Appendix A

Problem Definition

Many problems in the world that we wish to solve are intractable, that is, it will take more than polynomial time to find the optimal solution. In most situations a sub-optimal solution can be used instead and a range of techniques have been developed to provide a near optimal solution in minimal time.

Genetic algorithms are one such technique. They begin by generating a population of potential solutions to the problem, encoded on genes. Over a number of generations new solutions are bred by crossing and mutating. Solutions compete for space within the population and those who cannot compete, die out. The process of selection, similar to natural selection, will find a near optimal solution after a number of generations.

There are five characteristic components in every genetic algorithm:

- a genetic representation for solutions to the problem
- a way to create an initial population of potential solutions
- an evaluation function that plays the role of the environment, rating solutions in terms of their fitness
- genetic operators that alter the composition of children during reproduction
- values for various parameters that the genetic algorithm uses (population size, probability of applying genetic operators)

Each component can be represented in a number of different ways and used in many algorithms. MUTANTS will provide a library of these genetic algorithm components. Reuse of components speeds the creation of an algorithm giving the chance for experimentation to produce faster or better algorithms.

Appendix B

Statement of Requirements

I. Customer Goals
   A. The aim is to provide a library of components to simplify the construction of genetic algorithms.
   B. Components should be written in Ada 95.
   C. As a minimum the toolkit should be able to implement the classic genetic algorithm with:
      1. Bit string genetic representation,
      2. Random binary initialisation,
      3. Binary mutation and single point crossover genetic operator,
      4. Generation replacement model.
   D. In addition other techniques may be included, such as:
      1. Vector, ordered list, number genetic representation,
      2. Distributed and random search initialisation,
      3. Other mutation, multi-point crossover, and inversion genetic operators,
      4. Steady-state, elitist, and without-duplicate models.
   E. Other facilities which the toolkit should support:
      1. User provided fitness values for potential solutions,
      2. Family trees to relate all previous solutions,
      3. Decision support which helps the user explore the solution space,
      4. Saving of genetic algorithm state to be recreated later.
   F. The toolkit should be extensible so that components specific to a particular genetic algorithm can be added by the programmer.
   G. Components should be contained so the use of one does not require the use of another.
   H. Advanced techniques such as meta-genetic algorithms, adaptive genetics, and interpolative genetics should be implementible using the toolkit but need not be implemented.
   I. The toolkit should be proved by creating two dissimilar genetic algorithms using its components.

II. Technical Goals
   A. Functional Requirements
      1. Toolkit components should be divided into three classes
         a) core toolkit - components required for classic genetic

        algorithm with: binary string representation, random binary initialisation, simple mutation and single point crossover, generation replacement model
    - b) auxiliary toolkit - components to extend functionality of genetic algorithms such as decision support
    - c) extended toolkit - components for other genetic algorithm techniques
  2. The core toolkit has highest priority, then the auxiliary toolkit and finally the extended toolkit.
  3. Component organisation must:
    - a) allow for the large number of possible genetic algorithms,
    - b) be extensible with user defined components.
    - c) not restrict the user within components
    - d) be savable
  4. Two dissimilar genetic algorithms should be created to show the power and functionality of the toolkit.
  5. A manual to explain the use of the toolkit should be made.

B. Non-functional Requirements
  1. All components should be written in Ada 95.
  2. Typical population sizes in genetic algorithms do not exceed a few hundred but storing individuals permanently will increase this by several orders of magnitude.
  3. Deadlines:
    - a) The core toolkit and the first demonstration genetic algorithm should be completed by the end of week 10,
    - b) The second demonstration genetic algorithm and the rest of the project should be delivered by the end of week 20.

Appendix C

External Specification

The toolkit will be composed of a collection of components organised in a number of component hierarchies which describe their relationships. The taxonomic hierarchy groups components with similar tasks together, for example apples are under fruit. The functional hierarchy shows a components use of other components, for example apples are under teeth. The containment hierarchy shows a components storage of other components, for example pips are under apples. All components are also grouped as part of the core, auxiliary, or extended toolkit. Components are presented here according to the taxonomic hierarchy with other hierarchies described in the text.

To create a genetic algorithm with this toolkit the programmer has to combine a number of predefined components with a few that have been hand coded. A simple example of constructing a genetic algorithm using the toolkit follows with component names emphasised. A description of each component can be found later in the specification.

Classic Genetic Algorithm

A genetic algorithm is required to optimise a simple function.

$$f(x) = x.\sin(10\pi.x) + 1.0$$

The problem is to find x from the range [-1, 2] which maximises the function f, ie to find x_0, such that

$$f(x_0) \geq f(x) \text{ for all } x \in [-1, 2]$$

Two important decisions have to made about the genetic algorithm: how to represent potential solutions, and which model will drive the genetic algorithm. The use of classic genetic algorithm techniques dictates our choices here. The *Generational Model* will define the order of actions within our algorithm and solutions will be represented using a *Bit Vector*. To store a solution to an accuracy of 6 decimal places will require 22 bits for each vector.

To decide which solutions are better an *Evaluation* function must take the bit vector representation and return a number

$$E = -1.0 + v.3/(2^{22} - 1)$$

where v is the bit vector interpreted as a positional code integer, this can be done using *Function Binary Bit Vector Evaluation.*

The initial *Population* of solutions will be created using *Bit Vector Random Initialisation* and subsequent solutions will be bred using two *Reproduction* operators, *Bit Random Single Vector Mutation* and *One Point Vector Crossover*. Operator distribution is 22% and 25% respectively with the remainder made up using *Clone.*

The Toolkits

The *core toolkit* consists of those components required to implement the classic genetic algorithm "The Blues" as shown in the report.

> Generational Model, Count Ending, Bit Representation, Vector Representation, Individual, Population, Bit Random Initialise, Vector Random Initialise, Evaluation, Unity Fitness, Roulette Wheel Selector, Clone Reproduction, Xor Reproduction, Position-based Ordered Mutation Reproduction, Order-based Ordered Crossover Reproduction, Random

The *auxiliary toolkit* consists of those components which extend functionality for genetic algorithms such as decision support.

> History

The *extended toolkit* consists of all components suitable for other genetic algorithms.

Component Overview

The *Model*, or breeding technique, component controls the general activity of the genetic algorithm. A genetic algorithm consists of a single Model and subsidiary components controlled by it.

The *Ending* component controls the length of time spent on the genetic algorithm.

The *Representation* component encodes a possible solutions. Choosing which Representation is important to the design of the entire genetic algorithm.

The *Individual* component is a particular instance of a possible solution. Each Individual is associated with a Representation and a unique identification which separates it from all other Individuals, even those with identical Representations.

The *Population* component is a bag of Individuals or a bag of Populations. Operations include all standard bag operations.

The *Evaluation* component converts a Representation to a numerical representation of how good a solution it encodes.

The *Fitness* component converts and Individuals Evaluation into its fitness compared to the rest of the Population.

The *Selector* component chooses an Individual from a Population, perhaps using information about that Individual and the rest of the Population.

The *Reproduction* component creates N new Individuals for the destination Population from the source Population.

The *History* component can store an association of Individuals, their parents and Reproduction components or Initialisers, and the Random settings.

The *Random* component generates pseudo random numbers starting from either a given seed or the seed is taken from the time.

Model Component

The Model, or breeding technique, component controls the general activity of the genetic algorithm. A genetic algorithm consists of a single Model and subsidiary components controlled by it. Although the toolkit provides a range of Model components it is impossible to provide the variety which may be required. Instead unusual Models can be hand coded making use of subsidiary components from the toolkit.

Functional: Ending, Individual, Population, Initialise, Evaluation, Fitness, Selector, Reproduction, History, Random

Containment: Ending, Population, Initialise, Evaluation, Fitness, Selector, Reproduction, History, Random

Generational
The first Population is Initialised then a new Population of the same size is created using the Reproduction operator. The current Population is then replaced and the new Population used for further breeding.
Generational Elitism
Generational but the first Individual of the new Population is a clone of the best Individual of the previous Population.
Tournament
Generational but new Individuals are together with current Individuals. N Individuals are selected and the best is placed in the next generation. This is repeated until the next generation is full.
Steady State
The first Population is Initialised then N new Individuals are created using the Reproduction operator. These replace N individuals selected from the old population.
Steady State Without Duplicates
Steady State but new Individuals must not have identical Representations to current Individuals.
Preselection
Steady State but Individuals only replace inferior parents.

Ending Component

The Ending component controls the length of time spent on the genetic algorithm.

Functional:

Containment: Ending

And
The Model continues until both given ending conditions are true.
Or
The Model continues until either of two ending conditions are true.
Count

The Model is allowed N cycles before halting.

Time

The Model is allowed N seconds before halting.

Evaluation

The Model continues until a given Evaluation is reached.

Convergence

The Model continues until the population converges and no further change in Evaluation is seen.

Representation Component

The Representation component encodes a possible solutions. Choosing which Representation is important to the design of the entire genetic algorithm.

Functional:

Containment: Representation

Bit

A single bit, 0 or 1.

Number

A single number with value between upper and lower bounds.

Vector

A sequence of elements with identical Representations.

Matrix

A matrix of elements with identical Representations.

Ordered

A particular permutation of elements draw from a particular Representation.

Tag

A integer position paired with another Representation, can be used to make a tagged vector for use with the Inversion operator.

Individual Component

The Individual component is a particular instance of a possible solution. Each Individual is associated with a Representation and a unique identification which separates it from all other Individuals, even those with identical Representations.

Functional: Representation

Containment: Representation

Population Component

The Population component is a bag of Individuals or a bag of Populations. Operations include all standard bag operations.

Functional: Representation, Individual, Evaluation

Containment: Individual, Population

Initialise Component

The Initialise component is used to create a number of new Individuals for a given Population. The Initialiser must be tailored to particular Representations.

Functional: Representation, Population, Evaluation, Selector, Reproduction, Random

Containment: Initialise, Reproduction

Random
- Bit
  - Assigned 0 or 1 with equal probability.
- Number
  - Assigned a number between upper and lower boundary with equal probability.
- Vector
  - Initialise each element separately.
- Matrix
  - Initialise each element separately.
- Ordered
  - Assign one of the possible permutation of elements with equal probability.
- Tag Vector
  - Initialise each Tag separately and assign the correct integer position.

Search

Initialise N Individuals at a time and pick the one with the highest Evaluation. Selection between Individuals with equal fitness will be made randomly.

Distributed

Generated Individuals are as different as possible from each other. This technique must be tailored for each Representation.

Heuristic

Individuals with particular characteristics are generated using heuristics particular the problem. This technique must be tailored to each Representation and each problem.

Evaluation Component

The Evaluation component converts a Representation to a numerical representation of how good a solution it encodes. The evaluation is dependent on both the Representation and the problem. In most situations the Evaluation component will be hand coded by the programmer.

Functional: Representation, Individual

Containment: Evaluation

- Vector
  - Bit
    - Binary
      - Evaluation is equal to the positional code interpretation of the bit vector.
    - Gray
      - Evaluation is equal to the gray code interpretation of the bit vector.
- Function
  - Apply a known function to the result of another Evaluation.

Fitness Component

The Fitness component converts and Individuals Evaluation into its fitness compared to the rest of the Population.

Functional: Individual, Population

Containment:

Unity

Fitness is equal to evaluation.

Windowing

Fitness is equal to the amount an Individual's Evaluation exceeds the minimum evaluation within the Population minus some known guard value.

Linear Normalisation

Order Individuals by decreasing Evaluation. The best Individual is given a known fitness and thereafter the fitness is decreased by a constant amount.

Linear Scaling

Fitness is equal to (a * evaluation + b) where a and b are normally selected so that the average fitness is mapped to itself and the best fitness is increased by some desired multiple.

Sigma Truncation

Fitness is equal to (evaluation + average evaluation - c * s) where c is chosen from around 1 to 5 and s is the Population's standard deviation. Negative fitness values are set to zero.

Power Law Scaling

Fitness is equal to (evaluation^{k}) where k is a problem dependent value close to 1.

Selector Component

The Selector component chooses an Individual from a Population, perhaps using information about that Individual and the rest of the Population.

Functional: Individual, Population, Fitness, Random

Containment:

Best

Individual with the highest fitness within the Population is selected. Selection between Individuals with equal fitness will be made randomly.

Worst

Individual with the lowest fitness within the Population is selected. Selection between Individuals with equal fitness will be made randomly.

Random

Each Individual has an equal probability of being selected.

Roulette Wheel

Each Individual has a probability of being selected proportional to its fitness divided by the average fitness of the Population.

Anti Roulette Wheel

Each Individual has a probability of being selected inversely proportional to its fitness divided by the average fitness of the Population.

Stochastic

The first Individual is chosen using a roulette wheel and the subsequent (N - 1) Individuals taken evenly from the entire population, then the cycle repeats.

Expected Value

Associated with each Individual is a count equal to that Individual's fitness divided by the average fitness of the Population. Whenever an Individual is selected to reproduce, using another Selector component, its count is decremented by one. When a count falls below zero the Individual is no long available for further selection.

Similarity

The Individual selected is that closest to the given Individual. Selection between equally similar Individuals will be made randomly. This technique must be tailored to each Representation and each problem.

Reproduction Component

The Reproduction component creates N new Individuals for the destination Population from the source Population. Reproduction components must be tailored to a each Representation and each Problem.

Functional: Representation, Individual, Population, Selector, Random

Containment: Selector, Reproduction

Clone

Individual's Representation is an exact copy of the parent's Representation.

Xor

Individuals are created with either one Reproduction component or another but not both using the given probability distribution.

And

Individuals are created with one Reproduction followed by another. The second Reproduction component may require several Individuals to be

  generated by the first component.
- Or
  - Individuals are created with either of two Reproduction components or both using the given probabilities. The second Reproduction component may require several Individuals to be generated by the first component.
- Inversion
  - Tag Vector
    - Biased
      - Individual's Representation is the same as the parent's Representation except that the order of a sequence of elements between two random points has been reversed.
    - Unbiased
      - Individual's Representation is the same as the parent's Representation except that the order of a sequence of element between a random point and a randomly selected end has been reversed.
- Mutation
  - Bit
    - Inverts value.
  - Number
    - Random
      - Assign a number between upper and lower boundary with equal probability.
    - Creep
      - Modify the number by either adding or subtracting a know value.
  - Vector
    - Single
      - Each element has an equal probability of being chosen and mutated.
    - Deletion
      - A randomly chosen element is deleted. Only suitable for variable length vectors.
    - Addition
      - Duplication
        - A randomly chosen element is duplicated with the duplicate placed next to the original. Only suitable for variable length vectors.
      - Initialised
        - A newly initialised element is inserted into the list. Only

suitable for variable length vectors.

Related

A pair of adjacent elements is randomly chosen and a new element related to each is inserted between them. This must be tailored to each Representation and each problem. Only suitable for variable length vectors.

Tag Vector

Cut

Parent vector is cut at a random point. Only suitable for messy genetic algorithms.

Single

Each element has an equal probability of being chosen and the non-position value mutated.

Ordered

Order-based

Two randomly selected elements are swapped.

Position-based

One randomly selected element is placed before another.

Scramble Sublist

Select a sublist of size N and randomly permute it.

Random Hill Climb

N new Representations are created using a Mutation component and the one with the best Evaluation is chosen. Selection between Individuals with equal evaluation will be made randomly.

Crossover

Number

Average

Assign the average of two parent numbers.

Vector

One Point

A crosspoint is selected at random and the tail of each parent is swapped with the other parent.

Two Point

Two crosspoints are selected at random and the elements between are swapped with the other parent. Initial and final elements of the vector are considered adjacent.

Multi Point

N pairs of crosspoints are selected at random and the elements between are swapped with the other parent. Initial and final elements of the vector are considered adjacent.

- Segmented
    - Multi Point Crossover in which the number of segments can vary. For each element there is a probability that the current segment will end and a new one begin.
- Uniform
    - Each element is taken from either parent with equal probability.
- Shuffle
    - Randomly permute both parent vectors, perform a Crossover, and reverse permute them.
- Traverse
    - Apply given Crossover to each pair of elements in parent vectors.

Tag Vector

- Splice
    - Parent's representations are joined end to end. Only suitable for messy genetic algorithms.

Ordered

- Position-based
    - A set of positions is randomly selected and the positions of elements selected in one parent is imposed on the corresponding elements in the other parent.
- Order-based
    - A set of positions is randomly selected and the order of elements in the selected positions in one parent is imposed on the corresponding elements in the other parent.

Repair

The parents representation is corrected so it no longer violates constraints made on the solution.

History Component

The History component can store an association of Individuals, their parents and Reproduction components or Initialisers, and the Random settings.

Functional: Individual, Reproduction, Initialise, Random

Containment: Individual

Random Component

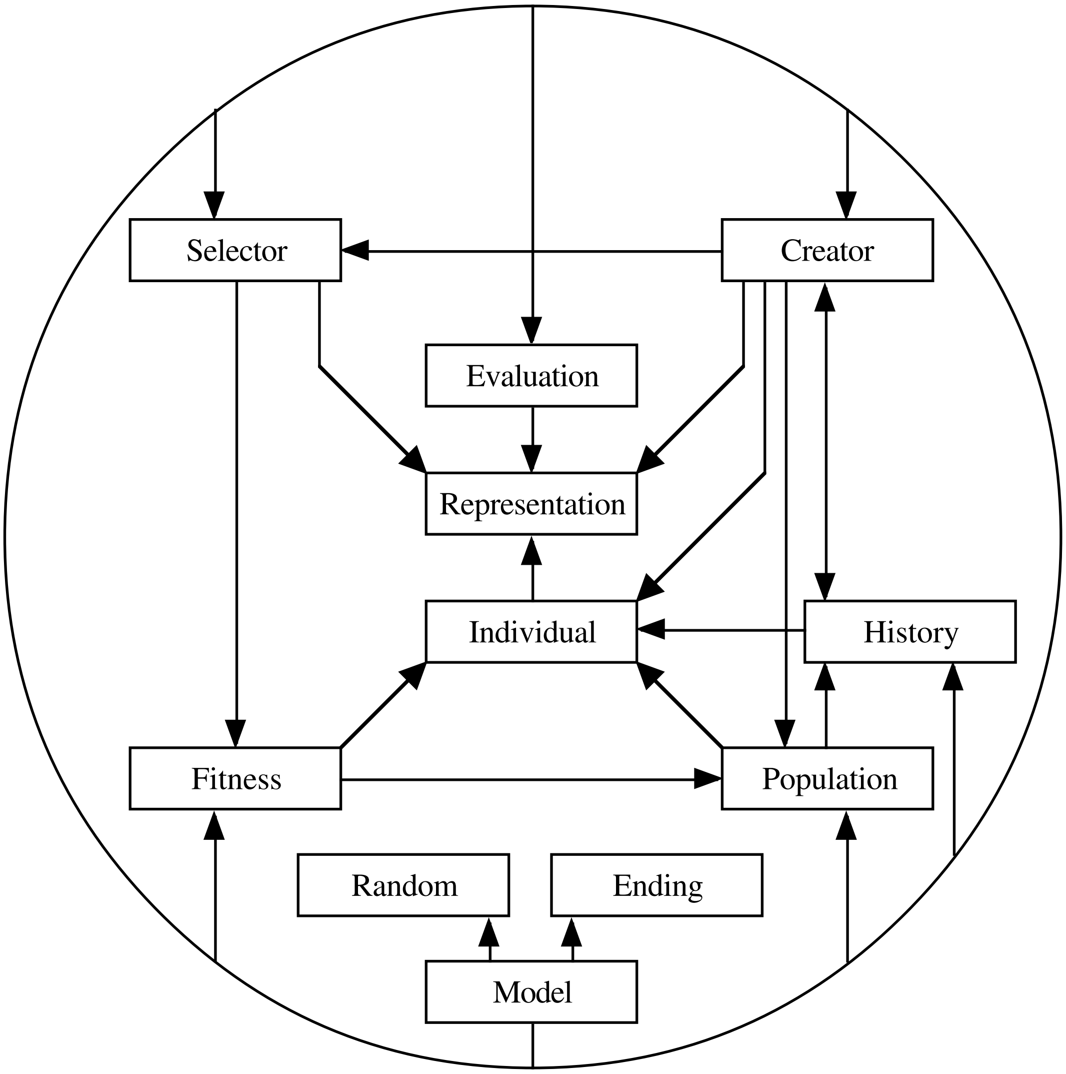
The Random component generates pseudo random numbers starting from either a given seed or the seed is taken from the time.

Functional:

Containment:

Appendix D

Class Design



Class design information is formatted as -

method: arguments > result:

Object creators are refered to as *ctor* methods. A backslash before an arguement indicates the value is passed by value rather than by reference. Arguements enclosed in squared brackets symbolise an array of elements rather than a single value. An underlined method indicates that the child classes may override this method and an underlined reference arguement means the value of the object may change.

Model Class

class Model:
ctor: Creator, \size > Model
ctor: Creator, \size, Random > Model
ctor: Istream > Model
print: Model, Ostream
run: Model, Evaluation, Fitness, Selector, Creator, Ending
run: Model, Evaluation, Fitness, Selector, Creator, Ending, History
population: Model > Population

class Generational_Model:
print: Model, Ostream
run: Generational_Model, Evaluation, Fitness, Selector, Creator, Ending
run: Generational_Model, Evaluation, Fitness, Selector, Creator, Ending, History
population: Generational_Model > Population

class GenerationalElitism_Model:
print: Model, Ostream
run: GenerationalElitism_Model, Evaluation, Fitness, Selector, Creator, Ending
run: GenerationalElitism_Model, Evaluation, Fitness, Selector, Creator, Ending, History
population: GenerationalElitism_Model > Population

class GenerationalTournament_Model:
print: Model, Ostream
run: GenerationalTournament_Model, Evaluation, Fitness, Selector, Creator, Ending, \tournament_size
run: GenerationalTournament_Model, Evaluation, Fitness, Selector, Creator, Ending, History, \tournament_size
population: GenerationalTournament_Model > Population

class SteadyState_Model:
print: Model, Ostream
run: SteadyState_Model, Evaluation, Fitness, Selector, Creator, Ending, \brood_size
run: SteadyState_Model, Evaluation, Fitness, Selector, Creator, Ending, History, \brood_size
population: SteadyState_Model > Population

class SteadyStateNoDuplicates_Model:

print: Model, Ostream
run: SteadyStateNoDuplicates_Model, Evaluation, Fitness, Selector, Creator, Ending, \brood_size
run: SteadyStateNoDuplicates_Model, Evaluation, Fitness, Selector, Creator, Ending, History, \brood_size
population: SteadyStateNoDuplicates_Model > Population

class SteadyStatePreselection_Model:
print: Model, Ostream
run: SteadyStatePreselection_Model, Evaluation, Fitness, Selector, Creator, Ending, \brood_size
run: SteadyStatePreselection_Model, Evaluation, Fitness, Selector, Creator, Ending, History, \brood_size
population: SteadyStatePreselection_Model > Population

Ending Class

class Ending:
ctor: > Ending
print: Ending, Ostream
end: Ending > \end

class And_Ending:
ctor: Ending, Ending > And_Ending
print: And_Ending, Ostream
end: And_Ending > \end

class Or_Ending:
ctor: Ending, Ending > Or_Ending
print: Or_Ending, Ostream
end: Or_Ending > \end

class Count_Ending:
ctor: \count, \decrement > Count_Ending
print: Count_Ending, Ostream
end: Count_Ending > \end

class Time_Ending:
ctor: \duration > Time_Ending
print: Time_Ending, Ostream

end: Time_Ending > \end

class Evaluation_Ending:
ctor: Model, Evaluation > Evaluation_Ending
print: Evaluation_Ending, Ostream
end: Evaluation_Ending > \end

class Convergence_Ending:
ctor: Model, Evaluation, \improvement, \period > Convergence_Ending
print: Convergence_Ending, Ostream
end: Convergence_Ending > \end

Representation Class

class Representation:
ctor: Istream > Representation
print: Representation, Ostream

class Bit_Representation:
ctor: Bit > Bit_Representation
ctor: Istream > Bit_Representation
print: Bit_Representation, Ostream
get: Bit_Representation > Bit
set: Bit_Representation, Bit

class Number_Representation:
ctor: Number > Number_Representation
ctor: Istream > Number_Representation
print: Number_Representation, Ostream
get: Number_Representation > Number
set: Number_Representation, Number

class Vector_Representation:
ctor: [Representation] > Vector_Representation
ctor: Istream > Vector_Representation
print: Vector_Representation, Ostream
get: Vector_Representation, \index > Representation
set: Vector_Representation, \index, Representation
length: Vector_Representation > \length

class Tagged_Vector_Representation:
ctor: [Representation] > Tagged_Vector_Representation
ctor: Istream > Tagged_Vector_Representation
print: Tagged_Vector_Representation, Ostream
get: Tagged_Vector_Representation, \index > Representation
set: Tagged_Vector_Representation, \index, Representation
tagged_get: Tagged_Vector_Representation, \index > Representation
tagged_set: Tagged_Vector_Representation, \index, Representation
length: Tagged_Vector_Representation > \length

class Matrix_Representation:
ctor: [[Representation]] > Matrix_Representation
ctor: Istream > Matrix_Representation
print: Matrix_Representation, Ostream
get: Matrix_Representation, \x, \y > Representation
set: Matrix_Representation, \x, \y, Representation
width: Matrix_Representation > \width
height: Matrix_Representation > \height

class Ordered_Representation:
ctor: [Representation] > Ordered_Representation
ctor: Istream > Ordered_Representation
print: Ordered_Representation, Ostream
get: Ordered_Representation, \index > Representation
set: Ordered_Representation, \index, Representation
length: Ordered_Representation > \length

Individual Class

class Individual:
ctor: Representation > Individual
ctor: Istream > Individual
print: Individual, Ostream
representation: Individual > Representation
evaluation: Individual, \evaluation
evaluation: > \evaluation

Evaluation Class

class Evaluation:
ctor: > Evaluation
ctor: Istream > Evaluation
print: Evaluation, Ostream
evaluate: Evaluation, Representation > \evaluation

class Function_Evaluation:
ctor: \function, Evaluation > Function_Evaluation
ctor: Istream > Function_Evaluation
print: Function_Evaluation, Ostream
evaluate: Function_Evaluation, Representation > \evaluation

class Binary_Bit_Vector_Evaluation:
ctor: > Binary_Bit_Vector_Evaluation
ctor: Istream, Binary_Bit_Vector_Evaluation
print: Binary_Bit_Vector_Evaluation, Ostream
evaluate: Binary_Bit_Vector_Evaluation, Bit_Vector_Representation > \evaluation

class Gray_Bit_Vector_Evaluation:
ctor: > Gray_Bit_Vector_Evaluation
ctor: Istream, Gray_Bit_Vector_Evaluation
print: Gray_Bit_Vector_Evaluation, Ostream
evaluate: Gray_Bit_Vector_Evaluation, Bit_Vector_Representation > \evaluation

Population Class

class Population:
ctor: > Population
ctor: [Individual] > Population
ctor: Istream > Population
print: Population, Ostream
add: Population, Individual
sub: Population, Individual
length: Population > \length
iterator: Population > PopulationIterator

class PopulationIterator:

ctor: Population > PopulationIterator
ctor: Istream > PopulationIterator
print: PopulationIterator, Ostream
next: PopulationIterator > \more, Individual, \fitness

Fitness Class

class Fitness:
ctor: > Fitness
ctor: Istream > Fitness
print: Fitness, Ostream
attach: Fitness, Population
fitness: Fitness, Individual > \fitness
length: Fitness > \size
iterator: Fitness > FitnessIterator

class FitnessIterator:
ctor: Fitness > FitnessIterator
ctor: Istream > FitnessIterator
print: FitnessIterator, Ostream
next: FitnessIterator > \more, Individual, \fitness

class Unity_Fitness:
ctor: > Unity_Fitness
ctor: Istream > Unity_Fitness
print: Unity_Fitness, Ostream
attach: Unity_Fitness, Population
fitness: Unity_Fitness, Individual > \fitness
length: Unity_Fitness > \size
iterator: Unity_Fitness > Unity_FitnessIterator

class Unity_FitnessIterator:
ctor: Unity_Fitness > Unity_FitnessIterator
ctor: Istream > Unity_FitnessIterator
print: Unity_FitnessIterator, Ostream
next: Unity_FitnessIterator > \more, Individual, \fitness

class Window_Fitness:
ctor: > Window_Fitness

ctor: Istream > Window_Fitness
print: Window_Fitness, Ostream
attach: Window_Fitness, Population, \guard
fitness: Window_Fitness, Individual > \fitness
length: Window_Fitness > \size
iterator: Window_Fitness > Window_FitnessIterator

class Window_FitnessIterator:
ctor: Window_Fitness > Window_FitnessIterator
ctor: Istream > Window_FitnessIterator
print: Window_FitnessIterator, Ostream
next: Window_FitnessIterator > \more, Individual, \fitness

class LinearNormalisation_Fitness:
ctor: > LinearNormalisation_Fitness
ctor: Istream > LinearNormalisation_Fitness
print: LinearNormalisation_Fitness, Ostream
attach: LinearNormalisation_Fitness, Population, \maximum, \decrement
fitness: LinearNormalisation_Fitness, Individual > \fitness
length: LinearNormalisation_Fitness > \size
iterator: LinearNormalisation_Fitness > LinearNormalisation_FitnessIterator

class LinearNormalisation_FitnessIterator:
ctor: LinearNormalisation_Fitness > LinearNormalisation_FitnessIterator
ctor: Istream > LinearNormalisation_FitnessIterator
print: LinearNormalisation_FitnessIterator, Ostream
next: LinearNormalisation_FitnessIterator > \more, Individual, \fitness

class LinearScaling_Fitness:
ctor: > LinearNormalisation_Fitness
ctor: Istream > LinearScaling_Fitness
print: LinearScaling_Fitness, Ostream
attach: LinearScaling_Fitness, Population, \a, \b
fitness: LinearScaling_Fitness, Individual > \fitness
length: LinearScaling_Fitness > \size
iterator: LinearScaling_Fitness > LinearScaling_FitnessIterator

class LinearScaling_FitnessIterator:
ctor: LinearScaling_Fitness > LinearScaling_FitnessIterator
ctor: Istream > LinearScaling_FitnessIterator

print: LinearScaling_FitnessIterator, Ostream
next: LinearScaling_FitnessIterator > \more, Individual, \fitness

class SigmaTruncation_Fitness:
ctor: > SigmaTruncation_Fitness
ctor: Istream > SigmaTruncation_Fitness
print: SigmaTruncation_Fitness, Ostream
attach: SigmaTruncation_Fitness, Population, \c
fitness: SigmaTruncation_Fitness, Individual > \fitness
length: SigmaTruncation_Fitness > \size
iterator: SigmaTruncation_Fitness > SigmaTruncation_FitnessIterator

class SigmaTruncation_FitnessIterator:
ctor: SigmaTruncation_Fitness > SigmaTruncation_FitnessIterator
ctor: Istream > SigmaTruncation_FitnessIterator
print: SigmaTruncation_FitnessIterator, Ostream
next: SigmaTruncation_FitnessIterator > \more, Individual, \fitness

class PowerLawScaling_Fitness:
ctor: > PowerLawScaling_Fitness
ctor: Istream > PowerLawScaling_Fitness
print: PowerLawScaling_Fitness, Ostream
attach: PowerLawScaling_Fitness, Population, \k
fitness: PowerLawScaling_Fitness, Individual > \fitness
length: PowerLawScaling_Fitness > \size
iterator: PowerLawScaling_Fitness > PowerLawScaling_FitnessIterator

class PowerLawScaling_FitnessIterator:
ctor: PowerLawScaling_Fitness > PowerLawScaling_FitnessIterator
ctor: Istream > PowerLawScaling_FitnessIterator
print: PowerLawScaling_FitnessIterator, Ostream
next: PowerLawScaling_FitnessIterator > \more, Individual, \fitness

Selector Class

class Selector:
ctor: > Selector
ctor: Istream > Selector
print: Selector, Ostream

attach: Selector, Fitness
select: Selector > Individual
unselect: Selector, Individual

class Best_Selector:
ctor: > Best_Selector
ctor: Istream > Best_Selector
print: Best_Selector, Ostream
attach: Best_Selector, Fitness
select: Best_Selector > Individual
unselect: Best_Selector, Individual

class Worst_Selector:
ctor: > Worst_Selector
ctor: Istream > Best_Selector
print: Worst_Selector, Ostream
attach: Worst_Selector, Fitness
select: Worst_Selector > Individual
unselect: Worst_Selector, Individual

class Random_Selector:
ctor: Random > Random_Selector
ctor: Istream > Random_Selector
print: Random_Selector, Ostream
attach: Random_Selector, Fitness
select: Random_Selector > Individual
unselect: Random_Selector, Individual

class RouletteWheel_Selector:
ctor: Random > RouletteWheel_Selector
ctor: Istream > RouletteWheel_Selector
print: RouletteWheel_Selector, Ostream
attach: RouletteWheel_Selector, Fitness
select: RouletteWheel_Selector > Individual
unselect: RouletteWheel_Selector, Individual

class AntiRouletteWheel_Selector:
ctor: Random > AntiRouletteWheel_Selector
ctor: Istream > AntiRouletteWheel_Selector
print: RouletteWheel_Selector, Ostream

attach: AntiRouletteWheel_Selector, Fitness
select: AntiRouletteWheel_Selector > Individual
unselect: AntiRouletteWheel_Selector, Individual

class Stochastic_Selector:
ctor: Random > Stochastic_Selector
ctor: Istream > Stochastic_Selector
print: Stochastic_Selector, Ostream
attach: Stochastic_Selector, Fitness, \cycle
select: Stochastic_Selector > Individual
unselect: Stochastic_Selector, Individual

class ExpectedValue_Selector:
ctor: Random > ExpectedValue_Selector
ctor: Istream > ExpectedValue_Selector
print: ExpectedValue_Selector, Ostream
attach: ExpectedValue_Selector, Fitness, \cycle
select: ExpectedValue_Selector > Individual
unselect: ExpectedValue_Selector, Individual

class Similarity_Selector:
ctor: Random > Similarity_Selector
ctor: Istream > Similarity_Selector
print: Similarity_Selector, Ostream
attach: Similarity_Selector, Fitness, Individual
select: Similarity_Selector > Individual
unselect: Similarity_Selector, Individual

Creator Class

class Creator:
ctor: > Creator
create: Creator, Population, \size
create: Creator, History, Population, \size
selector: Creator, Selector > \required
private Creator:
parents: Creator > \number
breed: Creator, [Representation], Population > \succeed

class And_Creator:
ctor: Creator, Creator > And_Creator
create: And_Creator, Population, \size
create: And_Creator, History, Population, \size
selector: And_Creator, Selector > \required
private And_Creator:
parents: And_Creator > \number
breed: And_Creator, [Representation], Population > \succeed

class Or_Creator:
ctor: Creator, \probability, Creator, \probability > Or_Creator
create: Or_Creator, Population, \size
create: Or_Creator, History, Population, \size
selector: Or_Creator, Selector > \required
private Or_Creator:
parents: Or_Creator > \number
breed: Or_Creator, [Representation], Population > \succeed

class Xor_Creator:
ctor: Creator, \weight, Creator, \weight > Xor_Creator
create: Xor_Creator, Population, \size
create: Xor_Creator, History, Population, \size
selector: Xor_Creator, Selector > \required
private Xor_Creator:
parents: Xor_Creator > \number
breed: Xor_Creator, [Representation], Population > \succeed

class Initialise:
ctor: > Initialise
create: Initialise, Population, \size
create: Initialise, History, Population, \size
private Initialise:
parents: Initialise > \number
breed: Initialise, [Representation], Population > \succeed

class Random_Initialise:
ctor: Random > Random_Initialise
create: Random_Initialise, Population, \size
create: Random_Initialise, History, Population, \size

class Bit_Random_Initialise:
ctor: Random > Bit_Random_Initialise
create: Bit_Random_Initialise, Population, \size
create: Bit_Random_Initialise, History, Population, \size

class Number_Random_Initialise:
ctor: Random, \lower, \upper > Number_Random_Initialise
create: Number_Random_Initialise Population, \size
create: Number_Random_Initialise, History, Population, \size

class Vector_Random_Initialise:
ctor: Random, Initialise > Vector_Random_Initialise
create: Vector_Random_Initialise, Population, \size
create: Vector_Random_Initialise, History, Population, \size

class Tagged_Vector_Random_Initialise:
ctor: Random, Initialise > Tagged_Vector_Random_Initialise
create: Tagged_Vector_Random_Initialise, Population, \size
create: Tagged_Vector_Random_Initialise, History, Population, \size

class Matrix_Random_Initialise:
ctor: Random, Initialise > Vector_Random_Initialise
create: Matrix_Random_Initialise, Population, \size
create: Matrix_Random_Initialise, History, Population, \size

class Ordered_Random_Initialise:
ctor: Random, Initialise > Ordered_Random_Initialise
create: Ordered_Random_Initialise, Population, \size
create: Ordered_Random_Initialise, History, Population, \size

class Search_Initialise:
ctor: Creator, \n > Search_Initialise
create: Search_Initialise, Population, \size
create: Search_Initialise, History, Population, \size
private Search_Initialise:
parents: Search_Initialise > \number
breed: Search_Initialise, [Representation], Population > \succeed

class Distributed_Initialise:
ctor: Random > Search_Initialise

create: Distributed_Initialise, Population, \size
create: Distributed_Initialise, History, Population, \size
class Heuristic_Initialise:
ctor: Random > Heuristic_Initialise
create: Heuristic_Initialise, Population, \size
create: Heuristic_Initialise, History, Population, \size

class Reproduction:
ctor: > Reproduction
create: Reproduction, Population, \size
create: Reproduction, History, Population, \size
selector: Reproduction, Selector > \required

class Clone_Reproduction:
ctor: > Clone_Reproduction
create: Clone_Reproduction, Population, \size
create: Clone_Reproduction, History, Population, \size
private Clone_Reproduction:
parents: Clone_Reproduction > \number
breed: Clone_Reproduction, [Representation], Population > \succeed

class Inversion_Reproduction:
ctor: Random, \portion > Inversion_Reproduction
private Inversion_Reproduction:
parents: Inversion_Reproduction > \number

class Biased_Inversion_Reproduction:
ctor: Random, \portion > Biased_Inversion_Reproduction
create: Biased_Inversion_Reproduction, Population, \size
create: Biased_Inversion_Reproduction, History, Population, \size
private Inversion_Reproduction:
breed: Biased_Inversion_Reproduction, [Representation], Population > \succeed

class Unbiased_Inversion_Reproduction:
ctor: Random, \portion > Unbiased_Inversion_Reproduction
create: Unbiased_Inversion_Reproduction, Population, \size
create: Unbiased_Inversion_Reproduction, History, Population, \size
private Inversion_Reproduction:
breed: Unbiased_Inversion_Reproduction, [Representation], Population > \succeed

class Mutation_Reproduction:
ctor: Random > Mutation_Reproduction
private Mutation_Reproduction:
parents: Mutation_Reproduction > \number

class Bit_Mutation_Reproduction:
ctor: Random > Bit_Mutation_Reproduction
create: Bit_Mutation_Reproduction, Population, \size
create: Bit_Mutation_Reproduction, History, Population, \size
private Bit_Mutation_Reproduction:
breed: Bit_Mutation_Reproduction, [Bit_Representation], Population > \succeed

class Number_Mutation_Reproduction:
ctor: Random > Number_Mutation_Reproduction

class Random_Number_Mutation_Reproduction:
ctor: Random, \lower, \upper > Random_Number_Mutation_Reproduction
create: Random_Number_Mutation_Reproduction, Population, \size
create: Random_Number_Mutation_Reproduction, History, Population, \size
private Random_Number_Mutation_Reproduction:
breed: Random_Number_Mutation_Reproduction, [Number_Representation],
    Population > \succeed

class Creep_Number_Mutation_Reproduction:
ctor: Random, \lower, \upper, \creep > Creep_Number_Mutation_Reproduction
create: Creep_Number_Mutation_Reproduction, Population, \size
create: Creep_Number_Mutation_Reproduction, History, Population, \size
private Creep_Number_Mutation_Reproduction:
breed: Creep_Number_Mutation_Reproduction, [Number_Representation], Population
    > \succeed

class Vector_Mutation_Reproduction:
ctor: Random > Vector_Mutation_Reproduction

class Single_Vector_Mutation_Reproduction:
ctor: Random, Mutation_Reproduction > Single_Vector_Mutation_Reproduction
create: Single_Vector_Mutation_Reproduction, Population, \size
create: Single_Vector_Mutation_Reproduction, History, Population, \size
private Single_Vector_Mutation_Reproduction:
breed: Single_Vector_Mutation_Reproduction, [Vector_Representation], Population >

```
    \succeed

class Deletion_Vector_Mutation_Reproduction:
ctor: Random > Deletion_Vector_Mutation_Reproduction
create: Deletion_Vector_Mutation_Reproduction, Population, \size
create: Deletion_Vector_Mutation_Reproduction, History, Population, \size
private Deletion_Vector_Mutation_Reproduction:
breed: Deletion_Vector_Mutation_Reproduction, [Vector_Representation], Population > \succeed

class Addition_Vector_Mutation_Reproduction:
ctor: Random > Addition_Vector_Mutation_Reproduction
create: Addition_Vector_Mutation_Reproduction, Population, \size
create: Addition_Vector_Mutation_Reproduction, History, Population, \size

class Duplication_Addition_Vector_Mutation_Reproduction:
ctor: Random > Duplication_Addition_Vector_Mutation_Reproduction
create: Duplication_Addition_Vector_Mutation_Reproduction, Population, \size
create: Duplication_Addition_Vector_Mutation_Reproduction, History, Population, \size
private Duplication_Addition_Vector_Mutation_Reproduction:
breed: Duplication_Addition_Vector_Mutation_Reproduction, [Vector_Representation], Population > \succeed

class Initialised_Addition_Vector_Mutation_Reproduction:
ctor: Random, Initialise > Initialised_Addition_Vector_Mutation_Reproduction
create: Initialised_Addition_Vector_Mutation_Reproduction, Population, \size
create: Initialised_Addition_Vector_Mutation_Reproduction, History, Population, \size
private Initialised_Addition_Vector_Mutation_Reproduction:
breed: Initialised_Addition_Vector_Mutation_Reproduction, [Vector_Representation], Population > \succeed

class Related_Addition_Vector_Mutation_Reproduction:
ctor: Random > Related_Addition_Vector_Mutation_Reproduction
create: Related_Addition_Vector_Mutation_Reproduction, Population, \size
create: Related_Addition_Vector_Mutation_Reproduction, History, Population, \size
private Related_Addition_Vector_Mutation_Reproduction:
breed: Related_Addition_Vector_Mutation_Reproduction, [Vector_Representation], Population > \succeed
```

class Cut_Vector_Mutation_Reproduction:
ctor: Random > Cut_Vector_Mutation_Reproduction
create: Cut_Mutation_Reproduction, Population, \size
create: Cut_Mutation_Reproduction, History, Population, \size
private Cut_Vector_Mutation_Reproduction:
breed: Cut_Vector_Mutation_Reproduction, [Vector_Representation], Population > \succeed

class Ordered_Mutation_Reproduction:
ctor: Random > Ordered_Mutation_Reproduction

class OrderBased_Ordered_Mutation_Reproduction:
ctor: Random > OrderBased_Ordered_Mutation_Reproduction
create: OrderBased_Ordered_Mutation_Reproduction, Population, \size
create: OrderBased_Ordered_Mutation_Reproduction, History, Population, \size
private OrderBased_Ordered_Mutation_Reproduction:
breed: OrderBased_Ordered_Mutation_Reproduction, [Ordered_Representation], Population > \succeed

class PositionBased_Ordered_Mutation_Reproduction:
ctor: Random > PositionBased_Ordered_Mutation_Reproduction
create: PositionBased_Ordered_Mutation_Reproduction, Population, \size
create: PositionBased_Ordered_Mutation_Reproduction, History, Population, \size
private PositionBased_Ordered_Mutation_Reproduction:
breed: PositionBased_Ordered_Mutation_Reproduction, [Ordered_Representation], Population > \succeed

class ScrambleSublist_Ordered_Mutation_Reproduction:
ctor: Random > ScrambleSublist_Ordered_Mutation_Reproduction
create: ScrambleSublist_Ordered_Mutation_Reproduction, Population, \size
create: ScrambleSublist_Ordered_Mutation_Reproduction, History, Population, \size
private ScrambleSublist_Ordered_Mutation_Reproduction:
breed: ScrambleSublist_Ordered_Mutation_Reproduction, [Ordered_Representation], Population > \succeed

class RandomHillClimb_Mutation_Reproduction:
ctor: Random, Mutation > RandomHillClimb_Mutation_Reproduction
create: RandomHillClimb_Mutation_Reproduction, Population, \size
create: RandomHillClimb_Mutation_Reproduction, History, Population, \size
private RandomHillClimb_Mutation_Reproduction:

breed: RandomHillClimb_Mutation_Reproduction, [Representation], Population > \succeed

class Crossover_Reproduction:
ctor: Random > Crossover_Reproduction
create: Crossover_Reproduction, Population, \size
create: Crossover_Reproduction, History, Population, \size
private Crossover_Reproduction:
parents: Crossover_Reproduction > \number

class Number_Crossover_Reproduction:
ctor: Random > Number_Crossover_Reproduction

class Average_Number_Crossover_Reproduction:
ctor: Random > Average_Number_Crossover_Reproduction
create: Average_Number_Crossover_Reproduction, Population, \size
create: Average_Number_Crossover_Reproduction, History, Population, \size
private Average_Number_Crossover_Reproduction:
breed: Average_Number_Crossover_Reproduction, [Number_Representation], Population > \succeed

class Vector_Crossover_Reproduction:
ctor: Random > Vector_Crossover_Reproduction

class OnePoint_Vector_Crossover_Reproduction:
ctor: Random > OnePoint_Vector_Crossover_Reproduction
create: OnePoint_Vector_Crossover_Reproduction, Population, \size
create: OnePoint_Vector_Crossover_Reproduction, History, Population, \size
private OnePoint_Vector_Crossover_Reproduction:
breed: OnePoint_Vector_Crossover_Reproduction, [Vector_Representation], Population > \succeed

class MultiPoint_Vector_Crossover_Reproduction:
ctor: Random, \n > MultiPoint_Vector_Crossover_Reproduction
create: MultiPoint_Vector_Crossover_Reproduction, Population, \size
create: MultiPoint_Vector_Crossover_Reproduction, History, Population, \size
private MultiPoint_Vector_Crossover_Reproduction:
breed: MultiPoint_Vector_Crossover_Reproduction, [Vector_Representation], Population > \succeed

```
class Segmented_Vector_Crossover_Reproduction:
ctor: Random, \probability > Segmented_Vector_Crossover_Reproduction
create: Segmented_Vector_Crossover_Reproduction, Population, \size
create: Segmented_Vector_Crossover_Reproduction, History, Population, \size
private Segmented_Vector_Crossover_Reproduction:
breed: Segmented_Vector_Crossover_Reproduction, [Vector_Representation],
    Population > \succeed

class Uniform_Vector_Crossover_Reproduction:
ctor: Random > Uniform_Vector_Crossover_Reproduction
create: Uniform_Vector_Crossover_Reproduction, Population, \size
create: Uniform_Vector_Crossover_Reproduction, History, Population, \size
private Uniform_Vector_Crossover_Reproduction:
breed: Uniform_Vector_Crossover_Reproduction, [Vector_Representation],
    Population > \succeed

class Shuffle_Vector_Crossover_Reproduction:
ctor: Random, Crossover > Shuffle_Vector_Crossover_Reproduction
create: Shuffle_Vector_Crossover_Reproduction, Population, \size
create: Shuffle_Vector_Crossover_Reproduction, History, Population, \size
private Shuffle_Vector_Crossover_Reproduction:
breed: Shuffle_Vector_Crossover_Reproduction, [Vector_Representation], Population
    > \succeed

class Traverse_Vector_Crossover_Reproduction:
ctor: Random, Crossover > Traverse_Vector_Crossover_Reproduction
create: Traverse_Vector_Crossover_Reproduction, Population, \size
create: Traverse_Vector_Crossover_Reproduction, History, Population, \size
private Traverse_Vector_Crossover_Reproduction:
breed: Traverse_Vector_Crossover_Reproduction, [Vector_Representation],
    Population > \succeed

class Splice_Vector_Crossover_Reproduction:
ctor: Random, Crossover > Splice_Vector_Crossover_Reproduction
create: Splice_Vector_Crossover_Reproduction, Population, \size
create: Splice_Vector_Crossover_Reproduction, History, Population, \size
private Splice_Vector_Crossover_Reproduction:
breed: Splice_Vector_Crossover_Reproduction, [Vector_Representation], Population >
    \succeed
```

class Ordered_Crossover_Reproduction:
ctor: Random > Ordered_Crossover_Reproduction

class PositionBased_Ordered_Crossover_Reproduction:
ctor: Random > PositionBased_Ordered_Crossover_Reproduction
create: PositionBased_Ordered_Crossover_Reproduction, Population, \size
create: PositionBased_Ordered_Crossover_Reproduction, History, Population, \size
private PositionBased_Ordered_Crossover_Reproduction:
breed: PositionBased_Ordered_Crossover_Reproduction, [Ordered_Representation], Population > \succeed

class OrderBased_Ordered_Crossover_Reproduction:
ctor: Random > OrderBased_Ordered_Crossover_Reproduction
create: OrderBased_Ordered_Crossover_Reproduction, Population, \size
create: OrderBased_Ordered_Crossover_Reproduction, History, Population, \size
private OrderBased_Ordered_Crossover_Reproduction:
breed: OrderBased_Ordered_Crossover_Reproduction, [Ordered_Representation], Population > \succeed

class Repair_Reproduction:
ctor: > Repair_Reproduction
create: Repair_Reproduction, Population, \size
create: Repair_Reproduction, History, Population, \size
private Crossover_Reproduction:
parents: Repair_Reproduction > \number

History Class

class History:
ctor: > History
ctor: Istream > History
print: History, Ostream
add: History, Individual, [Individual]
retrieve: History, Individual > [Individual]

Random Class

class Random:
ctor: > Random
ctor: \seed > Random
ctor: Istream > Random
print: Random, Ostream
value: Random > \boolean
value: Random, \range > \integer
value: Random, \lower, \upper > \integer
value: Random, \range, \precision > \real
value: Random, \lower, \upper, \precision > \real

Appendix E

Maintenance Document

The toolkit is a large collection of object many of which depend on each other for their function. Classes for the core toolkit were written and tested in a specific order to try to minimise the construction of test harnesses. The Random and Ending classes are entirely self contain and so they were chosen for the initial stages of implementation. Each test harness used tries all method calls individually and also in certain combinations which might otherwise have hidden certain flaws. Next the Representation, Evaluation, and Individual were written. Each of these has very minimal interfaces and required only small harnesses. The Population, Selection, and Fitness classes form a sequence of dependence and their creation and testing followed this. Because of its smaller size Model was written next although testing was held back. The most difficult implementation area was the Creation class and its children which are all highly connected. Testing of these classes was combined with the integration of all other classes. Fault finding was more difficult because of this but otherwise complex test harnesses would have been required. After the initial problems had been solved in the simplest possible toolkit organisation new components were completed and tested in place.

Appendix F

Status Report

There are three issues present in considering the status of the project - size, functionality, and quality of the toolkit. The Statement of Requirements demanded support for a variety of genetic algorithm techniques and also suggested that others would be helpful.

As a top priority a core toolkit was defined to allow the classic genetic algorithm to be implemented using the toolkit. This first target was reached, to achieve it approximately 20 components had to function in cooperation. Although the basic algorithm performs perfectly well there are some design decision, such as the *Attach* method for Fitness and Selector classes, which, in retrospect, might have been made differently. One current difficultly with the toolkit is that Individual objects are never disposed of and because of this any genetic algorithm built using the toolkit will eventually run out of memory.

There were a number of other suggested techniques such as vector, ordered list, and number representations which have also been completed to provide support for the more complex example algorithms. Work on reproduction operators, particularly for crossover, has not reached the range discussed during the design stage. Amongst numerous breeding techniques which could have been used only the simplest, the generational model, was implemented.

Another section of the Statement of Requirements gives a list of other facilities which might be useful in a genetic algorithm toolkit. These were all considered during the design stage and should be possible within the confines of the current design but none have been implemented. There are two areas in which this is a particular disappointment - family trees provided by a History class and the saving the status of an active genetic algorithm.

The demonstration algorithms proved that the toolkit was capable of speeding the creation of genetic algorithm by a combination of predefined and extensible components. However, none of these examples are good examples of a genetic algorithm. The Blues is only a toy suitable for the teaching of ideas. My implementation of the Travelling Salesman Problem produces very poor results compared to other techniques and the violin music notation algorithm is only likely to perform well with small data sets but does show scope for improvement.

Appendix G

Summary Log

August - September 1996

- review of available material
- initial plans possible components

October 1996

- problem definition
- statement of requirements
- component design

November 1996

- external specification
- component design
- first summary of genetic algorithms
- The Blues in Java and Ada
- Travelling Salesman Problem in Ada

December - January 1997

- core toolkit coding and testing

February 1997

- Travelling Salesman Problem using toolkit
- violin music notation using toolkit
- second summary of genetic algorithms

March 1997

- first draft of report

April 1997

- finalise report
- presentation

The project was allocated 12 hours each week for 21 weeks and therefore the estimated total time spent on it is 252 hours.

Appendix H

Project Code

Presented here is all toolkit code completed during the project:

- mutants.ads
- mutants-core-creator.bit-mutation-reproduction.adb
- mutants-core-creator.bit-mutation-reproduction.ads
- mutants-core-creator.onepoint-vector-crossover-reproduction.ads
- mutants-core-creator.bit_random_initialise.adb
- mutants-core-creator.bit_random_initialise.ads
- mutants-core-creator.bxor.adb
- mutants-core-creator.bxor.ads
- mutants-core-creator.clone_reproduction.adb
- mutants-core-creator.clone_reproduction.ads
- mutants-core-creator.onepoint_vector_crossover-reproduction.adb
- mutants-core-creator.onepoint_vector_crossover-reproduction.ads
- mutants-core-creator.single_bit_vector_reproduction.ads
- mutants-core-creator.single_vector_reproduction.adb
- mutants-core-creator.single_vector_reproduction.ads
- mutants-core-creator.vector_bit_random_initialise.adb
- mutants-core-creator.vector_initialise.adb
- mutants-core-creator.vector_initialise.ads
- mutants-core-creator.ads
- mutants-core-ending-count.adb
- mutants-core-ending-count.ads
- mutants-core-ending-time.adb
- mutants-core-ending-time.ads
- mutants-core-ending.ads
- mutants-core-evaluation-binary_bit_vector.adb
- mutants-core-evaluation-binary_bit_vector.ads
- mutants-core-evaluation-one.adb
- mutants-core-evaluation-one.ads
- mutants-core-evaluation.ads
- mutants-core-fitness-unity.adb
- mutants-core-fitness-unity.ads
- mutants-core-fitness.ads
- mutants-core-model-generation.adb
- mutants-core-model-generation.ads

- mutants-core-model.ads
- mutants-core-model.adb
- mutants-core-population.adb
- mutants-core-population.ads
- mutants-core-random.adb
- mutants-core-random.ads
- mutants-core-representation-bit.adb
- mutants-core-representation-bit.ads
- mutants-core-representation-bit_vector.ads
- mutants-core-representation-integer.adb
- mutants-core-representation-integer.ads
- mutants-core-representation-order.adb
- mutants-core-representation-order.ads
- mutants-core-representation-vector.adb
- mutants-core-representation-vector.ads
- mutants-core-representation.ads
- mutants-core-selector-antiroulette.adb
- mutants-core-selector-antiroulette.ads
- mutants-core-selector-cyclic.adb
- mutants-core-selector-cyclic.ads
- mutants-core-selector-high.adb
- mutants-core-selector-high.ads
- mutants-core-selector-low.adb
- mutants-core-selector-low.ads
- mutants-core-selector-roulette.adb
- mutants-core-selector-roulette.ads
- mutants-core-selector.ads
- mutants-extended-creator-integer_order_initialise.adb
- mutants-extended-creator-integer_order_initialise.ads
- mutants-extended-creator-integer_order_mutation_reproduction.ads
- mutants-extended-creator-integer_order_random_reproduction.ads
- mutants-extended-creator-order_crossover_reproduction.adb
- mutants-extended-creator-order_crossover_reproduction.ads
- mutants-extended-creator-order_initialise.adb
- mutants-extended-creator-order_initialise.ads
- mutants-extended-creator-order_mutation_reproduction.adb
- mutants-extended-creator-order_mutation_reproduction.ads
- mutants-extended-creator-order_random_reproduction.ads
- mutants-extended-creator-order_random_reproduction.ads
- mutants-extended-creator.ads

- mutants-extended-evaluation.ads
- mutants-extended-representation-bit_vector.ads
- mutants-extended-representation-integer_order.ads
- mutants-extended-representation.ads
- mutants-extended.ads

This is the code for the example genetic algorithms constructed using the toolkit:

- blues.adb
- blues_evaluation.adb
- blues_evaluation.ads
- tsp.adb
- tsp_evaluation.adb
- tsp_evaluation.ads
- music.adb
- music_evaluation.adb
- music_evaluation.ads