Gerardium Rush: Pentlandite

Generated by Doxygen 1.9.7

Chapter 1

Class Index

1.1 Class List

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

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

Class Documentation

3.1 Circuit Class Reference

Circuit class.

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

Public Member Functions

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

  Construct a new Circuit:: Circuit object.
- ∼Circuit ()

  Destroy the Circuit object.
- void toFile ()

  To write the circuit information to a file.

Static Public Member Functions

- static bool Check_Validity (const std::vector< int > &circuit_vector)

  Function to check the validity of the circuit vector.

Public Attributes

- int num_units
- int feeder
- std::vector< int > circuit_vector
- std::vector< CUnit > units
- int it

3.1.1 Detailed Description

Circuit class.

This class is used to store the circuit information

Author

acse-sm222

3.1.2 Constructor & Destructor Documentation

3.1.2.1 Circuit()

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

Construct a new Circuit:: Circuit object.

Constructor taking the circuit vector

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

Parameters

circuit_vector	

```
{
  this->num_units = (circuit_vector.size() - 1) / 2 + 2;
  this->units.resize(this->num_units);
  this->circuit_vector = circuit_vector;
  this->feeder = circuit_vector[0];
  this->it = 0;

  // Initialize the units with unit ID and destinations
  for (int i = 0; i < this->num_units - 2; ++i)
  {
    this->units[i] = CUnit(i, circuit_vector[2 * i + 1], circuit_vector[2 * i + 2]);
  }
}
```

3.1.2.2 ∼Circuit()

```
Circuit::∼Circuit ( )
```

Destroy the Circuit object.

Destructor

```
{
}
```

3.1.3 Member Function Documentation

3.1.3.1 Check_Validity()

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

Function to check the validity of the circuit vector.

Function to check the validity of the circuit vector

Parameters

circuit_vector	

Returns

true

false TODO: Add more validity checks

```
{
  if (circuit_vector.size() % 2 != 1)
  {
    return false;
  }
  else
  {
    int feeder = circuit_vector[0];
    int num_units = (circuit_vector.size() - 1) / 2;

    // 0. No units conc should go directly to tails and vice versa
    for (int i = 0; i < num_units; ++i)
    {
      if (circuit_vector[2 * i + 1] == num_units + 1 || circuit_vector[2 * i + 2] == num_units)
      {
        return false;
      }
    }

    // 1. No self loops
    for (int i = 0; i < num_units; ++i)
    {
      if (circuit_vector[2 * i + 1] == i || circuit_vector[2 * i + 2] == i)
      {
        return false;
      }
    }

    // 2. Talilings and Concentrations don't have same destination
    for (int i = 0; i < num_units; ++i)
    {
      if (circuit_vector[2 * i + 1] == circuit_vector[2 * i + 2])
      {
        return false;
      }
    }

    // 3. Everynode has a feed - Each node must lie on atleast one of the graphs from the feeder
    std::vector<int> hasfeed(num_units + 2, 0);
    hasfeed[feeder] += 1;
    for (int i = 0; i < num_units; ++i)
    {
      hasfeed[circuit_vector[2 * i + 1]] += 1;
      hasfeed[circuit_vector[2 * i + 2]] += 1;
    }

    for (int i = 0; i < num_units + 2; ++i)
    {
      if (hasfeed[i] == 0)
      {
        return false;
      }
    }

    // 4. Every unit has a route forward to leaf nodes on BOTH graphs
    for (int i = 0; i < num_units; ++i)
    {
      std::vector<Units> myunits(num_units + 2);
      for (int j = 0; j < num_units; ++j)
      {
        myunits[j].conc_num = circuit_vector[2 * j + 1];
        myunits[j].tails_num = circuit_vector[2 * j + 2];
      }
      myunits[num_units].conc_num = 1e5;
      myunits[num_units + 1].tails_num = 1e5;
      markunits(myunits, i);
      if (myunits[num_units].marked == false || myunits[num_units + 1].marked == false)
      {
        return false;
      }
    }
    return true;
  }
}
```

3.1.3.2 toFile()

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

To write the circuit information to a file.

Function to write the circuit information to a file

This function writes the circuit information to a file. It creates a DirectedGraph object for the concentrate and tails and writes the graph to a file

```
{
  DirectedGraph Conc(num_units);
  DirectedGraph Tail(num_units);
  for (int i = 0; i < num_units - 2; ++i)
  {
    Conc.addEdge(i, circuit_vector[2 * i + 1]);
    Tail.addEdge(i, circuit_vector[2 * i + 2]);
  }

  Conc.addEdge(num_units - 2, num_units + 1);
  Conc.addEdge(num_units - 1, num_units + 1);
  Tail.addEdge(num_units - 2, num_units + 1);
  Tail.addEdge(num_units - 1, num_units + 1);

  Circuit_Parameters params;
  Calculate_Circuit(*this, params);

  std::fstream ConcFile;
  std::fstream TailFile;
  std::fstream DataFile;
  ConcFile.open("C.txt", std::ios::out);
  TailFile.open("T.txt", std::ios::out);
  DataFile.open("D.txt", std::ios::out);

  ConcFile << Conc;
  ConcFile << feeder << std::endl;
  TailFile << Tail;
  TailFile << feeder << std::endl;

  for (int i = 0; i < num_units - 2; ++i)
  {
    DataFile << i << " " << units[i].conc.ger << " " << units[i].conc.waste << " " << units[i].tail.ger << " "
  << units[i].tail.waste << std::endl;
  }

  ConcFile.close();
  TailFile.close();
  DataFile.close();
}
```

3.1.4 Member Data Documentation

3.1.4.1 circuit_vector

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

Circuit vector

3.1.4.2 feeder

`int Circuit::feeder`

Feeder unit

3.1.4.3 it

`int Circuit::it`

Number of iterations

3.1.4.4 num_units

```
int Circuit::num_units
```

Number of units in the circuit

3.1.4.5 units

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

Vector of Cunits

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

- CCircuit.h
- CCircuit.cpp

3.2 Circuit_Parameters Struct Reference

A Structure to store the parameters for the circuit simulator.

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

Public Attributes

- double tolerance = 1e-6
- int max_iterations = 1000
- double input_Fger = 10.0
- double input_Fwaste = 100.0
- double get_money = 100.0
- double lose_money = 500.0

3.2.1 Detailed Description

A Structure to store the parameters for the circuit simulator.

3.2.2 Member Data Documentation

3.2.2.1 get_money

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

Money earned per unit of mineral

3.2.2.2 input_Fger

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

Input flow rate of the mineral

3.2.2.3 input_Fwaste

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

Input flow rate of the waste

3.2.2.4 lose_money

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

Money lost per unit of waste

3.2.2.5 max_iterations

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

Maximum number of iterations

3.2.2.6 tolerance

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

Tolerance for simulator convergence

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

- CCircuit.h

3.3 CircuitOptimizer Struct Reference

Public Attributes

- int feeder = 0
- int num_units = 5

3.3.1 Member Data Documentation

3.3.1.1 feeder

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

3.3.1.2 num_units

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

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

- main.cpp

3.4 Crossover< T > Class Template Reference

Performs crossover operation between two parent individuals.

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

Static Public Member Functions

- static void one_pt_crossover (Individual< T > &parent1, Individual< T > &parent2, Individual< T > &child1, Individual< T > &child2)
- static void two_pts_crossover (Individual< T > &parent1, Individual< T > &parent2, Individual< T > &child1, Individual< T > &child2)
- static void uniform_crossover (Individual< T > &parent1, Individual< T > &parent2, Individual< T > &child1, Individual< T > &child2)
- static void mult_pts_crossover (Individual< T > &parent1, Individual< T > &parent2, Individual< T > &child1, Individual< T > &child2)
- static void crossover (Individual< T > &parent1, Individual< T > &parent2, Individual< T > &child1, Individual< T > &child2, CrossoverType type)

3.4.1 Detailed Description

template<typename T>
class Crossover< T >

Performs crossover operation between two parent individuals.

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

Parameters

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

3.4.2 Member Function Documentation

3.4.2.1 crossover()

```
template<typename T >
void Crossover< T >::crossover (
            Individual< T > & parent1,
            Individual< T > & parent2,
            Individual< T > & child1,
            Individual< T > & child2,
            CrossoverType type ) [static]
```

```
{
    switch (type)
    {

    case CrossoverType::ONE_PT:
        one_pt_crossover(parent1, parent2, child1, child2);
        break;

    case CrossoverType::TWO_PTS:
        two_pts_crossover(parent1, parent2, child1, child2);
        break;

    case CrossoverType::MULT_PTS:
        mult_pts_crossover(parent1, parent2, child1, child2);
        break;

    case CrossoverType::UNIFORM:
        uniform_crossover(parent1, parent2, child1, child2);
        break;
    default:
        one_pt_crossover(parent1, parent2, child1, child2);
    };
}
```

3.4.2.2 mult_pts_crossover()

```
template<typename T >
void Crossover< T >::mult_pts_crossover (
            Individual< T > & parent1,
            Individual< T > & parent2,
            Individual< T > & child1,
            Individual< T > & child2 ) [static]
```

```
{
    int chromosomeLength = parent1.chromosome.size();
    std::uniform_int_distribution<> distr(0, parent1.chromosome.size() - 1);

    // Generate a number of crossover points
    int numCrossoverPoints = 4;

    // Generate crossover points
    std::set<int> crossoverPoints;
    while (crossoverPoints.size() < numCrossoverPoints)
    {
        crossoverPoints.insert(distr(gen));
    }
    crossoverPoints.insert(chromosomeLength);

    // Perform crossover
    bool sw = false;
    auto startIter = crossoverPoints.begin();
    for (int i = 0; i < chromosomeLength; ++i)
    {
        // If i is in crossoverPoints, switch genes
        if (i == *startIter)
        {
            sw = !sw;
            ++startIter;
        }
        if (!sw)
        {
            child1.chromosome[i] = parent1.chromosome[i];
            child2.chromosome[i] = parent2.chromosome[i];
```

```
        }
        else
        {
            child1.chromosome[i] = parent2.chromosome[i];
            child2.chromosome[i] = parent1.chromosome[i];
        }
    }
};
```

3.4.2.3 one_pt_crossover()

```
template<typename T >
void Crossover< T >::one_pt_crossover (
            Individual< T > & parent1,
            Individual< T > & parent2,
            Individual< T > & child1,
            Individual< T > & child2 )  [static]
{
    std::uniform_int_distribution<> distr(1, parent1.chromosome.size() - 1);

    int crossoverPoint = distr(gen);
    for (int i = 0; i < parent1.chromosome.size(); ++i)
    {
        if (i < crossoverPoint)
        {
            child1.chromosome[i] = parent1.chromosome[i];
            child2.chromosome[i] = parent2.chromosome[i];
        }
        else
        {
            child1.chromosome[i] = parent2.chromosome[i];
            child2.chromosome[i] = parent1.chromosome[i];
        }
    }
}
```

3.4.2.4 two_pts_crossover()

```
template<typename T >
void Crossover< T >::two_pts_crossover (
            Individual< T > & parent1,
            Individual< T > & parent2,
            Individual< T > & child1,
            Individual< T > & child2 )  [static]
{
    // Define two crossover points
    std::uniform_int_distribution<> distr(0, parent1.chromosome.size() - 1);

    int crossoverPoint1 = distr(gen);
    int crossoverPoint2 = distr(gen);

    while (crossoverPoint1 == crossoverPoint2 ||
           (crossoverPoint1 == 0 && crossoverPoint2 == parent1.chromosome.size() - 1) ||
           (crossoverPoint2 == 0 && crossoverPoint1 == parent1.chromosome.size() - 1))
    {
        crossoverPoint2 = distr(gen);
    };

    if (crossoverPoint1 > crossoverPoint2)
        std::swap(crossoverPoint1, crossoverPoint2);

    for (int i = 0; i < parent1.chromosome.size(); ++i)
    {
        if (i < crossoverPoint1 || i > crossoverPoint2)
        {
            child1.chromosome[i] = parent1.chromosome[i];
            child2.chromosome[i] = parent2.chromosome[i];
        }
        else
        {
            child1.chromosome[i] = parent2.chromosome[i];
            child2.chromosome[i] = parent1.chromosome[i];
        }
    }
};
```

3.4.2.5 uniform_crossover()

```
template<typename T >
void Crossover< T >::uniform_crossover (
             Individual< T > & parent1,
             Individual< T > & parent2,
             Individual< T > & child1,
             Individual< T > & child2 )  [static]
{
    double crossoverRate = 0.5;
    double r;
    for (int i = 0; i < parent1.chromosome.size(); ++i)
    {
        r = RandomDouble(0, 1);
        if (r < crossoverRate)
        {
            child1.chromosome[i] = parent1.chromosome[i];
            child2.chromosome[i] = parent2.chromosome[i];
        }
        else
        {
            child1.chromosome[i] = parent2.chromosome[i];
            child2.chromosome[i] = parent1.chromosome[i];
        }
    }
};
```

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

- GeneticAlgorithm.hpp
- GeneticAlgorithm.cpp

3.5 CUnit Class Reference

A Unit class to store the unit information and processing.

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

Public Member Functions

- CUnit ()

    Construct a new CUnit object.
- CUnit (int id, int conc_num, int tails_num)

    Construct a new CUnit object.
- void calculateOutputFlowRates (const Stream &INP)

    Calculate the output flow rates.
- ∼CUnit ()

Public Attributes

- int conc_num
- int tails_num
- Stream input
- Stream conc
- Stream tail
- double F
- double F_old

Private Attributes

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

3.5.1 Detailed Description

A Unit class to store the unit information and processing.

Author

acse-yl1922, acse-sm222

3.5.2 Constructor & Destructor Documentation

3.5.2.1 CUnit() [1/2]

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

Construct a new CUnit object.

Default constructor

```
{
    this->id = 0;
    this->conc_num = -1;
    this->tails_num = -1;
}
```

3.5.2.2 CUnit() [2/2]

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

Construct a new CUnit object.

Constructor taking the id, conc_num and tails_num

Parameters

id	Unit ID
conc_num	Concentrate stream number
tails_num	Tails stream number

```
{
    this->id = id;
    this->conc_num = conc_num;
    this->tails_num = tails_num;
```

```
}
```

3.5.2.3 ∼CUnit()

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

Destructor

3.5.3 Member Function Documentation

3.5.3.1 calculateOutputFlowRates()

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

Calculate the output flow rates.

Function to calculate the output flow rates

Parameters

INP	Input stream

```
{

    // Calculate Tau
    double sumFlowRate = INP.ger + INP.waste;
    if (std::abs(sumFlowRate) < 1e-10)
    {
        sumFlowRate = 1e-10; // Set sumFlowRate to the minimum flow rate
    }
    double Tau = phi * V * rho / sumFlowRate;

    // Calculate R
    double R_ger = (k_ger * Tau) / (1 + (k_ger * Tau));
    double R_waste = (k_waste * Tau) / (1 + (k_waste * Tau));

    // Calculate C
    this->conc.ger = INP.ger * R_ger;
    this->conc.waste = INP.waste * R_waste;

    // Calculate T
    this->tail.ger = INP.ger * (1 - R_ger);
    this->tail.waste = INP.waste * (1 - R_waste);
}
```

3.5.4 Member Data Documentation

3.5.4.1 conc

```
Stream CUnit::conc
```

Concentrate stream

3.5.4.2 conc_num

```
int CUnit::conc_num
```

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

3.5.4.3 F

```
double CUnit::F
```

3.5.4.4 F_old

```
double CUnit::F_old
```

Feed values

3.5.4.5 id

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

Unit id

3.5.4.6 input

```
Stream CUnit::input
```

Input stream

3.5.4.7 k_ger

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

Germanium recovery rate

3.5.4.8 k_waste

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

Waste recovery rate

3.5.4.9 phi

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

Percentage of solid content in the stream

3.5.4.10 rho

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

Density of the mineral

3.5.4.11 tail

```
Stream CUnit::tail
```

Tails stream

3.5.4.12 tails_num

```
int CUnit::tails_num
```

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

3.5.4.13 V

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

Volume of the unit

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

- CUnit.h
- CUnit.cpp

3.6 DepthFirstSearch Class Reference

Depth first search class for directed graph.

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

Public Member Functions

- DepthFirstSearch ()
- DepthFirstSearch (const DirectedGraph &G, int s)

  Construct a new Depth First Search:: Depth First Search object.
- void dfs (const DirectedGraph &G, int u)

  Perform depth first search.
- bool visited (int v) const

  Check if a vertex is visited.

Private Attributes

- std::vector< bool > marked

3.6.1 Detailed Description

Depth first search class for directed graph.

Author

acse-sm222

3.6.2 Constructor & Destructor Documentation

3.6.2.1 DepthFirstSearch() [1/2]

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

3.6.2.2 DepthFirstSearch() [2/2]

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

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

Default constructor Constructor taking the graph and the source vertex

Parameters

G	Directed graph
s	Source vertex

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

3.6.3 Member Function Documentation

3.6.3.1 dfs()

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

Perform depth first search.

Function to perform depth first search

Parameters

G	Directed graph
u	Vertex

```
{
    marked[u] = true;
    for (int v : G.getNeighbors(u))
    {
        if (!marked[v])
        {
            dfs(G, v);
        }
    }
}
```

3.6.3.2 visited()

```
bool DepthFirstSearch::visited (
            int v ) const
```

Check if a vertex is visited.

Function to check if a vertex is visited

Parameters

v	Vertex

Returns

true If the vertex is visited

false If the vertex is not visited

```
{
    return marked[v];
}
```

3.6.4 Member Data Documentation

3.6.4.1 marked

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

Vector to store the visited vertices

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

- DirectedGraph.hpp
- DirectedGraph.cpp

3.7 DirectedGraph Class Reference

A directed graph.

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

Public Member Functions

- DirectedGraph ()
- DirectedGraph (int n)

    Construct a new Directed Graph:: Directed Graph object.
- void addEdge (int u, int v)

    Add an edge to the graph.
- int getNumVertices () const

    Get the number of vertices in the graph.
- int getNumEdges () const

    Get the number of edges in the graph.
- std::vector< int > getNeighbors (int u) const

    Get the neighbors of a vertex.
- void writeToFile (std::string filename) const

    Write the graph information to a file.

Private Attributes

- std::vector< std::vector< int > > adjList

Friends

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

    Overload the << operator.

3.7.1 Detailed Description

A directed graph.

Author

acse-sm222

3.7.2 Constructor & Destructor Documentation

3.7.2.1 DirectedGraph() [1/2]

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

3.7.2.2 DirectedGraph() [2/2]

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

Construct a new Directed Graph:: Directed Graph object.

Default constructor Constructor taking the number of vertices

Parameters

n	Number of vertices

```
{
    adjList.resize(n);
}
```

3.7.3 Member Function Documentation

3.7.3.1 addEdge()

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

Add an edge to the graph.

Function to add an edge

Parameters

u	Source vertex
v	Destination vertex

```
{
    adjList[u].push_back(v);
}
```

3.7.3.2 getNeighbors()

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

Get the neighbors of a vertex.

Function to get the neighbors of a vertex

Parameters

u	Vertex

Returns

std::vector<int> Neighbors of a vertex

```
{
    return adjList[u];
}
```

3.7.3.3 getNumEdges()

```
int DirectedGraph::getNumEdges ( ) const
```

Get the number of edges in the graph.

Function to get the number of edges

Returns

int Number of edges

```
{
    int numEdges = 0;
    for (int u = 0; u < adjList.size(); u++)
    {
        numEdges += adjList[u].size();
    }
    return numEdges;
}
```

3.7.3.4 getNumVertices()

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

Get the number of vertices in the graph.

Function to get the number of vertices

Returns

int Number of vertices

```
{
    return adjList.size();
}
```

3.7.3.5 writeToFile()

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

Write the graph information to a file.

Function to write the graph information to a file

Parameters

filename	Filename

```
{
    std::ofstream file;
    file.open(filename);
    file « *this;
    file.close();
}
```

3.7.4 Friends And Related Symbol Documentation

3.7.4.1 operator<<

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

Overload the << operator.

Overload the << operator

Parameters

os	Output stream
G	Directed graph

Returns

std::ostream& Output stream

```
{
    for (int u = 0; u < G.getNumVertices(); u++)
    {
        for (int v : G.getNeighbors(u))
        {
            os « u « " " « v « "\n";
        }
    }
    return os;
}
```

3.7.5 Member Data Documentation

3.7.5.1 adjList

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

Adjacency list

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

- DirectedGraph.hpp
- DirectedGraph.cpp

3.8 GA< T > Class Template Reference

Template class representing a Genetic Algorithm (GA).

`#include <GeneticAlgorithm.hpp>`

Public Member Functions

- GA ()=default
- GA (int populationSize, int chromosomeSize, T chromomin, T chromomax, int maxGeneration, double crossoverRate, double mutationRate)

  Constructor for the GA class.
- void setValidity (const std::function< bool(const std::vector< T > &)> &validity)

  Sets a validity checker function for ensuring the validity of individuals.
- void setFitness (const std::function< double(const std::vector< T > &)> &fitness)

  Sets a fitness function for evaluating the fitness of individuals.
- Individual< T > optimize (CrossoverType type)

  Runs the GA optimization and returns the best individual found.
- ∼GA ()

  Destructor for the GA class.
- void toFile (std::string fileDir)

  Write the best individual to a file.
- void mutate (Individual< T > &child)

  Performs mutation operation on an individual.
- int selectParent (const std::vector< double > &cf)

  Selects a parent based on the cumulative fitness values.

Public Attributes

- T CHROMOMIN
- T CHROMOMAX
- int POP_SIZE
- int CHROMOSOME_SIZE
- int MAX_GENERATION
- double CROSSOVER_RATE
- double MUTATION_RATE
- int ELITISM
- int CONV_GEN
- bool verbose = false
- std::vector< Individual< T > > population

Private Member Functions

- void initPopulation ()

  Initializes the population with random individuals.
- void updatePopulation (CrossoverType type)

  Updates the population for the next generation.
- int tournamentSelection (int k)

Private Attributes

- std::vector< Individual< T > > offspring
- std::vector< double > fitness
- std::vector< double > scaledFitness
- std::vector< double > cumulativeFitness
- int cross_cnt
- int mut_cnt
- double cross_contrib
- double mut_contrib
- std::vector< double > bestFitnessHistory
- std::function< bool(const std::vector< T > &)> validity_

  Function object used for checking the validity of individuals in the Genetic Algorithm (GA).
- std::function< double(const std::vector< T > &)> fitness_

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

3.8.1 Detailed Description

template<typename T>
class GA< T >

Template class representing a Genetic Algorithm (GA).

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

Template Parameters

T	The type of values in the chromosome.

3.8.2 Constructor & Destructor Documentation

3.8.2.1 GA() [1/2]

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

3.8.2.2 GA() [2/2]

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

Constructor for the GA class.

Parameters

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

```
{
    // GA parameters
    CHROMOMIN = chromomin;             // Minimum value of discrete allele
    CHROMOMAX = chromomax;             // Maximum value of discrete allele
    POP_SIZE = populationSize;         // Population size
    CHROMOSOME_SIZE = chromosomeSize;  // Chromosome size
    MAX_GENERATION = maxGeneration;    // Maximum generation
    CROSSOVER_RATE = crossoverRate;    // Crossover rate
    MUTATION_RATE = mutationRate;      // Mutation rate
    ELITISM = 1;                       // Elitism fraction
    CONV_GEN = MAX_GENERATION;         // Convergence generation

    // GA variables
    population.resize(POP_SIZE);

    validity_ = [](const std::vector<T> &chromosome) -> bool
    { return true; };
    fitness_ = [](const std::vector<T> &chromosome) -> double
    { return 0; };
}
```

3.8.2.3 ∼GA()

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

Destructor for the GA class.

```
{};
```

3.8.3 Member Function Documentation

3.8.3.1 initPopulation()

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

Initializes the population with random individuals.

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

```
{
    fitness.resize(POP_SIZE);

#ifdef PARALLEL
#pragma omp parallel for default(none) shared(std::cout, population, fitness) schedule(static)
#endif
    for (int i = 0; i < POP_SIZE; i++)
    {
        population[i].chromosome.resize(CHROMOSOME_SIZE);
        bool valid = false;
        while (!valid)
        {
            for (int j = 0; j < CHROMOSOME_SIZE; j++)
            {
                population[i].chromosome[j] = CHROMOMIN + (CHROMOMAX - CHROMOMIN) * RandomDouble(0.0, 1.0);
            }
            valid = validity_(population[i].chromosome);
        }
        if (verbose) std::cout << "\rGenerated: " << i << " of " << POP_SIZE << std::flush;
        population[i].fitness = fitness_(population[i].chromosome);
    }
    if (verbose) std::cout << std::endl;

    std::sort(population.begin(), population.end(), std::greater<Individual<T>>());
    for (int i = 0; i < POP_SIZE; i++)
    {
        fitness[i] = population[i].fitness;
    }

    std::sort(population.begin(), population.end(), std::greater<Individual<T>>());
    std::sort(fitness.begin(), fitness.end(), std::greater<double>());
}
```

3.8.3.2 mutate()

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

Performs mutation operation on an individual.

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

Parameters

child	The individual to mutate.

```
{
    for (int i = 0; i < CHROMOSOME_SIZE; i++)
    {
        double p = RandomDouble(0.0, 1.0);
        if (p <= MUTATION_RATE)
        {
            mut_cnt += 1;
            T randomGene;
            while ((randomGene = CHROMOMIN + (CHROMOMAX - CHROMOMIN) * RandomDouble(0.0, 1.0)) ==
      child.chromosome[i])
                ;
            child.chromosome[i] = randomGene;
        }
    }
}
```

3.8.3.3 optimize()

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

Runs the GA optimization and returns the best individual found.

Returns

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

```
{
    Individual<T> bestIndividual = population[0];
    Individual<T> oldBestIndividual = bestIndividual;
    double bestFitness = bestIndividual.fitness;
    if (verbose) std::cout « "Initialising population..." « std::endl;
    initPopulation();
    int conv = 0;
    if (verbose) std::cout « "Starting evolution..." « std::endl;
    for (int i = 0; i < MAX_GENERATION; i++)
    {
        updatePopulation(type);
        bestIndividual = population[0];
        bestFitness = bestIndividual.fitness;
        if (verbose)
        {
        std::cout « "Generation: " « i « std::endl;
        std::cout « "Best chromosome: " « std::endl;

        for (int k = 0; k < CHROMOSOME_SIZE; ++k)
        {
            std::cout « bestIndividual.chromosome[k] « " ";
        }
        std::cout « " => " « bestIndividual.fitness « std::endl;
        std::cout « std::endl;
        std::cout « "Worst chromosome: " « std::endl;
        for (int k = 0; k < CHROMOSOME_SIZE; ++k)
        {
            std::cout « population[POP_SIZE - 1].chromosome[k] « " ";
        }
        std::cout « " => " « population[POP_SIZE - 1].fitness « std::endl;
        std::cout « std::endl;

        std::cout « "Adaptive Crossover Rate: " « CROSSOVER_RATE « std::endl;
        std::cout « "Adaptive Mutation Rate: " « MUTATION_RATE « std::endl;

        std::cout « "--------------------------" « std::endl;
        }

        if (oldBestIndividual == bestIndividual)
        {
            ++conv;
        }
        bestFitnessHistory.push_back(bestFitness);

        if (conv > CONV_GEN)
```

```
        {
            break; // early stopping criteria
        }
        oldBestIndividual = bestIndividual;
    }

    if (verbose) std::cout « "Finshed.." « std::endl;
    return bestIndividual;
}
```

3.8.3.4 selectParent()

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

Selects a parent based on the cumulative fitness values.

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

Parameters

cf	The cumulative fitness values of the population.

Returns

The index of the selected parent individual.

```
{
    double p = RandomDouble(0.0, 1.0);
    for (int i = 0; i < POP_SIZE; i++)
    {
        if (p <= cf[i])
        {
            return i;
        }
    }
    return POP_SIZE - 1;
}
```

3.8.3.5 setFitness()

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

Sets a fitness function for evaluating the fitness of individuals.

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

Parameters

fitness	The fitness function.

```
{
    this->fitness_ = fitness;
}
```

3.8.3.6 setValidity()

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

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

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

Parameters

validity	The validity checker function.

```
{
    this->validity_ = validity;
}
```

3.8.3.7 toFile()

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

Write the best individual to a file.

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

```
{
    std::stringstream fname;
    std::fstream f1;
    fname « fileDir « "GA_p" « std::to_string(POP_SIZE) « "_n" « std::to_string(CHROMOSOME_SIZE) «
    "_g" « std::to_string(MAX_GENERATION) « "_c" « std::to_string(CROSSOVER_RATE) « "_m" «
    std::to_string(MUTATION_RATE) « ".dat";
    std::cout « "Saving output to " « fname.str() « std::endl;
    f1.open(fname.str().c_str(), std::ios_base::out);

    // handle the case when the file cannot be opened
    if (f1.fail())
    {
        std::cout « "Error opening file! with name: " « std::endl;
        std::cout.flush();
        exit(0);
    }
    for (int it = 0; it < bestFitnessHistory.size(); ++it)
        f1 « it « "\t" « bestFitnessHistory[it] « std::endl;
    f1.close();
}
```

3.8.3.8 tournamentSelection()

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

Parameters

k	

Returns

int

```
{
    std::vector<int> candidates(k);
    for (int i = 0; i < k; ++i)
    {
        candidates[i] = RandomInt(0, POP_SIZE - 1);
    }
    Individual<T> best = population[candidates[0]];
    int bestIndex = candidates[0];
    for (int i = 1; i < k; ++i)
    {
        if (population[candidates[i]].fitness > best.fitness)
        {
            best = population[candidates[i]];
            bestIndex = candidates[i];
        }
    }

    return bestIndex;
}
```

3.8.3.9 updatePopulation()

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

Updates the population for the next generation.

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

```
{
    cross_cnt = 0;
    mut_cnt = 0;
    cross_contrib = 0.0;
    mut_contrib = 0.0;

    double minFitness = fitness[POP_SIZE - 1];
    double maxFitness = fitness[0];
    double offset = (minFitness < 0.0) ? -minFitness : 0.0;
    double totalFitness = 0.0;
    for (int i = 0; i < POP_SIZE; i++)
    {
        fitness[i] += offset;
        totalFitness += fitness[i];
    }
    scaledFitness.resize(POP_SIZE, 0.0);
    for (int i = 0; i < POP_SIZE; i++)
    {
        scaledFitness[i] = fitness[i] / totalFitness;
    }
    cumulativeFitness.resize(POP_SIZE, 0.0);
    cumulativeFitness[0] = scaledFitness[0];
    for (int i = 1; i < POP_SIZE; i++)
    {
        cumulativeFitness[i] = cumulativeFitness[i - 1] + scaledFitness[i];
    }

    int remaining = POP_SIZE;
    offspring.resize(POP_SIZE);
    fitness.resize(POP_SIZE);

    // Elitism
    for (int i = 0; i < ELITISM; ++i)
    {
        offspring[i] = population[i];
        fitness[i] = offspring[i].fitness;
        --remaining;
    }

#ifdef PARALLEL
#pragma omp parallel for default(none) shared(population, offspring, fitness, cumulativeFitness, type)
  reduction(+ : cross_cnt, mut_cnt, cross_contrib, mut_contrib) schedule(static)
    for (int i = ELITISM; i < POP_SIZE; ++i)
    {
        bool validChild1 = false;
```

```
        bool validChild2 = false;
        Individual<T> child1;
        child1.chromosome.resize(CHROMOSOME_SIZE);
        Individual<T> child2;
        child2.chromosome.resize(CHROMOSOME_SIZE);

        while (!validChild1 || !validChild2)
        {

            int parent1 = selectParent(cumulativeFitness);
            int parent2 = selectParent(cumulativeFitness);
            // int parent1 = tournamentSelection(10);
            // int parent2 = tournamentSelection(10);

            double crossoverProb = RandomDouble(0.0, 1.0);
            if (crossoverProb > CROSSOVER_RATE)
            {
                Crossover<T>::crossover(population[parent1], population[parent2], child1, child2, type);
#ifdef ADAPTIVE
                cross_cnt++;
                child1.fitness = fitness_(child1.chromosome);
                child2.fitness = fitness_(child2.chromosome);
                cross_contrib += child1.fitness + child2.fitness - population[parent1].fitness - population[parent2].fitness;
#endif
            }
            else
            {
                child1 = population[parent1];
                child2 = population[parent2];
            }
            double fit_before = child1.fitness + child2.fitness;
            mutate(child1);
            mutate(child2);
#ifdef ADAPTIVE
            child1.fitness = fitness_(child1.chromosome);
            child2.fitness = fitness_(child2.chromosome);
            mut_contrib += child1.fitness + child2.fitness - fit_before;
#endif
            validChild1 = validity_(child1.chromosome);
            validChild2 = validity_(child2.chromosome);
        }

        if (validChild1 && !validChild2)
        {
#ifndef ADAPTIVE
            child1.fitness = fitness_(child1.chromosome);
#endif
            offspring[i] = child1;
            fitness[i] = offspring[i].fitness;
        }
        if (validChild2 && !validChild1)
        {
#ifndef ADAPTIVE
            child2.fitness = fitness_(child2.chromosome);
#endif
            offspring[i] = child2;
            fitness[i] = offspring[i].fitness;
        }
        else
        {
#ifndef ADAPTIVE
            child1.fitness = fitness_(child1.chromosome);
            child2.fitness = fitness_(child2.chromosome);
#endif
            if (child1.fitness > child2.fitness)
            {
                offspring[i] = child1;
                fitness[i] = offspring[i].fitness;
            }
            else
            {
                offspring[i] = child2;
                fitness[i] = offspring[i].fitness;
            }
        }
    }

#endif

#ifdef SERIAL
    // Crossover and mutation
    while (remaining > 0)
    {
        bool validChild1 = false;
        Individual<T> child1;
```

```
        child1.chromosome.resize(CHROMOSOME_SIZE);

        bool validChild2 = false;
        Individual<T> child2;
        child2.chromosome.resize(CHROMOSOME_SIZE);

        int parent1 = selectParent(cumulativeFitness);
        int parent2 = selectParent(cumulativeFitness);
        // int parent1 = tournamentSelection(10);
        // int parent2 = tournamentSelection(10);

        double crossoverProb = RandomDouble(0.0, 1.0);
        if (crossoverProb > CROSSOVER_RATE)
        {
            Crossover<T>::crossover(population[parent1], population[parent2], child1, child2, type);
#ifdef ADAPTIVE
            cross_cnt++;
            child1.fitness = fitness_(child1.chromosome);
            child2.fitness = fitness_(child2.chromosome);
            cross_contrib += child1.fitness + child2.fitness - population[parent1].fitness -
      population[parent2].fitness;
#endif
        }
        else
        {
            child1 = population[parent1];
            child2 = population[parent2];
        }
        double fit_before = child1.fitness + child2.fitness;
        mutate(child1);
        mutate(child2);
#ifdef ADAPTIVE
        child1.fitness = fitness_(child1.chromosome);
        child2.fitness = fitness_(child2.chromosome);
        mut_contrib += child1.fitness + child2.fitness - fit_before;
#endif
        validChild1 = validity_(child1.chromosome);
        validChild2 = validity_(child2.chromosome);

        if (validChild1 && remaining > 0)
        {
#ifndef ADAPTIVE
            child1.fitness = fitness_(child1.chromosome);
#endif
            offspring[POP_SIZE - remaining] = child1;
            fitness[POP_SIZE - remaining] = offspring[POP_SIZE - remaining].fitness;
            --remaining;
        }
        if (validChild2 && remaining > 0)
        {
#ifndef ADAPTIVE
            child2.fitness = fitness_(child2.chromosome);
#endif
            offspring[POP_SIZE - remaining] = child2;
            fitness[POP_SIZE - remaining] = offspring[POP_SIZE - remaining].fitness;
            --remaining;
        }
    }
#endif

#ifdef ADAPTIVE
    double cm = cross_contrib / cross_cnt;
    double mm = mut_contrib / mut_cnt;
    double theta = 0.1 * (maxFitness - totalFitness / POP_SIZE) / (maxFitness - totalFitness);
    if (cm > 0.0)
    {
        CROSSOVER_RATE = std::min(1.0, CROSSOVER_RATE + theta);
        MUTATION_RATE = std::max(0.0, MUTATION_RATE - theta);
    }
    else
    {
        CROSSOVER_RATE = std::max(0.0, CROSSOVER_RATE - theta);
        MUTATION_RATE = std::min(1.0, MUTATION_RATE + theta);
    }
#endif
    // Update population
    population = offspring;

    std::sort(population.begin(), population.end(), std::greater<Individual<T>>());
    for (int i = 0; i < POP_SIZE; i++)
    {
        fitness[i] = population[i].fitness;
    }
}
```

3.8.4 Member Data Documentation

3.8.4.1 bestFitnessHistory

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

3.8.4.2 CHROMOMAX

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

3.8.4.3 CHROMOMIN

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

3.8.4.4 CHROMOSOME_SIZE

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

3.8.4.5 CONV_GEN

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

3.8.4.6 cross_cnt

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

3.8.4.7 cross_contrib

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

3.8.4.8 CROSSOVER_RATE

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

3.8.4.9 cumulativeFitness

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

3.8.4.10 ELITISM

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

3.8.4.11 fitness

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

3.8.4.12 fitness_

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

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

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

Template Parameters

T	The type of values in the chromosome.

Parameters

chromosome	The chromosome vector for which to evaluate the fitness.

Returns

The fitness value of the chromosome.

3.8.4.13 MAX_GENERATION

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

3.8.4.14 mut_cnt

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

3.8.4.15 mut_contrib

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

3.8.4.16 MUTATION_RATE

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

3.8.4.17 offspring

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

3.8.4.18 POP_SIZE

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

3.8.4.19 population

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

3.8.4.20 scaledFitness

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

3.8.4.21 validity_

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

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

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

Template Parameters

T	The type of values in the chromosome.

Parameters

chromosome	The chromosome vector to be checked for validity.

Returns

True if the chromosome is valid, false otherwise.

3.8.4.22 verbose

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

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

- GeneticAlgorithm.hpp
- GeneticAlgorithm.cpp

3.9 Individual< T > Class Template Reference

Template class representing an individual in the GA population.

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

Public Member Functions

- Individual ()

  Default constructor for the Individual class.
- Individual (const std::vector< T > &chromosome)

  Constructs an Individual with the given chromosome.
- bool operator< (const Individual &ind) const

  Overloads the less-than operator for comparing individuals based on fitness.
- bool operator> (const Individual &ind) const

  Overloads the greater-than operator for comparing individuals based on fitness.
- bool operator== (const Individual &ind) const

  Overloads the equality operator for comparing individuals based on their chromosomes.
- Individual & operator= (const Individual &ind)

  Overloads the assignment operator for assigning the values of another Individual to this Individual.

Public Attributes

- std::vector< T > chromosome
- double fitness

3.9.1 Detailed Description

template<typename T>
class Individual< T >

Template class representing an individual in the GA population.

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

Template Parameters

T	The type of values in the chromosome.

3.9.2 Constructor & Destructor Documentation

3.9.2.1 Individual() [1/2]

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

Default constructor for the Individual class.

```
{}
```

3.9.2.2 Individual() [2/2]

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

Constructs an Individual with the given chromosome.

Parameters

chromosome	The chromosome representing the genetic information.

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

3.9.3 Member Function Documentation

3.9.3.1 operator<()

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

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

Parameters

ind	The Individual to compare.

Returns

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

```
{
    return fitness < ind.fitness;
}
```

3.9.3.2 operator=()

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

Overloads the assignment operator for assigning the values of another Individual to this Individual.

Parameters

ind	The Individual to assign.

Returns

Reference to the assigned Individual.

```
    {
        chromosome.resize(ind.chromosome.size());
        chromosome = ind.chromosome;
        fitness = ind.fitness;
        return *this;
    }
```

3.9.3.3 operator==()

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

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

Parameters

ind	The Individual to compare.

Returns

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

```
    {
        return chromosome == ind.chromosome;
    }
```

3.9.3.4 operator>()

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

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

Parameters

ind	The Individual to compare.

Returns

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

```
{
    return fitness > ind.fitness;
}
```

3.9.4 Member Data Documentation

3.9.4.1 chromosome

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

3.9.4.2 fitness

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

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

- GeneticAlgorithm.hpp

3.10 Stream Class Reference

A Stream class to store the stream information.

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

Public Member Functions

- Stream ()
- Stream (double GER, double WASTE)
- Stream operator+ (const Stream &s)
- void operator+= (const Stream &s)
- ∼Stream ()

Public Attributes

- double ger
- double waste

3.10.1 Detailed Description

A Stream class to store the stream information.

Author

acse-sm222

3.10.2 Constructor & Destructor Documentation

3.10.2.1 Stream() [1/2]

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

```
{
    ger = 0;
    waste = 0;
};
```

3.10.2.2 Stream() [2/2]

```
Stream::Stream (
            double GER,
            double WASTE )  [inline]
```

```
: ger(GER), waste(WASTE){};
```

3.10.2.3 ∼Stream()

```
Stream::∼Stream ( )  [inline]
```

Destructor

3.10.3 Member Function Documentation

3.10.3.1 operator+()

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

Parameters

s	Overload the + operator

```
{
    Stream stream;
    stream.ger = this->ger + s.ger;
    stream.waste = this->waste + s.waste;
    return stream;
}
```

3.10.3.2 operator+=()

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

Parameters

s	Overload the += operator

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

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

3.10.4 Member Data Documentation

3.10.4.1 ger

```
double Stream::ger
```

Constructor taking the GER and WASTE Germanium flow rate

3.10.4.2 waste

```
double Stream::waste
```

Waste flow rate

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

- CUnit.h

3.11 Units Class Reference

Public Member Functions

- Units ()
- Units (int conc_num, int tails_num)

Public Attributes

- int conc_num
- int tails_num
- bool marked

3.11.1 Constructor & Destructor Documentation

3.11.1.1 Units() [1/2]

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

3.11.1.2 Units() [2/2]

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

3.11.2 Member Data Documentation

3.11.2.1 conc_num

```
int Units::conc_num
```

3.11.2.2 marked

```
bool Units::marked
```

3.11.2.3 tails_num

```
int Units::tails_num
```

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

- CCircuit.cpp

3.12 UnitTestCircuit Class Reference

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

Public Member Functions

- UnitTestCircuit ()
  Default constructor.
- ∼UnitTestCircuit ()
  Destructor.
- void test_circuit_evaluation ()
- void test_circuit_validity ()
- void test_circuit_unit ()
- void run ()

3.12.1 Constructor & Destructor Documentation

3.12.1.1 UnitTestCircuit()

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

Default constructor.

```
00006 {}
```

3.12.1.2 ∼UnitTestCircuit()

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

Destructor.

```
{}
```

3.12.2 Member Function Documentation

3.12.2.1 run()

```
void UnitTestCircuit::run ( )
                             {
    test_circuit_evaluation();
    test_circuit_validity();
    test_circuit_unit();
}
```

3.12.2.2 test_circuit_evaluation()

```
void UnitTestCircuit::test_circuit_evaluation ( )
                                                 {
    std::cout « "============================Test Circuit Evaluation============================\n";

    std::vector<int> circuit_vector = {4, 5, 1, 2, 4, 0, 1, 1, 6, 1, 3};
    bool validity = Circuit::Check_Validity(circuit_vector);
    assert(validity==true);

    Circuit_Parameters parameters;
    parameters.tolerance = 1e-6;

    double value = Evaluate_Circuit(circuit_vector, parameters);
    std::cout « value « std::endl;
    assert(std::fabs(value - 107.204) <= 1e-3);
    std::cout « "completed" « std::endl;
}
```

3.12.2.3 test_circuit_unit()

```
void UnitTestCircuit::test_circuit_unit ( )
                                           {
    std::cout « "============================Test Circuit Unit============================\n";

    CUnit test_unit;

    test_unit.input.ger = 20;
    test_unit.input.waste = 80;

    test_unit.calculateOutputFlowRates(test_unit.input);

    double eps = 1e-3;

    bool check1 = fabs(test_unit.conc.ger - 2.6087) < eps;
    bool check2 = fabs(test_unit.conc.waste - 1.18227) < eps;
    bool check3 = fabs(test_unit.tail.ger - 17.3913) < eps;
    bool check4 = fabs(test_unit.tail.waste - 78.8177) < eps;

    assert(check1==true);
    assert(check2==true);
    assert(check3==true);
    assert(check4==true);
    std::cout « "completed" « std::endl;
}
```

3.12.2.4 test_circuit_validity()

```
void UnitTestCircuit::test_circuit_validity ( )
{
    std::cout « "============================Test Circuit Validity============================\n";

    // Cases when there are 3 units
    // Case 1: self-recycle
    std::vector<int> circuit_vector1 = {0, 1, 2, 0, 2, 1, 0};
    // Case 2: points to itself
    std::vector<int> circuit_vector2 = {0, 1, 2, 1, 2, 3, 4};
    // Case 3: there are units not ending up in 2 destinations
    std::vector<int> circuit_vector3 = {0, 1, 1, 3, 2, 1, 4};
    // Case 4: there is a unit not accessible from the feed
    std::vector<int> circuit_vector4 = {0, 1, 4, 3, 4, 3, 0};
    bool check1 = Circuit::Check_Validity(circuit_vector1);
    bool check2 = Circuit::Check_Validity(circuit_vector2);
    bool check3 = Circuit::Check_Validity(circuit_vector3);
    bool check4 = Circuit::Check_Validity(circuit_vector4);
    assert(check1==false);
    assert(check2==false);
    assert(check3==false);
    assert(check4==false);
    std::cout « "completed" « std::endl;
}
```

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

- Unittests.h
- test_Circuit.cpp

3.13 UnitTestCrossover Class Reference

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

Public Member Functions

- UnitTestCrossover ()

  Default constructor.
- ∼UnitTestCrossover ()

  Destructor.
- void test_one_pt_crossover ()
- void test_two_pts_crossover ()
- void test_mult_pts_crossover ()
- void test_uniform_crossover ()

3.13.1 Constructor & Destructor Documentation

3.13.1.1 UnitTestCrossover()

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

Default constructor.

```
{
    std::cout « "                              Test Crossover Start
\n";
}
```

3.13.1.2 ∼UnitTestCrossover()

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

Destructor.

```
{}
```

3.13.2 Member Function Documentation

3.13.2.1 test_mult_pts_crossover()

```
void UnitTestCrossover::test_mult_pts_crossover ( )
{
    std::cout « "===========================Test Multi-Points Crossover
method===========================\n";

    std::vector<int> chromosome1 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    std::vector<int> chromosome2 = {10, 11, 12, 13, 14, 15, 16, 17, 18, 19};

    Individual<int> parent1, parent2, child1, child2;

    parent1.chromosome = chromosome1;
    parent2.chromosome = chromosome2;

    child1.chromosome.resize(chromosome1.size());
    child2.chromosome.resize(chromosome2.size());

    crossover.mult_pts_crossover(parent1, parent2, child1, child2);

    std::cout « "Parent 1: ";
    printChromosome(parent1.chromosome);

    std::cout « "Parent 2: ";
    printChromosome(parent2.chromosome);

    std::cout « "Child 1: ";
    printChromosome(child1.chromosome);

    std::cout « "Child 2: ";
    printChromosome(child2.chromosome);

    // Check that the chromosomes have been altered
    assert(child1.chromosome != parent1.chromosome);
    assert(child1.chromosome != parent2.chromosome);
    assert(child2.chromosome != parent1.chromosome);
    assert(child2.chromosome != parent2.chromosome);
}
```

3.13.2.2 test_one_pt_crossover()

```
void UnitTestCrossover::test_one_pt_crossover ( )
{
    std::cout « "============================Test One-Point Crossover
method============================\n";

    std::vector<int> chromosome1 = {1, 2, 3, 4, 5};
    std::vector<int> chromosome2 = {6, 7, 8, 9, 10};

    Individual<int> parent1, parent2, child1, child2;

    parent1.chromosome = chromosome1;
    parent2.chromosome = chromosome2;

    child1.chromosome.resize(chromosome1.size());
    child2.chromosome.resize(chromosome2.size());

    crossover.one_pt_crossover(parent1, parent2, child1, child2);

    std::cout « "Parent 1: ";
    printChromosome(parent1.chromosome);

    std::cout « "Parent 2: ";
    printChromosome(parent2.chromosome);
```

```
    std::cout « "Child 1: ";
    printChromosome(child1.chromosome);

    std::cout « "Child 2: ";
    printChromosome(child2.chromosome);

    // Check that the chromosomes have been altered
    assert(child1.chromosome != parent1.chromosome);
    assert(child1.chromosome != parent2.chromosome);
    assert(child2.chromosome != parent1.chromosome);
    assert(child2.chromosome != parent2.chromosome);
}
```

3.13.2.3 test_two_pts_crossover()

```
void UnitTestCrossover::test_two_pts_crossover ( )
{
    std::cout « "============================Test Two-Points Crossover method============================\n";

    std::vector<int> chromosome1 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    std::vector<int> chromosome2 = {10, 11, 12, 13, 14, 15, 16, 17, 18, 19};

    Individual<int> parent1, parent2, child1, child2;

    parent1.chromosome = chromosome1;
    parent2.chromosome = chromosome2;

    child1.chromosome.resize(chromosome1.size());
    child2.chromosome.resize(chromosome2.size());

    crossover.two_pts_crossover(parent1, parent2, child1, child2);

    std::cout « "Parent 1: ";
    printChromosome(parent1.chromosome);

    std::cout « "Parent 2: ";
    printChromosome(parent2.chromosome);

    std::cout « "Child 1: ";
    printChromosome(child1.chromosome);

    std::cout « "Child 2: ";
    printChromosome(child2.chromosome);

    // Check that the chromosomes have been altered
    assert(child1.chromosome != parent1.chromosome);
    assert(child1.chromosome != parent2.chromosome);
    assert(child2.chromosome != parent1.chromosome);
    assert(child2.chromosome != parent2.chromosome);
}
```

3.13.2.4 test_uniform_crossover()

```
void UnitTestCrossover::test_uniform_crossover ( )
{
    std::cout « "=============================Test Uniform Crossover method==============================\n";

    std::vector<int> chromosome1 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    std::vector<int> chromosome2 = {9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
    std::vector<int> expected_chromosome1 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    std::vector<int> expected_chromosome2 = {9, 8, 7, 6, 5, 4, 3, 2, 1, 0};

    Individual<int> parent1, parent2, child1, child2;

    parent1.chromosome = chromosome1;
    parent2.chromosome = chromosome2;

    child1.chromosome.resize(chromosome1.size());
    child2.chromosome.resize(chromosome2.size());

    crossover.uniform_crossover(parent1, parent2, child1, child2);

    // Check that the chromosomes have been altered
    assert(child1.chromosome != expected_chromosome1);
    assert(child2.chromosome != expected_chromosome2);
```

```
    std::cout « "Results match expected outputs.\n";
}
```

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

- Unittests.h
- test_Crossover.cpp

3.14 UnitTestGA Class Reference

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

Public Member Functions

- UnitTestGA (int populationSize, int chromosomeSize, int chromomin, int chromomax, int maxGeneration, double crossoverRate, double mutationRate)

    Constructor for the UnitTestGeneticAlgorithm class.
- ∼UnitTestGA ()

    Destructor for the UnitTestGeneticAlgorithm class..
- void test_mutation ()
- void test_select_parent ()
- void test_global_optimum ()

    Test with Rosenbrock function to check if the algorithm is capable in finding the global optimum successfully.
- void run ()

Public Attributes

- int POPULATION_SIZE = 100
- int CHROMOSOME_SIZE = 10
- int CHROMOMIN = 0
- int CHROMOMAX = 10
- int MAX_GENERATION = 2000
- int CONV_GEN = 2000
- double CROSSOVER_RATE = 0.8
- double MUTATION_RATE = 0.15
- GA< int > * ga

3.14.1 Constructor & Destructor Documentation

3.14.1.1 UnitTestGA()

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

Constructor for the UnitTestGeneticAlgorithm class.

```
            : POPULATION_SIZE(populationSize), CHROMOSOME_SIZE(chromosomeSize),
    CHROMOMIN(chromomin),
            CHROMOMAX(chromomax), MAX_GENERATION(maxGeneration), CROSSOVER_RATE(crossoverRate),
            MUTATION_RATE(mutationRate) {};
```

3.14.1.2 ∼UnitTestGA()

```
UnitTestGA::∼UnitTestGA ( )
```

Destructor for the UnitTestGeneticAlgorithm class..

```
{};
```

3.14.2 Member Function Documentation

3.14.2.1 run()

```
void UnitTestGA::run ( )
                    {
    test_mutation();
    test_select_parent();
    test_global_optimum();
}
```

3.14.2.2 test_global_optimum()

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

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

```
                                    {
    std::cout « "============================Test Global Optimum============================\n";
    // configure GA
    ga->setValidity(validity);
    ga->setFitness(Rosenbrock);
    ga->verbose = false;
    CrossoverType type = CrossoverType::UNIFORM;
    // Known global optimum for Rosenbrock function
    std::vector<int> expected_chromosome = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1};
    int success_count = 0;
    int num_runs = 10;

    // Run for 10 times to see if the algorithm can find the global optimum
    // if 50% of the time can find the global optimum, then the test is passed
    for (int i = 0; i < num_runs; i++) {
        auto best = ga->optimize(type);
        if (best.chromosome == expected_chromosome) {
            success_count++;
        }
    }
    assert(success_count >= num_runs / 2);
    std::cout « "successfully found global optimum for Rosenbrock: " « success_count;
    std::cout « "/" « num_runs « " times!" « std::endl;
    std::cout « "completed" « std::endl;
}
```

3.14.2.3 test_mutation()

```
void UnitTestGA::test_mutation ( )
                              {
    std::cout « "============================Test Mutation============================\n";
    std::vector<int> init_chromosome = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    Individual<int> init_individual(init_chromosome);
    Individual<int> result_individual(init_chromosome);
    ga->MUTATION_RATE = 0.5;

    ga->mutate(result_individual);
    assert(init_individual.chromosome != result_individual.chromosome);
    std::cout « "completed" « std::endl;
}
```

3.14.2.4 test_select_parent()

```
void UnitTestGA::test_select_parent ( )
                                     {
    std::cout « "===========================Test Select Parent===========================\n";
    int result;
    std::vector<double> comulative_fitness = {-1.0, 1.0, 1.0};

    result = ga->selectParent(comulative_fitness);
    assert(result == 1);
    std::cout « "completed" « std::endl;
}
```

3.14.3 Member Data Documentation

3.14.3.1 CHROMOMAX

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

3.14.3.2 CHROMOMIN

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

3.14.3.3 CHROMOSOME_SIZE

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

3.14.3.4 CONV_GEN

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

3.14.3.5 CROSSOVER_RATE

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

3.14.3.6 ga

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

Initial value:

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

3.14.3.7 MAX_GENERATION

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

3.14.3.8 MUTATION_RATE

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

3.14.3.9 POPULATION_SIZE

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

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

- Unittests.h
- test_GeneticAlgorithm.cpp

3.15 UnitTestIndividual Class Reference

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

Public Member Functions

- UnitTestIndividual ()
  Default constructor.
- ~UnitTestIndividual ()
  Destructor.
- void test_constructor ()
  Test the constructor and getter functions.
- void test_Get_Set_fitness ()
  Test the calculate fitness value function.
- void test_Get_numUnits ()
  Test the getter function for num_units.
- void test_Get_vectorLength ()
  Test the getter function for vector_length.

Protected Attributes

- Individual< int > individual

3.15.1 Constructor & Destructor Documentation

3.15.1.1 UnitTestIndividual()

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

Default constructor.

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

3.15.1.2 ∼UnitTestIndividual()

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

Destructor.

```
{
    std::cout « "Test Individual completed" « std::endl;
}
```

3.15.2 Member Function Documentation

3.15.2.1 test_constructor()

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

Test the constructor and getter functions.

```
{
    int chromosomeLength = 10;
    individual = Individual(chromosomeLength, 0, 10);

    assert(individual.Get_vectorLength() == 10);
    assert(individual.chromosome.size() == 10);
}
```

3.15.2.2 test_Get_numUnits()

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

Test the getter function for num_units.

3.15.2.3 test_Get_Set_fitness()

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

Test the calculate fitness value function.

Test the getter and setter function for fitness.

```
{
    std::vector<int> fitness_list = {10, 20, 30, 40, 50};

    for (int i = 0; i < fitness_list.size(); i++){
        individual.Set_fitness(fitness_list[i]);
        assert(individual.Get_fitness() == fitness_list[i]);
    }
}
```

3.15.2.4 test_Get_vectorLength()

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

Test the getter function for vector_length.

```
{
    std::vector<int> vectorLength_list = {10, 100, 1000};

    for (int item : vectorLength_list){
        individual = Individual(item, 0, 10);
         assert(individual.Get_vectorLength() == item);
    }
}
```

3.15.3 Member Data Documentation

3.15.3.1 individual

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

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

- Unittests.h
- test_Individual.cpp

Chapter 4

File Documentation

4.1 CCircuit.h File Reference

Circuit class.

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

Classes

- struct Circuit_Parameters

  A Structure to store the parameters for the circuit simulator.
- class Circuit

  Circuit class.

Functions

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

  Calculate the ger and waste rates in all streams of the circuit.
- void Calculate_Circuit (Circuit &circuit)

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

4.1.1 Detailed Description

Circuit class.

Author

acse-sm222

Copyright

Copyright (c) 2023

4.1.2 Function Documentation

4.1.2.1 Calculate_Circuit() [1/2]

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

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

Parameters

circuit_vector	

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

Parameters

circuit_vector	Circuit vector

```
{
  struct Circuit_Parameters default_circuit_parameters;
  Calculate_Circuit(circuit, default_circuit_parameters);
}
```

4.1.2.2 Calculate_Circuit() [2/2]

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

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

Parameters

circuit_vector	
parameters	

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

Parameters

circuit_vector	Circuit vector
parameters	Circuit parameters

```
{
  // 1. Asssign initial values for F_ger and F_waste in each unit of the circuit
  for (int i = 0; i < circuit.num_units; ++i)
  {
    circuit.units[i].input.ger = parameters.input_Fger;
    circuit.units[i].input.waste = parameters.input_Fwaste;
    circuit.units[i].F = circuit.units[i].input.ger + circuit.units[i].input.waste;
  }

  // 2. Calculate the flow rates for each unit of the circuit
  for (int i = 0; i < circuit.num_units; ++i)
  {
    circuit.units[i].calculateOutputFlowRates(circuit.units[i].input);
  }

  // 3. Set F_old = F
  for (int i = 0; i < circuit.num_units; ++i)
  {
    circuit.units[i].F_old = circuit.units[i].F;
  }

  // Total old feed rates
  std::vector<double> F_old(circuit.num_units, 0.0);
  for (int i = 0; i < circuit.num_units; ++i)
  {
    F_old[i] = circuit.units[i].F_old;
  }

  // New Feed rates
```

```
  std::vector<double> F(circuit.num_units, 0.0);

  // Boolean to check if the circuit has reached steady state
  bool terminate = false;

  circuit.it = 0;
  while (!terminate)
  {
    // 4. Calculate the input flow rates for each unit of the circuit
    for (int i = 0; i < circuit.num_units; ++i)
    {
      circuit.units[i].F_old = circuit.units[i].F;
      circuit.units[i].input.ger = 0;
      circuit.units[i].input.waste = 0;
    }

    for (int i = 0; i < circuit.num_units; ++i)
    {
      // First the concentration stream
      if (circuit.units[i].conc_num >= 0)
      {
        circuit.units[circuit.units[i].conc_num].input += circuit.units[i].conc;
      }

      // Then the tails stream
      if (circuit.units[i].tails_num >= 0)
      {
        circuit.units[circuit.units[i].tails_num].input += circuit.units[i].tail;
      }

      if (i == circuit.feeder)
      {
        circuit.units[i].input.ger += parameters.input_Fger;
        circuit.units[i].input.waste += parameters.input_Fwaste;
      }
    }

    // 5. Calculate the output flow rates for each unit of the circuit
    for (int i = 0; i < circuit.num_units; ++i)
    {
      circuit.units[i].calculateOutputFlowRates(circuit.units[i].input);
      F[i] = circuit.units[i].input.ger + circuit.units[i].input.waste;
    }

    // 6. Check for convergence between old and new feed rates of the circuit
    bool converged = true;
    for (int i = 0; i < circuit.num_units; ++i)
    {
      if (fabs(F_old[i] - F[i]) > parameters.tolerance)
      {
        // Circuit not converged yet. Can break and continue
        converged = false;
        break;
      }
    }

    if (converged)
    {
      terminate = true;
    }
    else
    {
      terminate = false;
    }

    ++circuit.it;
    F_old = F;
    // F.resize(circuit.num_units, 0.0);

    if (circuit.it > parameters.max_iterations)
    {
      return;
    }
  }
}
```

4.2 CCircuit.h

Go to the documentation of this file.

```
#pragma once

```

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

struct Circuit_Parameters
{
  double tolerance = 1e-6;
  int max_iterations = 1000;
  double input_Fger = 10.0;
  double input_Fwaste = 100.0;
  double get_money = 100.0;
  double lose_money = 500.0;
};

class Circuit
{
public:
  Circuit(std::vector<int> circuit_vector);
  ~Circuit();
  void toFile();
  int num_units;
  int feeder;
  std::vector<int> circuit_vector;
  std::vector<CUnit> units;
  static bool Check_Validity(const std::vector<int> &circuit_vector);
  int it;
};

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

void Calculate_Circuit(Circuit &circuit);
```

4.3 CSimulator.h File Reference

Simulator class.

```
#include <vector>
#include <float.h>
#include ¨CCircuit.h¨
```

Functions

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

  Function to evaluate the objective function.
- double Evaluate_Circuit (std::vector< int > circuit_vector)

  Overloaded function to evaluate the objective function.

4.3.1 Detailed Description

Simulator class.

Author

acse-sm222

Copyright

4.3.2 Function Documentation

4.3.2.1 Evaluate_Circuit() [1/2]

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

Overloaded function to evaluate the objective function.

Parameters

circuit_vector	Vector representing the separator circuit

Returns

double Objective function value

Overloaded function to evaluate the objective function.

Parameters

circuit_vector	Circuit vector

Returns

double Objective function value

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

4.3.2.2 Evaluate_Circuit() [2/2]

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

Function to evaluate the objective function.

Parameters

circuit_vector	Vector representing the separator circuit
parameters	Structure containing the parameters for the circuit simulator

Returns

double Objective function value

Function to evaluate the objective function.

Parameters

circuit_vector	Circuit vector
parameters	Circuit parameters

Returns

double Objective function value

```
{
  // 1. Create a circuit object
  Circuit circuit(circuit_vector);

  // 2. Calculate the circuit
  Calculate_Circuit(circuit, parameters);

  double objective = parameters.get_money * circuit.units[circuit.num_units - 2].input.ger -
parameters.lose_money * circuit.units[circuit.num_units - 2].input.waste;

  return objective;
}
```

4.4 CSimulator.h

Go to the documentation of this file.

```
#pragma once

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

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

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

4.5 CUnit.h File Reference

Unit class.

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

Classes

- class Stream

  A Stream class to store the stream information.
- class CUnit

  A Unit class to store the unit information and processing.

4.5.1 Detailed Description

Unit class.

Author

acse-yl1922, acse-sm222

Copyright

4.6 CUnit.h

Go to the documentation of this file.

```
#pragma once

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

class Stream
{
public:
    Stream()
    {
        ger = 0;
        waste = 0;
    };
    Stream(double GER, double WASTE) : ger(GER), waste(WASTE){};
    double ger;
    double waste;
    Stream operator+(const Stream &s)
    {
        Stream stream;
        stream.ger = this->ger + s.ger;
        stream.waste = this->waste + s.waste;
        return stream;
    }

    void operator+=(const Stream &s)
    {
        this->ger += s.ger;
        this->waste += s.waste;
    }

    ~Stream() {}
};

class CUnit
{
public:
    CUnit();
    CUnit(int id, int conc_num, int tails_num);
    int conc_num;
    int tails_num;
    Stream input;
    Stream conc;
    Stream tail;
    double F, F_old;
    void calculateOutputFlowRates(const Stream &INP);
    ~CUnit() {}
private:
    int id;
    double k_ger = 0.005;
    double k_waste = 0.0005;
    int rho = 3000;
    double phi = 0.1;
    int V = 10;
};
```

4.7 DirectedGraph.hpp File Reference

Directed graph class.

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

Classes

- class DirectedGraph

    A directed graph.
- class DepthFirstSearch

    Depth first search class for directed graph.

4.7.1 Detailed Description

Directed graph class.

Author

acse-sm222

Copyright

Copyright (c) 2023

4.8 DirectedGraph.hpp

Go to the documentation of this file.

```
#pragma once

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

class DirectedGraph
{
public:
    DirectedGraph(){};
    DirectedGraph(int n);
    void addEdge(int u, int v);
    int getNumVertices() const;
    int getNumEdges() const;
    std::vector<int> getNeighbors(int u) const;
    friend std::ostream &operator«(std::ostream &os, const DirectedGraph &G);
    void writeToFile(std::string filename) const;
private:
    std::vector<std::vector<int» adjList;
};

class DepthFirstSearch
{
public:
    DepthFirstSearch(){};
    DepthFirstSearch(const DirectedGraph &G, int s);
    void dfs(const DirectedGraph &G, int u);
    bool visited(int v) const;
private:
    std::vector<bool> marked;
};
```

4.9 GeneticAlgorithm.hpp File Reference

Header file for the GeneticAlgorithm class.

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

Classes

- class Individual< T >

    Template class representing an individual in the GA population.
- class Crossover< T >

    Performs crossover operation between two parent individuals.
- class GA< T >

    Template class representing a Genetic Algorithm (GA).

Macros

- #define PARALLEL

Enumerations

- enum CrossoverType { ONE_PT = 0 , TWO_PTS = 1 , UNIFORM = 2 , MULT_PTS = 3 }

Functions

- double RandomDouble (double min, double max)
- int RandomInt (int min, int max)

4.9.1 Detailed Description

Header file for the GeneticAlgorithm class.

Copyright

Copyright (c) 2023

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

4.9.2 Macro Definition Documentation

4.9.2.1 PARALLEL

```
#define PARALLEL
```

4.9.3 Enumeration Type Documentation

4.9.3.1 CrossoverType

```
enum CrossoverType
```

Enumerator

ONE_PT	
TWO_PTS	
UNIFORM	
MULT_PTS	

```
{
    ONE_PT = 0,
    TWO_PTS = 1,
    UNIFORM = 2,
    MULT_PTS = 3
};
```

4.9.4 Function Documentation

4.9.4.1 RandomDouble()

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

4.9.4.2 RandomInt()

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

4.10 GeneticAlgorithm.hpp

Go to the documentation of this file.

```
#pragma once

#include <vector>
#include <string>
#include <random>
#include <iostream>
#include <tuple>
#include <functional>
#include <set>
#include <algorithm>
#include <chrono>
#include <omp.h>
#define PARALLEL
// #define SERIAL
// #define ADAPTIVE

// Random number generator
static std::random_device rd;
static std::mt19937 gen(rd()); // Initialize random number generator only once

double RandomDouble(double min, double max);

int RandomInt(int min, int max);

template <typename T>
```

```
class Individual
{
public:
    std::vector<T> chromosome; // Chromosome representing the genetic information
    double fitness;            // Fitness value indicating the quality of the individual’s solution

    // overload comparison operator
    Individual() {}

    Individual(const std::vector<T> &chromosome) : chromosome(chromosome) {}

    // overload comparison operator
    bool operator<(const Individual &ind) const
    {
        return fitness < ind.fitness;
    }

    bool operator>(const Individual &ind) const
    {
        return fitness > ind.fitness;
    }

    bool operator==(const Individual &ind) const
    {
        return chromosome == ind.chromosome;
    }

    Individual &operator=(const Individual &ind)
    {
        chromosome.resize(ind.chromosome.size());
        chromosome = ind.chromosome;
        fitness = ind.fitness;
        return *this;
    }
};

enum CrossoverType
{
    ONE_PT = 0,
    TWO_PTS = 1,
    UNIFORM = 2,
    MULT_PTS = 3
};

template <typename T>
class Crossover
{
public:
    static void one_pt_crossover(Individual<T> &parent1, Individual<T> &parent2, Individual<T> &child1, Individual<T> &child2);
    static void two_pts_crossover(Individual<T> &parent1, Individual<T> &parent2, Individual<T> &child1, Individual<T> &child2);
    static void uniform_crossover(Individual<T> &parent1, Individual<T> &parent2, Individual<T> &child1, Individual<T> &child2);
    static void mult_pts_crossover(Individual<T> &parent1, Individual<T> &parent2, Individual<T> &child1, Individual<T> &child2);
    static void crossover(Individual<T> &parent1, Individual<T> &parent2, Individual<T> &child1, Individual<T> &child2, CrossoverType type);
};

template <typename T>
class GA
{
public:
    // Default constructor
    GA() = default;
    GA(int populationSize, int chromosomeSize, T chromomin, T chromomax, int maxGeneration, double crossoverRate, double mutationRate);

    void setValidity(const std::function<bool(const std::vector<T> &)> &validity);

    void setFitness(const std::function<double(const std::vector<T> &)> &fitness);

    Individual<T> optimize(CrossoverType type);

    ~GA(){};

    T CHROMOMIN;            // Minimum value of discrete allele // chromosome min
    T CHROMOMAX;            // Maximum value of discrete allele // chromosome max
    int POP_SIZE;           // Population size
    int CHROMOSOME_SIZE;    // Chromosome size
    int MAX_GENERATION;     // Maximum generation
    double CROSSOVER_RATE;  // Crossover rate
    double MUTATION_RATE;   // Mutation rate
    int ELITISM;            // Number of elite individuals
```

```
    int CONV_GEN;          // Number of generations for convergence
    bool verbose = false;  // Print progress to console

    std::vector<Individual<T>> population; // Population of individuals
    void toFile(std::string fileDir);

    void mutate(Individual<T> &child);

    int selectParent(const std::vector<double> &cf);


private:
    // GA variables
    std::vector<Individual<T>> offspring;  // Offspring population
    std::vector<double> fitness;            // Fitness values of individuals
    std::vector<double> scaledFitness;      // Scaled fitness values of individuals
    std::vector<double> cumulativeFitness; // Cumulative fitness values of individuals
    int cross_cnt, mut_cnt;                 // Crossover and mutation counters
    double cross_contrib, mut_contrib;      // Crossover and mutation contribution to the next generation
    std::vector<double> bestFitnessHistory;// Best fitness values of each generation

    // GA functions
    void initPopulation();

    void updatePopulation(CrossoverType type);

    int tournamentSelection(int k);




    std::function<bool(const std::vector<T> &)> validity_;

    std::function<double(const std::vector<T> &)> fitness_;
};

// Declare the template class
template class GA<double>;
template class GA<int>;
template class Crossover<double>;
template class Crossover<int>;
```

4.11 CCircuit.cpp File Reference

Circuit class implementation.

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

Classes

- class Units

Functions

- void markunits (std::vector< Units > &units, int unit_num)

4.11.1 Detailed Description

Circuit class implementation.

Author

acse-yl1922, acse-sm222

Copyright

Copyright (c) 2023

4.11.2 Function Documentation

4.11.2.1 markunits()

```
void markunits (
            std::vector< Units > & units,
            int unit_num )
{
  if (units[unit_num].marked == true)
  {
    return;
  }
  else
  {
    units[unit_num].marked = true;
  }

  if (units[unit_num].conc_num < units.size())
  {
    markunits(units, units[unit_num].conc_num);
  }
  else
  {
    return;
  }

  if (units[unit_num].tails_num < units.size())
  {
    markunits(units, units[unit_num].tails_num);
  }
  else
  {
    return;
  }
}
```

4.12 CSimulator.cpp File Reference

Simulator calculations.

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

Functions

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

  Calculates the steady state feed rates for the entire circuit.
- double Evaluate_Circuit (std::vector< int > circuit_vector, struct Circuit_Parameters parameters)

  Evaluate the circuit with given parameters.
- void Calculate_Circuit (Circuit &circuit)

  Calculate the circuit with default parameters.
- double Evaluate_Circuit (std::vector< int > circuit_vector)

  Evaluate the circuit with default parameters.

4.12.1 Detailed Description

Simulator calculations.

Author

acse-sm222

Copyright

Copyright (c) 2023

4.12.2 Function Documentation

4.12.2.1 Calculate_Circuit() [1/2]

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

Calculate the circuit with default parameters.

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

Parameters

circuit_vector	Circuit vector

```
{
  struct Circuit_Parameters default_circuit_parameters;
  Calculate_Circuit(circuit, default_circuit_parameters);
}
```

4.12.2.2 Calculate_Circuit() [2/2]

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

Calculates the steady state feed rates for the entire circuit.

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

Parameters

circuit_vector	Circuit vector
parameters	Circuit parameters

```
{
  // 1. Asssign initial values for F_ger and F_waste in each unit of the circuit
  for (int i = 0; i < circuit.num_units; ++i)
  {
    circuit.units[i].input.ger = parameters.input_Fger;
    circuit.units[i].input.waste = parameters.input_Fwaste;
    circuit.units[i].F = circuit.units[i].input.ger + circuit.units[i].input.waste;
  }
```

```

  // 2. Calculate the flow rates for each unit of the circuit
  for (int i = 0; i < circuit.num_units; ++i)
  {
    circuit.units[i].calculateOutputFlowRates(circuit.units[i].input);
  }

  // 3. Set F_old = F
  for (int i = 0; i < circuit.num_units; ++i)
  {
    circuit.units[i].F_old = circuit.units[i].F;
  }

  // Total old feed rates
  std::vector<double> F_old(circuit.num_units, 0.0);
  for (int i = 0; i < circuit.num_units; ++i)
  {
    F_old[i] = circuit.units[i].F_old;
  }

  // New Feed rates
  std::vector<double> F(circuit.num_units, 0.0);

  // Boolean to check if the circuit has reached steady state
  bool terminate = false;

  circuit.it = 0;
  while (!terminate)
  {
    // 4. Calculate the input flow rates for each unit of the circuit
    for (int i = 0; i < circuit.num_units; ++i)
    {
      circuit.units[i].F_old = circuit.units[i].F;
      circuit.units[i].input.ger = 0;
      circuit.units[i].input.waste = 0;
    }

    for (int i = 0; i < circuit.num_units; ++i)
    {
      // First the concentration stream
      if (circuit.units[i].conc_num >= 0)
      {
        circuit.units[circuit.units[i].conc_num].input += circuit.units[i].conc;
      }

      // Then the tails stream
      if (circuit.units[i].tails_num >= 0)
      {
        circuit.units[circuit.units[i].tails_num].input += circuit.units[i].tail;
      }

      if (i == circuit.feeder)
      {
        circuit.units[i].input.ger += parameters.input_Fger;
        circuit.units[i].input.waste += parameters.input_Fwaste;
      }
    }

    // 5. Calculate the output flow rates for each unit of the circuit
    for (int i = 0; i < circuit.num_units; ++i)
    {
      circuit.units[i].calculateOutputFlowRates(circuit.units[i].input);
      F[i] = circuit.units[i].input.ger + circuit.units[i].input.waste;
    }

    // 6. Check for convergence between old and new feed rates of the circuit
    bool converged = true;
    for (int i = 0; i < circuit.num_units; ++i)
    {
      if (fabs(F_old[i] - F[i]) > parameters.tolerance)
      {
        // Circuit not converged yet. Can break and continue
        converged = false;
        break;
      }
    }

    if (converged)
    {
      terminate = true;
    }
    else
    {
      terminate = false;
    }

    ++circuit.it;
```

```
    F_old = F;
    // F.resize(circuit.num_units, 0.0);

    if (circuit.it > parameters.max_iterations)
    {
      return;
    }
  }
}
```

4.12.2.3 Evaluate_Circuit() [1/2]

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

Evaluate the circuit with default parameters.

Overloaded function to evaluate the objective function.

Parameters

circuit_vector	Circuit vector

Returns

double Objective function value

```
{
  struct Circuit_Parameters default_circuit_parameters;
  return Evaluate_Circuit(circuit_vector, default_circuit_parameters);
};
```

4.12.2.4 Evaluate_Circuit() [2/2]

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

Evaluate the circuit with given parameters.

Function to evaluate the objective function.

Parameters

circuit_vector	Circuit vector
parameters	Circuit parameters

Returns

double Objective function value

```
{
  // 1. Create a circuit object
  Circuit circuit(circuit_vector);

  // 2. Calculate the circuit
  Calculate_Circuit(circuit, parameters);

```

```
  double objective = parameters.get_money * circuit.units[circuit.num_units - 2].input.ger -
parameters.lose_money * circuit.units[circuit.num_units - 2].input.waste;

  return objective;
}
```

4.13 CUnit.cpp File Reference

CUnit class implementation.

`#include "CUnit.h"`

4.13.1 Detailed Description

CUnit class implementation.

Author

your name (you@domain.com)

Copyright

Copyright (c) 2023

4.14 DirectedGraph.cpp File Reference

Directed graph implementation.

`#include "DirectedGraph.hpp"`

Functions

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

  Overload the << operator.

4.14.1 Detailed Description

Directed graph implementation.

Author

your name (you@domain.com)

Copyright

Copyright (c) 2023

4.14.2 Function Documentation

4.14.2.1 operator<<()

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

Overload the << operator.

Parameters

os	Output stream
G	Directed graph

Returns

std::ostream& Output stream

```
{
    for (int u = 0; u < G.getNumVertices(); u++)
    {
        for (int v : G.getNeighbors(u))
        {
            os « u « " " « v « "\n";
        }
    }
    return os;
}
```

4.15 GeneticAlgorithm.cpp File Reference

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

Functions

- double RandomDouble (double min, double max)
- int RandomInt (int min, int max)

4.15.1 Function Documentation

4.15.1.1 RandomDouble()

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

4.15.1.2 RandomInt()

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

4.16 main.cpp File Reference

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

Classes

- struct CircuitOptimizer

Functions

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

Variables

- CircuitOptimizer myCircuit

4.16.1 Function Documentation

4.16.1.1 fitness()

```
double fitness (
            const std::vector< int > & x )
```

```
{
    std::vector<int> circuit_vector(x.size() + 1);
    circuit_vector[0] = myCircuit.feeder;
    for (int i = 0; i < x.size(); ++i)
    {
        circuit_vector[i + 1] = x[i];
    }
    return Evaluate_Circuit(circuit_vector, parameter);
}
```

4.16.1.2 main()

```
int main ( )
```

< Input flow rate of the mineral

< Input flow rate of the waste

< Money earned per unit of mineral

< Money lost per unit of waste

```
{
```

```
    /****************************  Analysis Settings *****************************/
    // Setup the circuit parameters
    parameter.input_Fger = 10.0;
    parameter.input_Fwaste = 100.0;
    parameter.get_money = 100.0;
    parameter.lose_money = 500.0;
    parameter.tolerance = 1e-6;

    // Setup the GA
    int POP_SIZE = 200;
    int MAX_GENERATION = 2000;
    double CROSSOVER_RATE = 0.9;
    double MUTATION_RATE = 0.01;
    int CONV_GEN = MAX_GENERATION;
    int ELITE = POP_SIZE * 0.1;
    CrossoverType type = CrossoverType::UNIFORM;
    /*****************************************************************************/

    // Setup the cicuit optimiser
    CircuitOptimizer myCircuit;

    // Don't change these
    int CHROMOSOME_SIZE = 2 * myCircuit.num_units;
    int CHROMOMIN = 0;
    int CHROMOMAX = myCircuit.num_units + 2;

    GA<int> ga(POP_SIZE, CHROMOSOME_SIZE, CHROMOMIN, CHROMOMAX, MAX_GENERATION, CROSSOVER_RATE,
 MUTATION_RATE);
    ga.CONV_GEN = CONV_GEN;
    ga.ELITISM = ELITE;
    ga.setFitness(fitness);
    ga.setValidity(validity);

    double wtime = omp_get_wtime();
    auto bestIndividual = ga.optimize(type);
    wtime = omp_get_wtime() - wtime;
    std::cout « "Time taken: " « wtime « "s\n";

    ga.toFile("../out/");

    std::vector<int> bestCircuitVector = bestIndividual.chromosome;
    bestCircuitVector.insert(bestCircuitVector.begin(), myCircuit.feeder);

    Circuit bestCircuit(bestCircuitVector);

    bestCircuit.toFile();

    return 0;
}
```

4.16.1.3 validity()

```
bool validity (
        const std::vector< int > & x )
{
    std::vector<int> circuit_vector(x.size() + 1);
    circuit_vector[0] = myCircuit.feeder;
    for (int i = 0; i < x.size(); ++i)
    {
        circuit_vector[i + 1] = x[i];
    }

    return Circuit::Check_Validity(circuit_vector);
}
```

4.16.2 Variable Documentation

4.16.2.1 myCircuit

```
CircuitOptimizer myCircuit
```

4.17 test_all.cpp File Reference

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

Functions

- int main ()

4.17.1 Function Documentation

4.17.1.1 main()

```
int main ( )
           {
    // set a random seed for consistent results
    srand(42);

    int POPULATION_SIZE = 100;
    int CHROMOSOME_SIZE = 10;
    int CHROMOMIN = 0;
    int CHROMOMAX = 10;
    int MAX_GENERATION = 1500;
    int CONV_GEN = 1500;
    double CROSSOVER_RATE = 0.8;
    double MUTATION_RATE = 0.5;

    UnitTestGA test_ga(POPULATION_SIZE,
        CHROMOSOME_SIZE,
        CHROMOMIN,
        CHROMOMAX,
        MAX_GENERATION,
        CROSSOVER_RATE,
        MUTATION_RATE
    );
    test_ga.run();

    UnitTestCrossover x;
    x.test_one_pt_crossover();
    x.test_two_pts_crossover();
    x.test_mult_pts_crossover();
    x.test_uniform_crossover();

    UnitTestCircuit circuit;
    circuit.run();


}
```

4.18 test_Circuit.cpp File Reference

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

4.19 test_Circuit.cpp

Go to the documentation of this file.

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

UnitTestCircuit::UnitTestCircuit() {}
UnitTestCircuit::~UnitTestCircuit() {}

void UnitTestCircuit::test_circuit_evaluation(){
    std::cout « "============================Test Circuit Evaluation============================\n";

    std::vector<int> circuit_vector = {4, 5, 1, 2, 4, 0, 1, 1, 6, 1, 3};
    bool validity = Circuit::Check_Validity(circuit_vector);
    assert(validity==true);

    Circuit_Parameters parameters;
    parameters.tolerance = 1e-6;

    double value = Evaluate_Circuit(circuit_vector, parameters);
    std::cout « value « std::endl;
    assert(std::fabs(value - 107.204) <= 1e-3);
    std::cout « "completed" « std::endl;
}

void UnitTestCircuit::test_circuit_validity(){
    std::cout « "============================Test Circuit Validity============================\n";

    // Cases when there are 3 units
    // Case 1: self-recycle
    std::vector<int> circuit_vector1 = {0, 1, 2, 0, 2, 1, 0};
    // Case 2: points to itself
    std::vector<int> circuit_vector2 = {0, 1, 2, 1, 2, 3, 4};
    // Case 3: there are units not ending up in 2 destinations
    std::vector<int> circuit_vector3 = {0, 1, 1, 3, 2, 1, 4};
    // Case 4: there is a unit not accessible from the feed
    std::vector<int> circuit_vector4 = {0, 1, 4, 3, 4, 3, 0};
    bool check1 = Circuit::Check_Validity(circuit_vector1);
    bool check2 = Circuit::Check_Validity(circuit_vector2);
    bool check3 = Circuit::Check_Validity(circuit_vector3);
    bool check4 = Circuit::Check_Validity(circuit_vector4);
    assert(check1==false);
    assert(check2==false);
    assert(check3==false);
    assert(check4==false);
    std::cout « "completed" « std::endl;
}

void UnitTestCircuit::test_circuit_unit() {
    std::cout « "============================Test Circuit Unit============================\n";

    CUnit test_unit;

    test_unit.input.ger = 20;
    test_unit.input.waste = 80;

    test_unit.calculateOutputFlowRates(test_unit.input);

    double eps = 1e-3;

    bool check1 = fabs(test_unit.conc.ger - 2.6087) < eps;
    bool check2 = fabs(test_unit.conc.waste - 1.18227) < eps;
    bool check3 = fabs(test_unit.tail.ger - 17.3913) < eps;
    bool check4 = fabs(test_unit.tail.waste - 78.8177) < eps;

    assert(check1==true);
    assert(check2==true);
    assert(check3==true);
    assert(check4==true);
    std::cout « "completed" « std::endl;
}

void UnitTestCircuit::run(){
    test_circuit_evaluation();
    test_circuit_validity();
    test_circuit_unit();
}

```

4.20 test_CircuitEvaluation.cpp File Reference

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

Functions

- void test ()
- int main ()

4.20.1 Function Documentation

4.20.1.1 main()

```
int main ( )
{
    test();
    std::vector<int> circuit_vector = {4, 5, 1, 2, 4, 0, 1, 1, 6, 1, 3};
    bool validity = Circuit::Check_Validity(circuit_vector);
    if (!validity)
    {
        return 1;
    }
    std::cout « "Circuit is " « validity « std::endl;
    Circuit_Parameters parameters;
    parameters.tolerance = 1e-6;
    double value = Evaluate_Circuit(circuit_vector, parameters);
    std::cout « value « std::endl;
    if (std::fabs(value - 107.204) > 1e-3)
    {
        return 1;
    }

    return 0;
}
```

4.20.1.2 test()

```
void test ( )
{

    std::vector<int> circuit_vector = {0, 4, 6, 2, 4, 3, 4, 5, 4, 1, 0};
    bool validity = Circuit::Check_Validity(circuit_vector);
    Circuit circuit(circuit_vector);
    circuit.toFile();
    Circuit_Parameters parameters;
    parameters.tolerance = 1e-3;
    double value = Evaluate_Circuit(circuit_vector, parameters);
    std::cout « validity « " " « value « std::endl;
}
```

4.21 test_CircuitValidity.cpp File Reference

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

Functions

- bool testValidity ()
- int main ()

4.21.1 Function Documentation

4.21.1.1 main()

```
int main ( )
{
    // auto val = testValidity();
    // std::cout « val « std::endl;
    // Case 1
    std::vector<int> circuit_vector1 = {0, 1, 2, 0, 2, 1, 0};
    // Case 2: points to itself
    std::vector<int> circuit_vector2 = {0, 1, 2, 1, 2, 3, 4};
    // Case 3: there are units not ending up in 2 destinations
    std::vector<int> circuit_vector3 = {0, 1, 1, 3, 2, 1, 4};
    // Case 4: there is a unit not accessible from the feed
    std::vector<int> circuit_vector4 = {0, 1, 4, 3, 4, 3, 0};
    bool check1 = Circuit::Check_Validity(circuit_vector1);
    bool check2 = Circuit::Check_Validity(circuit_vector2);
    bool check3 = Circuit::Check_Validity(circuit_vector3);
    bool check4 = Circuit::Check_Validity(circuit_vector4);
    if (!check1 && !check2 && !check3 && !check4)
    {
        return 0;
    }
    else
    {
        return 1;
    }
    return 0;
}
```

4.21.1.2 testValidity()

```
bool testValidity ( )
{
    // std::vector<int> test = {0, 3, 1, 0, 2, 1, 4};
    // std::vector<int> test = {0, 2, 1, 0, 4, 3, 0};
    // std::vector<int> test = {0, 2, 1, 3, 2, 3, 4};
    // std::vector<int> test = {2, 10, 6, 8, 4, 8, 7, 8, 1, 8, 11, 6, 2, 0, 8, 8, 3, 6, 9, 6, 5};
    // std::vector<int> test = {0, 6, 7, 2, 6, 10, 1, 1, 8, 6, 5, 6, 9, 1, 3, 6, 4, 1, 0, 6, 8};
    std::vector<int> test = {0, 8, 3, 7, 2, 7, 4, 8, 5, 10, 7, 8, 9, 8, 11, 4, 8, 7, 1, 8, 6};
    Circuit c(test);
    c.toFile();
    bool check = Circuit::Check_Validity(test);
    return check;
}
```

4.22 test_Crossover.cpp File Reference

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

Functions

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

Variables

- Crossover< int > crossover

4.22.1 Function Documentation

4.22.1.1 printChromosome()

```
template<typename T >
void printChromosome (
            const std::vector< T > & chromosome )
{
    std::cout « "{ ";
    for (const auto &gene : chromosome)
    {
        std::cout « gene « " ";
    }
    std::cout « "}\n";
}
```

4.22.2 Variable Documentation

4.22.2.1 crossover

```
Crossover<int> crossover
```

4.23 test_Crossover.cpp

Go to the documentation of this file.

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

// Function for pretty printing a chromosome
template <typename T>
void printChromosome(const std::vector<T> &chromosome)
{
    std::cout « "{ ";
    for (const auto &gene : chromosome)
    {
        std::cout « gene « " ";
    }
    std::cout « "}\n";
}

Crossover<int> crossover;

UnitTestCrossover::UnitTestCrossover()
{
    std::cout « "                                    Test Crossover Start
\n";
}
UnitTestCrossover::~UnitTestCrossover() {}

void UnitTestCrossover::test_one_pt_crossover()
```

```
{
    std::cout « "============================Test One-Point Crossover method============================\n";

    std::vector<int> chromosome1 = {1, 2, 3, 4, 5};
    std::vector<int> chromosome2 = {6, 7, 8, 9, 10};

    Individual<int> parent1, parent2, child1, child2;

    parent1.chromosome = chromosome1;
    parent2.chromosome = chromosome2;

    child1.chromosome.resize(chromosome1.size());
    child2.chromosome.resize(chromosome2.size());

    crossover.one_pt_crossover(parent1, parent2, child1, child2);

    std::cout « "Parent 1: ";
    printChromosome(parent1.chromosome);

    std::cout « "Parent 2: ";
    printChromosome(parent2.chromosome);

    std::cout « "Child 1: ";
    printChromosome(child1.chromosome);

    std::cout « "Child 2: ";
    printChromosome(child2.chromosome);

    // Check that the chromosomes have been altered
    assert(child1.chromosome != parent1.chromosome);
    assert(child1.chromosome != parent2.chromosome);
    assert(child2.chromosome != parent1.chromosome);
    assert(child2.chromosome != parent2.chromosome);
}

void UnitTestCrossover::test_two_pts_crossover()
{
    std::cout « "============================Test Two-Points Crossover method============================\n";

    std::vector<int> chromosome1 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    std::vector<int> chromosome2 = {10, 11, 12, 13, 14, 15, 16, 17, 18, 19};

    Individual<int> parent1, parent2, child1, child2;

    parent1.chromosome = chromosome1;
    parent2.chromosome = chromosome2;

    child1.chromosome.resize(chromosome1.size());
    child2.chromosome.resize(chromosome2.size());

    crossover.two_pts_crossover(parent1, parent2, child1, child2);

    std::cout « "Parent 1: ";
    printChromosome(parent1.chromosome);

    std::cout « "Parent 2: ";
    printChromosome(parent2.chromosome);

    std::cout « "Child 1: ";
    printChromosome(child1.chromosome);

    std::cout « "Child 2: ";
    printChromosome(child2.chromosome);

    // Check that the chromosomes have been altered
    assert(child1.chromosome != parent1.chromosome);
    assert(child1.chromosome != parent2.chromosome);
    assert(child2.chromosome != parent1.chromosome);
    assert(child2.chromosome != parent2.chromosome);
}

void UnitTestCrossover::test_mult_pts_crossover()
{
    std::cout « "===========================Test Multi-Points Crossover method===========================\n";

    std::vector<int> chromosome1 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    std::vector<int> chromosome2 = {10, 11, 12, 13, 14, 15, 16, 17, 18, 19};

    Individual<int> parent1, parent2, child1, child2;

    parent1.chromosome = chromosome1;
    parent2.chromosome = chromosome2;

    child1.chromosome.resize(chromosome1.size());
```

```
    child2.chromosome.resize(chromosome2.size());

    crossover.mult_pts_crossover(parent1, parent2, child1, child2);

    std::cout « "Parent 1: ";
    printChromosome(parent1.chromosome);

    std::cout « "Parent 2: ";
    printChromosome(parent2.chromosome);

    std::cout « "Child 1: ";
    printChromosome(child1.chromosome);

    std::cout « "Child 2: ";
    printChromosome(child2.chromosome);

    // Check that the chromosomes have been altered
    assert(child1.chromosome != parent1.chromosome);
    assert(child1.chromosome != parent2.chromosome);
    assert(child2.chromosome != parent1.chromosome);
    assert(child2.chromosome != parent2.chromosome);
}

void UnitTestCrossover::test_uniform_crossover()
{
    std::cout « "============================Test Uniform Crossover
method==============================\n";

    std::vector<int> chromosome1 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    std::vector<int> chromosome2 = {9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
    std::vector<int> expected_chromosome1 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    std::vector<int> expected_chromosome2 = {9, 8, 7, 6, 5, 4, 3, 2, 1, 0};

    Individual<int> parent1, parent2, child1, child2;

    parent1.chromosome = chromosome1;
    parent2.chromosome = chromosome2;

    child1.chromosome.resize(chromosome1.size());
    child2.chromosome.resize(chromosome2.size());

    crossover.uniform_crossover(parent1, parent2, child1, child2);

    // Check that the chromosomes have been altered
    assert(child1.chromosome != expected_chromosome1);
    assert(child2.chromosome != expected_chromosome2);
    std::cout « "Results match expected outputs.\n";
}
```

4.24 test_Cunit.cpp File Reference

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

Functions

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

4.24.1 Function Documentation

4.24.1.1 main()

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

```
{

    CUnit test_unit;

    test_unit.input.ger = 20;
    test_unit.input.waste = 80;

    test_unit.calculateOutputFlowRates(test_unit.input);

    double eps = 1e-3;

    bool check1 = fabs(test_unit.conc.ger - 2.6087) < eps;
    bool check2 = fabs(test_unit.conc.waste - 1.18227) < eps;
    bool check3 = fabs(test_unit.tail.ger - 17.3913) < eps;
    bool check4 = fabs(test_unit.tail.waste - 78.8177) < eps;
    if (check1 && check2 && check3 && check4)
    {
        return 0;
    }
    else
    {
        return 1;
    }

    return 0;
}
```

4.25 test_DirectedGraph.cpp File Reference

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

Functions

- int testDigraph ()
- int testDFS ()
- int main ()

4.25.1 Function Documentation

4.25.1.1 main()

```
int main ( )
{
    int STATUS = 0;
    STATUS += testDigraph();
    STATUS += testDFS();
    return STATUS;
}
```

4.25.1.2 testDFS()

```
int testDFS ( )
{
    DirectedGraph G(5);
    G.addEdge(0, 1);
    G.addEdge(0, 2);
    G.addEdge(1, 2);
    G.addEdge(2, 0);
    G.addEdge(2, 3);
    G.addEdge(3, 3);

    DepthFirstSearch dfs(G, 2);
    dfs.dfs(G, 2);
```

```
    bool visited = dfs.visited(3);
    if (!visited)
        return 1;
    visited = dfs.visited(4);
    if (visited)
        return 1;
    visited = dfs.visited(2);
    if (!visited)
        return 1;
    return 0;
}
```

4.25.1.3 testDigraph()

```
int testDigraph ( )
{

    DirectedGraph G(5);
    G.addEdge(0, 1);
    G.addEdge(0, 2);
    G.addEdge(1, 2);
    G.addEdge(2, 0);
    G.addEdge(2, 3);
    G.addEdge(3, 3);

    int v = G.getNumVertices();
    int e = G.getNumEdges();
    std::vector<int> neighbors = G.getNeighbors(2);

    if (v != 5)
        return 1;
    if (e != 6)
        return 1;
    if (neighbors.size() != 2)
        return 1;
    if (neighbors[0] != 0)
        return 1;
    if (neighbors[1] != 3)
        return 1;

    return 0;
}
```

4.26 test_GeneticAlgorithm.cpp File Reference

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

Functions

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

4.26.1 Function Documentation

4.26.1.1 Rosenbrock()

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

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

4.26.1.2 validity()

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

4.27 test_GeneticAlgorithm.cpp

Go to the documentation of this file.

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

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

double Rosenbrock(const std::vector<int> &x)
{
    double fx = 0.0;
    for (size_t i = 0; i < x.size() - 1; ++i)
    {
        double term1 = std::pow(x[i + 1] - std::pow(x[i], 2), 2);
        double term2 = std::pow(1 - x[i], 2);
        fx += 100.0 * term1 + term2;
    }
    return 1.0 / (1.0 + fx); // turn a minimization problem into a maximization problem
}

UnitTestGA::~UnitTestGA(){};

void UnitTestGA::test_mutation() {
    std::cout « "============================Test Mutation============================\n";
    std::vector<int> init_chromosome = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    Individual<int> init_individual(init_chromosome);
    Individual<int> result_individual(init_chromosome);
    ga->MUTATION_RATE = 0.5;

    ga->mutate(result_individual);
    assert(init_individual.chromosome != result_individual.chromosome);
    std::cout « "completed" « std::endl;
}

void UnitTestGA::test_select_parent() {
    std::cout « "============================Test Select Parent============================\n";
    int result;
    std::vector<double> comulative_fitness = {-1.0, 1.0, 1.0};

    result = ga->selectParent(comulative_fitness);
    assert(result == 1);
    std::cout « "completed" « std::endl;
}

void UnitTestGA::test_global_optimum() {
    std::cout « "============================Test Global Optimum============================\n";
    // configure GA
    ga->setValidity(validity);
    ga->setFitness(Rosenbrock);
    ga->verbose = false;
    CrossoverType type = CrossoverType::UNIFORM;
    // Known global optimum for Rosenbrock function
    std::vector<int> expected_chromosome = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1};
```

```
    int success_count = 0;
    int num_runs = 10;

    // Run for 10 times to see if the algorithm can find the global optimum
    // if 50% of the time can find the global optimum, then the test is passed
    for (int i = 0; i < num_runs; i++) {
        auto best = ga->optimize(type);
        if (best.chromosome == expected_chromosome) {
            success_count++;
        }
    }
    assert(success_count >= num_runs / 2);
    std::cout « "successfully found global optimum for Rosenbrock: " « success_count;
    std::cout « "/" « num_runs « " times!" « std::endl;
    std::cout « "completed" « std::endl;
}

void UnitTestGA::run() {
    test_mutation();
    test_select_parent();
    test_global_optimum();
}

```

4.28 test_Individual.cpp File Reference

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

Functions

- int main ()

4.28.1 Function Documentation

4.28.1.1 main()

int main ( )

```
        {
    UnitTestIndividual u;
    u.test_constructor();
    // u.test_calculateFitness();
    u.test_Get_Set_fitness();
    u.test_Get_vectorLength();
}
```

4.29 test_RandomCircuit.cpp File Reference

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

Functions

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

4.29.1 Function Documentation

4.29.1.1 generateRandomVector()

```
std::vector< int > generateRandomVector (
            int size,
            int minRange,
            int maxRange )
{
    std::random_device rd;
    std::mt19937 rng(rd());
    std::uniform_int_distribution<int> dist(minRange, maxRange);

    std::vector<int> randomVector(size);

    for (int i = 0; i < size; i++)
    {
        randomVector[i] = dist(rng);
    }

    return randomVector;
}
```

4.29.1.2 main()

```
int main ( )
{
    struct Circuit_Parameters default_circuit_parameters;
    bool validity = false;
    int feeder = 0;
    int num_units = 5;
    std::vector<int> circuit_vector;
    circuit_vector.push_back(feeder);
    while (!validity)
    {
        std::vector<int> cvec = generateRandomVector(2 * num_units, 0, num_units + 1);
        for (const auto &i : cvec)
        {
            circuit_vector.push_back(i);
        }
        validity = Circuit::Check_Validity(circuit_vector);
        if (!validity)
        {
            circuit_vector.clear();
            circuit_vector.push_back(feeder);
        }
    }
    double value = Evaluate_Circuit(circuit_vector, default_circuit_parameters);
    Circuit circuit(circuit_vector);
    circuit.toFile();
    std::cout « "Given test value:" « value « std::endl;
}
```

4.30 Unittests.h File Reference

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

Classes

- class UnitTestGA
- class UnitTestIndividual
- class UnitTestCrossover
- class UnitTestCircuit

4.31 Unittests.h

Go to the documentation of this file.

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


class UnitTestGA
{
public:
    // SetUp
    int POPULATION_SIZE = 100;
    int CHROMOSOME_SIZE = 10;
    int CHROMOMIN = 0;
    int CHROMOMAX = 10;
    int MAX_GENERATION = 2000;
    int CONV_GEN = 2000;
    double CROSSOVER_RATE = 0.8;
    double MUTATION_RATE = 0.15;

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

    UnitTestGA(int populationSize, int chromosomeSize, int chromomin, int chromomax,
               int maxGeneration, double crossoverRate, double mutationRate)
              : POPULATION_SIZE(populationSize), CHROMOSOME_SIZE(chromosomeSize),
CHROMOMIN(chromomin),
                CHROMOMAX(chromomax), MAX_GENERATION(maxGeneration), CROSSOVER_RATE(crossoverRate),
                MUTATION_RATE(mutationRate) {};


    ~UnitTestGA();

    void test_mutation();
    void test_select_parent();

    void test_global_optimum();

    void run();

};

class UnitTestIndividual
{
public:
    UnitTestIndividual();
    ~UnitTestIndividual();

    void test_constructor();

    // void test_calculateFitness();

    void test_Get_Set_fitness();

    void test_Get_numUnits();

    void test_Get_vectorLength();

protected:
    Individual<int> individual;
```

```
};

class UnitTestCrossover
{
public:
    UnitTestCrossover();
    ~UnitTestCrossover();

    void test_one_pt_crossover();

    void test_two_pts_crossover();

    void test_mult_pts_crossover();

    void test_uniform_crossover();
};

class UnitTestCircuit
{
public:
    UnitTestCircuit();
    ~UnitTestCircuit();

    void test_circuit_evaluation();
    void test_circuit_validity();
    void test_circuit_unit();
    void run();
};
```

Index