GERARDIUM RUSH - MINERAL CIRCUIT OPTIMIZER

v1.0.0

Delivered by Pentlandite

Chapter 1

GERARDIUM RUSH - MINERAL CIRCUIT OPTIMIZER

This is an implementation for the **ACS - Gerardium Rush** project delivered by group **Pentlandite**.
This project is designed to optimize circuit configurations using genetic algorithms to enhance the recovery of valuable materials in separation technologies. The goal is to create an efficient and effective tool for optimizing circuits in minerals processing, particularly for the recovery of "gerardium". This project combines the power of genetic algorithms with mass balance simulations to find the optimal circuit configuration.

1.0.1 Overview

The project is organized into several key components, each with specific responsibilities:

1. **Circuit Simulation (CSimulator.cpp)**:
    - Core simulation logic for evaluating circuit performance
    - Simulates the mass balance for given circuit configurations.
    - Utilizes classes like `CCircuit` and `CUnit` to represent and manage circuit units and their interactions.
2. **Genetic Algorithm (Genetic_Algorithm.cpp)**:
    - Implements the genetic algorithm for optimizing circuit configurations.
    - Includes various operators for selection, crossover, and mutation.
    - Parallel processing enabled using OpenMP for efficient computation.

1.0.2 Features

- **Genetic Algorithm Optimization**: Efficiently explore numerous circuit configurations using genetic algorithms to find the most effective setups.
- **Mass Balance Simulation**: Simulate the mass balance for each circuit to ensure accurate recovery and grade calculations, ensuring the reliability of the proposed solutions.
- **Parallel Processing**: Leverage OpenMP for faster computations and simulations, enhancing performance and reducing computation time.
- **Comprehensive Unit Tests**: Ensure the correctness of the circuit simulator and genetic algorithm through extensive unit tests, providing a robust and reliable tool.
- **Visualization Tools**: Utilize Python scripts and Jupyter notebooks for post-processing and visualizing the results, helping to analyze and interpret the outcomes effectively.
- **Documentation**: Detailed API documentation and project reports to facilitate understanding and usage, making it easier for users to get started and for developers to extend the project.

1.0.3 Requirements

The project uses CMake for building and requires several tools and libraries. Below are the required tools and libraries:

Tool/Library	Purpose	Version
CMake	For managing the build process	3.10 or higher
GCC and G++	For compiling the code	13.0 or higher
OpenMP	For parallel processing	4.5 or higher
Python	For post-processing and visualization	3.10 or higher
Graphviz	For generating diagrams	2.50.0 or higher
Matplotlib	For plotting and visualizing data	3.8.2 or higher
Pandas	For data manipulation and analysis	2.1.3 or higher

To install these requirements on a Debian-based system, you can use the following commands:

```
$ sudo apt update
$ sudo apt install cmake gcc g++ libomp-dev python3 python3-pip graphviz
$ pip3 install matplotlib pandas
```

1.0.4 Installation

To build the project, you need to have CMake and GCC installed on your system.

1. Clone the repository:
```
$ git clone https://github.com/yourusername/circuit-optimizer.git
```
2. Create a build directory and navigate to it:
```
$ mkdir build
$ cd build
```
3. Configure the project using CMake:
```
$ cmake .. -DCMAKE_C_COMPILER=gcc -DCMAKE_CXX_COMPILER=g++
```
4. Build the project:
```
$ cmake --build .
```

1.0.5 Usage

Run Executable

To run the Circuit Optimizer, execute the following commands from the `build` directory:

```
$ export OMP_NUM_THREADS=10
$ ./bin/Circuit_Optimizer
```

Run Tests

The project includes a set of unit tests to verify the functionality of the circuit simulator and the genetic algorithm. The tests are located in the `tests` directory.

To run the tests, use the following command:

```
$ ctest --output-on-failure
```

Run Visualization

To run the visualization tool, use the following command:

```
$ python3 ../post_process/visualize.py <vector> <performance> <recovery> <grade>
```

Run Scripts (Recommended)

Alternatively, you can use the provided shell scripts:

- To run the optimizer, go to the root directory and run the following command:
```
$ ./scripts/run.sh
```
- To run the tests, go to the root directory and run the following command:
```
$ ./scripts/run_test.sh
```

You can modify the `OMP_NUM_THREADS` in corresponding shell scripts.

1.0.6 Parameter Setting

In `src/main.cpp`, you can set the Genetic Algorithm parameters and Circuit Simulator parameters. Here is an example:

```
int generations = 1000;
int populationSize = 100;
std::string selection = "NaryTournamentSelection";
// std::string selection = "RankSelection";
// std::string selection = "RouletteWheelSelection";
int tournamentSize = 5;
std::string crossover = "TwoPointCrossover";
// std::string crossover = "UniformCrossover";
// std::string crossover = "PureSinglePointCrossover";
double crossRate = 0.8;
std::string mutation = "GuidedMutation";
// std::string mutation = "GenProgMutation";
double mutationRate = 0.1;
double eliteRate = 0.2;
double initialTemp = 1000.0;
double deltT = 0.85;
unsigned int randomSeed = 0;

double tolerance = 1e-6;
int maxIteration = 1000;

Algorithm_Parameters ga_params(generations, populationSize, selection, tournamentSize, crossover, crossRate,
      mutation, mutationRate, eliteRate, initialTemp, deltT, randomSeed);

CircuitParameters c_params(tolerance, maxIteration);
```

Genetic Algorithm Operators

- **Selection**:
  - `NaryTournamentSelection`: Selects the best individuals based on tournament competition.
  - `RankSelection`: Selects individuals based on their rank within the population.
  - `RouletteWheelSelection`: Selects individuals based on their fitness proportionally.
- **Crossover**:
  - `TwoPointCrossover`: Combines genes from two parents at two crossover points.
  - `UniformCrossover`: Combines genes from two parents based on a uniform distribution.
  - `PureSinglePointCrossover`: Combines genes from two parents at a single crossover point.
- **Mutation**:
  - `GuidedMutation`: Performs guided mutation based on specific rules or heuristics.
  - `GenProgMutation`: Performs genetic programming mutation by changing operations and parameters.

1.0.7 Project Structure

```
.
 CMakeLists.txt                            # Main CMake configuration file for the entire project
 Doxygen.config                            # Configuration file for generating documentation with Doxygen
 LICENSE.txt                               # License information for the project
 README.md                                 # Project's readme file with instructions and documentation
 build                                     # Build directory containing compiled binaries and test
      executables
    bin                                   # Directory for main binary executables
       Circuit_Optimizer                 # Main executable for the Circuit Optimizer project
    tests                                 # Directory for test executables and related files
        bin                               # Binaries for the tests
           test_circuit_simulator        # Test executable for circuit simulator
           test_genetic_algorithm        # Test executable for genetic algorithm
           test_validity_checker         # Test executable for validity checker
        ...                               # Other test-related files
 circuit_diagrams                          # Directory for circuit diagrams
    circuit.png                           # Example circuit diagram image
 docs                                      # Documentation files
    API_Document.pdf                      # API documentation in PDF format
    CCircuit_Document_Analysis.md         # Markdown document with circuit design analysis
    CCircuit_Document_Analysis.pdf        # PDF document with circuit design analysis
    CSimulator_Document_and_Analysis.md   # Markdown document with simulator performance analysis
    CSimulator_Document_and_Analysis.pdf  # PDF document with simulator performance analysis
    GeneticAlgorithm_Design_Document.md   # Markdown document with algorithm design illstration
```

```
   GeneticAlgorithm_Design_Document.pdf   # PDF document with algorithm design illstration
   GeneticAlgorithm_Params_Analysis.md    # Markdown document with algorithm & circuit params analysis
   GeneticAlgorithm_Params_Analysis.pdf   # Markdown document with algorithm & circuit params analysis
   Problem_Statement.pdf                  # Project problem statement
   doxygen                                # Directory for Doxygen-generated documentation files
include                                    # Header files for the project
   CSimulator.h                           # Header for the circuit simulator
   Genetic_Algorithm.h                    # Header for the genetic algorithm
   circuit                                # Directory for circuit-related headers
      CCircuit.h                          # Header for the circuit class
      CUnit.h                             # Header for the unit class
   operators                              # Directory for operator-related headers
      Crossover                           # Crossover operator headers
         PureSinglePointCrossover.h       # Header for pure single point crossover
         TwoPointCrossover.h              # Header for two-point crossover
         UniformCrossover.h               # Header for uniform crossover
      Mutation                            # Mutation operator headers
         GenProgMutation.h                # Header for genetic programming mutation
         GuidedMutation.h                 # Header for guided mutation
      Selection                           # Selection operator headers
          NaryTournamentSelection.h       # Header for N-ary tournament selection
          RankSelection.h                 # Header for rank selection
          RouletteWheelSelection.h        # Header for roulette wheel selection
   utils                                  # Utility headers
       Helper.h                           # Header for helper functions
       Individual.h                       # Header for individual class
       Parameter.h                        # Header for parameter handling
       tqdm.hpp                           # Header for progress bar utility
investigate                                # Directory for investigation scripts and logs
   CCircuit.cpp                           # Circuit source file for investigation
   evaluate_circuit.ipynb                 # Jupyter notebook for evaluating circuits
   evaluate_circuit_time.ipynb            # Jupyter notebook for evaluating circuit times
   execution_times.txt                    # Log file for execution times
   timings_thread.log                     # Log file for thread timings
log                                        # Directory for log files
   2024-05-23_21-28-04                    # Log directory with timestamp
       circuit_info.log                   # Log file for circuit information
post_process                               # Directory for post-processing scripts
   visualize.ipynb                        # Jupyter notebook for visualization
   visualize.py                           # Python script for visualization
scripts                                    # Shell scripts for running the project
   run.sh                                 # Script to run the main optimizer
   run_test.sh                            # Script to run the tests
src                                        # Source code for the project
   CMakeLists.txt                         # CMake configuration file for source code
   CSimulator.cpp                         # Source file for the circuit simulator
   Genetic_Algorithm.cpp                  # Source file for the genetic algorithm
   circuit                                # Directory for circuit-related source files
      CCircuit.cpp                        # Source file for the circuit class
      CUnit.cpp                           # Source file for the unit class
   main.cpp                               # Main source file for the project
   operators                              # Directory for operator-related source files
      Crossover                           # Crossover operator source files
         PureSinglePointCrossover.cpp     # Source file for pure single point crossover
         TwoPointCrossover.cpp            # Source file for two-point crossover
         UniformCrossover.cpp             # Source file for uniform crossover
      Mutation                            # Mutation operator source files
         GenProgMutation.cpp              # Source file for genetic programming mutation
         GuidedMutation.cpp               # Source file for guided mutation
      Selection                           # Selection operator source files
          NaryTournamentSelection.cpp     # Source file for N-ary tournament selection
          RankSelection.cpp               # Source file for rank selection
          RouletteWheelSelection.cpp      # Source file for roulette wheel selection
   utils                                  # Utility source files
       Helper.cpp                         # Source file for helper functions
tests                                      # Test files
    CMakeLists.txt                         # CMake configuration file for tests
    test_circuit_simulator.cpp             # Test file for circuit simulator
    test_circuit_time.cpp                  # Test file for circuit timing
    test_genetic_algorithm.cpp             # Test file for genetic algorithm
    test_validity_checker.cpp              # Test file for validity checker
```

1.0.8 Contributing

Contributions are welcome! Please follow these steps:

1. Fork the repository.
2. Create a new branch (`git checkout -b feature-branch`).
3. Make your changes.
4. Commit your changes (‘git commit -m 'Add new feature’) **.**

5. `Push to the branch` (git push origin feature-branch`).
6. Create a new Pull Request.

1.0.9 License

This project is licensed under the MIT License. See the `LICENSE.txt` file for more details.

Chapter 2

Optimised Circuit Configuration vs Economic Factors

2.1 Introduction

To get a heuristic circuit design we utilised our base case solution and varied 3 economic factors (Number of units in the circuit, Price paid per gerardium relative to waste, Purity of the input feed) independently to observe how the optimal circuit configuration changes with each factor. This allows us to identify common patterns in the optimal circuit.

2.2 Base Case (10 Units with Specified Economic Factors)

From the base case circuit solution, we observed several features:

1. A row of units which handles further filtering of the tail streams. (Units 5, 0, 4, 7, 3)
2. A unit which takes in the concentrate streams from the row of units. (Unit 1)
3. A unit to handle one last filtering of the material before sending to the concentrate (Unit 8)
4. A unit which takes in the intermediate streams from this row of units, and recycles its tailing to the start of the tail stream unit row and directs its concentrate stream towards the concentrate unit (Unit 6)
5. A unit which allows recycling of material midway through the circuit (Unit 9)

2.3 Economic Factors:

2.3.1 1) Number of Units

Investigated circuits with a number of units from 5 to 15:
There are common features among the optimal configuration of circuits with a different number of units

- The concentrate and tailing streams are connected to only one other unit each
- Overall they have a similar structure to the base case as described above, these features includes:
  - A row of units to handle the tailing streams (This row is longer and more obvious as number of units increase)
  - 1 unit which handles the intermediate streams of these units
  - 1 unit which handles the concentrate stream of these units
- Circuits with more units, have units to handle recycling and further filtering between these structures, as seen in the base case and the circuit with 15 units below (2 units, Unit 6, 7, handles filtering before depositing in concentrate)

Circuit with 5 units:
Circuit with 15 units:

2.3.2 2) Price of Reward and Penalty

Investigated 4 different cases utilising circuits of 5 units, 10 units:

1. Reward of £100 $*$ 5 per kg
2. Reward of £100 $*$ 10 per kg
3. Penalties of £750 $*$ 5 per kg
4. Penalties of £750 $*$ 10 per kg

With 5 units:

- At higher penalty, the optimal circuit's intermediate streams showed tendency to recycle (Unit 3, 4, 6 conducts further filtering), following the route of the tail stream, as opposed to continuing forward in the circuit towards the concentrate
- For higher rewards, the circuit is comparable to the base case for 5 units, just with an additional unit to filter for the tailing stream (to extract as much value from this stream)

With 10 units:

- The same general structure mentioned multiple times above was observed for various degrees of penalities and rewards
- A similar trend to the circuit of 5 units was observed
- For higher rewards, there are more filters for tailing stream:

£1000 reward, 10 units circuit

- For a high penalty there is an increased number of units to filter out the material before depositing into the concentrate (Units 8, 4, 3), and fewer units to filter out tailing stream
- This is expected as we want to prevent as much waste in the concentrate as possible, therefore prioritise this over collecting material from tailing stream

£7500 penalty, 10 units circuit

2.3.3 3) Purity

Investigated a range of gerardium feed purity, from 10-90%:

- There was a general trend where an increase in purity of gerardium corresponded to more intermediate flow streams directly connected to the concentrate.
  - For an extreme case of 90% geranium and 10% waste for a circuit of 5 units it connects all unit's concentrate and intermediate streams directly to the concentrate.

90% gerardium feed, 5 units circuit:
90% gerardium feed, 10 units circuit:

- A similar trend, to a lesser degree, can be seen in the 10 units circuit, where earlier on in the row of units (Units 0, 8, 7) the intermediate streams directly connect to the concentrate, but later on in the row (with increasing waste content) the intermediate streams is recycled to the start of the tailing row. Note that there is no unit designated for recycling like in the base case.
- Overall the general design for having a row of units to filter out the tailing stream holds true for the range of purities explored.
- The circuit design differs at lower purities where it is closer to the base case with fewer units in a row and units utilised for further filtering prior to deposition to the concentrate.

2.4 Heuritic Circuit Design

Based on the above observations we can see some similarities in the design which could provide an insight into a heuristic circuit design to consider:

1. Having a row of units to filter out the tailing streams leading to the tail.
2. A unit to handle concentrate streams from this row of units.
3. Given sufficient units, the intermediate streams from these rows will be fed into another unit before either being sent to the concentrate unit/recycled to the top of the row of units

Additional Considerations:

1. Depending on the penalty cost, we can also arrange more units (if higher penalty) or less units (if lower penalty) before material reaches concentrate unit to further filter out the waste material from concentrate
2. Purity can also be taken into consideration when optimising a circuit. We can arrange more units for the tailing stream row (to extract as much valuable material from waste before deposal) the more pure the feed, the more units we can assign for this purpose

Chapter 3

Document & Analysis for Circuit Simulator

3.1 Introduction

This document provides a comprehensive design framework for the `CSimulator` class and its associated components. The `CSimulator` class is designed to simulate the mass balance of a separation circuit composed of multiple units, evaluate the performance of the circuit, and ensure that the circuit operates correctly under specified constraints.

3.2 1. Overview of `CSimulator`

3.2.1 1.1 Purpose

The purpose of the `CSimulator` class is to simulate the mass balance of a circuit that separates valuable material (gerardium) from waste. It calculates the mass flows through each unit in the circuit and determines the overall performance of the circuit.

3.2.2 1.2 Scope

The scope of the `CSimulator` includes:

- Parsing the circuit vector into a structured representation.
- Simulating the mass balance for the entire circuit.
- Evaluating the performance of the circuit based on the mass balance.
- Ensuring the circuit reaches a steady-state solution.

3.3 2. Class Design

3.3.1 2.1 UML Class Diagram

```
+------------------+        +----------------+
|   CSimulator     |        |     CUnit      |
+------------------+        +----------------+
| - circuit_vector: std::vector<int>        |
| - units: std::vector<CUnit>               |
| - feed_rate: double                       |
| - tolerance: double                       |
| - max_iterations: int                     |
+------------------+        +----------------+
| + CSimulator(circuit_vector, feed_rate,   |
|   tolerance, max_iterations)              |
| + EvaluateCircuit(): double               |
| + SimulateUnit(unit: CUnit): bool         |
| + SolveSteadyState(): bool                |
| + ParseCircuitVector(): void              |
+------------------+        +----------------+
                          | + CUnit(conc_num: int,        |
                          |   inter_num: int, tails_num: int)|
                          | + CalculateFlows(feed_rate: double, |
```

```
                          |   kC: double, kI: double): void|
+-----------------+        +----------------+
|     Attributes   |        |  Attributes   |
|  + concNum: int          |  + concNum: int            |
|  + interNum: int         |  + interNum: int           |
|  + tailsNum: int         |  + tailsNum: int           |
|  + feedGerardium: double  |  + feedGerardium: double   |
|  + feedWaste: double      |  + feedWaste: double        |
|  + newFeedGerardium: double |  + newFeedGerardium: double |
|  + newFeedWaste: double    |  + newFeedWaste: double    |
|  + differenceG: double     |  + differenceG: double     |
|  + differenceW: double     |  + differenceW: double     |
|  + concPtr: CUnit*         |  + concPtr: CUnit*         |
|  + interPtr: CUnit*        |  + interPtr: CUnit*        |
|  + tailsPtr: CUnit*        |  + tailsPtr: CUnit*        |
|  + mark: bool              |  + mark: bool              |
+-----------------+          +-----------------+
```

3.3.2 2.2 `CSimulator` Class

3.3.2.1 2.2.1 Attributes

- **circuit_vector:** A vector of integers representing the configuration of the circuit.
- **units:** A vector of `CUnit` objects, each representing a unit in the circuit.
- **feed_rate:** A double representing the feed rate of the input stream.
- **tolerance:** A double specifying the tolerance for convergence in the simulation.
- **max_iterations:** An integer representing the maximum number of iterations allowed for achieving convergence.

3.3.2.2 2.2.2 Methods

Constructor: The constructor initializes the `CSimulator` with a given circuit vector, feed rate, tolerance, and maximum number of iterations. It also calls a method to parse the circuit vector into a structured format.
EvaluateCircuit: This method evaluates the entire circuit and returns a performance score. It checks if the circuit reaches a steady-state solution within the specified maximum iterations. If convergence is not achieved, it returns a very negative value to indicate failure.
SimulateUnit: This method calculates the mass balance for a single unit in the circuit. It uses the feed rate and rate constants to compute the flow rates of the high-grade concentrate, intermediate stream, and tailings stream.
SolveSteadyState: This method iteratively solves for the steady-state mass balance of the entire circuit. It repeatedly updates the feed rates of all units and checks for convergence within the specified tolerance. If all units' feed rates stabilize within the tolerance, it returns true; otherwise, it continues until the maximum iterations are reached.
ParseCircuitVector: This method parses the circuit vector into a structured representation by creating `CUnit` objects for each unit in the circuit. It sets the connections between units based on the vector.

3.3.3 2.3 `CUnit` Class

3.3.4 2.3.1 Attributes

- **concNum:** An integer representing the index of the unit for the high-grade concentrate stream.
- **interNum:** An integer representing the index of the unit for the intermediate stream.
- **tailsNum:** An integer representing the index of the unit for the tailings stream.
- **feedGerardium:** A double representing the feed rate of gerardium to the unit.
- **feedWaste:** A double representing the feed rate of waste to the unit.
- **newFeedGerardium:** A double representing the new feed rate of gerardium after processing.
- **newFeedWaste:** A double representing the new feed rate of waste after processing.
- **differenceG:** A double representing the relative change in feed rate of gerardium.
- **differenceW:** A double representing the relative change in feed rate of waste.

- **concPtr:** A pointer to the CUnit receiving the concentrate stream.
- **interPtr:** A pointer to the CUnit receiving the intermediate stream.
- **tailsPtr:** A pointer to the CUnit receiving the tailings stream.
- **mark:** A boolean indicating whether the unit has been marked for traversal.

3.3.5 2.3.2 Methods

Constructor: The constructor initializes a CUnit with the indices for its concentrate, intermediate, and tailings streams. It also initializes the feed rates and other attributes.
CalculateFlows: This method calculates the mass flows for the unit based on the feed rate and given rate constants for the high-grade and intermediate streams. It computes the residence time and uses it to determine the flow rates for the concentrate, intermediate, and tailings streams.

3.4 3. Function Definitions

3.4.1 3.1 `CSimulator` Methods

Constructor:

- The constructor initializes the `CSimulator` with the provided circuit vector, feed rate, tolerance, and maximum iterations.
- It calls the `ParseCircuitVector` method to convert the circuit vector into a structured format, creating a collection of CUnit objects representing the units in the circuit.

EvaluateCircuit:

- This method first calls `SolveSteadyState` to iteratively solve for the steady-state mass balance.
- If the steady-state solution is not achieved, it returns a very negative value to indicate failure.
- If convergence is achieved, it calculates the overall performance score by summing the contributions from all units based on their concentrate and tailings rates.

SimulateUnit:

- This method calculates the mass flows for a single unit.
- It uses the feed rate and rate constants to compute the recovery rates for the high-grade concentrate and intermediate streams.
- It then determines the flow rates for the concentrate, intermediate, and tailings streams based on these recovery rates.

SolveSteadyState:

- This method iteratively updates the feed rates of all units and checks for convergence within the specified tolerance.
- It stores the old feed rates and compares them with the new feed rates to check for convergence.
- If all units' feed rates stabilize within the tolerance, it returns true; otherwise, it continues until the maximum number of iterations is reached.

ParseCircuitVector:

- This method converts the circuit vector into a structured representation by creating CUnit objects for each unit.
- It sets the connections between units based on the vector, ensuring each unit is correctly linked to its concentrate, intermediate, and tailings streams.

3.4.2 3.2 `CUnit` Methods

Constructor:

- The constructor initializes a `CUnit` with the indices for its concentrate, intermediate, and tailings streams.
- It sets the initial feed rates and other attributes.

CalculateFlows:

- This method calculates the mass flows for the unit based on the feed rate and given rate constants.
- It computes the residence time and uses it to determine the flow rates for the concentrate, intermediate, and tailings streams.
- The calculated flow rates are then assigned to the respective attributes of the `CUnit`.

3.5 4. Validation and Testing

3.5.1 4.1 Unit Tests

- **CSimulator Constructor:**
  - Test initialization with valid and invalid circuit vectors to ensure proper setup and error handling.
- **EvaluateCircuit:**
  - Validate performance evaluation by comparing the calculated performance score with expected values for known valid circuits.
- **SimulateUnit:**
  - Verify mass flow calculations by checking the output flow rates for various input feed rates and rate constants.
- **SolveSteadyState:**
  - Check convergence behavior for different circuit configurations and feed rates, ensuring the method stops iterating once convergence is achieved.
- **ParseCircuitVector:**
  - Test the parsing of circuit vectors to ensure that the resulting unit configurations match the expected structure.

3.5.2 4.2 Integration Tests

- **Complete Circuit Simulation:**
  - Validate end-to-end simulation for sample circuits, ensuring correct performance score and convergence for the entire circuit.
- **Boundary Conditions:**
  - Test simulator behavior with edge cases such as zero feed rate, maximum allowable iterations, and extreme values for tolerance.

3.6 5. Performance Considerations

- **Efficiency:**
  - Optimize the iterative solver to achieve faster convergence by improving the algorithm and reducing computational overhead.
- **Scalability:**
  - Ensure the simulator can handle larger circuits with more units without significant performance degradation, potentially through parallel processing.
- **Parallelization:**
  - Consider parallelizing unit simulations to leverage multi-core processors, which can improve performance and reduce simulation time.

3.7 6. Usage

- **User Guide:**
  - Provide detailed instructions for setting up the environment, running the simulator, and interpreting the output. Include examples and usage scenarios to help users understand how to use the tool effectively.
  - Ensure that the user guide includes steps to compile and run the simulator, required dependencies, and configuration options.
  - Example:

3.7.0.1 Setup

1. Clone the repository.
2. Navigate to the project directory.
3. Install required dependencies:

`shell sudo apt-get install g++ sudo apt-get install cmake`

3.7.0.2 Build the project

`shell mkdir build cd build cmake .. make`

3.7.0.3 Running the Simulator

To run the simulator, use the following command: `shell ./simulator <input_file>` Replace `<input_file>` with the path to your input configuration file.

3.7.0.4 Example

`shell ./simulator config.txt`

3.7.0.5 Interpreting the Output

The simulator will output the performance score and the detailed mass flow rates for each unit in the circuit.

- **Code Comments:**
  - Ensure all methods and key sections of the code are well-documented with comments explaining their functionality, parameters, and return values.

  - Example:

```
// Constructor for CSimulator
// Initializes the simulator with the given circuit vector, feed rate, tolerance, and max
iterations.
CSimulator::CSimulator(std::vector<int>& circuit_vector, double feed_rate, double tolerance, int
max_iterations) {
    // Initialization code
}
```

- **Examples:**
  - Include example circuit vectors and expected outputs to help users understand the simulator's behavior and validate its performance.
  - Example:

3.7.1 Example Circuit Vector

```
0, 1, 3, 3, 2, 2, 0, 4, 1, 1, 1, 0, 5
```

3.7.2 Expected Output

```
Performance Score: 110.25
Unit 0: Concentrate Rate: 1.5, Intermediate Rate: 0.3, Tailings Rate: 8.2
...
```

By following this detailed design framework, the `CSimulator` class can be developed to provide robust and efficient simulation of circuit configurations, meeting the requirements of the project and delivering accurate and reliable performance evaluations.

Chapter 4

Genetic Algorithm Design

4.1 Overview

This project employs a genetic algorithm to optimize circuit configurations. Genetic algorithms are heuristic search algorithms inspired by the process of natural selection. They use operations such as selection, crossover, and mutation to find the optimal solution. This document provides a detailed explanation of the design structure and optimization strategies implemented in the genetic algorithm.

4.2 UML Graph

The following documentation provides an in-depth look at the design and implementation of the genetic algorithm components within our project. The structure of the project is depicted in the accompanying diagram, which illustrates the relationships between various header and implementation files.
Relationships

- The central file `Operator.h` is included by all specific operator header files (`Crossover`, `Mutation`, `Selection`), establishing a common base or interface for different types of genetic algorithm operators.
- Each operator header (`.h`) file is paired with its corresponding implementation (`.cpp`) file, where the declared methods and classes are defined.
- `Genetic_Algorithm.h` includes various operator headers to utilize the defined crossover, mutation, and selection strategies within the genetic algorithm.
- The main program (`main.cpp`) and test file (`test_genetic_algorithm.cpp`) include `Genetic_Algorithm.h` to execute and test the genetic algorithm functionalities.

The diagram effectively illustrates the modular structure of the genetic algorithm, showing clear separations between declarations and implementations, and the hierarchical inclusion of various components. This modularity facilitates easy maintenance, testing, and extension of the genetic algorithm with new operators or strategies. The project is organized to ensure that each component can be developed and tested independently, making it easier to manage and extend the genetic algorithm framework.

4.3 Directory Structure

The following is the directory structure relevant to the genetic algorithm:

```
include
   Genetic_Algorithm.h                   # Main genetic algorithm components and functions.
   operators
      Crossover
         PureSinglePointCrossover.h    # Header for PureSinglePointCrossover class.
         TwoPointCrossover.h           # Header for TwoPointCrossover class.
         UniformCrossover.h            # Header for UniformCrossover class.
      Mutation
         GenProgMutation.h             # Header for GenProgMutation class.
         GuidedMutation.h              # Header for GuidedMutation class.
      Selection
          NaryTournamentSelection.h     # Header for NaryTournamentSelection class.
```

```
        RankSelection.h              # Header for RankSelection class.
        RouletteWheelSelection.h     # Header for RouletteWheelSelection class.
  utils
      Helper.h                        # Utility functions for logging and current time.
      Individual.h                    # Definition of the Individual class used in the genetic
    algorithm.
      Parameter.h                     # Definition of the parameter structures for the algorithm.
src
    Genetic_Algorithm.cpp               # Implementation of the main genetic algorithm components and
     functions.
    operators
       Crossover
          PureSinglePointCrossover.cpp  # Implementation of PureSinglePointCrossover class.
          TwoPointCrossover.cpp         # Implementation of TwoPointCrossover class.
          UniformCrossover.cpp          # Implementation of UniformCrossover class.
       Mutation
          GenProgMutation.cpp           # Implementation of GenProgMutation class.
          GuidedMutation.cpp            # Implementation of GuidedMutation class.
       Selection
           NaryTournamentSelection.cpp   # Implementation of NaryTournamentSelection class.
           RankSelection.cpp             # Implementation of RankSelection class.
           RouletteWheelSelection.cpp    # Implementation of RouletteWheelSelection class.
```

4.4 Major Components

4.4.1 1. Genetic_Algorithm.h and Genetic_Algorithm.cpp

These files define the core module of the genetic algorithm, including the main flow and operations of the algorithm, such as population initialization, parent selection, crossover, mutation, and generation of new populations.

4.4.1.1 Key Functions

- `initializeRandomSeed`: Initializes the random seed.
- `initializePopulation`: Initializes the population.
- `elitism`: Implements the elitism strategy, preserving the best individuals.
- `optimize`: The main optimization function, executing the main loop of the genetic algorithm.

4.4.2 2. Selection Operators

Selection operators are responsible for selecting individuals from the current population to serve as parents for the next generation.

4.4.2.1 Files

- `NaryTournamentSelection.h` / `NaryTournamentSelection.cpp`: n-ary tournament selection, randomly selects n individuals to compete, and selects the best one.
- `RankSelection.h` / `RankSelection.cpp`: Rank-based selection, individuals are sorted by fitness, and selection probability is based on rank.
- `RouletteWheelSelection.h` / `RouletteWheelSelection.cpp`: Roulette wheel selection, selection probability is proportional to fitness.

4.4.3 3. Crossover Operators

Crossover operators combine the genes of parent individuals to create new offspring.

4.4.3.1 Files

- `PureSinglePointCrossover.h` / `PureSinglePointCrossover.cpp`: Single-point crossover, swaps genes between two parents at a random point.
- `TwoPointCrossover.h` / `TwoPointCrossover.cpp`: Two-point crossover, swaps genes between two random points.

- UniformCrossover.h / UniformCrossover.cpp: Uniform crossover, randomly selects genes from each parent.

4.4.4 4. Mutation Operators

Mutation operators introduce random changes to the genes of an individual, increasing the diversity of the population.

4.4.4.1 Files

- GenProgMutation.h / GenProgMutation.cpp: General-purpose mutation, randomly modifies the genes of an individual.
- GuidedMutation.h / GuidedMutation.cpp: Guided mutation, modifies genes based on specific rules or heuristics.

4.4.5 5. Utilities

Auxiliary functions and structures definitions.

4.4.5.1 Files

- Helper.h / Helper.cpp: Contains auxiliary functions like logging results and getting the current time.
- Individual.h: Defines the individual structure, including the gene sequence and fitness.
- Parameter.h: Defines the structure for algorithm parameters, including population size, crossover rate, mutation rate, etc.

4.5 Optimization Strategies

4.5.1 1. Elitism Strategy

The elitism strategy in genetic algorithms ensures that the best-performing individuals from the current generation are preserved and carried over to the next generation. This helps in maintaining the quality of solutions over generations and prevents the loss of the best solutions found so far.

4.5.1.1 Implementation Details

The implementation of the elitism strategy is done in the `elitism` function in Genetic_Algorithm.cpp. This function sorts the population based on fitness and selects the top individuals to be directly transferred to the new population.

4.5.1.2 Code Explanation

`**elitism**`
The `elitism` function takes the current population and a reference to a new population vector where the elite individuals will be stored. It also takes the algorithm parameters which include the proportion of the population to be preserved as elites.

```
void elitism(const std::vector<Individual> &population, std::vector<Individual> &newPopulation, const
    Algorithm_Parameters &params) {
    // Create a copy of the population to sort
    std::vector<Individual> sortedPopulation = population;

    // Sort the population based on fitness in descending order
    std::sort(sortedPopulation.begin(), sortedPopulation.end(), [](const Individual &a, const Individual &b)
      {
        return a.fitness > b.fitness;
    });

    // Calculate the number of elite individuals to be preserved
    int numElites = static_cast<int>(params.populationSize * params.elitePercentage);

    // Add the top elite individuals to the new population
    for (int i = 0; i < numElites; i++) {
```

```
        // If the fitness of the individual is invalid, stop adding more elites
        if (sortedPopulation[i].fitness == INVALID_FITNESS) {
            break;
        }
        newPopulation.push_back(sortedPopulation[i]);
    }
}
```

4.5.1.3 Steps in the `elitism` Function

1. **Copy the Population**: A copy of the current population is created to avoid modifying the original population directly.
```
std::vector<Individual> sortedPopulation = population;
```
2. **Sort the Population**: The copied population is sorted based on the fitness values of the individuals in descending order. This means the individuals with the highest fitness values come first.
```
std::sort(sortedPopulation.begin(), sortedPopulation.end(), [](const Individual &a, const Individual
&b) {
    return a.fitness > b.fitness;
});
```
3. **Calculate the Number of Elites**: The number of elite individuals to be preserved is calculated based on the elite percentage parameter.
```
int numElites = static_cast<int>(params.populationSize * params.elitePercentage);
```
4. **Add Elite Individuals to the New Population**: The top elite individuals are added to the new population. If an individual's fitness is invalid, the process stops to avoid adding unfit individuals.
```
for (int i = 0; i < numElites; i++) {
    if (sortedPopulation[i].fitness == INVALID_FITNESS) {
        break;
    }
    newPopulation.push_back(sortedPopulation[i]);
}
```

4.5.1.4 Integration with the Main Optimization Loop

In the main optimization function (`optimize`), the `elitism` function is called to add the best individuals from the current population to the new population before proceeding with the selection, crossover, and mutation operations.
```
std::vector<Individual> newPopulation;
elitism(population, newPopulation, params);
```
By preserving the best individuals, the elitism strategy ensures that the genetic algorithm does not lose the highest quality solutions found in each generation, thereby maintaining or improving the overall quality of the population over time.

4.5.2 2. Selection Strategies

Selection strategies are critical in genetic algorithms as they determine how individuals are chosen to create the next generation. The implemented selection strategies in this project include Nary Tournament Selection, Rank Selection, and Roulette Wheel Selection. Each strategy has its advantages in maintaining diversity and preventing premature convergence.

4.5.2.1 Nary Tournament Selection

Nary Tournament Selection is a method where 'n' individuals are randomly chosen from the population, and the one with the highest fitness among them is selected as a parent. This process helps in maintaining a healthy diversity in the population and prevents premature convergence to suboptimal solutions.

4.5.2.2 Rank Selection

Rank Selection assigns selection probabilities to individuals based on their rank rather than their raw fitness values. This ensures that even individuals with lower fitness have a chance of being selected, thereby maintaining diversity in the population.

4.5.2.3 Roulette Wheel Selection

Roulette Wheel Selection (also known as fitness proportionate selection) assigns selection probabilities proportional to the individuals' fitness values. Individuals with higher fitness have a higher chance of being selected, but there is still a probability for lower fitness individuals to be chosen, thus maintaining diversity.

4.5.3 3. Crossover Strategies

Crossover strategies play a vital role in genetic algorithms by combining the genetic information of parent individuals to create offspring for the next generation. This process helps in exploring the search space and finding better solutions.

4.5.3.1 PureSinglePointCrossover

PureSinglePointCrossover involves selecting a random crossover point and exchanging the subsequences after this point between two parent individuals. This strategy helps to combine different features from the parents, potentially leading to better offspring.

4.5.3.2 TwoPointCrossover

TwoPointCrossover is similar to the single-point crossover but involves selecting two random points. The segments between these two points are exchanged between the parents, leading to potentially more diverse offspring.

4.5.3.3 UniformCrossover

UniformCrossover randomly selects genes from each parent, independent of their positions, resulting in a higher level of genetic diversity in the offspring. This method can be particularly effective in preventing premature convergence.

4.5.4 4. Mutation Strategies

Mutation strategies introduce variability in the genetic algorithm population, helping to explore the search space more thoroughly and avoid local optima.

4.5.4.1 GenProgMutation

GenProgMutation introduces random changes to the genes of an individual. This randomness helps the algorithm to avoid being trapped in local optima by exploring new areas of the search space.

4.5.4.2 GuidedMutation

GuidedMutation uses heuristic information to guide the mutation process. This approach increases the effectiveness of the mutations by focusing on areas more likely to improve the individual's fitness.

4.5.5 5. Simulated Annealing

Simulated Annealing (SA) is a probabilistic technique used for approximating the global optimum of a given function. In the context of genetic algorithms, SA is utilized to accept or reject new individuals (offspring) based on a temperature-controlled probability. This helps in balancing the exploration and exploitation of the search space.
Simulated Annealing helps the genetic algorithm escape local optima by occasionally accepting worse solutions with a certain probability. This probability decreases as the algorithm progresses, controlled by a parameter called "temperature."

4.5.5.1 Key Components of Simulated Annealing in Genetic Algorithms

1. **Acceptance Probability**: Determines whether to accept a new solution based on the difference in fitness and the current temperature.

2. **Temperature Schedule**: Controls how the temperature decreases over time.

3. **Integration with Genetic Algorithm**: Used in conjunction with crossover and mutation to guide the search process.

4.5.5.2 Code Explanation

Acceptance Probability

The acceptance probability is calculated using the formula: $$ P = exp(\frac{\Delta E}{T}) $$ where:

- is the change in fitness (new fitness - current fitness).
- is the current temperature.

```
// Genetic_Algorithm.cpp

// Calculate acceptance probability of Simulated Annealing functions
double acceptanceProbability(double currentEnergy, double newEnergy, double temperature) {
    if (newEnergy > currentEnergy) {
        return 1.0;
    }
    return std::exp((newEnergy - currentEnergy) / temperature);
}
```

If the new solution is better (>0), the probability is greater than 1, and the new solution is accepted. If the new solution is worse (<0), it might still be accepted based on the calculated probability, allowing the algorithm to escape local optima.

The temperature is gradually decreased according to a cooling schedule. This reduces the likelihood of accepting worse solutions as the algorithm progresses, focusing more on exploitation of the best solutions found so far. The cooling schedule can be a simple linear decrement or a more complex function, but in our implementation, a straightforward decrement is used: $$ T = T - \Delta T $$ where is a small decrement value.

Simulated Annealing Process

During each iteration of the genetic algorithm, after generating offspring through crossover and mutation, the `applySimulatedAnnealing` function is applied to decide whether to accept the offspring. This mechanism ensures that even suboptimal solutions can occasionally be accepted, promoting diversity and preventing premature convergence to suboptimal solutions.

```
// Genetic_Algorithm.cpp

bool applySimulatedAnnealing(Individual &offspring, Individual &parent1, Individual &parent2, int
      vector_size,
                             double (&func)(int, int *, struct CircuitParameters), bool (&validity)(int, int
      *),
                             double &Temp, const Algorithm_Parameters &params, const CircuitParameters
      c_params,
                             std::mt19937 &generator) {
    // Calculate the fitness (energy) of the offspring
    if (validity(vector_size, offspring.genes.data())) {
        offspring.fitness = func(vector_size, offspring.genes.data(), c_params);
    } else {
        offspring.fitness = INVALID_FITNESS; // Give a negative fitness for invalid individuals
        return false; // Reject invalid offspring
    }
    double parents_fitness = (parent1.fitness + parent2.fitness) / 2;

    // Simulated annealing acceptance criterion
    std::uniform_real_distribution<double> distribution(0.0, 1.0);

    // Check the threshold
    if (Temp > 1) {
        if (acceptanceProbability(parents_fitness, offspring.fitness, Temp) > distribution(generator)) {
            return true; // Accept the offspring
        }
        // Cool down the temperature
        Temp -= params.deltT;
    } else {
        // Keep the original parents
        if (parent1.fitness > parent2.fitness) {
            offspring = parent1;
        } else {
            offspring = parent2;
        }
        return true;
    }
    return false; // Reject the offspring
}
```

Explanation of the Code

- **Fitness Calculation**: The fitness (energy) of the offspring is calculated using the provided fitness function `func`. If the offspring is valid, its fitness is evaluated; otherwise, it is assigned a negative fitness value, and the offspring is rejected.

- **Parents' Fitness**: The average fitness of the two parents is computed. This serves as the baseline for comparing the fitness of the offspring.
- **Acceptance Criterion**: A uniform random number generator determines whether the offspring is accepted based on the calculated acceptance probability.
- **Temperature Check and Cooling**: The temperature is checked to see if it is above a certain threshold. If so, the acceptance probability is compared with a random number to decide if the offspring should be accepted. The temperature is then decreased according to the cooling schedule. If the temperature is too low, the offspring is not accepted unless it is better than the worse parent.
- **Updating Offspring**: If the offspring is accepted, it becomes part of the population. If not, the better of the two parents is retained.

Integration with the Genetic Algorithm

In the main loop of the genetic algorithm, simulated annealing is applied after the crossover and mutation steps. This ensures that the offspring are accepted based on the temperature-controlled probability.

```
// Genetic_Algorithm.cpp

// Main function to optimize the solution using a genetic algorithm
int optimize(int vector_size, int *vector, double (&func)(int, int *, struct CircuitParameters),
             bool (&validity)(int, int *), Algorithm_Parameters params, CircuitParameters c_params) {
                    // ...

                // Parameters for Simulated Annealing
                double Temp = params.initialTemp;

                // Create offspring1 with Simulated Annealing Acceptance
                do {
                   // ...
                if (localNewPopulation.size() < params.populationSize) {
                    do {
                        crossover->setParents(offspring2, selectedParents[parent1_index],
      selectedParents[parent2_index]);
                        crossover->apply(generators[omp_get_thread_num()]);

                        mutation->setIndividual(offspring2);
                        mutation->apply(generators[omp_get_thread_num()]);
                    } while (!applySimulatedAnnealing(offspring2, selectedParents[parent1_index],
      selectedParents[parent2_index], vector_size, func, validity, Temp, params, c_params,
      generators[omp_get_thread_num()]));

                    localNewPopulation.push_back(offspring2);
                }
            }
        // ...
}
```

In summary, simulated annealing is an essential component of the genetic algorithm in this project, allowing the algorithm to balance between exploration and exploitation by accepting new individuals based on a temperature-controlled probability. This helps the algorithm to escape local optima and find better solutions over time.

4.5.6 6. Progress Tracking with tqdm

In our genetic algorithm implementation, we use the self-implemented `tqdm.hpp` to provide real-time progress tracking and visualization of the optimization process. This helps developers and users monitor the progress of the algorithm, gain insights into the computational workload, and estimate the time remaining for completion.

4.5.6.1 Integration of tqdm in Genetic Algorithm

Tqdm is integrated into the main loop of the genetic algorithm to track and display the progress of each generation. This provides a dynamic and user-friendly way to observe the algorithm's performance.

1. ##### Initialization of tqdm

We initialize tqdm to set up the progress bar before entering the main loop of the genetic algorithm. The progress bar is configured to track the number of generations.

1. ##### Progress Update

Within the main loop, tqdm updates the progress bar at each iteration, showing the current generation number and other relevant metrics, such as the maximum fitness of the population.
Here's the code snippet demonstrating the integration of tqdm:
```
#include "tqdm.hpp"  // Include the tqdm library

// Main function to optimize the solution using a genetic algorithm
int optimize(int vector_size, int *vector, double (&func)(int, int *, struct CircuitParameters),
             bool (&validity)(int, int *), Algorithm_Parameters params, CircuitParameters c_params) {
    // ...
        // Initialize tqdm for progress tracking
    auto iter = tqdm::trange(params.maxGenerations);
    iter.set_prefix("Iterating ");  // Set the prefix for the progress bar
    for (int generation: iter) {
            // ...
            iter « "Max Fitness: " « maxFitness;
            // ...
    }
    // ...
 }
```

4.5.6.2 Detailed Explanation

- **Initialization**: The `tqdm` progress bar is initialized using `tqdm::trange` with the maximum number of generations as the parameter. The prefix "Iterating " is set to make it clear what the progress bar represents.
- **Progress Update**: Inside the main loop of the genetic algorithm, the progress bar is updated at each generation. The current maximum fitness value is also displayed, providing a real-time metric for the algorithm's performance.
- **Benefits**: This integration provides the following benefits:
  - **Real-time Monitoring**: Users can monitor the progress and performance of the algorithm in real-time.
  - **User-Friendly Visualization**: The progress bar offers a user-friendly way to track the computational process.
  - **Performance Insights**: Displaying metrics like maximum fitness helps in understanding the algorithm's behavior and performance trends.

The use of `tqdm` significantly enhances the usability and transparency of the genetic algorithm, making it easier to track and understand the optimization process.

4.5.7 7. Optimization with OpenMP

In addition to tqdm, the genetic algorithm implementation leverages OpenMP for parallel processing. OpenMP is used to parallelize the evaluation of the population's fitness and other computationally intensive tasks, significantly improving the algorithm's performance on multi-core systems.

4.5.7.1 Integration of OpenMP in Genetic Algorithm

OpenMP is integrated into the main loop of the genetic algorithm to parallelize the evaluation of fitness and other operations. This provides a substantial performance boost by utilizing multiple CPU cores.
Here's the code snippet demonstrating the integration of OpenMP:
```
#include <omp.h>  // Include the OpenMP library

// Main function to optimize the solution using a genetic algorithm
int optimize(int vector_size, int *vector, double (&func)(int, int *, struct CircuitParameters),
             bool (&validity)(int, int *), Algorithm_Parameters params, CircuitParameters c_params) {
    // ...

    // Initialize population in parallel
    #pragma omp parallel for
    for (int i = 0; i < params.populationSize; i++) {
        // ... Population initialization code ...
    }

    // Main loop for generations
    auto iter = tqdm::trange(params.maxGenerations);
    iter.set_prefix("Iterating ");  // Set the prefix for the progress bar
    for (int generation: iter) {
        // Evaluate fitness in parallel
        #pragma omp parallel for
        for (int i = 0; i < population.size(); i++) {
            population[i].fitness = func(vector_size, population[i].genes.data(), c_params);
```

```
        }

        // Selection, crossover, and mutation operations
        // ...
        #pragma omp parallel for
        for (int i = 0; i < params.populationSize; i++) {
            // ... Genetic operations code ...
        }

        // Update progress bar
        iter « "Max Fitness: " « maxFitness;
    }
    // ...
}
```

4.5.7.2 Detailed Explanation

- **Parallel Initialization**: The population initialization is parallelized using `#pragma omp parallel for`, allowing multiple individuals to be initialized simultaneously.
- **Parallel Fitness Evaluation**: The evaluation of the population's fitness is parallelized, significantly speeding up the process, especially for large populations.
- **Parallel Genetic Operations**: Selection, crossover, and mutation operations are also parallelized to maximize performance.

4.5.7.3 Benefits of Using OpenMP

- **Improved Performance**: Leveraging multiple CPU cores significantly reduces the time required for fitness evaluation and genetic operations.
- **Scalability**: The algorithm can handle larger populations and more generations within a reasonable time frame.
- **Efficiency**: Parallel processing makes the genetic algorithm more efficient, enabling faster convergence to optimal solutions.

The integration of OpenMP with tqdm provides a powerful combination of performance and usability, making the genetic algorithm both fast and user-friendly.

Chapter 5

Genetic Algorithm Parameters Analysis

This document presents a detailed analysis of the various parameters used in the Genetic Algorithm for optimizing circuit configurations. The goal is to determine the optimal settings for each parameter to ensure efficient and effective performance of the algorithm.

5.1 Introduction

Genetic Algorithms (GAs) are powerful optimization tools inspired by the principles of natural selection and genetics. The performance of GAs is highly dependent on the proper tuning of their parameters. This analysis focuses on two main categories of parameters:

- Algorithm Parameters
- Circuit Parameters

5.2 Algorithm Parameters Analysis

5.2.1 Population Size

Population size determines the number of individuals in each generation. A larger population size increases the genetic diversity but also requires more computational resources.

5.2.1.1 Analysis Results

- Population sizes of 600, 800, and 1000 start with relatively high initial fitness values, indicating that they quickly find high-quality solutions.
- The fitness values increase rapidly in the early generations.
- A population size of 400 shows the fastest increase in fitness, reaching 167.378 by generation 20.
- All population sizes eventually stabilize at a fitness value of 167.378, demonstrating effective convergence of the algorithm.

5.2.1.2 Parameter Choice

A population size of 400 is optimal as it balances computational cost and convergence speed effectively.

5.2.2 Tournament Size

Tournament size determines the number of individuals randomly selected from the population for each tournament. The winner of each tournament, being the individual with the highest fitness, is selected to be a parent for the next generation.

5.2.2.1 Analysis Results

- All tournament sizes show a rapid increase in fitness within the first 10 generations.
- Tournament Size 5 reaches the highest fitness value (around 165) by generation 20 and remains stable.
- Sizes 6 and 7 also achieve high fitness values but are slightly lower than Tournament Size 5.

5.2.2.2 Parameter Choice

Choosing a tournament size of 5 ensures that the genetic algorithm efficiently explores the solution space and quickly converges to high-quality solutions, making it suitable for most optimization problems.

5.2.3 Crossover Rate

The crossover rate is a probability value (usually between 0 and 1) that dictates the likelihood of applying the crossover operation to generate new offspring from two parent individuals. It helps control the genetic diversity in the population by mixing genes from different parents.

5.2.3.1 Analysis Results

- Crossover rates of 0.7 and 0.8 demonstrate the fastest convergence, achieving the maximum fitness value the quickest.
- All crossover rates eventually reach the same maximum fitness value, indicating robustness across different rates.

5.2.3.2 Parameter Choice

For future runs, consider using a crossover rate of 0.8 to achieve the fastest improvement and robust performance.

5.2.4 Mutation Rate

Mutation rate is a crucial parameter in genetic algorithms, representing the probability of randomly altering genes in an individual's genome. It helps maintain genetic diversity within the population, preventing premature convergence to local optima and enabling the exploration of the solution space.

5.2.4.1 Analysis Results

- Lower mutation rates (0.01, 0.05, and 0.1) demonstrate quicker convergence and higher final fitness values compared to higher mutation rates (0.2 and 0.3).
- Mutation rate 0.01 shows the fastest and most stable improvement.

5.2.4.2 Parameter Choice

Use a mutation rate of 0.01 to achieve rapid convergence and robust performance.

5.2.5 Elite Percentage

The elitePercentage parameter in a genetic algorithm controls the elitism strategy. This strategy directly preserves the best-performing individuals from the current generation into the next generation without any mutation or crossover. This ensures that the top-quality genes are passed on, enhancing the algorithm's efficiency and stability.

5.2.5.1 Analysis Results

- elitePercentage values of 0.01 show the highest fitness values consistently, reaching a fitness of 836.835 by generation 250.
- elitePercentage values of 0.05, 0.1, and 0.2 show slower fitness increases, with all stabilizing around 829.175 by generation 250.

5.2.5.2 Parameter Choice

0.01 ensures rapid convergence to high fitness values while maintaining the robustness of the genetic algorithm.

5.3 Circuit Parameters Analysis

5.3.1 Number of Units

The number of units in the circuit configuration can significantly affect performance metrics like elapsed time, performance, recovery, and grade.

5.3.1.1 Analysis Results

Num Units	Elapsed Time (s)	Performance	Recovery	Grade	Final Circuit Configuration
4	1.35789	110.25	0.150332	0.965671	2 1 1 2 4 0 0 0 3 3 0 2 5
6	2.45652	232.583	0.280938	0.977566	1 6 5 5 5 2 3 0 5 1 5 2 4 5 2 7 0 0 2
8	4.5209	341.632	0.37187	0.989275	4 7 7 1 7 0 3 0 3 6 0 1 4 0 3 5 0 3 2 0 3 9 8 0 0
10	6.82153	437.72	0.471582	0.990517	0 8 1 6 2 7 0 10 8 8 8 1 4 8 1 9 8 0 11 8 1 3 2 8 1 2 2 7 8 1 5

5.3.1.2 Visualization

5.3.1.3 Trends

- **Elapsed Time**: Increases almost linearly with the number of units, reflecting increased complexity and processing time.
- **Performance**: Improves significantly with an increasing number of units, indicating better processing capabilities and efficiency.
- **Recovery**: Increases with the number of units, suggesting improved system capacity to recover from errors or faults.
- **Grade**: Slight improvement with the number of units, indicating overall better performance and efficiency in larger configurations.

5.3.2 Purity of Input Feed (Gerardium:Waste Feed Ratio)

The purity of the input feed, represented as the ratio of gerardium feed to waste feed, can significantly affect performance metrics like elapsed time, performance, recovery, and grade.

5.3.2.1 Analysis Results

Gerardium:Waste Feed	Elapsed Time (s)	Performance	Recovery	Grade	Final Circuit Configuration
10:90	6.64633	437.72	0.471582	0.990517	5 7 1 6 2 3 5 10 7 7 2 7 1 7 1 0 7 1 8 7 1 9 2 2 3 7 1 4 7 5 11
20:80	6.44538	862.07	0.494251	0.983232	8 3 6 9 3 6 4 5 3 6 10 5 2 3 6 11 10 3 3 5 2 8 3 2 0 5 2 7 3 6 1
30:70	6.97441	1320.43	0.498122	0.984718	7 2 2 8 2 8 9 10 10 3 10 2 0 2 8 1 3 8 11 2 0 4 2 0 6 2 3 7 2 8 5

Gerardium:Waste Feed	Elapsed Time (s)	Performance	Recovery	Grade	Final Circuit Configuration
40:60	6.51499	1838.62	0.522236	0.984274	5 10 10 7 0 5 6 0 5 11 10 0 5 0 3 9 0 3 8 0 5 2 10 0 3 0 3 4 0 3 1
50:50	6.95478	2369.51	0.534505	0.985107	7 10 8 4 8 4 6 8 4 1 10 8 0 10 3 7 8 0 9 8 4 11 8 0 5 10 10 3 8 0 2
60:40	6.10315	3029.69	0.652294	0.970762	6 10 8 3 9 8 4 10 8 1 10 8 2 9 8 5 9 6 7 10 9 0 9 6 11 10 9 6 10 10 8
70:30	5.96057	4044.07	0.758418	0.969211	5 10 5 3 10 5 0 10 5 1 10 5 8 10 5 2 10 4 4 5 5 7 5 2 11 10 5 9 10 5 6
80:20	5.83807	5417.92	0.816378	0.97778	2 10 2 1 10 2 6 10 3 3 10 2 7 2 5 11 10 2 8 10 2 4 10 2 5 10 2 9 10 2 0
90:10	5.46021	7134.92	0.902775	0.984013	7 10 7 5 10 2 9 10 7 3 10 7 0 10 7 8 10 7 4 10 10 2 10 10 6 10 7 1 10 5 11

5.3.2.2 Visualization

5.3.2.3 Trends

- **Performance** (red line, circles) shows a significant increase as the proportion of gerardium feed increases.
- **Recovery** (blue line, dashed) improves with higher gerardium feed, indicating better extraction efficiency.
- **Grade** (green line, dotted) remains relatively stable with slight fluctuations.
- **Elapsed Time** (orange line, squares) shows a decreasing trend, suggesting improved computational efficiency with higher purity feeds.

5.3.3 Price Ratio of Gerardium to Waste Disposal

The economic parameters, specifically the reward for gerardium and the penalties for waste disposal, significantly affect the performance metrics of the circuit. This analysis examines how changes in these economic parameters impact the optimum circuit configuration, performance, recovery, and grade.

5.3.3.1 Analysis Results

Default reward and penalty is 100 and -750.

Reward/$\hookleftarrow$ Penalty Coefficient	Num Units	Elapsed Time (s)	Performance	Recovery	Grade	Final Circuit Configuration
Reward of £100 $*$ 5 per kg	5	1.99066	167.379	0.202835	0.977224	3 1 4 6 2 2 4 5 1 1 1 4 0 1 1 3
Reward of £100 $*$ 5 per kg	10	6.86852	437.72	0.471582	0.990517	1 6 9 8 6 9 2 6 9 4 10 6 6 6 9 0 3 6 9 3 3 5 6 1 11 6 9 7 3 5 1

Reward/← Penalty Coefficient	Num Units	Elapsed Time (s)	Performance	Recovery	Grade	Final Circuit Configuration
Reward of £100 $*$ 10 per kg	5	2.08692	167.379	0.202835	0.977224	2 3 4 6 5 3 3 3 4 0 1 1 4 3 3 2
Reward of £100 $*$ 10 per kg	10	6.76027	437.72	0.471582	0.990517	0 7 5 3 7 0 11 7 5 1 7 5 9 7 5 2 6 8 0 10 7 7 6 6 8 6 7 5 7 5 4
Penalty of £750 $*$ 5 per kg	5	1.94702	167.379	0.202835	0.977224	1 5 4 4 4 3 2 4 3 6 4 4 1 0 0 3
Penalty of £750 $*$ 5 per kg	10	6.7229	437.72	0.471582	0.990517	7 6 9 1 6 9 4 5 6 9 6 9 8 6 9 3 10 6 6 5 5 2 6 9 0 6 7 11 5 2 7
Penalty of £750 $*$ 10 per kg	5	1.87025	162.125	0.196688	0.977106	0 3 4 4 5 3 3 3 0 6 1 1 0 3 0 2
Penalty of £750 $*$ 10 per kg	10	7.42508	437.72	0.471582	0.990517	2 9 2 11 9 8 7 9 8 1 6 9 8 9 8 0 9 8 4 10 9 9 9 8 5 6 3 2 6 6 3

5.3.3.2 Visualization

The visualizations above show the impact of different economic parameters on the circuit performance metrics for configurations with 5 and 10 units:

- **Performance**: Remains constant at 167.379 for lower penalty coefficients but drops to 162.125 when the penalty increases significantly. For 10 units, the performance remains stable at 437.72 across all economic parameters.
- **Elapsed Time**: Shows slight variations but generally increases with higher penalties. For 5 units, elapsed time decreases slightly with increased reward but rises with higher penalties.
- **Recovery**: Remains relatively constant for both 5 and 10 units, indicating that recovery is less sensitive to changes in economic coefficients.
- **Grade**: Remains stable for both 5 and 10 units, showing minimal fluctuation.

5.3.4 Recommendations

- **Optimize for Higher Units**: Configurations with more units (10 units) are less sensitive to economic parameter changes, maintaining stable performance and grade.
- **Penalty Management**: When penalties for waste disposal are high, consider optimizing circuit configurations to minimize waste, as higher penalties can negatively impact performance.
- **Reward Adjustments**: Increasing rewards has minimal impact on performance and recovery, suggesting that optimizing for penalties might be more critical in achieving better economic outcomes.

Chapter 6

References

6.1 Research Papers

1. **Lambora, A., Gupta, K., & Chopra, K. (2019).** "Genetic algorithm-A literature review." *2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon)*. IEEE. doi:10.↩ 1109/COMITCon.2019.8862254. Link

2. **Yuan, Y., & Banzhaf, W. (2018).** "Arja: Automated repair of Java programs via multi-objective genetic programming." *IEEE Transactions on Software Engineering*, 46(10), 1040-1067. doi:10.1109/TSE.2018.↩ 2876535. Link

3. **Bertsimas, D., & Tsitsiklis, J. (1993).** "Simulated annealing." *Statistical Science*, 8(1), 10-15. doi:10.↩ 1214/ss/1177011077. Link

6.2 Official Documentation

1. **tqdm Documentation**. (2024). *tqdm: A Fast, Extensible Progress Bar for Python and CLI*. Retrieved from https://tqdm.github.io/

2. **Doxygen Documentation**. (2024). *Doxygen: The Standard for Code Documentation*. Retrieved from https://www.doxygen.nl/manual/

3. **CMake Documentation**. (2024). *CMake: Build, Test, and Package Software*. Retrieved from https↩ ://cmake.org/documentation/

6.3 AI Tools

1. **OpenAI ChatGPT**. (2024). *Conversation with GPT-4 on Genetic Algorithm Implementation*. Retrieved from https://chatgpt.com/share/02d1db48-b452-425d-9303-9183f7f1841a

2. **OpenAI ChatGPT**. (2024). *Conversation with GPT-4o on Elite Reservation Strategy*. Retrieved from https://chatgpt.com/share/ac8b9b73-cd17-4677-9677-daa1d8151d66

6.4 Books

1. **Henderson, D., Jacobson, S. H., & Johnson, A. W. (2003).** "The theory and practice of simulated annealing." *Handbook of Metaheuristics*, 287-319. doi:10.1007/978-1-4757-3482-7_10. Link

6.5 Code Repositories

1. **GitHub Repository**. (2016). *tqdm.cpp*. Retrieved from https://github.com/tqdm/tqdm.cpp

2. **GitHub Repository**. (2019). *arja*. Retrieved from https://github.com/yyxhdy/arja

6.6 Miscellaneous

1. **Author's Notes**. (2024). *Project notes and internal documentation*.

Chapter 7

Namespace Index

7.1 Namespace List

Here is a list of all namespaces with brief descriptions:

Chapter 8

Hierarchical Index

8.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 9

Data Structure Index

9.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 10

File Index

10.1 File List

Here is a list of all files with brief descriptions:

Chapter 11

Namespace Documentation

11.1 tqdm Namespace Reference

Data Structures

- class Chronometer

    A simple chronometer to measure elapsed time.
- class int_iterator

    A random access iterator for integers.
- class iter_wrapper

    A wrapper class for iterators to update the progress bar.
- class progress_bar

    A class representing a progress bar.
- class range

    A range of integers represented by iterators.
- struct timer

    A timer that can be used to create a timing iterator.
- class timing_iterator

    A forward iterator that returns the elapsed time.
- class timing_iterator_end_sentinel

    An end sentinel for timing iterators.
- class tqdm_for_lvalues

    A class for progress bars that iterate over lvalues.
- class tqdm_for_rvalues

    A class for progress bars that iterate over rvalues.
- class tqdm_timer

    A progress bar for a timer.

Typedefs

- using index = std::ptrdiff_t
- using time_point_t = std::chrono::time_point<std::chrono::steady_clock>

Functions

- double elapsed_seconds (time_point_t from, time_point_t to)

    Calculate the elapsed seconds between two time points.
- void clamp (double &x, double a, double b)

    Clamp a value between two bounds.
- int get_terminal_width ()

    Get the width of the terminal window.

- template<class Container >
  tqdm_for_lvalues (Container &) -> tqdm_for_lvalues< typename Container::iterator >
- template<class Container >
  tqdm_for_lvalues (const Container &) -> tqdm_for_lvalues< typename Container::const_iterator >
- template<class Container >
  tqdm_for_rvalues (Container &&) -> tqdm_for_rvalues< Container >
- template<class ForwardIter >
  auto tqdm (const ForwardIter &first, const ForwardIter &last)

  Create a tqdm progress bar for a range of iterators.
- template<class ForwardIter >
  auto tqdm (const ForwardIter &first, const ForwardIter &last, index total)

  Create a tqdm progress bar for a range of iterators with a specified total size.
- template<class Container >
  auto tqdm (const Container &C)

  Create a tqdm progress bar for a container.
- template<class Container >
  auto tqdm (Container &C)

  Create a tqdm progress bar for a container.
- template<class Container >
  auto tqdm (Container &&C)

  Create a tqdm progress bar for a container (rvalue reference).
- template<class IntType >
  auto trange (IntType first, IntType last)

  Create a tqdm progress bar for a range of integers.
- template<class IntType >
  auto trange (IntType last)

  Create a tqdm progress bar for a range of integers.
- auto tqdm (timer t)

  Create a tqdm progress bar for a timer.

11.1.1 Typedef Documentation

11.1.1.1 index

```
using tqdm::index = std::ptrdiff_t
```

11.1.1.2 time_point_t

```
using tqdm::time_point_t = std::chrono::time_point<std::chrono::steady_clock>
```

11.1.2 Function Documentation

11.1.2.1 clamp()

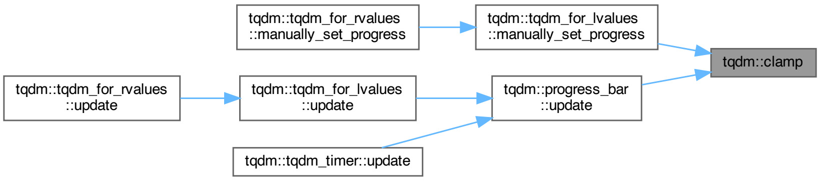
```
void tqdm::clamp (
            double & x,
            double a,
            double b )  [inline]
```

Clamp a value between two bounds.

Parameters

x	The value to clamp.
a	The lower bound.
b	The upper bound.

Here is the caller graph for this function:



11.1.2.2 elapsed_seconds()

```
double tqdm::elapsed_seconds (
            time_point_t from,
            time_point_t to ) [inline]
```

Calculate the elapsed seconds between two time points.
This function calculates the elapsed time in seconds between two time points.

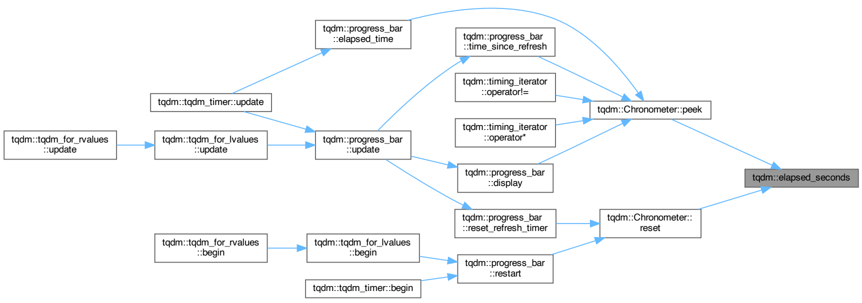
Parameters

from	The starting time point.
to	The ending time point.

Returns

The elapsed time in seconds.

Here is the caller graph for this function:



11.1.2.3 get_terminal_width()

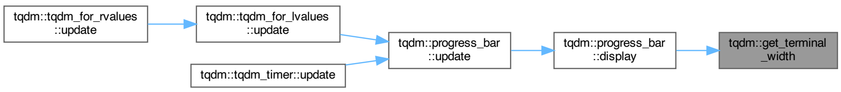
```
int tqdm::get_terminal_width ( ) [inline]
```

Get the width of the terminal window.

Returns

The width of the terminal window in columns.

Here is the caller graph for this function:



11.1.2.4 tqdm() [1/6]

```
template<class Container >
auto tqdm::tqdm (
            const Container & C )
```

Create a tqdm progress bar for a container.

Template Parameters

Container	The type of the container.

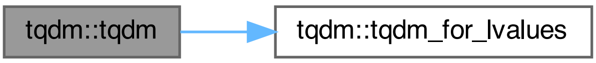
Parameters

C	The container.

Returns

A tqdm progress bar.

Here is the call graph for this function:



11.1.2.5 tqdm() [2/6]

```
template<class ForwardIter >
auto tqdm::tqdm (
            const ForwardIter & first,
            const ForwardIter & last )
```

Create a tqdm progress bar for a range of iterators.

Template Parameters

ForwardIter	The type of the forward iterator.

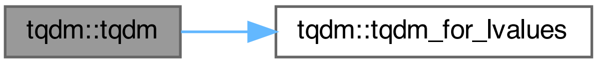
Parameters

first	The beginning of the range.
last	The end of the range.

Returns

A tqdm progress bar.

Here is the call graph for this function:



11.1.2.6 tqdm() [3/6]

```
template<class ForwardIter >
auto tqdm::tqdm (
            const ForwardIter & first,
            const ForwardIter & last,
            index total )
```

Create a tqdm progress bar for a range of iterators with a specified total size.

Template Parameters

ForwardIter	The type of the forward iterator.

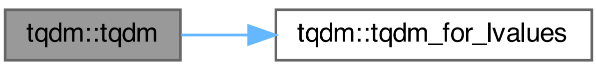
Parameters

first	The beginning of the range.
last	The end of the range.
total	The total size of the range.

Returns

A tqdm progress bar.

Here is the call graph for this function:



11.1.2.7 tqdm() [4/6]

```
template<class Container >
auto tqdm::tqdm (
              Container && C )
```

Create a tqdm progress bar for a container (rvalue reference).

Template Parameters

Container	The type of the container.

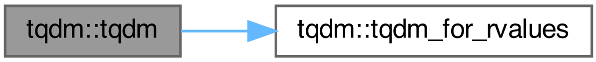
Parameters

C	The container.

Returns

A tqdm progress bar.

Here is the call graph for this function:



11.1.2.8 tqdm() [5/6]

```
template<class Container >
auto tqdm::tqdm (
              Container & C )
```

Create a tqdm progress bar for a container.

Template Parameters

Container	The type of the container.

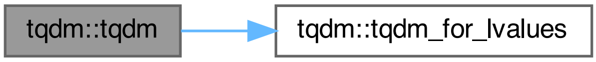
Parameters

C	The container.

Returns

A tqdm progress bar.

Here is the call graph for this function:



11.1.2.9 tqdm() [6/6]

```
auto tqdm::tqdm (
            timer t )  [inline]
```

Create a tqdm progress bar for a timer.

Parameters

t	The timer.

Returns

A tqdm progress bar.

Here is the call graph for this function:

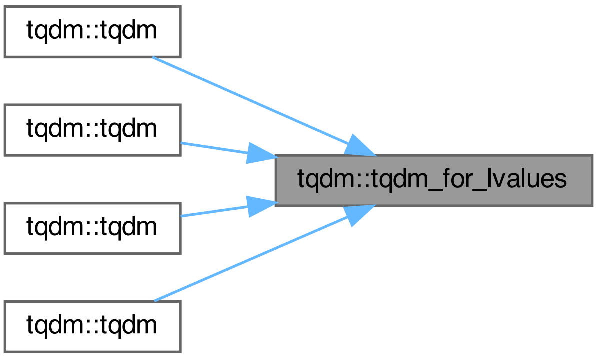
11.1.2.10 tqdm_for_lvalues() [1/2]

```
template<class Container >
tqdm::tqdm_for_lvalues (
            const Container &  ) -> tqdm_for_lvalues< typename Container::const_iterator >
```

11.1.2.11 tqdm_for_lvalues() [2/2]

```
template<class Container >
tqdm::tqdm_for_lvalues (
            Container &  ) -> tqdm_for_lvalues< typename Container::iterator >
```

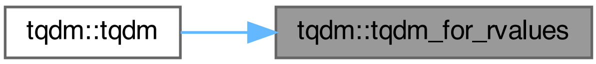
Here is the caller graph for this function:



11.1.2.12 tqdm_for_rvalues()

```
template<class Container >
tqdm::tqdm_for_rvalues (
            Container &&  ) -> tqdm_for_rvalues< Container >
```

Here is the caller graph for this function:



11.1.2.13 trange() [1/2]

```
template<class IntType >
auto tqdm::trange (
            IntType first,
            IntType last )
```

Create a tqdm progress bar for a range of integers.

Template Parameters

IntType	The type of the integer.

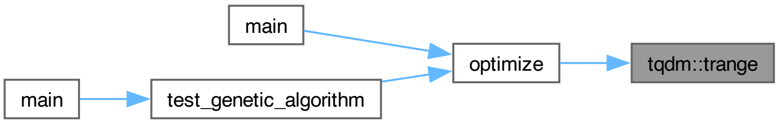
Parameters

first	The first integer.
last	The last integer.

Returns

A tqdm progress bar.

Here is the caller graph for this function:



11.1.2.14 trange() [2/2]

```
template<class IntType >
auto tqdm::trange (
            IntType last )
```

Create a tqdm progress bar for a range of integers.

Template Parameters

IntType	The type of the integer.

Parameters

last	The last integer.

Returns

A tqdm progress bar.

11.2 visualize Namespace Reference

Functions

- parse_list (string)

Variables

- parser
- type
- args = parser.parse_args()
- x = args.vector
- p = args.perf
- r = args.recovery
- g = args.grade
- graph = graphviz.Digraph()
- rankdir
- nodesep
- ranksep
- splines

- overlap
- dpi
- str feed_node = 'Feed'
- str concentrate = 'Concentrate'
- str tailings = 'Tailings'
- shape
- color = 1 else 'red'
- tuple num_nodes = (len(x) - 1) // 3
- str from_node = f'Unit {i // 3}'
- tuple to_node = (num_nodes + 1) else f'Unit {x[i + 1]}'
- cleanup
- True
- format
- view
- dict legend_labels
- list legend_handles
- list columns = ['Feed'] + [f'Unit {i}' for i in range(0, num_nodes)]
- list reshaped_list
- df = pd.DataFrame(reshaped_list).transpose()
- figsize
- handles
- labels
- loc
- bbox_to_anchor
- fontsize
- cellText
- values
- colLabels
- cellLoc
- bbox
- ha
- va
- transform
- bbox_inches

11.2.1 Function Documentation

11.2.1.1 parse_list()

```
visualize.parse_list (
            string )
```

11.2.2 Variable Documentation

11.2.2.1 args

```
visualize.args = parser.parse_args()
```

11.2.2.2 bbox

```
visualize.bbox
```

11.2.2.3 bbox_inches

```
visualize.bbox_inches
```

11.2.2.4 bbox_to_anchor

```
visualize.bbox_to_anchor
```

11.2.2.5 cellLoc

```
visualize.cellLoc
```

11.2.2.6 cellText

```
visualize.cellText
```

11.2.2.7 cleanup

```
visualize.cleanup
```

11.2.2.8 colLabels

```
visualize.colLabels
```

11.2.2.9 color

```
str visualize.color = 1 else 'red'
```

11.2.2.10 columns

```
visualize.columns = ['Feed'] + [f'Unit {i}' for i in range(0, num_nodes)]
```

11.2.2.11 concentrate

```
visualize.concentrate = 'Concentrate'
```

11.2.2.12 df

```
visualize.df = pd.DataFrame(reshaped_list).transpose()
```

11.2.2.13 dpi

```
visualize.dpi
```

11.2.2.14 feed_node

```
visualize.feed_node = 'Feed'
```

11.2.2.15 figsize

```
visualize.figsize
```

11.2.2.16 fontsize

```
visualize.fontsize
```

11.2.2.17 format

```
visualize.format
```

11.2.2.18 from_node

```
visualize.from_node = f'Unit {i // 3}'
```

11.2.2.19 g

```
visualize.g = args.grade
```

11.2.2.20 graph

```
visualize.graph = graphviz.Digraph()
```

11.2.2.21 ha

```
visualize.ha
```

11.2.2.22 handles

```
visualize.handles
```

11.2.2.23 labels

```
visualize.labels
```

11.2.2.24 legend_handles

```
visualize.legend_handles
```

Initial value:

```
=  [plt.Line2D([0], [0], marker='o', color='w', markersize=5,
                          markerfacecolor=color, label=label)
                 for label, color in legend_labels.items()]
```

11.2.2.25 legend_labels

```
dict visualize.legend_labels
```

Initial value:

```
=  {'Concentrate': 'blue', 'Intermediate': 'purple',
                 'Tailings': 'red'}
```

11.2.2.26 loc

```
visualize.loc
```

11.2.2.27 nodesep

```
visualize.nodesep
```

11.2.2.28 num_nodes

```
tuple visualize.num_nodes = (len(x) - 1) // 3
```

11.2.2.29 overlap

```
visualize.overlap
```

11.2.2.30 p

```
visualize.p = args.perf
```

11.2.2.31 parser

```
visualize.parser
```

Initial value:

```
=  argparse.ArgumentParser(description='Command line arguments for \
                              visualizing the circuit diagram.')
```

11.2.2.32 r

```
visualize.r = args.recovery
```

11.2.2.33 rankdir

```
visualize.rankdir
```

11.2.2.34 ranksep

```
visualize.ranksep
```

11.2.2.35 reshaped_list

```
list visualize.reshaped_list
```

Initial value:

```
00001 =  [[x[0]]] + [[' '.join(map(str, x[i:i + 3]))]
00002                            for i in range(1, len(x), 3)]
```

11.2.2.36 shape

```
visualize.shape
```

11.2.2.37 splines

```
visualize.splines
```

11.2.2.38 tailings

```
visualize.tailings = 'Tailings'
```

11.2.2.39 to_node

```
visualize.to_node = (num_nodes + 1) else f'Unit {x[i + 1]}'
```

11.2.2.40 transform

```
visualize.transform
```

11.2.2.41 True

```
visualize.True
```

11.2.2.42 type

```
visualize.type
```

11.2.2.43 va

```
visualize.va
```

11.2.2.44 values

```
visualize.values
```

11.2.2.45 view

```
visualize.view
```

11.2.2.46 x

```
visualize.x = args.vector
```

Chapter 12

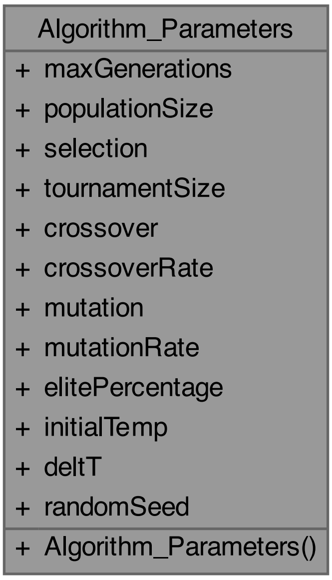
Data Structure Documentation

12.1 Algorithm_Parameters Struct Reference

Configures the parameters of a genetic algorithm.

`#include <Parameter.h>`

Collaboration diagram for Algorithm_Parameters:



Public Member Functions

- Algorithm_Parameters (int gens, int popSize, std::string selection, int tourSize, std::string crossover, double crossRate, std::string mutation, double mutRate, double elitePerc, double initialTemp, double deltT, double randomSeed)

  Constructor to initialize the parameters with custom values.

Data Fields

- int maxGenerations = 1000
- int populationSize = 100

- std::string selection = "NaryTournamentSelection"
- int tournamentSize = 3
- std::string crossover = "TwoPointCrossover"
- double crossoverRate = 0.8
- std::string mutation = "GuidedMutation"
- double mutationRate = 0.1
- double elitePercentage = 0.2
- double initialTemp = 1000.0
- double deltT = 1.0
- unsigned int randomSeed = 0

12.1.1 Detailed Description

Configures the parameters of a genetic algorithm.
The Algorithm_Parameters structure contains various parameters used to configure the behavior of a genetic algorithm, including the number of generations, population size, tournament size, crossover rate, mutation rate, elite percentage, and random seed.

12.1.2 Constructor & Destructor Documentation

12.1.2.1 Algorithm_Parameters()

```
Algorithm_Parameters::Algorithm_Parameters (
            int gens,
            int popSize,
            std::string selection,
            int tourSize,
            std::string crossover,
            double crossRate,
            std::string mutation,
            double mutRate,
            double elitePerc,
            double initialTemp,
            double deltT,
            double randomSeed )  [inline]
```

Constructor to initialize the parameters with custom values.

Parameters

gens	Maximum number of generations.
popSize	Size of the population.
tourSize	Number of individuals to select in each tournament.
crossRate	Rate at which crossover occurs.
mutRate	Rate at which mutation occurs.
elitePerc	Percentage of elite individuals to keep in the next generation.
seed	Random seed, 0 means using current time.

12.1.3 Field Documentation

12.1.3.1 crossover

```
std::string Algorithm_Parameters::crossover = "TwoPointCrossover"
```

Crossover method

12.1.3.2 crossoverRate

`double Algorithm_Parameters::crossoverRate = 0.8`

Rate at which crossover occurs

12.1.3.3 deltT

`double Algorithm_Parameters::deltT = 1.0`

The temperature decrement of Simulated Annealing algorithm

12.1.3.4 elitePercentage

`double Algorithm_Parameters::elitePercentage = 0.2`

Percentage of elite individuals to keep in the next generation

12.1.3.5 initialTemp

`double Algorithm_Parameters::initialTemp = 1000.0`

Initial temperature of Simulated Annealing algorithm

12.1.3.6 maxGenerations

`int Algorithm_Parameters::maxGenerations = 1000`

Maximum number of generations

12.1.3.7 mutation

`std::string Algorithm_Parameters::mutation = "GuidedMutation"`

Mutation method

12.1.3.8 mutationRate

`double Algorithm_Parameters::mutationRate = 0.1`

Rate at which mutation occurs

12.1.3.9 populationSize

`int Algorithm_Parameters::populationSize = 100`

Size of the population

12.1.3.10 randomSeed

`unsigned int Algorithm_Parameters::randomSeed = 0`

Random seed, 0 means using current time

12.1.3.11 selection

`std::string Algorithm_Parameters::selection = "NaryTournamentSelection"`

Selection method

12.1.3.12 tournamentSize

`int Algorithm_Parameters::tournamentSize = 3`

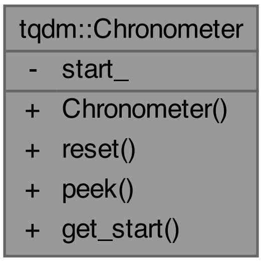
Number of individuals to select in each tournament

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

- include/utils/Parameter.h

12.2 tqdm::Chronometer Class Reference

A simple chronometer to measure elapsed time.
`#include <tqdm.hpp>`
Collaboration diagram for tqdm::Chronometer:



Public Member Functions

- Chronometer ()

  Construct a new Chronometer object.
- double reset ()

  Reset the chronometer and return the elapsed time since the last reset.
- double peek () const

  Get the elapsed time without resetting the chronometer.
- time_point_t get_start () const

  Get the starting time point.

Private Attributes

- time_point_t start_

12.2.1 Detailed Description

A simple chronometer to measure elapsed time.

12.2.2 Constructor & Destructor Documentation

12.2.2.1 Chronometer()

`tqdm::Chronometer::Chronometer ( ) [inline]`
Construct a new Chronometer object.

12.2.3 Member Function Documentation

12.2.3.1 get_start()

`time_point_t tqdm::Chronometer::get_start ( ) const [inline]`
Get the starting time point.

Returns

The starting time point.

12.2.3.2 peek()

`double tqdm::Chronometer::peek ( ) const [inline]`

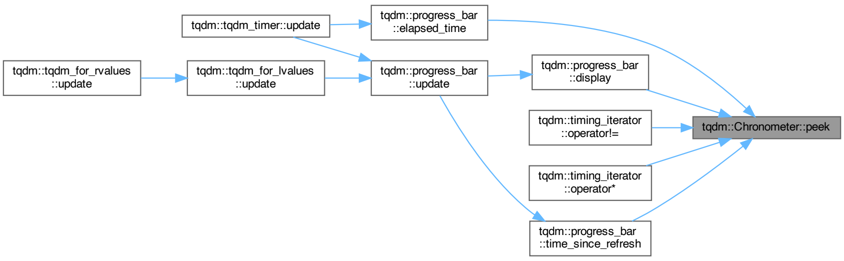
Get the elapsed time without resetting the chronometer.

Returns

The elapsed time in seconds.

Here is the call graph for this function:

Here is the caller graph for this function:



12.2.3.3 reset()

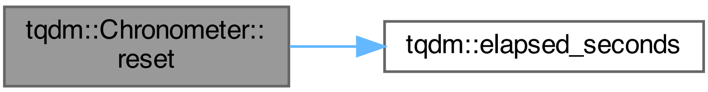
`double tqdm::Chronometer::reset ( ) [inline]`

Reset the chronometer and return the elapsed time since the last reset.

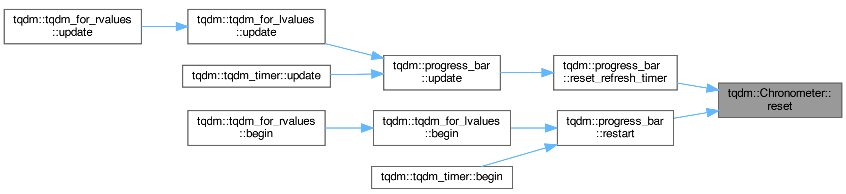
Returns

The elapsed time in seconds.

Here is the call graph for this function:



Here is the caller graph for this function:



12.2.4 Field Documentation

12.2.4.1 start_

`time_point_t tqdm::Chronometer::start_ [private]`

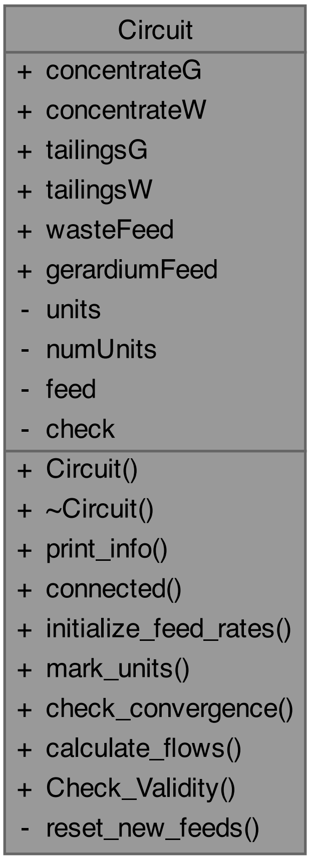
The documentation for this class was generated from the following file:

- include/utils/tqdm.hpp

12.3 Circuit Class Reference

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

Collaboration diagram for Circuit:



Public Member Functions

- Circuit (int num_units)

    Constructs a Circuit object with a specified number of units.
- ∼Circuit ()

    Destructs the Circuit object.
- void print_info () const

    Prints information about the circuit.
- void connected (int ∗circuitVector)

    Connects the units in the circuit based on the circuit vector.
- void initialize_feed_rates (double gerardium_=10, double waste_=90)

    Initializes the feed rates for gerardium and waste.
- void mark_units (int unitNum, std::vector< CUnit ∗ > &units, bool check[3])

    Marks a unit in the circuit and recursively marks its connected units.
- bool check_convergence (double threshold)

    Checks if the circuit has converged based on a threshold value.
- void calculate_flows ()

    Calculates the flows within the circuit.

Static Public Member Functions

- static bool Check_Validity (int vectorSize, int ∗circuitVector)

  Checks the validity of the circuit vector.

Data Fields

- double concentrateG
- double concentrateW
- double tailingsG
- double tailingsW
- double wasteFeed
- double gerardiumFeed

Private Member Functions

- void reset_new_feeds ()

  Resets the new feeds in the circuit.

Private Attributes

- std::vector< CUnit ∗ > units
- int numUnits
- int feed

Static Private Attributes

- static bool check [3]

12.3.1 Constructor & Destructor Documentation

12.3.1.1 Circuit()

```
Circuit::Circuit (
                int num_units )
```

Constructs a Circuit object with a specified number of units.

Parameters

num_units	The number of units in the circuit.

12.3.1.2 ∼Circuit()

```
Circuit::∼Circuit ( )
```

Destructs the Circuit object.

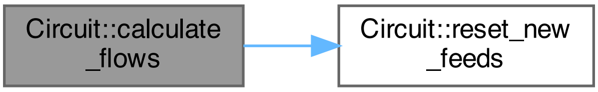
12.3.2 Member Function Documentation

12.3.2.1 calculate_flows()

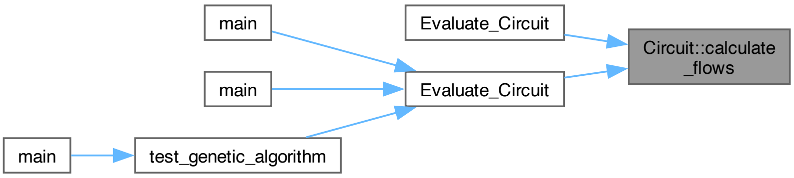
```
void Circuit::calculate_flows ( )
```

Calculates the flows within the circuit.

This member function calculates the flow rates of gerardium and waste through the circuit's units. Here is the call graph for this function:



Here is the caller graph for this function:



12.3.2.2 check_convergence()

```
bool Circuit::check_convergence (
            double threshold = 1e-6 )
```

Checks if the circuit has converged based on a threshold value.

This member function checks whether the circuit's calculations have converged by comparing the changes in feed rates to a specified threshold.

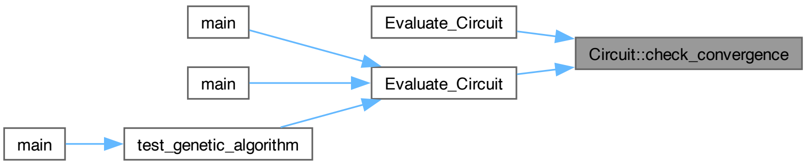
Parameters

threshold	The convergence threshold.

Returns

True if the circuit has converged, otherwise false.

Here is the caller graph for this function:



12.3.2.3 Check_Validity()

```
bool Circuit::Check_Validity (
            int vectorSize,
            int * circuitVector ) [static]
```

Checks the validity of the circuit vector.

This static member function checks whether a given circuit vector is valid. It verifies the connections and configurations of the units in the circuit.

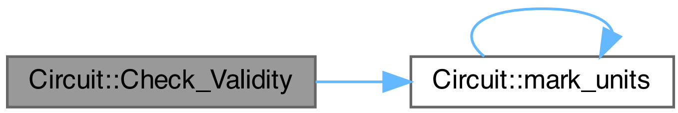
Parameters

vectorSize	The size of the circuit vector.
circuitVector	A pointer to the circuit vector.

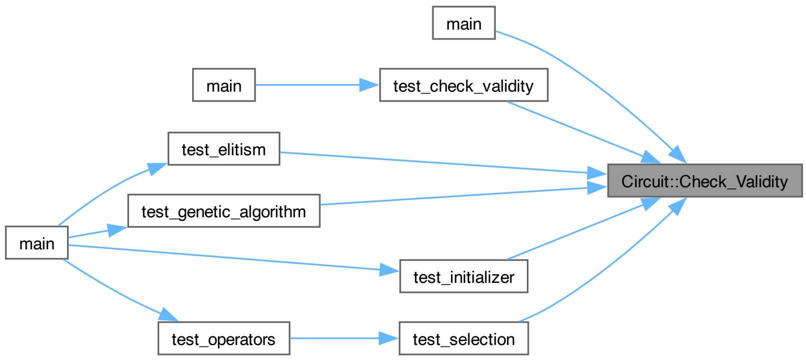
Returns

True if the circuit vector is valid, otherwise false.

Here is the call graph for this function:



Here is the caller graph for this function:



12.3.2.4 connected()

```
void Circuit::connected (
            int * circuitVector )
```

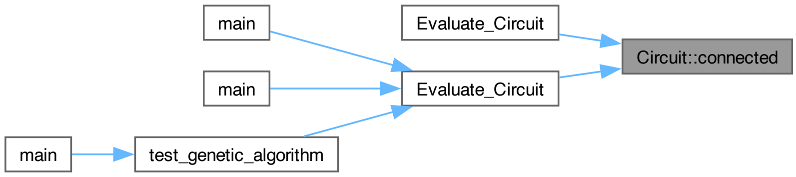
Connects the units in the circuit based on the circuit vector.

This member function establishes connections between the units in the circuit according to the specified circuit vector.

Parameters

circuitVector	A pointer to the circuit vector.

Here is the caller graph for this function:



12.3.2.5 initialize_feed_rates()

```
void Circuit::initialize_feed_rates (
            double gerardium_ = 10,
            double waste_ = 90 )
```

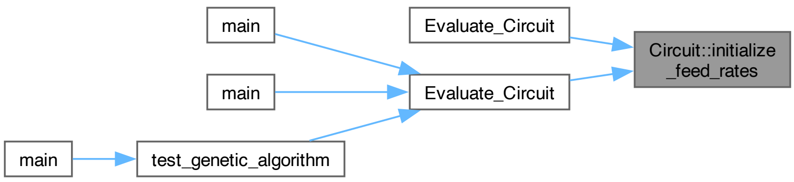
Initializes the feed rates for gerardium and waste.

This member function initializes the feed rates for gerardium and waste to the specified values.

Parameters

gerardium↵_	The feed rate of gerardium (default is 10).
*waste_*	The feed rate of waste (default is 90).

Here is the caller graph for this function:



12.3.2.6 mark_units()

```
void Circuit::mark_units (
            int unitNum,
            std::vector< CUnit * > & units,
            bool check[3] )
```

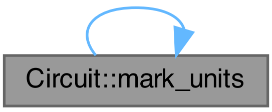
Marks a unit in the circuit and recursively marks its connected units.

This function marks a specified unit in the circuit as visited. It then recursively marks all units connected to this unit through its concNum, interNum, and tailsNum connections. If a connection points to an outlet, it updates the corresponding check flag to indicate an exit has been seen.

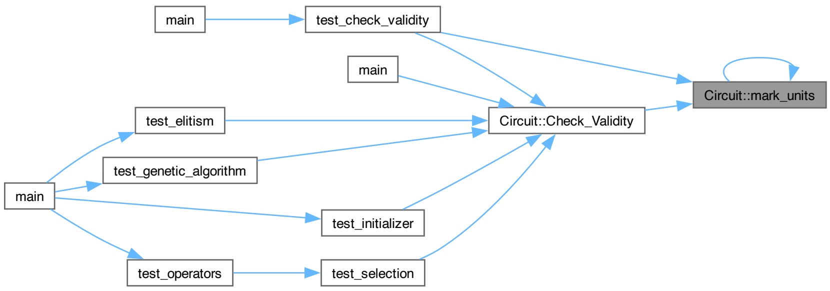
Parameters

unit_num	The index of the unit to be marked.
units	A vector of pointers to CUnit objects representing the units in the circuit.
check	An array of three booleans that track if the concNum, interNum, or tailsNum connections lead to a circuit outlet.

Here is the call graph for this function:



Here is the caller graph for this function:



12.3.2.7 print_info()

`void Circuit::print_info ( ) const`

Prints information about the circuit.

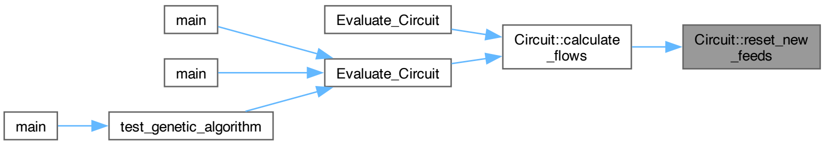
This member function prints detailed information about the circuit, including the number of units, feed rates, and concentrations.

12.3.2.8 reset_new_feeds()

`void Circuit::reset_new_feeds ( ) [private]`

Resets the new feeds in the circuit.

This helper function resets the new feed rates in the circuit to their initial values. Here is the caller graph for this function:



12.3.3 Field Documentation

12.3.3.1 check

`bool Circuit::check[3] [static], [private]`

Static array to check the units

12.3.3.2 concentrateG

`double Circuit::concentrateG`

Concentration of gerardium in the circuit

12.3.3.3 concentrateW

`double Circuit::concentrateW`

Concentration of waste in the circuit

12.3.3.4 feed

`int Circuit::feed [private]`

The feed rate for the circuit

12.3.3.5 gerardiumFeed

`double Circuit::gerardiumFeed`

Feed rate of gerardium in the circuit

12.3.3.6 numUnits

`int Circuit::numUnits [private]`

The number of units in the circuit

12.3.3.7 tailingsG

`double Circuit::tailingsG`

Tailings of gerardium in the circuit

12.3.3.8 tailingsW

`double Circuit::tailingsW`

Tailings of waste in the circuit

12.3.3.9 units

`std::vector<CUnit *> Circuit::units [private]`

Vector of pointers to the units in the circuit

12.3.3.10 wasteFeed

`double Circuit::wasteFeed`

Feed rate of waste in the circuit

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

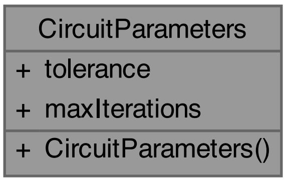
- include/circuit/CCircuit.h
- investigate/CCircuit.cpp
- src/circuit/CCircuit.cpp

12.4 CircuitParameters Struct Reference

Defines the parameters for the circuit simulator.

`#include <Parameter.h>`

Collaboration diagram for CircuitParameters:



Public Member Functions

- CircuitParameters (double tol, int maxIter)

    Constructor to initialize the parameters with custom values.

Data Fields

- double tolerance = 1e-6
- int maxIterations = 1000

12.4.1 Detailed Description

Defines the parameters for the circuit simulator.

The CircuitParameters structure contains various parameters used for simulating the circuit, such as tolerance and maximum iterations.

12.4.2 Constructor & Destructor Documentation

12.4.2.1 CircuitParameters()

```
CircuitParameters::CircuitParameters (
            double tol,
            int maxIter )  [inline]
```

Constructor to initialize the parameters with custom values.

Parameters

tol	The tolerance level for the simulation.
maxIter	The maximum number of iterations for the simulation.

12.4.3 Field Documentation

12.4.3.1 maxIterations

`int CircuitParameters::maxIterations = 1000`

The maximum number of iterations for the simulation.

12.4.3.2 tolerance

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

The tolerance level for the simulation.

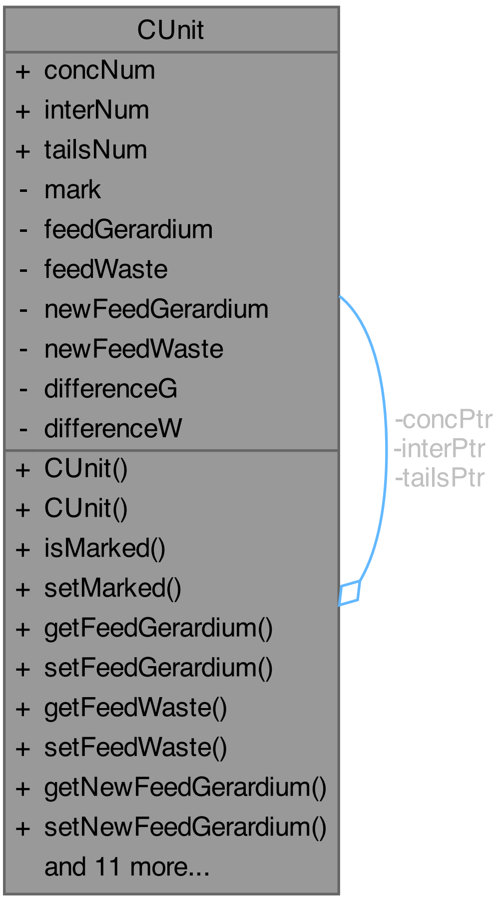
The documentation for this struct was generated from the following file:

- include/utils/Parameter.h

12.5 CUnit Class Reference

`#include <CUnit.h>`

Collaboration diagram for CUnit:



Public Member Functions

- CUnit ()

Default constructor for CUnit.

- CUnit (int concNum, int interNum, int tailsNum)

  Parameterized constructor for CUnit.
- bool isMarked () const

  Checks if the unit is marked.
- void setMarked (bool mark)

  Sets the mark status of the unit.
- double getFeedGerardium () const

  Gets the feed rate of gerardium.
- void setFeedGerardium (double feedGerardium)

  Sets the feed rate of gerardium.
- double getFeedWaste () const

  Gets the feed rate of waste.
- void setFeedWaste (double feedWaste)

  Sets the feed rate of waste.
- double getNewFeedGerardium () const

  Gets the new feed rate of gerardium.
- void setNewFeedGerardium (double newFeedGerardium)

  Sets the new feed rate of gerardium.
- double getNewFeedWaste () const

  Gets the new feed rate of waste.
- void setNewFeedWaste (double newFeedWaste)

  Sets the new feed rate of waste.
- double getDifferenceG () const

  Gets the difference in feed rate of gerardium.
- double getDifferenceW () const

  Gets the difference in feed rate of waste.
- void setDifference ()

  Sets the differences in feed rates for gerardium and waste.
- CUnit ∗ getConcPtr () const

  Gets the pointer to the concentrate connection.
- void setConcPtr (CUnit ∗ptr)

  Sets the pointer to the concentrate connection.
- CUnit ∗ getInterPtr () const

  Gets the pointer to the intermediate connection.
- void setInterPtr (CUnit ∗ptr)

  Sets the pointer to the intermediate connection.
- CUnit ∗ getTailsPtr () const

  Gets the pointer to the tailings connection.
- void setTailsPtr (CUnit ∗ptr)

  Sets the pointer to the tailings connection.

Data Fields

- int concNum
- int interNum
- int tailsNum

Private Attributes

- bool mark
- double feedGerardium
- double feedWaste
- double newFeedGerardium
- double newFeedWaste
- double differenceG
- double differenceW
- CUnit * concPtr
- CUnit * interPtr
- CUnit * tailsPtr

12.5.1 Constructor & Destructor Documentation

12.5.1.1 CUnit() [1/2]

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

Default constructor for CUnit.
Constructs a CUnit object with default values for concentrate, intermediate, and tailings numbers.

12.5.1.2 CUnit() [2/2]

```
CUnit::CUnit (
            int concNum,
            int interNum,
            int tailsNum )
```

Parameterized constructor for CUnit.
Constructs a CUnit object with specified values for concentrate, intermediate, and tailings numbers.

Parameters

concNum	The concentrate number.
interNum	The intermediate number.
tailsNum	The tailings number.

12.5.2 Member Function Documentation

12.5.2.1 getConcPtr()

```
CUnit * CUnit::getConcPtr ( ) const
```

Gets the pointer to the concentrate connection.

Returns

The pointer to the concentrate connection.

12.5.2.2 getDifferenceG()

```
double CUnit::getDifferenceG ( ) const
```

Gets the difference in feed rate of gerardium.

Returns

The difference in feed rate of gerardium.

12.5.2.3 getDifferenceW()

`double CUnit::getDifferenceW ( ) const`

Gets the difference in feed rate of waste.

Returns

The difference in feed rate of waste.

12.5.2.4 getFeedGerardium()

`double CUnit::getFeedGerardium ( ) const`

Gets the feed rate of gerardium.

Returns

The feed rate of gerardium.

12.5.2.5 getFeedWaste()

`double CUnit::getFeedWaste ( ) const`

Gets the feed rate of waste.

Returns

The feed rate of waste.

12.5.2.6 getInterPtr()

`CUnit * CUnit::getInterPtr ( ) const`

Gets the pointer to the intermediate connection.

Returns

The pointer to the intermediate connection.

12.5.2.7 getNewFeedGerardium()

`double CUnit::getNewFeedGerardium ( ) const`

Gets the new feed rate of gerardium.

Returns

The new feed rate of gerardium.

12.5.2.8 getNewFeedWaste()

`double CUnit::getNewFeedWaste ( ) const`

Gets the new feed rate of waste.

Returns

The new feed rate of waste.

12.5.2.9 getTailsPtr()

`CUnit * CUnit::getTailsPtr ( ) const`

Gets the pointer to the tailings connection.

Returns

The pointer to the tailings connection.

12.5.2.10 isMarked()

```
bool CUnit::isMarked ( ) const
```

Checks if the unit is marked.

Returns

True if the unit is marked, otherwise false.

12.5.2.11 setConcPtr()

```
void CUnit::setConcPtr (
             CUnit * ptr )
```

Sets the pointer to the concentrate connection.

Parameters

ptr	The pointer to the concentrate connection to set.

12.5.2.12 setDifference()

```
void CUnit::setDifference ( )
```

Sets the differences in feed rates for gerardium and waste.

12.5.2.13 setFeedGerardium()

```
void CUnit::setFeedGerardium (
             double feedGerardium )
```

Sets the feed rate of gerardium.

Parameters

feedGerardium	The feed rate of gerardium to set.

12.5.2.14 setFeedWaste()

```
void CUnit::setFeedWaste (
             double feedWaste )
```

Sets the feed rate of waste.

Parameters

feedWaste	The feed rate of waste to set.

12.5.2.15 setInterPtr()

```
void CUnit::setInterPtr (
             CUnit * ptr )
```

Sets the pointer to the intermediate connection.

Parameters

ptr	The pointer to the intermediate connection to set.

12.5.2.16 setMarked()

```
void CUnit::setMarked (
            bool mark )
```

Sets the mark status of the unit.

Parameters

mark	The mark status to set.

12.5.2.17 setNewFeedGerardium()

```
void CUnit::setNewFeedGerardium (
            double newFeedGerardium )
```

Sets the new feed rate of gerardium.

Parameters

newFeedGerardium	The new feed rate of gerardium to set.

12.5.2.18 setNewFeedWaste()

```
void CUnit::setNewFeedWaste (
            double newFeedWaste )
```

Sets the new feed rate of waste.

Parameters

newFeedWaste	The new feed rate of waste to set.

12.5.2.19 setTailsPtr()

```
void CUnit::setTailsPtr (
            CUnit * ptr )
```

Sets the pointer to the tailings connection.

Parameters

ptr	The pointer to the tailings connection to set.

12.5.3 Field Documentation

12.5.3.1 concNum

```
int CUnit::concNum
```

The concentrate number

12.5.3.2 concPtr

```
CUnit* CUnit::concPtr [private]
```

Pointer to the concentrate connection

12.5.3.3 differenceG

```
double CUnit::differenceG [private]
```

Difference in feed rate of gerardium

12.5.3.4 differenceW

`double CUnit::differenceW [private]`

Difference in feed rate of waste

12.5.3.5 feedGerardium

`double CUnit::feedGerardium [private]`

Feed rate of gerardium

12.5.3.6 feedWaste

`double CUnit::feedWaste [private]`

Feed rate of waste

12.5.3.7 interNum

`int CUnit::interNum`

The intermediate number

12.5.3.8 interPtr

`CUnit* CUnit::interPtr [private]`

Pointer to the intermediate connection

12.5.3.9 mark

`bool CUnit::mark [private]`

Mark status of the unit

12.5.3.10 newFeedGerardium

`double CUnit::newFeedGerardium [private]`

New feed rate of gerardium

12.5.3.11 newFeedWaste

`double CUnit::newFeedWaste [private]`

New feed rate of waste

12.5.3.12 tailsNum

`int CUnit::tailsNum`

The tailings number

12.5.3.13 tailsPtr

`CUnit* CUnit::tailsPtr [private]`

Pointer to the tailings connection

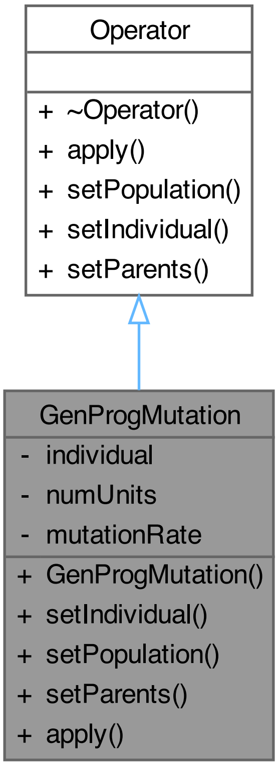
The documentation for this class was generated from the following files:

- include/circuit/CUnit.h
- src/circuit/CUnit.cpp

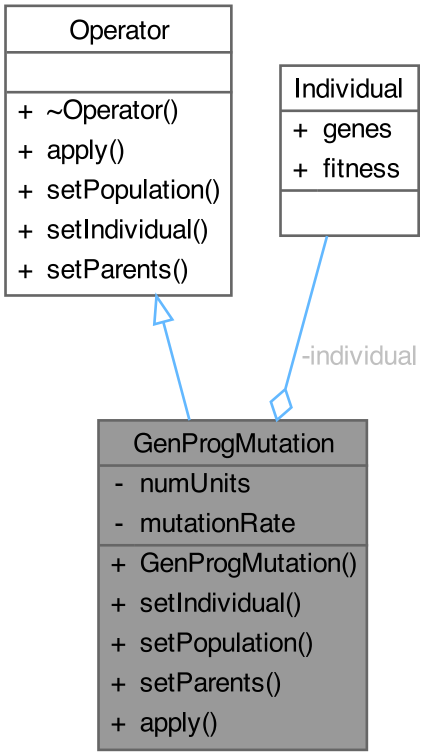
12.6 GenProgMutation Class Reference

`#include <GenProgMutation.h>`

Inheritance diagram for GenProgMutation:



Collaboration diagram for GenProgMutation:



Public Member Functions

- GenProgMutation (int vectorSize, const Algorithm_Parameters &params)

    Constructs a GenProgMutation object with specified vector size and algorithm parameters.
- void setIndividual (Individual &individual) override

    Sets the individual to be mutated.
- void setPopulation (const std::vector< Individual > &population, std::vector< Individual > &selected)

    Sets the population and selected vectors for the mutation operation.
- void setParents (Individual &offspring, const Individual &parent1, const Individual &parent2)

    Sets the parents and offspring for the mutation operation.
- void apply (std::mt19937 &generator)

    Performs the mutation operation on the individual.

Public Member Functions inherited from Operator

- virtual ~Operator ()=default

    Virtual destructor for the Operator class.

Private Attributes

- Individual * individual
- int numUnits
- double mutationRate

12.6.1 Constructor & Destructor Documentation

12.6.1.1 GenProgMutation()

```
GenProgMutation::GenProgMutation (
            int vectorSize,
            const Algorithm_Parameters & params )
```

Constructs a GenProgMutation object with specified vector size and algorithm parameters.

Parameters

vectorSize	The size of the vector representing the individual's genetic material.
params	The algorithm parameters including the mutation rate.

12.6.2 Member Function Documentation

12.6.2.1 apply()

```
void GenProgMutation::apply (
            std::mt19937 & generator ) [virtual]
```

Performs the mutation operation on the individual.

This method performs mutation on the individual's genetic material based on the specified mutation rate. The mutation modifies the genes of the individual to introduce variations.

Parameters

generator	The random number generator used to determine if mutation should occur.

Implements Operator.

12.6.2.2 setIndividual()

```
void GenProgMutation::setIndividual (
            Individual & individual ) [override], [virtual]
```

Sets the individual to be mutated.

Parameters

individual	The individual to be mutated.

Implements Operator.

12.6.2.3 setParents()

```
void GenProgMutation::setParents (
            Individual & offspring,
            const Individual & parent1,
            const Individual & parent2 ) [inline], [virtual]
```

Sets the parents and offspring for the mutation operation.

This method is not used in the mutation operation.

Parameters

offspring	The offspring individual to be generated.
parent1	The first parent individual.
parent2	The second parent individual.

Implements Operator.

12.6.2.4 setPopulation()

```
void GenProgMutation::setPopulation (
            const std::vector< Individual > & population,
            std::vector< Individual > & selected )  [inline], [virtual]
```

Sets the population and selected vectors for the mutation operation.
This method is not used in the mutation operation.

Parameters

population	The vector of individuals in the population.
selected	The vector of selected individuals.

Implements Operator.

12.6.3 Field Documentation

12.6.3.1 individual

```
Individual* GenProgMutation::individual  [private]
```

Pointer to the individual to be mutated

12.6.3.2 mutationRate

```
double GenProgMutation::mutationRate  [private]
```

The mutation rate used to determine if mutation should occur

12.6.3.3 numUnits

```
int GenProgMutation::numUnits  [private]
```

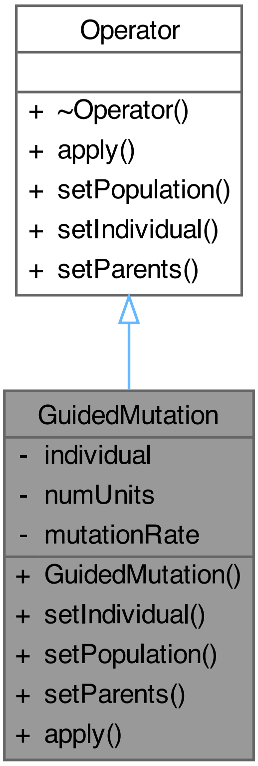
Represents the number of units or types of operations/nodes
The documentation for this class was generated from the following files:

- include/operators/Mutation/GenProgMutation.h
- src/operators/Mutation/GenProgMutation.cpp

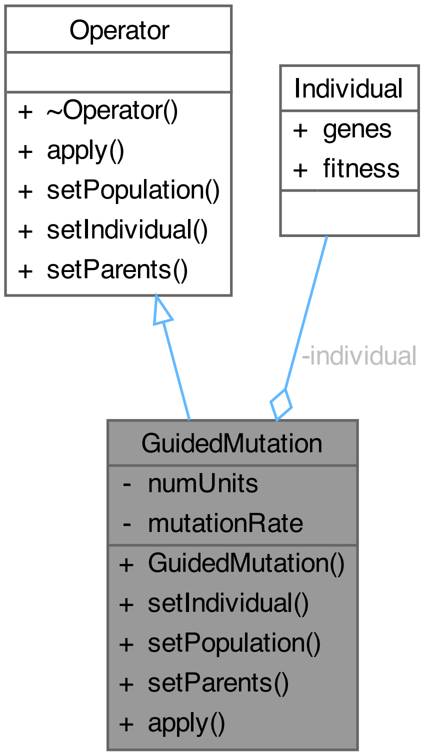
12.7 GuidedMutation Class Reference

```
#include <GuidedMutation.h>
```

Inheritance diagram for GuidedMutation:



Collaboration diagram for GuidedMutation:



Public Member Functions

- GuidedMutation (int vectorSize, const Algorithm_Parameters &params)

  Constructs a GuidedMutation object with specified vector size and algorithm parameters.
- void setIndividual (Individual &individual) override

  Sets the individual to be mutated.
- void setPopulation (const std::vector< Individual > &population, std::vector< Individual > &selected)

  Performs the guided mutation operation on the individual.
- void setParents (Individual &offspring, const Individual &parent1, const Individual &parent2)

  Sets the parents and offspring for the crossover operation.
- void apply (std::mt19937 &generator) override

  Performs the guided mutation operation on the individual.

Public Member Functions inherited from Operator

- virtual ~Operator ()=default

  Virtual destructor for the Operator class.

Private Attributes

- Individual ∗ individual
- int numUnits
- double mutationRate

12.7.1 Constructor & Destructor Documentation

12.7.1.1 GuidedMutation()

```
GuidedMutation::GuidedMutation (
            int vectorSize,
            const Algorithm_Parameters & params )
```

Constructs a GuidedMutation object with specified vector size and algorithm parameters.

Parameters

vectorSize	The size of the vector representing the individual's genetic material.
params	The algorithm parameters including the mutation rate.

12.7.2 Member Function Documentation

12.7.2.1 apply()

```
void GuidedMutation::apply (
            std::mt19937 & generator )  [override], [virtual]
```

Performs the guided mutation operation on the individual.

This method performs mutation on the individual's genetic material based on the specified mutation rate. The mutation modifies the genes of the individual to introduce variations according to specific rules or heuristics.

Parameters

generator	The random number generator used to determine if mutation should occur.

Implements Operator.

12.7.2.2 setIndividual()

```
void GuidedMutation::setIndividual (
            Individual & individual )  [override], [virtual]
```

Sets the individual to be mutated.

Parameters

individual	The individual to be mutated.

Implements Operator.

12.7.2.3 setParents()

```
void GuidedMutation::setParents (
            Individual & offspring,
            const Individual & parent1,
            const Individual & parent2 )  [inline], [virtual]
```

Sets the parents and offspring for the crossover operation.

This method is not used by the GuidedMutation operator.

Parameters

offspring	The offspring individual to be generated.
parent1	The first parent individual.
parent2	The second parent individual.

Implements Operator.

12.7.2.4 setPopulation()

```
void GuidedMutation::setPopulation (
            const std::vector< Individual > & population,
            std::vector< Individual > & selected )  [inline], [virtual]
```

Performs the guided mutation operation on the individual.

This method performs mutation on the individual's genetic material based on the specified mutation rate. The mutation modifies the genes of the individual to introduce variations according to specific rules or heuristics.

Parameters

generator	The random number generator used to determine if mutation should occur.

Implements Operator.

12.7.3 Field Documentation

12.7.3.1 individual

```
Individual* GuidedMutation::individual  [private]
```

Pointer to the individual to be mutated

12.7.3.2 mutationRate

```
double GuidedMutation::mutationRate  [private]
```

The mutation rate used to determine if mutation should occur

12.7.3.3 numUnits

```
int GuidedMutation::numUnits  [private]
```

Represents the number of units or types of operations/nodes

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

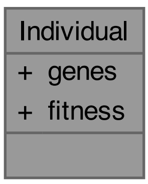
- include/operators/Mutation/GuidedMutation.h
- src/operators/Mutation/GuidedMutation.cpp

12.8 Individual Struct Reference

Represents an individual in a genetic algorithm.

`#include <Individual.h>`

Collaboration diagram for Individual:



Data Fields

- std::vector< int > genes
- double fitness

12.8.1 Detailed Description

Represents an individual in a genetic algorithm.
The Individual structure contains a vector of genes that represent the circuit configuration and a fitness value that indicates the quality or performance of the circuit configuration.

12.8.2 Field Documentation

12.8.2.1 fitness

`double Individual::fitness`

Stores the fitness of the circuit configuration as a double

12.8.2.2 genes

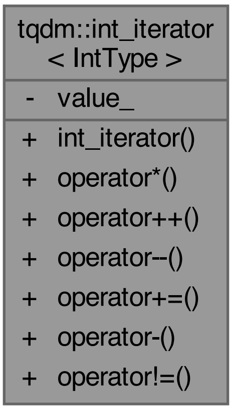
`std::vector<int> Individual::genes`

Stores the circuit configuration as a vector of integers
The documentation for this struct was generated from the following file:

- include/utils/Individual.h

12.9 tqdm::int_iterator< IntType > Class Template Reference

A random access iterator for integers.
`#include <tqdm.hpp>`
Collaboration diagram for tqdm::int_iterator< IntType >:



Public Types

- using iterator_category = std::random_access_iterator_tag
- using value_type = IntType
- using difference_type = IntType
- using pointer = IntType ∗

- using reference = IntType &

Public Member Functions

- int_iterator (IntType val)
- IntType & operator* ()
- int_iterator & operator++ ()
- int_iterator & operator-- ()
- int_iterator & operator+= (difference_type d)
- difference_type operator- (const int_iterator &other) const
- bool operator!= (const int_iterator &other) const

Private Attributes

- IntType value_

12.9.1 Detailed Description

template<class IntType>
class tqdm::int_iterator< IntType >

A random access iterator for integers.

Template Parameters

IntType	The type of the integer.

12.9.2 Member Typedef Documentation

12.9.2.1 difference_type

```
template<class IntType >
using tqdm::int_iterator< IntType >::difference_type = IntType
```

12.9.2.2 iterator_category

```
template<class IntType >
using tqdm::int_iterator< IntType >::iterator_category = std::random_access_iterator_tag
```

12.9.2.3 pointer

```
template<class IntType >
using tqdm::int_iterator< IntType >::pointer = IntType *
```

12.9.2.4 reference

```
template<class IntType >
using tqdm::int_iterator< IntType >::reference = IntType &
```

12.9.2.5 value_type

```
template<class IntType >
using tqdm::int_iterator< IntType >::value_type = IntType
```

12.9.3 Constructor & Destructor Documentation

12.9.3.1 int_iterator()

```
template<class IntType >
tqdm::int_iterator< IntType >::int_iterator (
            IntType val ) [inline], [explicit]
```

12.9.4 Member Function Documentation

12.9.4.1 operator"!=()

```
template<class IntType >
bool tqdm::int_iterator< IntType >::operator!= (
            const int_iterator< IntType > & other ) const [inline]
```

12.9.4.2 operator*()

```
template<class IntType >
IntType & tqdm::int_iterator< IntType >::operator* ( ) [inline]
```

12.9.4.3 operator++()

```
template<class IntType >
int_iterator & tqdm::int_iterator< IntType >::operator++ ( ) [inline]
```

12.9.4.4 operator+=()

```
template<class IntType >
int_iterator & tqdm::int_iterator< IntType >::operator+= (
            difference_type d ) [inline]
```

12.9.4.5 operator-()

```
template<class IntType >
difference_type tqdm::int_iterator< IntType >::operator- (
            const int_iterator< IntType > & other ) const [inline]
```

12.9.4.6 operator--()

```
template<class IntType >
int_iterator & tqdm::int_iterator< IntType >::operator-- ( ) [inline]
```

12.9.5 Field Documentation

12.9.5.1 value_

```
template<class IntType >
IntType tqdm::int_iterator< IntType >::value_ [private]
```

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

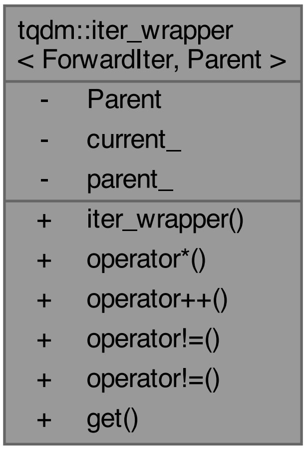
- include/utils/tqdm.hpp

12.10 tqdm::iter_wrapper< ForwardIter, Parent > Class Template Reference

A wrapper class for iterators to update the progress bar.

```
#include <tqdm.hpp>
```

Collaboration diagram for tqdm::iter_wrapper< ForwardIter, Parent >:



Public Types

- using iterator_category = typename ForwardIter::iterator_category
- using value_type = typename ForwardIter::value_type
- using difference_type = typename ForwardIter::difference_type
- using pointer = typename ForwardIter::pointer
- using reference = typename ForwardIter::reference

Public Member Functions

- iter_wrapper (ForwardIter it, Parent ∗parent)
- auto operator∗ ()
- void operator++ ()
- template< class Other >
  bool operator!= (const Other &other) const
- bool operator!= (const iter_wrapper &other) const
- const ForwardIter & get () const

Private Attributes

- friend Parent
- ForwardIter current_
- Parent ∗ parent_

12.10.1 Detailed Description

template< class ForwardIter, class Parent>
class tqdm::iter_wrapper< ForwardIter, Parent >

A wrapper class for iterators to update the progress bar.

Template Parameters

ForwardIter	The type of the underlying iterator.
Parent	The type of the parent progress bar.

12.10.2 Member Typedef Documentation

12.10.2.1 difference_type

```
template<class ForwardIter , class Parent >
using tqdm::iter_wrapper< ForwardIter, Parent >::difference_type = typename ForwardIter↩
::difference_type
```

12.10.2.2 iterator_category

```
template<class ForwardIter , class Parent >
using tqdm::iter_wrapper< ForwardIter, Parent >::iterator_category = typename ForwardIter↩
::iterator_category
```

12.10.2.3 pointer

```
template<class ForwardIter , class Parent >
using tqdm::iter_wrapper< ForwardIter, Parent >::pointer = typename ForwardIter::pointer
```

12.10.2.4 reference

```
template<class ForwardIter , class Parent >
using tqdm::iter_wrapper< ForwardIter, Parent >::reference = typename ForwardIter::reference
```

12.10.2.5 value_type

```
template<class ForwardIter , class Parent >
using tqdm::iter_wrapper< ForwardIter, Parent >::value_type = typename ForwardIter::value_type
```

12.10.3 Constructor & Destructor Documentation

12.10.3.1 iter_wrapper()

```
template<class ForwardIter , class Parent >
tqdm::iter_wrapper< ForwardIter, Parent >::iter_wrapper (
            ForwardIter it,
            Parent * parent )  [inline]
```

12.10.4 Member Function Documentation

12.10.4.1 get()

```
template<class ForwardIter , class Parent >
const ForwardIter & tqdm::iter_wrapper< ForwardIter, Parent >::get ( ) const  [inline]
```

12.10.4.2 operator"!=() [1/2]

```
template<class ForwardIter , class Parent >
bool tqdm::iter_wrapper< ForwardIter, Parent >::operator!= (
            const iter_wrapper< ForwardIter, Parent > & other ) const  [inline]
```

12.10.4.3 operator"!=() [2/2]

```
template<class ForwardIter , class Parent >
template<class Other >
bool tqdm::iter_wrapper< ForwardIter, Parent >::operator!= (
            const Other & other ) const  [inline]
```

12.10.4.4 operator*()

```
template<class ForwardIter , class Parent >
auto tqdm::iter_wrapper< ForwardIter, Parent >::operator* ( )  [inline]
```

12.10.4.5 operator++()

```
template<class ForwardIter , class Parent >
void tqdm::iter_wrapper< ForwardIter, Parent >::operator++ ( )  [inline]
```

12.10.5 Field Documentation

12.10.5.1 current_

```
template<class ForwardIter , class Parent >
ForwardIter tqdm::iter_wrapper< ForwardIter, Parent >::current_  [private]
```

12.10.5.2 Parent

```
template<class ForwardIter , class Parent >
friend tqdm::iter_wrapper< ForwardIter, Parent >::Parent  [private]
```

12.10.5.3 parent_

```
template<class ForwardIter , class Parent >
Parent* tqdm::iter_wrapper< ForwardIter, Parent >::parent_  [private]
```

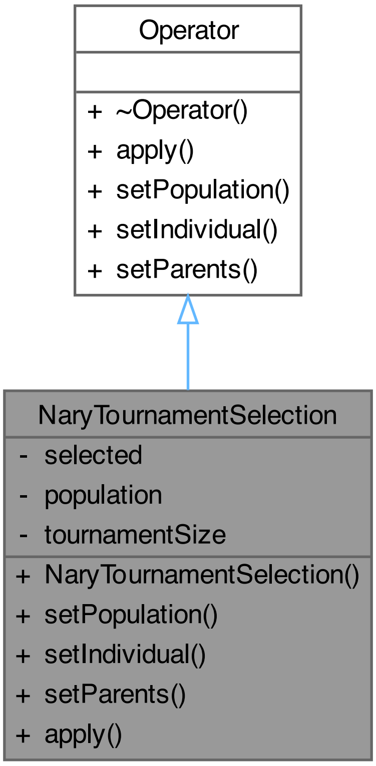
The documentation for this class was generated from the following file:

- include/utils/tqdm.hpp

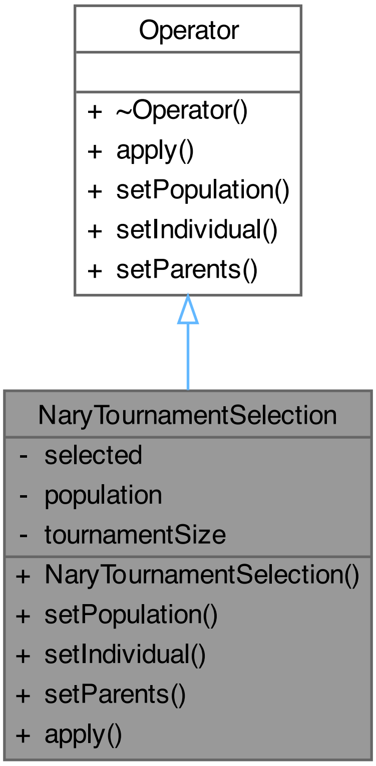
12.11 NaryTournamentSelection Class Reference

```
#include <NaryTournamentSelection.h>
```

Inheritance diagram for NaryTournamentSelection:



Collaboration diagram for NaryTournamentSelection:



Public Member Functions

- NaryTournamentSelection (const Algorithm_Parameters &params)

  Constructs an NaryTournamentSelection object with specified algorithm parameters.
- void setPopulation (const std::vector< Individual > &population, std::vector< Individual > &selected) override

  Sets the population and selected vectors for the selection operation.
- void setIndividual (Individual &individual)

  Sets the individual to be mutated.
- void setParents (Individual &offspring, const Individual &parent1, const Individual &parent2)

  Sets the parents and offspring for the crossover operation.
- void apply (std::mt19937 &generator) override

  Performs the n-ary tournament selection operation.

Public Member Functions inherited from Operator

- virtual ∼Operator ()=default

  Virtual destructor for the Operator class.

Private Attributes

- std::vector< Individual > ∗ selected

- const std::vector< Individual > * population
- int tournamentSize

12.11.1 Constructor & Destructor Documentation

12.11.1.1 NaryTournamentSelection()

```
NaryTournamentSelection::NaryTournamentSelection (
            const Algorithm_Parameters & params )
```

Constructs an NaryTournamentSelection object with specified algorithm parameters.

Parameters

params	The algorithm parameters including the tournament size.

12.11.2 Member Function Documentation

12.11.2.1 apply()

```
void NaryTournamentSelection::apply (
            std::mt19937 & generator ) [override], [virtual]
```

Performs the n-ary tournament selection operation.

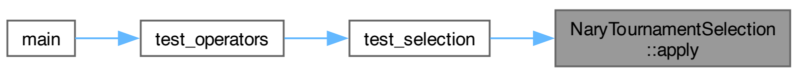
This method selects individuals from the population based on n-ary tournament competition and stores the selected individuals in the provided vector.

Parameters

generator	The random number generator used for selecting individuals.

Implements Operator.

Here is the caller graph for this function:



12.11.2.2 setIndividual()

```
void NaryTournamentSelection::setIndividual (
            Individual & individual ) [inline], [virtual]
```

Sets the individual to be mutated.

This method sets the individual to be mutated by the operator.

Parameters

individual	The individual to be mutated.

Implements Operator.

12.11.2.3 setParents()

```
void NaryTournamentSelection::setParents (
```

```
Individual & offspring,
const Individual & parent1,
const Individual & parent2 )  [inline], [virtual]
```

Sets the parents and offspring for the crossover operation.
This method is not used by the NaryTournamentSelection operator.

Parameters

offspring	The offspring individual to be generated.
parent1	The first parent individual.
parent2	The second parent individual.

Implements Operator.

12.11.2.4 setPopulation()

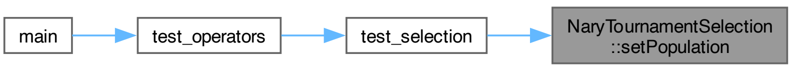
```
void NaryTournamentSelection::setPopulation (
    const std::vector< Individual > & population,
    std::vector< Individual > & selected )  [override], [virtual]
```

Sets the population and selected vectors for the selection operation.

Parameters

population	The vector of individuals in the population.
selected	The vector of selected individuals.

Implements Operator.
Here is the caller graph for this function:



12.11.3 Field Documentation

12.11.3.1 population

```
const std::vector<Individual>* NaryTournamentSelection::population  [private]
```

Pointer to the vector of individuals in the population

12.11.3.2 selected

```
std::vector<Individual>* NaryTournamentSelection::selected  [private]
```

Pointer to the vector of selected individuals

12.11.3.3 tournamentSize

```
int NaryTournamentSelection::tournamentSize  [private]
```

Size of the tournament (number of individuals competing)
The documentation for this class was generated from the following files:

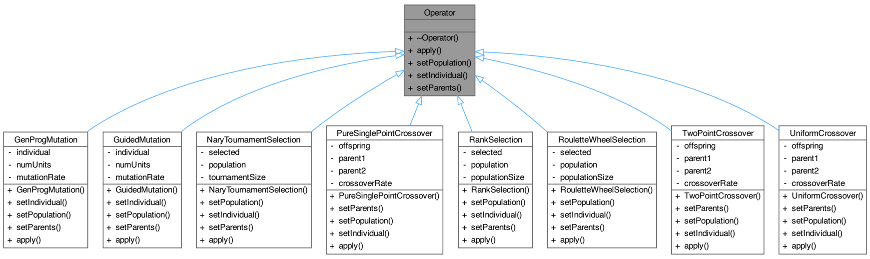
- include/operators/Selection/NaryTournamentSelection.h
- src/operators/Selection/NaryTournamentSelection.cpp

12.12 Operator Class Reference

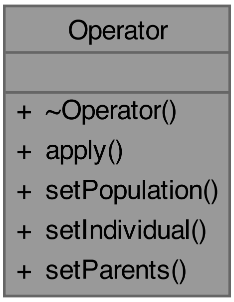
Abstract base class for genetic algorithm operators.

`#include <Operator.h>`

Inheritance diagram for Operator:



Collaboration diagram for Operator:



Public Member Functions

- virtual ~Operator ()=default

  Virtual destructor for the Operator class.
- virtual void apply (std::mt19937 &generator)=0

  Pure virtual function to apply the operator.
- virtual void setPopulation (const std::vector< Individual > &population, std::vector< Individual > &selected)=0

  Sets the population and selected individuals for the operator.
- virtual void setIndividual (Individual &individual)=0

  Sets the individual for the operator.
- virtual void setParents (Individual &offspring, const Individual &parent1, const Individual &parent2)=0

  Sets the parents for the operator.

12.12.1 Detailed Description

Abstract base class for genetic algorithm operators.

The Operator class provides an abstract interface for genetic algorithm operators. It defines a pure virtual function `apply()`, which must be implemented by derived classes. This class serves as a base for various types of genetic algorithm operators, such as selection, crossover, and mutation.

12.12.2 Constructor & Destructor Documentation

12.12.2.1 ~Operator()

```
virtual Operator::~Operator ( ) [virtual], [default]
```

Virtual destructor for the Operator class.
The virtual destructor ensures that derived class destructors are called correctly when an object is deleted through a base class pointer.

12.12.3 Member Function Documentation

12.12.3.1 apply()

```
virtual void Operator::apply (
            std::mt19937 & generator ) [pure virtual]
```

Pure virtual function to apply the operator.
This function must be implemented by derived classes to perform the specific operation of the genetic algorithm operator.
Implemented in GenProgMutation, PureSinglePointCrossover, TwoPointCrossover, UniformCrossover, GuidedMutation, NaryTournamentSelection, RankSelection, and RouletteWheelSelection.

12.12.3.2 setIndividual()

```
virtual void Operator::setIndividual (
            Individual & individual ) [pure virtual]
```

Sets the individual for the operator.
This function sets the individual for the operator. It is used to provide the operator with the necessary data to perform its operation on a single individual.

Parameters

individual	The individual to operate on.

Implemented in PureSinglePointCrossover, TwoPointCrossover, UniformCrossover, NaryTournamentSelection, RankSelection, RouletteWheelSelection, GenProgMutation, and GuidedMutation.

12.12.3.3 setParents()

```
virtual void Operator::setParents (
            Individual & offspring,
            const Individual & parent1,
            const Individual & parent2 ) [pure virtual]
```

Sets the parents for the operator.
This function sets the parents for the operator. It is used to provide the operator with the necessary data to perform its operation on two parent individuals.

Parameters

parent1	The first parent individual.
parent2	The second parent individual.

Implemented in GenProgMutation, GuidedMutation, NaryTournamentSelection, RankSelection, RouletteWheelSelection, PureSinglePointCrossover, TwoPointCrossover, and UniformCrossover.

12.12.3.4 setPopulation()

```
virtual void Operator::setPopulation (
            const std::vector< Individual > & population,
            std::vector< Individual > & selected ) [pure virtual]
```

Sets the population and selected individuals for the operator.
This function sets the population and selected individuals for the operator. It is used to provide the operator with the necessary data to perform its operation.

Parameters

population	The population of individuals to operate on.
selected	The selected individuals to store the results of the operation.

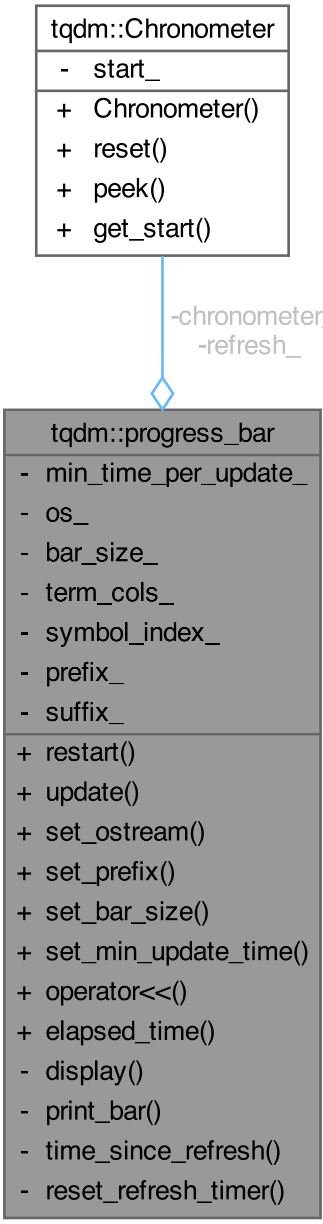
Implemented in PureSinglePointCrossover, TwoPointCrossover, UniformCrossover, GenProgMutation, GuidedMutation, RouletteWheelSelection, NaryTournamentSelection, and RankSelection.
The documentation for this class was generated from the following file:

- include/operators/Operator.h

12.13 tqdm::progress_bar Class Reference

A class representing a progress bar.
`#include <tqdm.hpp>`

Collaboration diagram for tqdm::progress_bar:



Public Member Functions

- void restart ()

  Restart the progress bar.
- void update (double progress)

  Update the progress bar with the current progress.
- void set_ostream (std::ostream &os)

  Set the output stream for the progress bar.
- void set_prefix (std::string s)

Set the prefix string for the progress bar.
- void set_bar_size (int size)

  Set the size of the progress bar.
- void set_min_update_time (double time)

  Set the minimum time between updates.
- template<class T >
  progress_bar & operator<< (const T &t)

  Append a value to the progress bar suffix.
- double elapsed_time () const

  Get the elapsed time since the progress bar started.

Private Member Functions

- void display (double progress)

  Display the progress bar.
- void print_bar (std::stringstream &ss, double filled) const

  Print the progress bar with symbols.
- double time_since_refresh () const

  Get the time since the last refresh.
- void reset_refresh_timer ()

  Reset the refresh timer.

Private Attributes

- Chronometer chronometer_ {}
- Chronometer refresh_ {}
- double min_time_per_update_ {0.15}
- std::ostream ∗ os_ {&std::cerr}
- index bar_size_ {40}
- index term_cols_ {1}
- index symbol_index_ {0}
- std::string prefix_ {}
- std::stringstream suffix_ {}

12.13.1 Detailed Description

A class representing a progress bar.

12.13.2 Member Function Documentation

12.13.2.1 display()

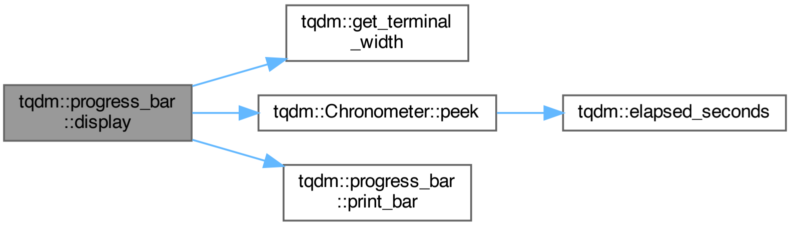
```
void tqdm::progress_bar::display (
            double progress )  [inline], [private]
```

Display the progress bar.

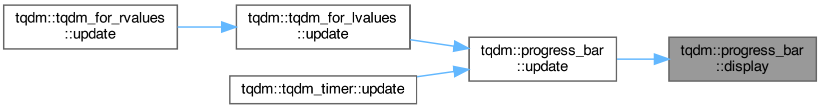
Parameters

progress	The current progress (a value between 0 and 1).

Here is the call graph for this function:



Here is the caller graph for this function:



12.13.2.2 elapsed_time()

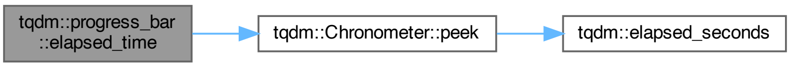
`double tqdm::progress_bar::elapsed_time ( ) const [inline]`

Get the elapsed time since the progress bar started.

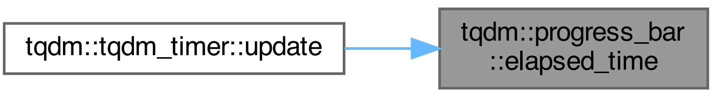
Returns

The elapsed time in seconds.

Here is the call graph for this function:



Here is the caller graph for this function:



12.13.2.3 operator<<()

```
template<class T >
progress_bar & tqdm::progress_bar::operator<< (
            const T & t )  [inline]
```

Append a value to the progress bar suffix.

Parameters

t	The value to append.

Returns

A reference to the progress bar.

12.13.2.4 print_bar()

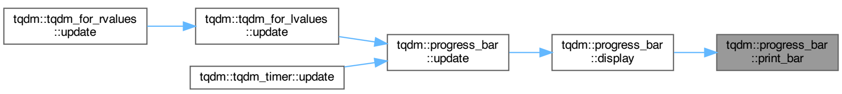
```
void tqdm::progress_bar::print_bar (
            std::stringstream & ss,
            double filled ) const  [inline], [private]
```

Print the progress bar with symbols.

Parameters

ss	The stringstream to print the bar into.
filled	The fraction of the bar that is filled.

Here is the caller graph for this function:

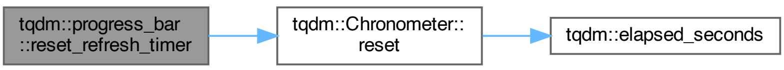


12.13.2.5 reset_refresh_timer()

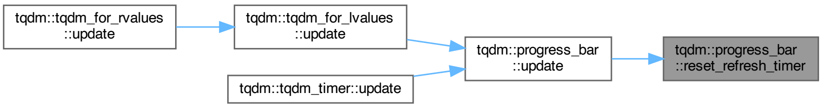
```
void tqdm::progress_bar::reset_refresh_timer ( )  [inline], [private]
```

Reset the refresh timer.

Here is the call graph for this function:



Here is the caller graph for this function:

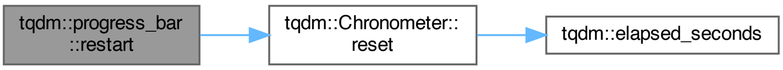


12.13.2.6 restart()

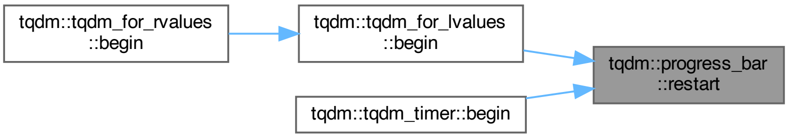
`void tqdm::progress_bar::restart ( ) [inline]`

Restart the progress bar.

Here is the call graph for this function:



Here is the caller graph for this function:



12.13.2.7 set_bar_size()

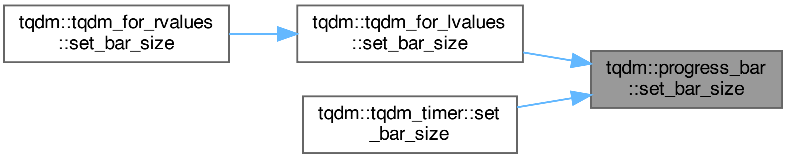
```
void tqdm::progress_bar::set_bar_size (
                int size ) [inline]
```

Set the size of the progress bar.

Parameters

size	The size of the progress bar in characters.

Here is the caller graph for this function:



12.13.2.8 set_min_update_time()

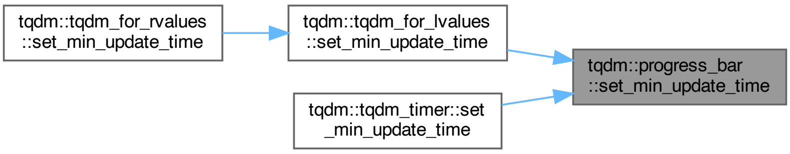
```
void tqdm::progress_bar::set_min_update_time (
            double time ) [inline]
```

Set the minimum time between updates.

Parameters

time	The minimum time between updates in seconds.

Here is the caller graph for this function:



12.13.2.9 set_ostream()

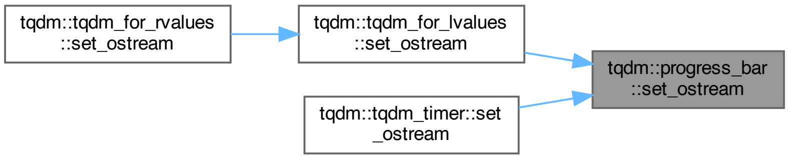
```
void tqdm::progress_bar::set_ostream (
            std::ostream & os ) [inline]
```

Set the output stream for the progress bar.

Parameters

os	The output stream.

Here is the caller graph for this function:



12.13.2.10 set_prefix()

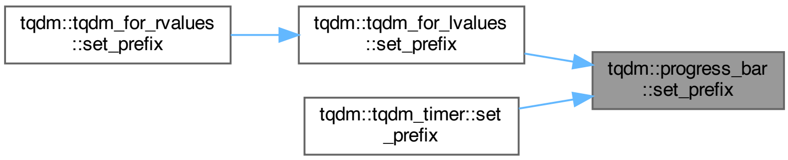
```
void tqdm::progress_bar::set_prefix (
             std::string s ) [inline]
```

Set the prefix string for the progress bar.

Parameters

s	The prefix string.

Here is the caller graph for this function:



12.13.2.11 time_since_refresh()

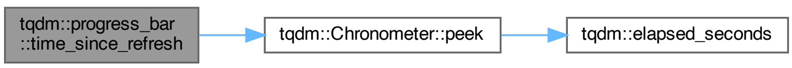
```
double tqdm::progress_bar::time_since_refresh ( ) const [inline], [private]
```

Get the time since the last refresh.

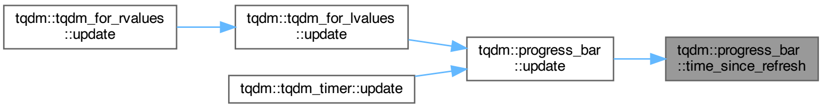
Returns

The time since the last refresh in seconds.

Here is the call graph for this function:



Here is the caller graph for this function:



12.13.2.12 update()

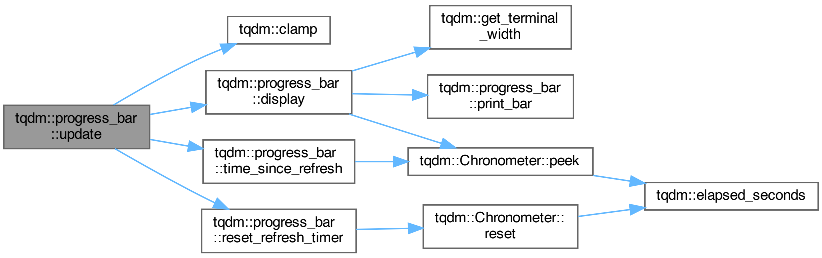
```
void tqdm::progress_bar::update (
            double progress )  [inline]
```

Update the progress bar with the current progress.

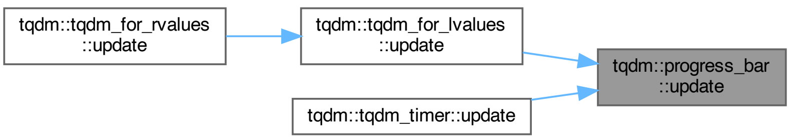
Parameters

progress	The current progress (a value between 0 and 1).

Here is the call graph for this function:



Here is the caller graph for this function:



12.13.3 Field Documentation

12.13.3.1 bar_size_

```
index tqdm::progress_bar::bar_size_ {40}  [private]
```

12.13.3.2 chronometer_

```
Chronometer tqdm::progress_bar::chronometer_ {}  [private]
```

12.13.3.3 min_time_per_update_

```
double tqdm::progress_bar::min_time_per_update_ {0.15} [private]
```

12.13.3.4 os_

```
std::ostream* tqdm::progress_bar::os_ {&std::cerr} [private]
```

12.13.3.5 prefix_

```
std::string tqdm::progress_bar::prefix_ {} [private]
```

12.13.3.6 refresh_

```
Chronometer tqdm::progress_bar::refresh_ {} [private]
```

12.13.3.7 suffix_

```
std::stringstream tqdm::progress_bar::suffix_ {} [private]
```

12.13.3.8 symbol_index_

```
index tqdm::progress_bar::symbol_index_ {0} [private]
```

12.13.3.9 term_cols_

```
index tqdm::progress_bar::term_cols_ {1} [private]
```

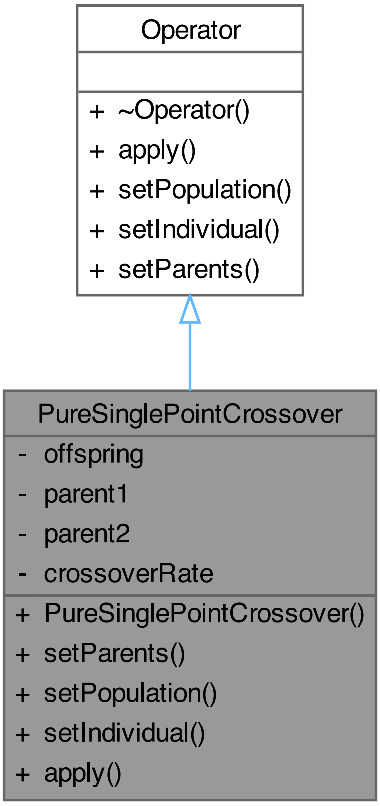
The documentation for this class was generated from the following file:

- include/utils/tqdm.hpp

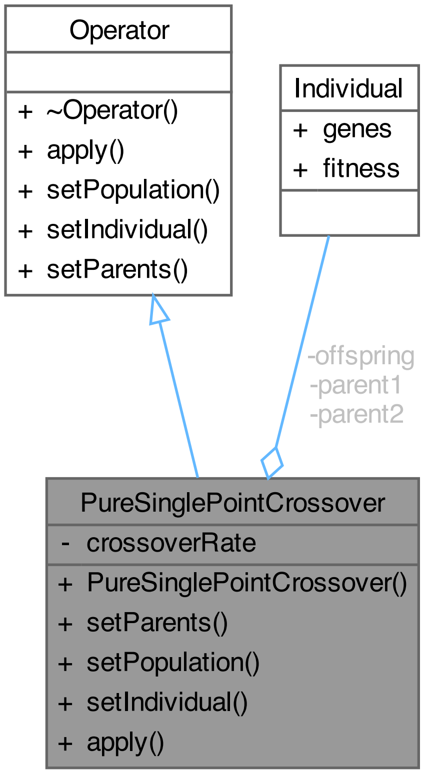
12.14 PureSinglePointCrossover Class Reference

```
#include <PureSinglePointCrossover.h>
```

Inheritance diagram for PureSinglePointCrossover:



Collaboration diagram for PureSinglePointCrossover:



Public Member Functions

- PureSinglePointCrossover (const Algorithm_Parameters &params)

  Constructs a PureSinglePointCrossover object with specified algorithm parameters.
- void setParents (Individual &offspring, const Individual &parent1, const Individual &parent2) override

  Sets the parents and offspring for the crossover operation.
- void setPopulation (const std::vector< Individual > &population, std::vector< Individual > &selected)

  Sets the population and selected vectors for the selection operation.
- void setIndividual (Individual &individual)

  Sets the individual for the selection operation.
- void apply (std::mt19937 &generator) override

  Performs the single-point crossover operation.

Public Member Functions inherited from Operator

- virtual ∼Operator ()=default

  Virtual destructor for the Operator class.

Private Attributes

- Individual ∗ offspring
- const Individual ∗ parent1

- const Individual ∗ parent2
- double crossoverRate

12.14.1 Constructor & Destructor Documentation

12.14.1.1 PureSinglePointCrossover()

```
PureSinglePointCrossover::PureSinglePointCrossover (
            const Algorithm_Parameters & params )
```

Constructs a PureSinglePointCrossover object with specified algorithm parameters.

Parameters

params	The algorithm parameters including the crossover rate.

12.14.2 Member Function Documentation

12.14.2.1 apply()

```
void PureSinglePointCrossover::apply (
            std::mt19937 & generator ) [override], [virtual]
```

Performs the single-point crossover operation.

This method performs a single-point crossover on the two parent individuals to generate an offspring individual. The crossover point is chosen randomly, and the genetic material from the two parents is combined to create the offspring.

Parameters

generator	The random number generator used to determine the crossover point.

Implements Operator.

12.14.2.2 setIndividual()

```
void PureSinglePointCrossover::setIndividual (
            Individual & individual ) [inline], [virtual]
```

Sets the individual for the selection operation.

This method is not used in the PureSinglePointCrossover class.

Parameters

individual	

Implements Operator.

12.14.2.3 setParents()

```
void PureSinglePointCrossover::setParents (
            Individual & offspring,
            const Individual & parent1,
            const Individual & parent2 ) [override], [virtual]
```

Sets the parents and offspring for the crossover operation.

Parameters

offspring	The offspring individual to be generated.
parent1	The first parent individual.
parent2	The second parent individual.

Implements Operator.

12.14.2.4 setPopulation()

```
void PureSinglePointCrossover::setPopulation (
            const std::vector< Individual > & population,
            std::vector< Individual > & selected )  [inline], [virtual]
```

Sets the population and selected vectors for the selection operation.
This method is not used in the PureSinglePointCrossover class.

Parameters

population	
selected	

Implements Operator.

12.14.3 Field Documentation

12.14.3.1 crossoverRate

```
double PureSinglePointCrossover::crossoverRate  [private]
```

The crossover rate used to determine if crossover should occur

12.14.3.2 offspring

```
Individual* PureSinglePointCrossover::offspring  [private]
```

Pointer to the offspring individual

12.14.3.3 parent1

```
const Individual* PureSinglePointCrossover::parent1  [private]
```

Pointer to the first parent individual

12.14.3.4 parent2

```
const Individual* PureSinglePointCrossover::parent2  [private]
```

Pointer to the second parent individual
The documentation for this class was generated from the following files:

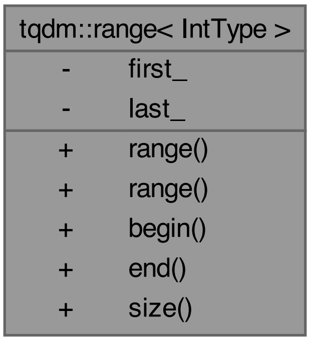
- include/operators/Crossover/PureSinglePointCrossover.h
- src/operators/Crossover/PureSinglePointCrossover.cpp

12.15 tqdm::range< IntType > Class Template Reference

A range of integers represented by iterators.

```
#include <tqdm.hpp>
```

Collaboration diagram for tqdm::range< IntType >:



Public Types

- using iterator = int_iterator<IntType>
- using const_iterator = iterator
- using value_type = IntType

Public Member Functions

- range (IntType first, IntType last)
- range (IntType last)
- iterator begin () const
- iterator end () const
- index size () const

Private Attributes

- iterator first_
- iterator last_

12.15.1 Detailed Description

template<class IntType>
class tqdm::range< IntType >

A range of integers represented by iterators.

Template Parameters

IntType	The type of the integer.

12.15.2 Member Typedef Documentation

12.15.2.1 const_iterator

```
template<class IntType >
using tqdm::range< IntType >::const_iterator = iterator
```

12.15.2.2 iterator

```
template<class IntType >
using tqdm::range< IntType >::iterator = int_iterator<IntType>
```

12.15.2.3 value_type

```
template<class IntType >
using tqdm::range< IntType >::value_type = IntType
```

12.15.3 Constructor & Destructor Documentation

12.15.3.1 range() [1/2]

```
template<class IntType >
tqdm::range< IntType >::range (
            IntType first,
            IntType last )  [inline]
```

12.15.3.2 range() [2/2]

```
template<class IntType >
tqdm::range< IntType >::range (
            IntType last )  [inline], [explicit]
```

12.15.4 Member Function Documentation

12.15.4.1 begin()

```
template<class IntType >
iterator tqdm::range< IntType >::begin ( ) const  [inline]
```

12.15.4.2 end()

```
template<class IntType >
iterator tqdm::range< IntType >::end ( ) const  [inline]
```

12.15.4.3 size()

```
template<class IntType >
index tqdm::range< IntType >::size ( ) const  [inline]
```

12.15.5 Field Documentation

12.15.5.1 first_

```
template<class IntType >
iterator tqdm::range< IntType >::first_  [private]
```

12.15.5.2 last_

```
template<class IntType >
iterator tqdm::range< IntType >::last_  [private]
```

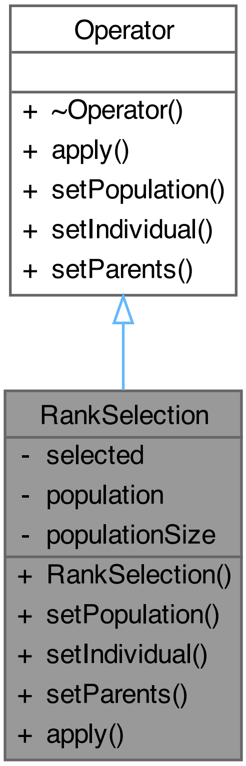
The documentation for this class was generated from the following file:

- include/utils/tqdm.hpp

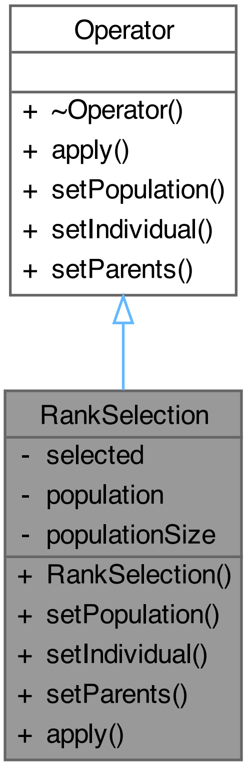
12.16 RankSelection Class Reference

`#include <RankSelection.h>`

Inheritance diagram for RankSelection:



Collaboration diagram for RankSelection:



Public Member Functions

- RankSelection (const Algorithm_Parameters &params)

  Constructs a RankSelection object with specified algorithm parameters.
- void setPopulation (const std::vector< Individual > &population, std::vector< Individual > &selected) override

  Sets the population and selected vectors for the selection operation.
- void setIndividual (Individual &individual)

  Sets the individual to be selected.
- void setParents (Individual &offspring, const Individual &parent1, const Individual &parent2)

  Sets the parents and offspring for the selection operation.
- void apply (std::mt19937 &generator) override

  Performs the rank-based selection operation.

Public Member Functions inherited from Operator

- virtual ∼Operator ()=default

  Virtual destructor for the Operator class.

Private Attributes

- std::vector< Individual > ∗ selected

- const std::vector< Individual > * population
- int populationSize

12.16.1 Constructor & Destructor Documentation

12.16.1.1 RankSelection()

```
RankSelection::RankSelection (
        const Algorithm_Parameters & params )
```

Constructs a RankSelection object with specified algorithm parameters.

Parameters

params	The algorithm parameters including the population size.

12.16.2 Member Function Documentation

12.16.2.1 apply()

```
void RankSelection::apply (
        std::mt19937 & generator ) [override], [virtual]
```

Performs the rank-based selection operation.

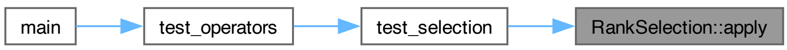
This method selects individuals from the population based on their rank and stores the selected individuals in the provided vector.

Parameters

generator	The random number generator used for selection.

Implements Operator.

Here is the caller graph for this function:



12.16.2.2 setIndividual()

```
void RankSelection::setIndividual (
        Individual & individual ) [inline], [virtual]
```

Sets the individual to be selected.

This method sets the individual to be selected from the population.

Parameters

individual	The individual to be selected.

Implements Operator.

12.16.2.3 setParents()

```
void RankSelection::setParents (
        Individual & offspring,
```

```
            const Individual & parent1,
            const Individual & parent2 )  [inline], [virtual]
```

Sets the parents and offspring for the selection operation.
This method is not used by the RankSelection operator.

Parameters

offspring	The offspring individual to be generated.
parent1	The first parent individual.
parent2	The second parent individual.

Implements Operator.

12.16.2.4 setPopulation()

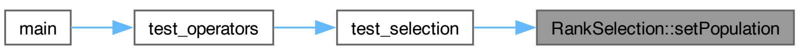
```
void RankSelection::setPopulation (
            const std::vector< Individual > & population,
            std::vector< Individual > & selected )  [override], [virtual]
```

Sets the population and selected vectors for the selection operation.

Parameters

population	The vector of individuals in the population.
selected	The vector of selected individuals.

Implements Operator.
Here is the caller graph for this function:



12.16.3 Field Documentation

12.16.3.1 population

```
const std::vector<Individual>* RankSelection::population  [private]
```

Pointer to the vector of individuals in the population

12.16.3.2 populationSize

```
int RankSelection::populationSize  [private]
```

Size of the population

12.16.3.3 selected

```
std::vector<Individual>* RankSelection::selected  [private]
```

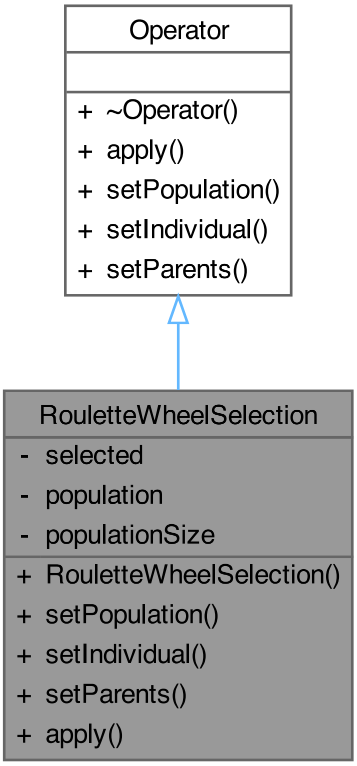
Pointer to the vector of selected individuals
The documentation for this class was generated from the following files:

- include/operators/Selection/RankSelection.h
- src/operators/Selection/RankSelection.cpp

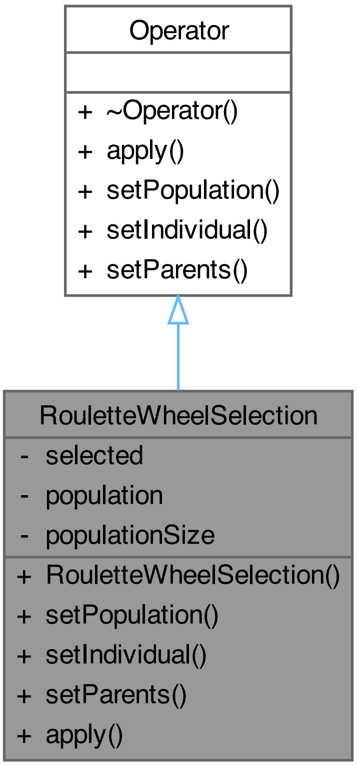
12.17 RouletteWheelSelection Class Reference

`#include <RouletteWheelSelection.h>`

Inheritance diagram for RouletteWheelSelection:



Collaboration diagram for RouletteWheelSelection:



Public Member Functions

- RouletteWheelSelection (const Algorithm_Parameters &params)

    Constructs a RouletteWheelSelection object with specified algorithm parameters.
- void setPopulation (const std::vector< Individual > &population, std::vector< Individual > &selected)

    Sets the population and selected vectors for the selection operation.
- void setIndividual (Individual &individual)

    Sets the individual to be mutated.
- void setParents (Individual &offspring, const Individual &parent1, const Individual &parent2)

    Sets the parents and offspring for the crossover operation.
- void apply (std::mt19937 &generator) override

    Performs the roulette wheel selection operation.

Public Member Functions inherited from Operator

- virtual ∼Operator ()=default

    Virtual destructor for the Operator class.

Private Attributes

- std::vector< Individual > ∗ selected
- const std::vector< Individual > ∗ population
- int populationSize

12.17.1 Constructor & Destructor Documentation

12.17.1.1 RouletteWheelSelection()

```
RouletteWheelSelection::RouletteWheelSelection (
            const Algorithm_Parameters & params )
```

Constructs a RouletteWheelSelection object with specified algorithm parameters.

Parameters

params	The algorithm parameters.

12.17.2 Member Function Documentation

12.17.2.1 apply()

```
void RouletteWheelSelection::apply (
            std::mt19937 & generator )  [override], [virtual]
```

Performs the roulette wheel selection operation.

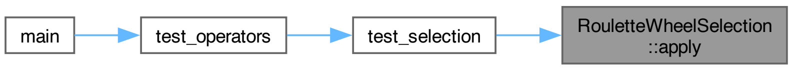
This method selects individuals from the population based on fitness-proportional selection and stores the selected individuals in the provided vector.

Parameters

generator	The random number generator used for selection.

Implements Operator.

Here is the caller graph for this function:



12.17.2.2 setIndividual()

```
void RouletteWheelSelection::setIndividual (
            Individual & individual )  [inline], [virtual]
```

Sets the individual to be mutated.

Parameters

individual	The individual to be mutated.
individual	The individual to be mutated.

Implements Operator.

12.17.2.3 setParents()

```
void RouletteWheelSelection::setParents (
            Individual & offspring,
            const Individual & parent1,
            const Individual & parent2 )  [inline], [virtual]
```

Sets the parents and offspring for the crossover operation.

This method is not used by the RouletteWheelSelection operator.

Parameters

offspring	The offspring individual to be generated.
parent1	The first parent individual.
parent2	The second parent individual.

Implements Operator.

12.17.2.4 setPopulation()

```
void RouletteWheelSelection::setPopulation (
            const std::vector< Individual > & population,
            std::vector< Individual > & selected )  [virtual]
```

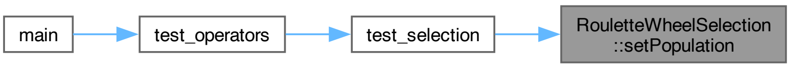
Sets the population and selected vectors for the selection operation.

Parameters

population	The vector of individuals in the population.
selected	The vector of selected individuals.

Implements Operator.

Here is the caller graph for this function:



12.17.3 Field Documentation

12.17.3.1 population

```
const std::vector<Individual>* RouletteWheelSelection::population  [private]
```

Pointer to the vector of individuals in the population

12.17.3.2 populationSize

```
int RouletteWheelSelection::populationSize  [private]
```

Size of the population

12.17.3.3 selected

```
std::vector<Individual>* RouletteWheelSelection::selected  [private]
```

Pointer to the vector of selected individuals

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

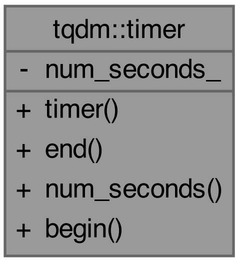
- include/operators/Selection/RouletteWheelSelection.h
- src/operators/Selection/RouletteWheelSelection.cpp

12.18 tqdm::timer Struct Reference

A timer that can be used to create a timing iterator.

```
#include <tqdm.hpp>
```

Collaboration diagram for tqdm::timer:



Public Types

- using iterator = timing_iterator
- using end_iterator = timing_iterator_end_sentinel
- using const_iterator = iterator
- using value_type = double

Public Member Functions

- timer (double num_seconds)
- end_iterator end () const
- double num_seconds () const

Static Public Member Functions

- static iterator begin ()

Private Attributes

- double num_seconds_

12.18.1 Detailed Description

A timer that can be used to create a timing iterator.

12.18.2 Member Typedef Documentation

12.18.2.1 const_iterator

```
using tqdm::timer::const_iterator = iterator
```

12.18.2.2 end_iterator

```
using tqdm::timer::end_iterator = timing_iterator_end_sentinel
```

12.18.2.3 iterator

```
using tqdm::timer::iterator = timing_iterator
```

12.18.2.4 value_type

```
using tqdm::timer::value_type = double
```

12.18.3 Constructor & Destructor Documentation

12.18.3.1 timer()

```
tqdm::timer::timer (
            double num_seconds )  [inline], [explicit]
```

12.18.4 Member Function Documentation

12.18.4.1 begin()

```
static iterator tqdm::timer::begin ( )  [inline], [static]
```

12.18.4.2 end()

```
end_iterator tqdm::timer::end ( ) const  [inline]
```

12.18.4.3 num_seconds()

```
double tqdm::timer::num_seconds ( ) const  [inline]
```

Here is the caller graph for this function:

12.18.5 Field Documentation

12.18.5.1 num_seconds_

```
double tqdm::timer::num_seconds_  [private]
```

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

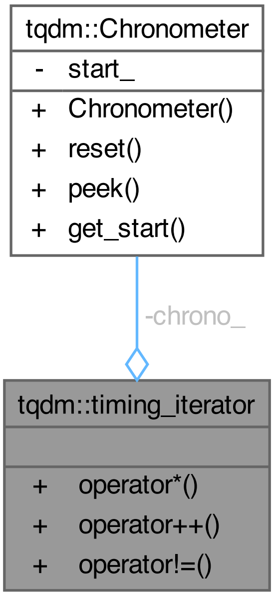
- include/utils/tqdm.hpp

12.19 tqdm::timing_iterator Class Reference

A forward iterator that returns the elapsed time.

```
#include <tqdm.hpp>
```

Collaboration diagram for tqdm::timing_iterator:



Public Types

- using iterator_category = std::forward_iterator_tag
- using value_type = double
- using difference_type = double
- using pointer = double ∗
- using reference = double &

Public Member Functions

- double operator∗ () const
- timing_iterator & operator++ ()
- bool operator!= (const timing_iterator_end_sentinel &other) const

Private Attributes

- tqdm::Chronometer chrono_

12.19.1 Detailed Description

A forward iterator that returns the elapsed time.

12.19.2 Member Typedef Documentation

12.19.2.1 difference_type

```
using tqdm::timing_iterator::difference_type = double
```

12.19.2.2 iterator_category

```
using tqdm::timing_iterator::iterator_category = std::forward_iterator_tag
```

12.19.2.3 pointer

```
using tqdm::timing_iterator::pointer = double *
```

12.19.2.4 reference

```
using tqdm::timing_iterator::reference = double &
```

12.19.2.5 value_type

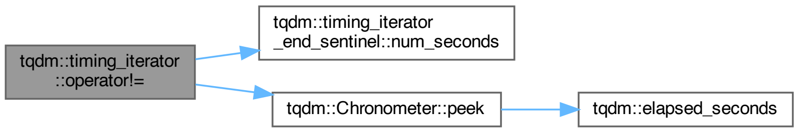
```
using tqdm::timing_iterator::value_type = double
```

12.19.3 Member Function Documentation

12.19.3.1 operator"!=()

```
bool tqdm::timing_iterator::operator!= (
            const timing_iterator_end_sentinel & other ) const  [inline]
```

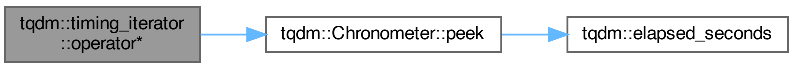
Here is the call graph for this function:



12.19.3.2 operator*()

```
double tqdm::timing_iterator::operator* ( ) const  [inline]
```

Here is the call graph for this function:



12.19.3.3 operator++()

```
timing_iterator & tqdm::timing_iterator::operator++ ( )  [inline]
```

12.19.4 Field Documentation

12.19.4.1 chrono_

```
tqdm::Chronometer tqdm::timing_iterator::chrono_  [private]
```

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

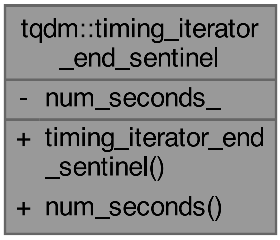
- include/utils/tqdm.hpp

12.20 tqdm::timing_iterator_end_sentinel Class Reference

An end sentinel for timing iterators.

`#include <tqdm.hpp>`

Collaboration diagram for tqdm::timing_iterator_end_sentinel:



Public Member Functions

- timing_iterator_end_sentinel (double num_seconds)
- double num_seconds () const

Private Attributes

- double num_seconds_

12.20.1 Detailed Description

An end sentinel for timing iterators.

12.20.2 Constructor & Destructor Documentation

12.20.2.1 timing_iterator_end_sentinel()

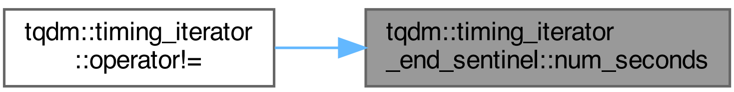
```
tqdm::timing_iterator_end_sentinel::timing_iterator_end_sentinel (
            double num_seconds )  [inline], [explicit]
```

12.20.3 Member Function Documentation

12.20.3.1 num_seconds()

```
double tqdm::timing_iterator_end_sentinel::num_seconds ( ) const  [inline]
```

Here is the caller graph for this function:



12.20.4 Field Documentation

12.20.4.1 num_seconds_

`double tqdm::timing_iterator_end_sentinel::num_seconds_ [private]`

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

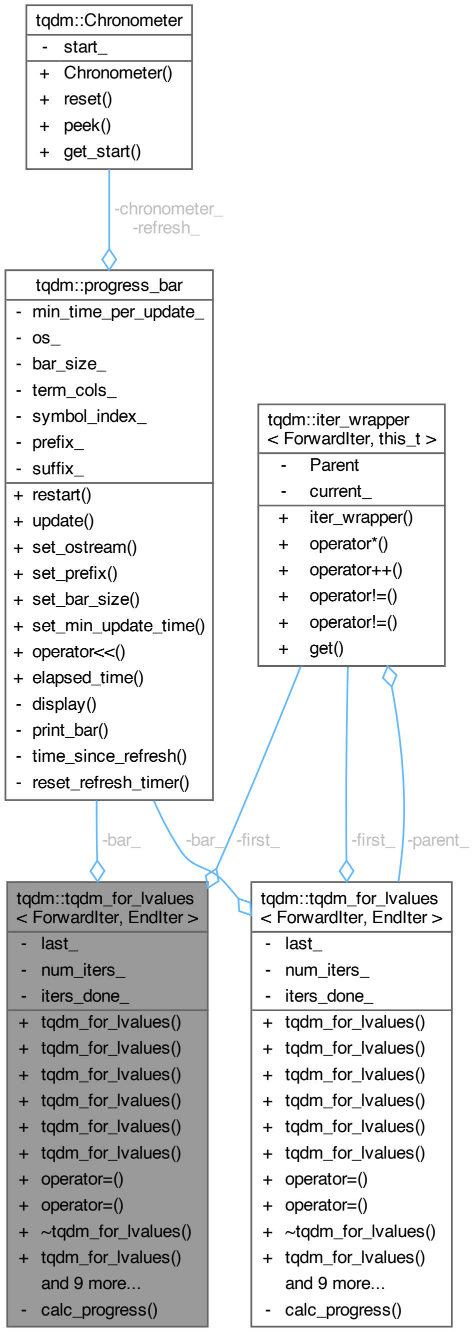
- include/utils/tqdm.hpp

12.21 tqdm::tqdm_for_lvalues< ForwardIter, EndIter > Class Template Reference

A class for progress bars that iterate over lvalues.

`#include <tqdm.hpp>`

Collaboration diagram for tqdm::tqdm_for_lvalues< ForwardIter, EndIter >:



Public Types

- using this_t = tqdm_for_lvalues<ForwardIter, EndIter>
- using iterator = iter_wrapper<ForwardIter, this_t>
- using value_type = typename ForwardIter::value_type
- using size_type = index
- using difference_type = index

Public Member Functions

- tqdm_for_lvalues (ForwardIter begin, EndIter end)
- tqdm_for_lvalues (ForwardIter begin, EndIter end, index total)
- template<class Container >
  tqdm_for_lvalues (Container &C)
- template<class Container >
  tqdm_for_lvalues (const Container &C)
- tqdm_for_lvalues (const tqdm_for_lvalues &)=delete
- tqdm_for_lvalues (tqdm_for_lvalues &&)=delete
- tqdm_for_lvalues & operator= (tqdm_for_lvalues &&)=delete
- tqdm_for_lvalues & operator= (const tqdm_for_lvalues &)=delete
- ~tqdm_for_lvalues ()=default
- template<class Container >
  tqdm_for_lvalues (Container &&)=delete
- iterator begin ()
- EndIter end () const
- void update ()
- void set_ostream (std::ostream &os)
- void set_prefix (std::string s)
- void set_bar_size (int size)
- void set_min_update_time (double time)
- template<class T >
  tqdm_for_lvalues & operator<< (const T &t)
- void manually_set_progress (double to)

Private Member Functions

- double calc_progress () const

Private Attributes

- iterator first_
- EndIter last_
- index num_iters_ {0}
- index iters_done_ {0}
- progress_bar bar_

12.21.1 Detailed Description

template<class ForwardIter, class EndIter = ForwardIter>
class tqdm::tqdm_for_lvalues< ForwardIter, EndIter >

A class for progress bars that iterate over lvalues.

Template Parameters

ForwardIter	The type of the forward iterator.
EndIter	The type of the end iterator.

12.21.2 Member Typedef Documentation

12.21.2.1 difference_type

```
template<class ForwardIter , class EndIter = ForwardIter>
using tqdm::tqdm_for_lvalues< ForwardIter, EndIter >::difference_type = index
```

12.21.2.2 iterator

```
template<class ForwardIter , class EndIter = ForwardIter>
using tqdm::tqdm_for_lvalues< ForwardIter, EndIter >::iterator = iter_wrapper<ForwardIter,
this_t>
```

12.21.2.3 size_type

```
template<class ForwardIter , class EndIter = ForwardIter>
using tqdm::tqdm_for_lvalues< ForwardIter, EndIter >::size_type = index
```

12.21.2.4 this_t

```
template<class ForwardIter , class EndIter = ForwardIter>
using tqdm::tqdm_for_lvalues< ForwardIter, EndIter >::this_t = tqdm_for_lvalues<ForwardIter,
EndIter>
```

12.21.2.5 value_type

```
template<class ForwardIter , class EndIter = ForwardIter>
using tqdm::tqdm_for_lvalues< ForwardIter, EndIter >::value_type = typename ForwardIter↩
::value_type
```

12.21.3 Constructor & Destructor Documentation

12.21.3.1 tqdm_for_lvalues() [1/7]

```
template<class ForwardIter , class EndIter = ForwardIter>
tqdm::tqdm_for_lvalues< ForwardIter, EndIter >::tqdm_for_lvalues (
            ForwardIter begin,
            EndIter end )  [inline]
```

12.21.3.2 tqdm_for_lvalues() [2/7]

```
template<class ForwardIter , class EndIter = ForwardIter>
tqdm::tqdm_for_lvalues< ForwardIter, EndIter >::tqdm_for_lvalues (
            ForwardIter begin,
            EndIter end,
            index total )  [inline]
```

12.21.3.3 tqdm_for_lvalues() [3/7]

```
template<class ForwardIter , class EndIter = ForwardIter>
template<class Container >
tqdm::tqdm_for_lvalues< ForwardIter, EndIter >::tqdm_for_lvalues (
            Container & C )  [inline], [explicit]
```

12.21.3.4 tqdm_for_lvalues() [4/7]

```
template<class ForwardIter , class EndIter = ForwardIter>
template<class Container >
tqdm::tqdm_for_lvalues< ForwardIter, EndIter >::tqdm_for_lvalues (
            const Container & C )  [inline], [explicit]
```

12.21.3.5 tqdm_for_lvalues() [5/7]

```
template<class ForwardIter , class EndIter = ForwardIter>
tqdm::tqdm_for_lvalues< ForwardIter, EndIter >::tqdm_for_lvalues (
            const tqdm_for_lvalues< ForwardIter, EndIter > & )  [delete]
```

12.21.3.6 tqdm_for_lvalues() [6/7]

```
template<class ForwardIter , class EndIter = ForwardIter>
tqdm::tqdm_for_lvalues< ForwardIter, EndIter >::tqdm_for_lvalues (
            tqdm_for_lvalues< ForwardIter, EndIter > &&  )  [delete]
```

12.21.3.7 ~tqdm_for_lvalues()

```
template<class ForwardIter , class EndIter = ForwardIter>
tqdm::tqdm_for_lvalues< ForwardIter, EndIter >::~tqdm_for_lvalues ( )  [default]
```

12.21.3.8 tqdm_for_lvalues() [7/7]

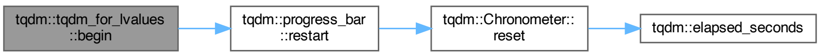
```
template<class ForwardIter , class EndIter = ForwardIter>
template<class Container >
tqdm::tqdm_for_lvalues< ForwardIter, EndIter >::tqdm_for_lvalues (
            Container &&  )  [delete]
```

12.21.4 Member Function Documentation

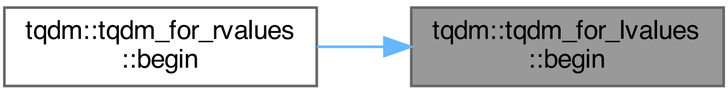
12.21.4.1 begin()

```
template<class ForwardIter , class EndIter = ForwardIter>
iterator tqdm::tqdm_for_lvalues< ForwardIter, EndIter >::begin ( )  [inline]
```

Here is the call graph for this function:



Here is the caller graph for this function:



12.21.4.2 calc_progress()

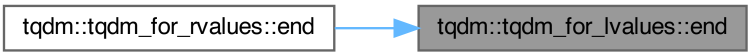
```
template<class ForwardIter , class EndIter = ForwardIter>
double tqdm::tqdm_for_lvalues< ForwardIter, EndIter >::calc_progress ( ) const  [inline],
[private]
```

Here is the caller graph for this function:

12.21.4.3 end()

```
template<class ForwardIter , class EndIter = ForwardIter>
EndIter tqdm::tqdm_for_lvalues< ForwardIter, EndIter >::end ( ) const  [inline]
```

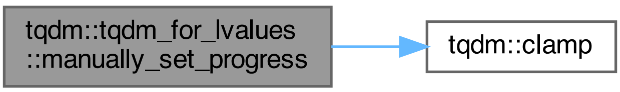
Here is the caller graph for this function:



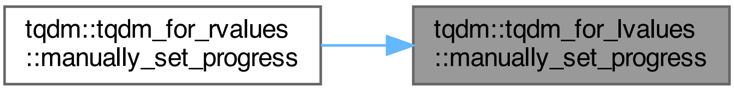
12.21.4.4 manually_set_progress()

```
template<class ForwardIter , class EndIter = ForwardIter>
void tqdm::tqdm_for_lvalues< ForwardIter, EndIter >::manually_set_progress (
             double to )  [inline]
```

Here is the call graph for this function:



Here is the caller graph for this function:



12.21.4.5 operator<<()

```
template<class ForwardIter , class EndIter = ForwardIter>
template<class T >
tqdm_for_lvalues & tqdm::tqdm_for_lvalues< ForwardIter, EndIter >::operator<< (
            const T & t ) [inline]
```

12.21.4.6 operator=() [1/2]

```
template<class ForwardIter , class EndIter = ForwardIter>
tqdm_for_lvalues & tqdm::tqdm_for_lvalues< ForwardIter, EndIter >::operator= (
            const tqdm_for_lvalues< ForwardIter, EndIter > & ) [delete]
```

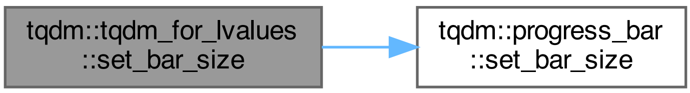
12.21.4.7 operator=() [2/2]

```
template<class ForwardIter , class EndIter = ForwardIter>
tqdm_for_lvalues & tqdm::tqdm_for_lvalues< ForwardIter, EndIter >::operator= (
            tqdm_for_lvalues< ForwardIter, EndIter > && ) [delete]
```

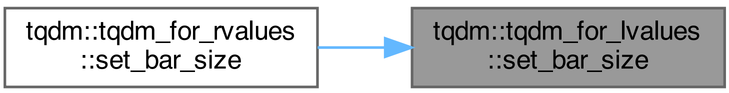
12.21.4.8 set_bar_size()

```
template<class ForwardIter , class EndIter = ForwardIter>
void tqdm::tqdm_for_lvalues< ForwardIter, EndIter >::set_bar_size (
            int size ) [inline]
```

Here is the call graph for this function:



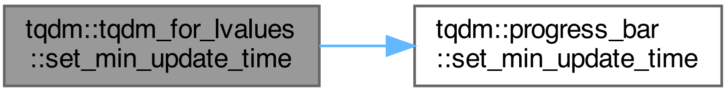
Here is the caller graph for this function:



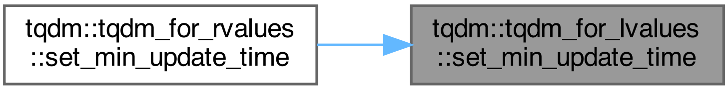
12.21.4.9 set_min_update_time()

```
template<class ForwardIter , class EndIter = ForwardIter>
void tqdm::tqdm_for_lvalues< ForwardIter, EndIter >::set_min_update_time (
            double time ) [inline]
```

Here is the call graph for this function:



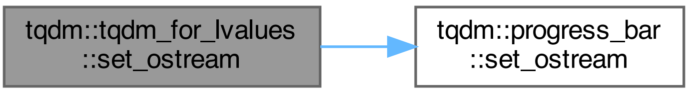
Here is the caller graph for this function:



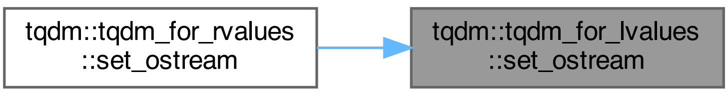
12.21.4.10 set_ostream()

```
template<class ForwardIter , class EndIter = ForwardIter>
void tqdm::tqdm_for_lvalues< ForwardIter, EndIter >::set_ostream (
            std::ostream & os )  [inline]
```

Here is the call graph for this function:



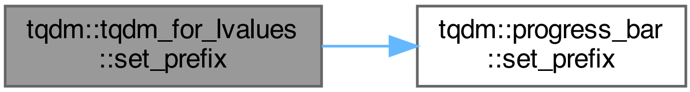
Here is the caller graph for this function:



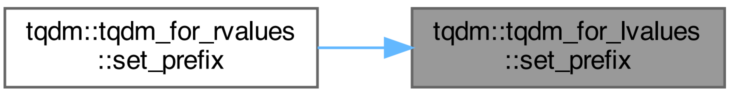
12.21.4.11 set_prefix()

```
template<class ForwardIter , class EndIter = ForwardIter>
void tqdm::tqdm_for_lvalues< ForwardIter, EndIter >::set_prefix (
             std::string s ) [inline]
```

Here is the call graph for this function:



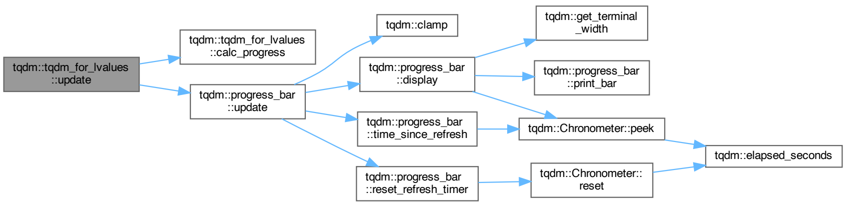
Here is the caller graph for this function:



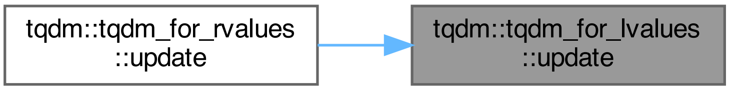
12.21.4.12 update()

```
template<class ForwardIter , class EndIter = ForwardIter>
void tqdm::tqdm_for_lvalues< ForwardIter, EndIter >::update ( ) [inline]
```

Here is the call graph for this function:



Here is the caller graph for this function:



12.21.5 Field Documentation

12.21.5.1 bar_

```
template<class ForwardIter , class EndIter = ForwardIter>
progress_bar tqdm::tqdm_for_lvalues< ForwardIter, EndIter >::bar_ [private]
```

12.21.5.2 first_

```
template<class ForwardIter , class EndIter = ForwardIter>
iterator tqdm::tqdm_for_lvalues< ForwardIter, EndIter >::first_ [private]
```

12.21.5.3 iters_done_

```
template<class ForwardIter , class EndIter = ForwardIter>
index tqdm::tqdm_for_lvalues< ForwardIter, EndIter >::iters_done_ {0} [private]
```

12.21.5.4 last_

```
template<class ForwardIter , class EndIter = ForwardIter>
EndIter tqdm::tqdm_for_lvalues< ForwardIter, EndIter >::last_ [private]
```

12.21.5.5 num_iters_

```
template<class ForwardIter , class EndIter = ForwardIter>
index tqdm::tqdm_for_lvalues< ForwardIter, EndIter >::num_iters_ {0} [private]
```

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

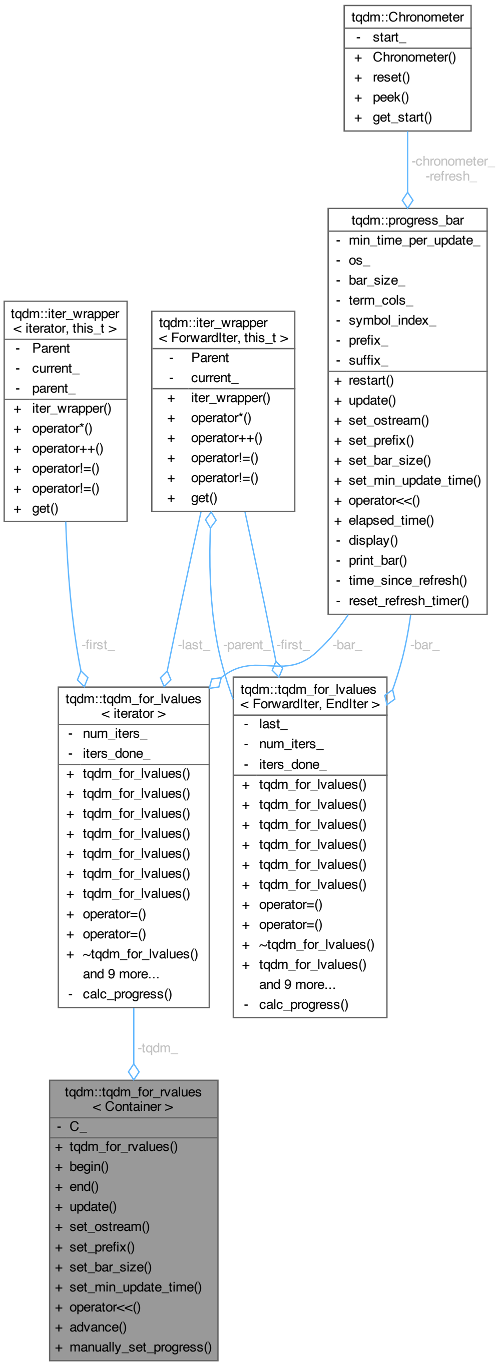
- include/utils/tqdm.hpp

12.22 tqdm::tqdm_for_rvalues< Container > Class Template Reference

A class for progress bars that iterate over rvalues.

```
#include <tqdm.hpp>
```

Collaboration diagram for tqdm::tqdm_for_rvalues< Container >:



Public Types

- using iterator = typename Container::iterator
- using const_iterator = typename Container::const_iterator
- using value_type = typename Container::value_type

Public Member Functions

- tqdm_for_rvalues (Container &&C)

- auto begin ()
- auto end ()
- void update ()
- void set_ostream (std::ostream &os)
- void set_prefix (std::string s)
- void set_bar_size (int size)
- void set_min_update_time (double time)
- template<class T >
  auto & operator<< (const T &t)
- void advance (index amount)
- void manually_set_progress (double to)

Private Attributes

- Container C_
- tqdm_for_lvalues< iterator > tqdm_

12.22.1 Detailed Description

template<class Container>
class tqdm::tqdm_for_rvalues< Container >

A class for progress bars that iterate over rvalues.

Template Parameters

Container	The type of the container.

12.22.2 Member Typedef Documentation

12.22.2.1 const_iterator

```
template<class Container >
using tqdm::tqdm_for_rvalues< Container >::const_iterator = typename Container::const_iterator
```

12.22.2.2 iterator

```
template<class Container >
using tqdm::tqdm_for_rvalues< Container >::iterator = typename Container::iterator
```

12.22.2.3 value_type

```
template<class Container >
using tqdm::tqdm_for_rvalues< Container >::value_type = typename Container::value_type
```

12.22.3 Constructor & Destructor Documentation

12.22.3.1 tqdm_for_rvalues()

```
template<class Container >
tqdm::tqdm_for_rvalues< Container >::tqdm_for_rvalues (
            Container && C )  [inline], [explicit]
```

12.22.4 Member Function Documentation

12.22.4.1 advance()

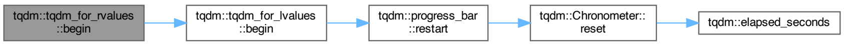
```
template<class Container >
void tqdm::tqdm_for_rvalues< Container >::advance (
```

index *amount*) [inline]

12.22.4.2 begin()

```
template<class Container >
auto tqdm::tqdm_for_rvalues< Container >::begin ( ) [inline]
```

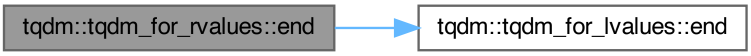
Here is the call graph for this function:



12.22.4.3 end()

```
template<class Container >
auto tqdm::tqdm_for_rvalues< Container >::end ( ) [inline]
```

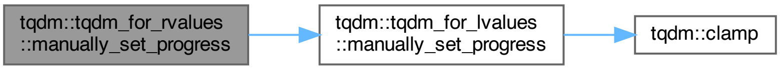
Here is the call graph for this function:



12.22.4.4 manually_set_progress()

```
template<class Container >
void tqdm::tqdm_for_rvalues< Container >::manually_set_progress (
            double to ) [inline]
```

Here is the call graph for this function:



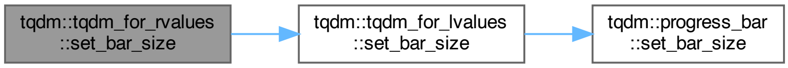
12.22.4.5 operator<<()

```
template<class Container >
template<class T >
auto & tqdm::tqdm_for_rvalues< Container >::operator<< (
            const T & t ) [inline]
```

12.22.4.6 set_bar_size()

```
template<class Container >
void tqdm::tqdm_for_rvalues< Container >::set_bar_size (
            int size ) [inline]
```

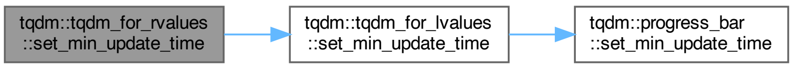
Here is the call graph for this function:



12.22.4.7 set_min_update_time()

```
template<class Container >
void tqdm::tqdm_for_rvalues< Container >::set_min_update_time (
            double time ) [inline]
```

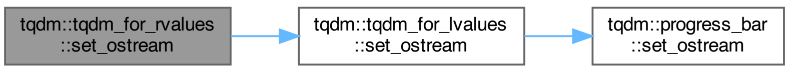
Here is the call graph for this function:



12.22.4.8 set_ostream()

```
template<class Container >
void tqdm::tqdm_for_rvalues< Container >::set_ostream (
            std::ostream & os ) [inline]
```

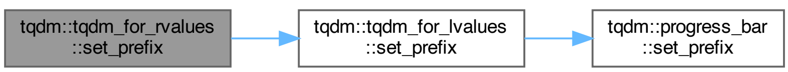
Here is the call graph for this function:



12.22.4.9 set_prefix()

```
template<class Container >
void tqdm::tqdm_for_rvalues< Container >::set_prefix (
            std::string s ) [inline]
```

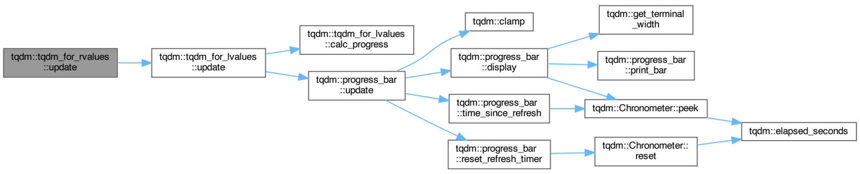
Here is the call graph for this function:



12.22.4.10 update()

```
template<class Container >
void tqdm::tqdm_for_rvalues< Container >::update ( ) [inline]
```

Here is the call graph for this function:



12.22.5 Field Documentation

12.22.5.1 C_

```
template<class Container >
Container tqdm::tqdm_for_rvalues< Container >::C_ [private]
```

12.22.5.2 tqdm_

```
template<class Container >
tqdm_for_lvalues<iterator> tqdm::tqdm_for_rvalues< Container >::tqdm_ [private]
```

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

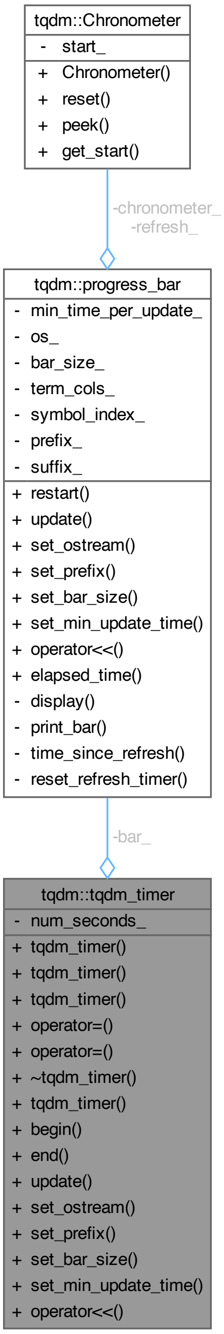
- include/utils/tqdm.hpp

12.23 tqdm::tqdm_timer Class Reference

A progress bar for a timer.

```
#include <tqdm.hpp>
```

Collaboration diagram for tqdm::tqdm_timer:



Public Types

- using iterator = iter_wrapper< timing_iterator, tqdm_timer >
- using end_iterator = timer::end_iterator
- using value_type = typename timing_iterator::value_type
- using size_type = index
- using difference_type = index

Public Member Functions

- tqdm_timer (double num_seconds)
- tqdm_timer (const tqdm_timer &)=delete
- tqdm_timer (tqdm_timer &&)=delete
- tqdm_timer & operator= (tqdm_timer &&)=delete
- tqdm_timer & operator= (const tqdm_timer &)=delete
- ~tqdm_timer ()=default
- template<class Container >
  tqdm_timer (Container &&)=delete
- iterator begin ()
- end_iterator end () const
- void update ()
- void set_ostream (std::ostream &os)
- void set_prefix (std::string s)
- void set_bar_size (int size)
- void set_min_update_time (double time)
- template<class T >
  tqdm_timer & operator<< (const T &t)

Private Attributes

- double num_seconds_
- progress_bar bar_

12.23.1 Detailed Description

A progress bar for a timer.

12.23.2 Member Typedef Documentation

12.23.2.1 difference_type

```
using tqdm::tqdm_timer::difference_type = index
```

12.23.2.2 end_iterator

```
using tqdm::tqdm_timer::end_iterator = timer::end_iterator
```

12.23.2.3 iterator

```
using tqdm::tqdm_timer::iterator = iter_wrapper<timing_iterator, tqdm_timer>
```

12.23.2.4 size_type

```
using tqdm::tqdm_timer::size_type = index
```

12.23.2.5 value_type

```
using tqdm::tqdm_timer::value_type = typename timing_iterator::value_type
```

12.23.3 Constructor & Destructor Documentation

12.23.3.1 tqdm_timer() [1/4]

```
tqdm::tqdm_timer::tqdm_timer (
            double num_seconds )  [inline], [explicit]
```

12.23.3.2 tqdm_timer() [2/4]

```
tqdm::tqdm_timer::tqdm_timer (
            const tqdm_timer & ) [delete]
```

12.23.3.3 tqdm_timer() [3/4]

```
tqdm::tqdm_timer::tqdm_timer (
            tqdm_timer && ) [delete]
```

12.23.3.4 ∼tqdm_timer()

```
tqdm::tqdm_timer::∼tqdm_timer ( ) [default]
```

12.23.3.5 tqdm_timer() [4/4]

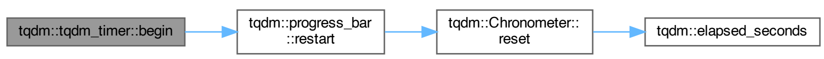
```
template<class Container >
tqdm::tqdm_timer::tqdm_timer (
            Container && ) [delete]
```

12.23.4 Member Function Documentation

12.23.4.1 begin()

```
iterator tqdm::tqdm_timer::begin ( ) [inline]
```

Here is the call graph for this function:



12.23.4.2 end()

```
end_iterator tqdm::tqdm_timer::end ( ) const [inline]
```

12.23.4.3 operator<<()

```
template<class T >
tqdm_timer & tqdm::tqdm_timer::operator<< (
            const T & t ) [inline]
```

12.23.4.4 operator=() [1/2]

```
tqdm_timer & tqdm::tqdm_timer::operator= (
            const tqdm_timer & ) [delete]
```

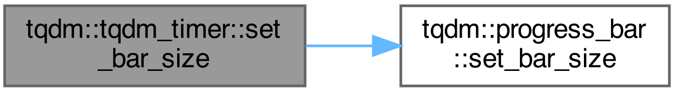
12.23.4.5 operator=() [2/2]

```
tqdm_timer & tqdm::tqdm_timer::operator= (
            tqdm_timer && ) [delete]
```

12.23.4.6 set_bar_size()

```
void tqdm::tqdm_timer::set_bar_size (
            int size ) [inline]
```

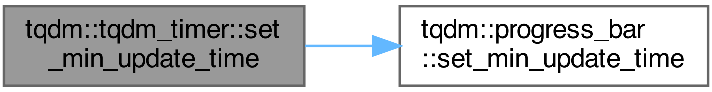
Here is the call graph for this function:



12.23.4.7 set_min_update_time()

```
void tqdm::tqdm_timer::set_min_update_time (
            double time )  [inline]
```

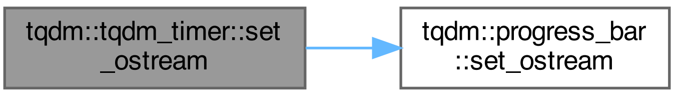
Here is the call graph for this function:



12.23.4.8 set_ostream()

```
void tqdm::tqdm_timer::set_ostream (
            std::ostream & os )  [inline]
```

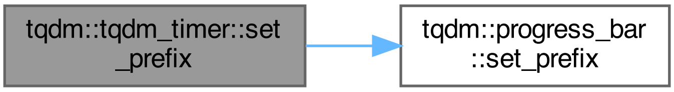
Here is the call graph for this function:



12.23.4.9 set_prefix()

```
void tqdm::tqdm_timer::set_prefix (
            std::string s )  [inline]
```

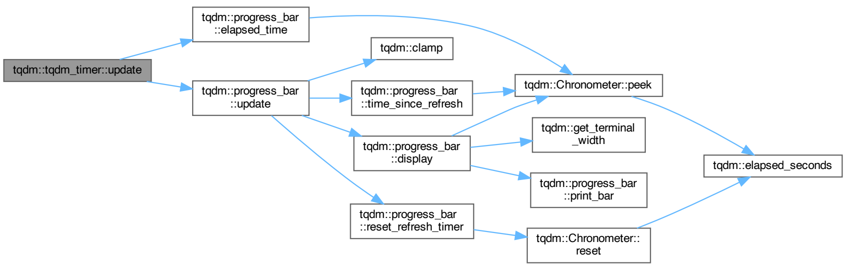
Here is the call graph for this function:



12.23.4.10 update()

```
void tqdm::tqdm_timer::update ( ) [inline]
```

Here is the call graph for this function:



12.23.5 Field Documentation

12.23.5.1 bar_

```
progress_bar tqdm::tqdm_timer::bar_ [private]
```

12.23.5.2 num_seconds_

```
double tqdm::tqdm_timer::num_seconds_ [private]
```

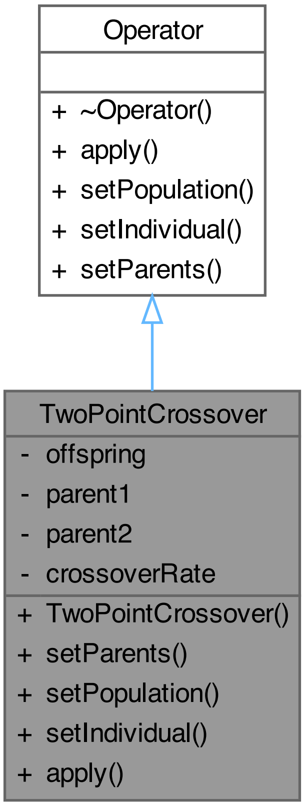
The documentation for this class was generated from the following file:

- include/utils/tqdm.hpp

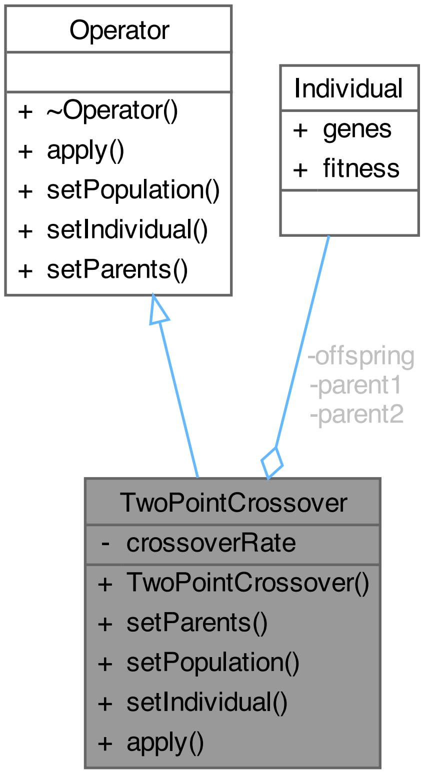
12.24 TwoPointCrossover Class Reference

```
#include <TwoPointCrossover.h>
```

Inheritance diagram for TwoPointCrossover:



Collaboration diagram for TwoPointCrossover:



Public Member Functions

- TwoPointCrossover (const Algorithm_Parameters &params)

    Constructs a TwoPointCrossover object with specified algorithm parameters.
- void setParents (Individual &offspring, const Individual &parent1, const Individual &parent2) override

    Sets the parents and offspring for the crossover operation.
- void setPopulation (const std::vector< Individual > &population, std::vector< Individual > &selected)

    Sets the population and selected vectors for the selection operation.
- void setIndividual (Individual &individual)

    Sets the individual for the operator.
- void apply (std::mt19937 &generator) override

    Performs the two-point crossover operation.

Public Member Functions inherited from Operator

- virtual ∼Operator ()=default

    Virtual destructor for the Operator class.

Private Attributes

- Individual ∗ offspring
- const Individual ∗ parent1

- const Individual * parent2
- double crossoverRate

12.24.1 Constructor & Destructor Documentation

12.24.1.1 TwoPointCrossover()

```
TwoPointCrossover::TwoPointCrossover (
            const Algorithm_Parameters & params )
```

Constructs a TwoPointCrossover object with specified algorithm parameters.

Parameters

params	The algorithm parameters including the crossover rate.

12.24.2 Member Function Documentation

12.24.2.1 apply()

```
void TwoPointCrossover::apply (
            std::mt19937 & generator ) [override], [virtual]
```

Performs the two-point crossover operation.

This method performs a two-point crossover on the two parent individuals to generate an offspring individual. Two crossover points are chosen randomly, and the genetic material from the two parents is combined to create the offspring.

Parameters

generator	The random number generator used to determine the crossover points.

Implements Operator.

12.24.2.2 setIndividual()

```
void TwoPointCrossover::setIndividual (
            Individual & individual ) [inline], [virtual]
```

Sets the individual for the operator.

This method is not used by the TwoPointCrossover operator.

Parameters

individual	The individual to be set.

Implements Operator.

12.24.2.3 setParents()

```
void TwoPointCrossover::setParents (
            Individual & offspring,
            const Individual & parent1,
            const Individual & parent2 ) [override], [virtual]
```

Sets the parents and offspring for the crossover operation.

Parameters

offspring	The offspring individual to be generated.
parent1	The first parent individual.
parent2	The second parent individual.

Implements Operator.

12.24.2.4 setPopulation()

```
void TwoPointCrossover::setPopulation (
            const std::vector< Individual > & population,
            std::vector< Individual > & selected )  [inline], [virtual]
```

Sets the population and selected vectors for the selection operation.
This method is not used by the TwoPointCrossover operator.

Parameters

population	The vector of individuals in the population.
selected	The vector of selected individuals.

Implements Operator.

12.24.3 Field Documentation

12.24.3.1 crossoverRate

```
double TwoPointCrossover::crossoverRate  [private]
```

The crossover rate used to determine if crossover should occur

12.24.3.2 offspring

```
Individual* TwoPointCrossover::offspring  [private]
```

Pointer to the offspring individual

12.24.3.3 parent1

```
const Individual* TwoPointCrossover::parent1  [private]
```

Pointer to the first parent individual

12.24.3.4 parent2

```
const Individual* TwoPointCrossover::parent2  [private]
```

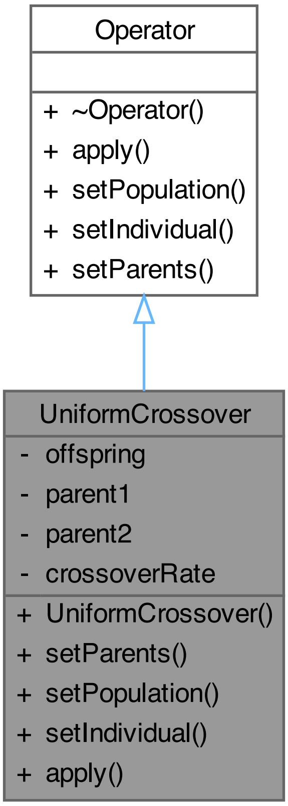
Pointer to the second parent individual
The documentation for this class was generated from the following files:

- include/operators/Crossover/TwoPointCrossover.h
- src/operators/Crossover/TwoPointCrossover.cpp

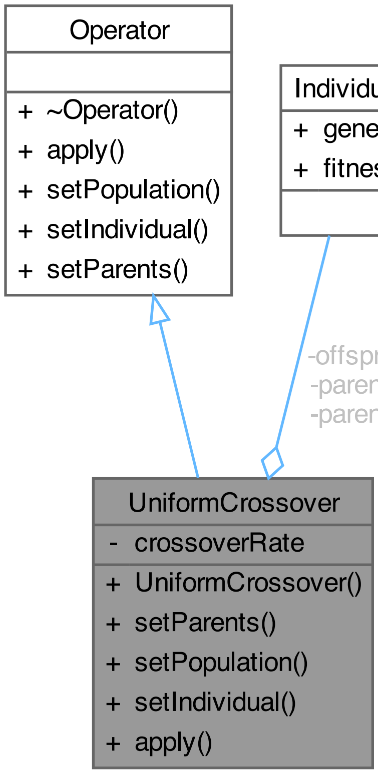
12.25 UniformCrossover Class Reference

```
#include <UniformCrossover.h>
```

Inheritance diagram for UniformCrossover:



Collaboration diagram for UniformCrossover:



Public Member Functions

- UniformCrossover (const Algorithm_Parameters &params)

  Constructs a UniformCrossover object with specified algorithm parameters.
- void setParents (Individual &offspring, const Individual &parent1, const Individual &parent2) override

  Sets the parents and offspring for the crossover operation.
- void setPopulation (const std::vector< Individual > &population, std::vector< Individual > &selected)

  Sets the population and selected vectors for the selection operation.
- void setIndividual (Individual &individual)

  Sets the individual for the selection operation.
- void apply (std::mt19937 &generator) override

  Performs the uniform crossover operation.

Public Member Functions inherited from Operator

- virtual ~Operator ()=default

  Virtual destructor for the Operator class.

Private Attributes

- Individual * offspring
- const Individual * parent1

- const Individual * parent2
- double crossoverRate

12.25.1 Constructor & Destructor Documentation

12.25.1.1 UniformCrossover()

```
UniformCrossover::UniformCrossover (
            const Algorithm_Parameters & params )
```

Constructs a UniformCrossover object with specified algorithm parameters.

Parameters

params	The algorithm parameters including the crossover rate.

12.25.2 Member Function Documentation

12.25.2.1 apply()

```
void UniformCrossover::apply (
            std::mt19937 & generator ) [override], [virtual]
```

Performs the uniform crossover operation.

This method performs a uniform crossover on the two parent individuals to generate an offspring individual. Each gene in the offspring is chosen randomly from one of the two parents.

Parameters

generator	The random number generator used to determine gene selection.

Implements Operator.

12.25.2.2 setIndividual()

```
void UniformCrossover::setIndividual (
            Individual & individual ) [inline], [virtual]
```

Sets the individual for the selection operation.

This method is not used in the UniformCrossover class.

Parameters

individual	The individual to be set.

Implements Operator.

12.25.2.3 setParents()

```
void UniformCrossover::setParents (
            Individual & offspring,
            const Individual & parent1,
            const Individual & parent2 ) [override], [virtual]
```

Sets the parents and offspring for the crossover operation.

Parameters

offspring	The offspring individual to be generated.
parent1	The first parent individual.
parent2	The second parent individual.

Implements Operator.

12.25.2.4 setPopulation()

```
void UniformCrossover::setPopulation (
            const std::vector< Individual > & population,
            std::vector< Individual > & selected ) [inline], [virtual]
```

Sets the population and selected vectors for the selection operation.
This method is not used in the UniformCrossover class.

Parameters

population	The vector of individuals in the population.
selected	The vector of selected individuals.

Implements Operator.

12.25.3 Field Documentation

12.25.3.1 crossoverRate

```
double UniformCrossover::crossoverRate [private]
```

The crossover rate used to determine if crossover should occur

12.25.3.2 offspring

```
Individual* UniformCrossover::offspring [private]
```

Pointer to the offspring individual

12.25.3.3 parent1

```
const Individual* UniformCrossover::parent1 [private]
```

Pointer to the first parent individual

12.25.3.4 parent2

```
const Individual* UniformCrossover::parent2 [private]
```

Pointer to the second parent individual
The documentation for this class was generated from the following files:

- include/operators/Crossover/UniformCrossover.h
- src/operators/Crossover/UniformCrossover.cpp

Chapter 13

File Documentation

13.1 docs/CCircuit_Document_Analysis.md File Reference

13.2 docs/CSimulator_Document_and_Analysis.md File Reference

13.3 docs/GeneticAlgorithm_Design_Document.md File Reference

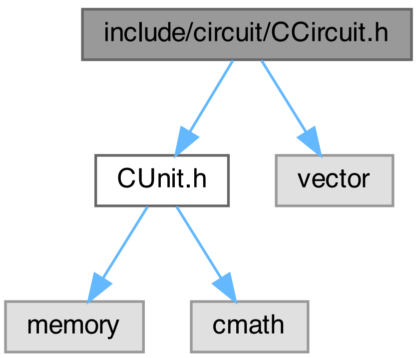
13.4 docs/GeneticAlgorithm_Params_Analysis.md File Reference

13.5 include/circuit/CCircuit.h File Reference

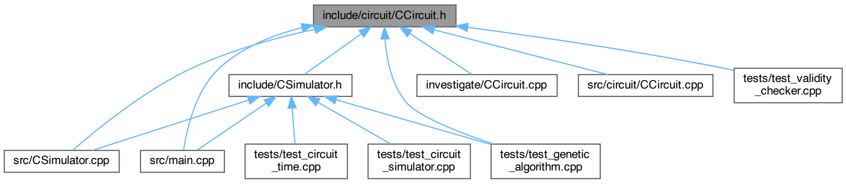
Defines the Circuit class for modeling and simulating a circuit system.

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

Include dependency graph for CCircuit.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- class Circuit

Macros

- #define CIRCUIT_MODELLING_CCIRCUIT_H

13.5.1 Detailed Description

Defines the Circuit class for modeling and simulating a circuit system.
The Circuit class provides methods to initialize, configure, and simulate a circuit with multiple units. It includes methods for checking validity, printing information, connecting units, initializing feed rates, checking convergence, and calculating flows. The Circuit class is designed to facilitate the modeling and simulation of circuits in various applications, including industrial processes and scientific research.

Date

Created on May 20, 2024

Authors

ACS Gerardium Rush - Pentlandite:

- Alex N Njeumi
- Geyu Ji
- Melissa Y S Sim
- Mingsheng Cai
- Peifeng Tan
- Yongwen Chen
- Zihan Li

13.5.2 Macro Definition Documentation

13.5.2.1 CIRCUIT_MODELLING_CCIRCUIT_H

```
#define CIRCUIT_MODELLING_CCIRCUIT_H
```

13.6 CCircuit.h

Go to the documentation of this file.

```
00001
00024 #pragma once
00025
00026 #ifndef CIRCUIT_MODELLING_CCIRCUIT_H
00027 #define CIRCUIT_MODELLING_CCIRCUIT_H
00028
00029 #include "CUnit.h"
```

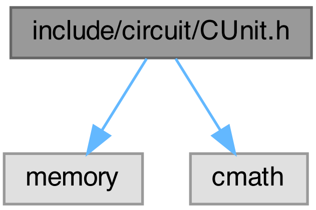
```
#include <vector>

class Circuit {
public:
    // Concentrates
    double concentrateG;
    double concentrateW;
    // Tailings
    double tailingsG;
    double tailingsW;
    // Feed rates
    double wasteFeed;
    double gerardiumFeed;
    Circuit(int num_units);

    ~Circuit();

    static bool Check_Validity(int vectorSize, int *circuitVector);

    void print_info() const;

    void connected(int *circuitVector);

    void initialize_feed_rates(double gerardium_ = 10, double waste_ = 90);

    void mark_units(int unitNum, std::vector<CUnit *> &units, bool check[3]);

    bool check_convergence(double threshold);

    void calculate_flows();

private:
    void reset_new_feeds();

    std::vector<CUnit *> units;
    int numUnits;
    static bool check[3];
    int feed;
};

#endif // CIRCUIT_MODELLING_CCIRCUIT_H
```

13.7 include/circuit/CUnit.h File Reference

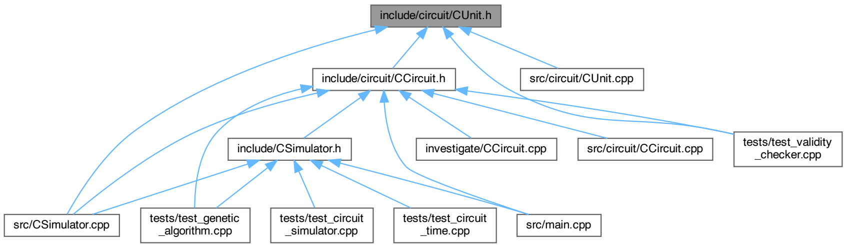
Defines the CUnit class for representing a unit within a circuit system.

`#include <memory>`
`#include <cmath>`

Include dependency graph for CUnit.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- class CUnit

Macros

- #define CIRCUIT_MODELLING_CUNIT_H

13.7.1 Detailed Description

Defines the CUnit class for representing a unit within a circuit system.
The CUnit class provides methods to manage and simulate individual units within a circuit. It includes methods for setting and getting various properties of the unit, such as feed rates, mark status, and connection pointers. This class is designed to facilitate the modeling and simulation of circuits in various applications, including industrial processes and scientific research.

Date

Created on May 20, 2024

Authors

ACS Gerardium Rush - Pentlandite:

- Alex N Njeumi
- Geyu Ji
- Melissa Y S Sim
- Mingsheng Cai
- Peifeng Tan
- Yongwen Chen
- Zihan Li

13.7.2 Macro Definition Documentation

13.7.2.1 CIRCUIT_MODELLING_CUNIT_H

```
#define CIRCUIT_MODELLING_CUNIT_H
```

13.8 CUnit.h

Go to the documentation of this file.

```

#pragma once

#ifndef CIRCUIT_MODELLING_CUNIT_H
```

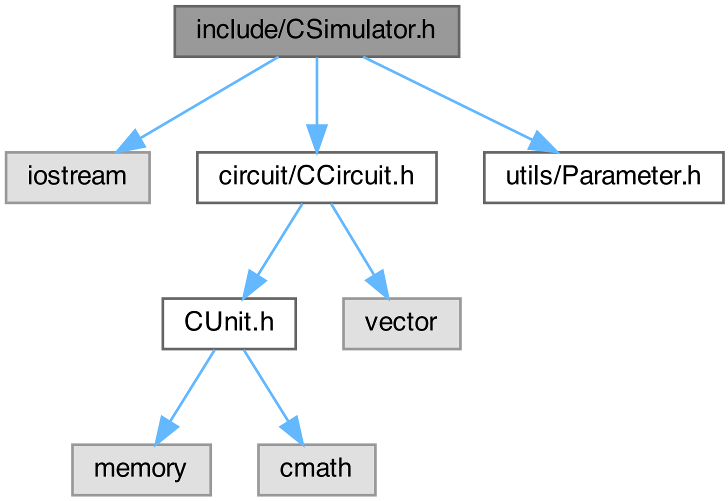
```
#define CIRCUIT_MODELLING_CUNIT_H

#include <memory> //
#include <cmath>

class CUnit {
public:
    CUnit();

    CUnit(int concNum, int interNum, int tailsNum);

    int concNum;
    int interNum;
    int tailsNum;
    bool isMarked() const;

    void setMarked(bool mark);

    double getFeedGerardium() const;

    void setFeedGerardium(double feedGerardium);

    double getFeedWaste() const;

    void setFeedWaste(double feedWaste);

    double getNewFeedGerardium() const;

    void setNewFeedGerardium(double newFeedGerardium);

    double getNewFeedWaste() const;

    void setNewFeedWaste(double newFeedWaste);

    double getDifferenceG() const;

    double getDifferenceW() const;

    void setDifference();


    CUnit *getConcPtr() const;

    void setConcPtr(CUnit *ptr);

    CUnit *getInterPtr() const;

    void setInterPtr(CUnit *ptr);

    CUnit *getTailsPtr() const;

    void setTailsPtr(CUnit *ptr);

private:
    bool mark;
    double feedGerardium;
    double feedWaste;
    double newFeedGerardium;
    double newFeedWaste;
    double differenceG;
    double differenceW;
    CUnit *concPtr;
    CUnit *interPtr;
    CUnit *tailsPtr;
};

#endif // CIRCUIT_MODELLING_CUNIT_H
```

13.9 include/CSimulator.h File Reference

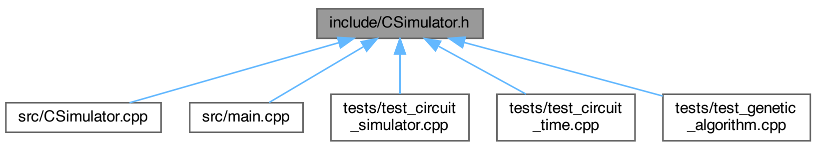
Defines the functions and structures for simulating and evaluating circuits in a circuit modeling framework.

```
#include <iostream>
#include "circuit/CCircuit.h"
#include "utils/Parameter.h"
```

Include dependency graph for CSimulator.h:



This graph shows which files directly or indirectly include this file:



Macros

- #define CIRCUIT_MODELLING_CSIMULATOR_H

Functions

- double Evaluate_Circuit (int vectorSize, int ∗circuitVector, struct CircuitParameters parameters)

  Evaluates the performance of a circuit.
- double Evaluate_Circuit (int vectorSize, int ∗circuitVector, double &Recovery, double &Grade, struct CircuitParameters parameters)

  Evaluates the performance, recovery, and grade of a circuit.
- double performance (double concentrateG, double concentrateW)

  Computes the performance metric of a circuit.
- double grade (double concentrateG, double concentrateW)

  Computes the grade of a circuit.
- double recovery (double concentrateG, double gerardiumFeed)

  Computes the recovery of a circuit.

13.9.1 Detailed Description

Defines the functions and structures for simulating and evaluating circuits in a circuit modeling framework.
The CSimulator module provides functions for evaluating the performance, grade, and recovery of circuits in a circuit modeling framework. It includes definitions for circuit parameters and multiple overloads for evaluating circuits with or without additional parameters.

Date

Created on May 20, 2024

Authors

ACS Gerardium Rush - Pentlandite:

- Alex N Njeumi
- Geyu Ji
- Melissa Y S Sim
- Mingsheng Cai
- Peifeng Tan
- Yongwen Chen
- Zihan Li

13.9.2 Macro Definition Documentation

13.9.2.1 CIRCUIT_MODELLING_CSIMULATOR_H

```
#define CIRCUIT_MODELLING_CSIMULATOR_H
```

13.9.3 Function Documentation

13.9.3.1 Evaluate_Circuit() [1/2]

```
double Evaluate_Circuit (
            int vectorSize,
            int * circuitVector,
            double & Recovery,
            double & Grade,
            struct CircuitParameters parameters )
```

Evaluates the performance, recovery, and grade of a circuit.
This function evaluates the performance, recovery, and grade of a circuit based on the given circuit vector and additional circuit parameters.

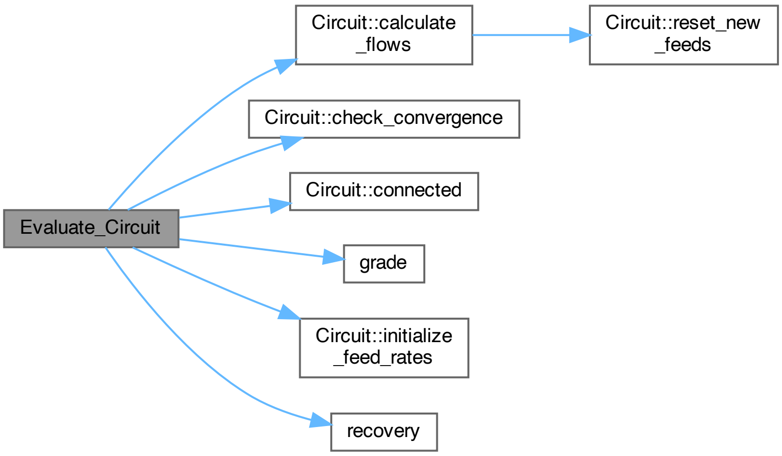
Parameters

vectorSize	The size of the circuit vector.
circuitVector	Pointer to the array representing the circuit configuration.
Recovery	Reference to a double variable to store the recovery value.
Grade	Reference to a double variable to store the grade value.
parameters	Additional parameters for the circuit simulation.

Returns

The performance metric of the circuit.

Here is the call graph for this function:



13.9.3.2 Evaluate_Circuit() [2/2]

```
double Evaluate_Circuit (
            int vectorSize,
            int * circuitVector,
            struct CircuitParameters parameters )
```

Evaluates the performance of a circuit.

This function evaluates the performance of a circuit based on the given circuit vector and additional circuit parameters.

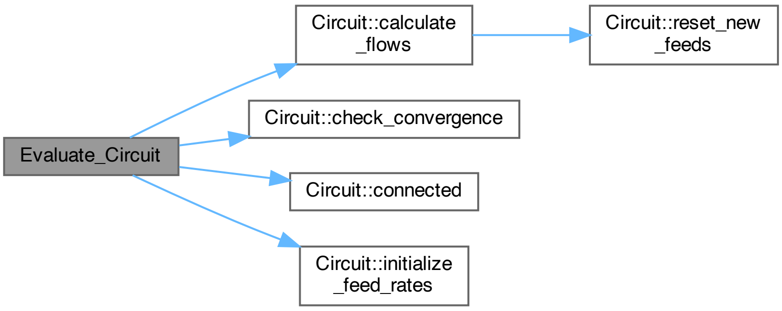
Parameters

vectorSize	The size of the circuit vector.
circuitVector	Pointer to the array representing the circuit configuration.
parameters	Additional parameters for the circuit simulation.

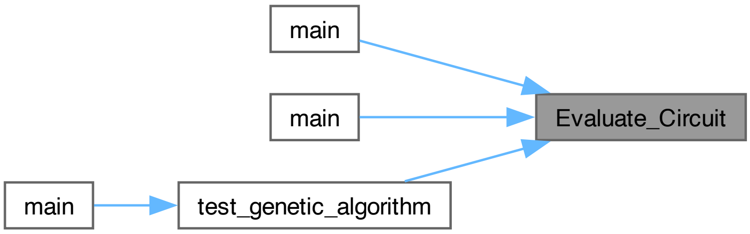
Returns

The performance metric of the circuit.

Here is the call graph for this function:



Here is the caller graph for this function:



13.9.3.3 grade()

```
double grade (
            double concentrateG,
            double concentrateW )
```

Computes the grade of a circuit.

This function computes the grade of a circuit based on the given concentrate gerardium and waste values.

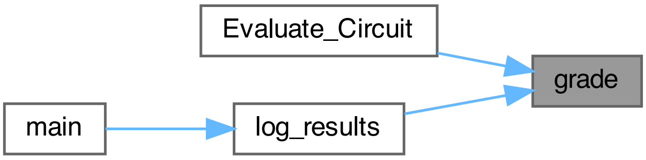
Parameters

concentrateG	The amount of gerardium in the concentrate.
concentrateW	The amount of waste in the concentrate.

Returns

The grade of the circuit.

Here is the caller graph for this function:



13.9.3.4 performance()

```
double performance (
            double concentrateG,
            double concentrateW )
```

Computes the performance metric of a circuit.
This function computes the performance metric of a circuit based on the given concentrate gerardium and waste values.

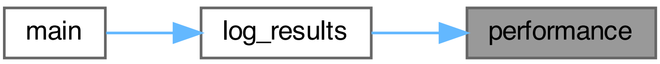
Parameters

concentrateG	The amount of gerardium in the concentrate.
concentrateW	The amount of waste in the concentrate.

Returns

The performance metric of the circuit.

Here is the caller graph for this function:



13.9.3.5 recovery()

```
double recovery (
            double concentrateG,
            double gerardiumFeed )
```

Computes the recovery of a circuit.
This function computes the recovery of a circuit based on the given concentrate gerardium and gerardium feed values.

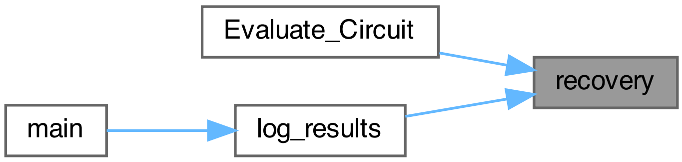
Parameters

concentrateG	The amount of gerardium in the concentrate.
gerardiumFeed	The amount of gerardium fed into the circuit.

Returns

The recovery of the circuit.

Here is the caller graph for this function:



13.10 CSimulator.h

Go to the documentation of this file.

```
#pragma once

#ifndef CIRCUIT_MODELLING_CSIMULATOR_H
#define CIRCUIT_MODELLING_CSIMULATOR_H

#pragma once

#include <iostream>
#include "circuit/CCircuit.h"
#include "utils/Parameter.h"

double Evaluate_Circuit(int vectorSize, int *circuitVector, struct CircuitParameters parameters);

double Evaluate_Circuit(int vectorSize, int *circuitVector, double& Recovery, double& Grade, struct
      CircuitParameters parameters);

double performance(double concentrateG, double concentrateW);

double grade(double concentrateG, double concentrateW);

double recovery(double concentrateG, double gerardiumFeed);

#endif //CIRCUIT_MODELLING_CSIMULATOR_H
```

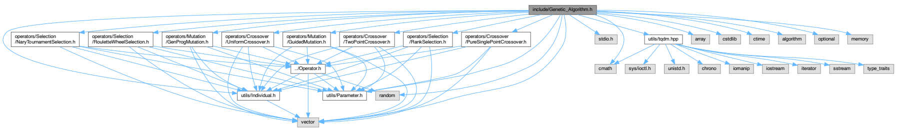
13.11 include/Genetic_Algorithm.h File Reference

Defines the functions and structures for implementing a genetic algorithm.

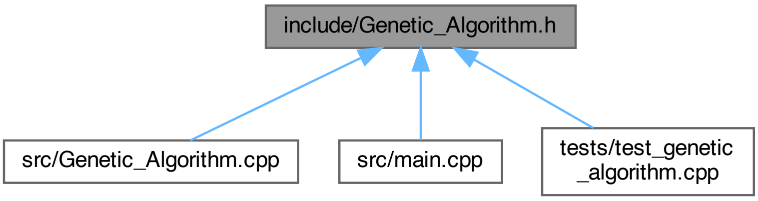
```
#include <vector>
#include <stdio.h>
#include <cmath>
#include <array>
#include <cstdlib>
#include <ctime>
#include <algorithm>
#include <optional>
#include <random>
```

```
#include <memory>
#include "utils/Individual.h"
#include "utils/Parameter.h"
#include "utils/tqdm.hpp"
#include "operators/Selection/NaryTournamentSelection.h"
#include "operators/Selection/RankSelection.h"
#include "operators/Selection/RouletteWheelSelection.h"
#include "operators/Mutation/GuidedMutation.h"
#include "operators/Mutation/GenProgMutation.h"
#include "operators/Crossover/PureSinglePointCrossover.h"
#include "operators/Crossover/TwoPointCrossover.h"
#include "operators/Crossover/UniformCrossover.h"
```

Include dependency graph for Genetic_Algorithm.h:



This graph shows which files directly or indirectly include this file:



Macros

- #define GENETIC_ALGORITHM_H

Functions

- void initializeRandomSeed (const Algorithm_Parameters &params, std::vector< std::mt19937 > &generators)

  Initializes the random seed based on algorithm parameters.
- std::vector< Individual > initializePopulation (const Algorithm_Parameters &params, int vectorSize, bool(&validity)(int, int *), std::vector< std::mt19937 > &generators)

  Initializes the population for the genetic algorithm.
- void elitism (const std::vector< Individual > &population, std::vector< Individual > &newPopulation, const Algorithm_Parameters &params)

  Apply elitism to preserve the best individuals in a new population.
- double acceptanceProbability (double currentEnergy, double newEnergy, double temperature)

  Calculate the acceptance probability for a new state in Simulated Annealing.
- bool applySimulatedAnnealing (Individual &offspring, Individual &parent1, Individual &parent2, int vector_size, double(&func)(int, int *, struct CircuitParameters), bool(&validity)(int, int *), double &Temp, const Algorithm_Parameters &params, const CircuitParameters c_params, std::mt19937 &generator)

  Applies simulated annealing technique to decide whether to accept an offspring in a genetic algorithm.

- int optimize (int vector_size, int ∗vector, double(&func)(int, int ∗, struct CircuitParameters), bool(&validity)(int, int ∗), struct Algorithm_Parameters params, struct CircuitParameters c_params)

  Optimizes a solution using the genetic algorithm.

13.11.1 Detailed Description

Defines the functions and structures for implementing a genetic algorithm.
The Genetic_Algorithm module provides functions for initializing populations, setting random seeds, and optimizing solutions using a genetic algorithm framework. This includes selection, crossover, and mutation operators.

Date

Created on May 20, 2024

Authors

ACS Gerardium Rush - Pentlandite:

- Alex N Njeumi
- Geyu Ji
- Melissa Y S Sim
- Mingsheng Cai
- Peifeng Tan
- Yongwen Chen
- Zihan Li

13.11.2 Macro Definition Documentation

13.11.2.1 GENETIC_ALGORITHM_H

```
#define GENETIC_ALGORITHM_H
```

13.11.3 Function Documentation

13.11.3.1 acceptanceProbability()

```
double acceptanceProbability (
            double currentEnergy,
            double newEnergy,
            double temperature )
```

Calculate the acceptance probability for a new state in Simulated Annealing.
This function calculates the probability with which a new state should be accepted over the current state, based on their respective energies and the current temperature of the system. If the new energy is better (higher value), the function returns 1.0, meaning the new state is always accepted. Otherwise, it returns a value calculated using the Boltzmann probability distribution which considers both the difference in energy and the current temperature.

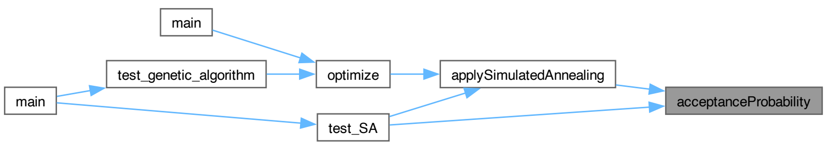
Parameters

currentEnergy	The energy of the current state.
newEnergy	The energy of the new state being considered.
temperature	The current temperature of the system (controls the probability threshold).

Returns

double The probability of accepting the new state.

Here is the caller graph for this function:



13.11.3.2 applySimulatedAnnealing()

```
bool applySimulatedAnnealing (
            Individual & offspring,
            Individual & parent1,
            Individual & parent2,
            int vector_size,
            double(&)(int, int *, struct CircuitParameters) func,
            bool(&)(int, int *) validity,
            double & Temp,
            const Algorithm_Parameters & params,
            const CircuitParameters c_params,
            std::mt19937 & generator )
```

Applies simulated annealing technique to decide whether to accept an offspring in a genetic algorithm.
This function performs the simulated annealing decision process on a newly generated offspring. It compares the offspring's fitness to the average fitness of its parents using the simulated annealing acceptance probability, which depends on the current temperature of the system. The function also handles the cooling of the system temperature and validates the offspring using a provided validity function.

Parameters

offspring	A reference to the offspring individual whose acceptance is being determined.
parent1	A reference to the first parent of the offspring.
parent2	A reference to the second parent of the offspring.
vector_size	The size of the individual's gene vector.
func	A function that calculates the fitness of an individual based on its genes.
validity	A function that checks the validity of an individual's gene configuration.
Temp	A reference to the current temperature used for the simulated annealing process.
params	A structure containing parameters relevant to the annealing process, such as the temperature decrement.
c_params	The circuit parameters used in the fitness evaluation function.
generator	A random number generator used for probabilistic decisions.

Returns

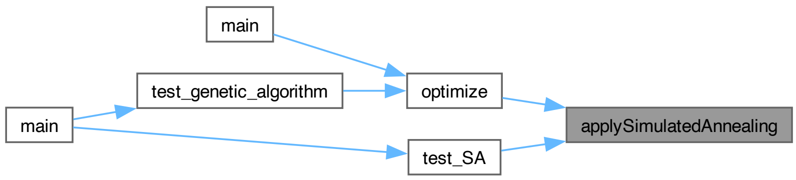
bool Returns true if the offspring is accepted, false otherwise. The function can also modify the offspring directly, setting it to one of the parents if the temperature is low and the acceptance criteria are not met.

Note

The function directly modifies the temperature, decrementing it according to `deltT` from `params`. If the offspring is invalid (as determined by the `validity` function), it is immediately rejected and assigned a fitness value of INVALID_FITNESS.

Here is the call graph for this function:

Here is the caller graph for this function:



13.11.3.3 elitism()

```
void elitism (
            const std::vector< Individual > & population,
            std::vector< Individual > & newPopulation,
            const Algorithm_Parameters & params )
```

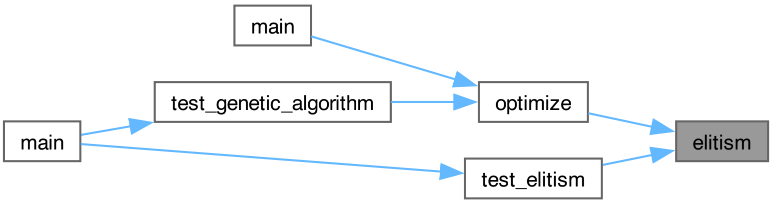
Apply elitism to preserve the best individuals in a new population.

This function sorts the given population in descending order based on their fitness, then selectively copies the top percentage of individuals as specified by the elitePercentage in the Algorithm_Parameters. This is used to ensure that the best individuals are preserved for the next generation, promoting genetic diversity and preventing loss of the best found solutions.

Parameters

population	The current generation of individuals (const reference).
newPopulation	The next generation of individuals where elites will be added (reference).
params	The parameters of the algorithm including population size and elite percentage.

Here is the caller graph for this function:



13.11.3.4 initializePopulation()

```
std::vector< Individual > initializePopulation (
            const Algorithm_Parameters & params,
            int vectorSize,
            bool(&)(int, int *) validity,
            std::vector< std::mt19937 > & generators )
```

Initializes the population for the genetic algorithm.

This function initializes the population for the genetic algorithm, ensuring each individual meets the validity criteria specified by the user.

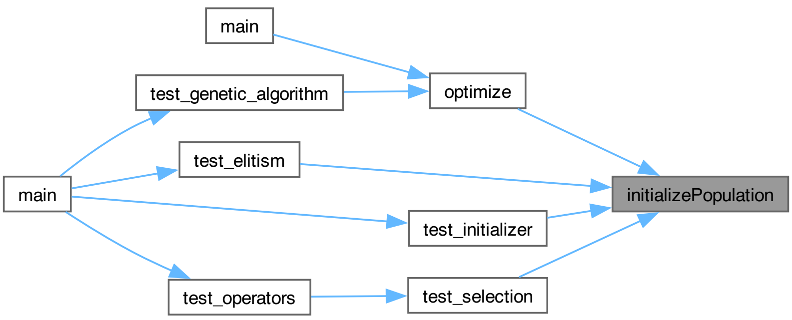
Parameters

params	The algorithm parameters containing population size and other configurations.
vectorSize	The size of the gene vector for each individual.
validity	A function pointer to the validity checking function.
generators	A vector of random number generators for each thread.

Returns

A vector of individuals representing the initial population.

Here is the caller graph for this function:



13.11.3.5 initializeRandomSeed()

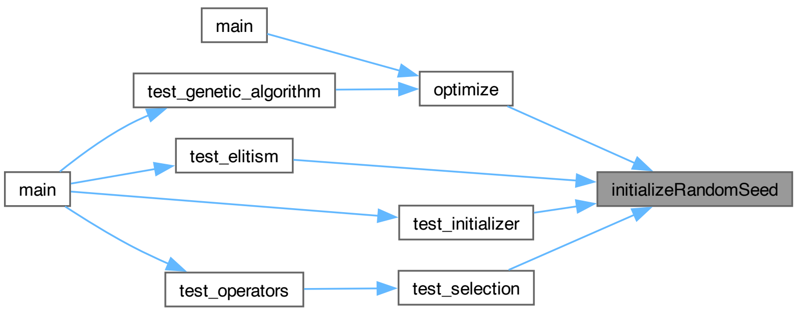
```
void initializeRandomSeed (
            const Algorithm_Parameters & params,
            std::vector< std::mt19937 > & generators )
```

Initializes the random seed based on algorithm parameters.
This function initializes the random seed using the specified seed in the algorithm parameters. If the seed is zero, it uses the current time as the seed.

Parameters

params	The algorithm parameters containing the random seed.
generators	A vector of random number generators for each thread.

Here is the caller graph for this function:



13.11.3.6 optimize()

```
int optimize (
            int vector_size,
            int * vector,
            double(&)(int, int *, struct CircuitParameters) func,
            bool(&)(int, int *) validity,
            struct Algorithm_Parameters params,
            struct CircuitParameters c_params )
```

Optimizes a solution using the genetic algorithm.
This function performs optimization using a genetic algorithm. It initializes the population, assesses the fitness of each individual, and iterates through generations applying selection, crossover, and mutation operators.

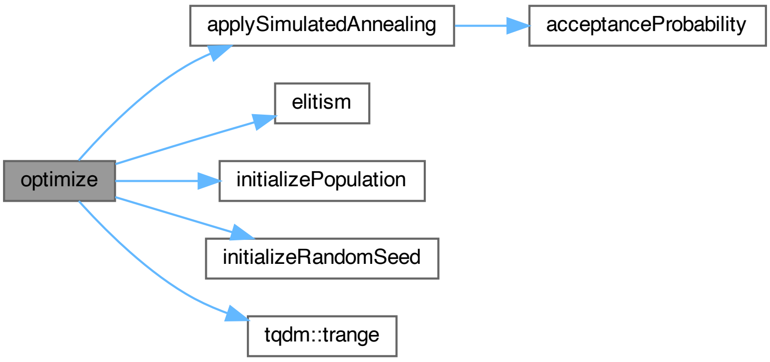
Parameters

vector_size	The size of the gene vector.
vector	Pointer to the array representing the solution vector.
func	A function pointer to the fitness evaluation function.
validity	A function pointer to the validity checking function.
parameters	The algorithm parameters containing various configurations.

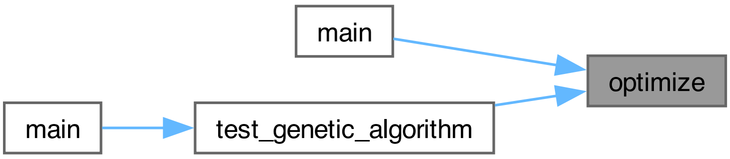
Returns

An integer indicating the success (0) or failure (-1) of the optimization.

Here is the call graph for this function:



Here is the caller graph for this function:



13.12 Genetic_Algorithm.h

Go to the documentation of this file.

```

#pragma once

#ifndef GENETIC_ALGORITHM_H
#define GENETIC_ALGORITHM_H

#include <vector>
#include <stdio.h>
#include <cmath>
#include <array>
#include <cstdlib>
#include <ctime>
#include <algorithm>
#include <optional>
#include <random>
#include <memory>

#include "utils/Individual.h"
#include "utils/Parameter.h"
#include "utils/tqdm.hpp"
#include "operators/Selection/NaryTournamentSelection.h"
#include "operators/Selection/RankSelection.h"
#include "operators/Selection/RouletteWheelSelection.h"
#include "operators/Mutation/GuidedMutation.h"
#include "operators/Mutation/GenProgMutation.h"
```

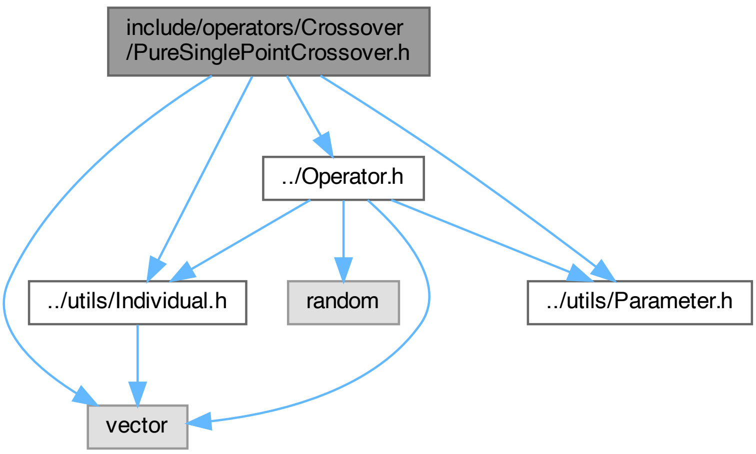
```
#include "operators/Crossover/PureSinglePointCrossover.h"
#include "operators/Crossover/TwoPointCrossover.h"
#include "operators/Crossover/UniformCrossover.h"

void initializeRandomSeed(const Algorithm_Parameters &params, std::vector<std::mt19937> &generators);

std::vector<Individual>
initializePopulation(const Algorithm_Parameters &params, int vectorSize, bool (&validity)(int, int *), std::vector<std::mt19937> &generators);

void elitism(const std::vector<Individual> &population, std::vector<Individual> &newPopulation,
             const Algorithm_Parameters &params);


double acceptanceProbability(double currentEnergy, double newEnergy, double temperature);

bool applySimulatedAnnealing(Individual &offspring, Individual &parent1, Individual &parent2, int vector_size,
                             double (&func)(int, int *, struct CircuitParameters), bool (&validity)(int, int *),
                             double &Temp, const Algorithm_Parameters &params, const CircuitParameters c_params,
                             std::mt19937 &generator);

int optimize(int vector_size, int *vector,
             double (&func)(int, int *, struct CircuitParameters),
             bool (&validity)(int, int *),
             struct Algorithm_Parameters params,
             struct CircuitParameters c_params);

#endif //GENETIC_ALGORITHM_H
```

13.13 include/operators/Crossover/PureSinglePointCrossover.h File Reference

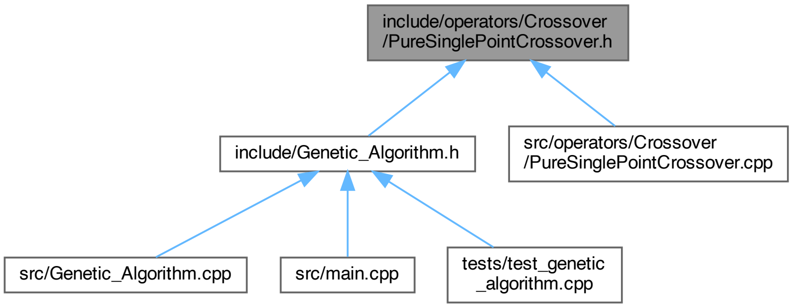
Defines the PureSinglePointCrossover class for performing single-point crossover in a genetic algorithm.

```
#include <vector>
#include "../Operator.h"
#include "../../utils/Individual.h"
#include "../../utils/Parameter.h"
```

Include dependency graph for PureSinglePointCrossover.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- class PureSinglePointCrossover

Macros

- #define GENETIC_ALGORITHM_PURESINGLEPOINTCROSSOVER_H

13.13.1 Detailed Description

Defines the PureSinglePointCrossover class for performing single-point crossover in a genetic algorithm.
The PureSinglePointCrossover class provides methods to perform a single-point crossover on two parent individuals to generate an offspring individual. This class is designed to be used in genetic algorithms to combine genetic material from two parents to create new individuals, which can then be used in subsequent generations of the algorithm.

Date

Created on May 20, 2024

Authors

ACS Gerardium Rush - Pentlandite:

- Alex N Njeumi
- Geyu Ji
- Melissa Y S Sim
- Mingsheng Cai
- Peifeng Tan
- Yongwen Chen
- Zihan Li

13.13.2 Macro Definition Documentation

13.13.2.1 GENETIC_ALGORITHM_PURESINGLEPOINTCROSSOVER_H

```
#define GENETIC_ALGORITHM_PURESINGLEPOINTCROSSOVER_H
```

13.14 PureSinglePointCrossover.h

Go to the documentation of this file.

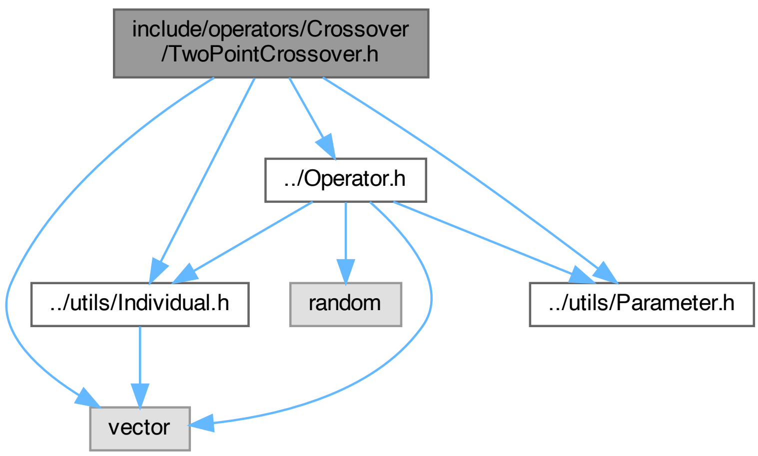
```
#pragma once

#ifndef GENETIC_ALGORITHM_PURESINGLEPOINTCROSSOVER_H
#define GENETIC_ALGORITHM_PURESINGLEPOINTCROSSOVER_H

#include <vector>
#include "../Operator.h"
#include "../../utils/Individual.h"
#include "../../utils/Parameter.h"

class PureSinglePointCrossover : public Operator {
private:
    Individual *offspring;
    const Individual *parent1;
    const Individual *parent2;
    double crossoverRate;
public:
    PureSinglePointCrossover(const Algorithm_Parameters &params);

    void setParents(Individual &offspring, const Individual &parent1, const Individual &parent2) override;

    void setPopulation(const std::vector<Individual> &population, std::vector<Individual> &selected) {};

    void setIndividual(Individual &individual) {};

    void apply(std::mt19937 &generator) override;
};

#endif //GENETIC_ALGORITHM_PURESINGLEPOINTCROSSOVER_H
```

13.15 include/operators/Crossover/TwoPointCrossover.h File Reference

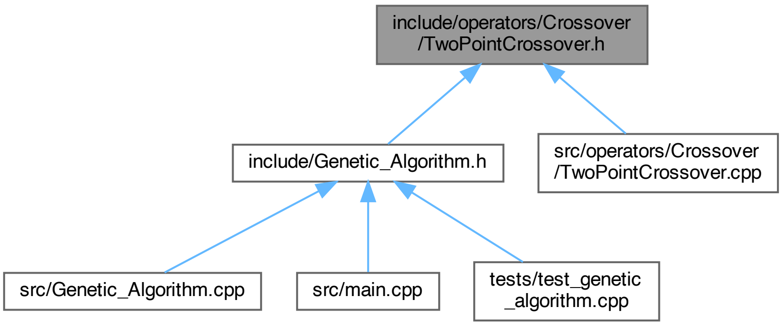
Defines the TwoPointCrossover class for performing two-point crossover in a genetic algorithm.

```
#include <vector>
#include "../Operator.h"
#include "../../utils/Individual.h"
#include "../../utils/Parameter.h"
```

Include dependency graph for TwoPointCrossover.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- class TwoPointCrossover

Macros

- #define GENETIC_ALGORITHM_TWOPOINTCROSSOVER_H

13.15.1 Detailed Description

Defines the TwoPointCrossover class for performing two-point crossover in a genetic algorithm.
The TwoPointCrossover class provides methods to perform a two-point crossover on two parent individuals to generate an offspring individual. This class is designed to be used in genetic algorithms to combine genetic material from two parents to create new individuals, which can then be used in subsequent generations of the algorithm.

Date

Created on May 20, 2024

Authors

ACS Gerardium Rush - Pentlandite:

- Alex N Njeumi
- Geyu Ji
- Melissa Y S Sim
- Mingsheng Cai
- Peifeng Tan
- Yongwen Chen
- Zihan Li

13.15.2 Macro Definition Documentation

13.15.2.1 GENETIC_ALGORITHM_TWOPOINTCROSSOVER_H

```
#define GENETIC_ALGORITHM_TWOPOINTCROSSOVER_H
```

13.16 TwoPointCrossover.h

Go to the documentation of this file.

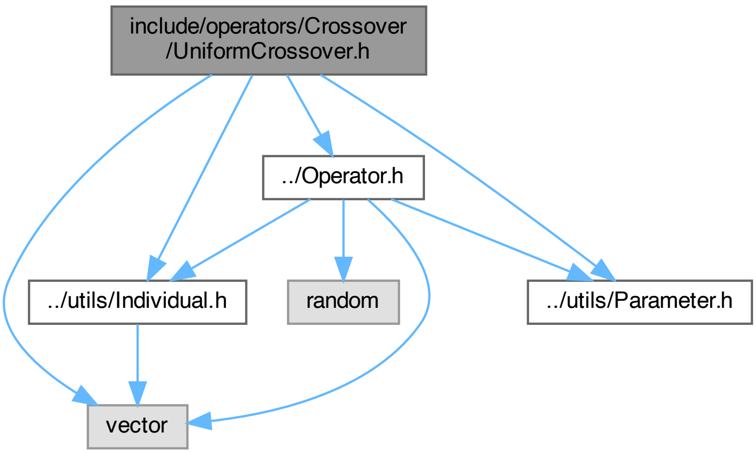
```
#pragma once

#ifndef GENETIC_ALGORITHM_TWOPOINTCROSSOVER_H
#define GENETIC_ALGORITHM_TWOPOINTCROSSOVER_H

#include <vector>
#include "../Operator.h"
#include "../../utils/Individual.h"
#include "../../utils/Parameter.h"

class TwoPointCrossover : public Operator {
private:
    Individual *offspring;
    const Individual *parent1;
    const Individual *parent2;
    double crossoverRate;
public:
    TwoPointCrossover(const Algorithm_Parameters &params);

    void setParents(Individual &offspring, const Individual &parent1, const Individual &parent2)
override;

    void setPopulation(const std::vector<Individual> &population, std::vector<Individual> &selected)
{};

    void setIndividual(Individual &individual) {};

    void apply(std::mt19937 &generator) override;
};

#endif //GENETIC_ALGORITHM_TWOPOINTCROSSOVER_H
```

13.17 include/operators/Crossover/UniformCrossover.h File Reference

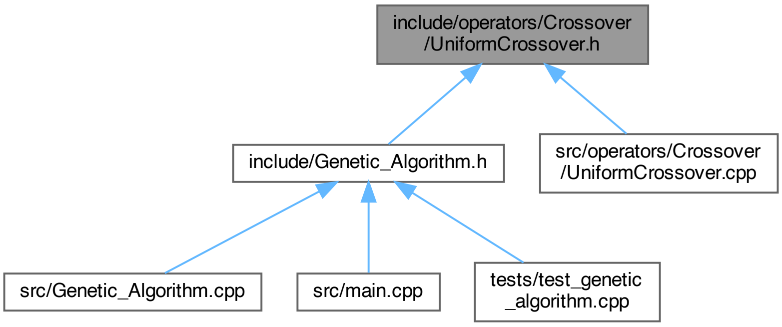
Defines the UniformCrossover class for performing uniform crossover in a genetic algorithm.

```
#include <vector>
#include "../Operator.h"
#include "../../utils/Individual.h"
#include "../../utils/Parameter.h"
```

Include dependency graph for UniformCrossover.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- class UniformCrossover

Macros

- #define GENETIC_ALGORITHM_UNIFORMCROSSOVER_H

13.17.1 Detailed Description

Defines the UniformCrossover class for performing uniform crossover in a genetic algorithm.
The UniformCrossover class provides methods to perform a uniform crossover on two parent individuals to generate an offspring individual. This class is designed to be used in genetic algorithms to combine genetic material from two parents to create new individuals, which can then be used in subsequent generations of the algorithm.

Date

Created on May 20, 2024

Authors

ACS Gerardium Rush - Pentlandite:

- Alex N Njeumi
- Geyu Ji
- Melissa Y S Sim
- Mingsheng Cai
- Peifeng Tan
- Yongwen Chen
- Zihan Li

13.17.2 Macro Definition Documentation

13.17.2.1 GENETIC_ALGORITHM_UNIFORMCROSSOVER_H

```
#define GENETIC_ALGORITHM_UNIFORMCROSSOVER_H
```

13.18 UniformCrossover.h

Go to the documentation of this file.

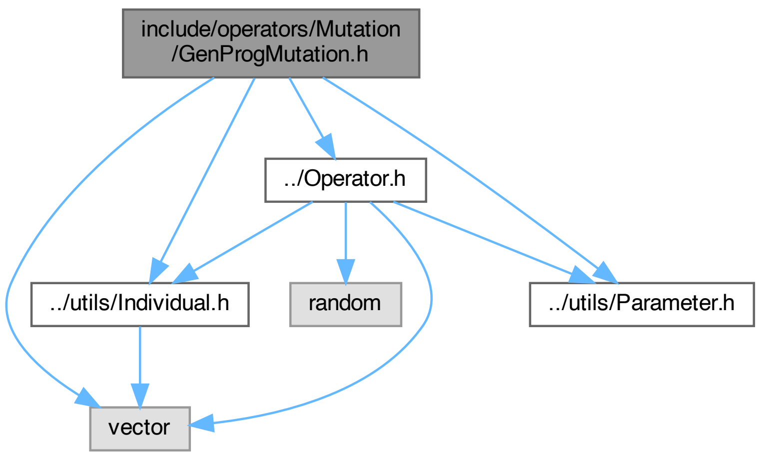
```
#pragma once

#ifndef GENETIC_ALGORITHM_UNIFORMCROSSOVER_H
#define GENETIC_ALGORITHM_UNIFORMCROSSOVER_H

#include <vector>
#include "../Operator.h"
#include "../../utils/Individual.h"
#include "../../utils/Parameter.h"

class UniformCrossover : public Operator {
private:
    Individual *offspring;
    const Individual *parent1;
    const Individual *parent2;
    double crossoverRate;
public:
    UniformCrossover(const Algorithm_Parameters &params);

    void setParents(Individual &offspring, const Individual &parent1, const Individual &parent2)
override;

    void setPopulation(const std::vector<Individual> &population, std::vector<Individual> &selected)
{};

    void setIndividual(Individual &individual) {};

    void apply(std::mt19937 &generator) override;
};

#endif //GENETIC_ALGORITHM_UNIFORMCROSSOVER_H
```

13.19 include/operators/Mutation/GenProgMutation.h File Reference

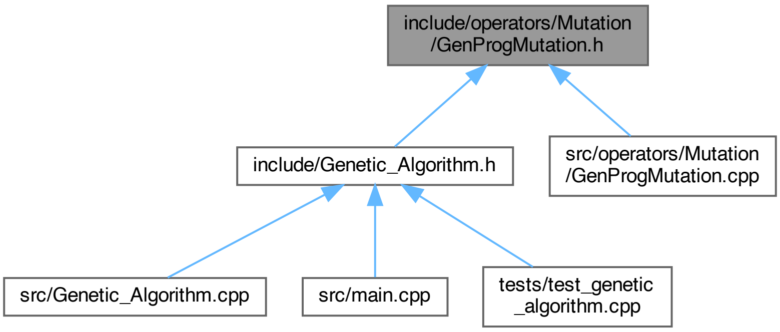
Defines the GenProgMutation class for performing mutation operations in a genetic programming algorithm.

```
#include <vector>
#include "../Operator.h"
#include "../../utils/Individual.h"
#include "../../utils/Parameter.h"
```

Include dependency graph for GenProgMutation.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- class GenProgMutation

Macros

- #define GENETIC_ALGORITHM_GENPROGMUTATION_H

13.19.1 Detailed Description

Defines the GenProgMutation class for performing mutation operations in a genetic programming algorithm.
The GenProgMutation class provides methods to perform mutation operations on individuals in a genetic programming algorithm. This class is designed to introduce variations into the population by modifying individuals' genetic material, which can then be used in subsequent generations of the algorithm.

Date

Created on May 20, 2024

Authors

ACS Gerardium Rush - Pentlandite:

- Alex N Njeumi
- Geyu Ji
- Melissa Y S Sim
- Mingsheng Cai
- Peifeng Tan
- Yongwen Chen
- Zihan Li

13.19.2 Macro Definition Documentation

13.19.2.1 GENETIC_ALGORITHM_GENPROGMUTATION_H

```
#define GENETIC_ALGORITHM_GENPROGMUTATION_H
```

13.20 GenProgMutation.h

Go to the documentation of this file.

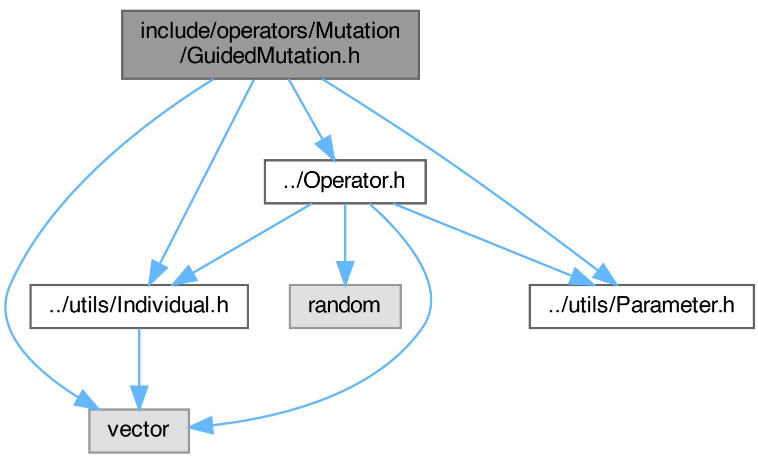
```
#pragma once

#ifndef GENETIC_ALGORITHM_GENPROGMUTATION_H
#define GENETIC_ALGORITHM_GENPROGMUTATION_H

#include <vector>
#include "../Operator.h"
#include "../../utils/Individual.h"
#include "../../utils/Parameter.h"

class GenProgMutation : public Operator {
public:
    GenProgMutation(int vectorSize, const Algorithm_Parameters &params);

    void setIndividual(Individual &individual) override;

    void setPopulation(const std::vector<Individual> &population, std::vector<Individual> &selected)
    {};

    void setParents(Individual &offspring, const Individual &parent1, const Individual &parent2) {};

    void apply(std::mt19937 &generator);

private:
    Individual *individual;
    int numUnits;
    double mutationRate;
};

#endif // GENPROG_MUTATION_H
```

13.21 include/operators/Mutation/GuidedMutation.h File Reference

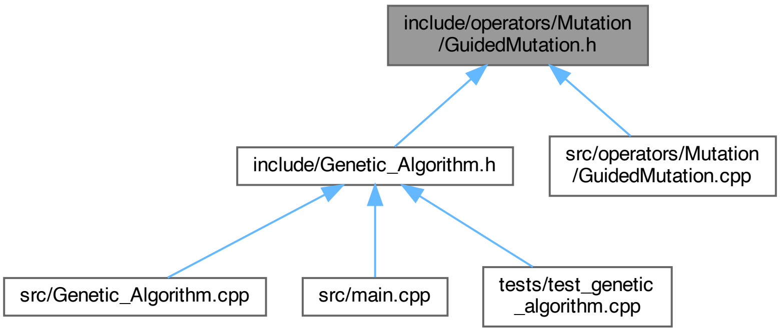
Defines the GuidedMutation class for performing guided mutation operations in a genetic algorithm.

```
#include <vector>
#include "../Operator.h"
#include "../../utils/Individual.h"
#include "../../utils/Parameter.h"
```

Include dependency graph for GuidedMutation.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- class GuidedMutation

Macros

- #define GENETIC_ALGORITHM_GUIDEDMUTATION_H

13.21.1 Detailed Description

Defines the GuidedMutation class for performing guided mutation operations in a genetic algorithm.
The GuidedMutation class provides methods to perform guided mutation operations on individuals in a genetic algorithm. This class is designed to introduce targeted variations into the population by modifying individuals' genetic material based on specific rules or heuristics, which can then be used in subsequent generations of the algorithm.

Date

Created on May 20, 2024

Authors

ACS Gerardium Rush - Pentlandite:

- Alex N Njeumi
- Geyu Ji
- Melissa Y S Sim
- Mingsheng Cai
- Peifeng Tan
- Yongwen Chen
- Zihan Li

13.21.2 Macro Definition Documentation

13.21.2.1 GENETIC_ALGORITHM_GUIDEDMUTATION_H

```
#define GENETIC_ALGORITHM_GUIDEDMUTATION_H
```

13.22 GuidedMutation.h

Go to the documentation of this file.

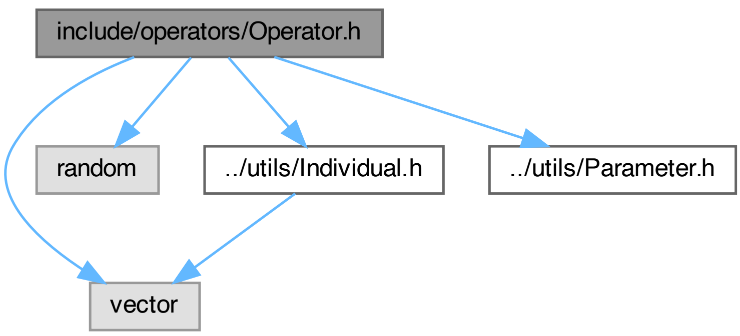
```
#pragma once

#ifndef GENETIC_ALGORITHM_GUIDEDMUTATION_H
#define GENETIC_ALGORITHM_GUIDEDMUTATION_H

#include <vector>
#include "../Operator.h"
#include "../../utils/Individual.h"
#include "../../utils/Parameter.h"

class GuidedMutation : public Operator {
private:
    Individual *individual;
    int numUnits;
    double mutationRate;
public:
    GuidedMutation(int vectorSize, const Algorithm_Parameters &params);

    void setIndividual(Individual &individual) override;

    void setPopulation(const std::vector<Individual> &population, std::vector<Individual> &selected)
{};

    void setParents(Individual &offspring, const Individual &parent1, const Individual &parent2) {};

    void apply(std::mt19937 &generator) override;
};


#endif //GENETIC_ALGORITHM_GUIDEDMUTATION_H
```

13.23 include/operators/Operator.h File Reference

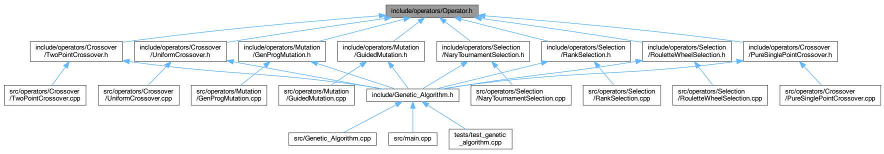
Defines the abstract base class for genetic algorithm operators.

```
#include <vector>
#include <random>
#include "../utils/Individual.h"
#include "../utils/Parameter.h"
```

Include dependency graph for Operator.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- class Operator

    Abstract base class for genetic algorithm operators.

Macros

- #define GENETIC_ALGORITHM_OPERATOR_H

13.23.1 Detailed Description

Defines the abstract base class for genetic algorithm operators.
The Operator class serves as an abstract base class for various genetic algorithm operators, such as selection, crossover, and mutation. Each specific operator must inherit from this class and implement the pure virtual function `apply()`. This design allows for a consistent interface and enables polymorphism, making it easier to manage and apply different operators within a genetic algorithm framework.

Date

Created on May 20, 2024

Authors

ACS Gerardium Rush - Pentlandite:

- Alex N Njeumi
- Geyu Ji
- Melissa Y S Sim
- Mingsheng Cai
- Peifeng Tan
- Yongwen Chen
- Zihan Li

13.23.2 Macro Definition Documentation

13.23.2.1 GENETIC_ALGORITHM_OPERATOR_H

```
#define GENETIC_ALGORITHM_OPERATOR_H
```

13.24 Operator.h

Go to the documentation of this file.

```

#pragma once

 #ifndef GENETIC_ALGORITHM_OPERATOR_H
#define GENETIC_ALGORITHM_OPERATOR_H

#include <vector>
#include <random>
```

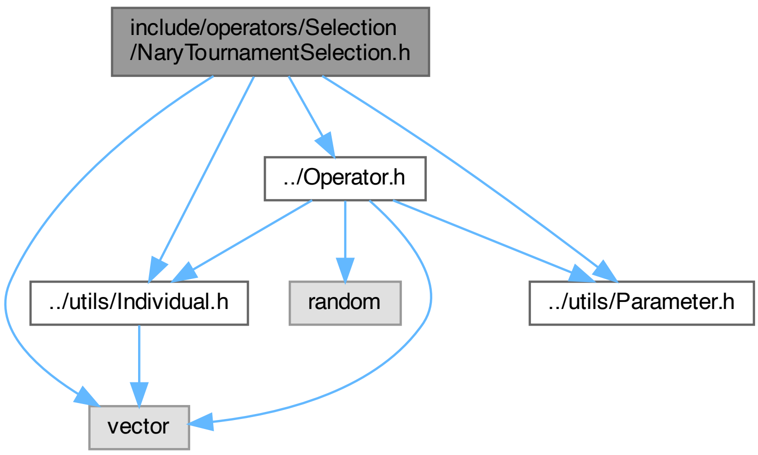
```

#include "../utils/Individual.h"
#include "../utils/Parameter.h"

class Operator {
public:
    virtual ~Operator() = default;

    virtual void apply(std::mt19937 &generator) = 0;

    virtual void setPopulation(const std::vector<Individual> &population, std::vector<Individual>
    &selected) = 0;

    virtual void setIndividual(Individual &individual) = 0;

    virtual void setParents(Individual &offspring, const Individual &parent1, const Individual
    &parent2) = 0;
};

#endif //GENETIC_ALGORITHM_OPERATOR_H
```

13.25 include/operators/Selection/NaryTournamentSelection.h File Reference

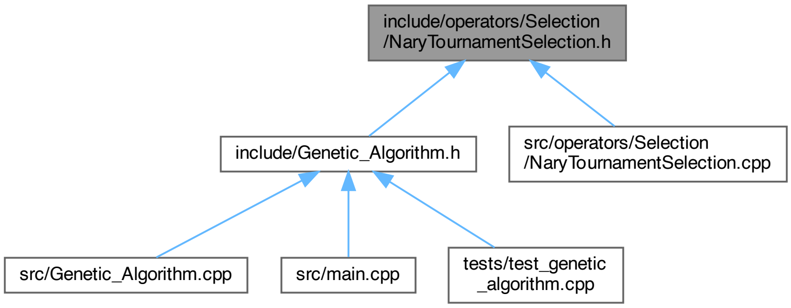
Defines the NaryTournamentSelection class for performing n-ary tournament selection in a genetic algorithm.

```
#include <vector>
#include "../Operator.h"
#include "../../utils/Individual.h"
#include "../../utils/Parameter.h"
```

Include dependency graph for NaryTournamentSelection.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- class NaryTournamentSelection

Macros

- #define GENETIC_ALGORITHM_NARYTOURNAMENTSELECTION_H

13.25.1 Detailed Description

Defines the NaryTournamentSelection class for performing n-ary tournament selection in a genetic algorithm.
The NaryTournamentSelection class provides methods to perform n-ary tournament selection on a population of individuals to select individuals for the next generation. This class is designed to be used in genetic algorithms to select the fittest individuals based on tournament competition among a subset of the population.

Date

Created on May 20, 2024

Authors

ACS Gerardium Rush - Pentlandite:

- Alex N Njeumi
- Geyu Ji
- Melissa Y S Sim
- Mingsheng Cai
- Peifeng Tan
- Yongwen Chen
- Zihan Li

13.25.2 Macro Definition Documentation

13.25.2.1 GENETIC_ALGORITHM_NARYTOURNAMENTSELECTION_H

```
#define GENETIC_ALGORITHM_NARYTOURNAMENTSELECTION_H
```

13.26 NaryTournamentSelection.h

Go to the documentation of this file.

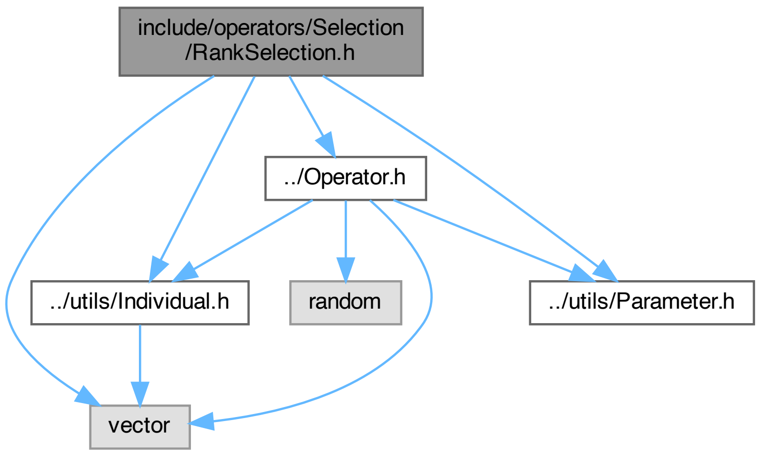
```

#pragma once

#ifndef GENETIC_ALGORITHM_NARYTOURNAMENTSELECTION_H
#define GENETIC_ALGORITHM_NARYTOURNAMENTSELECTION_H


#include <vector>
#include "../Operator.h"
#include "../../utils/Individual.h"
#include "../../utils/Parameter.h"

class NaryTournamentSelection : public Operator {
private:
    std::vector<Individual> *selected;
    const std::vector<Individual> *population;
    int tournamentSize;
public:
    NaryTournamentSelection(const Algorithm_Parameters &params);

    void setPopulation(const std::vector<Individual> &population, std::vector<Individual> &selected)
override;

    void setIndividual(Individual &individual) {};

    void setParents(Individual &offspring, const Individual &parent1, const Individual &parent2) {};

    void apply(std::mt19937 &generator) override;
};

#endif //GENETIC_ALGORITHM_NARYTOURNAMENTSELECTION_H
```

13.27 include/operators/Selection/RankSelection.h File Reference

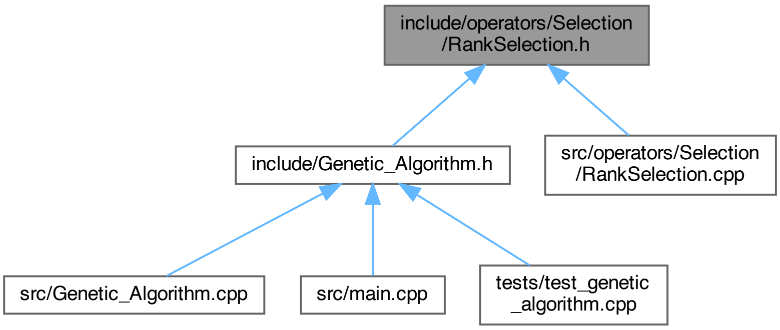
Defines the RankSelection class for performing rank-based selection in a genetic algorithm.

```
#include <vector>
#include "../Operator.h"
#include "../../utils/Individual.h"
#include "../../utils/Parameter.h"
```

Include dependency graph for RankSelection.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- class RankSelection

Macros

- #define GENETIC_ALGORITHM_RANKSELECTION_H

13.27.1 Detailed Description

Defines the RankSelection class for performing rank-based selection in a genetic algorithm.
The RankSelection class provides methods to perform rank-based selection on a population of individuals to select individuals for the next generation. This class is designed to be used in genetic algorithms to select the fittest individuals based on their rank within the population.

Date

Created on May 21, 2024

Authors

ACS Gerardium Rush - Pentlandite:

- Alex N Njeumi
- Geyu Ji
- Melissa Y S Sim
- Mingsheng Cai
- Peifeng Tan
- Yongwen Chen
- Zihan Li

13.27.2 Macro Definition Documentation

13.27.2.1 GENETIC_ALGORITHM_RANKSELECTION_H

```
#define GENETIC_ALGORITHM_RANKSELECTION_H
```

13.28 RankSelection.h

Go to the documentation of this file.

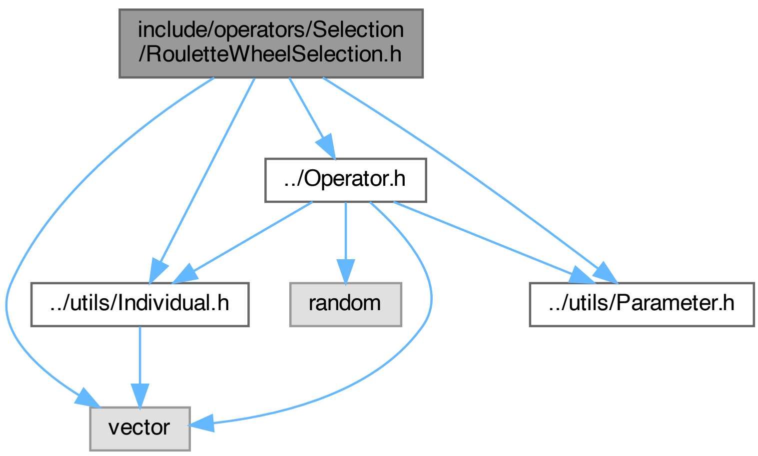
```
#pragma once

#ifndef GENETIC_ALGORITHM_RANKSELECTION_H
#define GENETIC_ALGORITHM_RANKSELECTION_H


#include <vector>
#include "../Operator.h"
#include "../../utils/Individual.h"
#include "../../utils/Parameter.h"

class RankSelection : public Operator {
private:
    std::vector<Individual> *selected;
    const std::vector<Individual> *population;
    int populationSize;
public:
    RankSelection(const Algorithm_Parameters &params);

    void setPopulation(const std::vector<Individual> &population, std::vector<Individual> &selected)
    override;

    void setIndividual(Individual &individual) {};

    void setParents(Individual &offspring, const Individual &parent1, const Individual &parent2) {};

    void apply(std::mt19937 &generator) override;
};

#endif //GENETIC_ALGORITHM_NARYTOURNAMENTSELECTION_H
```

13.29 include/operators/Selection/RouletteWheelSelection.h File Reference

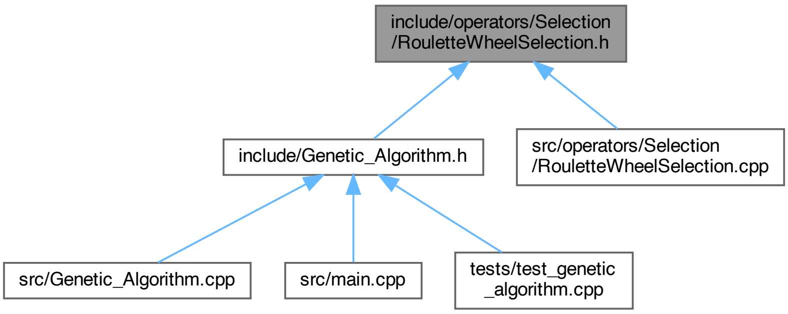
Defines the RouletteWheelSelection class for performing roulette wheel selection in a genetic algorithm.

```
#include <vector>
#include "../Operator.h"
#include "../../utils/Individual.h"
#include "../../utils/Parameter.h"
```

Include dependency graph for RouletteWheelSelection.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- class RouletteWheelSelection

Macros

- #define GENETIC_ALGORITHM_ROULETTEWHEELSELECTION_H

13.29.1 Detailed Description

Defines the RouletteWheelSelection class for performing roulette wheel selection in a genetic algorithm.
The RouletteWheelSelection class provides methods to perform roulette wheel selection on a population of individuals to select individuals for the next generation. This class is designed to be used in genetic algorithms to select individuals based on their fitness proportionally.

Date

Created on May 21, 2024

Authors

ACS Gerardium Rush - Pentlandite:

- Alex N Njeumi
- Geyu Ji
- Melissa Y S Sim
- Mingsheng Cai
- Peifeng Tan
- Yongwen Chen
- Zihan Li

13.29.2 Macro Definition Documentation

13.29.2.1 GENETIC_ALGORITHM_ROULETTEWHEELSELECTION_H

```
#define GENETIC_ALGORITHM_ROULETTEWHEELSELECTION_H
```

13.30 RouletteWheelSelection.h

Go to the documentation of this file.

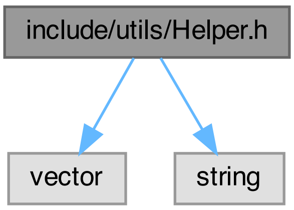
```
#pragma once

#ifndef GENETIC_ALGORITHM_ROULETTEWHEELSELECTION_H
#define GENETIC_ALGORITHM_ROULETTEWHEELSELECTION_H

#include <vector>
#include "../Operator.h"
#include "../../utils/Individual.h"
#include "../../utils/Parameter.h"

class RouletteWheelSelection: public Operator {
private:
    std::vector<Individual> *selected;
    const std::vector<Individual> *population;
    int populationSize;
public:
    RouletteWheelSelection(const Algorithm_Parameters &params);

    void setPopulation(const std::vector<Individual> &population, std::vector<Individual> &selected);

    void setIndividual(Individual &individual) {};

    void setParents(Individual &offspring, const Individual &parent1, const Individual &parent2) {};

    void apply(std::mt19937 &generator) override;
};

#endif //GENETIC_ALGORITHM_ROULETTEWHEELSELECTION_H
```

13.31 include/utils/Helper.h File Reference

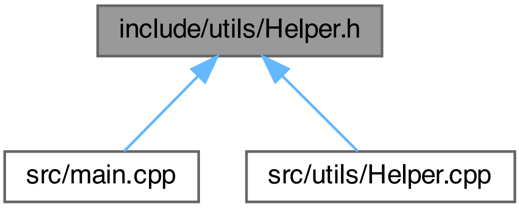
Header file containing helper functions for logging and time retrieval.

`#include <vector>`
`#include <string>`

Include dependency graph for Helper.h:



This graph shows which files directly or indirectly include this file:



Macros

- #define HELPER_H

Functions

- std::string get_current_time_str ()

    Retrieves the current time as a formatted string.
- void log_results (const std::vector< int > &vector, double elapsed_time, double performance, double recovery, double grade)

    Logs the results of the genetic algorithm optimization to a file.

13.31.1 Detailed Description

Header file containing helper functions for logging and time retrieval.
This header file declares functions for retrieving the current time as a string and logging the results of a genetic algorithm optimization process. The logging function saves performance metrics and the final configuration of the optimized circuit to a log file.

Date

Created on May 23, 2024

Authors

ACS Gerardium Rush - Pentlandite:

- Alex N Njeumi
- Geyu Ji
- Melissa Y S Sim
- Mingsheng Cai
- Peifeng Tan
- Yongwen Chen
- Zihan Li

13.31.2 Macro Definition Documentation

13.31.2.1 HELPER_H

```
#define HELPER_H
```

13.31.3 Function Documentation

13.31.3.1 get_current_time_str()

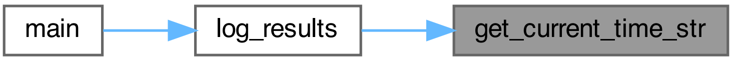
```
std::string get_current_time_str ( )
```

Retrieves the current time as a formatted string.
This function returns the current time formatted as "YYYY-MM-DD_HH-MM-SS".

Returns

A string representing the current time.

Here is the caller graph for this function:



13.31.3.2 log_results()

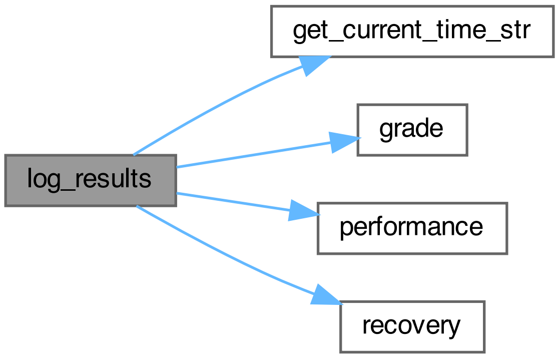
```
void log_results (
            const std::vector< int > & vector,
            double elapsed_time,
            double performance,
            double recovery,
            double grade )
```

Logs the results of the genetic algorithm optimization to a file.
This function creates a new log folder with the current timestamp, and saves the performance metrics, recovery, grade, and the final configuration of the optimized circuit to a log file.

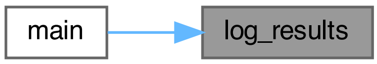
Parameters

vector	The final configuration vector of the optimized circuit.
elapsed_time	The time taken to complete the optimization process, in seconds.
performance	The performance metric of the optimized circuit.
recovery	The recovery metric of the optimized circuit.
grade	The grade metric of the optimized circuit.

Here is the call graph for this function:



Here is the caller graph for this function:



13.32 Helper.h

Go to the documentation of this file.

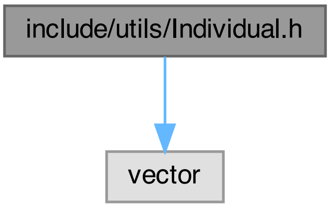
```
00001
00023 #pragma once
00024
00025 #ifndef HELPER_H
00026 #define HELPER_H
00027
00028 #include <vector>
00029 #include <string>
00030
00038 std::string get_current_time_str();
00039
00052 void log_results(const std::vector<int>& vector, double elapsed_time, double performance, double
      recovery, double grade);
00053
00054 #endif // HELPER_H
```

13.33 include/utils/Individual.h File Reference

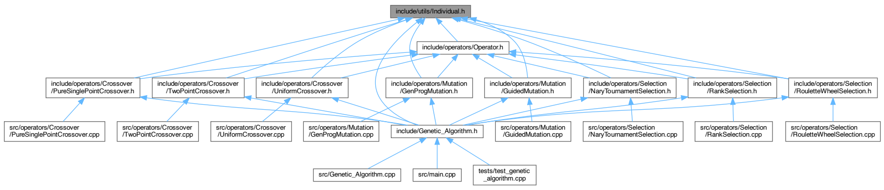
Defines the Individual structure for representing individuals in a genetic algorithm.

`#include <vector>`

Include dependency graph for Individual.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct Individual

  Represents an individual in a genetic algorithm.

Macros

- #define GENETIC_ALGORITHM_INDIVIDUAL_H

13.33.1 Detailed Description

Defines the Individual structure for representing individuals in a genetic algorithm.

The Individual structure is used to represent a single individual in a genetic algorithm population. It contains a vector of genes that represent the circuit configuration and a fitness value that represents the quality or performance of the circuit configuration.

Date

Created on May 20, 2024

Authors

ACS Gerardium Rush - Pentlandite:

- Alex N Njeumi
- Geyu Ji
- Melissa Y S Sim
- Mingsheng Cai
- Peifeng Tan
- Yongwen Chen
- Zihan Li

13.33.2 Macro Definition Documentation

13.33.2.1 GENETIC_ALGORITHM_INDIVIDUAL_H

```
#define GENETIC_ALGORITHM_INDIVIDUAL_H
```

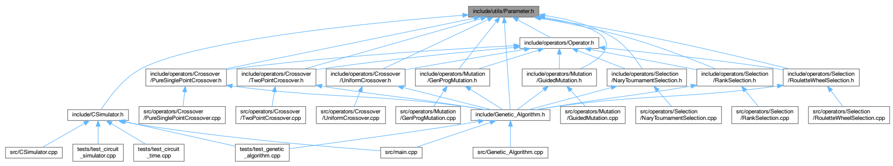
13.34 Individual.h

Go to the documentation of this file.

```
00001
00023 #pragma once
00024
00025 #ifndef GENETIC_ALGORITHM_INDIVIDUAL_H
00026 #define GENETIC_ALGORITHM_INDIVIDUAL_H
00027
00028 #include <vector>
00029
00030
00039 struct Individual {
00040     std::vector<int> genes;
00041     double fitness;
00042 };
00043
00044 #endif //GENETIC_ALGORITHM_INDIVIDUAL_H
```

13.35 include/utils/Parameter.h File Reference

Defines the Algorithm_Parameters structure for configuring the parameters of a genetic algorithm.
This graph shows which files directly or indirectly include this file:



Data Structures

- struct Algorithm_Parameters

    Configures the parameters of a genetic algorithm.
- struct CircuitParameters

    Defines the parameters for the circuit simulator.

Macros

- #define GENETIC_ALGORITHM_PARAMETER_H

13.35.1 Detailed Description

Defines the Algorithm_Parameters structure for configuring the parameters of a genetic algorithm.
The Algorithm_Parameters structure is used to configure various parameters of a genetic algorithm, such as the number of generations, population size, tournament size, crossover rate, mutation rate, elite percentage, and random seed. These parameters control the behavior of the genetic algorithm and can be customized to optimize performance.

Date

Created on May 20, 2024

Authors

ACS Gerardium Rush - Pentlandite:

- Alex N Njeumi
- Geyu Ji
- Melissa Y S Sim
- Mingsheng Cai
- Peifeng Tan
- Yongwen Chen
- Zihan Li

13.35.2 Macro Definition Documentation

13.35.2.1 GENETIC_ALGORITHM_PARAMETER_H

```
#define GENETIC_ALGORITHM_PARAMETER_H
```

13.36 Parameter.h

Go to the documentation of this file.

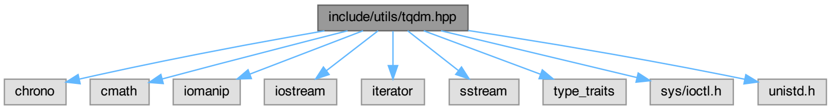
```

#pragma once

#ifndef GENETIC_ALGORITHM_PARAMETER_H
#define GENETIC_ALGORITHM_PARAMETER_H

struct Algorithm_Parameters {
    int maxGenerations = 1000;
    int populationSize = 100;
    std::string selection = "NaryTournamentSelection";
    int tournamentSize = 3;
    std::string crossover = "TwoPointCrossover";
    double crossoverRate = 0.8;
    std::string mutation = "GuidedMutation";
    double mutationRate = 0.1;
    double elitePercentage = 0.2;
    double initialTemp = 1000.0;
    double deltT = 1.0;
    unsigned int randomSeed = 0;
    Algorithm_Parameters(int gens, int popSize, std::string selection, int tourSize,
                         std::string crossover, double crossRate, std::string mutation, double mutRate,
                         double elitePerc, double initialTemp, double deltT, double randomSeed)
            : maxGenerations(gens), populationSize(popSize), selection(selection), tournamentSize(tourSize),
              crossover(crossover), crossoverRate(crossRate), mutation(mutation), mutationRate(mutRate),
              elitePercentage(elitePerc), initialTemp(initialTemp), deltT(deltT), randomSeed(randomSeed) {}
};

struct CircuitParameters {
    double tolerance = 1e-6;
    int maxIterations = 1000;
    CircuitParameters(double tol, int maxIter) : tolerance(tol), maxIterations(maxIter) {}
};

#endif //GENETIC_ALGORITHM_PARAMETER_H
```

13.37 include/utils/tqdm.hpp File Reference

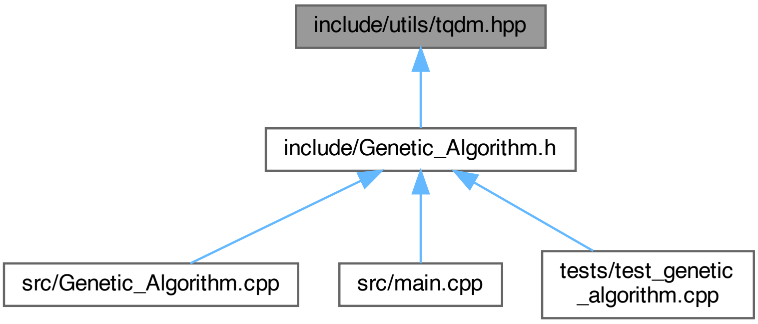
```
#include <chrono>
#include <cmath>
#include <iomanip>
#include <iostream>
#include <iterator>
#include <sstream>
#include <type_traits>
```

```
#include <sys/ioctl.h>
#include <unistd.h>
```

Include dependency graph for tqdm.hpp:



This graph shows which files directly or indirectly include this file:



Data Structures

- class tqdm::Chronometer

    A simple chronometer to measure elapsed time.
- class tqdm::progress_bar

    A class representing a progress bar.
- class tqdm::iter_wrapper< ForwardIter, Parent >

    A wrapper class for iterators to update the progress bar.
- class tqdm::tqdm_for_lvalues< ForwardIter, EndIter >

    A class for progress bars that iterate over lvalues.
- class tqdm::tqdm_for_rvalues< Container >

    A class for progress bars that iterate over rvalues.
- class tqdm::int_iterator< IntType >

    A random access iterator for integers.
- class tqdm::range< IntType >

    A range of integers represented by iterators.
- class tqdm::timing_iterator_end_sentinel

    An end sentinel for timing iterators.
- class tqdm::timing_iterator

    A forward iterator that returns the elapsed time.
- struct tqdm::timer

    A timer that can be used to create a timing iterator.
- class tqdm::tqdm_timer

    A progress bar for a timer.

Namespaces

- namespace tqdm

Typedefs

- using tqdm::index = std::ptrdiff_t
- using tqdm::time_point_t = std::chrono::time_point<std::chrono::steady_clock>

Functions

- double tqdm::elapsed_seconds (time_point_t from, time_point_t to)

  Calculate the elapsed seconds between two time points.
- void tqdm::clamp (double &x, double a, double b)

  Clamp a value between two bounds.
- int tqdm::get_terminal_width ()

  Get the width of the terminal window.
- template<class Container >
  tqdm::tqdm_for_lvalues (Container &) -> tqdm_for_lvalues< typename Container::iterator >
- template<class Container >
  tqdm::tqdm_for_lvalues (const Container &) -> tqdm_for_lvalues< typename Container::const_iterator >
- template<class Container >
  tqdm::tqdm_for_rvalues (Container &&) -> tqdm_for_rvalues< Container >
- template<class ForwardIter >
  auto tqdm::tqdm (const ForwardIter &first, const ForwardIter &last)

  Create a tqdm progress bar for a range of iterators.
- template<class ForwardIter >
  auto tqdm::tqdm (const ForwardIter &first, const ForwardIter &last, index total)

  Create a tqdm progress bar for a range of iterators with a specified total size.
- template<class Container >
  auto tqdm::tqdm (const Container &C)

  Create a tqdm progress bar for a container.
- template<class Container >
  auto tqdm::tqdm (Container &C)

  Create a tqdm progress bar for a container.
- template<class Container >
  auto tqdm::tqdm (Container &&C)

  Create a tqdm progress bar for a container (rvalue reference).
- template<class IntType >
  auto tqdm::trange (IntType first, IntType last)

  Create a tqdm progress bar for a range of integers.
- template<class IntType >
  auto tqdm::trange (IntType last)

  Create a tqdm progress bar for a range of integers.
- auto tqdm::tqdm (timer t)

  Create a tqdm progress bar for a timer.

13.38 tqdm.hpp

Go to the documentation of this file.

```

 #pragma once

#include <chrono>
#include <cmath>
#include <iomanip>
#include <iostream>
```

```
#include <iterator>
#include <sstream>
#include <type_traits>
#include <sys/ioctl.h>
#include <unistd.h>

// -------------------- chrono stuff --------------------

namespace tqdm {
    using index = std::ptrdiff_t; // maybe std::size_t, but I hate unsigned types.
    using time_point_t = std::chrono::time_point<std::chrono::steady_clock>;

    inline double elapsed_seconds(time_point_t from, time_point_t to) {
        using seconds = std::chrono::duration<double>;
        return std::chrono::duration_cast<seconds>(to - from).count();
    }

    class Chronometer {
    public:
        Chronometer() : start_(std::chrono::steady_clock::now()) {}

        double reset() {
            auto previous = start_;
            start_ = std::chrono::steady_clock::now();

            return elapsed_seconds(previous, start_);
        }

        [[nodiscard]] double peek() const {
            auto now = std::chrono::steady_clock::now();

            return elapsed_seconds(start_, now);
        }

        [[nodiscard]] time_point_t get_start() const { return start_; }

    private:
        time_point_t start_;
    };

// -------------------- progress_bar --------------------

    inline void clamp(double &x, double a, double b) {
        if (x < a) x = a;
        if (x > b) x = b;
    }

    inline int get_terminal_width() {
        struct winsize w;
        ioctl(STDOUT_FILENO, TIOCGWINSZ, &w);
        return w.ws_col;
    }

    class progress_bar {
    public:
        void restart() {
            chronometer_.reset();
            refresh_.reset();
            symbol_index_ = 0;
        }

        void update(double progress) {
            clamp(progress, 0, 1);
            symbol_index_++;

            if (time_since_refresh() > min_time_per_update_ || progress == 0 ||
                progress == 1) {
                reset_refresh_timer();
                display(progress);
            }
            suffix_.str("");
        }

        void set_ostream(std::ostream &os) { os_ = &os; }

        void set_prefix(std::string s) { prefix_ = std::move(s); }

        void set_bar_size(int size) { bar_size_ = size; }

        void set_min_update_time(double time) { min_time_per_update_ = time; }

        template<class T>
        progress_bar &operator«(const T &t) {
            suffix_ « t;
            return *this;
        }

```

```
        double elapsed_time() const { return chronometer_.peek(); }

    private:
        void display(double progress) {
            auto flags = os_->flags();

            double t = chronometer_.peek();
            double eta = t / progress - t;

            std::stringstream bar;

            // Lengthy prefix
            std::stringstream temp;
            temp « prefix_ « '{' « std::fixed « std::setprecision(1)
                 « std::setw(5) « 100 * progress « "%} "
                 « " (" « t « "s < " « eta « "s) ";

            // Dynamic bar size
            int terminal_width = get_terminal_width();
            int fixed_length = temp.str().size() + suffix_.str().size();
            bar_size_ = terminal_width - fixed_length - 10; // Safety margin

            bar « '\r' « prefix_ « '{' « std::fixed « std::setprecision(1)
                « std::setw(5) « 100 * progress « "%} ";

            print_bar(bar, progress);

            bar « " (" « t « "s < " « eta « "s) ";

            std::string sbar = bar.str();
            std::string suffix = suffix_.str();

            index out_size = sbar.size() + suffix.size();
            term_cols_ = std::max(term_cols_, out_size);
            index num_blank = term_cols_ - out_size;

            (*os_) « sbar « suffix « std::string(num_blank, ' ') « std::flush;

            os_->flags(flags);
        }

        void print_bar(std::stringstream &ss, double filled) const {
            auto num_filled = static_cast<index>(std::round(filled * bar_size_));
            static const char symbols[] = {'!', '@', '$', '%', '^', '&', '*', '+', '~'};
            int num_symbols = sizeof(symbols) / sizeof(symbols[0]);

            ss « '[';
            for (index i = 0; i < bar_size_; ++i) {
                if (i < num_filled) {
                    ss « '#';
                } else if (i == num_filled && num_filled > 0) {
                    ss « symbols[symbol_index_ % num_symbols]; // Use symbol_index_ for frequency
                } else {
                    ss « '.';
                }
            }
            ss « ']';
        }

        double time_since_refresh() const { return refresh_.peek(); }

        void reset_refresh_timer() { refresh_.reset(); }

        Chronometer chronometer_{};
        Chronometer refresh_{};
        double min_time_per_update_{0.15}; // found experimentally

        std::ostream *os_{&std::cerr};

        index bar_size_{40};
        index term_cols_{1};
        index symbol_index_{0};

        std::string prefix_{};
        std::stringstream suffix_{};
    };

// -------------------- iter_wrapper --------------------

    template<class ForwardIter, class Parent>
    class iter_wrapper {
    public:
        using iterator_category = typename ForwardIter::iterator_category;
        using value_type = typename ForwardIter::value_type;
        using difference_type = typename ForwardIter::difference_type;
        using pointer = typename ForwardIter::pointer;
        using reference = typename ForwardIter::reference;
```

```

        iter_wrapper(ForwardIter it, Parent *parent) : current_(it), parent_(parent) {}

        auto operator*() { return *current_; }

        void operator++() { ++current_; }

        template<class Other>
        bool operator!=(const Other &other) const {
            parent_->update(); // here and not in ++ because I need to run update
            // before first advancement!
            return current_ != other;
        }

        bool operator!=(const iter_wrapper &other) const {
            parent_->update(); // here and not in ++ because I need to run update
            // before first advancement!
            return current_ != other.current_;
        }

        [[nodiscard]] const ForwardIter &get() const { return current_; }

    private:
        friend Parent;
        ForwardIter current_;
        Parent *parent_;
    };

// -------------------- tqdm_for_lvalues --------------------

    template<class ForwardIter, class EndIter = ForwardIter>
    class tqdm_for_lvalues {
    public:
        using this_t = tqdm_for_lvalues<ForwardIter, EndIter>;
        using iterator = iter_wrapper<ForwardIter, this_t>;
        using value_type = typename ForwardIter::value_type;
        using size_type = index;
        using difference_type = index;

        tqdm_for_lvalues(ForwardIter begin, EndIter end)
                : first_(begin, this), last_(end), num_iters_(std::distance(begin, end)) {}

        tqdm_for_lvalues(ForwardIter begin, EndIter end, index total)
                : first_(begin, this), last_(end), num_iters_(total) {}

        template<class Container>
        explicit tqdm_for_lvalues(Container &C)
                : first_(C.begin(), this), last_(C.end()), num_iters_(C.size()) {}

        template<class Container>
        explicit tqdm_for_lvalues(const Container &C)
                : first_(C.begin(), this), last_(C.end()), num_iters_(C.size()) {}

        tqdm_for_lvalues(const tqdm_for_lvalues &) = delete;

        tqdm_for_lvalues(tqdm_for_lvalues &&) = delete;

        tqdm_for_lvalues &operator=(tqdm_for_lvalues &&) = delete;

        tqdm_for_lvalues &operator=(const tqdm_for_lvalues &) = delete;

        ~tqdm_for_lvalues() = default;

        template<class Container>
        tqdm_for_lvalues(Container &&) = delete; // prevent misuse!

        iterator begin() {
            bar_.restart();
            iters_done_ = 0;
            return first_;
        }

        EndIter end() const { return last_; }

        void update() {
            ++iters_done_;
            bar_.update(calc_progress());
        }

        void set_ostream(std::ostream &os) { bar_.set_ostream(os); }

        void set_prefix(std::string s) { bar_.set_prefix(std::move(s)); }

        void set_bar_size(int size) { bar_.set_bar_size(size); }

        void set_min_update_time(double time) { bar_.set_min_update_time(time); }

```

```
        template<class T>
        tqdm_for_lvalues &operator«(const T &t) {
            bar_ « t;
            return *this;
        }

        void manually_set_progress(double to) {
            clamp(to, 0, 1);
            iters_done_ = std::round(to * num_iters_);
        }

    private:
        double calc_progress() const {
            double denominator = num_iters_;
            if (num_iters_ == 0) denominator += 1e-9;
            return iters_done_ / denominator;
        }

        iterator first_;
        EndIter last_;
        index num_iters_{0};
        index iters_done_{0};
        progress_bar bar_;
    };

    template<class Container>
    tqdm_for_lvalues(Container &) -> tqdm_for_lvalues<typename Container::iterator>;

    template<class Container>
    tqdm_for_lvalues(const Container &)
    -> tqdm_for_lvalues<typename Container::const_iterator>;

// -------------------- tqdm_for_rvalues --------------------

    template<class Container>
    class tqdm_for_rvalues {
    public:
        using iterator = typename Container::iterator;
        using const_iterator = typename Container::const_iterator;
        using value_type = typename Container::value_type;

        explicit tqdm_for_rvalues(Container &&C)
                : C_(std::forward<Container>(C)), tqdm_(C_) {}

        auto begin() { return tqdm_.begin(); }

        auto end() { return tqdm_.end(); }

        void update() { return tqdm_.update(); }

        void set_ostream(std::ostream &os) { tqdm_.set_ostream(os); }

        void set_prefix(std::string s) { tqdm_.set_prefix(std::move(s)); }

        void set_bar_size(int size) { tqdm_.set_bar_size(size); }

        void set_min_update_time(double time) { tqdm_.set_min_update_time(time); }

        template<class T>
        auto &operator«(const T &t) {
            return tqdm_ « t;
        }

        void advance(index amount) { tqdm_.advance(amount); }

        void manually_set_progress(double to) { tqdm_.manually_set_progress(to); }

    private:
        Container C_;
        tqdm_for_lvalues<iterator> tqdm_;
    };

    template<class Container>
    tqdm_for_rvalues(Container &&) -> tqdm_for_rvalues<Container>;

// -------------------- tqdm --------------------

    template<class ForwardIter>
    auto tqdm(const ForwardIter &first, const ForwardIter &last) {
        return tqdm_for_lvalues(first, last);
    }

    template<class ForwardIter>
    auto tqdm(const ForwardIter &first, const ForwardIter &last, index total) {
        return tqdm_for_lvalues(first, last, total);
    }
```

```
    template<class Container>
    auto tqdm(const Container &C) {
        return tqdm_for_lvalues(C);
    }

    template<class Container>
    auto tqdm(Container &C) {
        return tqdm_for_lvalues(C);
    }

    template<class Container>
    auto tqdm(Container &&C) {
        return tqdm_for_rvalues(std::forward<Container>(C));
    }

// -------------------- int_iterator --------------------

    template<class IntType>
    class int_iterator {
    public:
        using iterator_category = std::random_access_iterator_tag;
        using value_type = IntType;
        using difference_type = IntType;
        using pointer = IntType *;
        using reference = IntType &;

        explicit int_iterator(IntType val) : value_(val) {}

        IntType &operator*() { return value_; }

        int_iterator &operator++() {
            ++value_;
            return *this;
        }

        int_iterator &operator--() {
            --value_;
            return *this;
        }

        int_iterator &operator+=(difference_type d) {
            value_ += d;
            return *this;
        }

        difference_type operator-(const int_iterator &other) const {
            return value_ - other.value_;
        }

        bool operator!=(const int_iterator &other) const {
            return value_ != other.value_;
        }

    private:
        IntType value_;
    };

// -------------------- range --------------------

    template<class IntType>
    class range {
    public:
        using iterator = int_iterator<IntType>;
        using const_iterator = iterator;
        using value_type = IntType;

        range(IntType first, IntType last) : first_(first), last_(last) {}

        explicit range(IntType last) : first_(0), last_(last) {}

        [[nodiscard]] iterator begin() const { return first_; }

        [[nodiscard]] iterator end() const { return last_; }

        [[nodiscard]] index size() const { return last_ - first_; }

    private:
        iterator first_;
        iterator last_;
    };

    template<class IntType>
    auto trange(IntType first, IntType last) {
        return tqdm(range(first, last));
    }

    template<class IntType>
```

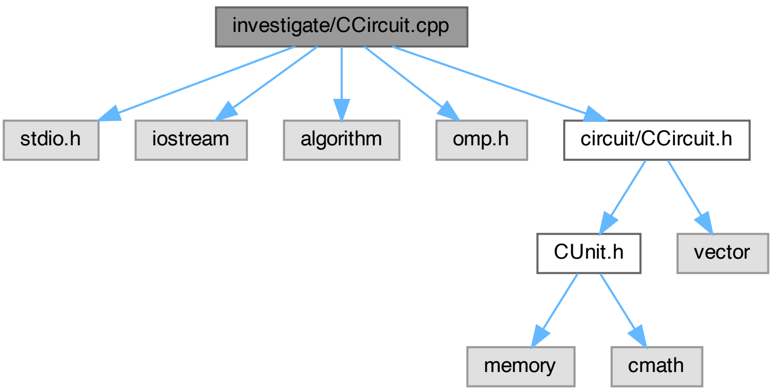
```
    auto trange(IntType last) {
        return tqdm(range(last));
    }

// -------------------- timing_iterator --------------------

    class timing_iterator_end_sentinel {
    public:
        explicit timing_iterator_end_sentinel(double num_seconds)
                : num_seconds_(num_seconds) {}

        [[nodiscard]] double num_seconds() const { return num_seconds_; }

    private:
        double num_seconds_;
    };

    class timing_iterator {
    public:
        using iterator_category = std::forward_iterator_tag;
        using value_type = double;
        using difference_type = double;
        using pointer = double *;
        using reference = double &;

        double operator*() const { return chrono_.peek(); }

        timing_iterator &operator++() { return *this; }

        bool operator!=(const timing_iterator_end_sentinel &other) const {
            return chrono_.peek() < other.num_seconds();
        }

    private:
        tqdm::Chronometer chrono_;
    };

// -------------------- timer -------------------

    struct timer {
    public:
        using iterator = timing_iterator;
        using end_iterator = timing_iterator_end_sentinel;
        using const_iterator = iterator;
        using value_type = double;

        explicit timer(double num_seconds) : num_seconds_(num_seconds) {}

        [[nodiscard]] static iterator begin() { return iterator(); }

        [[nodiscard]] end_iterator end() const {
            return end_iterator(num_seconds_);
        }

        [[nodiscard]] double num_seconds() const { return num_seconds_; }

    private:
        double num_seconds_;
    };

    class tqdm_timer {
    public:
        using iterator = iter_wrapper<timing_iterator, tqdm_timer>;
        using end_iterator = timer::end_iterator;
        using value_type = typename timing_iterator::value_type;
        using size_type = index;
        using difference_type = index;

        explicit tqdm_timer(double num_seconds) : num_seconds_(num_seconds) {}

        tqdm_timer(const tqdm_timer &) = delete;

        tqdm_timer(tqdm_timer &&) = delete;

        tqdm_timer &operator=(tqdm_timer &&) = delete;

        tqdm_timer &operator=(const tqdm_timer &) = delete;

        ~tqdm_timer() = default;

        template<class Container>
        tqdm_timer(Container &&) = delete; // prevent misuse!

        iterator begin() {
            bar_.restart();
            return iterator(timing_iterator(), this);
        }
```

```
        end_iterator end() const { return end_iterator(num_seconds_); }

        void update() {
            double t = bar_.elapsed_time();

            bar_.update(t / num_seconds_);
        }

        void set_ostream(std::ostream &os) { bar_.set_ostream(os); }

        void set_prefix(std::string s) { bar_.set_prefix(std::move(s)); }

        void set_bar_size(int size) { bar_.set_bar_size(size); }

        void set_min_update_time(double time) { bar_.set_min_update_time(time); }

        template<class T>
        tqdm_timer &operator«(const T &t) {
            bar_ « t;
            return *this;
        }

    private:
        double num_seconds_;
        progress_bar bar_;
    };

    inline auto tqdm(timer t) { return tqdm_timer(t.num_seconds()); }

} // namespace tqdm
```

13.39 investigate/CCircuit.cpp File Reference

```
#include <stdio.h>
#include <iostream>
#include <algorithm>
#include <omp.h>
#include "circuit/CCircuit.h"
```

Include dependency graph for CCircuit.cpp:



Variables

- const double k_G_C = 0.004
- const double k_G_I = 0.001
- const double k_w_C = 0.0002
- const double k_w_I = 0.0003
- const double V = 10.0

- const double phi = 0.1
- const double rho = 3000.0

13.39.1 Variable Documentation

13.39.1.1 k_G_C

```
const double k_G_C = 0.004
```

13.39.1.2 k_G_I

```
const double k_G_I = 0.001
```

13.39.1.3 k_w_C

```
const double k_w_C = 0.0002
```

13.39.1.4 k_w_I

```
const double k_w_I = 0.0003
```

13.39.1.5 phi

```
const double phi = 0.1
```

13.39.1.6 rho

```
const double rho = 3000.0
```

13.39.1.7 V

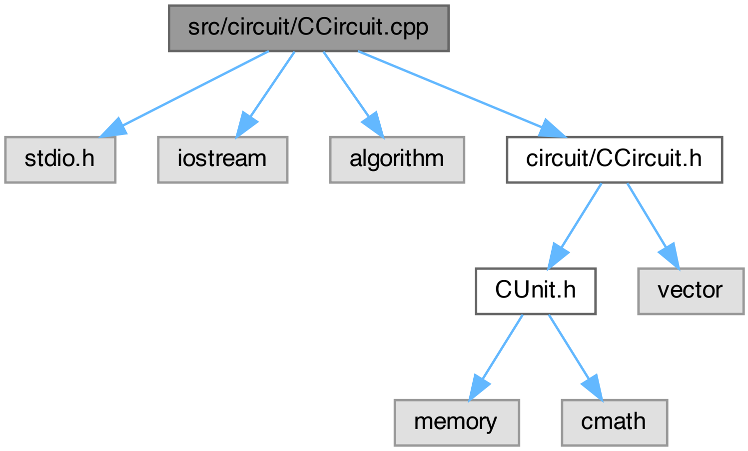
```
const double V = 10.0
```

13.40 src/circuit/CCircuit.cpp File Reference

Defines the Circuit class for modeling and simulating a circuit system.

```
#include <stdio.h>
#include <iostream>
#include <algorithm>
#include "circuit/CCircuit.h"
```

Include dependency graph for CCircuit.cpp:



Variables

- const double k_G_C = 0.004
- const double k_G_I = 0.001
- const double k_w_C = 0.0002
- const double k_w_I = 0.0003
- const double V = 10.0
- const double phi = 0.1
- const double rho = 3000.0

13.40.1 Detailed Description

Defines the Circuit class for modeling and simulating a circuit system.
The Circuit class provides methods to initialize, configure, and simulate a circuit with multiple units. It includes methods for checking validity, printing information, connecting units, initializing feed rates, checking convergence, and calculating flows. The Circuit class is designed to facilitate the modeling and simulation of circuits in various applications, including industrial processes and scientific research.

Date

Created on May 20, 2024

Authors

ACS Gerardium Rush - Pentlandite:

- Alex N Njeumi
- Geyu Ji
- Melissa Y S Sim
- Mingsheng Cai
- Peifeng Tan
- Yongwen Chen
- Zihan Li

13.40.2 Variable Documentation

13.40.2.1 k_G_C

```
const double k_G_C = 0.004
```

13.40.2.2 k_G_I

```
const double k_G_I = 0.001
```

Constants for Gerardium recovery

13.40.2.3 k_w_C

```
const double k_w_C = 0.0002
```

13.40.2.4 k_w_I

```
const double k_w_I = 0.0003
```

Constants for waste recovery

13.40.2.5 phi

```
const double phi = 0.1
```

13.40.2.6 rho

```
const double rho = 3000.0
```

Constants for the circuit

13.40.2.7 V

```
const double V = 10.0
```

13.41 post_process/visualize.py File Reference

Namespaces

- namespace visualize

Functions

- visualize.parse_list (string)

Variables

- visualize.parser
- visualize.type
- visualize.args = parser.parse_args()
- visualize.x = args.vector
- visualize.p = args.perf
- visualize.r = args.recovery
- visualize.g = args.grade
- visualize.graph = graphviz.Digraph()
- visualize.rankdir
- visualize.nodesep
- visualize.ranksep
- visualize.splines
- visualize.overlap
- visualize.dpi

- str visualize.feed_node = 'Feed'
- str visualize.concentrate = 'Concentrate'
- str visualize.tailings = 'Tailings'
- visualize.shape
- visualize.color = 1 else 'red'
- tuple visualize.num_nodes = (len(x) - 1) // 3
- str visualize.from_node = f'Unit {i // 3}'
- tuple visualize.to_node = (num_nodes + 1) else f'Unit {x[i + 1]}'
- visualize.cleanup
- visualize.True
- visualize.format
- visualize.view
- dict visualize.legend_labels
- list visualize.legend_handles
- list visualize.columns = ['Feed'] + [f'Unit {i}' for i in range(0, num_nodes)]
- list visualize.reshaped_list
- visualize.df = pd.DataFrame(reshaped_list).transpose()
- visualize.figsize
- visualize.handles
- visualize.labels
- visualize.loc
- visualize.bbox_to_anchor
- visualize.fontsize
- visualize.cellText
- visualize.values
- visualize.colLabels
- visualize.cellLoc
- visualize.bbox
- visualize.ha
- visualize.va
- visualize.transform
- visualize.bbox_inches

13.42 README.md File Reference

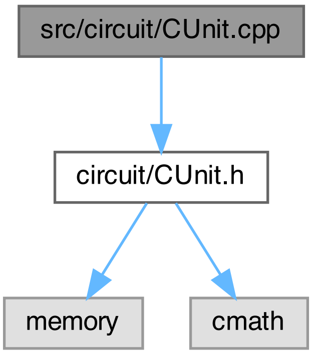
13.43 REFERENCE.md File Reference

13.44 src/circuit/CUnit.cpp File Reference

Defines the CUnit class for representing a unit within a circuit system.

```
#include "circuit/CUnit.h"
```

Include dependency graph for CUnit.cpp:



13.44.1 Detailed Description

Defines the CUnit class for representing a unit within a circuit system.
The CUnit class provides methods to manage and simulate individual units within a circuit. It includes methods for setting and getting various properties of the unit, such as feed rates, mark status, and connection pointers. This class is designed to facilitate the modeling and simulation of circuits in various applications, including industrial processes and scientific research.

Date

Created on May 20, 2024

Authors

ACS Gerardium Rush - Pentlandite:

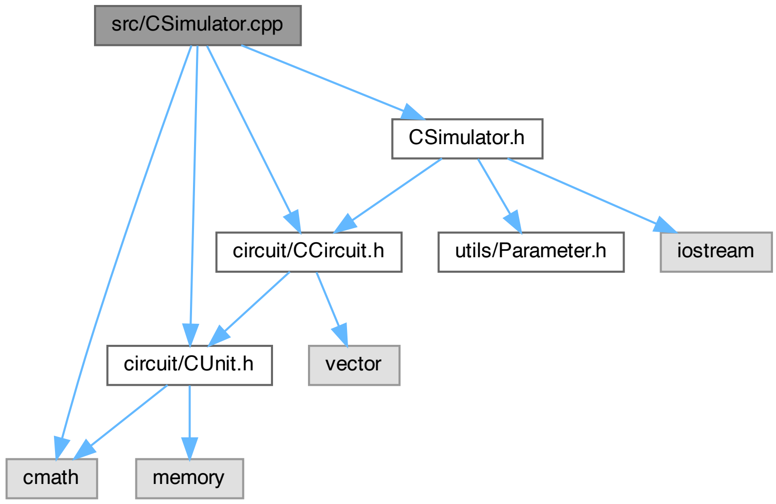
- Alex N Njeumi
- Geyu Ji
- Melissa Y S Sim
- Mingsheng Cai
- Peifeng Tan
- Yongwen Chen
- Zihan Li

13.45 src/CSimulator.cpp File Reference

Defines the functions and structures for simulating and evaluating circuits in a circuit modeling framework.

```
#include "circuit/CUnit.h"
#include "circuit/CCircuit.h"
#include "CSimulator.h"
#include <cmath>
```

Include dependency graph for CSimulator.cpp:



Functions

- double performance (double concentrateG, double concentrateW)

    Computes the performance metric of a circuit.
- double grade (double concentrateG, double concentrateW)

    Computes the grade of a circuit.
- double recovery (double concentrateG, double gerardiumFeed)

    Computes the recovery of a circuit.
- double Evaluate_Circuit (int vectorSize, int ∗circuitVector, struct CircuitParameters parameters)

    Evaluates the performance of a circuit.
- double Evaluate_Circuit (int vectorSize, int ∗circuitVector, double &Recovery, double &Grade, struct CircuitParameters parameters)

    Evaluates the performance, recovery, and grade of a circuit.

13.45.1 Detailed Description

Defines the functions and structures for simulating and evaluating circuits in a circuit modeling framework.
The CSimulator module provides functions for evaluating the performance, grade, and recovery of circuits in a circuit modeling framework. It includes definitions for circuit parameters and multiple overloads for evaluating circuits with or without additional parameters.

Date

Created on May 20, 2024

Authors

ACS Gerardium Rush - Pentlandite:

- Alex N Njeumi
- Geyu Ji
- Melissa Y S Sim
- Mingsheng Cai

- Peifeng Tan
- Yongwen Chen
- Zihan Li

13.45.2 Function Documentation

13.45.2.1 Evaluate_Circuit() [1/2]

```
double Evaluate_Circuit (
            int vectorSize,
            int * circuitVector,
            double & Recovery,
            double & Grade,
            struct CircuitParameters parameters )
```

Evaluates the performance, recovery, and grade of a circuit.

This function evaluates the performance, recovery, and grade of a circuit based on the given circuit vector and additional circuit parameters.

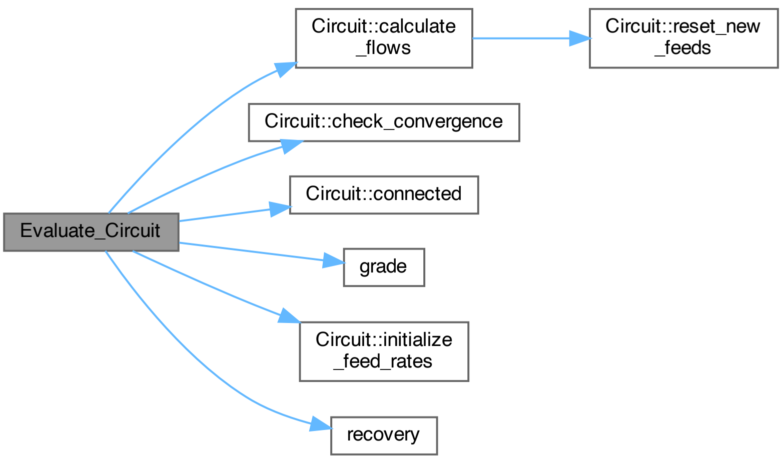
Parameters

vectorSize	The size of the circuit vector.
circuitVector	Pointer to the array representing the circuit configuration.
Recovery	Reference to a double variable to store the recovery value.
Grade	Reference to a double variable to store the grade value.
parameters	Additional parameters for the circuit simulation.

Returns

The performance metric of the circuit.

Here is the call graph for this function:



13.45.2.2 Evaluate_Circuit() [2/2]

```
double Evaluate_Circuit (
            int vectorSize,
```

```
int * circuitVector,
struct CircuitParameters parameters )
```

Evaluates the performance of a circuit.

This function evaluates the performance of a circuit based on the given circuit vector and additional circuit parameters.

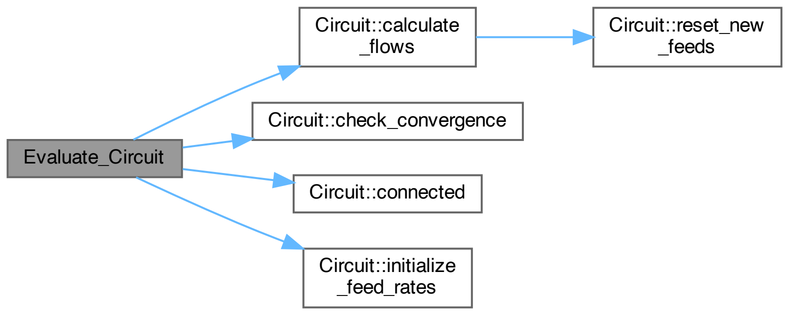
Parameters

vectorSize	The size of the circuit vector.
circuitVector	Pointer to the array representing the circuit configuration.
parameters	Additional parameters for the circuit simulation.

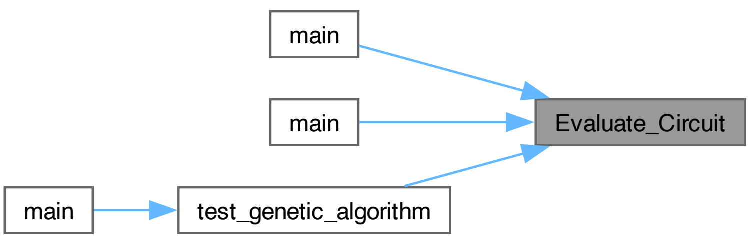
Returns

The performance metric of the circuit.

Here is the call graph for this function:



Here is the caller graph for this function:



13.45.2.3 grade()

```
double grade (
        double concentrateG,
        double concentrateW )
```

Computes the grade of a circuit.

This function computes the grade of a circuit based on the given concentrate gerardium and waste values.

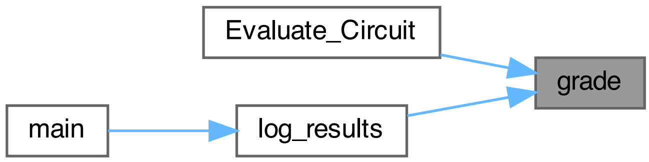
Parameters

concentrateG	The amount of gerardium in the concentrate.
concentrateW	The amount of waste in the concentrate.

Returns

The grade of the circuit.

Here is the caller graph for this function:



13.45.2.4 performance()

```
double performance (
            double concentrateG,
            double concentrateW )
```

Computes the performance metric of a circuit.

This function computes the performance metric of a circuit based on the given concentrate gerardium and waste values.

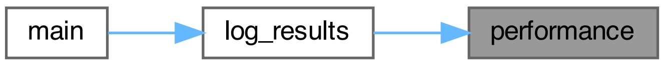
Parameters

concentrateG	The amount of gerardium in the concentrate.
concentrateW	The amount of waste in the concentrate.

Returns

The performance metric of the circuit.

Here is the caller graph for this function:



13.45.2.5 recovery()

```
double recovery (
```

```
double concentrateG,
double gerardiumFeed )
```

Computes the recovery of a circuit.

This function computes the recovery of a circuit based on the given concentrate gerardium and gerardium feed values.

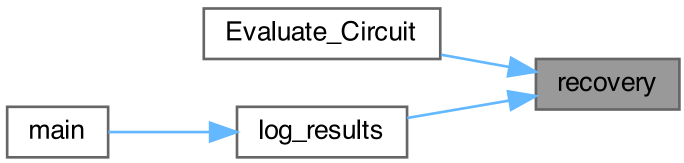
Parameters

concentrateG	The amount of gerardium in the concentrate.
gerardiumFeed	The amount of gerardium fed into the circuit.

Returns

The recovery of the circuit.

Here is the caller graph for this function:

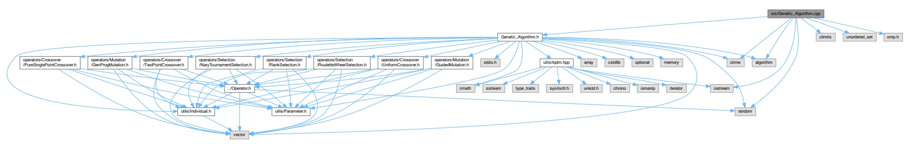


13.46 src/Genetic_Algorithm.cpp File Reference

Defines the functions and structures for implementing a genetic algorithm.

```
#include "Genetic_Algorithm.h"
#include <iostream>
#include <algorithm>
#include <ctime>
#include <climits>
#include <unordered_set>
#include <random>
#include <omp.h>
```

Include dependency graph for Genetic_Algorithm.cpp:



Macros

- #define INVALID_FITNESS -1000000000.0

Functions

- void initializeRandomSeed (const Algorithm_Parameters &params, std::vector< std::mt19937 > &generators)

Initializes the random seed based on algorithm parameters.

- std::vector< Individual > initializePopulation (const Algorithm_Parameters &params, int vectorSize, bool(&validity)(int, int ∗), std::vector< std::mt19937 > &generators)

  Initializes the population for the genetic algorithm.
- void elitism (const std::vector< Individual > &population, std::vector< Individual > &newPopulation, const Algorithm_Parameters &params)

  Apply elitism to preserve the best individuals in a new population.
- double acceptanceProbability (double currentEnergy, double newEnergy, double temperature)

  Calculate the acceptance probability for a new state in Simulated Annealing.
- bool applySimulatedAnnealing (Individual &offspring, Individual &parent1, Individual &parent2, int vector←_size, double(&func)(int, int ∗, struct CircuitParameters), bool(&validity)(int, int ∗), double &Temp, const Algorithm_Parameters &params, const CircuitParameters c_params, std::mt19937 &generator)

  Applies simulated annealing technique to decide whether to accept an offspring in a genetic algorithm.
- int optimize (int vector_size, int ∗vector, double(&func)(int, int ∗, struct CircuitParameters), bool(&validity)(int, int ∗), Algorithm_Parameters params, CircuitParameters c_params)

  Optimizes a solution using the genetic algorithm.

13.46.1 Detailed Description

Defines the functions and structures for implementing a genetic algorithm.
The Genetic_Algorithm module provides functions for initializing populations, setting random seeds, and optimizing solutions using a genetic algorithm framework. This includes selection, crossover, and mutation operators.

Date

Created on May 20, 2024

Authors

ACS Gerardium Rush - Pentlandite:

- Alex N Njeumi
- Geyu Ji
- Melissa Y S Sim
- Mingsheng Cai
- Peifeng Tan
- Yongwen Chen
- Zihan Li

13.46.2 Macro Definition Documentation

13.46.2.1 INVALID_FITNESS

```
#define INVALID_FITNESS -1000000000.0
```

13.46.3 Function Documentation

13.46.3.1 acceptanceProbability()

```
double acceptanceProbability (
            double currentEnergy,
            double newEnergy,
            double temperature )
```

Calculate the acceptance probability for a new state in Simulated Annealing.
This function calculates the probability with which a new state should be accepted over the current state, based on their respective energies and the current temperature of the system. If the new energy is better (higher value), the function returns 1.0, meaning the new state is always accepted. Otherwise, it returns a value calculated using the Boltzmann probability distribution which considers both the difference in energy and the current temperature.

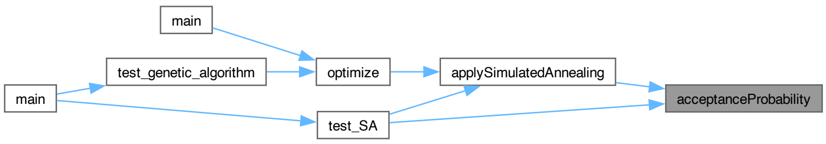
Parameters

currentEnergy	The energy of the current state.
newEnergy	The energy of the new state being considered.
temperature	The current temperature of the system (controls the probability threshold).

Returns

double The probability of accepting the new state.

Here is the caller graph for this function:



13.46.3.2 applySimulatedAnnealing()

```
bool applySimulatedAnnealing (
            Individual & offspring,
            Individual & parent1,
            Individual & parent2,
            int vector_size,
            double(&)(int, int *, struct CircuitParameters) func,
            bool(&)(int, int *) validity,
            double & Temp,
            const Algorithm_Parameters & params,
            const CircuitParameters c_params,
            std::mt19937 & generator )
```

Applies simulated annealing technique to decide whether to accept an offspring in a genetic algorithm.
This function performs the simulated annealing decision process on a newly generated offspring. It compares the offspring's fitness to the average fitness of its parents using the simulated annealing acceptance probability, which depends on the current temperature of the system. The function also handles the cooling of the system temperature and validates the offspring using a provided validity function.

Parameters

offspring	A reference to the offspring individual whose acceptance is being determined.
parent1	A reference to the first parent of the offspring.
parent2	A reference to the second parent of the offspring.
vector_size	The size of the individual's gene vector.
func	A function that calculates the fitness of an individual based on its genes.
validity	A function that checks the validity of an individual's gene configuration.
Temp	A reference to the current temperature used for the simulated annealing process.
params	A structure containing parameters relevant to the annealing process, such as the temperature decrement.
c_params	The circuit parameters used in the fitness evaluation function.
generator	A random number generator used for probabilistic decisions.

Returns

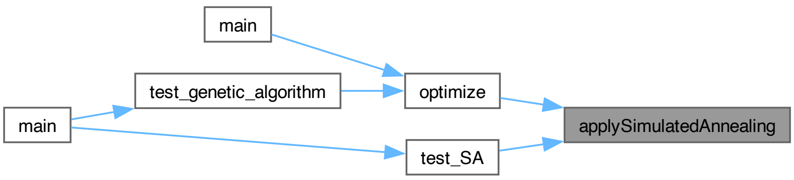
bool Returns true if the offspring is accepted, false otherwise. The function can also modify the offspring directly, setting it to one of the parents if the temperature is low and the acceptance criteria are not met.

Note

The function directly modifies the temperature, decrementing it according to `deltT` from `params`. If the offspring is invalid (as determined by the `validity` function), it is immediately rejected and assigned a fitness value of INVALID_FITNESS.

Here is the call graph for this function:

Here is the caller graph for this function:



13.46.3.3 elitism()

```
void elitism (
            const std::vector< Individual > & population,
            std::vector< Individual > & newPopulation,
            const Algorithm_Parameters & params )
```

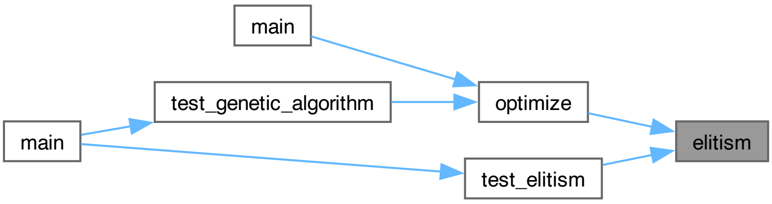
Apply elitism to preserve the best individuals in a new population.

This function sorts the given population in descending order based on their fitness, then selectively copies the top percentage of individuals as specified by the elitePercentage in the Algorithm_Parameters. This is used to ensure that the best individuals are preserved for the next generation, promoting genetic diversity and preventing loss of the best found solutions.

Parameters

population	The current generation of individuals (const reference).
newPopulation	The next generation of individuals where elites will be added (reference).
params	The parameters of the algorithm including population size and elite percentage.

Here is the caller graph for this function:



13.46.3.4 initializePopulation()

```
std::vector< Individual > initializePopulation (
            const Algorithm_Parameters & params,
            int vectorSize,
            bool(&)(int, int *) validity,
            std::vector< std::mt19937 > & generators )
```

Initializes the population for the genetic algorithm.

This function initializes the population for the genetic algorithm, ensuring each individual meets the validity criteria specified by the user.

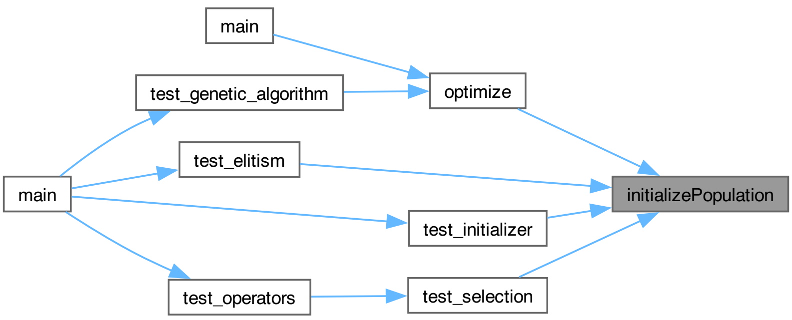
Parameters

params	The algorithm parameters containing population size and other configurations.
vectorSize	The size of the gene vector for each individual.
validity	A function pointer to the validity checking function.
generators	A vector of random number generators for each thread.

Returns

A vector of individuals representing the initial population.

Here is the caller graph for this function:



13.46.3.5 initializeRandomSeed()

```
void initializeRandomSeed (
            const Algorithm_Parameters & params,
            std::vector< std::mt19937 > & generators )
```

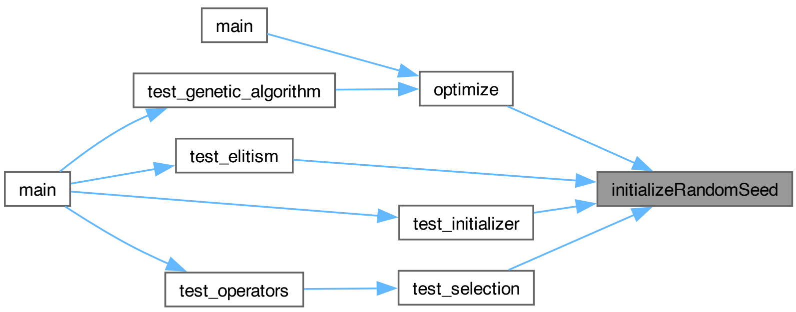
Initializes the random seed based on algorithm parameters.

This function initializes the random seed using the specified seed in the algorithm parameters. If the seed is zero, it uses the current time as the seed.

Parameters

params	The algorithm parameters containing the random seed.
generators	A vector of random number generators for each thread.

Here is the caller graph for this function:



13.46.3.6 optimize()

```
int optimize (
            int vector_size,
            int * vector,
            double(&)(int, int *, struct CircuitParameters) func,
            bool(&)(int, int *) validity,
            struct Algorithm_Parameters params,
            struct CircuitParameters c_params )
```

Optimizes a solution using the genetic algorithm.

This function performs optimization using a genetic algorithm. It initializes the population, assesses the fitness of each individual, and iterates through generations applying selection, crossover, and mutation operators.

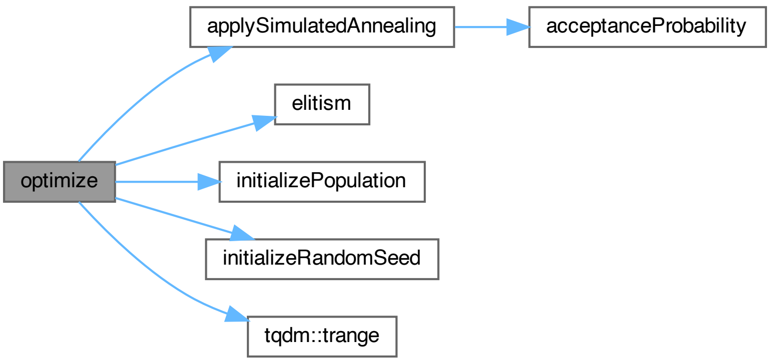
Parameters

vector_size	The size of the gene vector.
vector	Pointer to the array representing the solution vector.
func	A function pointer to the fitness evaluation function.
validity	A function pointer to the validity checking function.
parameters	The algorithm parameters containing various configurations.

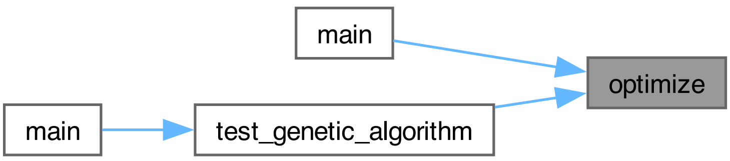
Returns

An integer indicating the success (0) or failure (-1) of the optimization.

Here is the call graph for this function:



Here is the caller graph for this function:

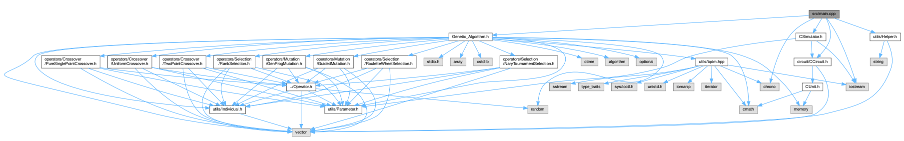


13.47 src/main.cpp File Reference

Main entry point for the circuit optimization program using genetic algorithms.

```
#include <iostream>
#include <chrono>
#include "circuit/CCircuit.h"
#include "CSimulator.h"
#include "Genetic_Algorithm.h"
#include "utils/Helper.h"
```

Include dependency graph for main.cpp:



Functions

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

13.47.1 Detailed Description

Main entry point for the circuit optimization program using genetic algorithms.
Test file for evaluating the functionalities of the Genetic Algorithm module.
Test file for evaluating execution times of the Evaluate_Circuit function with different circuit vector sizes.
Test file for evaluating circuit configurations using the Evaluate_Circuit function.
This program uses a genetic algorithm to optimize a circuit configuration. It sets up the initial circuit parameters and the genetic algorithm parameters, runs the optimization process, and then evaluates the optimized circuit. The results, including performance, recovery, and grade, are printed to the console.
The main steps include:

- Setting up circuit parameters.
- Defining the initial circuit vector.
- Configuring the genetic algorithm parameters.
- Running the genetic algorithm optimization.
- Evaluating the optimized circuit and printing the results.

Date

Created on May 20, 2024

Authors

ACS Gerardium Rush - Pentlandite:

- Alex N Njeumi
- Geyu Ji
- Melissa Y S Sim
- Mingsheng Cai
- Peifeng Tan
- Yongwen Chen
- Zihan Li

This file contains a series of test cases to evaluate different circuit configurations using the Evaluate_Circuit function. Each test case checks the performance, recovery, and grade of a given circuit vector against expected values. The results are printed to the console, and the test passes or fails based on the accuracy of the output.

Date

Created on May 20, 2024

Authors

ACS Gerardium Rush - Pentlandite:

- Alex N Njeumi
- Geyu Ji
- Melissa Y S Sim
- Mingsheng Cai
- Peifeng Tan
- Yongwen Chen
- Zihan Li

This file generates circuit vectors of various sizes, evaluates them using the Evaluate_Circuit function, and records the execution times. The results are printed to the console and also saved to an output file.

Date

Created on May 20, 2024

Authors

ACS Gerardium Rush - Pentlandite:

- Alex N Njeumi
- Geyu Ji
- Melissa Y S Sim
- Mingsheng Cai
- Peifeng Tan
- Yongwen Chen
- Zihan Li

This file contains various test cases to evaluate the correctness and performance of the Genetic Algorithm implementation, including tests for the initializer, crossover, mutation, selection, and simulated annealing.

Date

Created on May 20, 2024

Authors

ACS Gerardium Rush - Pentlandite:

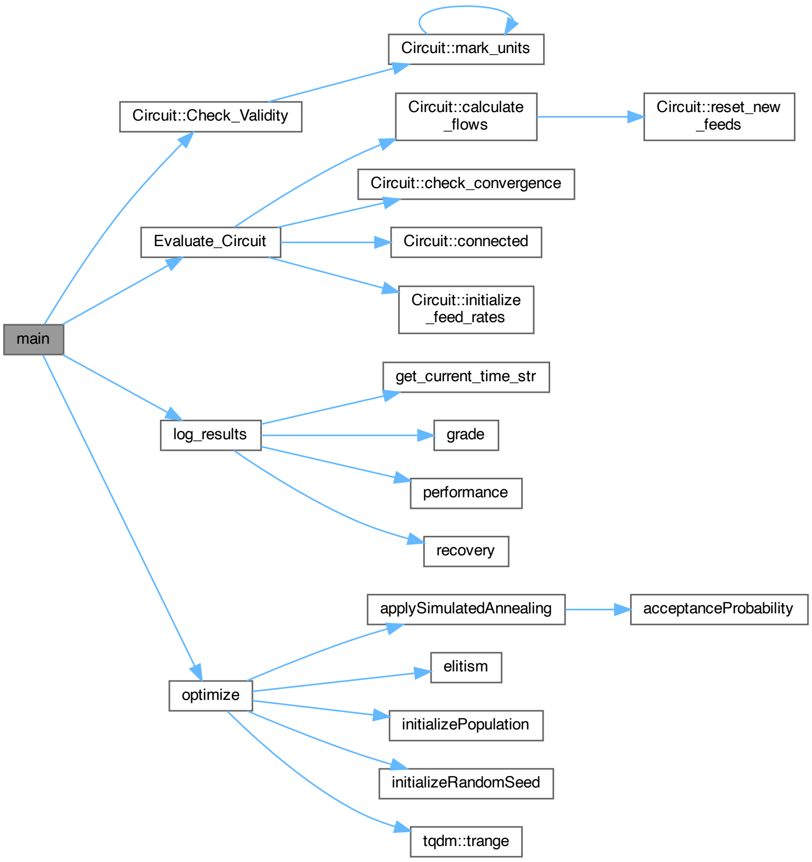
- Alex N Njeumi
- Geyu Ji
- Melissa Y S Sim
- Mingsheng Cai
- Peifeng Tan
- Yongwen Chen
- Zihan Li

13.47.2 Function Documentation

13.47.2.1 main()

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

Here is the call graph for this function:

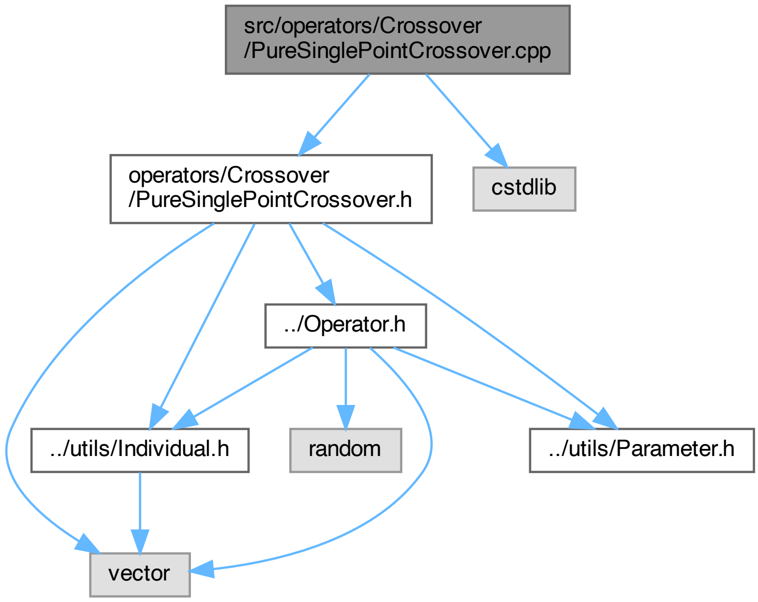


13.48 src/operators/Crossover/PureSinglePointCrossover.cpp File Reference

Defines the PureSinglePointCrossover class for performing single-point crossover in a genetic algorithm.

```
#include "operators/Crossover/PureSinglePointCrossover.h"
#include <cstdlib>
```

Include dependency graph for PureSinglePointCrossover.cpp:



13.48.1 Detailed Description

Defines the PureSinglePointCrossover class for performing single-point crossover in a genetic algorithm.
The PureSinglePointCrossover class provides methods to perform a single-point crossover on two parent individuals to generate an offspring individual. This class is designed to be used in genetic algorithms to combine genetic material from two parents to create new individuals, which can then be used in subsequent generations of the algorithm.

Date

Created on May 20, 2024

Authors

ACS Gerardium Rush - Pentlandite:

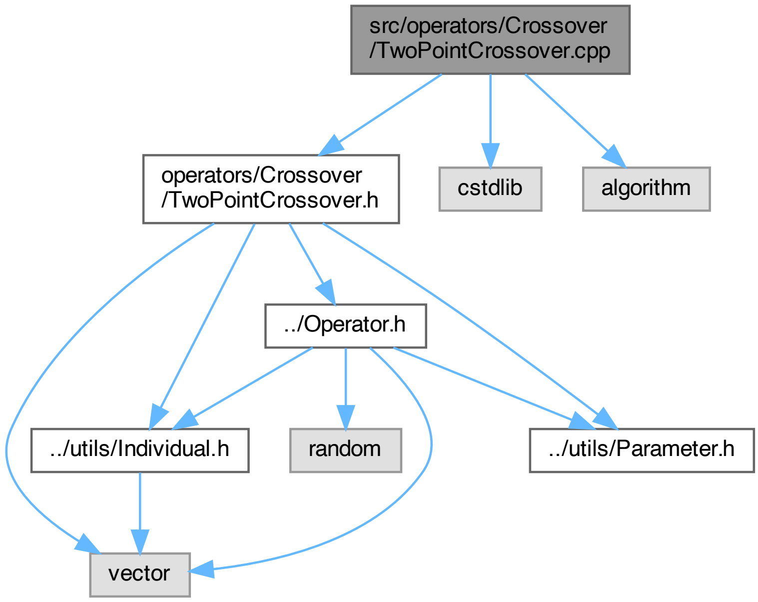
- Alex N Njeumi
- Geyu Ji
- Melissa Y S Sim
- Mingsheng Cai
- Peifeng Tan
- Yongwen Chen
- Zihan Li

13.49 src/operators/Crossover/TwoPointCrossover.cpp File Reference

Defines the TwoPointCrossover class for performing two-point crossover in a genetic algorithm.

```
#include "operators/Crossover/TwoPointCrossover.h"
#include <cstdlib>
```

`#include <algorithm>`
Include dependency graph for TwoPointCrossover.cpp:



13.49.1 Detailed Description

Defines the TwoPointCrossover class for performing two-point crossover in a genetic algorithm.
The TwoPointCrossover class provides methods to perform a two-point crossover on two parent individuals to generate an offspring individual. This class is designed to be used in genetic algorithms to combine genetic material from two parents to create new individuals, which can then be used in subsequent generations of the algorithm.

Date

Created on May 20, 2024

Authors

ACS Gerardium Rush - Pentlandite:

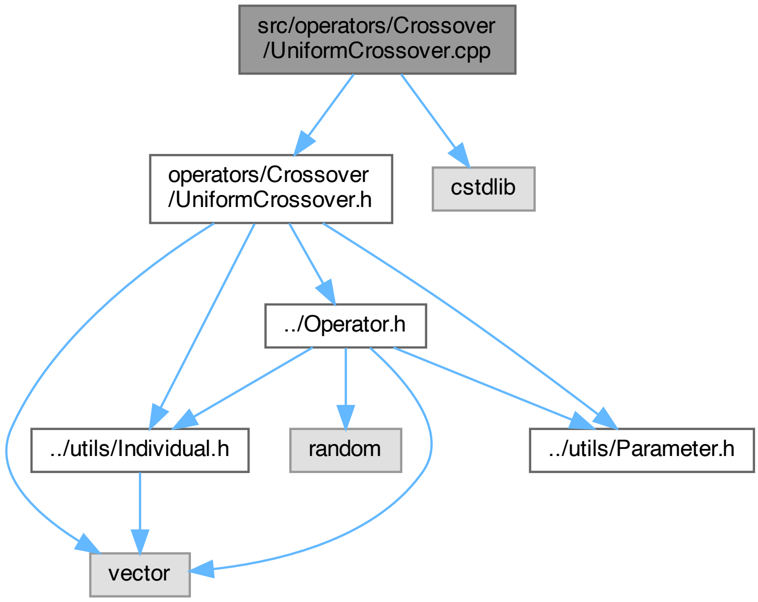
- Alex N Njeumi
- Geyu Ji
- Melissa Y S Sim
- Mingsheng Cai
- Peifeng Tan
- Yongwen Chen
- Zihan Li

13.50 src/operators/Crossover/UniformCrossover.cpp File Reference

Defines the UniformCrossover class for performing uniform crossover in a genetic algorithm.

```
#include "operators/Crossover/UniformCrossover.h"
#include <cstdlib>
```

Include dependency graph for UniformCrossover.cpp:



13.50.1 Detailed Description

Defines the UniformCrossover class for performing uniform crossover in a genetic algorithm.
The UniformCrossover class provides methods to perform a uniform crossover on two parent individuals to generate an offspring individual. This class is designed to be used in genetic algorithms to combine genetic material from two parents to create new individuals, which can then be used in subsequent generations of the algorithm.

Date

Created on May 20, 2024

Authors

ACS Gerardium Rush - Pentlandite:

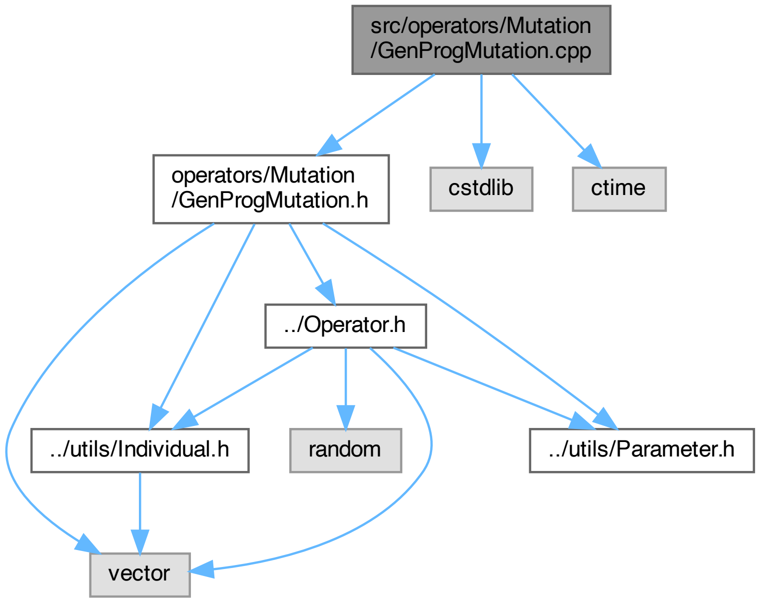
- Alex N Njeumi
- Geyu Ji
- Melissa Y S Sim
- Mingsheng Cai
- Peifeng Tan
- Yongwen Chen
- Zihan Li

13.51 src/operators/Mutation/GenProgMutation.cpp File Reference

Defines the GenProgMutation class for performing mutation operations in a genetic programming algorithm.

```
#include "operators/Mutation/GenProgMutation.h"
#include <cstdlib>
#include <ctime>
```

Include dependency graph for GenProgMutation.cpp:



13.51.1 Detailed Description

Defines the GenProgMutation class for performing mutation operations in a genetic programming algorithm.
The GenProgMutation class provides methods to perform mutation operations on individuals in a genetic programming algorithm. This class is designed to introduce variations into the population by modifying individuals' genetic material, which can then be used in subsequent generations of the algorithm.

Date

Created on May 20, 2024

Authors

ACS Gerardium Rush - Pentlandite:

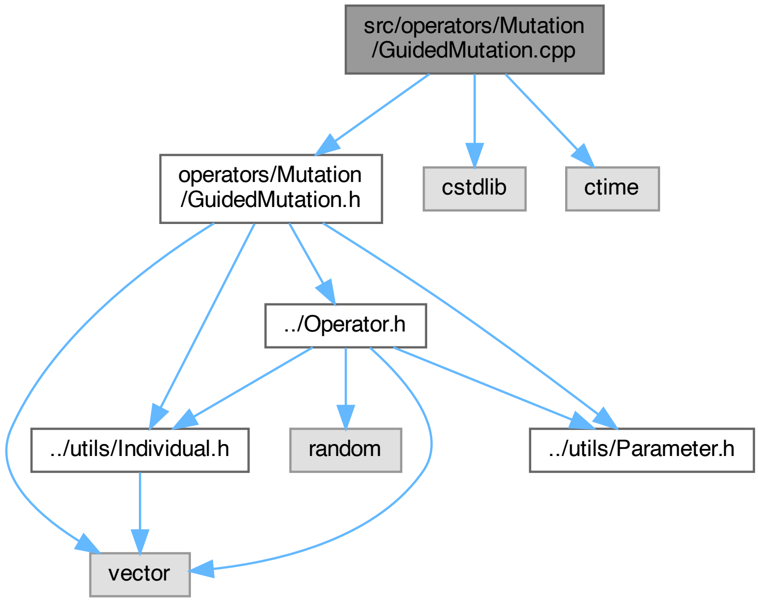
- Alex N Njeumi
- Geyu Ji
- Melissa Y S Sim
- Mingsheng Cai
- Peifeng Tan
- Yongwen Chen
- Zihan Li

13.52 src/operators/Mutation/GuidedMutation.cpp File Reference

Defines the GuidedMutation class for performing guided mutation operations in a genetic algorithm.

```
#include "operators/Mutation/GuidedMutation.h"
#include <cstdlib>
#include <ctime>
```

Include dependency graph for GuidedMutation.cpp:



13.52.1 Detailed Description

Defines the GuidedMutation class for performing guided mutation operations in a genetic algorithm.
The GuidedMutation class provides methods to perform guided mutation operations on individuals in a genetic algorithm. This class is designed to introduce targeted variations into the population by modifying individuals' genetic material based on specific rules or heuristics, which can then be used in subsequent generations of the algorithm.

Date

Created on May 20, 2024

Authors

ACS Gerardium Rush - Pentlandite:

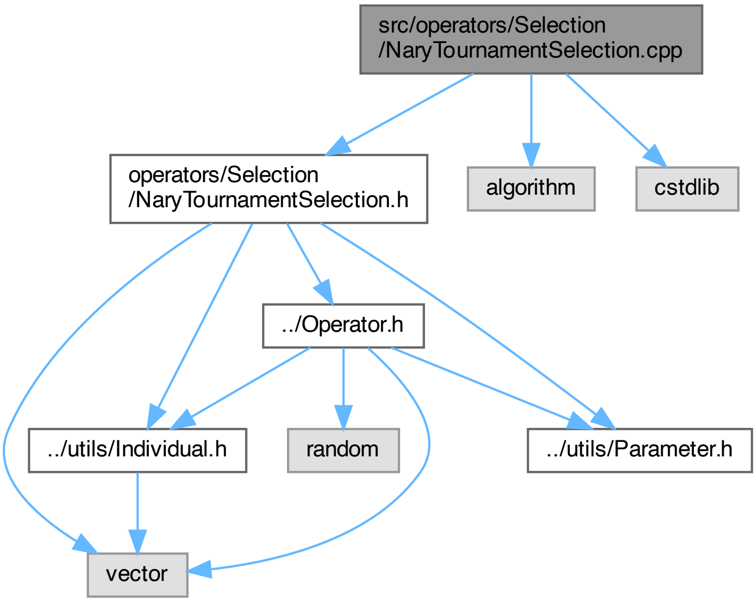
- Alex N Njeumi
- Geyu Ji
- Melissa Y S Sim
- Mingsheng Cai
- Peifeng Tan
- Yongwen Chen
- Zihan Li

13.53 src/operators/Selection/NaryTournamentSelection.cpp File Reference

Defines the NaryTournamentSelection class for performing n-ary tournament selection in a genetic algorithm.

```
#include "operators/Selection/NaryTournamentSelection.h"
#include <algorithm>
#include <cstdlib>
```

Include dependency graph for NaryTournamentSelection.cpp:



13.53.1 Detailed Description

Defines the NaryTournamentSelection class for performing n-ary tournament selection in a genetic algorithm.

The NaryTournamentSelection class provides methods to perform n-ary tournament selection on a population of individuals to select individuals for the next generation. This class is designed to be used in genetic algorithms to select the fittest individuals based on tournament competition among a subset of the population.

Date

Created on May 20, 2024

Authors

ACS Gerardium Rush - Pentlandite:

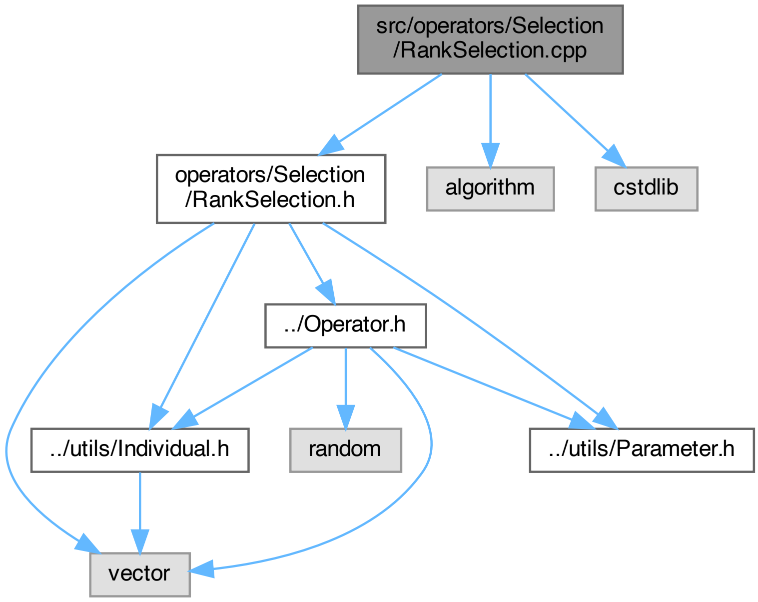
- Alex N Njeumi
- Geyu Ji
- Melissa Y S Sim
- Mingsheng Cai
- Peifeng Tan
- Yongwen Chen
- Zihan Li

13.54 src/operators/Selection/RankSelection.cpp File Reference

Defines the RankSelection class for performing rank-based selection in a genetic algorithm.

```
#include "operators/Selection/RankSelection.h"
#include <algorithm>
#include <cstdlib>
```

Include dependency graph for RankSelection.cpp:



13.54.1 Detailed Description

Defines the RankSelection class for performing rank-based selection in a genetic algorithm.

The RankSelection class provides methods to perform rank-based selection on a population of individuals to select individuals for the next generation. This class is designed to be used in genetic algorithms to select the fittest individuals based on their rank within the population.

Date

Created on May 21, 2024

Authors

ACS Gerardium Rush - Pentlandite:

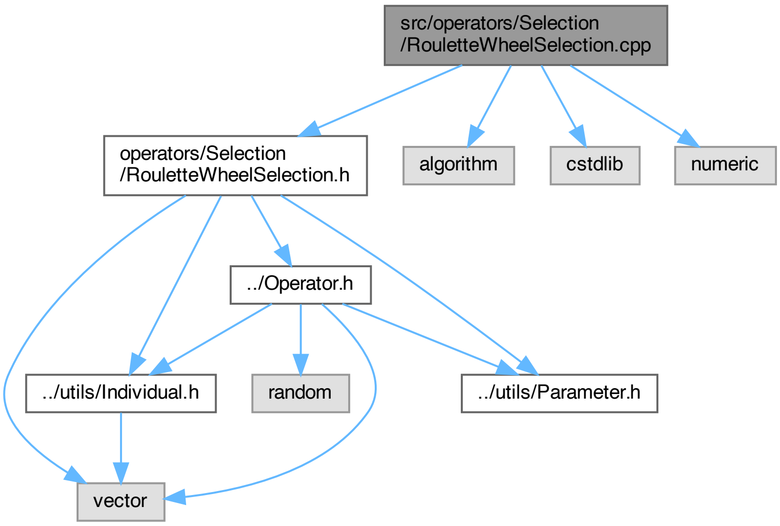
- Alex N Njeumi
- Geyu Ji
- Melissa Y S Sim
- Mingsheng Cai
- Peifeng Tan
- Yongwen Chen
- Zihan Li

13.55 src/operators/Selection/RouletteWheelSelection.cpp File Reference

Defines the RouletteWheelSelection class for performing roulette wheel selection in a genetic algorithm.

```
#include "operators/Selection/RouletteWheelSelection.h"
#include <algorithm>
#include <cstdlib>
#include <numeric>
```

Include dependency graph for RouletteWheelSelection.cpp:



13.55.1 Detailed Description

Defines the RouletteWheelSelection class for performing roulette wheel selection in a genetic algorithm.

The RouletteWheelSelection class provides methods to perform roulette wheel selection on a population of individuals to select individuals for the next generation. This class is designed to be used in genetic algorithms to select individuals based on their fitness proportionally.

Date

Created on May 21, 2024

Authors

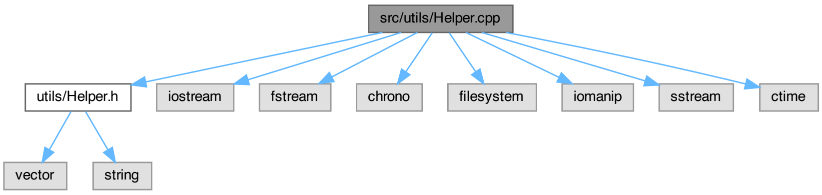
ACS Gerardium Rush - Pentlandite:

- Alex N Njeumi
- Geyu Ji
- Melissa Y S Sim
- Mingsheng Cai
- Peifeng Tan
- Yongwen Chen
- Zihan Li

13.56 src/utils/Helper.cpp File Reference

```
#include "utils/Helper.h"
#include <iostream>
#include <fstream>
#include <chrono>
#include <filesystem>
#include <iomanip>
#include <sstream>
#include <ctime>
```

Include dependency graph for Helper.cpp:



Functions

- std::string get_current_time_str ()

    Retrieves the current time as a formatted string.
- void log_results (const std::vector< int > &vector, double elapsed_time, double performance, double recovery, double grade)

    Logs the results of the genetic algorithm optimization to a file.

13.56.1 Function Documentation

13.56.1.1 get_current_time_str()

```
std::string get_current_time_str ( )
```

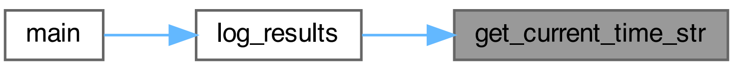
Retrieves the current time as a formatted string.

This function returns the current time formatted as "YYYY-MM-DD_HH-MM-SS".

Returns

A string representing the current time.

Here is the caller graph for this function:



13.56.1.2 log_results()

```
void log_results (
            const std::vector< int > & vector,
```

double *elapsed_time*,
double *performance*,
double *recovery*,
double *grade*)

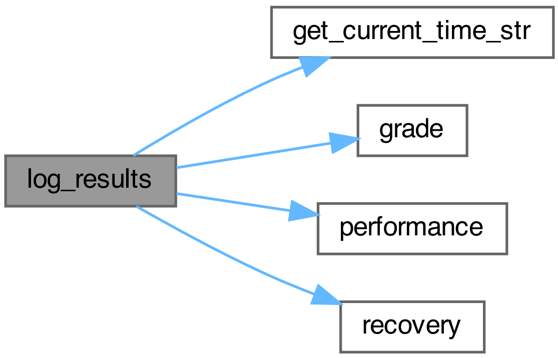
Logs the results of the genetic algorithm optimization to a file.

This function creates a new log folder with the current timestamp, and saves the performance metrics, recovery, grade, and the final configuration of the optimized circuit to a log file.

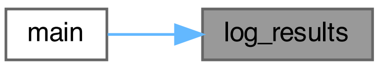
Parameters

vector	The final configuration vector of the optimized circuit.
elapsed_time	The time taken to complete the optimization process, in seconds.
performance	The performance metric of the optimized circuit.
recovery	The recovery metric of the optimized circuit.
grade	The grade metric of the optimized circuit.

Here is the call graph for this function:



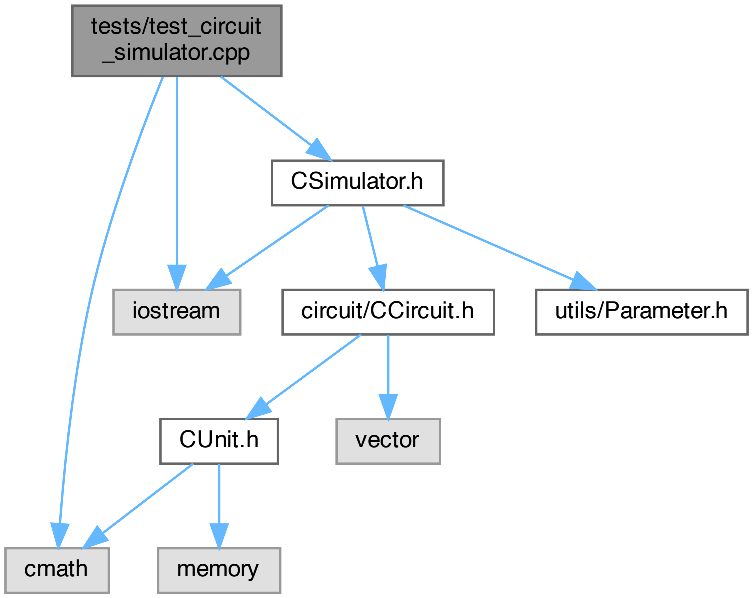
Here is the caller graph for this function:



13.57 tests/test_circuit_simulator.cpp File Reference

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

Include dependency graph for test_circuit_simulator.cpp:



Functions

- int main (int argc, char ∗argv[ ])
  Main function to run the test cases for evaluating circuit configurations.

13.57.1 Function Documentation

13.57.1.1 main()

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

Main function to run the test cases for evaluating circuit configurations.
This function initializes several circuit vectors and uses the Evaluate_Circuit function to check their performance, recovery, and grade. The results are compared against expected values, and the function prints whether each test passes or fails.

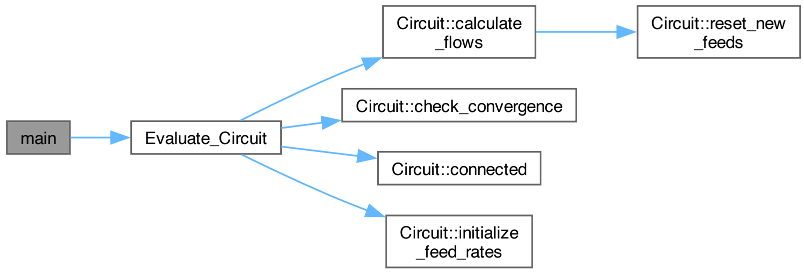
Parameters

argc	Number of command-line arguments.
argv	Array of command-line arguments.

Returns

int Returns 0 if all tests pass, otherwise returns 1.

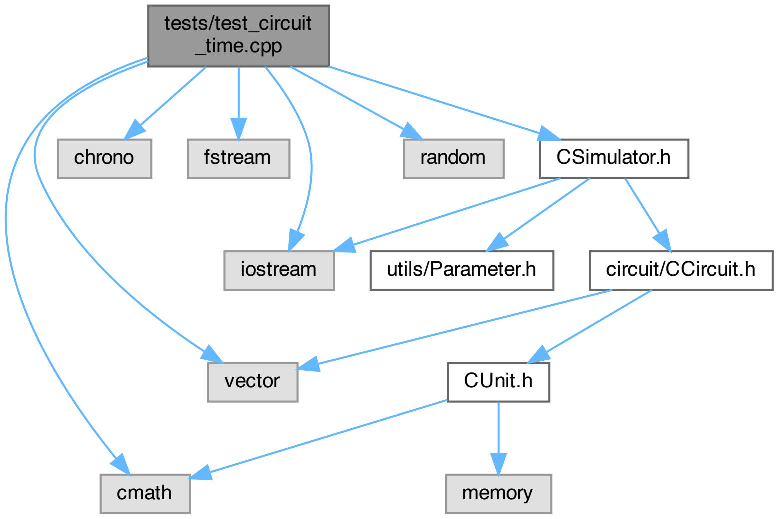
Here is the call graph for this function:



13.58 tests/test_circuit_time.cpp File Reference

```
#include <cmath>
#include <iostream>
#include <chrono>
#include <fstream>
#include <vector>
#include <random>
#include "CSimulator.h"
```

Include dependency graph for test_circuit_time.cpp:



Functions

- std::vector< int > generate_circuit_vector (int n)

  Generates a circuit vector of a given size.

- int main ()

  Main function to evaluate execution times of the Evaluate_Circuit function with different circuit vector sizes.

13.58.1 Function Documentation

13.58.1.1 generate_circuit_vector()

```
std::vector< int > generate_circuit_vector (
            int n )
```

Generates a circuit vector of a given size.

This function generates a random circuit vector with the specified number of units. The first element is set to a valid feed unit, and the remaining elements are set to random unit indices.

Parameters

n	The number of units in the circuit.

Returns

std::vector<int> The generated circuit vector.

Here is the caller graph for this function:

13.58.1.2 main()

```
int main ( )
```

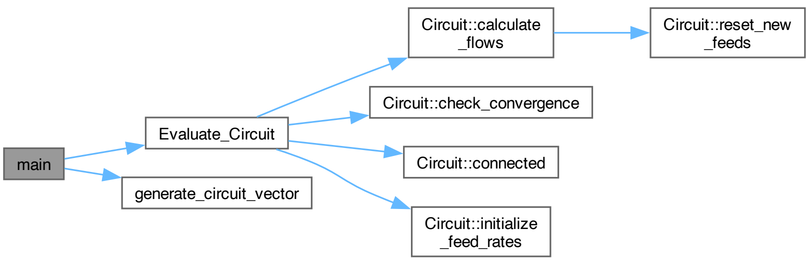
Main function to evaluate execution times of the Evaluate_Circuit function with different circuit vector sizes.

This function generates circuit vectors of various sizes, evaluates them using the Evaluate_Circuit function, and records the execution times. The results are printed to the console and also saved to an output file.

Returns

int Returns 0 if the program runs successfully, otherwise returns 1.

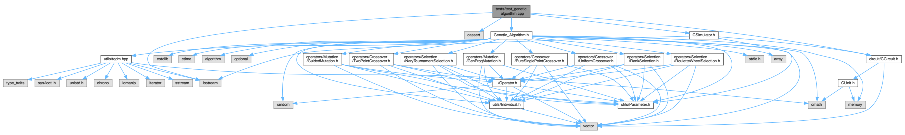
Here is the call graph for this function:



13.59 tests/test_genetic_algorithm.cpp File Reference

```
#include <iostream>
#include <cassert>
#include "Genetic_Algorithm.h"
#include "circuit/CCircuit.h"
#include "CSimulator.h"
```

Include dependency graph for test_genetic_algorithm.cpp:



Functions

- double test_function (int vector_size, int ∗vector, CircuitParameters pams)

    Test function for the genetic algorithm.
- double test_function_long (int vector_size, int ∗vector, CircuitParameters pams)

    Test function for the genetic algorithm with a long vector.
- bool Check_Validity (int vector_size, int ∗circuit_vector)

    Function to check the validity of a circuit vector.
- bool areArraysEqual (const int ∗a, const int ∗b, size_t size)

    Function to check if two arrays are equal.
- void test_genetic_algorithm ()

    Test the genetic algorithm for various cases.
- void test_initializer ()

    Test the initialization of the population.
- void test_crossover ()

    Test the crossover operators.
- void test_mutation ()

    Test the mutation operators.
- void test_selection ()

    Test the selection operators.
- void test_operators ()

    Test the elitism operator.
- void test_elitism ()

    Test the elitism operator.
- void test_SA ()

    Test the simulated annealing operator.
- int main ()

    Main function to run all the tests.

Variables

- int test_answer [10] = {2, 1, 1, 2, 0, 2, 3, 0, 4, 4}
- int test_answer1 [31]

13.59.1 Function Documentation

13.59.1.1 areArraysEqual()

```
bool areArraysEqual (
            const int * a,
            const int * b,
            size_t size )
```

Function to check if two arrays are equal.

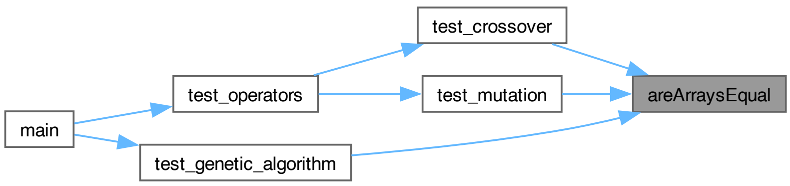
This function checks if two arrays are equal by comparing each element in the arrays.

Parameters

a	Check if this array is equal to b
b	Check if this array is equal to a
size	Size of the arrays

Returns

Here is the caller graph for this function:



13.59.1.2 Check_Validity()

```
bool Check_Validity (
            int vector_size,
            int * circuit_vector )
```

Function to check the validity of a circuit vector.

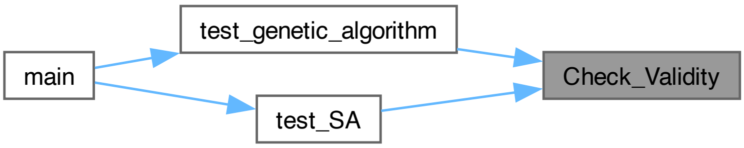
This function checks the validity of a circuit vector by ensuring that the feed ID and destination ID are within the range of the number of units in the circuit.

Parameters

vector_size	Size of the vector
circuit_vector	The circuit vector to be checked

Returns

Here is the caller graph for this function:



13.59.1.3 main()

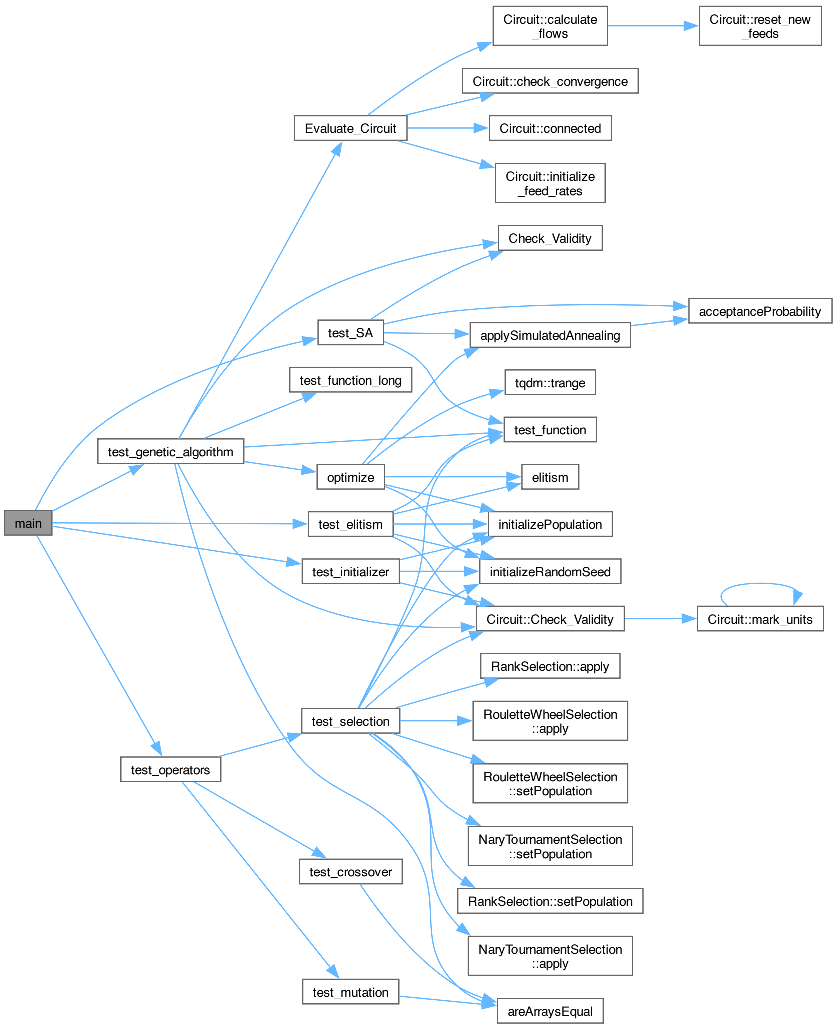
`int main ( )`

Main function to run all the tests.

This function runs all the tests for the Genetic Algorithm module.

Returns

Here is the call graph for this function:

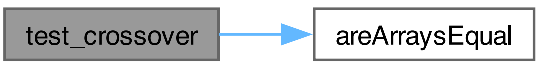


13.59.1.4 test_crossover()

```
void test_crossover ( )
```

Test the crossover operators.

This function tests various crossover operators including two-point, pure single-point, and uniform crossover. Here is the call graph for this function:



Here is the caller graph for this function:

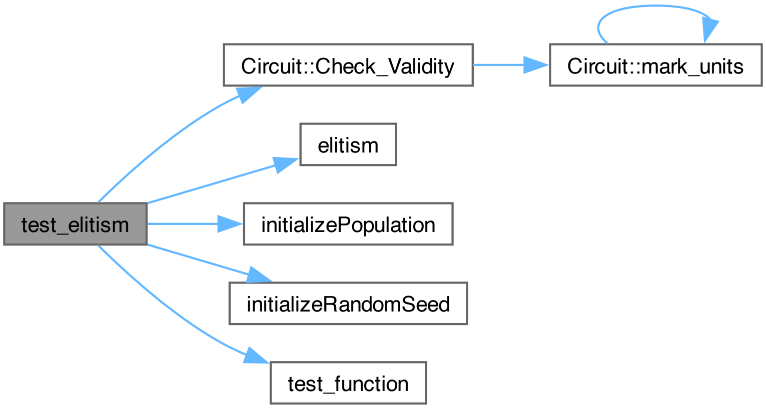
13.59.1.5 test_elitism()

```
void test_elitism ( )
```

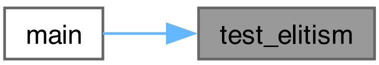
Test the elitism operator.

This function tests the elitism operator by checking if the performance of the population is improved after elitism.

Here is the call graph for this function:



Here is the caller graph for this function:



13.59.1.6 test_function()

```
double test_function (
            int vector_size,
            int * vector,
            CircuitParameters pams )
```

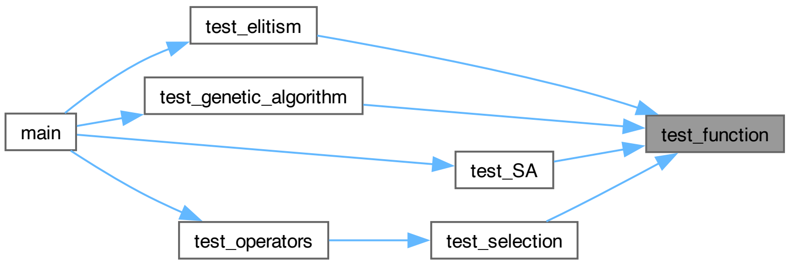
Test function for the genetic algorithm.
This function is used as the performance function for the genetic algorithm. It calculates the difference between the given vector and the test answer vector, and returns the negative of the sum of the squared differences.

Parameters

vector_size	Size of the vector
vector	The vector to be evaluated
pams	Circuit parameters

Returns

Here is the caller graph for this function:



13.59.1.7 test_function_long()

```
double test_function_long (
            int vector_size,
            int * vector,
            CircuitParameters pams )
```

Test function for the genetic algorithm with a long vector.
This function is used as the performance function for the genetic algorithm. It calculates the difference between the given vector and the test answer vector, and returns the negative of the sum of the squared differences.

Parameters

vector_size	Size of the vector
vector	The vector to be evaluated
pams	Circuit parameters

Returns

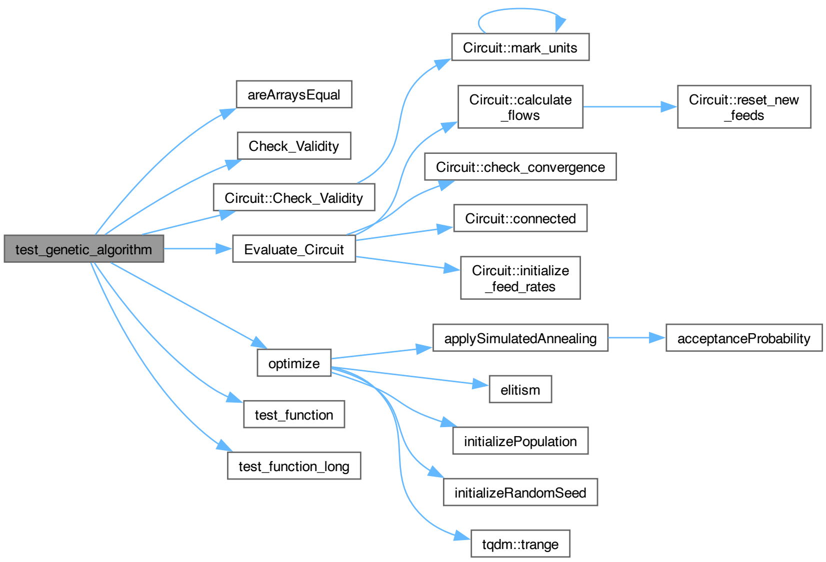
Here is the caller graph for this function:

13.59.1.8 test_genetic_algorithm()

```
void test_genetic_algorithm ( )
```

Test the genetic algorithm for various cases.

This function sets up the parameters and runs the genetic algorithm on different test cases. It checks the optimized results against expected values. Here is the call graph for this function:



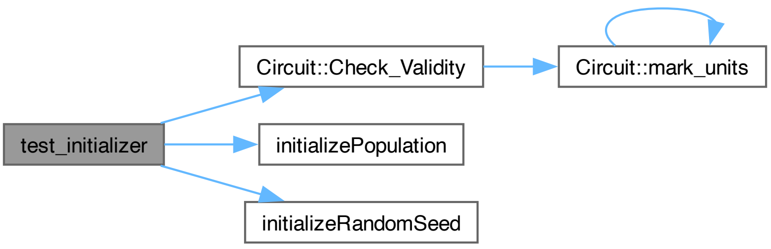
Here is the caller graph for this function:

13.59.1.9 test_initializer()

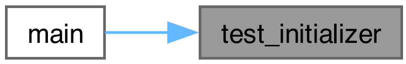
```
void test_initializer ( )
```

Test the initialization of the population.

This function tests if the population is correctly initialized with valid individuals and the correct size. Here is the call graph for this function:



Here is the caller graph for this function:

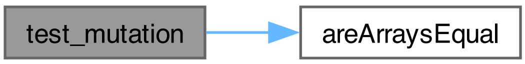


13.59.1.10 test_mutation()

```
void test_mutation ( )
```

Test the mutation operators.

This function tests various mutation operators including guided mutation and genprog mutation. Here is the call graph for this function:



Here is the caller graph for this function:

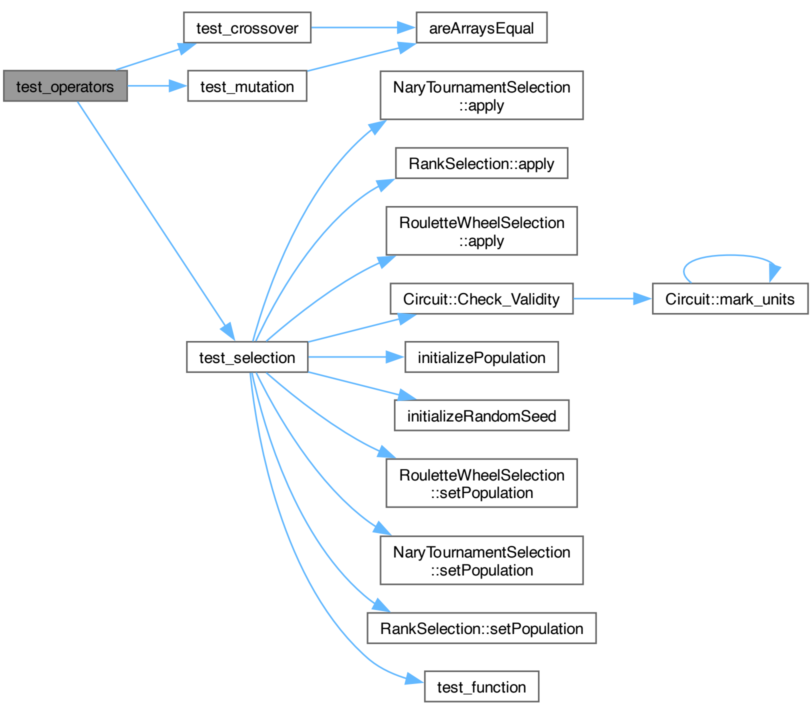
13.59.1.11 test_operators()

`void test_operators ( )`

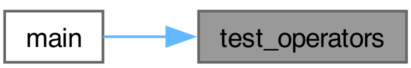
Test the elitism operator.

This function tests the elitism operator by checking if the performance of the population is improved after elitism.

Here is the call graph for this function:



Here is the caller graph for this function:

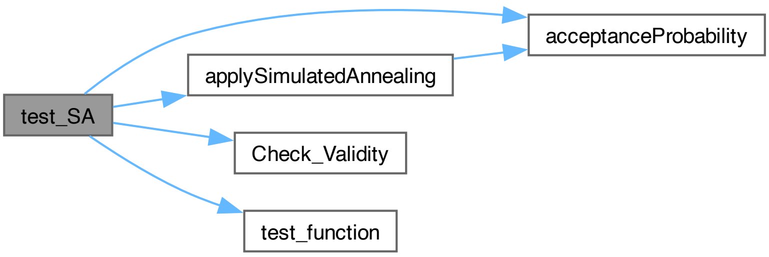


13.59.1.12 test_SA()

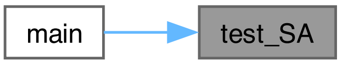
```
void test_SA ( )
```

Test the simulated annealing operator.

This function tests the simulated annealing operator by checking if the offspring is accepted when it is better, and if the temperature decreases correctly. Here is the call graph for this function:



Here is the caller graph for this function:

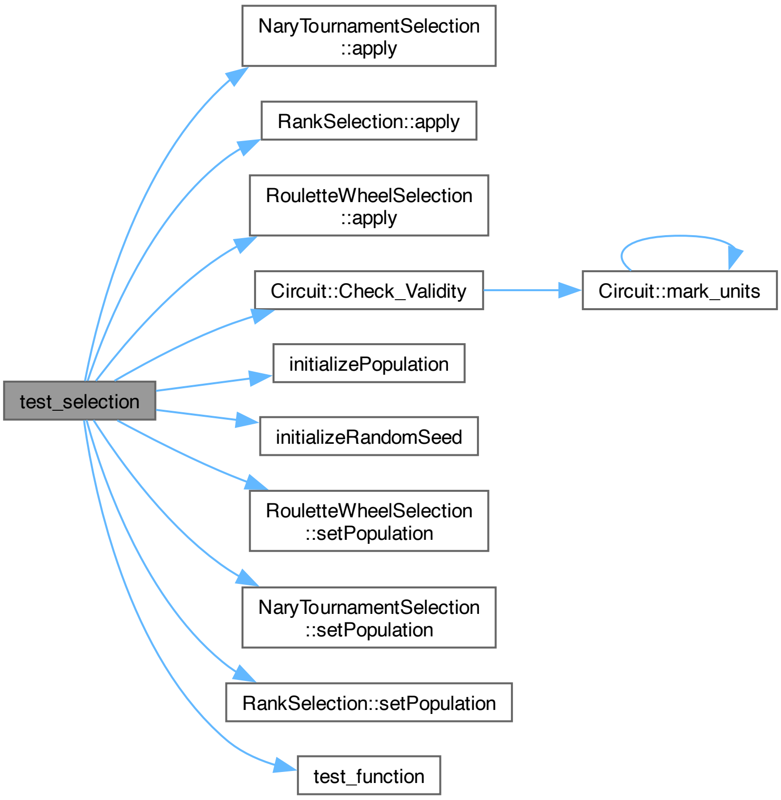


13.59.1.13 test_selection()

```
void test_selection ( )
```

Test the selection operators.

This function tests various selection operators including n-ary tournament, rank, and roulette wheel selection. Here is the call graph for this function:



Here is the caller graph for this function:

13.59.2 Variable Documentation

13.59.2.1 test_answer

```
int test_answer[10] = {2, 1, 1, 2, 0, 2, 3, 0, 4, 4}
```

13.59.2.2 test_answer1

```
int test_answer1[31]
```

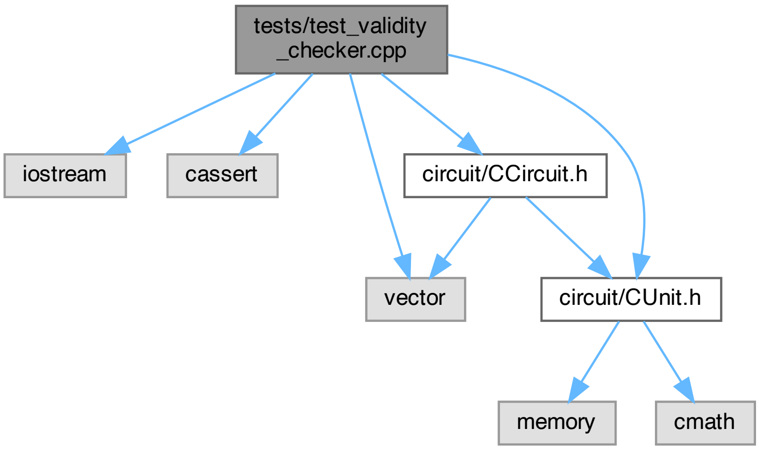
Initial value:

```
= {0, 10, 1, 11, 10, 2, 11, 10, 3, 11, 10, 4, 11, 10, 5, 11, 10, 6, 11, 10, 7, 11, 10, 8, 11, 10,
                        9, 11, 10, 0, 11}
```

13.60 tests/test_validity_checker.cpp File Reference

```
#include <iostream>
#include <cassert>
#include <vector>
#include "circuit/CUnit.h"
#include "circuit/CCircuit.h"
```

Include dependency graph for test_validity_checker.cpp:



Functions

- void test_check_validity ()

    Test function for checking the validity of circuit configurations.
- int main (int argc, char ∗argv[ ])

    Main function to run the circuit validity tests.

13.60.1 Function Documentation

13.60.1.1 main()

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

Main function to run the circuit validity tests.

This function runs the test_check_validity function which includes various test cases for checking the validity of circuit configurations.

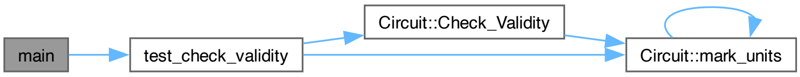
Parameters

argc	Number of command line arguments
argv	Array of command line arguments

Returns

Exit status of the program

Here is the call graph for this function:

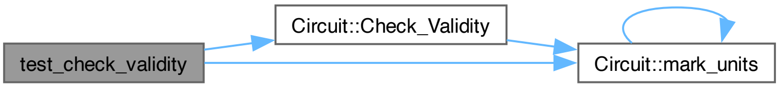


13.60.1.2 test_check_validity()

```
void test_check_validity ( )
```

Test function for checking the validity of circuit configurations.

This function runs various test cases to ensure that the circuit validity checks in the CCircuit class are functioning correctly. It includes tests for valid circuits, circuits with incorrect vector sizes, self-recycling circuits, negative feed/destination IDs, and other invalid circuit configurations. Here is the call graph for this function:



Here is the caller graph for this function:

Index