

Gerardium Rush: Pentlandite

Generated by Doxygen 1.9.7



<b>1 Class Index</b>	<b>1</b>
1.1 Class List	1
<b>2 File Index</b>	<b>3</b>
2.1 File List	3
<b>3 Class Documentation</b>	<b>5</b>
3.1 Circuit Class Reference	5
3.1.1 Detailed Description	5
3.1.2 Constructor & Destructor Documentation	6
3.1.2.1 Circuit()	6
3.1.2.2 ~Circuit()	6
3.1.3 Member Function Documentation	6
3.1.3.1 Check_Velocity()	6
3.1.3.2 toFile()	7
3.1.4 Member Data Documentation	8
3.1.4.1 circuit_vector	8
3.1.4.2 feeder	8
3.1.4.3 it	8
3.1.4.4 num_units	9
3.1.4.5 units	9
3.2 Circuit_Parameters Struct Reference	9
3.2.1 Detailed Description	9
3.2.2 Member Data Documentation	9
3.2.2.1 get_money	9
3.2.2.2 input_Fger	10
3.2.2.3 input_Fwaste	10
3.2.2.4 lose_money	10
3.2.2.5 max_iterations	10
3.2.2.6 tolerance	10
3.3 CircuitOptimizer Struct Reference	10
3.3.1 Member Data Documentation	10
3.3.1.1 feeder	10
3.3.1.2 num_units	11
3.4 Crossover< T > Class Template Reference	11
3.4.1 Detailed Description	11
3.4.2 Member Function Documentation	12
3.4.2.1 crossover()	12
3.4.2.2 mult_pts_crossover()	12
3.4.2.3 one_pt_crossover()	13
3.4.2.4 two_pts_crossover()	13
3.4.2.5 uniform_crossover()	14
3.5 CUnit Class Reference	14

3.5.1 Detailed Description	15
3.5.2 Constructor & Destructor Documentation	15
3.5.2.1 CUnit() [1/2]	15
3.5.2.2 CUnit() [2/2]	15
3.5.2.3 ~CUnit()	16
3.5.3 Member Function Documentation	16
3.5.3.1 calculateOutputFlowRates()	16
3.5.4 Member Data Documentation	16
3.5.4.1 conc	16
3.5.4.2 conc_num	16
3.5.4.3 F	17
3.5.4.4 F_old	17
3.5.4.5 id	17
3.5.4.6 input	17
3.5.4.7 k_ger	17
3.5.4.8 k_waste	17
3.5.4.9 phi	17
3.5.4.10 rho	17
3.5.4.11 tail	18
3.5.4.12 tails_num	18
3.5.4.13 V	18
3.6 DepthFirstSearch Class Reference	18
3.6.1 Detailed Description	19
3.6.2 Constructor & Destructor Documentation	19
3.6.2.1 DepthFirstSearch() [1/2]	19
3.6.2.2 DepthFirstSearch() [2/2]	19
3.6.3 Member Function Documentation	19
3.6.3.1 dfs()	19
3.6.3.2 visited()	20
3.6.4 Member Data Documentation	20
3.6.4.1 marked	20
3.7 DirectedGraph Class Reference	20
3.7.1 Detailed Description	21
3.7.2 Constructor & Destructor Documentation	21
3.7.2.1 DirectedGraph() [1/2]	21
3.7.2.2 DirectedGraph() [2/2]	21
3.7.3 Member Function Documentation	22
3.7.3.1 addEdge()	22
3.7.3.2 getNeighbors()	22
3.7.3.3 getNumEdges()	22
3.7.3.4 getNumVertices()	23
3.7.3.5 writeToFile()	23

3.7.4 Friends And Related Symbol Documentation . . . . .	23
3.7.4.1 operator<< . . . . .	23
3.7.5 Member Data Documentation . . . . .	24
3.7.5.1 adjList . . . . .	24
3.8 GA< T > Class Template Reference . . . . .	24
3.8.1 Detailed Description . . . . .	25
3.8.2 Constructor & Destructor Documentation . . . . .	26
3.8.2.1 GA() [1/2] . . . . .	26
3.8.2.2 GA() [2/2] . . . . .	26
3.8.2.3 ~GA() . . . . .	27
3.8.3 Member Function Documentation . . . . .	27
3.8.3.1 initPopulation() . . . . .	27
3.8.3.2 mutate() . . . . .	27
3.8.3.3 optimize() . . . . .	28
3.8.3.4 selectParent() . . . . .	29
3.8.3.5 setFitness() . . . . .	29
3.8.3.6 setValidity() . . . . .	30
3.8.3.7 toFile() . . . . .	30
3.8.3.8 tournamentSelection() . . . . .	30
3.8.3.9 updatePopulation() . . . . .	31
3.8.4 Member Data Documentation . . . . .	34
3.8.4.1 bestFitnessHistory . . . . .	34
3.8.4.2 CHROMOMAX . . . . .	34
3.8.4.3 CHROMOMIN . . . . .	34
3.8.4.4 CHROMOSOME_SIZE . . . . .	34
3.8.4.5 CONV_GEN . . . . .	34
3.8.4.6 cross_cnt . . . . .	34
3.8.4.7 cross_contrib . . . . .	34
3.8.4.8 Crossover_RATE . . . . .	34
3.8.4.9 cumulativeFitness . . . . .	35
3.8.4.10 ELITISM . . . . .	35
3.8.4.11 fitness . . . . .	35
3.8.4.12 fitness_ . . . . .	35
3.8.4.13 MAX_GENERATION . . . . .	35
3.8.4.14 mut_cnt . . . . .	35
3.8.4.15 mut_contrib . . . . .	36
3.8.4.16 MUTATION_RATE . . . . .	36
3.8.4.17 offspring . . . . .	36
3.8.4.18 POP_SIZE . . . . .	36
3.8.4.19 population . . . . .	36
3.8.4.20 scaledFitness . . . . .	36
3.8.4.21 validity_ . . . . .	36

3.8.4.22 verbose	37
3.9 Individual< T > Class Template Reference	37
3.9.1 Detailed Description	37
3.9.2 Constructor & Destructor Documentation	38
3.9.2.1 Individual() [1/2]	38
3.9.2.2 Individual() [2/2]	38
3.9.3 Member Function Documentation	38
3.9.3.1 operator<()	38
3.9.3.2 operator=()	39
3.9.3.3 operator==()	39
3.9.3.4 operator>()	39
3.9.4 Member Data Documentation	40
3.9.4.1 chromosome	40
3.9.4.2 fitness	40
3.10 Stream Class Reference	40
3.10.1 Detailed Description	40
3.10.2 Constructor & Destructor Documentation	41
3.10.2.1 Stream() [1/2]	41
3.10.2.2 Stream() [2/2]	41
3.10.2.3 ~Stream()	41
3.10.3 Member Function Documentation	41
3.10.3.1 operator+()	41
3.10.3.2 operator+=()	41
3.10.4 Member Data Documentation	42
3.10.4.1 ger	42
3.10.4.2 waste	42
3.11 Units Class Reference	42
3.11.1 Constructor & Destructor Documentation	42
3.11.1.1 Units() [1/2]	42
3.11.1.2 Units() [2/2]	42
3.11.2 Member Data Documentation	43
3.11.2.1 conc_num	43
3.11.2.2 marked	43
3.11.2.3 tails_num	43
3.12 UnitTestCircuit Class Reference	43
3.12.1 Constructor & Destructor Documentation	43
3.12.1.1 UnitTestCircuit()	43
3.12.1.2 ~UnitTestCircuit()	44
3.12.2 Member Function Documentation	44
3.12.2.1 run()	44
3.12.2.2 test_circuit_evaluation()	44
3.12.2.3 test_circuit_unit()	44

3.12.2.4 test_circuit_validity()	45
3.13 UnitTestCrossover Class Reference	45
3.13.1 Constructor & Destructor Documentation	45
3.13.1.1 UnitTestCrossover()	45
3.13.1.2 ~UnitTestCrossover()	46
3.13.2 Member Function Documentation	46
3.13.2.1 test_mult_pts_crossover()	46
3.13.2.2 test_one_pt_crossover()	46
3.13.2.3 test_two_pts_crossover()	47
3.13.2.4 test_uniform_crossover()	47
3.14 UnitTestGA Class Reference	48
3.14.1 Constructor & Destructor Documentation	48
3.14.1.1 UnitTestGA()	48
3.14.1.2 ~UnitTestGA()	49
3.14.2 Member Function Documentation	49
3.14.2.1 run()	49
3.14.2.2 test_global_optimum()	49
3.14.2.3 test_mutation()	49
3.14.2.4 test_select_parent()	50
3.14.3 Member Data Documentation	50
3.14.3.1 CHROMOMAX	50
3.14.3.2 CHROMOMIN	50
3.14.3.3 CHROMOSOME_SIZE	50
3.14.3.4 CONV_GEN	50
3.14.3.5 CROSSOVER_RATE	50
3.14.3.6 ga	50
3.14.3.7 MAX_GENERATION	51
3.14.3.8 MUTATION_RATE	51
3.14.3.9 POPULATION_SIZE	51
3.15 UnitTestIndividual Class Reference	51
3.15.1 Constructor & Destructor Documentation	51
3.15.1.1 UnitTestIndividual()	51
3.15.1.2 ~UnitTestIndividual()	52
3.15.2 Member Function Documentation	52
3.15.2.1 test_constructor()	52
3.15.2.2 test_Get_numUnits()	52
3.15.2.3 test_Get_Set_fitness()	52
3.15.2.4 test_Get_vectorLength()	52
3.15.3 Member Data Documentation	53
3.15.3.1 individual	53
<b>4 File Documentation</b>	<b>55</b>

4.1 CCircuit.h File Reference	55
4.1.1 Detailed Description	55
4.1.2 Function Documentation	56
4.1.2.1 Calculate_Circuit() [1/2]	56
4.1.2.2 Calculate_Circuit() [2/2]	57
4.2 CCircuit.h	58
4.3 CSimulator.h File Reference	59
4.3.1 Detailed Description	59
4.3.2 Function Documentation	60
4.3.2.1 Evaluate_Circuit() [1/2]	60
4.3.2.2 Evaluate_Circuit() [2/2]	61
4.4 CSimulator.h	62
4.5 CUnit.h File Reference	62
4.5.1 Detailed Description	62
4.6 CUnit.h	63
4.7 DirectedGraph.hpp File Reference	63
4.7.1 Detailed Description	64
4.8 DirectedGraph.hpp	64
4.9 GeneticAlgorithm.hpp File Reference	64
4.9.1 Detailed Description	65
4.9.2 Macro Definition Documentation	65
4.9.2.1 PARALLEL	65
4.9.3 Enumeration Type Documentation	65
4.9.3.1 CrossoverType	65
4.9.4 Function Documentation	66
4.9.4.1 RandomDouble()	66
4.9.4.2 RandomInt()	66
4.10 GeneticAlgorithm.hpp	66
4.11 CCircuit.cpp File Reference	68
4.11.1 Detailed Description	68
4.11.2 Function Documentation	69
4.11.2.1 markunits()	69
4.12 CSimulator.cpp File Reference	69
4.12.1 Detailed Description	70
4.12.2 Function Documentation	70
4.12.2.1 Calculate_Circuit() [1/2]	70
4.12.2.2 Calculate_Circuit() [2/2]	70
4.12.2.3 Evaluate_Circuit() [1/2]	72
4.12.2.4 Evaluate_Circuit() [2/2]	72
4.13 CUnit.cpp File Reference	73
4.13.1 Detailed Description	73
4.14 DirectedGraph.cpp File Reference	73



4.14.1 Detailed Description . . . . .	73
4.14.2 Function Documentation . . . . .	73
4.14.2.1 operator<<() . . . . .	73
4.15 GeneticAlgorithm.cpp File Reference . . . . .	74
4.15.1 Function Documentation . . . . .	74
4.15.1.1 RandomDouble() . . . . .	74
4.15.1.2 RandomInt() . . . . .	74
4.16 main.cpp File Reference . . . . .	75
4.16.1 Function Documentation . . . . .	75
4.16.1.1 fitness() . . . . .	75
4.16.1.2 main() . . . . .	75
4.16.1.3 validity() . . . . .	76
4.16.2 Variable Documentation . . . . .	76
4.16.2.1 myCircuit . . . . .	76
4.17 test_all.cpp File Reference . . . . .	77
4.17.1 Function Documentation . . . . .	77
4.17.1.1 main() . . . . .	77
4.18 test_Circuit.cpp File Reference . . . . .	77
4.19 test_Circuit.cpp . . . . .	78
4.20 test_CircuitEvaluation.cpp File Reference . . . . .	79
4.20.1 Function Documentation . . . . .	79
4.20.1.1 main() . . . . .	79
4.20.1.2 test() . . . . .	79
4.21 test_CircuitValidity.cpp File Reference . . . . .	79
4.21.1 Function Documentation . . . . .	80
4.21.1.1 main() . . . . .	80
4.21.1.2 testValidity() . . . . .	80
4.22 test_Crossover.cpp File Reference . . . . .	80
4.22.1 Function Documentation . . . . .	81
4.22.1.1 printChromosome() . . . . .	81
4.22.2 Variable Documentation . . . . .	81
4.22.2.1 crossover . . . . .	81
4.23 test_Crossover.cpp . . . . .	81
4.24 test_Cunit.cpp File Reference . . . . .	83
4.24.1 Function Documentation . . . . .	83
4.24.1.1 main() . . . . .	83
4.25 test_DirectedGraph.cpp File Reference . . . . .	84
4.25.1 Function Documentation . . . . .	84
4.25.1.1 main() . . . . .	84
4.25.1.2 testDFS() . . . . .	84
4.25.1.3 testDigraph() . . . . .	85
4.26 test_GeneticAlgorithm.cpp File Reference . . . . .	85

4.26.1 Function Documentation . . . . .	85
4.26.1.1 Rosenbrock() . . . . .	85
4.26.1.2 validity() . . . . .	86
4.27 test_GeneticAlgorithm.cpp . . . . .	86
4.28 test_Individual.cpp File Reference . . . . .	87
4.28.1 Function Documentation . . . . .	87
4.28.1.1 main() . . . . .	87
4.29 test_RandomCircuit.cpp File Reference . . . . .	87
4.29.1 Function Documentation . . . . .	88
4.29.1.1 generateRandomVector() . . . . .	88
4.29.1.2 main() . . . . .	88
4.30 Unittests.h File Reference . . . . .	88
4.31 Unittests.h . . . . .	89
<b>Index</b>	<b>91</b>

# Chapter 1

## Class Index

### 1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

<a href="#">Circuit</a>	
<a href="#">Circuit</a> class . . . . .	5
<a href="#">Circuit_Parameters</a>	
A Structure to store the parameters for the circuit simulator . . . . .	9
<a href="#">CircuitOptimizer</a> . . . . .	10
<a href="#">Crossover&lt; T &gt;</a>	
Performs crossover operation between two parent individuals . . . . .	11
<a href="#">CUnit</a>	
A Unit class to store the unit information and processing . . . . .	14
<a href="#">DepthFirstSearch</a>	
Depth first search class for directed graph . . . . .	18
<a href="#">DirectedGraph</a>	
A directed graph . . . . .	20
<a href="#">GA&lt; T &gt;</a>	
Template class representing a Genetic Algorithm ( <a href="#">GA</a> ) . . . . .	24
<a href="#">Individual&lt; T &gt;</a>	
Template class representing an individual in the <a href="#">GA</a> population . . . . .	37
<a href="#">Stream</a>	
A <a href="#">Stream</a> class to store the stream information . . . . .	40
<a href="#">Units</a> . . . . .	42
<a href="#">UnitTestCircuit</a> . . . . .	43
<a href="#">UnitTestCrossover</a> . . . . .	45
<a href="#">UnitTestGA</a> . . . . .	48
<a href="#">UnitTestIndividual</a> . . . . .	51



## Chapter 2

# File Index

### 2.1 File List

Here is a list of all files with brief descriptions:

<a href="#">CCircuit.h</a>	
<a href="#">Circuit</a> class . . . . .	55
<a href="#">CSimulator.h</a>	
Simulator class . . . . .	59
<a href="#">CUnit.h</a>	
Unit class . . . . .	62
<a href="#">DirectedGraph.hpp</a>	
Directed graph class . . . . .	63
<a href="#">GeneticAlgorithm.hpp</a>	
Header file for the GeneticAlgorithm class . . . . .	64
<a href="#">CCircuit.cpp</a>	
Circuit class implementation . . . . .	68
<a href="#">CSimulator.cpp</a>	
Simulator calculations . . . . .	69
<a href="#">CUnit.cpp</a>	
CUnit class implementation . . . . .	73
<a href="#">DirectedGraph.cpp</a>	
Directed graph implementation . . . . .	73
<a href="#">GeneticAlgorithm.cpp</a>	
. . . . .	74
<a href="#">main.cpp</a>	75
<a href="#">test_all.cpp</a>	77
<a href="#">test_Circuit.cpp</a>	77
<a href="#">test_CircuitEvaluation.cpp</a>	79
<a href="#">test_CircuitValidity.cpp</a>	79
<a href="#">test_Crossover.cpp</a>	80
<a href="#">test_Cunit.cpp</a>	83
<a href="#">test_DirectedGraph.cpp</a>	84
<a href="#">test_GeneticAlgorithm.cpp</a>	85
<a href="#">test_Individual.cpp</a>	87
<a href="#">test_RandomCircuit.cpp</a>	87
<a href="#">Unittests.h</a>	88



## Chapter 3

# Class Documentation

### 3.1 Circuit Class Reference

[Circuit](#) class.

```
#include <CCircuit.h>
```

#### Public Member Functions

- [Circuit](#) (std::vector< int > [circuit\\_vector](#))  
*Construct a new [Circuit](#):: [Circuit](#) object.*
- [~Circuit](#) ()  
*Destroy the [Circuit](#) object.*
- void [ToFile](#) ()  
*To write the circuit information to a file.*

#### Static Public Member Functions

- static bool [Check\\_VValidity](#) (const std::vector< int > &[circuit\\_vector](#))  
*Function to check the validity of the circuit vector.*

#### Public Attributes

- int [num\\_units](#)
- int [feeder](#)
- std::vector< int > [circuit\\_vector](#)
- std::vector< [CUnit](#) > [units](#)
- int [it](#)

#### 3.1.1 Detailed Description

[Circuit](#) class.

This class is used to store the circuit information

Author

acse-sm222

### 3.1.2 Constructor & Destructor Documentation

#### 3.1.2.1 Circuit()

```
Circuit::Circuit (
    std::vector< int > circuit_vector )
```

Construct a new [Circuit::Circuit](#) object.

Constructor taking the circuit vector

This constructor takes the circuit vector and initializes the circuit The circuit vector is of the form: Feeder, C0, T0, C1, T1, C2, T2, ... , Cn, Tn Two extra units are added to the circuit, one for the concentrate and one for the tails Each unit is stored in the units vector

##### Parameters

<i>circuit_vector</i>	
-----------------------	--

```
00064 {
00065     this->num_units = (circuit_vector.size() - 1) / 2 + 2;
00066     this->units.resize(this->num_units);
00067     this->circuit_vector = circuit_vector;
00068     this->feeder = circuit_vector[0];
00069     this->it = 0;
00070
00071     // Initialize the units with unit ID and destinations
00072     for (int i = 0; i < this->num_units - 2; ++i)
00073     {
00074         this->units[i] = CUnit(i, circuit_vector[2 * i + 1], circuit_vector[2 * i + 2]);
00075     }
00076 }
```

#### 3.1.2.2 ~Circuit()

```
Circuit::~Circuit ( )
```

Destroy the [Circuit](#) object.

##### Destructor

```
00128 {
00129 }
```

### 3.1.3 Member Function Documentation

#### 3.1.3.1 Check\_Validity()

```
bool Circuit::Check_Validity (
    const std::vector< int > & circuit_vector ) [static]
```

Function to check the validity of the circuit vector.

Function to check the validity of the circuit vector

##### Parameters

<i>circuit_vector</i>	
-----------------------	--



## Returns

true

false TODO: Add more validity checks

```

00140 {
00141     if (circuit_vector.size() % 2 != 1)
00142     {
00143         return false;
00144     }
00145     else
00146     {
00147         int feeder = circuit_vector[0];
00148         int num_units = (circuit_vector.size() - 1) / 2;
00149
00150         // 0. No units conc should go directly to tails and vice versa
00151         for (int i = 0; i < num_units; ++i)
00152         {
00153             if (circuit_vector[2 * i + 1] == num_units + 1 || circuit_vector[2 * i + 2] == num_units)
00154             {
00155                 return false;
00156             }
00157         }
00158
00159         // 1. No self loops
00160         for (int i = 0; i < num_units; ++i)
00161         {
00162             if (circuit_vector[2 * i + 1] == i || circuit_vector[2 * i + 2] == i)
00163             {
00164                 return false;
00165             }
00166         }
00167
00168         // 2. Tailings and Concentrations don't have same destination
00169         for (int i = 0; i < num_units; ++i)
00170         {
00171             if (circuit_vector[2 * i + 1] == circuit_vector[2 * i + 2])
00172             {
00173                 return false;
00174             }
00175         }
00176
00177         // 3. Every node has a feed - Each node must lie on atleast one of the graphs from the feeder
00178         std::vector<int> hasfeed(num_units + 2, 0);
00179         hasfeed[feeder] += 1;
00180         for (int i = 0; i < num_units; ++i)
00181         {
00182             hasfeed[circuit_vector[2 * i + 1]] += 1;
00183             hasfeed[circuit_vector[2 * i + 2]] += 1;
00184         }
00185
00186         for (int i = 0; i < num_units + 2; ++i)
00187         {
00188             if (hasfeed[i] == 0)
00189             {
00190                 return false;
00191             }
00192         }
00193
00194         // 4. Every unit has a route forward to leaf nodes on BOTH graphs
00195         for (int i = 0; i < num_units; ++i)
00196         {
00197             std::vector<Units> myunits(num_units + 2);
00198             for (int j = 0; j < num_units; ++j)
00199             {
00200                 myunits[j].conc_num = circuit_vector[2 * j + 1];
00201                 myunits[j].tails_num = circuit_vector[2 * j + 2];
00202             }
00203             myunits[num_units].conc_num = 1e5;
00204             myunits[num_units + 1].tails_num = 1e5;
00205             markunits(myunits, i);
00206             if (myunits[num_units].marked == false || myunits[num_units + 1].marked == false)
00207             {
00208                 return false;
00209             }
00210         }
00211         return true;
00212     }
00213 }

```

## 3.1.3.2 toFile()

```
void Circuit::toFile ( )
```

To write the circuit information to a file.

Function to write the circuit information to a file

This function writes the circuit information to a file. It creates a [DirectedGraph](#) object for the concentrate and tails and writes the graph to a file

```
00084 {
00085     DirectedGraph Conc(num_units);
00086     DirectedGraph Tail(num_units);
00087     for (int i = 0; i < num_units - 2; ++i)
00088     {
00089         Conc.addEdge(i, circuit_vector[2 * i + 1]);
00090         Tail.addEdge(i, circuit_vector[2 * i + 2]);
00091     }
00092
00093     Conc.addEdge(num_units - 2, num_units + 1);
00094     Conc.addEdge(num_units - 1, num_units + 1);
00095     Tail.addEdge(num_units - 2, num_units + 1);
00096     Tail.addEdge(num_units - 1, num_units + 1);
00097
00098     Circuit_Parameters params;
00099     Calculate_Circuit(*this, params);
00100
00101     std::fstream ConcFile;
00102     std::fstream TailFile;
00103     std::fstream DataFile;
00104     ConcFile.open("C.txt", std::ios::out);
00105     TailFile.open("T.txt", std::ios::out);
00106     DataFile.open("D.txt", std::ios::out);
00107
00108     ConcFile << Conc;
00109     ConcFile << feeder << std::endl;
00110     TailFile << Tail;
00111     TailFile << feeder << std::endl;
00112
00113     for (int i = 0; i < num_units - 2; ++i)
00114     {
00115         DataFile << i << " " << units[i].conc.ger << " " << units[i].conc.waste << " " << units[i].tail.ger << " "
00116         << units[i].tail.waste << std::endl;
00117     }
00118     ConcFile.close();
00119     TailFile.close();
00120     DataFile.close();
00121 }
```

### 3.1.4 Member Data Documentation

#### 3.1.4.1 circuit\_vector

```
std::vector<int> Circuit::circuit_vector
```

[Circuit](#) vector

#### 3.1.4.2 feeder

```
int Circuit::feeder
```

Feeder unit

#### 3.1.4.3 it

```
int Circuit::it
```

Number of iterations

#### 3.1.4.4 num\_units

```
int Circuit::num_units
```

Number of units in the circuit

#### 3.1.4.5 units

```
std::vector<CUnit> Circuit::units
```

Vector of Cunits

The documentation for this class was generated from the following files:

- [CCircuit.h](#)
- [CCircuit.cpp](#)

## 3.2 Circuit\_Parameters Struct Reference

A Structure to store the parameters for the circuit simulator.

```
#include <CCircuit.h>
```

### Public Attributes

- double [tolerance](#) = 1e-6
- int [max\\_iterations](#) = 1000
- double [input\\_Fger](#) = 10.0
- double [input\\_Fwaste](#) = 100.0
- double [get\\_money](#) = 100.0
- double [lose\\_money](#) = 500.0

### 3.2.1 Detailed Description

A Structure to store the parameters for the circuit simulator.

### 3.2.2 Member Data Documentation

#### 3.2.2.1 get\_money

```
double Circuit_Parameters::get_money = 100.0
```

Money earned per unit of mineral

### 3.2.2.2 input\_Fger

```
double Circuit_Parameters::input_Fger = 10.0
```

Input flow rate of the mineral

### 3.2.2.3 input\_Fwaste

```
double Circuit_Parameters::input_Fwaste = 100.0
```

Input flow rate of the waste

### 3.2.2.4 lose\_money

```
double Circuit_Parameters::lose_money = 500.0
```

Money lost per unit of waste

### 3.2.2.5 max\_iterations

```
int Circuit_Parameters::max_iterations = 1000
```

Maximum number of iterations

### 3.2.2.6 tolerance

```
double Circuit_Parameters::tolerance = 1e-6
```

Tolerance for simulator convergence

The documentation for this struct was generated from the following file:

- [CCircuit.h](#)

## 3.3 CircuitOptimizer Struct Reference

### Public Attributes

- int [feeder](#) = 0
- int [num\\_units](#) = 5

### 3.3.1 Member Data Documentation

#### 3.3.1.1 feeder

```
int CircuitOptimizer::feeder = 0
```

### 3.3.1.2 num\_units

```
int CircuitOptimizer::num_units = 5
```

The documentation for this struct was generated from the following file:

- [main.cpp](#)

## 3.4 Crossover< T > Class Template Reference

Performs crossover operation between two parent individuals.

```
#include <GeneticAlgorithm.hpp>
```

### Static Public Member Functions

- static void [one\\_pt\\_crossover](#) ([Individual](#)< T > &parent1, [Individual](#)< T > &parent2, [Individual](#)< T > &child1, [Individual](#)< T > &child2)
- static void [two\\_pts\\_crossover](#) ([Individual](#)< T > &parent1, [Individual](#)< T > &parent2, [Individual](#)< T > &child1, [Individual](#)< T > &child2)
- static void [uniform\\_crossover](#) ([Individual](#)< T > &parent1, [Individual](#)< T > &parent2, [Individual](#)< T > &child1, [Individual](#)< T > &child2)
- static void [mult\\_pts\\_crossover](#) ([Individual](#)< T > &parent1, [Individual](#)< T > &parent2, [Individual](#)< T > &child1, [Individual](#)< T > &child2)
- static void [crossover](#) ([Individual](#)< T > &parent1, [Individual](#)< T > &parent2, [Individual](#)< T > &child1, [Individual](#)< T > &child2, [CrossoverType](#) type)

### 3.4.1 Detailed Description

```
template<typename T>
class Crossover< T >
```

Performs crossover operation between two parent individuals.

Functions perform crossover between two parent individuals, producing two child individuals. The crossover operation combines genetic information from the parent individuals to create new individuals.

#### Parameters

<i>parent1</i>	The first parent individual.
<i>parent2</i>	The second parent individual.
<i>child1</i>	Reference to the first child individual to be created.
<i>child2</i>	Reference to the second child individual to be created.

## 3.4.2 Member Function Documentation

### 3.4.2.1 crossover()

```

template<typename T >
void Crossover< T >::crossover (
    Individual< T > & parent1,
    Individual< T > & parent2,
    Individual< T > & child1,
    Individual< T > & child2,
    CrossoverType type ) [static]
00555 {
00556     switch (type)
00557     {
00558
00559     case CrossoverType::ONE_PT:
00560         one_pt_crossover(parent1, parent2, child1, child2);
00561         break;
00562
00563     case CrossoverType::TWO_PTS:
00564         two_pts_crossover(parent1, parent2, child1, child2);
00565         break;
00566
00567     case CrossoverType::MULT_PTS:
00568         mult_pts_crossover(parent1, parent2, child1, child2);
00569         break;
00570
00571     case CrossoverType::UNIFORM:
00572         uniform_crossover(parent1, parent2, child1, child2);
00573         break;
00574     default:
00575         one_pt_crossover(parent1, parent2, child1, child2);
00576     };
00577 }

```

### 3.4.2.2 mult\_pts\_crossover()

```

template<typename T >
void Crossover< T >::mult_pts_crossover (
    Individual< T > & parent1,
    Individual< T > & parent2,
    Individual< T > & child1,
    Individual< T > & child2 ) [static]
00493 {
00494     int chromosomeLength = parent1.chromosome.size();
00495     std::uniform_int_distribution<> distr(0, parent1.chromosome.size() - 1);
00496
00497     // Generate a number of crossover points
00498     int numCrossoverPoints = 4;
00499
00500     // Generate crossover points
00501     std::set<int> crossoverPoints;
00502     while (crossoverPoints.size() < numCrossoverPoints)
00503     {
00504         crossoverPoints.insert(distr(gen));
00505     }
00506     crossoverPoints.insert(chromosomeLength);
00507
00508     // Perform crossover
00509     bool sw = false;
00510     auto startIter = crossoverPoints.begin();
00511     for (int i = 0; i < chromosomeLength; ++i)
00512     {
00513         // If i is in crossoverPoints, switch genes
00514         if (i == *startIter)
00515         {
00516             sw = !sw;
00517             ++startIter;
00518         }
00519         if (!sw)
00520         {
00521             child1.chromosome[i] = parent1.chromosome[i];
00522             child2.chromosome[i] = parent2.chromosome[i];

```

```

00523     }
00524     else
00525     {
00526         child1.chromosome[i] = parent2.chromosome[i];
00527         child2.chromosome[i] = parent1.chromosome[i];
00528     }
00529 }
00530 };

```

### 3.4.2.3 one\_pt\_crossover()

```

template<typename T >
void Crossover< T >::one_pt_crossover (
    Individual< T > & parent1,
    Individual< T > & parent2,
    Individual< T > & child1,
    Individual< T > & child2 ) [static]
{
    00438 {
    00439     std::uniform_int_distribution<> distr(1, parent1.chromosome.size() - 1);
    00440
    00441     int crossoverPoint = distr(gen);
    00442     for (int i = 0; i < parent1.chromosome.size(); ++i)
    00443     {
    00444         if (i < crossoverPoint)
    00445         {
    00446             child1.chromosome[i] = parent1.chromosome[i];
    00447             child2.chromosome[i] = parent2.chromosome[i];
    00448         }
    00449         else
    00450         {
    00451             child1.chromosome[i] = parent2.chromosome[i];
    00452             child2.chromosome[i] = parent1.chromosome[i];
    00453         }
    00454     }
    00455 }

```

### 3.4.2.4 two\_pts\_crossover()

```

template<typename T >
void Crossover< T >::two_pts_crossover (
    Individual< T > & parent1,
    Individual< T > & parent2,
    Individual< T > & child1,
    Individual< T > & child2 ) [static]
{
    00459 {
    00460     // Define two crossover points
    00461     std::uniform_int_distribution<> distr(0, parent1.chromosome.size() - 1);
    00462
    00463     int crossoverPoint1 = distr(gen);
    00464     int crossoverPoint2 = distr(gen);
    00465
    00466     while (crossoverPoint1 == crossoverPoint2 ||
    00467            (crossoverPoint1 == 0 && crossoverPoint2 == parent1.chromosome.size() - 1) ||
    00468            (crossoverPoint2 == 0 && crossoverPoint1 == parent1.chromosome.size() - 1))
    00469     {
    00470         crossoverPoint2 = distr(gen);
    00471     };
    00472
    00473     if (crossoverPoint1 > crossoverPoint2)
    00474         std::swap(crossoverPoint1, crossoverPoint2);
    00475
    00476     for (int i = 0; i < parent1.chromosome.size(); ++i)
    00477     {
    00478         if (i < crossoverPoint1 || i > crossoverPoint2)
    00479         {
    00480             child1.chromosome[i] = parent1.chromosome[i];
    00481             child2.chromosome[i] = parent2.chromosome[i];
    00482         }
    00483         else
    00484         {
    00485             child1.chromosome[i] = parent2.chromosome[i];
    00486             child2.chromosome[i] = parent1.chromosome[i];
    00487         }
    00488     }
    00489 };

```

### 3.4.2.5 uniform\_crossover()

```

template<typename T >
void Crossover< T >::uniform_crossover (
    Individual< T > & parent1,
    Individual< T > & parent2,
    Individual< T > & child1,
    Individual< T > & child2 ) [static]
00534 {
00535     double crossoverRate = 0.5;
00536     double r;
00537     for (int i = 0; i < parent1.chromosome.size(); ++i)
00538     {
00539         r = RandomDouble(0, 1);
00540         if (r < crossoverRate)
00541         {
00542             child1.chromosome[i] = parent1.chromosome[i];
00543             child2.chromosome[i] = parent2.chromosome[i];
00544         }
00545         else
00546         {
00547             child1.chromosome[i] = parent2.chromosome[i];
00548             child2.chromosome[i] = parent1.chromosome[i];
00549         }
00550     }
00551 };

```

The documentation for this class was generated from the following files:

- [GeneticAlgorithm.hpp](#)
- [GeneticAlgorithm.cpp](#)

## 3.5 CUnit Class Reference

A Unit class to store the unit information and processing.

```
#include <CUnit.h>
```

### Public Member Functions

- [CUnit \(\)](#)  
*Construct a new CUnit object.*
- [CUnit \(int id, int conc\\_num, int tails\\_num\)](#)  
*Construct a new CUnit object.*
- void [calculateOutputFlowRates](#) (const [Stream](#) &INP)  
*Calculate the output flow rates.*
- [~CUnit \(\)](#)

### Public Attributes

- int [conc\\_num](#)
- int [tails\\_num](#)
- [Stream](#) input
- [Stream](#) conc
- [Stream](#) tail
- double [F](#)
- double [F\\_old](#)



## Private Attributes

- int `id`
- double `k_ger` = 0.005
- double `k_waste` = 0.0005
- int `rho` = 3000
- double `phi` = 0.1
- int `V` = 10

### 3.5.1 Detailed Description

A Unit class to store the unit information and processing.

#### Author

acse-yl1922, acse-sm222

### 3.5.2 Constructor & Destructor Documentation

#### 3.5.2.1 CUnit() [1/2]

```
CUnit::CUnit ( )
```

Construct a new `CUnit` object.

##### Default constructor

```
00019 {
00020     this->id = 0;
00021     this->conc_num = -1;
00022     this->tails_num = -1;
00023 }
```

#### 3.5.2.2 CUnit() [2/2]

```
CUnit::CUnit (
    int id,
    int conc_num,
    int tails_num )
```

Construct a new `CUnit` object.

Constructor taking the `id`, `conc_num` and `tails_num`

#### Parameters

<i>id</i>	Unit ID
<i>conc_num</i>	Concentrate stream number
<i>tails_num</i>	Tails stream number

```
00033 {
00034     this->id = id;
00035     this->conc_num = conc_num;
00036     this->tails_num = tails_num;
```

```
00037 }
```

### 3.5.2.3 ~CUnit()

```
CUnit::~CUnit ( ) [inline]
```

Destructor

## 3.5.3 Member Function Documentation

### 3.5.3.1 calculateOutputFlowRates()

```
void CUnit::calculateOutputFlowRates (
    const Stream & INP )
```

Calculate the output flow rates.

Function to calculate the output flow rates

#### Parameters

<i>INP</i>	Input stream
------------	--------------

```
00046 {
00047
00048     // Calculate Tau
00049     double sumFlowRate = INP.ger + INP.waste;
00050     if (std::abs(sumFlowRate) < 1e-10)
00051     {
00052         sumFlowRate = 1e-10; // Set sumFlowRate to the minimum flow rate
00053     }
00054     double Tau = phi * V * rho / sumFlowRate;
00055
00056     // Calculate R
00057     double R_ger = (k_ger * Tau) / (1 + (k_ger * Tau));
00058     double R_waste = (k_waste * Tau) / (1 + (k_waste * Tau));
00059
00060     // Calculate C
00061     this->conc.ger = INP.ger * R_ger;
00062     this->conc.waste = INP.waste * R_waste;
00063
00064     // Calculate T
00065     this->tail.ger = INP.ger * (1 - R_ger);
00066     this->tail.waste = INP.waste * (1 - R_waste);
00067 }
```

## 3.5.4 Member Data Documentation

### 3.5.4.1 conc

```
Stream CUnit::conc
```

Concentrate stream

### 3.5.4.2 conc\_num

```
int CUnit::conc_num
```

index of the unit to which this unit's concentrate stream is connected

### 3.5.4.3 F

```
double CUnit::F
```

### 3.5.4.4 F\_old

```
double CUnit::F_old
```

Feed values

### 3.5.4.5 id

```
int CUnit::id [private]
```

Unit id

### 3.5.4.6 input

```
Stream CUnit::input
```

Input stream

### 3.5.4.7 k\_ger

```
double CUnit::k_ger = 0.005 [private]
```

Germanium recovery rate

### 3.5.4.8 k\_waste

```
double CUnit::k_waste = 0.0005 [private]
```

Waste recovery rate

### 3.5.4.9 phi

```
double CUnit::phi = 0.1 [private]
```

Percentage of solid content in the stream

### 3.5.4.10 rho

```
int CUnit::rho = 3000 [private]
```

Density of the mineral

### 3.5.4.11 tail

```
Stream CUnit::tail
```

Tails stream

### 3.5.4.12 tails\_num

```
int CUnit::tails_num
```

index of the unit to which this unit's tails stream is connected

### 3.5.4.13 V

```
int CUnit::V = 10 [private]
```

Volume of the unit

The documentation for this class was generated from the following files:

- [CUnit.h](#)
- [CUnit.cpp](#)

## 3.6 DepthFirstSearch Class Reference

Depth first search class for directed graph.

```
#include <DirectedGraph.hpp>
```

### Public Member Functions

- [DepthFirstSearch](#) ()
- [DepthFirstSearch](#) (const [DirectedGraph](#) &G, int s)  
*Construct a new Depth First Search:: Depth First Search object.*
- void [dfs](#) (const [DirectedGraph](#) &G, int u)  
*Perform depth first search.*
- bool [visited](#) (int v) const  
*Check if a vertex is visited.*

### Private Attributes

- std::vector< bool > [marked](#)

### 3.6.1 Detailed Description

Depth first search class for directed graph.

Author

acse-sm222

### 3.6.2 Constructor & Destructor Documentation

#### 3.6.2.1 DepthFirstSearch() [1/2]

```
DepthFirstSearch::DepthFirstSearch ( ) [inline]
00046 {};
```

#### 3.6.2.2 DepthFirstSearch() [2/2]

```
DepthFirstSearch::DepthFirstSearch (
    const DirectedGraph & G,
    int s )
```

Construct a new Depth First Search:: Depth First Search object.

Default constructor Constructor taking the graph and the source vertex

Parameters

<i>G</i>	Directed graph
<i>s</i>	Source vertex

```
00108 {
00109     marked.resize(G.getNumVertices());
00110     dfs(G, s);
00111 }
```

### 3.6.3 Member Function Documentation

#### 3.6.3.1 dfs()

```
void DepthFirstSearch::dfs (
    const DirectedGraph & G,
    int u )
```

Perform depth first search.

Function to perform depth first search

Parameters

<i>G</i>	Directed graph
<i>u</i>	Vertex

```

00120 {
00121     marked[u] = true;
00122     for (int v : G.getNeighbors(u))
00123     {
00124         if (!marked[v])
00125         {
00126             dfs(G, v);
00127         }
00128     }
00129 }

```

### 3.6.3.2 visited()

```

bool DepthFirstSearch::visited (
    int v ) const

```

Check if a vertex is visited.

Function to check if a vertex is visited

#### Parameters

<i>v</i>	Vertex
----------	--------

#### Returns

true If the vertex is visited

false If the vertex is not visited

```

00139 {
00140     return marked[v];
00141 }

```

## 3.6.4 Member Data Documentation

### 3.6.4.1 marked

```

std::vector<bool> DepthFirstSearch::marked [private]

```

Vector to store the visited vertices

The documentation for this class was generated from the following files:

- [DirectedGraph.hpp](#)
- [DirectedGraph.cpp](#)

## 3.7 DirectedGraph Class Reference

A directed graph.

```

#include <DirectedGraph.hpp>

```

## Public Member Functions

- [DirectedGraph](#) ()  
*Construct a new Directed Graph:: Directed Graph object.*
- [DirectedGraph](#) (int n)  
*Construct a new Directed Graph:: Directed Graph object.*
- void [addEdge](#) (int u, int v)  
*Add an edge to the graph.*
- int [getNumVertices](#) () const  
*Get the number of vertices in the graph.*
- int [getNumEdges](#) () const  
*Get the number of edges in the graph.*
- std::vector< int > [getNeighbors](#) (int u) const  
*Get the neighbors of a vertex.*
- void [writeToFile](#) (std::string filename) const  
*Write the graph information to a file.*

## Private Attributes

- std::vector< std::vector< int > > [adjList](#)

## Friends

- std::ostream & [operator<<](#) (std::ostream &os, const [DirectedGraph](#) &G)  
*Overload the << operator.*

## 3.7.1 Detailed Description

A directed graph.

### Author

acse-sm222

## 3.7.2 Constructor & Destructor Documentation

### 3.7.2.1 DirectedGraph() [1/2]

```
DirectedGraph::DirectedGraph ( ) [inline]
00024 {};
```

### 3.7.2.2 DirectedGraph() [2/2]

```
DirectedGraph::DirectedGraph (
    int n )
```

Construct a new Directed Graph:: Directed Graph object.

Default constructor Constructor taking the number of vertices

**Parameters**

$n$	Number of vertices
-----	--------------------

```

00018 {
00019     adjList.resize(n);
00020 }

```

**3.7.3 Member Function Documentation****3.7.3.1 addEdge()**

```

void DirectedGraph::addEdge (
    int  $u$ ,
    int  $v$  )

```

Add an edge to the graph.

Function to add an edge

**Parameters**

$u$	Source vertex
$v$	Destination vertex

```

00029 {
00030     adjList[ $u$ ].push_back( $v$ );
00031 }

```

**3.7.3.2 getNeighbors()**

```

std::vector< int > DirectedGraph::getNeighbors (
    int  $u$  ) const

```

Get the neighbors of a vertex.

Function to get the neighbors of a vertex

**Parameters**

$u$	Vertex
-----	--------

**Returns**

std::vector<int> Neighbors of a vertex

```

00065 {
00066     return adjList[ $u$ ];
00067 }

```

**3.7.3.3 getNumEdges()**

```

int DirectedGraph::getNumEdges ( ) const

```

Get the number of edges in the graph.

Function to get the number of edges



**Returns**

int Number of edges

```
00049 {
00050     int numEdges = 0;
00051     for (int u = 0; u < adjList.size(); u++)
00052     {
00053         numEdges += adjList[u].size();
00054     }
00055     return numEdges;
00056 }
```

**3.7.3.4 getNumVertices()**

```
int DirectedGraph::getNumVertices ( ) const
```

Get the number of vertices in the graph.

Function to get the number of vertices

**Returns**

int Number of vertices

```
00039 {
00040     return adjList.size();
00041 }
```

**3.7.3.5 writeToFile()**

```
void DirectedGraph::writeToFile (
    std::string filename ) const
```

Write the graph information to a file.

Function to write the graph information to a file

**Parameters**

<i>filename</i>	Filename
-----------------	----------

```
00094 {
00095     std::ofstream file;
00096     file.open(filename);
00097     file << *this;
00098     file.close();
00099 }
```

**3.7.4 Friends And Related Symbol Documentation****3.7.4.1 operator<<**

```
std::ostream & operator<< (
    std::ostream & os,
    const DirectedGraph & G ) [friend]
```

Overload the << operator.

Overload the << operator

## Parameters

<i>os</i>	Output stream
<i>G</i>	Directed graph

## Returns

std::ostream& Output stream

```

00077 {
00078     for (int u = 0; u < G.getNumVertices(); u++)
00079     {
00080         for (int v : G.getNeighbors(u))
00081         {
00082             os << u << " " << v << "\n";
00083         }
00084     }
00085     return os;
00086 }
```

### 3.7.5 Member Data Documentation

#### 3.7.5.1 adjList

```
std::vector<std::vector<int> > DirectedGraph::adjList [private]
```

Adjacency list

The documentation for this class was generated from the following files:

- [DirectedGraph.hpp](#)
- [DirectedGraph.cpp](#)

## 3.8 GA< T > Class Template Reference

Template class representing a Genetic Algorithm ([GA](#)).

```
#include <GeneticAlgorithm.hpp>
```

### Public Member Functions

- [GA](#) ()=default
- [GA](#) (int populationSize, int chromosomeSize, T chromomin, T chromomax, int maxGeneration, double crossoverRate, double mutationRate)  
*Constructor for the [GA](#) class.*
- void [setValidity](#) (const std::function< bool(const std::vector< T > &)> &validity)  
*Sets a validity checker function for ensuring the validity of individuals.*
- void [setFitness](#) (const std::function< double(const std::vector< T > &)> &fitness)  
*Sets a fitness function for evaluating the fitness of individuals.*
- [Individual](#)< T > [optimize](#) ([CrossoverType](#) type)  
*Runs the [GA](#) optimization and returns the best individual found.*
- [~GA](#) ()  
*Destructor for the [GA](#) class.*
- void [toFile](#) (std::string fileDir)  
*Write the best individual to a file.*
- void [mutate](#) ([Individual](#)< T > &child)  
*Performs mutation operation on an individual.*
- int [selectParent](#) (const std::vector< double > &cf)  
*Selects a parent based on the cumulative fitness values.*

### Public Attributes

- T [CHROMOMIN](#)
- T [CHROMOMAX](#)
- int [POP\\_SIZE](#)
- int [CHROMOSOME\\_SIZE](#)
- int [MAX\\_GENERATION](#)
- double [CROSSOVER\\_RATE](#)
- double [MUTATION\\_RATE](#)
- int [ELITISM](#)
- int [CONV\\_GEN](#)
- bool [verbose](#) = false
- std::vector< [Individual](#)< T > > [population](#)

### Private Member Functions

- void [initPopulation](#) ()  
*Initializes the population with random individuals.*
- void [updatePopulation](#) ([CrossoverType](#) type)  
*Updates the population for the next generation.*
- int [tournamentSelection](#) (int k)

### Private Attributes

- std::vector< [Individual](#)< T > > [offspring](#)
- std::vector< double > [fitness](#)
- std::vector< double > [scaledFitness](#)
- std::vector< double > [cumulativeFitness](#)
- int [cross\\_cnt](#)
- int [mut\\_cnt](#)
- double [cross\\_contrib](#)
- double [mut\\_contrib](#)
- std::vector< double > [bestFitnessHistory](#)
- std::function< bool(const std::vector< T > &)> [validity\\_](#)  
*Function object used for checking the validity of individuals in the Genetic Algorithm (GA).*
- std::function< double(const std::vector< T > &)> [fitness\\_](#)  
*Function object used for evaluating the fitness of individuals in the Genetic Algorithm (GA).*

## 3.8.1 Detailed Description

```
template<typename T>
class GA< T >
```

Template class representing a Genetic Algorithm ([GA](#)).

The [GA](#) class encapsulates the operations and parameters required to perform a [GA](#) optimization. The [GA](#) class is completely abstract and can be used for any optimization problem.

## Template Parameters

<i>T</i>	The type of values in the chromosome.
----------	---------------------------------------

## 3.8.2 Constructor &amp; Destructor Documentation

## 3.8.2.1 GA() [1/2]

```
template<typename T >
GA< T >::GA ( ) [default]
```

## 3.8.2.2 GA() [2/2]

```
template<typename T >
GA< T >::GA (
    int populationSize,
    int chromosomeSize,
    T chromomin,
    T chromomax,
    int maxGeneration,
    double crossoverRate,
    double mutationRate )
```

Constructor for the [GA](#) class.

## Parameters

<i>populationSize</i>	The size of the <a href="#">GA</a> population.
<i>chromosomeSize</i>	The size of the chromosome for each individual.
<i>chromomin</i>	The minimum value of the discrete allele of a chromosome.
<i>chromomax</i>	The maximum value of the discrete allele of a chromosome.
<i>maxGeneration</i>	The maximum number of generations for the <a href="#">GA</a> optimization.
<i>crossoverRate</i>	The crossover rate (probability) for creating offspring.
<i>mutationRate</i>	The mutation rate (probability) for mutating the chromosome.

```
00021 {
00022     // GA parameters
00023     CHROMOMIN = chromomin;           // Minimum value of discrete allele
00024     CHROMOMAX = chromomax;           // Maximum value of discrete allele
00025     POP_SIZE = populationSize;       // Population size
00026     CHROMOSOME_SIZE = chromosomeSize; // Chromosome size
00027     MAX_GENERATION = maxGeneration;   // Maximum generation
00028     Crossover_RATE = crossoverRate;    // Crossover rate
00029     MUTATION_RATE = mutationRate;      // Mutation rate
00030     ELITISM = 1;                      // Elitism fraction
00031     CONV_GEN = MAX_GENERATION;         // Convergence generation
00032
00033     // GA variables
00034     population.resize(POP_SIZE);
00035
00036     validity_ = [](const std::vector<T> &chromosome) -> bool
00037     { return true; };
00038     fitness_ = [](const std::vector<T> &chromosome) -> double
00039     { return 0; };
00040 }
```

### 3.8.2.3 ~GA()

```
template<typename T >
GA< T >::~~GA ( ) [inline]
```

Destructor for the GA class.

```
00222 {};
```

## 3.8.3 Member Function Documentation

### 3.8.3.1 initPopulation()

```
template<typename T >
GA< T >::initPopulation [private]
```

Initializes the population with random individuals.

This function creates the initial population for the GA optimization, where each individual's chromosome is randomly generated within the specified range.

```
00056 {
00057     fitness.resize(POP_SIZE);
00058
00059 #ifdef PARALLEL
00060 #pragma omp parallel for default(none) shared(std::cout, population, fitness) schedule(static)
00061 #endif
00062     for (int i = 0; i < POP_SIZE; i++)
00063     {
00064         population[i].chromosome.resize(CHROMOSOME_SIZE);
00065         bool valid = false;
00066         while (!valid)
00067         {
00068             for (int j = 0; j < CHROMOSOME_SIZE; j++)
00069             {
00070                 population[i].chromosome[j] = CHROMOMIN + (CHROMOMAX - CHROMOMIN) * RandomDouble(0.0,
00071 1.0);
00072             }
00073             valid = validity_(population[i].chromosome);
00074             if (verbose) std::cout << "\rGenerated: " << i << " of " << POP_SIZE << std::flush;
00075             population[i].fitness = fitness_(population[i].chromosome);
00076         }
00077         if (verbose) std::cout << std::endl;
00078
00079         std::sort(population.begin(), population.end(), std::greater<Individual<T>>());
00080         for (int i = 0; i < POP_SIZE; i++)
00081         {
00082             fitness[i] = population[i].fitness;
00083         }
00084
00085         std::sort(population.begin(), population.end(), std::greater<Individual<T>>());
00086         std::sort(fitness.begin(), fitness.end(), std::greater<double>());
00087     }
```

### 3.8.3.2 mutate()

```
template<typename T >
GA< T >::mutate (
    Individual< T > & child )
```

Performs mutation operation on an individual.

This function performs mutation on the given individual's chromosome. Mutation introduces small random changes in the chromosome to explore new solutions.

## Parameters

<i>child</i>	The individual to mutate.
--------------	---------------------------

```

00342 {
00343     for (int i = 0; i < CHROMOSOME_SIZE; i++)
00344     {
00345         double p = RandomDouble(0.0, 1.0);
00346         if (p <= MUTATION_RATE)
00347         {
00348             mut_cnt += 1;
00349             T randomGene;
00350             while ((randomGene = CHROMOMIN + (CHROMOMAX - CHROMOMIN) * RandomDouble(0.0, 1.0)) ==
child.chromosome[i])
00351                 ;
00352             child.chromosome[i] = randomGene;
00353         }
00354     }
00355 }

```

## 3.8.3.3 optimize()

```

template<typename T >
GA< T >::optimize (
    CrossoverType type )

```

Runs the [GA](#) optimization and returns the best individual found.

## Returns

The best individual (solution) found by the [GA](#) optimization.

```

00359 {
00360     Individual<T> bestIndividual = population[0];
00361     Individual<T> oldBestIndividual = bestIndividual;
00362     double bestFitness = bestIndividual.fitness;
00363     if (verbose) std::cout << "Initialising population..." << std::endl;
00364     initPopulation();
00365     int conv = 0;
00366     if (verbose) std::cout << "Starting evolution..." << std::endl;
00367     for (int i = 0; i < MAX_GENERATION; i++)
00368     {
00369         updatePopulation(type);
00370         bestIndividual = population[0];
00371         bestFitness = bestIndividual.fitness;
00372         if (verbose)
00373         {
00374             std::cout << "Generation: " << i << std::endl;
00375             std::cout << "Best chromosome: " << std::endl;
00376             for (int k = 0; k < CHROMOSOME_SIZE; ++k)
00377             {
00378                 std::cout << bestIndividual.chromosome[k] << " ";
00379             }
00380             std::cout << " => " << bestIndividual.fitness << std::endl;
00381             std::cout << std::endl;
00382             std::cout << "Worst chromosome: " << std::endl;
00383             for (int k = 0; k < CHROMOSOME_SIZE; ++k)
00384             {
00385                 std::cout << population[POP_SIZE - 1].chromosome[k] << " ";
00386             }
00387             std::cout << " => " << population[POP_SIZE - 1].fitness << std::endl;
00388             std::cout << std::endl;
00389             std::cout << "Adaptive Crossover Rate: " << CROSSOVER_RATE << std::endl;
00390             std::cout << "Adaptive Mutation Rate: " << MUTATION_RATE << std::endl;
00391             std::cout << "-----" << std::endl;
00392             std::cout << std::endl;
00393             if (oldBestIndividual == bestIndividual)
00394             {
00395                 ++conv;
00396             }
00397             bestFitnessHistory.push_back(bestFitness);
00398             if (conv > CONV_GEN)
00399             {
00400                 break;
00401             }
00402         }
00403     }

```

```

00404     {
00405         break; // early stopping criteria
00406     }
00407     oldBestIndividual = bestIndividual;
00408 }
00409
00410 if (verbose) std::cout << "Finshed.." << std::endl;
00411 return bestIndividual;
00412 }

```

### 3.8.3.4 selectParent()

```

template<typename T >
GA< T >::selectParent (
    const std::vector< double > & cf )

```

Selects a parent based on the cumulative fitness values.

This function selects a parent individual from the population based on the cumulative fitness values. The probability of selection is proportional to the fitness value of each individual.

#### Parameters

<i>cf</i>	The cumulative fitness values of the population.
-----------	--

#### Returns

The index of the selected parent individual.

```

00328 {
00329     double p = RandomDouble(0.0, 1.0);
00330     for (int i = 0; i < POP_SIZE; i++)
00331     {
00332         if (p <= cf[i])
00333         {
00334             return i;
00335         }
00336     }
00337     return POP_SIZE - 1;
00338 }

```

### 3.8.3.5 setFitness()

```

template<typename T >
GA< T >::setFitness (
    const std::function< double(const std::vector< T > &) > & fitness )

```

Sets a fitness function for evaluating the fitness of individuals.

The fitness function should take a const reference to the chromosome vector and return a fitness value (double) indicating the quality of the individual.

#### Parameters

<i>fitness</i>	The fitness function.
----------------	-----------------------

```

00050 {
00051     this->fitness_ = fitness;
00052 }

```

### 3.8.3.6 setValidity()

```
template<typename T >
GA< T >::setValidity (
    const std::function< bool(const std::vector< T > &)> & validity )
```

Sets a validity checker function for ensuring the validity of individuals.

The validity checker function should take a const reference to the chromosome vector and return true if the chromosome is valid, and false otherwise.

#### Parameters

<i>validity</i>	The validity checker function.
-----------------	--------------------------------

```
00044 {
00045     this->validity_ = validity;
00046 }
```

### 3.8.3.7 toFile()

```
template<typename T >
GA< T >::toFile (
    std::string fileDir )
```

Write the best individual to a file.

This function grabs the best individual and write its fitness score to an output file.

```
00417 {
00418     std::stringstream fname;
00419     std::fstream fl;
00420     fname << fileDir << "GA_p" << std::to_string(POP_SIZE) << "_n" << std::to_string(CHROMOSOME_SIZE) <<
    "_g" << std::to_string(MAX_GENERATION) << "_c" << std::to_string(CROSSOVER_RATE) << "_m" <<
    std::to_string(MUTATION_RATE) << ".dat";
00421     std::cout << "Saving output to " << fname.str() << std::endl;
00422     fl.open(fname.str().c_str(), std::ios_base::out);
00423
00424     // handle the case when the file cannot be opened
00425     if (fl.fail())
00426     {
00427         std::cout << "Error opening file! with name: " << std::endl;
00428         std::cout.flush();
00429         exit(0);
00430     }
00431     for (int it = 0; it < bestFitnessHistory.size(); ++it)
00432         fl << it << "\t" << bestFitnessHistory[it] << std::endl;
00433     fl.close();
00434 }
```

### 3.8.3.8 tournamentSelection()

```
template<typename T >
int GA< T >::tournamentSelection (
    int k ) [private]
```

#### Parameters

<i>k</i>	
----------	--



## Returns

int

```

00306 {
00307     std::vector<int> candidates(k);
00308     for (int i = 0; i < k; ++i)
00309     {
00310         candidates[i] = RandomInt(0, POP_SIZE - 1);
00311     }
00312     Individual<T> best = population[candidates[0]];
00313     int bestIndex = candidates[0];
00314     for (int i = 1; i < k; ++i)
00315     {
00316         if (population[candidates[i]].fitness > best.fitness)
00317         {
00318             best = population[candidates[i]];
00319             bestIndex = candidates[i];
00320         }
00321     }
00322     return bestIndex;
00323 }
00324 }
```

## 3.8.3.9 updatePopulation()

```

template<typename T >
GA< T >::updatePopulation (
    CrossoverType type ) [private]
```

Updates the population for the next generation.

This function creates the population for the next generation of the GA optimization, using the current population and the generated offspring. It performs selection, crossover, and mutation operations to create new individuals.

```

00091 {
00092     cross_cnt = 0;
00093     mut_cnt = 0;
00094     cross_contrib = 0.0;
00095     mut_contrib = 0.0;
00096
00097     double minFitness = fitness[POP_SIZE - 1];
00098     double maxFitness = fitness[0];
00099     double offset = (minFitness < 0.0) ? -minFitness : 0.0;
00100     double totalFitness = 0.0;
00101     for (int i = 0; i < POP_SIZE; i++)
00102     {
00103         fitness[i] += offset;
00104         totalFitness += fitness[i];
00105     }
00106     scaledFitness.resize(POP_SIZE, 0.0);
00107     for (int i = 0; i < POP_SIZE; i++)
00108     {
00109         scaledFitness[i] = fitness[i] / totalFitness;
00110     }
00111     cumulativeFitness.resize(POP_SIZE, 0.0);
00112     cumulativeFitness[0] = scaledFitness[0];
00113     for (int i = 1; i < POP_SIZE; i++)
00114     {
00115         cumulativeFitness[i] = cumulativeFitness[i - 1] + scaledFitness[i];
00116     }
00117
00118     int remaining = POP_SIZE;
00119     offspring.resize(POP_SIZE);
00120     fitness.resize(POP_SIZE);
00121
00122     // Elitism
00123     for (int i = 0; i < ELITISM; ++i)
00124     {
00125         offspring[i] = population[i];
00126         fitness[i] = offspring[i].fitness;
00127         --remaining;
00128     }
00129
00130 #ifdef PARALLEL
00131 #pragma omp parallel for default(none) shared(population, offspring, fitness, cumulativeFitness, type)
00132     reduction(+ : cross_cnt, mut_cnt, cross_contrib, mut_contrib) schedule(static)
00133     for (int i = ELITISM; i < POP_SIZE; ++i)
00134     {
00135         bool validChild1 = false;
```

```

00135         bool validChild2 = false;
00136         Individual<T> child1;
00137         child1.chromosome.resize(CHROMOSOME_SIZE);
00138         Individual<T> child2;
00139         child2.chromosome.resize(CHROMOSOME_SIZE);
00140
00141         while (!validChild1 || !validChild2)
00142         {
00143
00144             int parent1 = selectParent(cumulativeFitness);
00145             int parent2 = selectParent(cumulativeFitness);
00146             // int parent1 = tournamentSelection(10);
00147             // int parent2 = tournamentSelection(10);
00148
00149             double crossoverProb = RandomDouble(0.0, 1.0);
00150             if (crossoverProb > CROSSOVER_RATE)
00151             {
00152                 Crossover<T>::crossover(population[parent1], population[parent2], child1, child2,
type);
00153 #ifdef ADAPTIVE
00154                 cross_cnt++;
00155                 child1.fitness = fitness_(child1.chromosome);
00156                 child2.fitness = fitness_(child2.chromosome);
00157                 cross_contrib += child1.fitness + child2.fitness - population[parent1].fitness -
population[parent2].fitness;
00158 #endif
00159             }
00160             else
00161             {
00162                 child1 = population[parent1];
00163                 child2 = population[parent2];
00164             }
00165             double fit_before = child1.fitness + child2.fitness;
00166             mutate(child1);
00167             mutate(child2);
00168 #ifdef ADAPTIVE
00169             child1.fitness = fitness_(child1.chromosome);
00170             child2.fitness = fitness_(child2.chromosome);
00171             mut_contrib += child1.fitness + child2.fitness - fit_before;
00172 #endif
00173             validChild1 = validity_(child1.chromosome);
00174             validChild2 = validity_(child2.chromosome);
00175         }
00176
00177         if (validChild1 && !validChild2)
00178         {
00179 #ifndef ADAPTIVE
00180             child1.fitness = fitness_(child1.chromosome);
00181 #endif
00182             offspring[i] = child1;
00183             fitness[i] = offspring[i].fitness;
00184         }
00185         if (validChild2 && !validChild1)
00186         {
00187 #ifndef ADAPTIVE
00188             child2.fitness = fitness_(child2.chromosome);
00189 #endif
00190             offspring[i] = child2;
00191             fitness[i] = offspring[i].fitness;
00192         }
00193         else
00194         {
00195 #ifndef ADAPTIVE
00196             child1.fitness = fitness_(child1.chromosome);
00197             child2.fitness = fitness_(child2.chromosome);
00198 #endif
00199             if (child1.fitness > child2.fitness)
00200             {
00201                 offspring[i] = child1;
00202                 fitness[i] = offspring[i].fitness;
00203             }
00204             else
00205             {
00206                 offspring[i] = child2;
00207                 fitness[i] = offspring[i].fitness;
00208             }
00209         }
00210     }
00211 #endif
00212 #endif
00213 #ifdef SERIAL
00214     // Crossover and mutation
00215     while (remaining > 0)
00216     {
00217         bool validChild1 = false;
00218         Individual<T> child1;

```

```

00220         child1.chromosome.resize(CHROMOSOME_SIZE);
00221
00222         bool validChild2 = false;
00223         Individual<T> child2;
00224         child2.chromosome.resize(CHROMOSOME_SIZE);
00225
00226         int parent1 = selectParent(cumulativeFitness);
00227         int parent2 = selectParent(cumulativeFitness);
00228         // int parent1 = tournamentSelection(10);
00229         // int parent2 = tournamentSelection(10);
00230
00231         double crossoverProb = RandomDouble(0.0, 1.0);
00232         if (crossoverProb > CROSSOVER_RATE)
00233         {
00234             Crossover<T>::crossover(population[parent1], population[parent2], child1, child2, type);
00235 #ifdef ADAPTIVE
00236             cross_cnt++;
00237             child1.fitness = fitness_(child1.chromosome);
00238             child2.fitness = fitness_(child2.chromosome);
00239             cross_contrib += child1.fitness + child2.fitness - population[parent1].fitness -
population[parent2].fitness;
00240 #endif
00241         }
00242         else
00243         {
00244             child1 = population[parent1];
00245             child2 = population[parent2];
00246         }
00247         double fit_before = child1.fitness + child2.fitness;
00248         mutate(child1);
00249         mutate(child2);
00250 #ifdef ADAPTIVE
00251         child1.fitness = fitness_(child1.chromosome);
00252         child2.fitness = fitness_(child2.chromosome);
00253         mut_contrib += child1.fitness + child2.fitness - fit_before;
00254 #endif
00255         validChild1 = validity_(child1.chromosome);
00256         validChild2 = validity_(child2.chromosome);
00257
00258         if (validChild1 && remaining > 0)
00259         {
00260 #ifndef ADAPTIVE
00261             child1.fitness = fitness_(child1.chromosome);
00262 #endif
00263             offspring[POP_SIZE - remaining] = child1;
00264             fitness[POP_SIZE - remaining] = offspring[POP_SIZE - remaining].fitness;
00265             --remaining;
00266         }
00267         if (validChild2 && remaining > 0)
00268         {
00269 #ifndef ADAPTIVE
00270             child2.fitness = fitness_(child2.chromosome);
00271 #endif
00272             offspring[POP_SIZE - remaining] = child2;
00273             fitness[POP_SIZE - remaining] = offspring[POP_SIZE - remaining].fitness;
00274             --remaining;
00275         }
00276     }
00277 #endif
00278
00279 #ifdef ADAPTIVE
00280         double cm = cross_contrib / cross_cnt;
00281         double mm = mut_contrib / mut_cnt;
00282         double theta = 0.1 * (maxFitness - totalFitness / POP_SIZE) / (maxFitness - totalFitness);
00283         if (cm > 0.0)
00284         {
00285             CROSSOVER_RATE = std::min(1.0, CROSSOVER_RATE + theta);
00286             MUTATION_RATE = std::max(0.0, MUTATION_RATE - theta);
00287         }
00288         else
00289         {
00290             CROSSOVER_RATE = std::max(0.0, CROSSOVER_RATE - theta);
00291             MUTATION_RATE = std::min(1.0, MUTATION_RATE + theta);
00292         }
00293 #endif
00294         // Update population
00295         population = offspring;
00296
00297         std::sort(population.begin(), population.end(), std::greater<Individual<T>>());
00298         for (int i = 0; i < POP_SIZE; i++)
00299         {
00300             fitness[i] = population[i].fitness;
00301         }
00302 }

```

## 3.8.4 Member Data Documentation

### 3.8.4.1 bestFitnessHistory

```
template<typename T >
std::vector<double> GA< T >::bestFitnessHistory [private]
```

### 3.8.4.2 CHROMOMAX

```
template<typename T >
T GA< T >::CHROMOMAX
```

### 3.8.4.3 CHROMOMIN

```
template<typename T >
T GA< T >::CHROMOMIN
```

### 3.8.4.4 CHROMOSOME\_SIZE

```
template<typename T >
int GA< T >::CHROMOSOME_SIZE
```

### 3.8.4.5 CONV\_GEN

```
template<typename T >
int GA< T >::CONV_GEN
```

### 3.8.4.6 cross\_cnt

```
template<typename T >
int GA< T >::cross_cnt [private]
```

### 3.8.4.7 cross\_contrib

```
template<typename T >
double GA< T >::cross_contrib [private]
```

### 3.8.4.8 CROSSOVER\_RATE

```
template<typename T >
double GA< T >::CROSSOVER_RATE
```

### 3.8.4.9 cumulativeFitness

```
template<typename T >
std::vector<double> GA< T >::cumulativeFitness [private]
```

### 3.8.4.10 ELITISM

```
template<typename T >
int GA< T >::ELITISM
```

### 3.8.4.11 fitness

```
template<typename T >
std::vector<double> GA< T >::fitness [private]
```

### 3.8.4.12 fitness\_

```
template<typename T >
std::function<double(const std::vector<T> &)> GA< T >::fitness_ [private]
```

Function object used for evaluating the fitness of individuals in the Genetic Algorithm (GA).

The fitness function is a user-defined function that evaluates the fitness of an individual's chromosome, indicating the quality of the individual's solution. It takes a const reference to the chromosome vector and returns a fitness value (double).

#### Template Parameters

<i>T</i>	The type of values in the chromosome.
----------	---------------------------------------

#### Parameters

<i>chromosome</i>	The chromosome vector for which to evaluate the fitness.
-------------------	--

#### Returns

The fitness value of the chromosome.

### 3.8.4.13 MAX\_GENERATION

```
template<typename T >
int GA< T >::MAX_GENERATION
```

### 3.8.4.14 mut\_cnt

```
template<typename T >
int GA< T >::mut_cnt [private]
```

**3.8.4.15 mut\_contrib**

```
template<typename T >
double GA< T >::mut_contrib [private]
```

**3.8.4.16 MUTATION\_RATE**

```
template<typename T >
double GA< T >::MUTATION_RATE
```

**3.8.4.17 offspring**

```
template<typename T >
std::vector<Individual<T> > GA< T >::offspring [private]
```

**3.8.4.18 POP\_SIZE**

```
template<typename T >
int GA< T >::POP_SIZE
```

**3.8.4.19 population**

```
template<typename T >
std::vector<Individual<T> > GA< T >::population
```

**3.8.4.20 scaledFitness**

```
template<typename T >
std::vector<double> GA< T >::scaledFitness [private]
```

**3.8.4.21 validity\_**

```
template<typename T >
std::function<bool(const std::vector<T> &)> GA< T >::validity_ [private]
```

Function object used for checking the validity of individuals in the Genetic Algorithm ([GA](#)).

The validity function is a user-defined function that determines whether an individual's chromosome is valid or not. It takes a const reference to the chromosome vector and returns a boolean value: true if the chromosome is valid, and false otherwise.

**Template Parameters**

<i>T</i>	The type of values in the chromosome.
----------	---------------------------------------

## Parameters

<i>chromosome</i>	The chromosome vector to be checked for validity.
-------------------	---

## Returns

True if the chromosome is valid, false otherwise.

## 3.8.4.22 verbose

```
template<typename T >
bool GA< T >::verbose = false
```

The documentation for this class was generated from the following files:

- [GeneticAlgorithm.hpp](#)
- [GeneticAlgorithm.cpp](#)

## 3.9 Individual< T > Class Template Reference

Template class representing an individual in the [GA](#) population.

```
#include <GeneticAlgorithm.hpp>
```

## Public Member Functions

- [Individual](#) ()  
*Default constructor for the [Individual](#) class.*
- [Individual](#) (const std::vector< T > &chromosome)  
*Constructs an [Individual](#) with the given chromosome.*
- bool [operator<](#) (const [Individual](#) &ind) const  
*Overloads the less-than operator for comparing individuals based on fitness.*
- bool [operator>](#) (const [Individual](#) &ind) const  
*Overloads the greater-than operator for comparing individuals based on fitness.*
- bool [operator==](#) (const [Individual](#) &ind) const  
*Overloads the equality operator for comparing individuals based on their chromosomes.*
- [Individual](#) & [operator=](#) (const [Individual](#) &ind)  
*Overloads the assignment operator for assigning the values of another [Individual](#) to this [Individual](#).*

## Public Attributes

- std::vector< T > [chromosome](#)
- double [fitness](#)

### 3.9.1 Detailed Description

```
template<typename T>
class Individual< T >
```

Template class representing an individual in the [GA](#) population.

An [Individual](#) consists of a chromosome (a vector of a specific type), representing its genetic information, and a fitness value indicating the quality of the individual's solution.

### Template Parameters

<i>T</i>	The type of values in the chromosome.
----------	---------------------------------------

## 3.9.2 Constructor & Destructor Documentation

### 3.9.2.1 Individual() [1/2]

```
template<typename T >
Individual< T >::Individual ( ) [inline]
```

Default constructor for the `Individual` class.  
00072 {}

### 3.9.2.2 Individual() [2/2]

```
template<typename T >
Individual< T >::Individual (
    const std::vector< T > & chromosome ) [inline]
```

Constructs an `Individual` with the given chromosome.

#### Parameters

<i>chromosome</i>	The chromosome representing the genetic information.
-------------------	--

```
00079 : chromosome(chromosome) {}
```

## 3.9.3 Member Function Documentation

### 3.9.3.1 operator<()

```
template<typename T >
bool Individual< T >::operator< (
    const Individual< T > & ind ) const [inline]
```

Overloads the less-than operator for comparing individuals based on fitness.

#### Parameters

<i>ind</i>	The <code>Individual</code> to compare.
------------	---

#### Returns

True if this `Individual` has a lower fitness value than the other `Individual`, false otherwise.

```
00089     {
00090         return fitness < ind.fitness;
00091     }
```



### 3.9.3.2 operator=()

```
template<typename T >
Individual & Individual< T >::operator= (
    const Individual< T > & ind ) [inline]
```

Overloads the assignment operator for assigning the values of another [Individual](#) to this [Individual](#).

#### Parameters

<i>ind</i>	The <a href="#">Individual</a> to assign.
------------	---

#### Returns

Reference to the assigned [Individual](#).

```
00122     {
00123         chromosome.resize(ind.chromosome.size());
00124         chromosome = ind.chromosome;
00125         fitness = ind.fitness;
00126         return *this;
00127     }
```

### 3.9.3.3 operator==( )

```
template<typename T >
bool Individual< T >::operator==(
    const Individual< T > & ind ) const [inline]
```

Overloads the equality operator for comparing individuals based on their chromosomes.

#### Parameters

<i>ind</i>	The <a href="#">Individual</a> to compare.
------------	--

#### Returns

True if the chromosomes of the two Individuals are equal, false otherwise.

```
00111     {
00112         return chromosome == ind.chromosome;
00113     }
```

### 3.9.3.4 operator>()

```
template<typename T >
bool Individual< T >::operator> (
    const Individual< T > & ind ) const [inline]
```

Overloads the greater-than operator for comparing individuals based on fitness.

#### Parameters

<i>ind</i>	The <a href="#">Individual</a> to compare.
------------	--

**Returns**

True if this [Individual](#) has a higher fitness value than the other [Individual](#), false otherwise.

```
00100    {
00101        return fitness > ind.fitness;
00102    }
```

**3.9.4 Member Data Documentation****3.9.4.1 chromosome**

```
template<typename T >
std::vector<T> Individual< T >::chromosome
```

**3.9.4.2 fitness**

```
template<typename T >
double Individual< T >::fitness
```

The documentation for this class was generated from the following file:

- [GeneticAlgorithm.hpp](#)

**3.10 Stream Class Reference**

A [Stream](#) class to store the stream information.

```
#include <CUnit.h>
```

**Public Member Functions**

- [Stream](#) ()
- [Stream](#) (double GER, double WASTE)
- [Stream operator+](#) (const [Stream](#) &s)
- void [operator+=](#) (const [Stream](#) &s)
- [~Stream](#) ()

**Public Attributes**

- double [ger](#)
- double [waste](#)

**3.10.1 Detailed Description**

A [Stream](#) class to store the stream information.

**Author**

acse-sm222

## 3.10.2 Constructor & Destructor Documentation

### 3.10.2.1 Stream() [1/2]

```
Stream::Stream ( ) [inline]
00024     {
00025         ger = 0;
00026         waste = 0;
00027     };
```

### 3.10.2.2 Stream() [2/2]

```
Stream::Stream (
    double GER,
    double WASTE ) [inline]
00028 : ger(GER), waste(WASTE){};
```

### 3.10.2.3 ~Stream()

```
Stream::~~Stream ( ) [inline]
```

Destructor

## 3.10.3 Member Function Documentation

### 3.10.3.1 operator+()

```
Stream Stream::operator+ (
    const Stream & s ) [inline]
```

Parameters

<i>s</i>	Overload the + operator
----------	-------------------------

```
00032     {
00033         Stream stream;
00034         stream.ger = this->ger + s.ger;
00035         stream.waste = this->waste + s.waste;
00036         return stream;
00037     }
```

### 3.10.3.2 operator+=()

```
void Stream::operator+= (
    const Stream & s ) [inline]
```

Parameters

<i>s</i>	Overload the += operator
----------	--------------------------

```
00040     {
00041         this->ger += s.ger;
```

```
00042         this->waste += s.waste;
00043     }
```

### 3.10.4 Member Data Documentation

#### 3.10.4.1 ger

```
double Stream::ger
```

Constructor taking the GER and WASTE Germanium flow rate

#### 3.10.4.2 waste

```
double Stream::waste
```

Waste flow rate

The documentation for this class was generated from the following file:

- [CUnit.h](#)

## 3.11 Units Class Reference

### Public Member Functions

- [Units](#) ()
- [Units](#) (int [conc\\_num](#), int [tails\\_num](#))

### Public Attributes

- int [conc\\_num](#)
- int [tails\\_num](#)
- bool [marked](#)

### 3.11.1 Constructor & Destructor Documentation

#### 3.11.1.1 Units() [1/2]

```
Units::Units ( ) [inline]
00016 {};
```

#### 3.11.1.2 Units() [2/2]

```
Units::Units (
    int conc_num,
    int tails_num ) [inline]
00017 : conc_num(conc_num), tails_num(tails_num) {};
```

### 3.11.2 Member Data Documentation

#### 3.11.2.1 conc\_num

```
int Units::conc_num
```

#### 3.11.2.2 marked

```
bool Units::marked
```

#### 3.11.2.3 tails\_num

```
int Units::tails_num
```

The documentation for this class was generated from the following file:

- [CCircuit.cpp](#)

## 3.12 UnitTestCircuit Class Reference

```
#include <Unitttests.h>
```

### Public Member Functions

- [UnitTestCircuit](#) ()  
*Default constructor.*
- [~UnitTestCircuit](#) ()  
*Destructor.*
- void [test\\_circuit\\_evaluation](#) ()
- void [test\\_circuit\\_validity](#) ()
- void [test\\_circuit\\_unit](#) ()
- void [run](#) ()

### 3.12.1 Constructor & Destructor Documentation

#### 3.12.1.1 UnitTestCircuit()

```
UnitTestCircuit::UnitTestCircuit ( )
```

Default constructor.

```
00006 {}
```

### 3.12.1.2 ~UnitTestCircuit()

```
UnitTestCircuit::~~UnitTestCircuit ( )
```

Destructor.

```
00007 {}
```

## 3.12.2 Member Function Documentation

### 3.12.2.1 run()

```
void UnitTestCircuit::run ( )
00072 {
00073     test_circuit_evaluation();
00074     test_circuit_validity();
00075     test_circuit_unit();
00076 }
```

### 3.12.2.2 test\_circuit\_evaluation()

```
void UnitTestCircuit::test_circuit_evaluation ( )
00009 {
00010     std::cout << "=====Test Circuit Evaluation=====\\n";
00011
00012     std::vector<int> circuit_vector = {4, 5, 1, 2, 4, 0, 1, 1, 6, 1, 3};
00013     bool validity = Circuit::Check_Validity(circuit_vector);
00014     assert(validity==true);
00015
00016     Circuit_Parameters parameters;
00017     parameters.tolerance = 1e-6;
00018
00019     double value = Evaluate_Circuit(circuit_vector, parameters);
00020     std::cout << value << std::endl;
00021     assert(std::fabs(value - 107.204) <= 1e-3);
00022     std::cout << "completed" << std::endl;
00023 }
```

### 3.12.2.3 test\_circuit\_unit()

```
void UnitTestCircuit::test_circuit_unit ( )
00048 {
00049     std::cout << "=====Test Circuit Unit=====\\n";
00050
00051     CUnit test_unit;
00052
00053     test_unit.input.ger = 20;
00054     test_unit.input.waste = 80;
00055
00056     test_unit.calculateOutputFlowRates(test_unit.input);
00057
00058     double eps = 1e-3;
00059
00060     bool check1 = fabs(test_unit.conc.ger - 2.6087) < eps;
00061     bool check2 = fabs(test_unit.conc.waste - 1.18227) < eps;
00062     bool check3 = fabs(test_unit.tail.ger - 17.3913) < eps;
00063     bool check4 = fabs(test_unit.tail.waste - 78.8177) < eps;
00064
00065     assert(check1==true);
00066     assert(check2==true);
00067     assert(check3==true);
00068     assert(check4==true);
00069     std::cout << "completed" << std::endl;
00070 }
```

### 3.12.2.4 test\_circuit\_validity()

```

void UnitTestCircuit::test_circuit_validity ( )
00025     {
00026         std::cout << "=====Test Circuit Validity=====\\n";
00027
00028         // Cases when there are 3 units
00029         // Case 1: self-recycle
00030         std::vector<int> circuit_vector1 = {0, 1, 2, 0, 2, 1, 0};
00031         // Case 2: points to itself
00032         std::vector<int> circuit_vector2 = {0, 1, 2, 1, 2, 3, 4};
00033         // Case 3: there are units not ending up in 2 destinations
00034         std::vector<int> circuit_vector3 = {0, 1, 1, 3, 2, 1, 4};
00035         // Case 4: there is a unit not accessible from the feed
00036         std::vector<int> circuit_vector4 = {0, 1, 4, 3, 4, 3, 0};
00037         bool check1 = Circuit::Check_VValidity(circuit_vector1);
00038         bool check2 = Circuit::Check_VValidity(circuit_vector2);
00039         bool check3 = Circuit::Check_VValidity(circuit_vector3);
00040         bool check4 = Circuit::Check_VValidity(circuit_vector4);
00041         assert(check1==false);
00042         assert(check2==false);
00043         assert(check3==false);
00044         assert(check4==false);
00045         std::cout << "completed" << std::endl;
00046     }

```

The documentation for this class was generated from the following files:

- [Unittests.h](#)
- [test\\_Circuit.cpp](#)

## 3.13 UnitTestCrossover Class Reference

```
#include <Unittests.h>
```

### Public Member Functions

- [UnitTestCrossover \(\)](#)  
*Default constructor.*
- [~UnitTestCrossover \(\)](#)  
*Destructor.*
- void [test\\_one\\_pt\\_crossover \(\)](#)
- void [test\\_two\\_pts\\_crossover \(\)](#)
- void [test\\_mult\\_pts\\_crossover \(\)](#)
- void [test\\_uniform\\_crossover \(\)](#)

### 3.13.1 Constructor & Destructor Documentation

#### 3.13.1.1 UnitTestCrossover()

```
UnitTestCrossover::UnitTestCrossover ( )
```

Default constructor.

```

00022 {
00023     std::cout << "                                Test Crossover Start
        \\n";
00024 }

```

### 3.13.1.2 ~UnitTestCrossover()

```
UnitTestCrossover::~~UnitTestCrossover ( )
```

Destructor.

```
00025 {}
```

## 3.13.2 Member Function Documentation

### 3.13.2.1 test\_mult\_pts\_crossover()

```
void UnitTestCrossover::test_mult_pts_crossover ( )
00100 {
00101     std::cout << "=====Test Multi-Points Crossover
method=====\\n";
00102
00103     std::vector<int> chromosome1 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
00104     std::vector<int> chromosome2 = {10, 11, 12, 13, 14, 15, 16, 17, 18, 19};
00105
00106     Individual<int> parent1, parent2, child1, child2;
00107
00108     parent1.chromosome = chromosome1;
00109     parent2.chromosome = chromosome2;
00110
00111     child1.chromosome.resize(chromosome1.size());
00112     child2.chromosome.resize(chromosome2.size());
00113
00114     crossover.mult_pts_crossover(parent1, parent2, child1, child2);
00115
00116     std::cout << "Parent 1: ";
00117     printChromosome(parent1.chromosome);
00118
00119     std::cout << "Parent 2: ";
00120     printChromosome(parent2.chromosome);
00121
00122     std::cout << "Child 1: ";
00123     printChromosome(child1.chromosome);
00124
00125     std::cout << "Child 2: ";
00126     printChromosome(child2.chromosome);
00127
00128     // Check that the chromosomes have been altered
00129     assert(child1.chromosome != parent1.chromosome);
00130     assert(child1.chromosome != parent2.chromosome);
00131     assert(child2.chromosome != parent1.chromosome);
00132     assert(child2.chromosome != parent2.chromosome);
00133 }
```

### 3.13.2.2 test\_one\_pt\_crossover()

```
void UnitTestCrossover::test_one_pt_crossover ( )
00028 {
00029     std::cout << "=====Test One-Point Crossover
method=====\\n";
00030
00031     std::vector<int> chromosome1 = {1, 2, 3, 4, 5};
00032     std::vector<int> chromosome2 = {6, 7, 8, 9, 10};
00033
00034     Individual<int> parent1, parent2, child1, child2;
00035
00036     parent1.chromosome = chromosome1;
00037     parent2.chromosome = chromosome2;
00038
00039     child1.chromosome.resize(chromosome1.size());
00040     child2.chromosome.resize(chromosome2.size());
00041
00042     crossover.one_pt_crossover(parent1, parent2, child1, child2);
00043
00044     std::cout << "Parent 1: ";
00045     printChromosome(parent1.chromosome);
00046
00047     std::cout << "Parent 2: ";
00048     printChromosome(parent2.chromosome);
```



```

00049
00050     std::cout << "Child 1: ";
00051     printChromosome(child1.chromosome);
00052
00053     std::cout << "Child 2: ";
00054     printChromosome(child2.chromosome);
00055
00056     // Check that the chromosomes have been altered
00057     assert(child1.chromosome != parent1.chromosome);
00058     assert(child1.chromosome != parent2.chromosome);
00059     assert(child2.chromosome != parent1.chromosome);
00060     assert(child2.chromosome != parent2.chromosome);
00061 }

```

### 3.13.2.3 test\_two\_pts\_crossover()

```

void UnitTestCrossover::test_two_pts_crossover ( )
00064 {
00065     std::cout << "=====Test Two-Points Crossover
method=====\\n";
00066
00067     std::vector<int> chromosome1 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
00068     std::vector<int> chromosome2 = {10, 11, 12, 13, 14, 15, 16, 17, 18, 19};
00069
00070     Individual<int> parent1, parent2, child1, child2;
00071
00072     parent1.chromosome = chromosome1;
00073     parent2.chromosome = chromosome2;
00074
00075     child1.chromosome.resize(chromosome1.size());
00076     child2.chromosome.resize(chromosome2.size());
00077
00078     crossover.two_pts_crossover(parent1, parent2, child1, child2);
00079
00080     std::cout << "Parent 1: ";
00081     printChromosome(parent1.chromosome);
00082
00083     std::cout << "Parent 2: ";
00084     printChromosome(parent2.chromosome);
00085
00086     std::cout << "Child 1: ";
00087     printChromosome(child1.chromosome);
00088
00089     std::cout << "Child 2: ";
00090     printChromosome(child2.chromosome);
00091
00092     // Check that the chromosomes have been altered
00093     assert(child1.chromosome != parent1.chromosome);
00094     assert(child1.chromosome != parent2.chromosome);
00095     assert(child2.chromosome != parent1.chromosome);
00096     assert(child2.chromosome != parent2.chromosome);
00097 }

```

### 3.13.2.4 test\_uniform\_crossover()

```

void UnitTestCrossover::test_uniform_crossover ( )
00136 {
00137     std::cout << "=====Test Uniform Crossover
method=====\\n";
00138
00139     std::vector<int> chromosome1 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
00140     std::vector<int> chromosome2 = {9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
00141     std::vector<int> expected_chromosome1 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
00142     std::vector<int> expected_chromosome2 = {9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
00143
00144     Individual<int> parent1, parent2, child1, child2;
00145
00146     parent1.chromosome = chromosome1;
00147     parent2.chromosome = chromosome2;
00148
00149     child1.chromosome.resize(chromosome1.size());
00150     child2.chromosome.resize(chromosome2.size());
00151
00152     crossover.uniform_crossover(parent1, parent2, child1, child2);
00153
00154     // Check that the chromosomes have been altered
00155     assert(child1.chromosome != expected_chromosome1);
00156     assert(child2.chromosome != expected_chromosome2);

```

```
00157     std::cout << "Results match expected outputs.\n";
00158 }
```

The documentation for this class was generated from the following files:

- [Unittests.h](#)
- [test\\_Crossover.cpp](#)

## 3.14 UnitTestGA Class Reference

```
#include <Unittests.h>
```

### Public Member Functions

- [UnitTestGA](#) (int populationSize, int chromosomeSize, int chromomin, int chromomax, int maxGeneration, double crossoverRate, double mutationRate)  
*Constructor for the UnitTestGeneticAlgorithm class.*
- [~UnitTestGA](#) ()  
*Destructor for the UnitTestGeneticAlgorithm class..*
- void [test\\_mutation](#) ()
- void [test\\_select\\_parent](#) ()
- void [test\\_global\\_optimum](#) ()  
*Test with Rosenbrock function to check if the algorithm is capable in finding the global optimum successfully.*
- void [run](#) ()

### Public Attributes

- int [POPULATION\\_SIZE](#) = 100
- int [CHROMOSOME\\_SIZE](#) = 10
- int [CHROMOMIN](#) = 0
- int [CHROMOMAX](#) = 10
- int [MAX\\_GENERATION](#) = 2000
- int [CONV\\_GEN](#) = 2000
- double [CROSSOVER\\_RATE](#) = 0.8
- double [MUTATION\\_RATE](#) = 0.15
- [GA](#)< int > \* [ga](#)

### 3.14.1 Constructor & Destructor Documentation

#### 3.14.1.1 UnitTestGA()

```
UnitTestGA::UnitTestGA (
    int populationSize,
    int chromosomeSize,
    int chromomin,
    int chromomax,
    int maxGeneration,
    double crossoverRate,
    double mutationRate ) [inline]
```

Constructor for the UnitTestGeneticAlgorithm class.

```
00037     : POPULATION\_SIZE(populationSize), CHROMOSOME\_SIZE(chromosomeSize),
    CHROMOMIN(chromomin),
00038     CHROMOMAX(chromomax), MAX\_GENERATION(maxGeneration), CROSSOVER\_RATE(crossoverRate),
00039     MUTATION\_RATE(mutationRate) {};
```

### 3.14.1.2 ~UnitTestGA()

```
UnitTestGA::~~UnitTestGA ( )
```

Destructor for the UnitTestGeneticAlgorithm class..

```
00023 {};
```

## 3.14.2 Member Function Documentation

### 3.14.2.1 run()

```
void UnitTestGA::run ( )
00073     {
00074     test_mutation();
00075     test_select_parent();
00076     test_global_optimum();
00077 }
```

### 3.14.2.2 test\_global\_optimum()

```
void UnitTestGA::test_global_optimum ( )
```

Test with Rosenbrock function to check if the algorithm is capable in finding the global optimum successfully.

```
00047     {
00048     std::cout << "=====Test Global Optimum=====\\n";
00049     // configure GA
00050     ga->setValidity(validity);
00051     ga->setFitness(Rosenbrock);
00052     ga->verbose = false;
00053     CrossoverType type = CrossoverType::UNIFORM;
00054     // Known global optimum for Rosenbrock function
00055     std::vector<int> expected_chromosome = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1};
00056     int success_count = 0;
00057     int num_runs = 10;
00058
00059     // Run for 10 times to see if the algorithm can find the global optimum
00060     // if 50% of the time can find the global optimum, then the test is passed
00061     for (int i = 0; i < num_runs; i++) {
00062         auto best = ga->optimize(type);
00063         if (best.chromosome == expected_chromosome) {
00064             success_count++;
00065         }
00066     }
00067     assert(success_count >= num_runs / 2);
00068     std::cout << "successfully found global optimum for Rosenbrock: " << success_count;
00069     std::cout << "/" << num_runs << " times!" << std::endl;
00070     std::cout << "completed" << std::endl;
00071 }
```

### 3.14.2.3 test\_mutation()

```
void UnitTestGA::test_mutation ( )
00025     {
00026     std::cout << "=====Test Mutation=====\\n";
00027     std::vector<int> init_chromosome = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
00028     Individual<int> init_individual(init_chromosome);
00029     Individual<int> result_individual(init_chromosome);
00030     ga->MUTATION_RATE = 0.5;
00031
00032     ga->mutate(result_individual);
00033     assert(init_individual.chromosome != result_individual.chromosome);
00034     std::cout << "completed" << std::endl;
00035 }
```

### 3.14.2.4 test\_select\_parent()

```
void UnitTestGA::test_select_parent ( )
00037     {
00038     std::cout << "=====Test Select Parent=====\\n";
00039     int result;
00040     std::vector<double> cumulative_fitness = {-1.0, 1.0, 1.0};
00041
00042     result = ga->selectParent(cumulative_fitness);
00043     assert(result == 1);
00044     std::cout << "completed" << std::endl;
00045 }
```

## 3.14.3 Member Data Documentation

### 3.14.3.1 CHROMOMAX

```
int UnitTestGA::CHROMOMAX = 10
```

### 3.14.3.2 CHROMOMIN

```
int UnitTestGA::CHROMOMIN = 0
```

### 3.14.3.3 CHROMOSOME\_SIZE

```
int UnitTestGA::CHROMOSOME_SIZE = 10
```

### 3.14.3.4 CONV\_GEN

```
int UnitTestGA::CONV_GEN = 2000
```

### 3.14.3.5 CROSSOVER\_RATE

```
double UnitTestGA::CROSSOVER_RATE = 0.8
```

### 3.14.3.6 ga

```
GA<int>* UnitTestGA::ga
```

#### Initial value:

```
= new GA<int>(POPULATION_SIZE,
              CHROMOSOME_SIZE,
              CHROMOMIN,
              CHROMOMAX,
              MAX_GENERATION,
              CROSSOVER_RATE,
              MUTATION_RATE
            )
```

### 3.14.3.7 MAX\_GENERATION

```
int UnitTestGA::MAX_GENERATION = 2000
```

### 3.14.3.8 MUTATION\_RATE

```
double UnitTestGA::MUTATION_RATE = 0.15
```

### 3.14.3.9 POPULATION\_SIZE

```
int UnitTestGA::POPULATION_SIZE = 100
```

The documentation for this class was generated from the following files:

- [Unittests.h](#)
- [test\\_GeneticAlgorithm.cpp](#)

## 3.15 UnitTestIndividual Class Reference

```
#include <Unittests.h>
```

### Public Member Functions

- [UnitTestIndividual](#) ()  
*Default constructor.*
- [~UnitTestIndividual](#) ()  
*Destructor.*
- void [test\\_constructor](#) ()  
*Test the constructor and getter functions.*
- void [test\\_Get\\_Set\\_fitness](#) ()  
*Test the calculate fitness value function.*
- void [test\\_Get\\_numUnits](#) ()  
*Test the getter function for num\_units.*
- void [test\\_Get\\_vectorLength](#) ()  
*Test the getter function for vector\_length.*

### Protected Attributes

- [Individual](#)< int > [individual](#)

## 3.15.1 Constructor & Destructor Documentation

### 3.15.1.1 UnitTestIndividual()

```
UnitTestIndividual::UnitTestIndividual ( )
```

Default constructor.

```
00007                                     {
00008     std::cout << "Test Individual begins" << std::endl;
00009 }
```

### 3.15.1.2 ~UnitTestIndividual()

```
UnitTestIndividual::~~UnitTestIndividual ( )
```

Destructor.

```
00010 {
00011     std::cout << "Test Individual completed" << std::endl;
00012 }
```

## 3.15.2 Member Function Documentation

### 3.15.2.1 test\_constructor()

```
void UnitTestIndividual::test_constructor ( )
```

Test the constructor and getter functions.

```
00014 {
00015     int chromosomeLength = 10;
00016     individual = Individual(chromosomeLength, 0, 10);
00017
00018     assert(individual.Get_vectorLength() == 10);
00019     assert(individual.chromosome.size() == 10);
00020 }
```

### 3.15.2.2 test\_Get\_numUnits()

```
void UnitTestIndividual::test_Get_numUnits ( )
```

Test the getter function for num\_units.

### 3.15.2.3 test\_Get\_Set\_fitness()

```
void UnitTestIndividual::test_Get_Set_fitness ( )
```

Test the calculate fitness value function.

Test the getter and setter function for fitness.

```
00029 {
00030     std::vector<int> fitness_list = {10, 20, 30, 40, 50};
00031
00032     for (int i = 0; i < fitness_list.size(); i++){
00033         individual.Set_fitness(fitness_list[i]);
00034         assert(individual.Get_fitness() == fitness_list[i]);
00035     }
00036 }
```

### 3.15.2.4 test\_Get\_vectorLength()

```
void UnitTestIndividual::test_Get_vectorLength ( )
```

Test the getter function for vector\_length.

```
00038 {
00039     std::vector<int> vectorLength_list = {10, 100, 1000};
00040
00041     for (int item : vectorLength_list){
00042         individual = Individual(item, 0, 10);
00043         assert(individual.Get_vectorLength() == item);
00044     }
00045 }
```

### 3.15.3 Member Data Documentation

#### 3.15.3.1 individual

`Individual<int> UnitTestIndividual::individual` [protected]

The documentation for this class was generated from the following files:

- [Unittests.h](#)
- [test\\_Individual.cpp](#)





# Chapter 4

## File Documentation

### 4.1 CCircuit.h File Reference

[Circuit](#) class.

```
#include <vector>
#include "CUnit.h"
#include "DirectedGraph.hpp"
```

#### Classes

- struct [Circuit\\_Parameters](#)  
*A Structure to store the parameters for the circuit simulator.*
- class [Circuit](#)  
*[Circuit](#) class.*

#### Functions

- void [Calculate\\_Circuit](#) ([Circuit](#) &circuit, struct [Circuit\\_Parameters](#) parameters)  
*Calculate the ger and waste rates in all streams of the circuit.*
- void [Calculate\\_Circuit](#) ([Circuit](#) &circuit)  
*Overloaded function to calculate the ger and waste rates in all streams of the circuit.*

#### 4.1.1 Detailed Description

[Circuit](#) class.

#### Author

acse-sm222

#### Copyright

Copyright (c) 2023

## 4.1.2 Function Documentation

### 4.1.2.1 Calculate\_Circuit() [1/2]

```
void Calculate_Circuit (
    Circuit & circuit )
```

Overloaded function to calculate the ger and waste rates in all streams of the circuit.

## Parameters

<i>circuit_vector</i>	
-----------------------	--

Overloaded function to calculate the ger and waste rates in all streams of the circuit.

## Parameters

<i>circuit_vector</i>	<a href="#">Circuit</a> vector
-----------------------	--------------------------------

```
00154 {
00155     struct Circuit\_Parameters default_circuit_parameters;
00156     Calculate\_Circuit(circuit, default_circuit_parameters);
00157 }
```

## 4.1.2.2 Calculate\_Circuit() [2/2]

```
void Calculate_Circuit (
    Circuit & circuit,
    struct Circuit\_Parameters parameters )
```

Calculate the ger and waste rates in all streams of the circuit.

## Parameters

<i>circuit_vector</i>	
<i>parameters</i>	

Calculate the ger and waste rates in all streams of the circuit.

## Parameters

<i>circuit_vector</i>	<a href="#">Circuit</a> vector
<i>parameters</i>	<a href="#">Circuit</a> parameters

```
00023 {
00024     // 1. Assign initial values for F_ger and F_waste in each unit of the circuit
00025     for (int i = 0; i < circuit.num_units; ++i)
00026     {
00027         circuit.units[i].input.ger = parameters.input_Fger;
00028         circuit.units[i].input.waste = parameters.input_Fwaste;
00029         circuit.units[i].F = circuit.units[i].input.ger + circuit.units[i].input.waste;
00030     }
00031
00032     // 2. Calculate the flow rates for each unit of the circuit
00033     for (int i = 0; i < circuit.num_units; ++i)
00034     {
00035         circuit.units[i].calculateOutputFlowRates(circuit.units[i].input);
00036     }
00037
00038     // 3. Set F_old = F
00039     for (int i = 0; i < circuit.num_units; ++i)
00040     {
00041         circuit.units[i].F_old = circuit.units[i].F;
00042     }
00043
00044     // Total old feed rates
00045     std::vector<double> F_old(circuit.num_units, 0.0);
00046     for (int i = 0; i < circuit.num_units; ++i)
00047     {
00048         F_old[i] = circuit.units[i].F_old;
00049     }
00050
00051     // New Feed rates
```

```

00052     std::vector<double> F(circuit.num_units, 0.0);
00053
00054     // Boolean to check if the circuit has reached steady state
00055     bool terminate = false;
00056
00057     circuit.it = 0;
00058     while (!terminate)
00059     {
00060         // 4. Calculate the input flow rates for each unit of the circuit
00061         for (int i = 0; i < circuit.num_units; ++i)
00062         {
00063             circuit.units[i].F_old = circuit.units[i].F;
00064             circuit.units[i].input.ger = 0;
00065             circuit.units[i].input.waste = 0;
00066         }
00067
00068         for (int i = 0; i < circuit.num_units; ++i)
00069         {
00070             // First the concentration stream
00071             if (circuit.units[i].conc_num >= 0)
00072             {
00073                 circuit.units[circuit.units[i].conc_num].input += circuit.units[i].conc;
00074             }
00075
00076             // Then the tails stream
00077             if (circuit.units[i].tails_num >= 0)
00078             {
00079                 circuit.units[circuit.units[i].tails_num].input += circuit.units[i].tail;
00080             }
00081
00082             if (i == circuit.feeder)
00083             {
00084                 circuit.units[i].input.ger += parameters.input_Fger;
00085                 circuit.units[i].input.waste += parameters.input_Fwaste;
00086             }
00087         }
00088
00089         // 5. Calculate the output flow rates for each unit of the circuit
00090         for (int i = 0; i < circuit.num_units; ++i)
00091         {
00092             circuit.units[i].calculateOutputFlowRates(circuit.units[i].input);
00093             F[i] = circuit.units[i].input.ger + circuit.units[i].input.waste;
00094         }
00095
00096         // 6. Check for convergence between old and new feed rates of the circuit
00097         bool converged = true;
00098         for (int i = 0; i < circuit.num_units; ++i)
00099         {
00100             if (fabs(F_old[i] - F[i]) > parameters.tolerance)
00101             {
00102                 // Circuit not converged yet. Can break and continue
00103                 converged = false;
00104                 break;
00105             }
00106         }
00107
00108         if (converged)
00109         {
00110             terminate = true;
00111         }
00112         else
00113         {
00114             terminate = false;
00115         }
00116
00117         ++circuit.it;
00118         F_old = F;
00119         // F.resize(circuit.num_units, 0.0);
00120
00121         if (circuit.it > parameters.max_iterations)
00122         {
00123             return;
00124         }
00125     }
00126 }

```

## 4.2 CCircuit.h

[Go to the documentation of this file.](#)

```

00001 #pragma once
00002

```

```

00012 #include <vector>
00013 #include "CUnit.h"
00014 #include "DirectedGraph.hpp"
00015
00020 struct Circuit_Parameters
00021 {
00022     double tolerance = 1e-6;
00023     int max_iterations = 1000;
00024     double input_Fger = 10.0;
00025     double input_Fwaste = 100.0;
00026     double get_money = 100.0;
00027     double lose_money = 500.0;
00028 };
00029
00035 class Circuit
00036 {
00037 public:
00038     Circuit(std::vector<int> circuit_vector);
00039     ~Circuit();
00040     void toFile();
00041     int num_units;
00042     int feeder;
00043     std::vector<int> circuit_vector;
00044     std::vector<CUnit> units;
00045     static bool CheckValidity(const std::vector<int> &circuit_vector);
00046     int it;
00047 };
00048
00055 void Calculate_Circuit(Circuit &circuit, struct Circuit_Parameters parameters);
00056
00062 void Calculate_Circuit(Circuit &circuit);

```

## 4.3 CSimulator.h File Reference

Simulator class.

```

#include <vector>
#include <float.h>
#include "CCircuit.h"

```

### Functions

- double [Evaluate\\_Circuit](#) (std::vector< int > circuit\_vector, struct [Circuit\\_Parameters](#) parameters)  
*Function to evaluate the objective function.*
- double [Evaluate\\_Circuit](#) (std::vector< int > circuit\_vector)  
*Overloaded function to evaluate the objective function.*

### 4.3.1 Detailed Description

Simulator class.

Author

acse-sm222

Copyright

Copyright (c) 2023

## 4.3.2 Function Documentation

### 4.3.2.1 Evaluate\_Circuit() [1/2]

```
double Evaluate_Circuit (
    std::vector< int > circuit_vector )
```

Overloaded function to evaluate the objective function.

## Parameters

<i>circuit_vector</i>	Vector representing the separator circuit
-----------------------	---

## Returns

double Objective function value

Overloaded function to evaluate the objective function.

## Parameters

<i>circuit_vector</i>	<a href="#">Circuit</a> vector
-----------------------	--------------------------------

## Returns

double Objective function value

```
00166 {
00167     struct Circuit\_Parameters default_circuit_parameters;
00168     return Evaluate\_Circuit(circuit_vector, default_circuit_parameters);
00169 };
```

## 4.3.2.2 Evaluate\_Circuit() [2/2]

```
double Evaluate_Circuit (
    std::vector< int > circuit_vector,
    struct Circuit\_Parameters parameters )
```

Function to evaluate the objective function.

## Parameters

<i>circuit_vector</i>	Vector representing the separator circuit
<i>parameters</i>	Structure containing the parameters for the circuit simulator

## Returns

double Objective function value

Function to evaluate the objective function.

## Parameters

<i>circuit_vector</i>	<a href="#">Circuit</a> vector
<i>parameters</i>	<a href="#">Circuit</a> parameters

## Returns

double Objective function value

```

00136 {
00137     // 1. Create a circuit object
00138     Circuit circuit(circuit_vector);
00139
00140     // 2. Calculate the circuit
00141     Calculate_Circuit(circuit, parameters);
00142
00143     double objective = parameters.get_money * circuit.units[circuit.num_units - 2].input.get -
parameters.lose_money * circuit.units[circuit.num_units - 2].input.waste;
00144
00145     return objective;
00146 }

```

## 4.4 CSimulator.h

[Go to the documentation of this file.](#)

```

00001 #pragma once
00002
00012 #include <vector>
00013 #include <float.h>
00014 #include "CCircuit.h"
00015
00023 double Evaluate_Circuit(std::vector<int> circuit_vector, struct Circuit_Parameters parameters);
00024
00031 double Evaluate_Circuit(std::vector<int> circuit_vector);

```

## 4.5 CUnit.h File Reference

Unit class.

```

#include <stdio.h>
#include <cstdlib>

```

### Classes

- class [Stream](#)  
A [Stream](#) class to store the stream information.
- class [CUnit](#)  
A Unit class to store the unit information and processing.

### 4.5.1 Detailed Description

Unit class.

#### Author

acse-yl1922, acse-sm222

#### Copyright

Copyright (c) 2023



## 4.6 CUnit.h

[Go to the documentation of this file.](#)

```

00001 #pragma once
00002
00012 #include <stdio.h>
00013 #include <cstdlib>
00014
00020 class Stream
00021 {
00022 public:
00023     Stream()
00024     {
00025         ger = 0;
00026         waste = 0;
00027     };
00028     Stream(double GER, double WASTE) : ger(GER), waste(WASTE){};
00029     double ger;
00030     double waste;
00031     Stream operator+(const Stream &s)
00032     {
00033         Stream stream;
00034         stream.ger = this->ger + s.ger;
00035         stream.waste = this->waste + s.waste;
00036         return stream;
00037     }
00038
00039     void operator+=(const Stream &s)
00040     {
00041         this->ger += s.ger;
00042         this->waste += s.waste;
00043     }
00044
00045     ~Stream() {}
00046 };
00047
00054 class CUnit
00055 {
00056 public:
00057     CUnit();
00058     CUnit(int id, int conc_num, int tails_num);
00059     int conc_num;
00060     int tails_num;
00062     Stream input;
00063     Stream conc;
00064     Stream tail;
00065     double F, F_old;
00067     void calculateOutputFlowRates(const Stream &INP);
00069     ~CUnit() {}
00071 private:
00072     int id;
00073     double k_ger = 0.005;
00074     double k_waste = 0.0005;
00075     int rho = 3000;
00076     double phi = 0.1;
00077     int V = 10;
00078 };

```

## 4.7 DirectedGraph.hpp File Reference

Directed graph class.

```

#include <iostream>
#include <vector>
#include <fstream>

```

### Classes

- class [DirectedGraph](#)  
*A directed graph.*
- class [DepthFirstSearch](#)  
*Depth first search class for directed graph.*

### 4.7.1 Detailed Description

Directed graph class.

Author

acse-sm222

Copyright

Copyright (c) 2023

## 4.8 DirectedGraph.hpp

[Go to the documentation of this file.](#)

```
00001 #pragma once
00002
00011 #include <iostream>
00012 #include <vector>
00013 #include <fstream>
00014
00021 class DirectedGraph
00022 {
00023 public:
00024     DirectedGraph(){};
00025     DirectedGraph(int n);
00026     void addEdge(int u, int v);
00027     int getNumVertices() const;
00028     int getNumEdges() const;
00029     std::vector<int> getNeighbors(int u) const;
00030     friend std::ostream &operator<<(std::ostream &os, const DirectedGraph &G);
00031     void writeToFile(std::string filename) const;
00033 private:
00034     std::vector<std::vector<int>> adjList;
00035 };
00036
00043 class DepthFirstSearch
00044 {
00045 public:
00046     DepthFirstSearch(){};
00047     DepthFirstSearch(const DirectedGraph &G, int s);
00048     void dfs(const DirectedGraph &G, int u);
00049     bool visited(int v) const;
00051 private:
00052     std::vector<bool> marked;
00053 };
```

## 4.9 GeneticAlgorithm.hpp File Reference

Header file for the GeneticAlgorithm class.

```
#include <vector>
#include <string>
#include <random>
#include <iostream>
#include <tuple>
#include <functional>
#include <set>
#include <algorithm>
#include <chrono>
#include <omp.h>
```

## Classes

- class [Individual< T >](#)  
*Template class representing an individual in the [GA](#) population.*
- class [Crossover< T >](#)  
*Performs crossover operation between two parent individuals.*
- class [GA< T >](#)  
*Template class representing a Genetic Algorithm ([GA](#)).*

## Macros

- `#define` [PARALLEL](#)

## Enumerations

- enum [CrossoverType](#) { [ONE\\_PT](#) = 0 , [TWOPTS](#) = 1 , [UNIFORM](#) = 2 , [MULTPTS](#) = 3 }

## Functions

- double [RandomDouble](#) (double min, double max)
- int [RandomInt](#) (int min, int max)

### 4.9.1 Detailed Description

Header file for the GeneticAlgorithm class.

#### Copyright

Copyright (c) 2023

Adaptive crossover and mutation rates are implemented according to the following paper: Ref: Lin, W.Y., Lee, W.Y. and Hong, T.P., 2003. Adapting crossover and mutation rates in genetic algorithms. J. Inf. Sci. Eng., 19(5), pp.889-903. This works efficiently for single threaded applications. For parallel applications, the crossover and mutation rates fixed. Can be made for parallel by reduction if time permits

### 4.9.2 Macro Definition Documentation

#### 4.9.2.1 PARALLEL

```
#define PARALLEL
```

### 4.9.3 Enumeration Type Documentation

#### 4.9.3.1 CrossoverType

```
enum CrossoverType
```

## Enumerator

ONE_PT	
TWOPTS	
UNIFORM	
MULTPTS	

```

00131 {
00132     ONE_PT = 0,
00133     TWOPTS = 1,
00134     UNIFORM = 2,
00135     MULTPTS = 3
00136 };

```

## 4.9.4 Function Documentation

### 4.9.4.1 RandomDouble()

```

double RandomDouble (
    double min,
    double max )
00007 {
00008     std::uniform_real_distribution<double> dist(min, max); // Define the range of random numbers
00009     return dist(gen); // Generate and return a random number
00010 }

```

### 4.9.4.2 RandomInt()

```

int RandomInt (
    int min,
    int max )
00014 {
00015     std::uniform_int_distribution<int> dist(min, max); // Define the range of random numbers
00016     return dist(gen); // Generate and return a random number
00017 }

```

## 4.10 GeneticAlgorithm.hpp

[Go to the documentation of this file.](#)

```

00001 #pragma once
00002
00014 #include <vector>
00015 #include <string>
00016 #include <random>
00017 #include <iostream>
00018 #include <tuple>
00019 #include <functional>
00020 #include <set>
00021 #include <algorithm>
00022 #include <chrono>
00023 #include <omp.h>
00024 #define PARALLEL
00025 // #define SERIAL
00026 // #define ADAPTIVE
00027
00028 // Random number generator
00029 static std::random_device rd;
00030 static std::mt19937 gen(rd()); // Initialize random number generator only once
00031
00040 double RandomDouble(double min, double max);
00041
00050 int RandomInt(int min, int max);
00051
00061 template <typename T>

```

```

00062 class Individual
00063 {
00064 public:
00065     std::vector<T> chromosome; // Chromosome representing the genetic information
00066     double fitness;           // Fitness value indicating the quality of the individual's solution
00067
00068     // overload comparison operator
00072     Individual() {}
00073
00079     Individual(const std::vector<T> &chromosome) : chromosome(chromosome) {}
00080
00081     // overload comparison operator
00088     bool operator<(const Individual &ind) const
00089     {
00090         return fitness < ind.fitness;
00091     }
00092
00099     bool operator>(const Individual &ind) const
00100     {
00101         return fitness > ind.fitness;
00102     }
00103
00110     bool operator==(const Individual &ind) const
00111     {
00112         return chromosome == ind.chromosome;
00113     }
00114
00121     Individual &operator=(const Individual &ind)
00122     {
00123         chromosome.resize(ind.chromosome.size());
00124         chromosome = ind.chromosome;
00125         fitness = ind.fitness;
00126         return *this;
00127     }
00128 };
00129
00130 enum CrossoverType
00131 {
00132     ONE_PT = 0,
00133     TWOPTS = 1,
00134     UNIFORM = 2,
00135     MULTPTS = 3
00136 };
00137
00150 template <typename T>
00151 class Crossover
00152 {
00153 public:
00154     static void one_pt_crossover(Individual<T> &parent1, Individual<T> &parent2, Individual<T>
&child1, Individual<T> &child2);
00155     static void two_pts_crossover(Individual<T> &parent1, Individual<T> &parent2, Individual<T>
&child1, Individual<T> &child2);
00156     static void uniform_crossover(Individual<T> &parent1, Individual<T> &parent2, Individual<T>
&child1, Individual<T> &child2);
00157     static void mult_pts_crossover(Individual<T> &parent1, Individual<T> &parent2, Individual<T>
&child1, Individual<T> &child2);
00158     static void crossover(Individual<T> &parent1, Individual<T> &parent2, Individual<T> &child1,
Individual<T> &child2, CrossoverType type);
00159 };
00160
00170 template <typename T>
00171 class GA
00172 {
00173 public:
00174     // Default constructor
00175     GA() = default;
00187     GA(int populationSize, int chromosomeSize, T chromomin, T chromomax, int maxGeneration, double
crossoverRate, double mutationRate);
00188
00198     void setValidity(const std::function<bool(const std::vector<T> &)> &validity);
00199
00209     void setFitness(const std::function<double(const std::vector<T> &)> &fitness);
00210
00217     Individual<T> optimize(CrossoverType type);
00218
00222     ~GA(){};
00223
00224     T CHROMOMIN;           // Minimum value of discrete allele
00225     T CHROMOMAX;           // Maximum value of discrete allele
00226     int POP_SIZE;          // Population size
00227     int CHROMOSOME_SIZE;   // Chromosome size
00228     int MAX_GENERATION;    // Maximum generation
00229     double CROSSOVER_RATE; // Crossover rate
00230     double MUTATION_RATE;  // Mutation rate
00231     int ELITISM;           // Number of elite individuals

```

```

00232     int CONV_GEN;           // Number of generations for convergence
00233     bool verbose = false;   // Print progress to console
00234
00235     std::vector<Individual<T> > population; // Population of individuals
00242     void toFile(std::string fileDir);
00243
00253     void mutate(Individual<T> &child);
00254
00265     int selectParent(const std::vector<double> &cf);
00266
00267 private:
00268     // GA variables
00269     std::vector<Individual<T> > offspring; // Offspring population
00270     std::vector<double> fitness;           // Fitness values of individuals
00271     std::vector<double> scaledFitness;     // Scaled fitness values of individuals
00272     std::vector<double> cumulativeFitness; // Cumulative fitness values of individuals
00273     int cross_cnt, mut_cnt;               // Crossover and mutation counters
00274     double cross_contrib, mut_contrib;    // Crossover and mutation contribution to the next
00275     generation
00276     std::vector<double> bestFitnessHistory; // Best fitness values of each generation
00277
00278     // GA functions
00286     void initPopulation();
00287
00296     void updatePopulation(CrossoverType type);
00297
00304     int tournamentSelection(int k);
00305
00306
00307
00308
00320     std::function<bool(const std::vector<T> &)> validity_;
00321
00333     std::function<double(const std::vector<T> &)> fitness_;
00334 };
00335
00336 // Declare the template class
00337 template class GA<double>;
00338 template class GA<int>;
00339 template class Crossover<double>;
00340 template class Crossover<int>;

```

## 4.11 CCircuit.cpp File Reference

[Circuit](#) class implementation.

```

#include <vector>
#include <CCircuit.h>

```

### Classes

- class [Units](#)

### Functions

- void [markunits](#) (std::vector< [Units](#) > &units, int unit\_num)

#### 4.11.1 Detailed Description

[Circuit](#) class implementation.

#### Author

acse-yl1922, acse-sm222

#### Copyright

Copyright (c) 2023

## 4.11.2 Function Documentation

### 4.11.2.1 markunits()

```

void markunits (
    std::vector< Units > & units,
    int unit_num )
00024 {
00025     if (units[unit_num].marked == true)
00026     {
00027         return;
00028     }
00029     else
00030     {
00031         units[unit_num].marked = true;
00032     }
00033
00034     if (units[unit_num].conc_num < units.size())
00035     {
00036         markunits(units, units[unit_num].conc_num);
00037     }
00038     else
00039     {
00040         return;
00041     }
00042
00043     if (units[unit_num].tails_num < units.size())
00044     {
00045         markunits(units, units[unit_num].tails_num);
00046     }
00047     else
00048     {
00049         return;
00050     }
00051 }

```

## 4.12 CSimulator.cpp File Reference

Simulator calculations.

```

#include <cmath>
#include <cassert>
#include "CUnit.h"
#include "CCircuit.h"
#include "CSimulator.h"

```

### Functions

- void [Calculate\\_Circuit](#) ([Circuit](#) &circuit, struct [Circuit\\_Parameters](#) parameters)  
*Calculates the steady state feed rates for the entire circuit.*
- double [Evaluate\\_Circuit](#) (std::vector< int > circuit\_vector, struct [Circuit\\_Parameters](#) parameters)  
*Evaluate the circuit with given parameters.*
- void [Calculate\\_Circuit](#) ([Circuit](#) &circuit)  
*Calculate the circuit with default parameters.*
- double [Evaluate\\_Circuit](#) (std::vector< int > circuit\_vector)  
*Evaluate the circuit with default parameters.*

### 4.12.1 Detailed Description

Simulator calculations.

#### Author

acse-sm222

#### Copyright

Copyright (c) 2023

### 4.12.2 Function Documentation

#### 4.12.2.1 Calculate\_Circuit() [1/2]

```
void Calculate_Circuit (
    Circuit & circuit )
```

Calculate the circuit with default parameters.

Overloaded function to calculate the ger and waste rates in all streams of the circuit.

#### Parameters

<i>circuit_vector</i>	Circuit vector
-----------------------	----------------

```
00154 {
00155     struct Circuit_Parameters default_circuit_parameters;
00156     Calculate_Circuit(circuit, default_circuit_parameters);
00157 }
```

#### 4.12.2.2 Calculate\_Circuit() [2/2]

```
void Calculate_Circuit (
    Circuit & circuit,
    struct Circuit_Parameters parameters )
```

Calculates the steady state feed rates for the entire circuit.

Calculate the ger and waste rates in all streams of the circuit.

#### Parameters

<i>circuit_vector</i>	Circuit vector
<i>parameters</i>	Circuit parameters

```
00023 {
00024     // 1. Assign initial values for F_ger and F_waste in each unit of the circuit
00025     for (int i = 0; i < circuit.num_units; ++i)
00026     {
00027         circuit.units[i].input.ger = parameters.input_Fger;
00028         circuit.units[i].input.waste = parameters.input_Fwaste;
00029         circuit.units[i].F = circuit.units[i].input.ger + circuit.units[i].input.waste;
00030     }
```



```

00031
00032 // 2. Calculate the flow rates for each unit of the circuit
00033 for (int i = 0; i < circuit.num_units; ++i)
00034 {
00035     circuit.units[i].calculateOutputFlowRates(circuit.units[i].input);
00036 }
00037
00038 // 3. Set F_old = F
00039 for (int i = 0; i < circuit.num_units; ++i)
00040 {
00041     circuit.units[i].F_old = circuit.units[i].F;
00042 }
00043
00044 // Total old feed rates
00045 std::vector<double> F_old(circuit.num_units, 0.0);
00046 for (int i = 0; i < circuit.num_units; ++i)
00047 {
00048     F_old[i] = circuit.units[i].F_old;
00049 }
00050
00051 // New Feed rates
00052 std::vector<double> F(circuit.num_units, 0.0);
00053
00054 // Boolean to check if the circuit has reached steady state
00055 bool terminate = false;
00056
00057 circuit.it = 0;
00058 while (!terminate)
00059 {
00060     // 4. Calculate the input flow rates for each unit of the circuit
00061     for (int i = 0; i < circuit.num_units; ++i)
00062     {
00063         circuit.units[i].F_old = circuit.units[i].F;
00064         circuit.units[i].input.ger = 0;
00065         circuit.units[i].input.waste = 0;
00066     }
00067
00068     for (int i = 0; i < circuit.num_units; ++i)
00069     {
00070         // First the concentration stream
00071         if (circuit.units[i].conc_num >= 0)
00072         {
00073             circuit.units[circuit.units[i].conc_num].input += circuit.units[i].conc;
00074         }
00075
00076         // Then the tails stream
00077         if (circuit.units[i].tails_num >= 0)
00078         {
00079             circuit.units[circuit.units[i].tails_num].input += circuit.units[i].tail;
00080         }
00081
00082         if (i == circuit.feeder)
00083         {
00084             circuit.units[i].input.ger += parameters.input_Fger;
00085             circuit.units[i].input.waste += parameters.input_Fwaste;
00086         }
00087     }
00088
00089     // 5. Calculate the output flow rates for each unit of the circuit
00090     for (int i = 0; i < circuit.num_units; ++i)
00091     {
00092         circuit.units[i].calculateOutputFlowRates(circuit.units[i].input);
00093         F[i] = circuit.units[i].input.ger + circuit.units[i].input.waste;
00094     }
00095
00096     // 6. Check for convergence between old and new feed rates of the circuit
00097     bool converged = true;
00098     for (int i = 0; i < circuit.num_units; ++i)
00099     {
00100         if (fabs(F_old[i] - F[i]) > parameters.tolerance)
00101         {
00102             // Circuit not converged yet. Can break and continue
00103             converged = false;
00104             break;
00105         }
00106     }
00107
00108     if (converged)
00109     {
00110         terminate = true;
00111     }
00112     else
00113     {
00114         terminate = false;
00115     }
00116
00117     ++circuit.it;

```

```

00118     F_old = F;
00119     // F.resize(circuit.num_units, 0.0);
00120
00121     if (circuit.it > parameters.max_iterations)
00122     {
00123         return;
00124     }
00125 }
00126 }

```

#### 4.12.2.3 Evaluate\_Circuit() [1/2]

```

double Evaluate_Circuit (
    std::vector< int > circuit_vector )

```

Evaluate the circuit with default parameters.

Overloaded function to evaluate the objective function.

##### Parameters

<i>circuit_vector</i>	<a href="#">Circuit</a> vector
-----------------------	--------------------------------

##### Returns

double Objective function value

```

00166 {
00167     struct Circuit_Parameters default_circuit_parameters;
00168     return Evaluate_Circuit(circuit_vector, default_circuit_parameters);
00169 };

```

#### 4.12.2.4 Evaluate\_Circuit() [2/2]

```

double Evaluate_Circuit (
    std::vector< int > circuit_vector,
    struct Circuit_Parameters parameters )

```

Evaluate the circuit with given parameters.

Function to evaluate the objective function.

##### Parameters

<i>circuit_vector</i>	<a href="#">Circuit</a> vector
<i>parameters</i>	<a href="#">Circuit</a> parameters

##### Returns

double Objective function value

```

00136 {
00137     // 1. Create a circuit object
00138     Circuit circuit(circuit_vector);
00139
00140     // 2. Calculate the circuit
00141     Calculate_Circuit(circuit, parameters);
00142 }

```

```
00143     double objective = parameters.get_money * circuit.units[circuit.num_units - 2].input.get -
                                parameters.lose_money * circuit.units[circuit.num_units - 2].input.waste;
00144
00145     return objective;
00146 }
```

## 4.13 CUnit.cpp File Reference

CUnit class implementation.

```
#include "CUnit.h"
```

### 4.13.1 Detailed Description

CUnit class implementation.

Author

your name ( [you@domain.com](mailto:you@domain.com) )

Copyright

Copyright (c) 2023

## 4.14 DirectedGraph.cpp File Reference

Directed graph implementation.

```
#include "DirectedGraph.hpp"
```

### Functions

- `std::ostream & operator<< (std::ostream &os, const DirectedGraph &G)`  
*Overload the << operator.*

### 4.14.1 Detailed Description

Directed graph implementation.

Author

your name ( [you@domain.com](mailto:you@domain.com) )

Copyright

Copyright (c) 2023

### 4.14.2 Function Documentation

#### 4.14.2.1 operator<<()

```
std::ostream & operator<< (
    std::ostream & os,
    const DirectedGraph & G )
```

Overload the << operator.

**Parameters**

<i>os</i>	Output stream
<i>G</i>	Directed graph

**Returns**

std::ostream& Output stream

```

00077 {
00078     for (int u = 0; u < G.getNumVertices(); u++)
00079     {
00080         for (int v : G.getNeighbors(u))
00081         {
00082             os << u << " " << v << "\n";
00083         }
00084     }
00085     return os;
00086 }
```

## 4.15 GeneticAlgorithm.cpp File Reference

```

#include <fstream>
#include <sstream>
#include "GeneticAlgorithm.hpp"
```

**Functions**

- double [RandomDouble](#) (double min, double max)
- int [RandomInt](#) (int min, int max)

### 4.15.1 Function Documentation

#### 4.15.1.1 RandomDouble()

```

double RandomDouble (
    double min,
    double max )
00007 {
00008     std::uniform_real_distribution<double> dist(min, max); // Define the range of random numbers
00009     return dist(gen); // Generate and return a random number
00010 }
```

#### 4.15.1.2 RandomInt()

```

int RandomInt (
    int min,
    int max )
00014 {
00015     std::uniform_int_distribution<int> dist(min, max); // Define the range of random numbers
00016     return dist(gen); // Generate and return a random number
00017 }
```

## 4.16 main.cpp File Reference

```
#include <iostream>
#include <chrono>
#include <omp.h>
#include "CUnit.h"
#include "CCircuit.h"
#include "CSimulator.h"
#include "GeneticAlgorithm.hpp"
```

### Classes

- struct [CircuitOptimizer](#)

### Functions

- double [fitness](#) (const std::vector< int > &x)
- bool [validity](#) (const std::vector< int > &x)
- int [main](#) ()

### Variables

- [CircuitOptimizer myCircuit](#)

### 4.16.1 Function Documentation

#### 4.16.1.1 fitness()

```
double fitness (
    const std::vector< int > & x )
00020 {
00021     std::vector<int> circuit_vector(x.size() + 1);
00022     circuit_vector[0] = myCircuit.feeder;
00023     for (int i = 0; i < x.size(); ++i)
00024     {
00025         circuit_vector[i + 1] = x[i];
00026     }
00027     return Evaluate_Circuit(circuit_vector, parameter);
00028 }
```

#### 4.16.1.2 main()

```
int main ( )
```

< Input flow rate of the mineral

< Input flow rate of the waste

< Money earned per unit of mineral

< Money lost per unit of waste

```
00043 {
```

```

00044      /***** Analysis Settings *****/
00045      // Setup the circuit parameters
00046      parameter.input_Fger = 10.0;
00047      parameter.input_Fwaste = 100.0;
00048      parameter.get_money = 100.0;
00049      parameter.lose_money = 500.0;
00050      parameter.tolerance = 1e-6;
00051
00052      // Setup the GA
00053      int POP_SIZE = 200;
00054      int MAX_GENERATION = 2000;
00055      double CROSSOVER_RATE = 0.9;
00056      double MUTATION_RATE = 0.01;
00057      int CONV_GEN = MAX_GENERATION;
00058      int ELITE = POP_SIZE * 0.1;
00059      CrossoverType type = CrossoverType::UNIFORM;
00060      /*****
00061
00062      // Setup the circuit optimiser
00063      CircuitOptimizer myCircuit;
00064
00065      // Don't change these
00066      int CHROMOSOME_SIZE = 2 * myCircuit.num_units;
00067      int CHROMOMIN = 0;
00068      int CHROMOMAX = myCircuit.num_units + 2;
00069
00070      GA<int> ga(POP_SIZE, CHROMOSOME_SIZE, CHROMOMIN, CHROMOMAX, MAX_GENERATION, CROSSOVER_RATE,
00071      MUTATION_RATE);
00072      ga.CONV_GEN = CONV_GEN;
00073      ga.ELITISM = ELITE;
00074      ga.setFitness(fitness);
00075      ga.setValidity(validity);
00076
00077      double wtime = omp_get_wtime();
00078      auto bestIndividual = ga.optimize(type);
00079      wtime = omp_get_wtime() - wtime;
00080      std::cout << "Time taken: " << wtime << "s\n";
00081
00082      gaToFile("../out/");
00083
00084      std::vector<int> bestCircuitVector = bestIndividual.chromosome;
00085      bestCircuitVector.insert(bestCircuitVector.begin(), myCircuit.feeder);
00086
00087      Circuit bestCircuit(bestCircuitVector);
00088      bestCircuit.toFile();
00089
00090      return 0;
00091  }

```

#### 4.16.1.3 validity()

```

bool validity (
    const std::vector< int > & x )
{
    00031 {
    00032     std::vector<int> circuit_vector(x.size() + 1);
    00033     circuit_vector[0] = myCircuit.feeder;
    00034     for (int i = 0; i < x.size(); ++i)
    00035     {
    00036         circuit_vector[i + 1] = x[i];
    00037     }
    00038
    00039     return Circuit::Check_Validity(circuit_vector);
    00040 }

```

### 4.16.2 Variable Documentation

#### 4.16.2.1 myCircuit

CircuitOptimizer myCircuit

## 4.17 test\_all.cpp File Reference

```
#include "Unittests.h"
#include "test_GeneticAlgorithm.cpp"
#include "test_Crossover.cpp"
#include "test_Circuit.cpp"
```

### Functions

- int [main](#) ()

### 4.17.1 Function Documentation

#### 4.17.1.1 main()

```
int main ( )
00007     {
00008         // set a random seed for consistent results
00009         srand(42);
00010
00011         int POPULATION_SIZE = 100;
00012         int CHROMOSOME_SIZE = 10;
00013         int CHROMOMIN = 0;
00014         int CHROMOMAX = 10;
00015         int MAX_GENERATION = 1500;
00016         int CONV_GEN = 1500;
00017         double CROSSOVER_RATE = 0.8;
00018         double MUTATION_RATE = 0.5;
00019
00020         UnitTestGA test_ga(POPULATION_SIZE,
00021             CHROMOSOME_SIZE,
00022             CHROMOMIN,
00023             CHROMOMAX,
00024             MAX_GENERATION,
00025             CROSSOVER_RATE,
00026             MUTATION_RATE
00027         );
00028         test_ga.run();
00029
00030         UnitTestCrossover x;
00031         x.test_one_pt_crossover();
00032         x.test_two_pts_crossover();
00033         x.test_mult_pts_crossover();
00034         x.test_uniform_crossover();
00035
00036         UnitTestCircuit circuit;
00037         circuit.run();
00038
00039
00040 }
```

## 4.18 test\_Circuit.cpp File Reference

```
#include <vector>
#include <iostream>
#include <cmath>
#include "Unittests.h"
```

## 4.19 test\_Circuit.cpp

[Go to the documentation of this file.](#)

```

00001 #include <vector>
00002 #include <iostream>
00003 #include <cmath>
00004 #include "Unittests.h"
00005
00006 UnitTestCircuit::UnitTestCircuit() {}
00007 UnitTestCircuit::~~UnitTestCircuit() {}
00008
00009 void UnitTestCircuit::test_circuit_evaluation(){
00010     std::cout << "=====Test Circuit Evaluation=====\\n";
00011
00012     std::vector<int> circuit_vector = {4, 5, 1, 2, 4, 0, 1, 1, 6, 1, 3};
00013     bool validity = Circuit::Check_Validity(circuit_vector);
00014     assert(validity==true);
00015
00016     Circuit_Parameters parameters;
00017     parameters.tolerance = 1e-6;
00018
00019     double value = Evaluate_Circuit(circuit_vector, parameters);
00020     std::cout << value << std::endl;
00021     assert(std::fabs(value - 107.204) <= 1e-3);
00022     std::cout << "completed" << std::endl;
00023 }
00024
00025 void UnitTestCircuit::test_circuit_validity(){
00026     std::cout << "=====Test Circuit Validity=====\\n";
00027
00028     // Cases when there are 3 units
00029     // Case 1: self-recycle
00030     std::vector<int> circuit_vector1 = {0, 1, 2, 0, 2, 1, 0};
00031     // Case 2: points to itself
00032     std::vector<int> circuit_vector2 = {0, 1, 2, 1, 2, 3, 4};
00033     // Case 3: there are units not ending up in 2 destinations
00034     std::vector<int> circuit_vector3 = {0, 1, 1, 3, 2, 1, 4};
00035     // Case 4: there is a unit not accessible from the feed
00036     std::vector<int> circuit_vector4 = {0, 1, 4, 3, 4, 3, 0};
00037     bool check1 = Circuit::Check_Validity(circuit_vector1);
00038     bool check2 = Circuit::Check_Validity(circuit_vector2);
00039     bool check3 = Circuit::Check_Validity(circuit_vector3);
00040     bool check4 = Circuit::Check_Validity(circuit_vector4);
00041     assert(check1==false);
00042     assert(check2==false);
00043     assert(check3==false);
00044     assert(check4==false);
00045     std::cout << "completed" << std::endl;
00046 }
00047
00048 void UnitTestCircuit::test_circuit_unit() {
00049     std::cout << "=====Test Circuit Unit=====\\n";
00050
00051     CUnit test_unit;
00052
00053     test_unit.input.ger = 20;
00054     test_unit.input.waste = 80;
00055
00056     test_unit.calculateOutputFlowRates(test_unit.input);
00057
00058     double eps = 1e-3;
00059
00060     bool check1 = fabs(test_unit.conc.ger - 2.6087) < eps;
00061     bool check2 = fabs(test_unit.conc.waste - 1.18227) < eps;
00062     bool check3 = fabs(test_unit.tail.ger - 17.3913) < eps;
00063     bool check4 = fabs(test_unit.tail.waste - 78.8177) < eps;
00064
00065     assert(check1==true);
00066     assert(check2==true);
00067     assert(check3==true);
00068     assert(check4==true);
00069     std::cout << "completed" << std::endl;
00070 }
00071
00072 void UnitTestCircuit::run(){
00073     test_circuit_evaluation();
00074     test_circuit_validity();
00075     test_circuit_unit();
00076 }
00077

```



## 4.20 test\_CircuitEvaluation.cpp File Reference

```
#include <vector>
#include <iostream>
#include <cmath>
#include "CSimulator.h"
#include "CCircuit.h"
```

### Functions

- void [test](#) ()
- int [main](#) ()

### 4.20.1 Function Documentation

#### 4.20.1.1 main()

```
int main ( )
00021 {
00022     test();
00023     std::vector<int> circuit_vector = {4, 5, 1, 2, 4, 0, 1, 1, 6, 1, 3};
00024     bool validity = Circuit::Check_Validity(circuit_vector);
00025     if (!validity)
00026     {
00027         return 1;
00028     }
00029     std::cout << "Circuit is " << validity << std::endl;
00030     Circuit_Parameters parameters;
00031     parameters.tolerance = 1e-6;
00032     double value = Evaluate_Circuit(circuit_vector, parameters);
00033     std::cout << value << std::endl;
00034     if (std::fabs(value - 107.204) > 1e-3)
00035     {
00036         return 1;
00037     }
00038     return 0;
00039 }
00040 }
```

#### 4.20.1.2 test()

```
void test ( )
00008 {
00009
00010     std::vector<int> circuit_vector = {0, 4, 6, 2, 4, 3, 4, 5, 4, 1, 0};
00011     bool validity = Circuit::Check_Validity(circuit_vector);
00012     Circuit circuit(circuit_vector);
00013     circuitToFile();
00014     Circuit_Parameters parameters;
00015     parameters.tolerance = 1e-3;
00016     double value = Evaluate_Circuit(circuit_vector, parameters);
00017     std::cout << validity << " " << value << std::endl;
00018 }
```

## 4.21 test\_CircuitValidity.cpp File Reference

```
#include "CCircuit.h"
```

## Functions

- bool `testValidity()`
- int `main()`

### 4.21.1 Function Documentation

#### 4.21.1.1 `main()`

```

int main ( )
00018 {
00019     // auto val = testValidity();
00020     // std::cout << val << std::endl;
00021     // Case 1
00022     std::vector<int> circuit_vector1 = {0, 1, 2, 0, 2, 1, 0};
00023     // Case 2: points to itself
00024     std::vector<int> circuit_vector2 = {0, 1, 2, 1, 2, 3, 4};
00025     // Case 3: there are units not ending up in 2 destinations
00026     std::vector<int> circuit_vector3 = {0, 1, 1, 3, 2, 1, 4};
00027     // Case 4: there is a unit not accessible from the feed
00028     std::vector<int> circuit_vector4 = {0, 1, 4, 3, 4, 3, 0};
00029     bool check1 = Circuit::Check_Validity(circuit_vector1);
00030     bool check2 = Circuit::Check_Validity(circuit_vector2);
00031     bool check3 = Circuit::Check_Validity(circuit_vector3);
00032     bool check4 = Circuit::Check_Validity(circuit_vector4);
00033     if (!check1 && !check2 && !check3 && !check4)
00034     {
00035         return 0;
00036     }
00037     else
00038     {
00039         return 1;
00040     }
00041     return 0;
00042 }
```

#### 4.21.1.2 `testValidity()`

```

bool testValidity ( )
00004 {
00005     // std::vector<int> test = {0, 3, 1, 0, 2, 1, 4};
00006     // std::vector<int> test = {0, 2, 1, 0, 4, 3, 0};
00007     // std::vector<int> test = {0, 2, 1, 3, 2, 3, 4};
00008     // std::vector<int> test = {2, 10, 6, 8, 4, 8, 7, 8, 1, 8, 11, 6, 2, 0, 8, 8, 3, 6, 9, 6, 5};
00009     // std::vector<int> test = {0, 6, 7, 2, 6, 10, 1, 1, 8, 6, 5, 6, 9, 1, 3, 6, 4, 1, 0, 6, 8};
00010     std::vector<int> test = {0, 8, 3, 7, 2, 7, 4, 8, 5, 10, 7, 8, 9, 8, 11, 4, 8, 7, 1, 8, 6};
00011     Circuit c(test);
00012     cToFile();
00013     bool check = Circuit::Check_Validity(test);
00014     return check;
00015 }
```

## 4.22 test\_Crossover.cpp File Reference

```

#include <iostream>
#include <vector>
#include <cassert>
#include "GeneticAlgorithm.hpp"
#include "Unittests.h"
```

## Functions

- `template<typename T >`  
`void printChromosome (const std::vector< T > &chromosome)`

## Variables

- `Crossover<int> crossover`

### 4.22.1 Function Documentation

#### 4.22.1.1 printChromosome()

```
template<typename T >
void printChromosome (
    const std::vector< T > & chromosome )
00010 {
00011     std::cout << "{ ";
00012     for (const auto &gene : chromosome)
00013     {
00014         std::cout << gene << " ";
00015     }
00016     std::cout << "}\n";
00017 }
```

### 4.22.2 Variable Documentation

#### 4.22.2.1 crossover

```
Crossover<int> crossover
```

## 4.23 test\_Crossover.cpp

[Go to the documentation of this file.](#)

```
00001 #include <iostream>
00002 #include <vector>
00003 #include <cassert>
00004 #include "GeneticAlgorithm.hpp"
00005 #include "Unittests.h"
00006
00007 // Function for pretty printing a chromosome
00008 template <typename T>
00009 void printChromosome(const std::vector<T> &chromosome)
00010 {
00011     std::cout << "{ ";
00012     for (const auto &gene : chromosome)
00013     {
00014         std::cout << gene << " ";
00015     }
00016     std::cout << "}\n";
00017 }
00018
00019 Crossover<int> crossover;
00020
00021 UnitTestCrossover::UnitTestCrossover()
00022 {
00023     std::cout << "                                Test Crossover Start
00024     \n";
00025 UnitTestCrossover::~UnitTestCrossover() {}
00026
00027 void UnitTestCrossover::test_one_pt_crossover()
```

```

00028 {
00029     std::cout << "=====Test One-Point Crossover
method=====\\n";
00030
00031     std::vector<int> chromosome1 = {1, 2, 3, 4, 5};
00032     std::vector<int> chromosome2 = {6, 7, 8, 9, 10};
00033
00034     Individual<int> parent1, parent2, child1, child2;
00035
00036     parent1.chromosome = chromosome1;
00037     parent2.chromosome = chromosome2;
00038
00039     child1.chromosome.resize(chromosome1.size());
00040     child2.chromosome.resize(chromosome2.size());
00041
00042     crossover.one_pt_crossover(parent1, parent2, child1, child2);
00043
00044     std::cout << "Parent 1: ";
00045     printChromosome(parent1.chromosome);
00046
00047     std::cout << "Parent 2: ";
00048     printChromosome(parent2.chromosome);
00049
00050     std::cout << "Child 1: ";
00051     printChromosome(child1.chromosome);
00052
00053     std::cout << "Child 2: ";
00054     printChromosome(child2.chromosome);
00055
00056     // Check that the chromosomes have been altered
00057     assert(child1.chromosome != parent1.chromosome);
00058     assert(child1.chromosome != parent2.chromosome);
00059     assert(child2.chromosome != parent1.chromosome);
00060     assert(child2.chromosome != parent2.chromosome);
00061 }
00062
00063 void UnitTestCrossover::test_two_pts_crossover()
00064 {
00065     std::cout << "=====Test Two-Points Crossover
method=====\\n";
00066
00067     std::vector<int> chromosome1 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
00068     std::vector<int> chromosome2 = {10, 11, 12, 13, 14, 15, 16, 17, 18, 19};
00069
00070     Individual<int> parent1, parent2, child1, child2;
00071
00072     parent1.chromosome = chromosome1;
00073     parent2.chromosome = chromosome2;
00074
00075     child1.chromosome.resize(chromosome1.size());
00076     child2.chromosome.resize(chromosome2.size());
00077
00078     crossover.two_pts_crossover(parent1, parent2, child1, child2);
00079
00080     std::cout << "Parent 1: ";
00081     printChromosome(parent1.chromosome);
00082
00083     std::cout << "Parent 2: ";
00084     printChromosome(parent2.chromosome);
00085
00086     std::cout << "Child 1: ";
00087     printChromosome(child1.chromosome);
00088
00089     std::cout << "Child 2: ";
00090     printChromosome(child2.chromosome);
00091
00092     // Check that the chromosomes have been altered
00093     assert(child1.chromosome != parent1.chromosome);
00094     assert(child1.chromosome != parent2.chromosome);
00095     assert(child2.chromosome != parent1.chromosome);
00096     assert(child2.chromosome != parent2.chromosome);
00097 }
00098
00099 void UnitTestCrossover::test_mult_pts_crossover()
00100 {
00101     std::cout << "=====Test Multi-Points Crossover
method=====\\n";
00102
00103     std::vector<int> chromosome1 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
00104     std::vector<int> chromosome2 = {10, 11, 12, 13, 14, 15, 16, 17, 18, 19};
00105
00106     Individual<int> parent1, parent2, child1, child2;
00107
00108     parent1.chromosome = chromosome1;
00109     parent2.chromosome = chromosome2;
00110
00111     child1.chromosome.resize(chromosome1.size());

```

```

00112     child2.chromosome.resize(chromosome2.size());
00113
00114     crossover.mult_pts_crossover(parent1, parent2, child1, child2);
00115
00116     std::cout << "Parent 1: ";
00117     printChromosome(parent1.chromosome);
00118
00119     std::cout << "Parent 2: ";
00120     printChromosome(parent2.chromosome);
00121
00122     std::cout << "Child 1: ";
00123     printChromosome(child1.chromosome);
00124
00125     std::cout << "Child 2: ";
00126     printChromosome(child2.chromosome);
00127
00128     // Check that the chromosomes have been altered
00129     assert(child1.chromosome != parent1.chromosome);
00130     assert(child1.chromosome != parent2.chromosome);
00131     assert(child2.chromosome != parent1.chromosome);
00132     assert(child2.chromosome != parent2.chromosome);
00133 }
00134
00135 void UnitTestCrossover::test_uniform_crossover()
00136 {
00137     std::cout << "=====Test Uniform Crossover
method=====\\n";
00138
00139     std::vector<int> chromosome1 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
00140     std::vector<int> chromosome2 = {9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
00141     std::vector<int> expected_chromosome1 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
00142     std::vector<int> expected_chromosome2 = {9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
00143
00144     Individual<int> parent1, parent2, child1, child2;
00145
00146     parent1.chromosome = chromosome1;
00147     parent2.chromosome = chromosome2;
00148
00149     child1.chromosome.resize(chromosome1.size());
00150     child2.chromosome.resize(chromosome2.size());
00151
00152     crossover.uniform_crossover(parent1, parent2, child1, child2);
00153
00154     // Check that the chromosomes have been altered
00155     assert(child1.chromosome != expected_chromosome1);
00156     assert(child2.chromosome != expected_chromosome2);
00157     std::cout << "Results match expected outputs.\\n";
00158 }

```

## 4.24 test\_Cunit.cpp File Reference

```

#include <cmath>
#include <iostream>
#include <cassert>
#include "CUnit.h"

```

### Functions

- int [main](#) (int argc, char \*argv[])

### 4.24.1 Function Documentation

#### 4.24.1.1 main()

```

int main (
    int argc,
    char * argv[] )

```

```

00007 {
00008
00009     CUnit test_unit;
00010
00011     test_unit.input.ger = 20;
00012     test_unit.input.waste = 80;
00013
00014     test_unit.calculateOutputFlowRates(test_unit.input);
00015
00016     double eps = 1e-3;
00017
00018     bool check1 = fabs(test_unit.conc.ger - 2.6087) < eps;
00019     bool check2 = fabs(test_unit.conc.waste - 1.18227) < eps;
00020     bool check3 = fabs(test_unit.tail.ger - 17.3913) < eps;
00021     bool check4 = fabs(test_unit.tail.waste - 78.8177) < eps;
00022     if (check1 && check2 && check3 && check4)
00023     {
00024         return 0;
00025     }
00026     else
00027     {
00028         return 1;
00029     }
00030
00031     return 0;
00032 }

```

## 4.25 test\_DirectedGraph.cpp File Reference

```
#include "DirectedGraph.hpp"
```

### Functions

- int [testDigraph](#) ()
- int [testDFS](#) ()
- int [main](#) ()

### 4.25.1 Function Documentation

#### 4.25.1.1 main()

```

int main ( )
00057 {
00058     int STATUS = 0;
00059     STATUS += testDigraph();
00060     STATUS += testDFS();
00061     return STATUS;
00062 }

```

#### 4.25.1.2 testDFS()

```

int testDFS ( )
00033 {
00034     DirectedGraph G(5);
00035     G.addEdge(0, 1);
00036     G.addEdge(0, 2);
00037     G.addEdge(1, 2);
00038     G.addEdge(2, 0);
00039     G.addEdge(2, 3);
00040     G.addEdge(3, 3);
00041
00042     DepthFirstSearch dfs(G, 2);
00043     dfs.dfs(G, 2);

```

```

00044     bool visited = dfs.visited(3);
00045     if (!visited)
00046         return 1;
00047     visited = dfs.visited(4);
00048     if (visited)
00049         return 1;
00050     visited = dfs.visited(2);
00051     if (!visited)
00052         return 1;
00053     return 0;
00054 }

```

#### 4.25.1.3 testDigraph()

```

int testDigraph ( )
00004 {
00005
00006     DirectedGraph G(5);
00007     G.addEdge(0, 1);
00008     G.addEdge(0, 2);
00009     G.addEdge(1, 2);
00010     G.addEdge(2, 0);
00011     G.addEdge(2, 3);
00012     G.addEdge(3, 3);
00013
00014     int v = G.getNumVertices();
00015     int e = G.getNumEdges();
00016     std::vector<int> neighbors = G.getNeighbors(2);
00017
00018     if (v != 5)
00019         return 1;
00020     if (e != 6)
00021         return 1;
00022     if (neighbors.size() != 2)
00023         return 1;
00024     if (neighbors[0] != 0)
00025         return 1;
00026     if (neighbors[1] != 3)
00027         return 1;
00028
00029     return 0;
00030 }

```

## 4.26 test\_GeneticAlgorithm.cpp File Reference

```

#include <iostream>
#include <vector>
#include <cmath>
#include "Unittests.h"

```

### Functions

- bool [validity](#) (const std::vector< int > &chromosome)
- double [Rosenbrock](#) (const std::vector< int > &x)

### 4.26.1 Function Documentation

#### 4.26.1.1 Rosenbrock()

```

double Rosenbrock (
    const std::vector< int > & x )
00012 {

```

```

00013     double fx = 0.0;
00014     for (size_t i = 0; i < x.size() - 1; ++i)
00015     {
00016         double term1 = std::pow(x[i + 1] - std::pow(x[i], 2), 2);
00017         double term2 = std::pow(1 - x[i], 2);
00018         fx += 100.0 * term1 + term2;
00019     }
00020     return 1.0 / (1.0 + fx); // turn a minimization problem into a maximization problem
00021 }

```

#### 4.26.1.2 validity()

```

bool validity (
    const std::vector< int > & chromosome )
00007 {
00008     return true;
00009 }

```

## 4.27 test\_GeneticAlgorithm.cpp

[Go to the documentation of this file.](#)

```

00001 #include <iostream>
00002 #include <vector>
00003 #include <cmath>
00004 #include "Unittests.h"
00005
00006 bool validity(const std::vector<int> &chromosome)
00007 {
00008     return true;
00009 }
00010
00011 double Rosenbrock(const std::vector<int> &x)
00012 {
00013     double fx = 0.0;
00014     for (size_t i = 0; i < x.size() - 1; ++i)
00015     {
00016         double term1 = std::pow(x[i + 1] - std::pow(x[i], 2), 2);
00017         double term2 = std::pow(1 - x[i], 2);
00018         fx += 100.0 * term1 + term2;
00019     }
00020     return 1.0 / (1.0 + fx); // turn a minimization problem into a maximization problem
00021 }
00022
00023 UnitTestGA::~~UnitTestGA(){};
00024
00025 void UnitTestGA::test_mutation() {
00026     std::cout << "=====Test Mutation=====\\n";
00027     std::vector<int> init_chromosome = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
00028     Individual<int> init_individual(init_chromosome);
00029     Individual<int> result_individual(init_chromosome);
00030     ga->MUTATION_RATE = 0.5;
00031
00032     ga->mutate(result_individual);
00033     assert(init_individual.chromosome != result_individual.chromosome);
00034     std::cout << "completed" << std::endl;
00035 }
00036
00037 void UnitTestGA::test_select_parent() {
00038     std::cout << "=====Test Select Parent=====\\n";
00039     int result;
00040     std::vector<double> cumulative_fitness = {-1.0, 1.0, 1.0};
00041
00042     result = ga->selectParent(cumulative_fitness);
00043     assert(result == 1);
00044     std::cout << "completed" << std::endl;
00045 }
00046
00047 void UnitTestGA::test_global_optimum() {
00048     std::cout << "=====Test Global Optimum=====\\n";
00049     // configure GA
00050     ga->setValidity(validity);
00051     ga->setFitness(Rosenbrock);
00052     ga->verbose = false;
00053     CrossoverType type = CrossoverType::UNIFORM;
00054     // Known global optimum for Rosenbrock function
00055     std::vector<int> expected_chromosome = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1};

```



```

00056     int success_count = 0;
00057     int num_runs = 10;
00058
00059     // Run for 10 times to see if the algorithm can find the global optimum
00060     // if 50% of the time can find the global optimum, then the test is passed
00061     for (int i = 0; i < num_runs; i++) {
00062         auto best = ga->optimize(type);
00063         if (best.chromosome == expected_chromosome) {
00064             success_count++;
00065         }
00066     }
00067     assert(success_count >= num_runs / 2);
00068     std::cout << "successfully found global optimum for Rosenbrock: " << success_count;
00069     std::cout << "/" << num_runs << " times!" << std::endl;
00070     std::cout << "completed" << std::endl;
00071 }
00072
00073 void UnitTestGA::run() {
00074     test_mutation();
00075     test_select_parent();
00076     test_global_optimum();
00077 }
00078

```

## 4.28 test\_Individual.cpp File Reference

```

#include <iostream>
#include <cassert>
#include <cmath>
#include "test_GeneticAlgorithm.h"

```

### Functions

- int [main](#) ()

### 4.28.1 Function Documentation

#### 4.28.1.1 main()

```

int main ( )
00047 {
00048     UnitTestIndividual u;
00049     u.test_constructor();
00050     // u.test_calculateFitness();
00051     u.test_Get_Set_fitness();
00052     u.test_Get_vectorLength();
00053 }

```

## 4.29 test\_RandomCircuit.cpp File Reference

```

#include "CSimulator.h"
#include "CCircuit.h"
#include <iostream>
#include <vector>
#include <random>

```

## Functions

- `std::vector< int > generateRandomVector (int size, int minRange, int maxRange)`
- `int main ()`

### 4.29.1 Function Documentation

#### 4.29.1.1 generateRandomVector()

```
std::vector< int > generateRandomVector (
    int size,
    int minRange,
    int maxRange )
00008 {
00009     std::random_device rd;
00010     std::mt19937 rng(rd());
00011     std::uniform_int_distribution<int> dist(minRange, maxRange);
00012
00013     std::vector<int> randomVector(size);
00014
00015     for (int i = 0; i < size; i++)
00016     {
00017         randomVector[i] = dist(rng);
00018     }
00019
00020     return randomVector;
00021 }
```

#### 4.29.1.2 main()

```
int main ( )
00024 {
00025     struct Circuit_Parameters default_circuit_parameters;
00026     bool validity = false;
00027     int feeder = 0;
00028     int num_units = 5;
00029     std::vector<int> circuit_vector;
00030     circuit_vector.push_back(feeder);
00031     while (!validity)
00032     {
00033         std::vector<int> cvec = generateRandomVector(2 * num_units, 0, num_units + 1);
00034         for (const auto &i : cvec)
00035         {
00036             circuit_vector.push_back(i);
00037         }
00038         validity = Circuit::Check_Validity(circuit_vector);
00039         if (!validity)
00040         {
00041             circuit_vector.clear();
00042             circuit_vector.push_back(feeder);
00043         }
00044     }
00045     double value = Evaluate_Circuit(circuit_vector, default_circuit_parameters);
00046     Circuit circuit(circuit_vector);
00047     circuitToFile();
00048     std::cout << "Given test value:" << value << std::endl;
00049 }
```

## 4.30 Unittests.h File Reference

```
#include <cstdlib>
#include <cassert>
#include "CSimulator.h"
#include "CCircuit.h"
#include "CUnit.h"
#include "GeneticAlgorithm.hpp"
```

## Classes

- class [UnitTestGA](#)
- class [UnitTestIndividual](#)
- class [UnitTestCrossover](#)
- class [UnitTestCircuit](#)

## 4.31 Unittests.h

[Go to the documentation of this file.](#)

```

00001 #pragma once
00002 #include <cstdlib>
00003 #include <cassert>
00004 #include "CSimulator.h"
00005 #include "CCircuit.h"
00006 #include "CUnit.h"
00007 #include "GeneticAlgorithm.hpp"
00008
00009
00010 class UnitTestGA
00011 {
00012 public:
00013     // SetUp
00014     int POPULATION_SIZE = 100;
00015     int CHROMOSOME_SIZE = 10;
00016     int CHROMOMIN = 0;
00017     int CHROMOMAX = 10;
00018     int MAX_GENERATION = 2000;
00019     int CONV_GEN = 2000;
00020     double CROSSOVER_RATE = 0.8;
00021     double MUTATION_RATE = 0.15;
00022
00023     GA<int> *ga = new GA<int>(POPULATION_SIZE,
00024                               CHROMOSOME_SIZE,
00025                               CHROMOMIN,
00026                               CHROMOMAX,
00027                               MAX_GENERATION,
00028                               CROSSOVER_RATE,
00029                               MUTATION_RATE
00030                               );
00031
00032     UnitTestGA(int populationSize, int chromosomeSize, int chromomin, int chromomax,
00033               int maxGeneration, double crossoverRate, double mutationRate)
00034         : POPULATION_SIZE(populationSize), CHROMOSOME_SIZE(chromosomeSize),
00035           CHROMOMIN(chromomin),
00036           CHROMOMAX(chromomax), MAX_GENERATION(maxGeneration), CROSSOVER_RATE(crossoverRate),
00037           MUTATION_RATE(mutationRate) {};
00038
00039
00040     ~UnitTestGA();
00041
00042     void test_mutation();
00043     void test_select_parent();
00044
00045     void test_global_optimum();
00046
00047     void run();
00048 };
00049
00050
00051 class UnitTestIndividual
00052 {
00053 public:
00054     UnitTestIndividual();
00055     ~UnitTestIndividual();
00056
00057     void test_constructor();
00058
00059     // void test_calculateFitness();
00060
00061     void test_Get_Set_fitness();
00062
00063     void test_Get_numUnits();
00064
00065     void test_Get_vectorLength();
00066
00067 protected:
00068     Individual<int> individual;

```

```
00099 };
00100
00101 class UnitTestCrossover
00102 {
00103 public:
00107     UnitTestCrossover();
00111     ~UnitTestCrossover();
00112
00116     void test_one_pt_crossover();
00117
00121     void test_two_pts_crossover();
00122
00126     void test_mult_pts_crossover();
00127
00131     void test_uniform_crossover();
00132 };
00133
00134 class UnitTestCircuit
00135 {
00136 public:
00140     UnitTestCircuit();
00144     ~UnitTestCircuit();
00145
00146     void test_circuit_evaluation();
00147     void test_circuit_validity();
00148     void test_circuit_unit();
00149     void run();
00150 };
```

# Index

- ~CUnit
  - CUnit, [16](#)
- ~Circuit
  - Circuit, [6](#)
- ~GA
  - GA< T >, [26](#)
- ~Stream
  - Stream, [41](#)
- ~UnitTestCircuit
  - UnitTestCircuit, [43](#)
- ~UnitTestCrossover
  - UnitTestCrossover, [45](#)
- ~UnitTestGA
  - UnitTestGA, [48](#)
- ~UnitTestIndividual
  - UnitTestIndividual, [51](#)
- addEdge
  - DirectedGraph, [22](#)
- adjList
  - DirectedGraph, [24](#)
- bestFitnessHistory
  - GA< T >, [34](#)
- Calculate\_Circuit
  - CCircuit.h, [56](#), [57](#)
  - CSimulator.cpp, [70](#)
- calculateOutputFlowRates
  - CUnit, [16](#)
- CCircuit.cpp, [68](#)
  - markunits, [69](#)
- CCircuit.h, [55](#), [58](#)
  - Calculate\_Circuit, [56](#), [57](#)
- Check\_Validity
  - Circuit, [6](#)
- CHROMOMAX
  - GA< T >, [34](#)
  - UnitTestGA, [50](#)
- CHROMOMIN
  - GA< T >, [34](#)
  - UnitTestGA, [50](#)
- chromosome
  - Individual< T >, [40](#)
- CHROMOSOME\_SIZE
  - GA< T >, [34](#)
  - UnitTestGA, [50](#)
- Circuit, [5](#)
  - ~Circuit, [6](#)
  - Check\_Validity, [6](#)
  - Circuit, [6](#)
  - circuit\_vector, [8](#)
  - feeder, [8](#)
  - it, [8](#)
  - num\_units, [8](#)
  - toFile, [7](#)
  - units, [9](#)
- Circuit\_Parameters, [9](#)
  - get\_money, [9](#)
  - input\_Fger, [9](#)
  - input\_Fwaste, [10](#)
  - lose\_money, [10](#)
  - max\_iterations, [10](#)
  - tolerance, [10](#)
- circuit\_vector
  - Circuit, [8](#)
- CircuitOptimizer, [10](#)
  - feeder, [10](#)
  - num\_units, [10](#)
- conc
  - CUnit, [16](#)
- conc\_num
  - CUnit, [16](#)
  - Units, [43](#)
- CONV\_GEN
  - GA< T >, [34](#)
  - UnitTestGA, [50](#)
- cross\_cnt
  - GA< T >, [34](#)
- cross\_contrib
  - GA< T >, [34](#)
- crossover
  - Crossover< T >, [12](#)
  - test\_Crossover.cpp, [81](#)
- Crossover< T >, [11](#)
  - crossover, [12](#)
  - mult\_pts\_crossover, [12](#)
  - one\_pt\_crossover, [13](#)
  - two\_pts\_crossover, [13](#)
  - uniform\_crossover, [13](#)
- CROSSOVER\_RATE
  - GA< T >, [34](#)
  - UnitTestGA, [50](#)
- CrossoverType
  - GeneticAlgorithm.hpp, [65](#)
- CSimulator.cpp, [69](#)
  - Calculate\_Circuit, [70](#)
  - Evaluate\_Circuit, [72](#)
- CSimulator.h, [59](#), [62](#)

- Evaluate\_Circuit, 60, 61
- cumulativeFitness
  - GA< T >, 34
- CUnit, 14
  - ~CUnit, 16
  - calculateOutputFlowRates, 16
  - conc, 16
  - conc\_num, 16
  - CUnit, 15
  - F, 16
  - F\_old, 17
  - id, 17
  - input, 17
  - k\_ger, 17
  - k\_waste, 17
  - phi, 17
  - rho, 17
  - tail, 17
  - tails\_num, 18
  - V, 18
- CUnit.cpp, 73
- CUnit.h, 62, 63
- DepthFirstSearch, 18
  - DepthFirstSearch, 19
  - dfs, 19
  - marked, 20
  - visited, 20
- dfs
  - DepthFirstSearch, 19
- DirectedGraph, 20
  - addEdge, 22
  - adjList, 24
  - DirectedGraph, 21
  - getNeighbors, 22
  - getNumEdges, 22
  - getNumVertices, 23
  - operator<<, 23
  - writeToFile, 23
- DirectedGraph.cpp, 73
  - operator<<, 73
- DirectedGraph.hpp, 63, 64
- ELITISM
  - GA< T >, 35
- Evaluate\_Circuit
  - CSimulator.cpp, 72
  - CSimulator.h, 60, 61
- F
  - CUnit, 16
- F\_old
  - CUnit, 17
- feeder
  - Circuit, 8
  - CircuitOptimizer, 10
- fitness
  - GA< T >, 35
  - Individual< T >, 40
  - main.cpp, 75
  - fitness\_
    - GA< T >, 35
- GA
  - GA< T >, 26
- ga
  - UnitTestGA, 50
- GA< T >, 24
  - ~GA, 26
  - bestFitnessHistory, 34
  - CHROMOMAX, 34
  - CHROMOMIN, 34
  - CHROMOSOME\_SIZE, 34
  - CONV\_GEN, 34
  - cross\_cnt, 34
  - cross\_contrib, 34
  - CROSSOVER\_RATE, 34
  - cumulativeFitness, 34
  - ELITISM, 35
  - fitness, 35
  - fitness\_, 35
  - GA, 26
  - initPopulation, 27
  - MAX\_GENERATION, 35
  - mut\_cnt, 35
  - mut\_contrib, 35
  - mutate, 27
  - MUTATION\_RATE, 36
  - offspring, 36
  - optimize, 28
  - POP\_SIZE, 36
  - population, 36
  - scaledFitness, 36
  - selectParent, 29
  - setFitness, 29
  - setValidity, 29
  - toFile, 30
  - tournamentSelection, 30
  - updatePopulation, 31
  - validity\_, 36
  - verbose, 37
- generateRandomVector
  - test\_RandomCircuit.cpp, 88
- GeneticAlgorithm.cpp, 74
  - RandomDouble, 74
  - RandomInt, 74
- GeneticAlgorithm.hpp, 64, 66
  - CrossoverType, 65
  - MULT\_PTS, 66
  - ONE\_PT, 66
  - PARALLEL, 65
  - RandomDouble, 66
  - RandomInt, 66
  - TWO\_PTS, 66
  - UNIFORM, 66
- ger
  - Stream, 42
- get\_money

- Circuit\_Parameters, 9
- getNeighbors
  - DirectedGraph, 22
- getNumEdges
  - DirectedGraph, 22
- getNumVertices
  - DirectedGraph, 23
- id
  - CUnit, 17
- Individual
  - Individual< T >, 38
- individual
  - UnitTestIndividual, 53
- Individual< T >, 37
  - chromosome, 40
  - fitness, 40
  - Individual, 38
  - operator<, 38
  - operator>, 39
  - operator=, 38
  - operator==, 39
- initPopulation
  - GA< T >, 27
- input
  - CUnit, 17
- input\_Fger
  - Circuit\_Parameters, 9
- input\_Fwaste
  - Circuit\_Parameters, 10
- it
  - Circuit, 8
- k\_ger
  - CUnit, 17
- k\_waste
  - CUnit, 17
- lose\_money
  - Circuit\_Parameters, 10
- main
  - main.cpp, 75
  - test\_all.cpp, 77
  - test\_CircuitEvaluation.cpp, 79
  - test\_CircuitValidity.cpp, 80
  - test\_Cunit.cpp, 83
  - test\_DirectedGraph.cpp, 84
  - test\_Individual.cpp, 87
  - test\_RandomCircuit.cpp, 88
- main.cpp, 75
  - fitness, 75
  - main, 75
  - myCircuit, 76
  - validity, 76
- marked
  - DepthFirstSearch, 20
  - Units, 43
- markunits
  - CCircuit.cpp, 69
- MAX\_GENERATION
  - GA< T >, 35
  - UnitTestGA, 50
- max\_iterations
  - Circuit\_Parameters, 10
- MULT\_PTS
  - GeneticAlgorithm.hpp, 66
- mult\_pts\_crossover
  - Crossover< T >, 12
- mut\_cnt
  - GA< T >, 35
- mut\_contrib
  - GA< T >, 35
- mutate
  - GA< T >, 27
- MUTATION\_RATE
  - GA< T >, 36
  - UnitTestGA, 51
- myCircuit
  - main.cpp, 76
- num\_units
  - Circuit, 8
  - CircuitOptimizer, 10
- offspring
  - GA< T >, 36
- ONE\_PT
  - GeneticAlgorithm.hpp, 66
- one\_pt\_crossover
  - Crossover< T >, 13
- operator<
  - Individual< T >, 38
- operator<<
  - DirectedGraph, 23
  - DirectedGraph.cpp, 73
- operator>
  - Individual< T >, 39
- operator+
  - Stream, 41
- operator+=
  - Stream, 41
- operator=
  - Individual< T >, 38
- operator==
  - Individual< T >, 39
- optimize
  - GA< T >, 28
- PARALLEL
  - GeneticAlgorithm.hpp, 65
- phi
  - CUnit, 17
- POP\_SIZE
  - GA< T >, 36
- population
  - GA< T >, 36
- POPULATION\_SIZE

- UnitTestGA, 51
- printChromosome
  - test\_Crossover.cpp, 81
- RandomDouble
  - GeneticAlgorithm.cpp, 74
  - GeneticAlgorithm.hpp, 66
- RandomInt
  - GeneticAlgorithm.cpp, 74
  - GeneticAlgorithm.hpp, 66
- rho
  - CUnit, 17
- Rosenbrock
  - test\_GeneticAlgorithm.cpp, 85
- run
  - UnitTestCircuit, 44
  - UnitTestGA, 49
- scaledFitness
  - GA< T >, 36
- selectParent
  - GA< T >, 29
- setFitness
  - GA< T >, 29
- setValidity
  - GA< T >, 29
- Stream, 40
  - ~Stream, 41
  - ger, 42
  - operator+, 41
  - operator+=, 41
  - Stream, 41
  - waste, 42
- tail
  - CUnit, 17
- tails\_num
  - CUnit, 18
  - Units, 43
- test
  - test\_CircuitEvaluation.cpp, 79
- test\_all.cpp, 77
  - main, 77
- test\_Circuit.cpp, 77, 78
- test\_circuit\_evaluation
  - UnitTestCircuit, 44
- test\_circuit\_unit
  - UnitTestCircuit, 44
- test\_circuit\_validity
  - UnitTestCircuit, 44
- test\_CircuitEvaluation.cpp, 79
  - main, 79
  - test, 79
- test\_CircuitValidity.cpp, 79
  - main, 80
  - testValidity, 80
- test\_constructor
  - UnitTestIndividual, 52
- test\_Crossover.cpp, 80, 81
  - crossover, 81
  - printChromosome, 81
- test\_Cunit.cpp, 83
  - main, 83
- test\_DirectedGraph.cpp, 84
  - main, 84
  - testDFS, 84
  - testDigraph, 85
- test\_GeneticAlgorithm.cpp, 85, 86
  - Rosenbrock, 85
  - validity, 86
- test\_Get\_numUnits
  - UnitTestIndividual, 52
- test\_Get\_Set\_fitness
  - UnitTestIndividual, 52
- test\_Get\_vectorLength
  - UnitTestIndividual, 52
- test\_global\_optimum
  - UnitTestGA, 49
- test\_Individual.cpp, 87
  - main, 87
- test\_mult\_pts\_crossover
  - UnitTestCrossover, 46
- test\_mutation
  - UnitTestGA, 49
- test\_one\_pt\_crossover
  - UnitTestCrossover, 46
- test\_RandomCircuit.cpp, 87
  - generateRandomVector, 88
  - main, 88
- test\_select\_parent
  - UnitTestGA, 49
- test\_two\_pts\_crossover
  - UnitTestCrossover, 47
- test\_uniform\_crossover
  - UnitTestCrossover, 47
- testDFS
  - test\_DirectedGraph.cpp, 84
- testDigraph
  - test\_DirectedGraph.cpp, 85
- testValidity
  - test\_CircuitValidity.cpp, 80
- toFile
  - Circuit, 7
  - GA< T >, 30
- tolerance
  - Circuit\_Parameters, 10
- tournamentSelection
  - GA< T >, 30
- TWO\_PTS
  - GeneticAlgorithm.hpp, 66
- two\_pts\_crossover
  - Crossover< T >, 13
- UNIFORM
  - GeneticAlgorithm.hpp, 66
- uniform\_crossover
  - Crossover< T >, 13
- Units, 42



- conc\_num, [43](#)
- marked, [43](#)
- tails\_num, [43](#)
- Units, [42](#)
- units
  - Circuit, [9](#)
- UnitTestCircuit, [43](#)
  - ~UnitTestCircuit, [43](#)
  - run, [44](#)
  - test\_circuit\_evaluation, [44](#)
  - test\_circuit\_unit, [44](#)
  - test\_circuit\_validity, [44](#)
  - UnitTestCircuit, [43](#)
- UnitTestCrossover, [45](#)
  - ~UnitTestCrossover, [45](#)
  - test\_mult\_pts\_crossover, [46](#)
  - test\_one\_pt\_crossover, [46](#)
  - test\_two\_pts\_crossover, [47](#)
  - test\_uniform\_crossover, [47](#)
  - UnitTestCrossover, [45](#)
- UnitTestGA, [48](#)
  - ~UnitTestGA, [48](#)
  - CHROMOMAX, [50](#)
  - CHROMOMIN, [50](#)
  - CHROMOSOME\_SIZE, [50](#)
  - CONV\_GEN, [50](#)
  - CROSSOVER\_RATE, [50](#)
  - ga, [50](#)
  - MAX\_GENERATION, [50](#)
  - MUTATION\_RATE, [51](#)
  - POPULATION\_SIZE, [51](#)
  - run, [49](#)
  - test\_global\_optimum, [49](#)
  - test\_mutation, [49](#)
  - test\_select\_parent, [49](#)
  - UnitTestGA, [48](#)
- UnitTestIndividual, [51](#)
  - ~UnitTestIndividual, [51](#)
  - individual, [53](#)
  - test\_constructor, [52](#)
  - test\_Get\_numUnits, [52](#)
  - test\_Get\_Set\_fitness, [52](#)
  - test\_Get\_vectorLength, [52](#)
  - UnitTestIndividual, [51](#)
- Unittests.h, [88](#), [89](#)
- updatePopulation
  - GA< T >, [31](#)
- V
  - CUnit, [18](#)
- validity
  - main.cpp, [76](#)
  - test\_GeneticAlgorithm.cpp, [86](#)
- validity\_
  - GA< T >, [36](#)
- verbose
  - GA< T >, [37](#)
- visited
  - DepthFirstSearch, [20](#)
- waste
  - Stream, [42](#)
- writeToFile
  - DirectedGraph, [23](#)