and can be integrated into the SlotMachine system.

1.3 Intended readership

The SlotMachine project team will use this document as reference for further development effort regarding the integration of the SlotMachine system's individual components. The experimental evaluation presented in this document will serve as the basis for decisions regarding further development effort in the area of performance tuning. Beyond the project, researchers and practitioners find a novel approach in this document for solving an optimization problem in a privacy-preserving way using evolutionary algorithms.

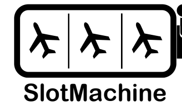
1.4 Background

The Heuristic Optimizer component described in this document is an extension of the non-privacy-preserving variant described in D4.1 – Report on State of the Art of Relevant Concepts [1]. The presented implementation of the Heuristic Optimizer builds on related work on evolutionary optimization algorithms [2], [3] in general and employs an open-source framework for the implementation of genetic algorithms in Java [4].

1.5 Structure of the document and relation to other deliverables

The general structure of this document is as follows.

- Chapter 1 (this section) provides a general idea of the entire document. It includes the purpose, readership, inputs from other projects, component purpose and high-level overview and acronyms used in the document.
- Chapter 2 characterizes the optimization problem solved by the Heuristic Optimizer component and explains the intuition of the privacy-preserving optimization process.
- Chapter 3 describes the Heuristic Optimizer's general framework, which was designed having the goal of extensibility in mind.



- Chapter 4 describes the genetic algorithm implementation which was plugged into the Heuristic Optimizer's general framework; this implementation serves as the basis for experimentation and will form the basis for further development.
- Chapter 5 discusses the Heuristic Optimizer's approach to evaluate the found solutions using the information provided by the Privacy Engine.
- Chapter 6 examines the Heuristic Optimizer's relation to the requirements and presents the results of experiments using generated synthetic data for different scenarios.
- Chapter 7 concludes the document.
- Chapter 8 lists the references.
- Appendix A lists the third-party libraries used in the implementation.

This document relates to the other deliverables of the SlotMachine project as follows.

- D2.1 – Requirements Specification [5]: The requirements are the foundation for the design and implementation of the Heuristic Optimizer described in this document; this document refers to the requirements in D2.1.
- D2.2 – System Design Document [6]: The system design document describes the interfaces of the Heuristic Optimizer and details the interactions of the Heuristic Optimizer with other components.
- D2.3 – Business Concepts [7]: More details on operational background, deployment options, and market mechanisms can be found in D3.2.
- D3.2 – Specification of the Privacy Engine component [8]: The Privacy Engine and related components (MPC Nodes and Credit Handling) are described in further detail in D3.2. The Heuristic Optimizer invokes the Privacy Engine for evaluating the found solutions.
- D4.1 – Report on State of the Art of Relevant Concepts [9]: An overview of genetic algorithms and local search can be found in D4.1. A non-privacy-preserving implementation of the Heuristic Optimizer was used for experiments to determine the suitability of different types of evolutionary algorithms for the SlotMachine project. The cases from the experiments in D4.1 were the basis for the experiments described in this document using the extended implementation of the Heuristic Optimizer.
- D5.1 – SlotMachine Platform Demonstrator [10]: The platform demonstrator will integrate the individual components for evaluation.

2 Heuristic Optimizer: Overview

In the following, we first characterize the SlotMachine flight prioritization problem that the Heuristic Optimizer solves. We then explain the principle of the Heuristic Optimizer's evolutionary algorithm driving the optimization process before explaining the relation between the Heuristic Optimizer and the Privacy Engine component of the SlotMachine system.

2.1 SlotMachine Flight Prioritization Problem

From an economic perspective (see also D2.3 – Business Concepts [7]) the SlotMachine system aims to maximize the overall utility of airspace users. In order to facilitate optimization of flight lists, each airspace user may provide a valuation for each slot and flight.

Generally speaking, the SlotMachine flight prioritization problem is an *assignment problem* that is possibly unbalanced [11]. The problem to be solved is to find a mapping between two sets of objects – flights and slots (see Figure 1) – minimizing overall costs or maximizing the overall benefit given a cost or a benefit matrix, respectively. If the goal is to minimize costs or maximize benefits of the assignment, the problem can be considered a single-objective assignment problem. The problem is unbalanced if the number of elements differs between the sets to be mapped; in the SlotMachine flight prioritization problem there may be possibly more slots than flights.

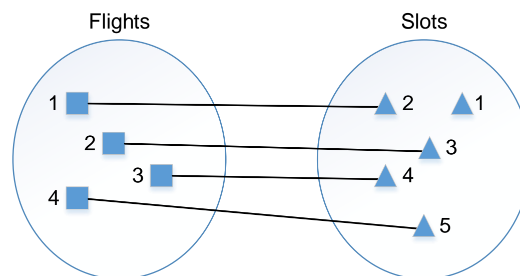


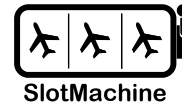
Figure 1. The SlotMachine flight prioritization problem as an unbalanced assignment problem

Formally, given a set *F* of flights and a set *S* of slots, the objective of the optimization is to maximize the overall utility *U* given a utility function *u* and an assignment function *m* of flights to slots as follows.

$$U = \sum_{f \in F} \sum_{s \in S} u_{f\,s} \cdot m_{f\,s}$$

The assignment function *m* defines for each combination of flight and slot whether the flight is assigned to that particular slot.

$$\forall\, f \in F : \forall\, s \in S : m_{f\,s} = \begin{cases} 1 \text{ if flight } f \text{ is assigned to slot } s \\ 0 \text{ otherwise} \end{cases}$$



The utility of a slot for a flight is a numeric value. The utility function *u* is often expressed in form of a utility matrix.

$$\begin{bmatrix} u_{1\,1} & \cdots & u_{1\,s} \\ \vdots & \ddots & \vdots \\ u_{f\,1} & \cdots & u_{f\,s} \end{bmatrix}$$

Each row in the utility matrix specifies the utilities of the different slots for a particular flight, with each column representing a different slot. Each entry $u_{f\,s}$ in the utility matrix thus defines the utility of a slot *s* for a flight *f*. In this representation, the set of flights *F* and the set of slots *S* are assumed to be ordered. For example, the entry $u_{1\,5}$ of the utility matrix *u* then defines the utility of the 5th slot in *S* for the 1st flight in *F*. We also refer to the utility matrix as *weight map* or *weight table*.

We can also define the following constraints for the problem.

- Each flight must be assigned to exactly one slot, i.e., $\forall\, f \in F : \exists!\, s \in S : m_{f\,s} = 1$. Alternatively, the sum of the values of $m_{f\,s}$ for every flight f must be 1, i.e., $\forall\, f \in F : \sum_{s \in S} m_{f\,s} = 1$.
- A flight must not be assigned to a slot with a time that is before the flight's originally scheduled time, i.e.,

  $$\forall\, f \in F, s \in S : \left(m_{f\,s} = 1\right) \Rightarrow scheduledTime(f) \leq time(s)$$

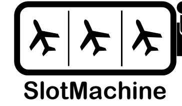
  where the function $scheduledTime$ denotes the originally scheduled time of a flight, the function $time$ denotes the time of a slot, and $\leq$ is the total order of times; we say $t \leq t'$ if and only if time t is before or equal to time t'.

The Heuristic Optimizer component described in this document solves the SlotMachine flight prioritization problem as a single-objective, possibly unbalanced assignment problem with utility maximization. Future work may, however, extend the Heuristic Optimizer to work with multi-objective optimization, particularly in relation to more elaborate market mechanisms.

2.2 Evolutionary Optimization Algorithm

A number of deterministic optimization algorithms have been proposed to solve the single-objective assignment problem. For example, the *Hungarian method* [12] [13] – also known as Hungarian algorithm, Kuhn–Munkres algorithm, or Munkres assignment algorithm – is arguably the most well-known optimization algorithm for the single-objective assignment problem. The Hungarian algorithm is of polynomial time [14] and a variant of the algorithm with a complexity of $O(n^3)$ can be devised [15], which would allow for efficiently solving the SlotMachine flight prioritization problem in a non-privacy-preserving setting.

In a privacy-preserving setting, a deterministic optimization algorithm for the single-objective assignment problem would have to be implemented entirely using multiparty computation. Preliminary results suggest that this approach is impractical for solving the SlotMachine flight prioritization problem (see D3.2 – Specification of the Privacy Engine Component [8]). Furthermore, when moving away from single-objective, purely utility-based optimization to a more complex multi-objective optimization, which may be necessary for realizing more complex market mechanisms (see



D2.3 – Business Concepts [7]), the complexity of a deterministic algorithm would increase even in a non-privacy-preserving setting, let alone the privacy-preserving setting. Thus, the choice for an evolutionary algorithm also provides additional flexibility for future development.

For solving the SlotMachine flight prioritization problem in a privacy-preserving way we propose to employ an evolutionary algorithm, which allows to separate the finding of solutions to the problem, i.e., flight lists, from the evaluation of the fitness, i.e., the overall utility, of the solutions. Figure 2 illustrates the principle of the optimization using a genetic algorithm, which is a type of evolutionary algorithm. The genetic algorithm starts with a population of solutions to the optimization problem, in this case flight lists, e.g., Flight A is assigned to the 1st slot, Flight B is assigned to the 2nd slot, and so on. The population is then evaluated, each individual is assigned a fitness value – the solution's overall utility. Through recombination and mutation of the evaluated solutions a new generation of solutions is created, replacing the previous population, and the cycle starts again. The genetic algorithm can be stopped at any time. In practice, different termination criteria can be specified, e.g., a time limit or a certain number of generations without increase in maximum fitness of the found solutions.

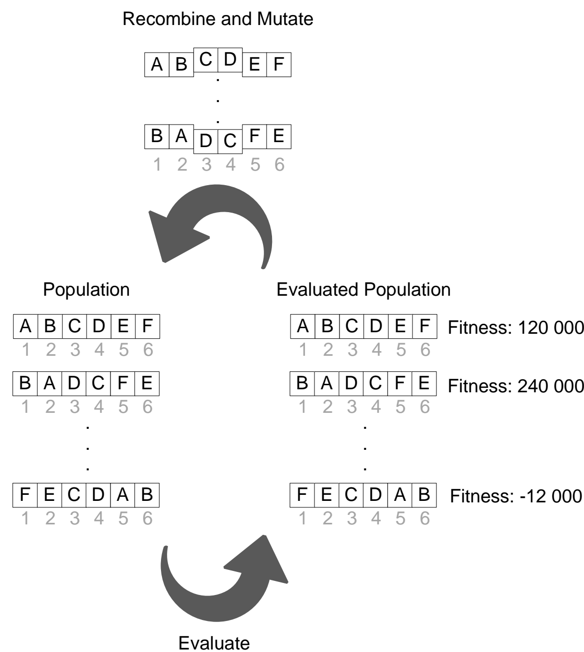
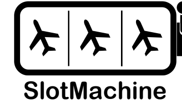


Figure 2. Principle of the iterative optimization process of a genetic algorithm

The Heuristic Optimizer is flexible regarding the employed evolutionary algorithm. In the current version the Heuristic Optimizer employs a genetic algorithm for running optimizations in a privacy-preserving way. Implementations of other types of evolutionary/heuristic optimization algorithms or possibly more efficient implementations of the genetic algorithm, however, may be easily plugged into the general framework of the Heuristic Optimizer.



2.3 Heuristic Optimizer and Privacy Engine

The Heuristic Optimizer is designed to work together with the SlotMachine system's Privacy Engine component, which takes over the evaluation of the individuals in the population of the evolutionary algorithm. The Privacy Engine employs multiparty computation over encrypted weight maps to determine the fitness of the solutions in a privacy-preserving manner (see D3.2 – Specification of the Privacy Engine Component [8]). In order to not leak too much information, the Privacy Engine does not disclose individual fitness values of the entire population to the Heuristic Optimizer. Rather, the Privacy Engine ranks the individuals of a population by fitness value and returns only the maximum fitness value of the population. In this case, an honest-but-curious platform provider may not simply infer the airspace users' preferences from the fitness values of the different populations. The cost for preserving the confidentiality of the inputs may, however, be reduced performance of the optimization algorithm in terms of achieved fitness of the found solution.

Figure 3 illustrates the principle of running the evolutionary optimization algorithm in connection with the Privacy Engine. In each iteration step, a population of candidate solutions (flight lists) is evaluated – which in privacy-preserving mode is taken over by the Privacy Engine. The result of the evaluation is a ranked list of individuals along with the maximum fitness in the population, which is disclosed to the Heuristic Optimizer. The Heuristic Optimizer then estimates the fitness values of the individuals using a specific estimation function; the estimation function used by the Heuristic Optimizer can be configured. The fitness estimates are the basis for recombination and mutation of the solutions, which produce the next generation of the population.

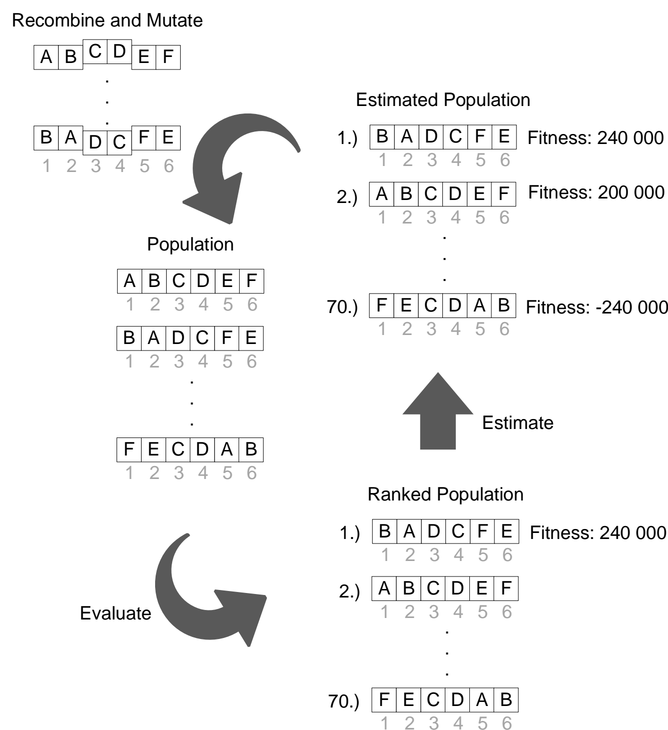
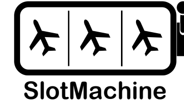


Figure 3. Principle of the iterative optimization algorithm with ranked population and fitness estimation



3 Framework

In this chapter we describe the general framework of the Heuristic Optimizer. The Heuristic Optimizer's implementation focuses on flexibility and extensibility regarding the type of (evolutionary) algorithm used to conduct the optimization of flight lists. The general framework of the Heuristic Optimizer can be used with different types of (evolutionary) algorithms. We refer to Chapter 4 for a concrete implementation of optimization runs using a genetic algorithm.

3.1 Structure

Figure 4 shows the main classes that make up the general framework of the Heuristic Optimizer. The central class is the `OptimizationService`, the methods of which are invoked by the `OptimizationEndpoint`, which realizes the REST interface to be accessed by the Controller (see D2.2 – System Design Document [6]). The `OptimizationService` is a Spring service component. The `OptimizationEndpoint` class is implemented as a Spring REST endpoint; Spring is an open-source framework for the development of web applications[1].

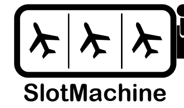
The Heuristic Optimizer is a framework that allows to easily plug in modules for different implementations of (evolutionary) algorithms to solve the SlotMachine flight prioritization problem. In particular, the Heuristic Optimizer employs the *abstract factory* design pattern [16, p. 87ff] to provide increased flexibility regarding the used optimization algorithm. The `OptimizationService` class refers to the abstract classes `Optimization` and `OptimizationFactory`. Which concrete implementations of these abstract classes are used depends on the configuration of the optimization run – more specifically, the `framework` parameter's value in the JSON configuration passed to the REST endpoint during creation and initialization of the optimization run.

The domain model consists of the classes `Flight` and `Slot`, which are used across different implementations of evolutionary algorithms to characterize the result of the optimization – a mapping of flights to slots. The different implementations may have separate internal representations that are used during optimization, e.g., the genetic algorithm employs a `Problem` class and an encoding during the optimization runs.

An `OptimizationConfiguration` defines the characteristics of the optimization. The `OptimizationConfiguration` class is abstract, each module implementing an evolutionary algorithm provides its own concrete implementation of `OptimizationConfiguration`. The `OptimizationConfiguration` class itself contains a `Map<String, Object>` to represent parameters. Concrete implementations may provide convenience methods to write parameters specific to the evolutionary algorithm into the map.

The abstract `FitnessEstimator` is required for the privacy-preserving mode when the Privacy Engine is used to evaluate the population (see Chapter 5).

[1] https://spring.io/



Flight
- margins : Margins
- weights : int[]
- scheduledTime : LocalDateTime
- flightId : String
- Flight(String, LocalDateTime, int[])
- computeWeightMap(Slot[]) : void
- getWeight(Slot) : int
- compareTo(Flight) : int
- flightId : String
- weights : int[]
- scheduledTime : LocalDateTime
- margins : Margins

Slot
- time : LocalDateTime
- Slot(LocalDateTime)
- Slot()
- compareTo(Slot) : int
- time : LocalDateTime

«create»

«create»

OptimizationFactory
- OptimizationFactory()
- createOptimization(Flight[], Slot[]) : Optimization
- createOptimization(Flight[], Slot[], Map<String, Object>) : Optimization

Optimization
- fitnessEstimator : FitnessEstimator
- optId : UUID
- flights : Flight[]
- traceFitnessEvolution : boolean
- privacyEngineService : PrivacyEngineService
- maximumFitness : double
- slots : Slot[]
- mode : OptimizationMode
- privacyEngineEndpoint : String
- results : List<Map<Flight, Slot>>
- status : OptimizationStatus
- Optimization(Flight[], Slot[])
- run() : Map<Flight, Slot>
- newConfiguration(Map<String, Object>) : void
- results : Map<Flight, Slot>[]
- maximumFitness : double
- defaultConfiguration : OptimizationConfiguration
- statistics : OptimizationStatistics
- fitnessEstimator : FitnessEstimator
- flights : Flight[]
- mode : OptimizationMode
- privacyEngineEndpoint : String
- traceFitnessEvolution : boolean
- slots : Slot[]
- optId : UUID
- status : OptimizationStatus
- configuration : OptimizationConfiguration
- privacyEngineService : PrivacyEngineService

OptimizationConfiguration
- OptimizationConfiguration()
- getStringParameter(String) : String
- setParameter(String, Object) : void
- getNumberParameter(String) : Number
- getIntegerParameter(String) : int
- getLongParameter(String) : Long
- getParameter(String) : Object
- getDoubleParameter(String) : double

OptimizationService
- optimizations : Map<UUID, Optimization>
- OptimizationService(PrivacyEngineService)
- runOptimizationAsynchronously(UUID) : Future<OptimizationResultDTO>
- convertResultMapToOptimizationResultMapDto(UUID, Map<Flight, Slot>) : OptimizationResultDTO
- registerThread(UUID, Future<OptimizationResultDTO>) : void
- existsOptimization(UUID) : boolean
- createAndInitializeOptimization(OptimizationDTO) : OptimizationDTO
- getOptimizationResult(UUID, int) : OptimizationResultDTO[]
- runOptimization(UUID) : OptimizationResultDTO
- getOptimization(UUID) : OptimizationDTO
- deleteOptimization(UUID) : OptimizationDTO
- getOptimizationStatistics(UUID) : OptimizationStatisticsDTO
- abortOptimization(UUID) : void
- optimizations : OptimizationDTO[]

OptimizationEndpoint
- OptimizationEndpoint(OptimizationService)
- deleteOptimizationResult(UUID) : ResponseEntity<Void>
- getOptimizationStatistics(UUID) : ResponseEntity<OptimizationStatisticsDTO>
- getOptimization(UUID) : ResponseEntity<OptimizationDTO>
- startOptimizationSync(UUID) : ResponseEntity<OptimizationResultDTO>
- createAndInitializeOptimization(OptimizationDTO) : ResponseEntity<OptimizationDTO>
- getOptimizationResult(UUID, int) : ResponseEntity<OptimizationResultDTO[]>
- startOptimization(UUID) : ResponseEntity<OptimizationDTO>
- abortOptimization(UUID) : ResponseEntity<OptimizationDTO>
- optimizations : ResponseEntity<OptimizationDTO[]>

FitnessEstimator
- FitnessEstimator()
- estimateFitnessDistribution(int, double, double) : double[]
- estimateFitnessDistribution(int, double) : double[]

Figure 4. UML class diagram for optimization runs in the Heuristic Optimizer component

3.2 Creating and Initializing an Optimization

Figure 5 illustrates the internal sequence of the creation and initialization of an optimization within the Heuristic Optimizer. To create and initialize an optimization, the Controller submits the corresponding request to the Heuristic Optimizer's REST endpoint (`OptimizationEndpoint`), which is implemented using Spring Boot; we refer to D2.2 – System Design Document [6] for a specification of the REST interface. The Heuristic Optimizer's REST endpoint then calls the `OptimizationService`.

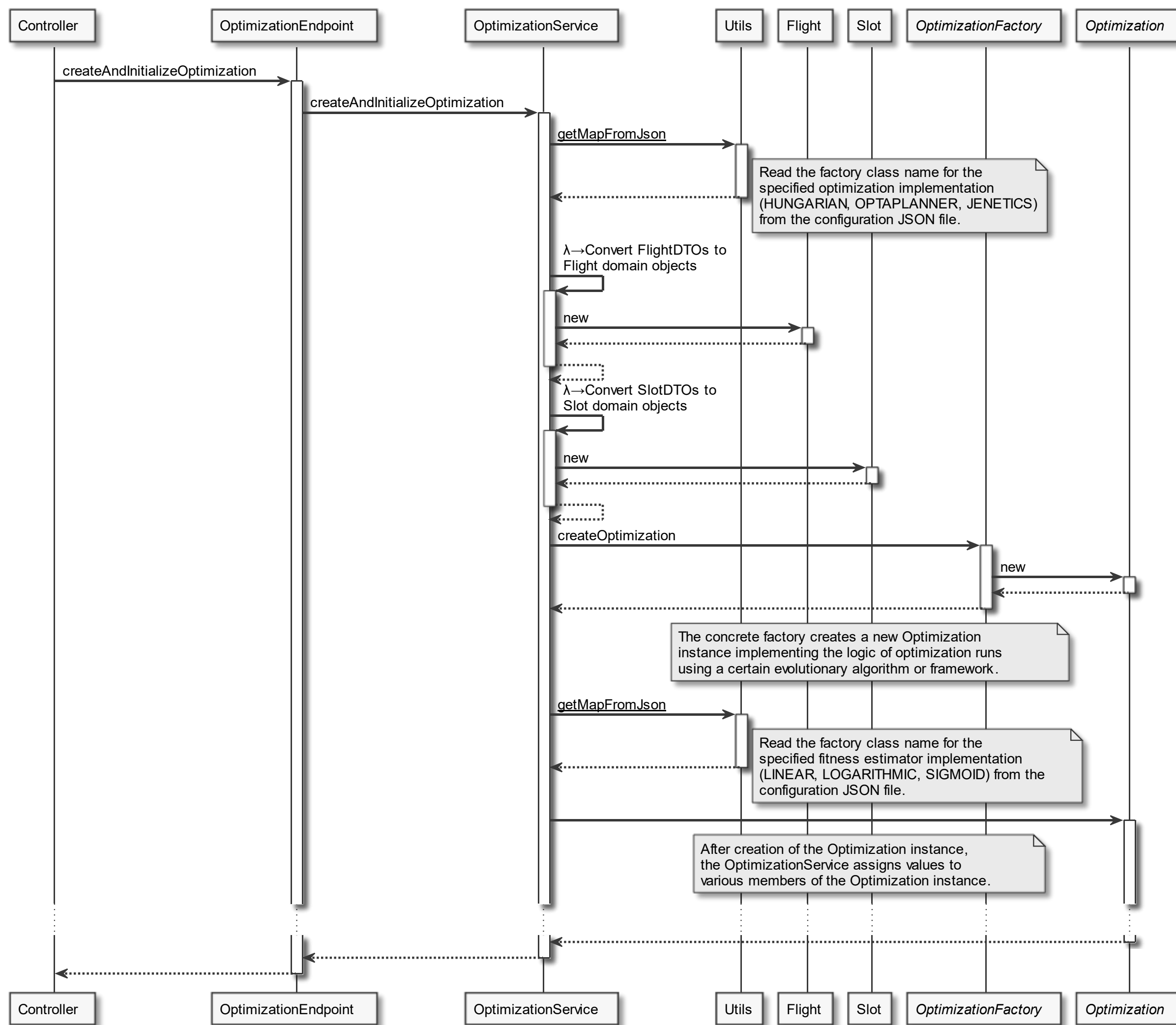


Figure 5. UML sequence diagram of the creation and initialization of an optimization run

3.3 Running an Optimization

Figure 6 illustrates the internal sequence for asynchronously running an optimization and retrieving the results. The optimization can be run in a separate thread.

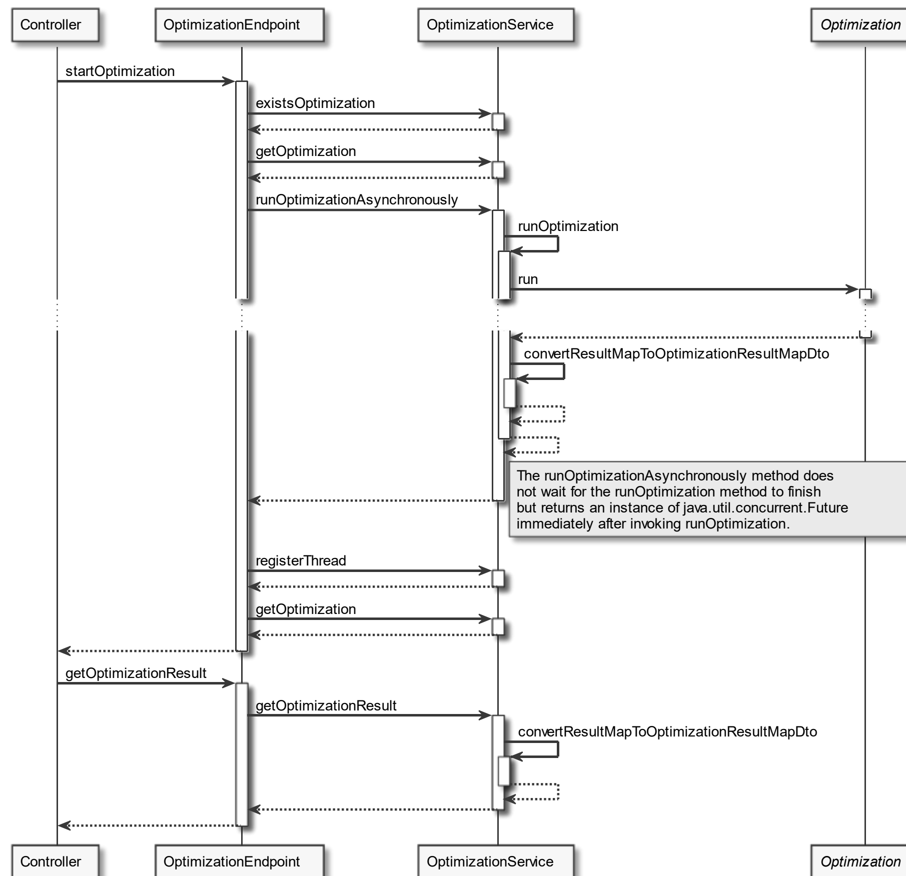
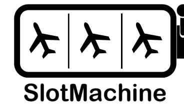


Figure 6. UML sequence diagram for starting an optimization run and retrieving the results



4 Genetic Algorithm

The Heuristic Optimizer's general framework allows to plug in different implementations of evolutionary algorithms to actually conduct the optimization. In order for the Heuristic Optimizer to work there must be at least one such implementation provided. In the following, we describe a genetic algorithm implementation employing the Jenetics framework[2].

4.1 Parameters

The genetic algorithm's behaviour is determined by different parameters that can be set in an optimization run's configuration upon creation and initialization of the optimization run. Consider the following example of an optimization run's configuration in JSON notation.

```
{
  "optId": "237dd935-bfd7-470a-998e-cfcb3be09a18",
  "flights": […],
  "slots": […],
  "optimizationMode": "PRIVACY_PRESERVING",
  "fitnessEstimator": "LOGARITHMIC",
  "optimizationFramework": "JENETICS",
  "parameters": {
    "populationSize": 500,
    "maximalPhenotypeAge": 80,
    "offspringFraction": 0.7,
    "crossover": "PARTIALLY_MATCHED_CROSSOVER",
    "crossoverAlterProbability": 0.35,
    "survivorsSelector": "TOURNAMENT_SELECTOR",
    "survivorsSelectorParameter": 50,
    "mutator": "SWAP_MUTATOR",
    "mutatorAlterProbability": 0.15,
    "offspringSelector": "TOURNAMENT_SELECTOR",
    "offspringSelectorParameter": 50,
    "terminationConditions": {
      "BY_EXECUTION_TIME": 10
    },
  }

}
```

The configuration indicates that the Jenetics framework, i.e., the genetic algorithm module of the Heuristic Optimizer, shall be used for running the optimization in privacy-preserving mode using the logarithmic fitness estimator (see Chapter 5). The parameters array differs between different implementations of evolutionary algorithm; the example shows the parameters for the Jenetics implementation of a genetic algorithm.

[2] https://jenetics.io/

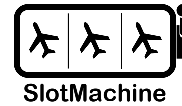
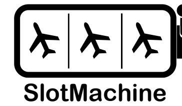


Table 1 describes the parameters of the Jenetics implementation of a genetic algorithm. The parameters are derived from the parameters that are passed to the various classes provided by the Jenetics framework; we refer to the Jenetics manual for further information [4].

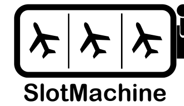
Table 1. Parameters for the genetic algorithm

Parameter	Description	Type
populationSize	The number of individuals considered in each iteration step.	(Positive) Integer
maximalPhenotypeAge	The maximum number of generations an individual is allowed to live	(Positive) Integer
offspringFraction	The fraction of the offspring that is kept in the evolutionary process.	Number in the interval [0, 1[
crossover	The recombination operator used to derive new solutions from the existing population.	String/Enum; in the current implementation only the option "PARTIALLY_MATCHED_CROSSOVER" is supported.
crossoverAlterProbability	The probability that an individual is selected for recombination.	Number in the interval [0, 1[
survivorsSelector	The selector that serves to select the survivors of a generation.	String/Enum; the following options are supported: "BOLTZMANN_SELECTOR", "EXPONENTIAL_RANK_SELECTOR", "LINEAR_RANK_SELECTOR", "ROULETTE_WHEEL_SELECTOR", "STOCHASTIC_UNIVERSAL_SELECTOR", "TOURNAMENT_SELECTOR", "TRUNCATION_SELECTOR".
survivorsSelectorParameter	The parameter passed to the chosen survivors selector.	Object; depending on the chosen offspring selector, the parameter is either an integer or a double value with a different meaning. Some selectors do not require a parameter.
mutator	The alterer used to mutate individuals in a population.	String/Enum; In the current implementation only the option "SWAP_MUTATOR" is supported.



Parameter	Description	Type
mutatorAlterProbability	The probability that an individual is selected for mutation.	Number in the interval [0, 1[
offspringSelector	The selector that serves to select the offspring that is kept in the evolutionary process.	String/Enum; the following options are supported: “BOLTZMANN_SELECTOR”, “EXPONENTIAL_RANK_SELECTOR”, “LINEAR_RANK_SELECTOR”, “ROULETTE_WHEEL_SELECTOR”, “STOCHASTIC_UNIVERSAL_SELECTOR”, “TOURNAMENT_SELECTOR”, “TRUNCATION_SELECTOR”.
offspringSelectorParameter	The parameter passed to the chosen offspring selector.	Object; depending on the chosen offspring selector, the parameter is either an integer or a double value with a different meaning. Some selectors do not require a parameter.
terminationConditions	The conditions that determine when the genetic algorithm stops.	Map<String/Enum, Object>; as key, the following values are supported: “BY_EXECUTION_TIME”, “WORST_FITNESS”, “BY_FITNESS_THRESHOLD”, “BY_STEADY_FITNESS”, “BY_FIXED_GENERATION”, “BY_POPULATION_CONVERGENCE”, “BY_FITNESS_CONVERGENCE”. The type of the value depends on the selected termination condition and represents the parameter for the termination condition.

The only recombination operator currently supported by the Heuristic Optimizer’s genetic algorithm module (not the Jenetics framework, which provides more implementations) is the partially matched crossover (PMX), which achieves good performance for the assignment problem. The only mutator currently supported by the module (again, Jenetics provides more) is the swap mutator. Regarding the choice of survivors and offspring selectors, the tournament selector is “often used in practice because of its lack of stochastic noise” and due to being “independent to the scaling of the genetic algorithm fitness function” [4, p. 13]. In preliminary experiments, some of the other selectors showed minimally better results on a first set of test data we employed for gauging the settings. In our experiments, however, we then focused on the tournament selector.



4.2 Implementation

We use the Jenetics framework to implement a genetic algorithm for the SlotMachine flight prioritization problem. Jenetics is a framework that facilitates the implementation of genetic algorithms. The Jenetics framework provides efficient encodings of common optimization problems and implements the most common recombination operators as well as selectors. The Jenetics framework's flexibility would also allow for the easy extension with new representations, recombination operators, and selectors in the future if necessary for improving performance of the SlotMachine system.

4.2.1 Structure

Figure 7 illustrates the structure of the Heuristic Optimizer's genetic algorithm module. The `JeneticsOptimization` class is an implementation of the abstract `Optimization` class using the Jenetics framework. The `JeneticsOptimizationFactory`, which is a specialization of the abstract `OptimizationFactory` class, creates an instance of `JeneticsOptimization`. The `JeneticsOptimization` class references a `JeneticsOptimizationConfiguration`, which is a specialization of `OptimizationConfiguration` with methods for setting the parameters of the genetic algorithm (see Section 4.1). The `JeneticsOptimizationConfiguration` determines how the Heuristic Optimizer runs the genetic algorithm to find solutions to the optimization problem. Internally, `JeneticsOptimizationConfiguration` employs a string–object map for storing the parameter values. This map can be passed to the `create` method of the `OptimizationFactory` class upon creation of the `JeneticsOptimization` instance.

The `JeneticsOptimization`'s `run` method employs a genetic algorithm to solve a `SlotConfigurationProblem` – an implementation of the Jenetics framework's `Problem` interface. An implementation of `Problem` comprises a definition of a *fitness function* and a *codec*. In case of the Heuristic Optimizer, the fitness function is not used in privacy-preserving mode (although the fitness function is used in non-privacy-preserving mode). The codec of the `SlotConfigurationProblem` corresponds to the Jenetics framework's built-in mapping codec (`Codec.ofMapping`), which handles transformations between an efficient encoding of a flight list used by the genetic algorithm and the more natural representation as a mapping of flights to slots (`Map<Flight, Slot>`).

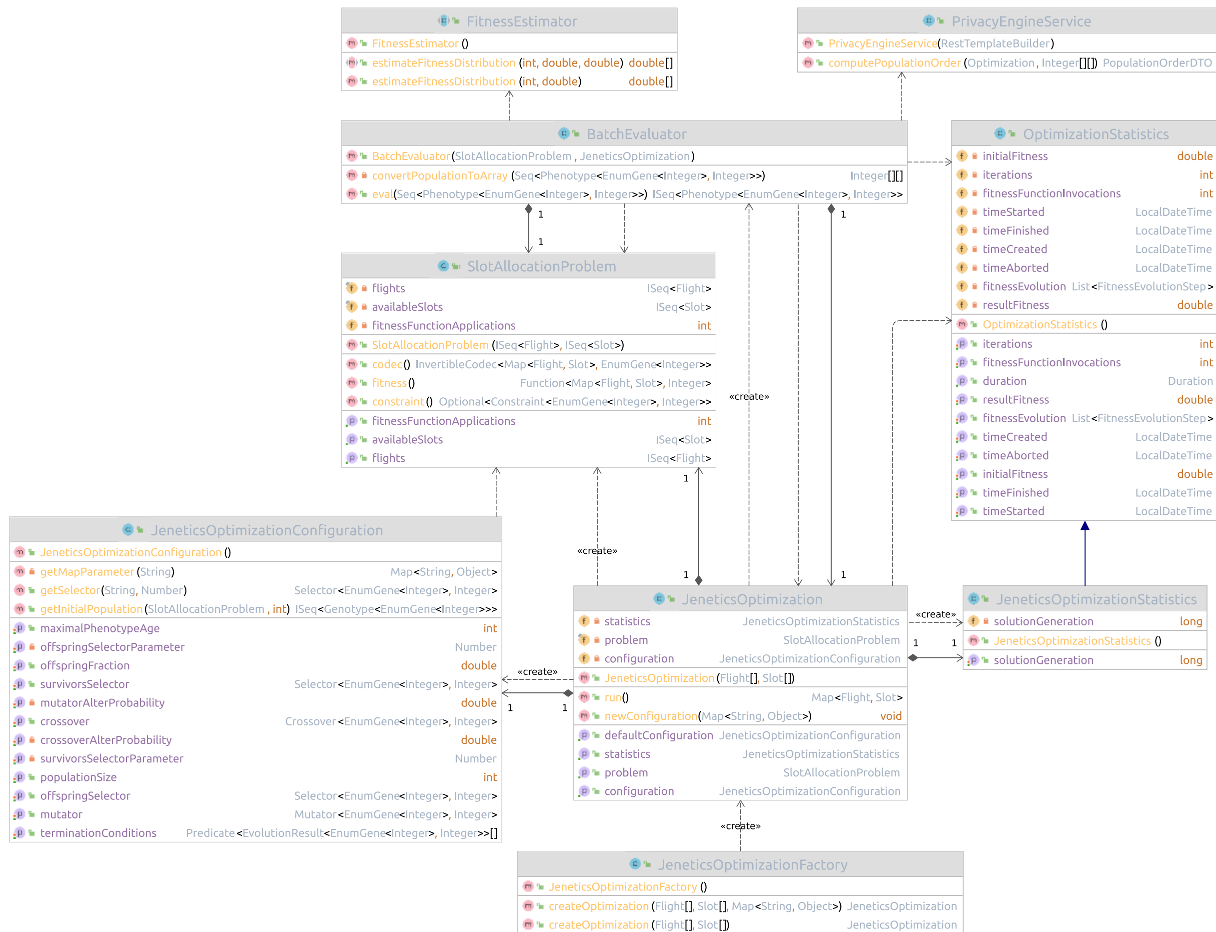


Figure 7. UML class diagram for initiating and running optimizations using the Jenetics framework

4.2.2 Creating and Initializing an Optimization

Figure 8 illustrates the internal sequence of creating and initializing a `JeneticsOptimization` run. The `JeneticsOptimizationFactory` initializes the optimization with the flights and slots as well as a configuration, setting the parameters of the optimization specified in the optimization run's configuration. Note that the `computeWeightMap` method of the `Flight` class is only invoked in case of non-privacy-preserving optimization runs.

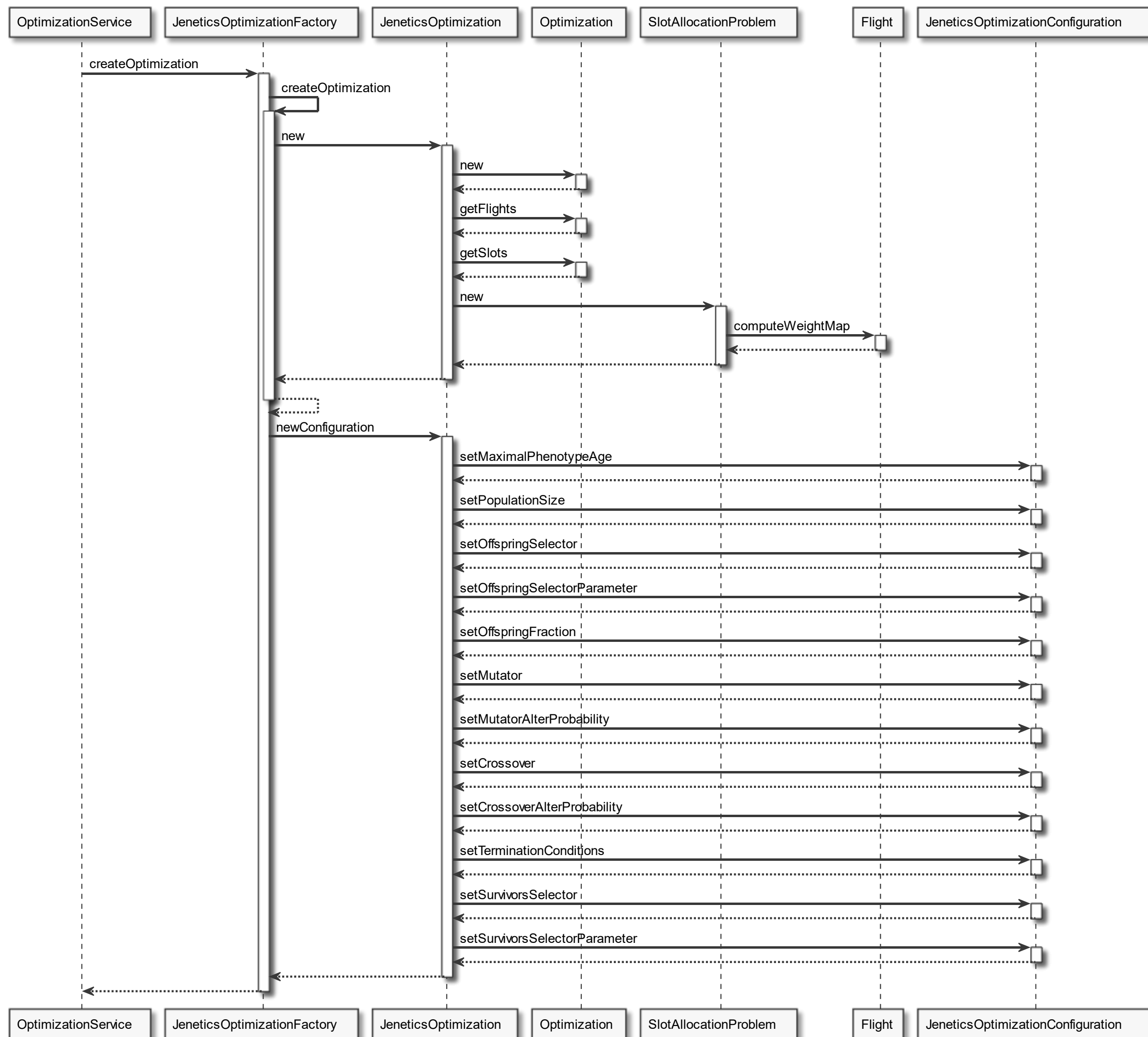
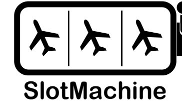


Figure 8. UML sequence diagram of the creation of an optimization run using the Jenetics framework



4.2.3 Running an Optimization

Figure 9 illustrates the internal sequence of running a `JeneticsOptimization`. First, the `JeneticsOptimization` obtains an initial population of flight lists as input for the genetic algorithm. The `JeneticsOptimizationConfiguration` returns the initial population for the genetic algorithm. The initial population is chosen by assigning the flights to the available slots in the order of their originally scheduled time and then swapping neighbouring flights to obtain different flight lists. The flight lists are represented as mappings from flights to slots, which are encoded using the `encode` function of the `SlotAllocationProblem`'s codec. The `JeneticsOptimization` run then initiales the optimization run by building a Jenetics `Engine` instance with a `BatchEvaluator` instance as well as the `SlotAllocationProblem`'s encoding and constraint.

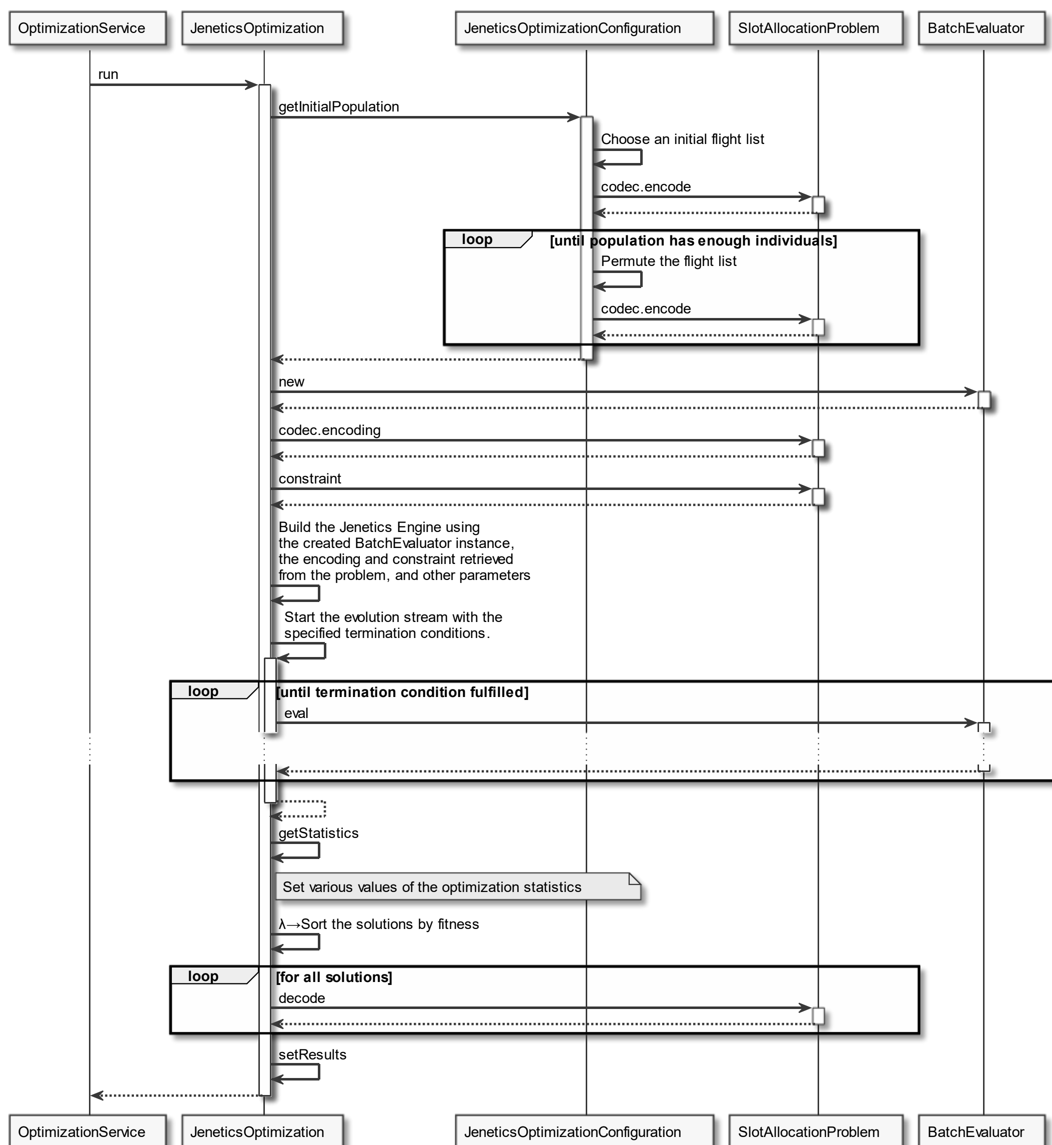
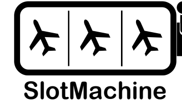


Figure 9. UML sequence diagram of an optimization run using the Jenetics framework



Once started, the Jenetics `EvolutionStream` obtained via the engine continues until one of the termination conditions is met. In addition to the termination conditions specified via the optimization configuration, another termination condition is whether the thread of the optimization run has been interrupted. That way, the optimization run can be aborted via the REST interface at any time and will still terminate properly.

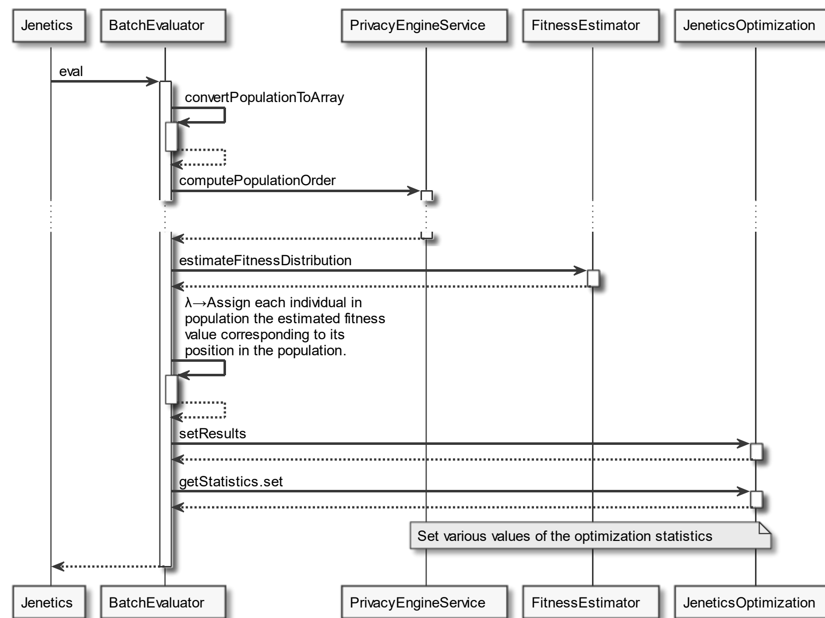
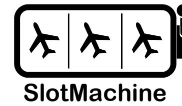


Figure 10. UML sequence diagram of batch evaluation of a population of flight lists

The Jenetics framework allows to implement a custom `Evaluator` class that takes over the evaluation of a population and can be passed to the engine. Figure 10 illustrates the internal sequence of evaluating a population using the `BatchEvaluator` class. The Jenetics evolution stream invokes the `BatchEvaluator`'s `eval` method for every generation. The `eval` method calls the `PrivacyEngineService`'s `computePopulationOrder` method, which establishes the connection to the Privacy Engine's REST interface. The `eval` method then calls the concrete `FitnessEstimator` implementation indicated by the configuration to obtain a distribution of fitness values, which serves to assign each individual a fitness value in the privacy-preserving setting. After evaluation of the population, the optimization's intermediate results and statistics are updated, which allows to obtain intermediate results and statistics at any time during the run via the REST interface.



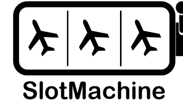
5 Fitness Function

A central component of a genetic algorithm is the fitness function, which allows to evaluate the individuals of a population of candidate solutions to an optimization problem. In SlotMachine, the privacy-preserving computation of the fitness of individuals requires a peculiar approach to implementing the fitness function for the employed evolutionary algorithm. In order to not disclose too much information about airspace users' preferences, the Privacy Engine that computes the fitness values does not return an absolute fitness value for every individual in a population. Rather, the Privacy Engine returns the ranks of individuals within a population along with the maximum fitness value in the population. When such relative fitness values are available to the Heuristic Optimizer, a *fitness estimator* is used to obtain an approximate fitness value for each flight list.

If the Heuristic Optimizer runs in privacy-preserving mode, the Heuristic Optimizer receives from the Privacy Engine only the rank of each solution within a population as well as the maximum fitness value within that population. The Heuristic Optimizer then uses a *fitness estimator* to obtain an approximate fitness value for each flight list in a population. Different approaches may be used to estimate the fitness of a flight list when given only the maximum fitness of the population and the rank of the flight list within the population. The Heuristic Optimizer provides implementations of linear, logarithmic, and sigmoid estimators but the architecture allows for easily extending the family of fitness estimators. In the following we briefly explain the intuition behind the provided implementations.

A fitness estimator takes the population size and the maximum/minimum fitness as input and returns an array of the length of the population size as output. The array contains fitness values, which are ordered from highest to lowest. The distribution of the values follows a linear, logarithmic, or sigmoid function, depending on the type of fitness estimator used. In the current implementation, when estimating fitness values, the minimum fitness is assumed to be equal to twice the absolute value of the maximum fitness subtracted from the maximum fitness. Figure 11 shows example distributions of fitness values estimated using the implemented fitness estimators – linear fitness estimator, logarithmic fitness estimator, and sigmoid fitness estimator – with a population size of 500 as well as a maximum fitness of 540 000 and a minimum fitness of -540 000. The vertical axis shows the fitness value. The horizontal axis shows the different positions/ranks within a population. The diagram thus plots the estimated fitness of the ith individual in a population ranked by fitness value.

The use of a fitness estimator represents a trade-off between information privacy and the quality of the result of the optimization algorithm. If absolute fitness values were returned by the Privacy Engine then it would be relatively easy to infer the confidential preferences submitted by the airspace users. Our experiments, however, suggest that the use of the fitness estimator does not have a considerable negative impact on the quality of the result of the optimization algorithm (see Section 6.2). The estimator that most closely resembles the real distribution seems to be the logarithmic estimator, although here further research may lead to improved implementations of fitness estimators.



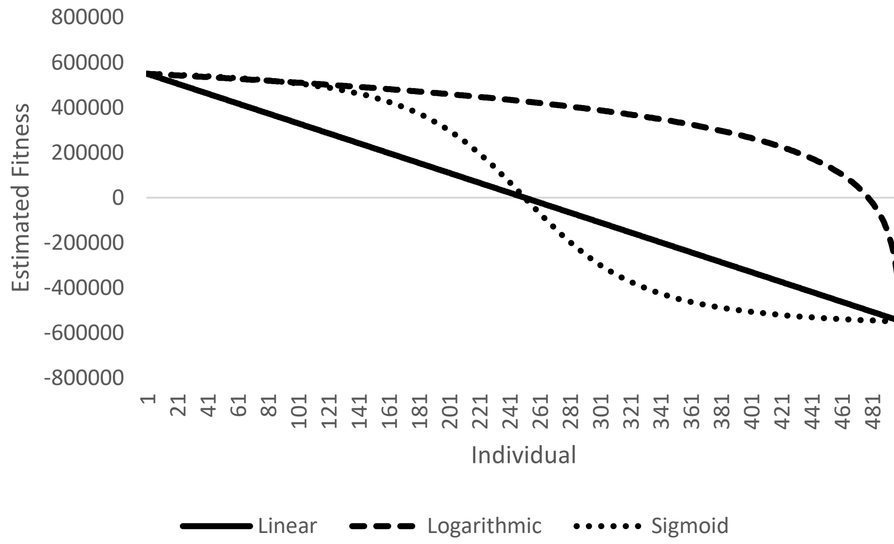
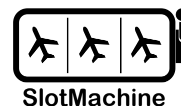


Figure 11. Distribution of fitness values for different fitness estimators



6 Evaluation

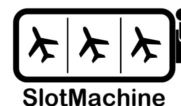
In this chapter we evaluate the implementation of the Heuristic Optimizer. First, we look at the implementation's relation to the requirements. We then look at the results of experiments conducted using the Heuristic Optimizer.

6.1 Relation to Requirements

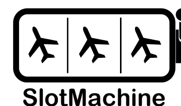
In the following we discuss the design and implementation decisions regarding the Heuristic Optimizer in relation to the SlotMachine requirements identified in D2.1 – Requirements Specification [5]. During the design and implementation of the Heuristic Optimizer, we discovered also that some functional requirements should be adapted.

Table 2. Requirements from D2.1 and their implications on the design of the Heuristic Optimizer

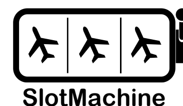
Req.	Description	Design Implications
perf_2	The solution shall support handling at least MIN_FLIGHT_NUMBER flights per flight prioritization session. *Necessary* requirement *as you need at least two flights to swap.*	We conducted experiments with different scenarios involving up to 100 flights (see D4.1 and Section 6.2). The results of these experiments suggest good feasibility of a SlotMachine flight prioritization with 100 flights. The result of those experiments, but also informal experiments during development, suggest good feasibility of optimization runs with even more flights since our experiments were conducted using a run time of up to 10 seconds only. In practice, even with the overhead of encryption, we expect considerably more time for conducting an individual optimization run.
perf_3	The found flight prioritization solutions shall be MIN_EFFICIENCY % more efficient than the existing flight list provided by the Network Manager in terms of the preferences submitted by the airspace users. *The SlotMachine system should provide a gain with respect to the existing systems in place, otherwise it would not be needed. The gain can be measured by comparing the overall utility of the baseline flight list with the flight list found by the SlotMachine using*	From the experiments with generated test data according to different scenarios (see Section 6.2 and D4.1 – State of the Art of Relevant Concepts) we know that the heuristic optimization finds solutions close to the optimum found by the non-deterministic optimization algorithm.



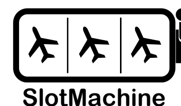
Req.	Description	Design Implications
	the preferences (utilities of slots) submitted by the airspace users.	
perf_4	A flight prioritization session should last at most MAX_SESSION_DURATION minutes. *Necessary requirement due to the synchronization process with the airport/airspace users, which takes some time.*	The Heuristic Optimizer allows to specify a time limit for an optimization run. In our experiments using the non-privacy-preserving mode, we ran optimizations with a fixed time of 10 seconds, which already yielded good results with respect to the deterministic optimum. Even with the overhead of encryption and privacy-preserving computation, a time limit imposed by operational constraints will likely not be a problem for running optimizations.
perf_6	The Heuristic Optimizer shall finish the optimization after reaching a specified threshold in terms of fitness/utility of the found solution or after a specified amount of time has passed for the optimization session and return the best flight prioritization found until that point. *See REQ perf_4. The optimization process cannot run forever and must return a result. The optimization algorithm must be chosen such that it will return a result no matter when it is aborted.*	The Heuristic Optimizer allows to specify a fitness threshold and/or a maximum run time as the criterion for finishing the optimization run. The Heuristic Optimizer also allows to specify other criteria for finishing an optimization run, e.g., when the found solutions do not improve in fitness any more. The implementation of the Heuristic Optimizer also supports aborting an optimization run at any time while still returning the best result obtained thus far. Furthermore, after each iteration step, the Heuristic Optimizer stores improved results in a cache, which allows to obtain intermediate results at any time.
priv_3	The airspace user (AU) flight prioritization preferences shall remain confidential and protected from honest-but-curious platform operator individuals.	The Heuristic Optimizer has a privacy-preserving mode. In this mode, the Heuristic Optimizer calls the Privacy Engine for evaluating a population of solutions to the flight prioritization problem. The Privacy Engine does not return a fitness value for each solution sent by the Heuristic Optimizer. Rather, the Privacy Engine returns only the rank of each solution within a population submitted by the Heuristic Optimizer along with the maximum fitness within the population.



Req.	Description	Design Implications
priv_4	The AU flight prioritization preferences shall be processed in encrypted/encoded form only in the platform.	In privacy-preserving mode, the weights remain encrypted and are not read by the Heuristic Optimizer.
priv_5	The internally used representation of AU flight prioritization by SlotMachine shall be securely derived from AU input for each flight prioritization and flight considered (e.g. weights).	In privacy-preserving mode, the weights of flights are not processed by the Heuristic Optimizer. The Controller submits the encrypted weight maps directly to the Privacy Engine.
co_4	The Controller shall call the Heuristic Optimizer to receive a ranked list of optimized flight prioritization based on the initial flight prioritization received from the NMF.	When retrieving the results of an optimization run from the Heuristic Optimizer, the Controller may specify a limit, i.e., how many results should be returned. The number of available results depends on the specific settings of the evolutionary algorithm. For example, if the population size of the genetic algorithm is 500 then up to 500 results are available but typically fewer since some of the found solutions will be duplicates. We expect, however, that for practical purposes there will be enough distinct solutions to find a flight list acceptable for all airspace users.
ho_1	The Heuristic Optimizer shall evaluate each flight prioritization independently from other flight prioritizations.	The Heuristic Optimizer uses fitness values of the flight lists found in one iteration step to determine how to modify the found solutions in the next iteration step. In order to protect the confidentiality of the slot preferences submitted by the airlines, the Privacy Engine does not return a fitness value for each individual flight list but only the maximum fitness of a population along with the ranks of the solutions within the population (relative fitness). The Heuristic Optimizer then estimates the fitness value of each individual flight list in a population of solutions.
ho_2	The Heuristic Optimizer shall receive information about flights from the Controller.	In the current implementation, the Heuristic Optimizer receives only the flight identifier and the scheduled time, which is required to determine invalid solutions; a flight shall not be planned for a slot earlier than the originally scheduled departure time, i.e., “the time on the ticket”. The Heuristic



Req.	Description	Design Implications
		Optimizer could be extended with a rule engine that uses additional information about flights, e.g., the size of the aircraft, to find better initial solutions. Based on the experiments conducted so far, we do not expect such more elaborate approach to finding initial solutions to be necessary in order to achieve good results.
ho_3	The Heuristic Optimizer shall receive encrypted flight prioritization preferences of AUs from the Controller.	We changed the design such that the Heuristic Optimizer does not receive any preferences regarding slots when running in privacy-preserving mode. Rather, the Controller submits the preferences directly to the Privacy Engine. In non-privacy-preserving mode, the Heuristic Optimizer receives the preferences in unencrypted, plain form.
ho_4	The Heuristic Optimizer shall receive public flight information from the Controller.	See ho_2
ho_5	The Heuristic Optimizer shall generate flight prioritizations under consideration of public flight information.	See ho_2
ho_6	The Heuristic Optimizer shall initialize a Privacy Engine session with encrypted flight prioritization preferences.	The Controller initializes the Privacy Engine session. We deviate from the original requirement in order to keep the Heuristic Optimizer component more flexible. The Controller should take care of all administration issues while the Heuristic Optimizer's focus is on providing the implementation of the evolutionary algorithm. The Controller submits the connection details of the employed and configured Privacy Engine instance to the Heuristic Optimizer.
ho_7	The Heuristic Optimizer shall use Privacy Engine to evaluate fitness of generated flight prioritizations.	The Heuristic Optimizer may run in privacy-preserving or non-privacy-preserving mode. In privacy-preserving mode, the Heuristic Optimizer submits found solutions to the Privacy Engine for evaluation.



Req.	Description	Design Implications
ho_8	The Heuristic Optimizer shall return a ranked list of flight prioritizations to the Controller.	The Controller may specify how many flight lists should be returned by the Heuristic Optimizer upon retrieval of the results of an optimization run. The returned flight lists are ranked from best to worst.
ho_9	The Heuristic Optimizer shall always return a solution, independent of the run time, meaning that an optimization can be aborted at any time and still return a valid result. *See REQ perf_6.*	The Heuristic Optimizer caches a population of solutions found in an iteration step if the population's best flight list is better than the best solution found thus far. Furthermore, if a genetic algorithm is used, the algorithm can be aborted at any time and still return a result. The implementation of the Heuristic Optimizer supports aborting the optimization run at any time (after the first iteration step has finished) while still returning a valid flight list.

6.2 Experiments

In order to evaluate the impact of using the `BatchEvaluator` class with relative fitness values we conducted experiments with generated test data corresponding to different scenarios. We used the scenarios and test data from previous experiments, the results of which are reported in D4.1 – Report on State-of-the-Art of Relevant Concepts [9]. We refer to that document for additional experiment using the non-privacy-preserving prototype with different configurations of genetic algorithms and other heuristic local search algorithms, using absolute fitness values.

6.2.1 Setup

Table 3 lists the cases that are used for each selected optimizer configuration. The parameter *no. of slots* refers to the number of slots/flights that have to be matched. The parameter *run time* refers to the amount of time which the optimization algorithm is allowed to run; the optimization is stopped when the time is over and the best solution found up to that point is the optimization result. We refer to Chapter 6 of D4.1 [9] for a detailed explanation of the parameters *concentration*, *priorities*, and *margins*. Intuitively, an "even" concentration refers to cases where each flight desires a separate slot, which makes optimization trivial. An "extreme" concentration refers to cases where many flights desire the same few slots. A "moderate" concentration refers to cases where there are some overlaps between flights regarding the desired slots but the times wished are not concentrated on only a few slots. Priorities in the "middle" refers to cases where the flights with wished time slots in the middle of the available timeline of slots have generally higher priorities. Priorities at the "fringes" refers to cases where the flights with wished time slots at the fringes of the available timeline of slots have generally higher priorities. "Even" priorities refers to cases where all flights are equally important. Finally, "broad" margins refers to cases where the time window around the time wished is ± 25 minutes, "normal" margins is ± 15 minutes, and "narrow" margins is ± 5 minutes.

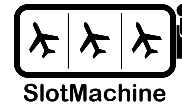


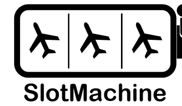
Table 3. Cases used in experiments

Case #	Concentration	Priorities	Margins	No. of Slots	Run Time
1	Even	Middle	Broad	100	10 s
2	Extreme	Middle	Broad	100	10 s
3	Moderate	Middle	Broad	100	10 s
4	Even	Fringes	Broad	100	10 s
5	Extreme	Fringes	Broad	100	10 s
6	Moderate	Fringes	Broad	100	10 s
7	Even	Even	Broad	100	10 s
8	Extreme	Even	Broad	100	10 s
9	Moderate	Even	Broad	100	10 s
10	Even	Middle	Normal	100	10 s
11	Extreme	Middle	Normal	100	10 s
12	Moderate	Middle	Normal	100	10 s
13	Even	Fringes	Normal	100	10 s
14	Extreme	Fringes	Normal	100	10 s
15	Moderate	Fringes	Normal	100	10 s
16	Even	Even	Normal	100	10 s
17	Extreme	Even	Normal	100	10 s
18	Moderate	Even	Normal	100	10 s
19	Even	Middle	Narrow	100	10 s
20	Extreme	Middle	Narrow	100	10 s
21	Moderate	Middle	Narrow	100	10 s
22	Even	Fringes	Narrow	100	10 s
23	Extreme	Fringes	Narrow	100	10 s
24	Moderate	Fringes	Narrow	100	10 s
25	Even	Even	Narrow	100	10 s
26	Extreme	Even	Narrow	100	10 s
27	Moderate	Even	Narrow	100	10 s

For our experiments, we selected the three most promising genetic algorithm configurations from those investigated in D4.1 – Report on State-of-the-Art of Relevant Concepts [9]. Table 4 lists the parameters of the selected genetic algorithm configurations used in the experiments. The Configuration J1 corresponds to Configuration 2 in the experiments from D4.1, Configuration J2 corresponds to Configuration 6, and Configuration J3 corresponds to Configuration 14. The maximal phenotype age was always 80, the offspring fraction 0.7.

Table 4. Genetic algorithm configurations used in experiments

Configuration	Population Size	Tournament Size	Mutator Alter Probability	Crossover Alter Probability
J1	500	50	0.15	0.90
J2	500	10	0.15	0.90
J3	70	10	0.15	0.90



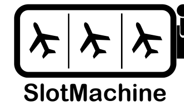
We ran the experiments on an OpenVZ virtual machine on a physical machine with an Intel Xeon CPU E5-2640 v4 with 2.40 GHz. The virtual machine had 8 GB of main memory and could use up to 40 cores of the physical CPU. The operating system of the virtual machine was CentOS Linux 7. The Java Virtual Machine ran with a heap size of 2 GB. We used OpenJDK 16 for running the Heuristic Optimizer.

6.2.2 Results

Our experiments were, on the one hand, aimed at finding out whether using an evolutionary algorithm with relative fitness values is a feasible approach and, on the other hand, which method for estimating fitness values works best given the information returned by the Privacy Engine. The benchmark for the genetic algorithm configurations is the Hungarian algorithm, which will find the optimal solution in terms of fitness but cannot be used together with the Privacy Engine. We first compare the different genetic algorithm configurations running with no estimator, i.e., computing an absolute fitness value for each individual in the population and using the absolute fitness value as the basis for finding new solutions, against the Hungarian algorithm. We then run the same genetic algorithm configurations using different types of fitness estimators; in those runs, the estimated fitness value serves as the basis for finding new solutions.

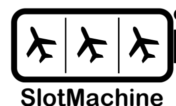
Figure 12 summarizes the results of running optimizations with no estimator (using actual fitness values) for the different cases. Light green denotes cases where the respective configuration found a solution with 90% or more of the optimal fitness. Light yellow denotes cases where the respective configuration found a solution with between 80% and 90% of the optimal fitness. Light red denotes cases where the respective configuration found a solution with less than 80% of the optimal fitness. In general, the genetic algorithm configurations perform well. We note that in general, the genetic algorithm optimization works well. The cases where the best solution found by the genetic algorithm is less than 80% of the optimal fitness are those where a reasonably “good” solution cannot be found due to narrow margins given by the airspace users and/or concentrated airspace user preferences for only a few slots. In that case, the optimal solution is simply the “least bad” solution and in practice, in such situations, there might not even be a globally acceptable solution for airspace users anyway. We also note that in our experiments, the run time of the genetic algorithm was set rather low, with 10 seconds. Longer run times may lead to improved solutions closer to the optimum.

Figure 13 shows results of running genetic algorithm optimizations with the linear estimator, Figure 14 shows the results of the optimizations with the logarithmic estimator, and Figure 15 shows the results of the optimizations with the sigmoid estimator. The results indicate that the use of an estimator does not considerably input the quality of the outcome of the optimization. Overall, we conclude that the proposed approach with a genetic algorithm and relative fitness values returned by the Privacy Engine is indeed feasible and can further be pursued.



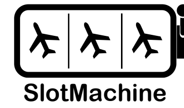
	Case #	No estimator / average fitness over 5 runs				No estimator / best fitness over 5 runs			
Configuration		J1	J2	J3	Hungarian	J1	J2	J3	Hungarian
Avg % of Hungarian		92.11%	93.06%	91.26%	100.00%	94.78%	93.52%	92.82%	100.00%
Rank		3	2	4	1	2	3	4	1
SUM		76.45	77.24	75.74	83.00	78.67	77.62	77.04	83.00
	1	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
	2	86.58%	87.94%	86.75%	100.00%	87.89%	88.68%	88.79%	100.00%
	3	98.94%	99.04%	99.00%	100.00%	99.11%	99.30%	99.14%	100.00%
	4	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
	5	94.02%	93.58%	94.43%	100.00%	94.70%	94.40%	94.73%	100.00%
	6	93.86%	94.31%	93.97%	100.00%	94.64%	94.77%	94.24%	100.00%
	7	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
	8	91.09%	90.09%	90.56%	100.00%	92.58%	91.85%	92.27%	100.00%
	9	96.76%	96.91%	96.38%	100.00%	97.48%	97.32%	96.99%	100.00%
	10	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
	11	84.04%	81.73%	84.48%	100.00%	85.46%	87.16%	86.35%	100.00%
	12	88.18%	88.32%	85.96%	100.00%	90.20%	90.31%	87.44%	100.00%
	13	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
	14	91.58%	90.25%	91.32%	100.00%	92.41%	91.83%	93.64%	100.00%
	15	90.48%	90.64%	91.12%	100.00%	92.50%	91.68%	92.53%	100.00%
	16	100.00%	100.00%	99.99%	100.00%	100.00%	100.00%	100.00%	100.00%
	17	85.16%	83.91%	86.68%	100.00%	87.24%	86.65%	87.96%	100.00%
	18	87.53%	85.31%	87.76%	100.00%	89.25%	87.77%	90.22%	100.00%
	19	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
	20	67.06%	60.93%	64.56%	100.00%	71.02%	65.90%	70.68%	100.00%
	21	73.46%	69.74%	72.93%	100.00%	79.69%	71.61%	76.98%	100.00%
	22	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
	23	78.07%	79.23%	78.87%	100.00%	80.28%	82.43%	84.10%	100.00%
	24	78.38%	75.74%	75.97%	100.00%	79.91%	82.08%	78.51%	100.00%
	25	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
	26	69.55%	72.09%	71.25%	100.00%	75.32%	74.05%	75.38%	100.00%
	27	72.10%	68.65%	72.59%	100.00%	75.36%	72.17%	75.37%	100.00%

Figure 12. Average/best fitness over five runs of various genetic algorithm configurations using no fitness estimator, i.e., evaluation through computation of absolute fitness.



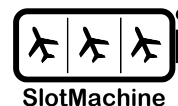
	Case #	Linear estimator / average fitness over 5 runs				Linear estimator / best fitness over 5 runs			
Configuration		J1	J2	J3	Hungarian	J1	J2	J3	Hungarian
Avg % of Hungarian		92.13%	92.66%	91.09%	100.00%	93.51%	94.04%	92.48%	100.00%
Rank		3	2	4	1	3	2	4	1
SUM		76.47	76.91	75.60	83.00	77.62	78.05	76.75	83.00
	1	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
	2	88.22%	86.18%	87.50%	100.00%	89.64%	87.19%	90.13%	100.00%
	3	98.99%	98.94%	98.96%	100.00%	99.10%	98.99%	99.06%	100.00%
	4	100.00%	100.00%	99.99%	100.00%	100.00%	100.00%	100.00%	100.00%
	5	93.81%	93.18%	93.86%	100.00%	95.06%	94.35%	95.05%	100.00%
	6	94.78%	93.96%	94.38%	100.00%	95.01%	94.18%	94.86%	100.00%
	7	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
	8	91.57%	89.52%	90.67%	100.00%	92.00%	91.84%	91.92%	100.00%
	9	96.61%	96.78%	96.61%	100.00%	97.42%	96.98%	97.29%	100.00%
	10	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
	11	83.27%	82.13%	82.19%	100.00%	84.77%	83.12%	83.57%	100.00%
	12	87.25%	88.24%	89.83%	100.00%	88.11%	89.23%	91.48%	100.00%
	13	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
	14	90.80%	89.84%	91.38%	100.00%	91.65%	90.78%	92.32%	100.00%
	15	90.83%	91.03%	91.17%	100.00%	92.07%	92.00%	93.20%	100.00%
	16	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
	17	85.62%	84.48%	87.12%	100.00%	86.94%	85.43%	90.32%	100.00%
	18	85.79%	84.76%	85.90%	100.00%	86.59%	85.69%	89.07%	100.00%
	19	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
	20	68.33%	58.08%	65.37%	100.00%	74.05%	68.58%	70.75%	100.00%
	21	73.74%	74.33%	72.01%	100.00%	76.66%	77.06%	75.33%	100.00%
	22	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
	23	76.46%	78.75%	77.49%	100.00%	81.45%	81.23%	79.17%	100.00%
	24	74.94%	74.39%	76.64%	100.00%	78.17%	77.33%	77.65%	100.00%
	25	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
	26	69.84%	69.87%	69.60%	100.00%	76.49%	73.31%	74.40%	100.00%
	27	71.34%	69.41%	73.57%	100.00%	75.71%	75.29%	77.08%	100.00%

Figure 13. Average/best fitness over five runs of various genetic algorithm configurations using a linear fitness estimator when maximum fitness of a population and a ranking of individuals within the population is known.



	Case #	Logarithmic estimator / average fitness over 5 runs				Logarithmic estimator / best fitness over 5 runs			
Configuration		J1	J2	J3	Hungarian	J1	J2	J3	Hungarian
Avg % of Hungarian		89.43%	89.37%	89.99%	100.00%	90.94%	91.14%	91.62%	100.00%
Rank		3	4	2	1	4	3	2	1
SUM		24.15	24.13	24.30	27.00	24.55	24.61	24.74	27.00
	1	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
	2	88.29%	85.18%	87.16%	100.00%	89.48%	88.06%	90.44%	100.00%
	3	98.99%	98.96%	98.92%	100.00%	99.07%	99.20%	99.02%	100.00%
	4	100.00%	100.00%	99.99%	100.00%	100.00%	100.00%	100.00%	100.00%
	5	93.55%	93.67%	93.80%	100.00%	94.23%	94.46%	95.26%	100.00%
	6	94.36%	94.34%	94.58%	100.00%	95.05%	94.74%	95.09%	100.00%
	7	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
	8	89.94%	89.96%	90.66%	100.00%	91.05%	90.55%	93.23%	100.00%
	9	96.65%	96.85%	96.38%	100.00%	96.98%	97.56%	96.78%	100.00%
	10	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
	11	81.53%	80.42%	84.18%	100.00%	84.17%	83.11%	89.09%	100.00%
	12	89.08%	88.74%	89.04%	100.00%	90.29%	89.59%	90.91%	100.00%
	13	100.00%	100.00%	99.99%	100.00%	100.00%	100.00%	100.00%	100.00%
	14	90.91%	90.66%	91.45%	100.00%	92.52%	91.80%	94.47%	100.00%
	15	90.95%	91.80%	90.13%	100.00%	92.42%	93.27%	91.81%	100.00%
	16	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
	17	85.39%	83.72%	85.80%	100.00%	87.20%	84.67%	87.17%	100.00%
	18	85.95%	85.09%	88.36%	100.00%	88.48%	87.81%	89.75%	100.00%
	19	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
	20	64.50%	63.39%	67.84%	100.00%	74.13%	72.57%	72.91%	100.00%
	21	71.00%	70.72%	72.77%	100.00%	74.88%	72.36%	76.29%	100.00%
	22	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
	23	78.73%	79.30%	79.32%	100.00%	80.42%	85.33%	81.00%	100.00%
	24	73.66%	76.95%	75.01%	100.00%	76.82%	84.02%	77.01%	100.00%
	25	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
	26	69.73%	70.09%	70.19%	100.00%	74.26%	74.46%	75.39%	100.00%
	27	71.34%	73.20%	74.22%	100.00%	73.83%	77.12%	78.00%	100.00%

Figure 14. Average/best fitness over five runs of various genetic algorithm configurations using a logarithmic fitness estimator when maximum fitness of a population and a ranking of individuals within the population is known.

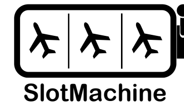


	Case #	Sigmoid estimator / average fitness over 5 runs				Sigmoid estimator / best fitness over 5 runs			
Configuration		J1	J2	J3	Hungarian	J1	J2	J3	Hungarian
Avg % of Hungarian		89.53%	88.86%	89.74%	100.00%	90.92%	90.22%	91.14%	100.00%
Rank		3	4	2	1	3	4	2	1
SUM		24.17	23.99	24.23	27.00	24.55	24.36	24.61	27.00
	1	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
	2	85.58%	85.95%	87.39%	100.00%	87.71%	87.39%	89.29%	100.00%
	3	98.89%	99.07%	99.05%	100.00%	98.98%	99.15%	99.16%	100.00%
	4	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
	5	93.82%	92.92%	94.31%	100.00%	94.47%	93.76%	95.46%	100.00%
	6	94.46%	94.28%	94.25%	100.00%	94.66%	94.47%	94.58%	100.00%
	7	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
	8	90.37%	90.57%	91.07%	100.00%	91.41%	92.20%	92.10%	100.00%
	9	96.39%	96.38%	96.49%	100.00%	96.93%	96.84%	96.78%	100.00%
	10	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
	11	82.06%	80.37%	84.02%	100.00%	85.00%	83.26%	85.53%	100.00%
	12	87.59%	87.20%	88.53%	100.00%	90.76%	89.68%	90.28%	100.00%
	13	100.00%	100.00%	99.99%	100.00%	100.00%	100.00%	100.00%	100.00%
	14	90.39%	89.75%	91.63%	100.00%	91.81%	90.94%	92.39%	100.00%
	15	91.34%	89.44%	89.42%	100.00%	92.16%	92.30%	91.27%	100.00%
	16	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
	17	84.31%	84.09%	86.30%	100.00%	86.61%	86.87%	87.71%	100.00%
	18	87.37%	86.28%	87.23%	100.00%	88.53%	86.44%	89.67%	100.00%
	19	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
	20	65.64%	61.47%	63.22%	100.00%	72.71%	65.23%	68.98%	100.00%
	21	71.40%	67.61%	72.05%	100.00%	74.62%	69.85%	76.83%	100.00%
	22	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
	23	79.58%	76.96%	79.77%	100.00%	81.09%	81.20%	84.71%	100.00%
	24	74.54%	74.09%	74.08%	100.00%	77.03%	76.72%	75.98%	100.00%
	25	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
	26	70.84%	71.57%	71.28%	100.00%	73.80%	73.40%	74.34%	100.00%
	27	72.81%	71.11%	72.98%	100.00%	76.62%	76.33%	75.75%	100.00%

Figure 15. Average/best fitness over five runs of various genetic algorithm configurations using a sigmoid fitness estimator when maximum fitness of a population and a ranking of individuals within the population is known.

The fitness estimator aims to derive the fitness of the individual solutions in a population given the ranks of the solutions and the population's maximum fitness value. We conclude that using a fitness estimator does not severely impact the performance of the genetic algorithm, demonstrating the general feasibility of the optimization process involving Heuristic Optimizer and Privacy Engine as proposed by the SlotMachine project.

We implemented different estimators (see Chapter 5) and investigated how well the different estimators represent the real distribution of the fitness in a population. In the following we show the evolution of the actual and estimated fitness values of the configuration J1 in Case 9 and Case 20, respectively, used with the different estimators. The cases were selected for the following reasons. Case 9 with moderate concentration and wide margins is comparatively unproblematic regarding optimization. Case 20 is more challenging, with extreme concentration and narrow margins.

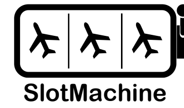


Fitness distribution in population (no estimator)

fitness value
position in population
generation 14 of 137
generation 41 of 137
generation 82 of 137
generation 130 of 137
evaluated fitness

Figure 16. Evaluated fitness distribution in different generations of J1 applied to Case 9 using no estimator

Figure 16 and Figure 17 show the evolution of the fitness distribution in a population for Case 9 and Case 20, respectively, using no fitness estimator, i.e., actual fitness values are used by the genetic algorithm. Each diagram plots the actual fitness value for the ith individual of a population ranked by fitness value. We conducted the runs with no estimator to have a comparison for the distributions returned by the estimators.



Fitness distribution in population (no estimator)

fitness value
200000
-200000
-600000
-1000000
evaluated fitness
0 100 200 300 400 500
position in population
generation 15 of 147

fitness value
200000
-200000
-600000
-1000000
evaluated fitness
0 100 200 300 400 500
position in population
generation 44 of 147

fitness value
200000
-200000
-600000
-1000000
evaluated fitness
0 100 200 300 400 500
position in population
generation 88 of 147

fitness value
200000
-200000
-600000
-1000000
evaluated fitness
0 100 200 300 400 500
position in population
generation 140 of 147

Figure 17. Evaluated fitness distribution in different generations of J1 applied to Case 20 using no estimator

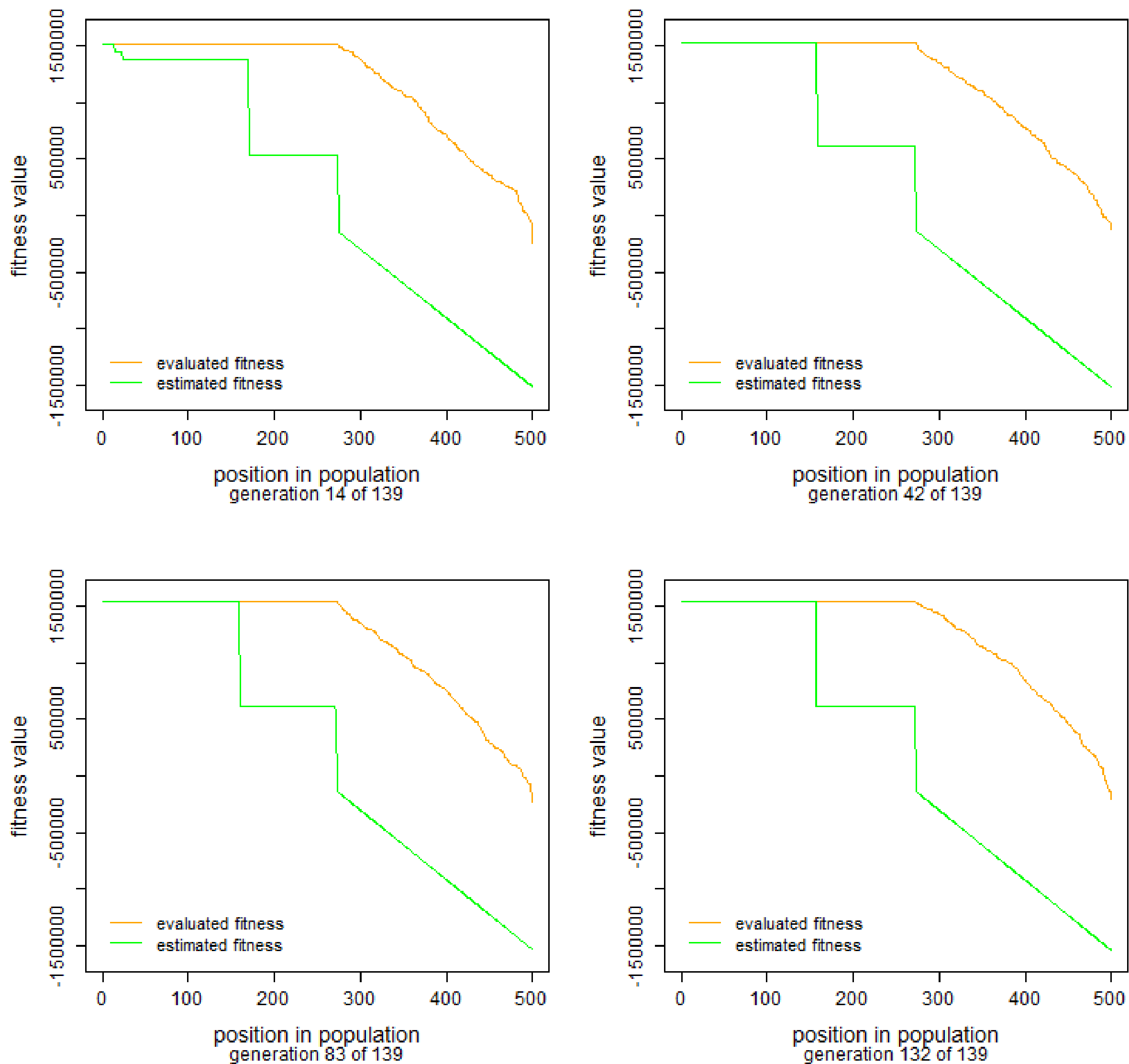
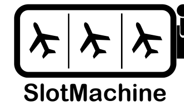


Figure 18. Estimated and evaluated fitness distributions in different generations of J1 applied to Case 9 using linear estimator

Figure 18 and Figure 19 show the evolution of the fitness distribution in a population for Case 9 and Case 20, respectively, using the linear fitness estimator. Since there are duplicate solutions in the population, the linear estimation is not perfectly linear but in the first half of the population decreases in steps. With Case 20, when the initial fitness is rather low, the estimator is a particularly bad reflection of the real distribution but also in general we note that the linear estimator is generallynot very accurate.



Fitness distribution in population (linear estimator)

Figure 19. Estimated and evaluated fitness distributions in different generations of J1 applied to Case 20 using linear estimator

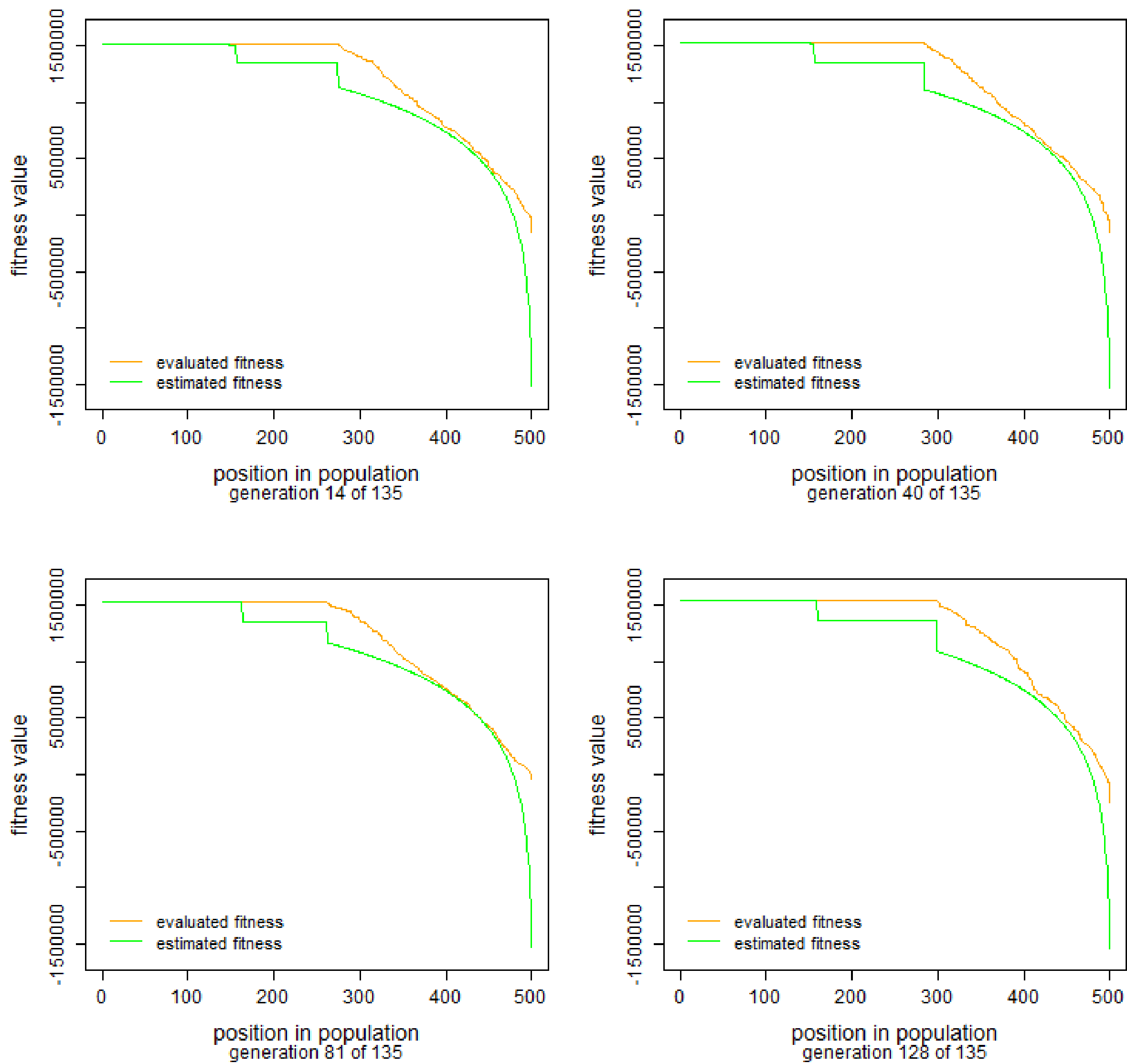
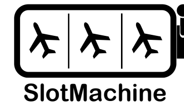


Figure 20. Estimated and evaluated fitness distributions in different generations of J1 applied to Case 9 using logarithmic estimator

Figure 20 and Figure 21 show the evolution of the fitness distribution in a population for Case 9 and Case 20, respectively, using the logarithmic fitness estimator. The logarithmic estimator seems to capture the real distribution best although the estimation of the minimum value was either too low (in the "good" Case 9) or too high (in the "bad" Case 20).



Fitness distribution in population (logarithmic estimator)

Figure 21. Estimated and evaluated fitness distributions in different generations of J1 applied to Case 20 using logarithmic estimator

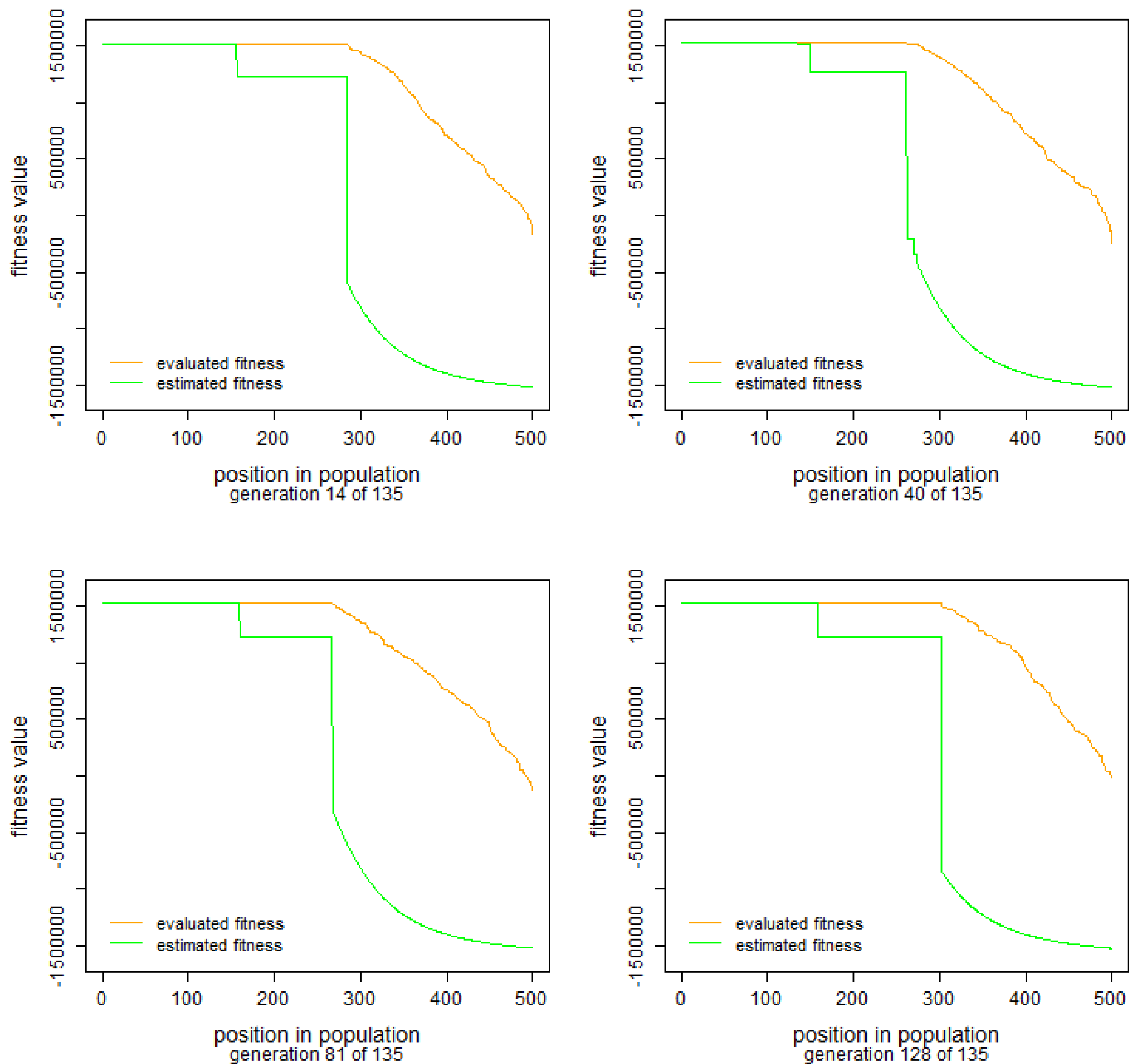
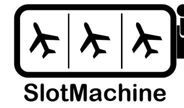


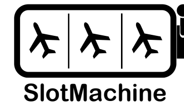
Figure 22. Estimated and evaluated fitness distributions in different generations of J1 applied to Case 9 using sigmoid estimator

Figure 22 and Figure 23 show the evolution of the fitness distribution in a population for Case 9 and Case 20, respectively, using the sigmoid fitness estimator. The sigmoid estimator does not seem to accurately capture the real distribution of the fitness values.



Fitness distribution in population (sigmoid estimator)

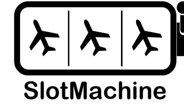
Figure 23. Estimated and evaluated fitness distributions in different generations of J1 applied to Case 20 using sigmoid estimator



7 Conclusions

The Heuristic Optimizer component of the SlotMachine system drives the optimization of flight lists. The privacy-preserving nature of the optimization process in the SlotMachine system means that a simple deterministic optimization algorithm cannot be employed. Rather, the Heuristic Optimizer employs an evolutionary optimization algorithm for finding solutions to the optimization problem and invokes the Privacy Engine for evaluation of the solutions. Nevertheless, even evolutionary algorithms cannot be readily employed. In order to preserve the confidentiality of the airspace users' preferences, the Privacy Engine does not return fitness values for the entire population of flight lists found by the Heuristic Optimizer but only ranks the solutions and returns the maximum fitness within the population. This particular approach to evaluation of the population also requires a specific fitness function that use an estimator to derive fitness values for the individual flight lists in the population during the optimization runs.

When implementing the Heuristic Optimizer we aimed for flexibility. Hence, the Heuristic Optimizer is a framework that allows to easily plug in different implementations of evolutionary algorithms. A genetic algorithm module has been implemented and can be used for privacy-preserving optimization runs in conjunction with the SlotMachine system. Non-privacy-preserving modules using other types of evolutionary algorithms and a deterministic algorithm, respectively, have also been implemented during an initial exploration phase (see D4.1 – Report on State of the Art of Relevant Concepts [9]). Those additional modules cannot (yet) be readily used together with the Privacy Engine although some of the employed local search algorithms could likely be adapted for use with the Privacy Engine. If further experimentation and validation activities in the SlotMachine project show that the current genetic algorithm implementation is insufficient an improved evolutionary algorithm module could be easily integrated into the general Heuristic Optimizer framework.



8 References

[1] SESAR, „D4.2 - Specification of Evolutionary Algorithm,“ SlotMachine, 890456, 2021.

[2] D. Simon, Evolutionary Optimization Algorithms: Biologically Inspired and Population-Based Approaches to Computer Intelligence, Wiley, 2013.

[3] M. Affenzeller, S. Winkler, S. Wagner und A. Beham, Genetic Algorithms and Genetic Programming: Modern Concepts and Practical Applications, CRC Press, 2009.

[4] F. Wilhelmstötter, „Jenetics Library User's Manual 6.3,“ 2021. [Online]. Available: https://jenetics.io/manual/manual-6.3.0.pdf. [Zugriff am 14 January 2022].

[5] SESAR, „D2.1 - Requirements Specification,“ SlotMachine, 890456, 2021.

[6] SESAR, „D2.2 - System Design Document,“ SlotMachine, 890456, 2021.

[7] SESAR, „D2.3 - Business Concepts,“ SlotMachine, 890456, 2021.

[8] SESAR, „D3.2 - Specification of the PrivacyEngine Component,“ SlotMachine, 890456, 2021.

[9] SESAR, „D4.1 - Report on State of the Art of Relevant Concepts,“ SlotMachine, 890456, 2021.

[10] SESAR, „D5.1 - SlotMachine Platform Demonstrator,“ SlotMachine, 890456, 2021.

[11] V. Yadaiah und V. V. Haragopal, „A new approach of solving single objective unbalanced assignment problem,“ *American Journal of Operations Research,* Bd. 6, Nr. 1, 2016.

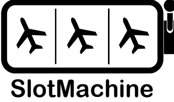
[12] H. W. Kuhn, „The Hungarian Method for the assignment problem,“ *Naval Research Logistics Quarterly,* Bd. 2, Nr. 1-2, pp. 83-97, 1955.

[13] H. W. Kuhn, „Variants of the Hungarian method for assignment problems,“ *Naval Research Logistics Quarterly,* Bd. 3, Nr. 4, pp. 253-258, 1956.

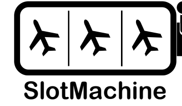
[14] J. Munkres, „Algorithms for the assignment and transportation problems,“ *Journal of the Society for Industrial and Applied Mathematics,* Bd. 5, Nr. 1, pp. 32-38, 1957.

[15] J. Edmonds und R. M. Karp, „Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems,“ *Journal of the ACM,* Bd. 19, Nr. 2, pp. 248-264, 1972.

[16] E. Gamma, R. Helm, R. Johnson und J. Vlissides, „Design Patterns: Elements of Reusable Object-Oriented Software,“ Addison Wesley, 1994.


SlotMachine

SESAR
JOINT UNDERTAKING

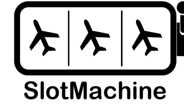


Appendix A Third-Party Libraries

Table 5 lists the directly used third-party libraries. Those libraries use also additional libraries, which are not listed here. We refer to the documentation of those libraries for any additional dependencies.

Table 5. List of third-party libraries

Library	Version	License	Purpose
Spring Boot	2.4.3	Apache License 2.0	The Spring Boot framework is used to realize the REST interface of the Heuristic Optimizer.
Springfox	2.6.1	Apache License 2.0	The Springfox framework generates a Swagger documentation for the REST interface.
Jenetics	6.3.0	Apache License 2.0	The Jenetics framework facilitates the implementation of genetic algorithms.
OptaPlanner	8.14.0.Final	Apache License 2.0	The OptaPlanner framework provides implementations of common local search algorithms.
Log4J2	2.17.0	Apache License 2.0	The Log4J2 framework provides logging capabilities.


SlotMachine

SESAR
JOINT UNDERTAKING

AIT
AUSTRIAN INSTITUTE OF TECHNOLOGY
TOMORROW TODAY

FREQUENTIS
FOR A SAFER WORLD

JKU
JOHANNES KEPLER
UNIVERSITÄT LINZ

SWISS