# ParallelCGP

1.0.0

Generated by Doxygen 1.13.2

# Chapter 1

# ParallelCGP

Završni rad na FER-u u akademskoj godini 2024/2025

# Chapter 2

# Hierarchical Index

## 2.1 Class Hierarchy

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

# Chapter 3

# Class Index

## 3.1 Class List

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

# Chapter 4

# File Index

## 4.1 File List

Here is a list of all documented files with brief descriptions:

# Chapter 5

# Class Documentation

## 5.1 parallel_cgp::ADParam Struct Reference

`#include <ADTester.hpp>`

### Public Member Functions

- ADParam (int gens, int rows, int cols, int levels, int pop)

### Public Attributes

- int gens
- int rows
- int cols
- int levels
- int pop

### 5.1.1 Detailed Description

Struktura koja se koristi za upravljanje test parametara.

Definition at line 12 of file ADTester.hpp.

### 5.1.2 Constructor & Destructor Documentation

#### 5.1.2.1 ADParam() [1/2]

`parallel_cgp::ADParam::ADParam () [inline]`

Definition at line 13 of file ADTester.hpp.

#### 5.1.2.2 ADParam() [2/2]

```
parallel_cgp::ADParam::ADParam (
            int gens,
            int rows,
            int cols,
            int levels,
            int pop)  [inline]
```

Definition at line 14 of file ADTester.hpp.

### 5.1.3 Member Data Documentation

#### 5.1.3.1 cols

```
int parallel_cgp::ADParam::cols
```

Broj stupaca za CGP.

Definition at line 20 of file ADTester.hpp.

#### 5.1.3.2 gens

```
int parallel_cgp::ADParam::gens
```

Broj generacija po testu.

Definition at line 16 of file ADTester.hpp.

#### 5.1.3.3 levels

```
int parallel_cgp::ADParam::levels
```

Broj razina iza na koliko se nodeovi mogu spajati u CGP.

Definition at line 22 of file ADTester.hpp.

#### 5.1.3.4 pop

```
int parallel_cgp::ADParam::pop
```

Velicina populacije.

Definition at line 24 of file ADTester.hpp.

#### 5.1.3.5 rows

```
int parallel_cgp::ADParam::rows
```

Broj redova za CGP.

Definition at line 18 of file ADTester.hpp.

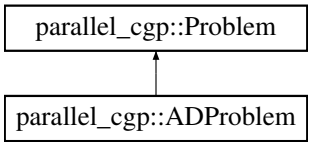
The documentation for this struct was generated from the following file:

- adProblem/ADTester.hpp

## 5.2 parallel_cgp::ADProblem Class Reference

```
#include <ADProblem.hpp>
```

Inheritance diagram for parallel_cgp::ADProblem:




### Public Member Functions

- ADProblem ()
- ADProblem (int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int POPULATION_SIZE)
- void problemRunner () override
- void printFunction () override
- void playGame ()

### Public Member Functions inherited from parallel_cgp::Problem

- virtual ~Problem ()=default
- virtual TYPE fitness (TYPE fit)

### Additional Inherited Members

### Public Attributes inherited from parallel_cgp::Problem

- CGPIndividual * bestI
- bool printGens = false

- int NUM_OPERANDS = 9
- int BI_OPERANDS = 5
- int GENERATIONS = 5000
- int ROWS = 8
- int COLUMNS = 8
- int LEVELS_BACK = 3
- int INPUTS = 6
- int OUTPUTS = 1
- int POPULATION_SIZE = 20

### 5.2.1 Detailed Description

Klasa koja predstavlja problem igranja Acey Deucey igre.

Definition at line 14 of file ADProblem.hpp.

### 5.2.2 Constructor & Destructor Documentation

#### 5.2.2.1 ADProblem() [1/2]

```
parallel_cgp::ADProblem::ADProblem ()  [inline]
```

Osnovni kostruktor koji kreira osnovnu jedinku na bazi prije zadanih vrijednosti.

Definition at line 61 of file ADProblem.hpp.

#### 5.2.2.2 ADProblem() [2/2]

```
parallel_cgp::ADProblem::ADProblem (
            int GENERATIONS,
            int ROWS,
            int COLUMNS,
            int LEVELS_BACK,
            int POPULATION_SIZE)  [inline]
```

Konstruktor koji prima sve promjenjive vrijednosti za Acey Deucey problem.

Definition at line 65 of file ADProblem.hpp.

### 5.2.3 Member Function Documentation

#### 5.2.3.1 playGame()

```
void ADProblem::playGame ()
```

Metoda prikaze kako najbolja jedinka igra jednu partiju igre.

Definition at line 194 of file ADProblem.cpp.

#### 5.2.3.2 printFunction()

```
void ADProblem::printFunction ()  [override], [virtual]
```

Metoda za ispis na kraju dobivene funkcije.

Implements parallel_cgp::Problem.

Definition at line 34 of file ADProblem.cpp.

#### 5.2.3.3 problemRunner()

```
void ADProblem::problemRunner ()  [override], [virtual]
```

Metoda za pokretanje problema.

Implements parallel_cgp::Problem.

Definition at line 116 of file ADProblem.cpp.

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

- adProblem/ADProblem.hpp
- adProblem/ADProblem.cpp

## 5.3 parallel_cgp::BoolParam Struct Reference

```
#include <BoolTester.hpp>
```

### Public Member Functions

- BoolParam (int gens, int rows, int cols, int levels, int pop)

### Public Attributes

- int gens
- int rows
- int cols
- int levels
- int pop

### 5.3.1 Detailed Description

Struktura koja se koristi za upravljanje test parametara.

Definition at line 12 of file BoolTester.hpp.

### 5.3.2 Constructor & Destructor Documentation

#### 5.3.2.1 BoolParam() [1/2]

```
parallel_cgp::BoolParam::BoolParam ()  [inline]
```

Definition at line 13 of file BoolTester.hpp.

#### 5.3.2.2 BoolParam() [2/2]

```
parallel_cgp::BoolParam::BoolParam (
            int gens,
            int rows,
            int cols,
            int levels,
            int pop)  [inline]
```

Definition at line 14 of file BoolTester.hpp.

### 5.3.3 Member Data Documentation

#### 5.3.3.1 cols

```
int parallel_cgp::BoolParam::cols
```

Broj stupaca za CGP.

Definition at line 19 of file BoolTester.hpp.

#### 5.3.3.2 gens

```
int parallel_cgp::BoolParam::gens
```

Definition at line 15 of file BoolTester.hpp.

#### 5.3.3.3 levels

```
int parallel_cgp::BoolParam::levels
```

Broj razina iza na koliko se nodeovi mogu spajati u CGP.

Definition at line 21 of file BoolTester.hpp.

#### 5.3.3.4 pop

```
int parallel_cgp::BoolParam::pop
```

Velicina populacije.

Definition at line 23 of file BoolTester.hpp.

#### 5.3.3.5 rows

```
int parallel_cgp::BoolParam::rows
```

Broj redova za CGP.

Definition at line 17 of file BoolTester.hpp.

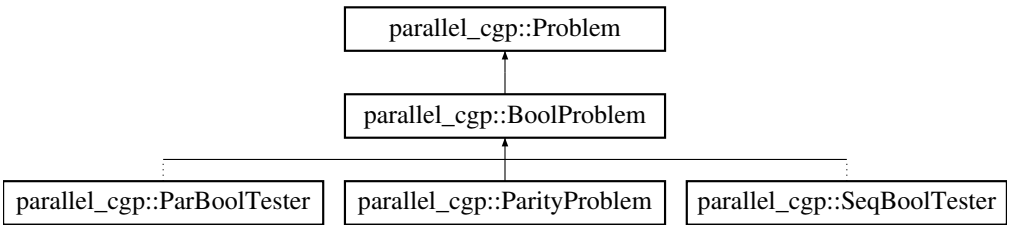
The documentation for this struct was generated from the following file:

- boolProblem/BoolTester.hpp

## 5.4 parallel_cgp::BoolProblem Class Reference

```
#include <BoolProblem.hpp>
```

Inheritance diagram for parallel_cgp::BoolProblem:




### Public Member Functions

- BoolProblem ()
- BoolProblem (int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int POPULATION_SIZE)
- BoolProblem (int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int POPULATION_SIZE, std::function< int(std::bitset< INPUTS > in)> boolFunc)
- void problemRunner () override
- void printFunction () override

### Public Member Functions inherited from parallel_cgp::Problem

- virtual ~Problem ()=default
- virtual TYPE fitness (TYPE fit)

### Protected Member Functions

- TYPE computeNode (int operand, TYPE value1, TYPE value2)
- TYPE fitness (std::bitset< INPUTS > input, TYPE res)
- void problemSimulator (CGPIndividual &individual, TYPE &fit)
- std::string evalFunction (int CGPNodeNum) override

**Protected Attributes**

- int GENERATIONS = 5000
- int ROWS = 10
- int COLUMNS = 10
- int LEVELS_BACK = 3
- int POPULATION_SIZE = 15
- bool isSimulated = false
- bool useFunc = true
- std::function< int(std::bitset< INPUTS > in)> boolFunc
- std::function< int(std::bitset< INPUTS > in)> parityFunc

**Static Protected Attributes**

- static const int NUM_OPERANDS = 4
- static const int BI_OPERANDS = 4
- static const int INPUTS = 7
- static const int OUTPUTS = 1

**Additional Inherited Members**

## Public Attributes inherited from parallel_cgp::Problem

- CGPIndividual * bestI
- bool printGens = false

- int NUM_OPERANDS = 9
- int BI_OPERANDS = 5
- int GENERATIONS = 5000
- int ROWS = 8
- int COLUMNS = 8
- int LEVELS_BACK = 3
- int INPUTS = 6
- int OUTPUTS = 1
- int POPULATION_SIZE = 20

### 5.4.1 Detailed Description

Klasa koja opisuje problem pronalaska boolean funkcije.

Definition at line 15 of file BoolProblem.hpp.

### 5.4.2 Constructor & Destructor Documentation

#### 5.4.2.1 BoolProblem() [1/3]

```
parallel_cgp::BoolProblem::BoolProblem ()  [inline]
```

Osnovni kostruktor koji kreira osnovnu jedinku na bazi prije zadanih vrijednosti.

Definition at line 65 of file BoolProblem.hpp.

#### 5.4.2.2 BoolProblem() [2/3]

```
parallel_cgp::BoolProblem::BoolProblem (
            int GENERATIONS,
            int ROWS,
            int COLUMNS,
            int LEVELS_BACK,
            int POPULATION_SIZE)  [inline]
```

Konstruktor koji prima sve promjenjive vrijednosti za bool problem osim funkcije.
Primarno se koristi kod kreacije ParityProblem klase.

Definition at line 70 of file BoolProblem.hpp.

#### 5.4.2.3 BoolProblem() [3/3]

```
parallel_cgp::BoolProblem::BoolProblem (
            int GENERATIONS,
            int ROWS,
            int COLUMNS,
            int LEVELS_BACK,
            int POPULATION_SIZE,
            std::function< int(std::bitset< INPUTS > in)> boolFunc)  [inline]
```

Konstruktor koji prima sve promjenjive vrijednosti za bool problem.

Definition at line 76 of file BoolProblem.hpp.

### 5.4.3 Member Function Documentation

#### 5.4.3.1 computeNode()

```
TYPE BoolProblem::computeNode (
            int operand,
            TYPE value1,
            TYPE value2)  [protected], [virtual]
```

Funkcija u kojoj su zapisani svi moguci operandi za dani problem.

**Parameters**

| | | |
|---|---|---|
| in | *operand* | Broj operanda. |
| in | *value1* | Prva vrijednost. |
| in | *value2* | Druga vrijednost. |

Reimplemented from parallel_cgp::Problem.

Definition at line 6 of file BoolProblem.cpp.

#### 5.4.3.2 evalFunction()

```
string BoolProblem::evalFunction (
            int CGPNodeNum)  [override], [protected], [virtual]
```

Rekurzivna funkcija koja se koristi kod ispisa funckije.

**Parameters**

| | | |
|---|---|---|
| in | *CGPNodeNum* | Broj noda na koji je spojen output. |

Implements parallel_cgp::Problem.

Definition at line 35 of file BoolProblem.cpp.

#### 5.4.3.3 fitness()

```
TYPE BoolProblem::fitness (
            std::bitset< INPUTS > input,
            TYPE res)  [protected]
```

Definition at line 21 of file BoolProblem.cpp.

#### 5.4.3.4 printFunction()

```
void BoolProblem::printFunction ()  [override], [virtual]
```

Metoda za ispis na kraju dobivene funkcije.

Implements parallel_cgp::Problem.

Definition at line 28 of file BoolProblem.cpp.

#### 5.4.3.5 problemRunner()

```
void BoolProblem::problemRunner ()  [override], [virtual]
```

Metoda za pokretanje problema.

Implements parallel_cgp::Problem.

Definition at line 81 of file BoolProblem.cpp.

#### 5.4.3.6 problemSimulator()

```
void BoolProblem::problemSimulator (
            CGPIndividual & individual,
            TYPE & fit)  [protected], [virtual]
```

Metoda koja predstavlja simulator u problemu.

**Parameters**

| | | |
|---|---|---|
| in | *individual* | Referenca na jedinku koja se koristi. |
| out | *fit* | Referenca na varijablu u koju se pohranjuje fitness. |

Reimplemented from parallel_cgp::Problem.

Definition at line 61 of file BoolProblem.cpp.

### 5.4.4 Member Data Documentation

#### 5.4.4.1 BI_OPERANDS

```
const int parallel_cgp::BoolProblem::BI_OPERANDS = 4  [static], [protected]
```

Definition at line 23 of file BoolProblem.hpp.

#### 5.4.4.2 boolFunc

```
std::function<int(std::bitset<INPUTS> in)> parallel_cgp::BoolProblem::boolFunc  [protected]
```

**Initial value:**

```
=
            [](std::bitset<INPUTS> in) { return (in[0] | ~in[1]) & ((in[0] ^ in[4]) | (in[3] & ~in[2])); }
```

Boolean funkcija koja oznacava funkciju koju CGP pokusava pronaci.

Definition at line 49 of file BoolProblem.hpp.

#### 5.4.4.3 COLUMNS

```
int parallel_cgp::BoolProblem::COLUMNS = 10  [protected]
```

Definition at line 33 of file BoolProblem.hpp.

#### 5.4.4.4 GENERATIONS

```
int parallel_cgp::BoolProblem::GENERATIONS = 5000  [protected]
```

Promjenjivi parametri za ovaj problem.
Svi su detaljno opisani u CGP klasi.

Definition at line 31 of file BoolProblem.hpp.

#### 5.4.4.5 INPUTS

```
const int parallel_cgp::BoolProblem::INPUTS = 7  [static], [protected]
```

Definition at line 24 of file BoolProblem.hpp.

#### 5.4.4.6 isSimulated

```
bool parallel_cgp::BoolProblem::isSimulated = false  [protected]
```

Parametar koji oznacava je li simulacija obavljena.

Definition at line 40 of file BoolProblem.hpp.

#### 5.4.4.7 LEVELS_BACK

```
int parallel_cgp::BoolProblem::LEVELS_BACK = 3  [protected]
```

Definition at line 34 of file BoolProblem.hpp.

#### 5.4.4.8 NUM_OPERANDS

```
const int parallel_cgp::BoolProblem::NUM_OPERANDS = 4  [static], [protected]
```

Nepromjenjivi parametri za ovaj problem.
Operandi jer ovise o funkcijama.
A broj inputa i outputa jer o njemu ovisi funkcija koja se trazi.

Definition at line 22 of file BoolProblem.hpp.

#### 5.4.4.9 OUTPUTS

```
const int parallel_cgp::BoolProblem::OUTPUTS = 1  [static], [protected]
```

Definition at line 25 of file BoolProblem.hpp.

#### 5.4.4.10 parityFunc

```
std::function<int(std::bitset<INPUTS> in)> parallel_cgp::BoolProblem::parityFunc  [protected]
```

**Initial value:**

```
=
            [](std::bitset<INPUTS> in) { return (in.count() % 2 == 0) ? 0 : 1; }
```

Parity 8bit funkcija koju CGP pokusava pronaci.

Definition at line 54 of file BoolProblem.hpp.

#### 5.4.4.11 POPULATION_SIZE

```
int parallel_cgp::BoolProblem::POPULATION_SIZE = 15  [protected]
```

Definition at line 35 of file BoolProblem.hpp.

#### 5.4.4.12 ROWS

```
int parallel_cgp::BoolProblem::ROWS = 10  [protected]
```

Definition at line 32 of file BoolProblem.hpp.

#### 5.4.4.13 useFunc

```
bool parallel_cgp::BoolProblem::useFunc = true  [protected]
```

Parametar koji oznacava koristi li se funkcija ili partiet.

Definition at line 44 of file BoolProblem.hpp.

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

- boolProblem/BoolProblem.hpp
- boolProblem/BoolProblem.cpp

## 5.5 parallel_cgp::CGP Class Reference

```
#include <CGP.hpp>
```

### Public Member Functions

- CGP (int rows, int columns, int levelsBack, int inputs, int outputs, int operands, int biOperands, int population↩ Size)
- void generatePopulation (std::vector< CGPIndividual > &population)
- void goldMutate (CGPIndividual parent, std::vector< CGPIndividual > &population)

### 5.5.1 Detailed Description

Klasa koja opisuje CGP instancu.

Definition at line 22 of file CGP.hpp.

### 5.5.2 Constructor & Destructor Documentation

#### 5.5.2.1 CGP()

```
parallel_cgp::CGP::CGP (
            int rows,
            int columns,
            int levelsBack,
            int inputs,
            int outputs,
            int operands,
            int biOperands,
            int populationSize)  [inline]
```

Konstruktor za CGP klasu.

**Parameters**

| | | |
|---|---|---|
| in | *rows* | Broj redova CGP mreze. |
| in | *columns* | Broj stupaca CGP mreze. |
| in | *levelsBack* | Broj stupaca ispred noda na koje se moze spojiti. |
| in | *inputs* | Broj ulaznih nodova. |
| in | *outputs* | Broj izlaznih nodova. |
| in | *operands* | Broj operanada koji su na raspolaganju. |
| in | *biOperands* | Broj prvog operanda koji prima jedan ulaz. |
| in | *populationSize* | Broj jedinki u populaciji. |

Definition at line 37 of file CGP.hpp.

### 5.5.3 Member Function Documentation

#### 5.5.3.1 generatePopulation()

```
void CGP::generatePopulation (
            std::vector< CGPIndividual > & population)
```

Funkcija za generiranje inicijalne populacije.
Broj jedinki u populaciji ovisi o konstanti POPULATION_SIZE.
Ostali parametri su navedeni u konstruktoru.

**Parameters**

| | | |
|---|---|---|
| out | *population* | Vector populacije koji se puni s generiranim jedinkama. |

Definition at line 6 of file CGP.cpp.

#### 5.5.3.2 goldMutate()

```
void CGP::goldMutate (
            CGPIndividual parent,
            std::vector< CGPIndividual > & population)
```

Funkcija za kreiranje nove generacije populacije na bazi roditeljske jedinke.
Koristi se **Goldman Mutacija** kojom se u roditeljskoj jedinci mutiraju geni sve dok se ne dode do gena koji se aktivno koristi. Taj gen se jos promjeni i s njime zavrsava mutacija nove jedinke.

**Parameters**

| | | |
|---|---|---|
| in | *parent* | Najbolja jedinka iz prosle generacija, roditelj za novu. |
| out | *population* | Vector populacije koji se puni s mutacijama roditelja. |

Definition at line 83 of file CGP.cpp.

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

- cgp/CGP.hpp
- cgp/CGP.cpp

## 5.6 parallel_cgp::CGPIndividual Class Reference

`#include <CGPIndividual.hpp>`

### Public Member Functions

- CGPIndividual ()
- CGPIndividual (std::vector< CGPNode > genes, std::vector< CGPOutput > outputGene, int rows, int columns, int levelsBack, int inputs, int outputs)
- CGPIndividual (std::vector< CGPNode > genes, std::vector< CGPOutput > outputGene, int rows, int columns, int levelsBack, int inputs, int outputs, bool evalDone)
- void printNodes ()
- void evaluateValue (std::vector< TYPE > input, std::function< TYPE(int, TYPE, TYPE)> &computeNode)
- void evaluateUsed ()
- bool findLoops (int nodeNum)
- void resolveLoops ()

### Public Attributes

- std::vector< CGPNode > genes
- std::vector< CGPOutput > outputGene
- std::vector< std::vector< int > > branches
- int rows
- int columns
- int levelsBack
- int inputs
- int outputs
- int evalDone

### 5.6.1 Detailed Description

Klasa koja reprezentira jednog CGP pojedinca.

Definition at line 21 of file CGPIndividual.hpp.

### 5.6.2 Constructor & Destructor Documentation

#### 5.6.2.1 CGPIndividual() [1/3]

```
CGPIndividual::CGPIndividual ()
```

Osnovni kostruktor koji kreira praznu jedinku.

Definition at line 6 of file CGPIndividual.cpp.

#### 5.6.2.2 CGPIndividual() [2/3]

```
parallel_cgp::CGPIndividual::CGPIndividual (
            std::vector< CGPNode > genes,
            std::vector< CGPOutput > outputGene,
            int rows,
            int columns,
            int levelsBack,
            int inputs,
            int outputs)
```

Konstruktor kojim se kreira jedinka.
Koristi se pri ucenju.

**Parameters**

| | | |
|---|---|---|
| in | *genes* | Vector gena. |
| in | *outputGene* | Vector izlaznih gena. |
| in | *rows* | Broj redova CGP mreze. |
| in | *columns* | Broj stupaca CGP mreze. |
| in | *levelsBack* | Broj stupaca ispred noda na koje se moze spojiti. |
| in | *inputs* | Broj ulaznih nodova. |
| in | *outputs* | Broj izlaznih nodova. |

#### 5.6.2.3 CGPIndividual() [3/3]

```
parallel_cgp::CGPIndividual::CGPIndividual (
            std::vector< CGPNode > genes,
            std::vector< CGPOutput > outputGene,
            int rows,
            int columns,
            int levelsBack,
            int inputs,
            int outputs,
            bool evalDone)
```

Konstruktor kojim se kreira jedinka.
Koristi se pri ucitavanju najbolje jedinke iz datoteke.
Gotovo isti kao i drugi kostruktor.

### 5.6.3 Member Function Documentation

#### 5.6.3.1 evaluateUsed()

```
void CGPIndividual::evaluateUsed ()
```

Metoda za oznacavanje koristenih gena u mrezi.

Definition at line 53 of file CGPIndividual.cpp.

#### 5.6.3.2 evaluateValue()

```
void CGPIndividual::evaluateValue (
            std::vector< TYPE > input,
            std::function< TYPE(int, TYPE, TYPE)> & computeNode)
```

Metoda za izracunavanje vrijednosti u izlaznim genima za dane ulazne vrijednosti.

**Parameters**

| | | |
|---|---|---|
| in | *input* | Vector ulaznih vrijednosti tipa TYPE (ovisno o problemu). |
| in | *computeNode* | Funkcija koja racuna izlaznu vrijednost nodeova. |

Definition at line 70 of file CGPIndividual.cpp.

#### 5.6.3.3 findLoops()

```
bool CGPIndividual::findLoops (
            int nodeNum)
```

Rekurzivna funkcija za pronalazak petlji u mrezi.

**Parameters**

| | | |
|---|---|---|
| in | *nodeNum* | Broj trenutnog noda. |

**Returns**

True ako je pronadjena petlja, inace false.

Definition at line 97 of file CGPIndividual.cpp.

#### 5.6.3.4 printNodes()

```
void CGPIndividual::printNodes ()
```

Metoda za ispis svih nodova na standardni izlaz.

Definition at line 43 of file CGPIndividual.cpp.

#### 5.6.3.5 resolveLoops()

```
void CGPIndividual::resolveLoops ()
```

Metoda za razrjesavanje petlji u mrezi.

Definition at line 126 of file CGPIndividual.cpp.

### 5.6.4 Member Data Documentation

#### 5.6.4.1 branches

```
std::vector<std::vector<int> > parallel_cgp::CGPIndividual::branches
```

2D vector koji reprezentira sve aktivne grane jedinke.
Koristi se za otklanjanje implicitnih petlji u mrezi nodeova.

Definition at line 40 of file CGPIndividual.hpp.

#### 5.6.4.2 columns

```
int parallel_cgp::CGPIndividual::columns
```

Broj stupaca u mrezi.

Definition at line 44 of file CGPIndividual.hpp.

#### 5.6.4.3 evalDone

```
int parallel_cgp::CGPIndividual::evalDone
```

Varijabla koja oznacava je li se proslo kroz mrezu i oznacilo koji se nodeovi koriste.

Definition at line 52 of file CGPIndividual.hpp.

#### 5.6.4.4 genes

```
std::vector<CGPNode> parallel_cgp::CGPIndividual::genes
```

Vector CGPNode koji reprezentira sve ulazne i gene mreze.

Definition at line 31 of file CGPIndividual.hpp.

#### 5.6.4.5 inputs

```
int parallel_cgp::CGPIndividual::inputs
```

Broj ulaznih gena.

Definition at line 48 of file CGPIndividual.hpp.

#### 5.6.4.6 levelsBack

```
int parallel_cgp::CGPIndividual::levelsBack
```

Broj stupaca ispred noda na koje se moze spojiti.

Definition at line 46 of file CGPIndividual.hpp.

#### 5.6.4.7 outputGene

```
std::vector<CGPOutput> parallel_cgp::CGPIndividual::outputGene
```

Vector CGPOutput koji reprezentira sve izlazne gene.

Definition at line 35 of file CGPIndividual.hpp.

#### 5.6.4.8 outputs

```
int parallel_cgp::CGPIndividual::outputs
```

Broj izlaznih gena.

Definition at line 50 of file CGPIndividual.hpp.

#### 5.6.4.9 rows

```
int parallel_cgp::CGPIndividual::rows
```

Broj redova u mrezi.

Definition at line 42 of file CGPIndividual.hpp.

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

- cgp/CGPIndividual.hpp
- cgp/CGPIndividual.cpp

## 5.7 parallel_cgp::CGPNode Struct Reference

```
#include <CGPNode.hpp>
```

### Public Attributes

- int operand
- int connection1
- int connection2
- bool used
- TYPE outValue

### 5.7.1 Detailed Description

Struktura koja opisuje gene mreze CGP jedinke.

Definition at line 12 of file CGPNode.hpp.

### 5.7.2 Member Data Documentation

#### 5.7.2.1 connection1

```
int parallel_cgp::CGPNode::connection1
```

Prva konekcija nodea na drugi node.

Definition at line 20 of file CGPNode.hpp.

#### 5.7.2.2 connection2

```
int parallel_cgp::CGPNode::connection2
```

Druga konekcija nodea na drugi node.

Definition at line 24 of file CGPNode.hpp.

#### 5.7.2.3 operand

```
int parallel_cgp::CGPNode::operand
```

Vrijednost koja oznacava koji se operand koristi u nodeu.

Definition at line 16 of file CGPNode.hpp.

#### 5.7.2.4 outValue

```
TYPE parallel_cgp::CGPNode::outValue
```

Izlazna vrijednost nakon racunanja vrijednosti.

Definition at line 32 of file CGPNode.hpp.

#### 5.7.2.5 used

```
bool parallel_cgp::CGPNode::used
```

Vrijednost koja oznacava koristi li se node.

Definition at line 28 of file CGPNode.hpp.

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

- cgp/CGPNode.hpp

## 5.8 parallel_cgp::CGPOutput Struct Reference

```
#include <CGPOutput.hpp>
```

### Public Attributes

- int connection
- TYPE value

### 5.8.1 Detailed Description

Struktura koja opisuje izlazne gene CGP jedinke.

Definition at line 12 of file CGPOutput.hpp.

### 5.8.2 Member Data Documentation

#### 5.8.2.1 connection

```
int parallel_cgp::CGPOutput::connection
```

Broj koji reprezentira na koji gen je spojen izlazni gen.

Definition at line 16 of file CGPOutput.hpp.

#### 5.8.2.2 value

```
TYPE parallel_cgp::CGPOutput::value
```

Izlazna vrijednost gena nakon izracuna.

Definition at line 20 of file CGPOutput.hpp.

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

- cgp/CGPOutput.hpp

## 5.9 parallel_cgp::FuncParam Struct Reference

```
#include <FuncTester.hpp>
```

#### Public Member Functions

- FuncParam (int gens, int rows, int cols, int levels, int pop, int thresh)

#### Public Attributes

- int gens
- int rows
- int cols
- int levels
- int pop
- int thresh

### 5.9.1 Detailed Description

Struktura koja se koristi za upravljanje test parametara.

Definition at line 12 of file FuncTester.hpp.

## 5.9.2 Constructor & Destructor Documentation

### 5.9.2.1 FuncParam() [1/2]

```
parallel_cgp::FuncParam::FuncParam ()  [inline]
```

Definition at line 13 of file FuncTester.hpp.

### 5.9.2.2 FuncParam() [2/2]

```
parallel_cgp::FuncParam::FuncParam (
            int gens,
            int rows,
            int cols,
            int levels,
            int pop,
            int thresh)  [inline]
```

Definition at line 14 of file FuncTester.hpp.

## 5.9.3 Member Data Documentation

### 5.9.3.1 cols

```
int parallel_cgp::FuncParam::cols
```

Broj stupaca za CGP.

Definition at line 20 of file FuncTester.hpp.

### 5.9.3.2 gens

```
int parallel_cgp::FuncParam::gens
```

Broj generacija po testu.

Definition at line 16 of file FuncTester.hpp.

### 5.9.3.3 levels

```
int parallel_cgp::FuncParam::levels
```

Broj razina iza na koliko se nodeovi mogu spajati u CGP.

Definition at line 22 of file FuncTester.hpp.

#### 5.9.3.4 pop

```
int parallel_cgp::FuncParam::pop
```

Velicina populacije.

Definition at line 24 of file FuncTester.hpp.

#### 5.9.3.5 rows

```
int parallel_cgp::FuncParam::rows
```

Broj redova za CGP.

Definition at line 18 of file FuncTester.hpp.

#### 5.9.3.6 thresh

```
int parallel_cgp::FuncParam::thresh
```

Vrijednost nakon koje se zaustavlja problem. Ako je manja od 0 onda se gledaju generacije.

Definition at line 26 of file FuncTester.hpp.

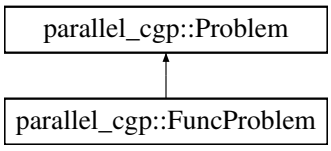
The documentation for this struct was generated from the following file:

- funcProblem/FuncTester.hpp

## 5.10 parallel_cgp::FuncProblem Class Reference

```
#include <FuncProblem.hpp>
```

Inheritance diagram for parallel_cgp::FuncProblem:




### Public Member Functions

- FuncProblem ()
- FuncProblem (int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int POPULATION_SIZE, int THRESHOLD, std::function< TYPE(TYPE x, TYPE y)> func)
- void problemRunner () override
- void printFunction () override

### Public Member Functions inherited from parallel_cgp::Problem

- virtual ~Problem ()=default
- virtual TYPE fitness (TYPE fit)

#### Additional Inherited Members

### Public Attributes inherited from parallel_cgp::Problem

- CGPIndividual * bestI
- bool printGens = false

- int NUM_OPERANDS = 9
- int BI_OPERANDS = 5
- int GENERATIONS = 5000
- int ROWS = 8
- int COLUMNS = 8
- int LEVELS_BACK = 3
- int INPUTS = 6
- int OUTPUTS = 1
- int POPULATION_SIZE = 20

## 5.10.1 Detailed Description

Klasa koja opisuje problem pronalaska funkcije.

Definition at line 14 of file FuncProblem.hpp.

## 5.10.2 Constructor & Destructor Documentation

### 5.10.2.1 FuncProblem() [1/2]

```
parallel_cgp::FuncProblem::FuncProblem ()  [inline]
```

Osnovni kostruktor koji kreira osnovnu jedinku na bazi prije zadanih vrijednosti.

Definition at line 56 of file FuncProblem.hpp.

### 5.10.2.2 FuncProblem() [2/2]

```
parallel_cgp::FuncProblem::FuncProblem (
            int GENERATIONS,
            int ROWS,
            int COLUMNS,
            int LEVELS_BACK,
            int POPULATION_SIZE,
            int THRESHOLD,
            std::function< TYPE(TYPE x, TYPE y)> func)  [inline]
```

Konstruktor koji prima sve promjenjive vrijednosti za func problem.

Definition at line 60 of file FuncProblem.hpp.

### 5.10.3 Member Function Documentation

#### 5.10.3.1 printFunction()

```
void FuncProblem::printFunction ()  [override], [virtual]
```

Metoda za ispis na kraju dobivene funkcije.

Implements parallel_cgp::Problem.

Definition at line 35 of file FuncProblem.cpp.

#### 5.10.3.2 problemRunner()

```
void FuncProblem::problemRunner ()  [override], [virtual]
```

Metoda za pokretanje problema.

Implements parallel_cgp::Problem.

Definition at line 115 of file FuncProblem.cpp.

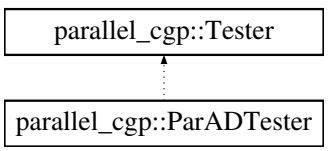
The documentation for this class was generated from the following files:

- funcProblem/FuncProblem.hpp
- funcProblem/FuncProblem.cpp

## 5.11 parallel_cgp::ParADTester Class Reference

```
#include <ADTester.hpp>
```

Inheritance diagram for parallel_cgp::ParADTester:




#### Public Member Functions

- ParADTester ()

### 5.11.1 Detailed Description

Klasa koja opisuje paralelni tester Acey Deucey problema.

Definition at line 65 of file ADTester.hpp.

### 5.11.2 Constructor & Destructor Documentation

#### 5.11.2.1 ParADTester()

```
parallel_cgp::ParADTester::ParADTester () [inline]
```

Konstruktor testera koji odmah i pokrece testiranje.
Parametar ROUNDS je opisan u Tester.hpp.

Definition at line 90 of file ADTester.hpp.

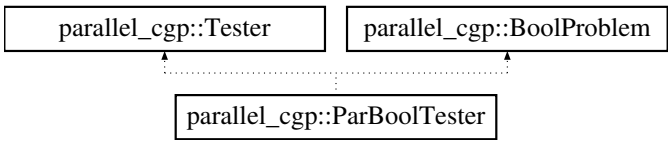
The documentation for this class was generated from the following file:

- adProblem/ADTester.hpp

## 5.12 parallel_cgp::ParBoolTester Class Reference

```
#include <BoolTester.hpp>
```

Inheritance diagram for parallel_cgp::ParBoolTester:




#### Public Member Functions

- ParBoolTester ()

### 5.12.1 Detailed Description

Klasa koja opisuje paralelni tester Bool problema.

Definition at line 71 of file BoolTester.hpp.

### 5.12.2 Constructor & Destructor Documentation

#### 5.12.2.1 ParBoolTester()

```
parallel_cgp::ParBoolTester::ParBoolTester () [inline]
```

Konstruktor testera koji odmah i pokrece testiranje.
Parametar ROUNDS je opisan u Tester.

Definition at line 100 of file BoolTester.hpp.

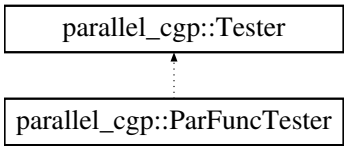
The documentation for this class was generated from the following file:

- boolProblem/BoolTester.hpp

## 5.13 parallel_cgp::ParFuncTester Class Reference

`#include <FuncTester.hpp>`

Inheritance diagram for parallel_cgp::ParFuncTester:




### Public Member Functions

- ParFuncTester ()

### 5.13.1 Detailed Description

Klasa koja opisuje sekvencijski tester Func problema.

Definition at line 74 of file FuncTester.hpp.

### 5.13.2 Constructor & Destructor Documentation

#### 5.13.2.1 ParFuncTester()

`parallel_cgp::ParFuncTester::ParFuncTester () [inline]`

Konstruktor testera koji odmah i pokrece testiranje.
Parametar ROUNDS je opisan u Tester.

Definition at line 103 of file FuncTester.hpp.

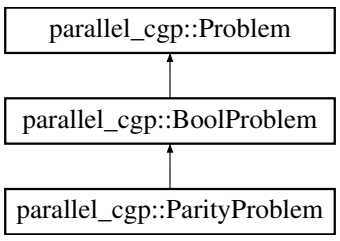
The documentation for this class was generated from the following file:

- funcProblem/FuncTester.hpp

## 5.14 parallel_cgp::ParityProblem Class Reference

`#include <BoolProblem.hpp>`

Inheritance diagram for parallel_cgp::ParityProblem:



#### Public Member Functions

- ParityProblem ()
- ParityProblem (int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int POPULATION_SIZE)

### Public Member Functions inherited from parallel_cgp::BoolProblem

- BoolProblem ()
- BoolProblem (int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int POPULATION_SIZE)
- BoolProblem (int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int POPULATION_SIZE, std::function< int(std::bitset< INPUTS > in)> boolFunc)
- void problemRunner () override
- void printFunction () override

### Public Member Functions inherited from parallel_cgp::Problem

- virtual ~Problem ()=default
- virtual TYPE fitness (TYPE fit)

#### Additional Inherited Members

### Public Attributes inherited from parallel_cgp::Problem

- CGPIndividual * bestI
- bool printGens = false

- int NUM_OPERANDS = 9
- int BI_OPERANDS = 5
- int GENERATIONS = 5000
- int ROWS = 8
- int COLUMNS = 8
- int LEVELS_BACK = 3
- int INPUTS = 6
- int OUTPUTS = 1
- int POPULATION_SIZE = 20

### Protected Member Functions inherited from parallel_cgp::BoolProblem

- TYPE computeNode (int operand, TYPE value1, TYPE value2)
- TYPE fitness (std::bitset< INPUTS > input, TYPE res)
- void problemSimulator (CGPIndividual &individual, TYPE &fit)
- std::string evalFunction (int CGPNodeNum) override

### Protected Attributes inherited from parallel_cgp::BoolProblem

- int GENERATIONS = 5000
- int ROWS = 10
- int COLUMNS = 10
- int LEVELS_BACK = 3
- int POPULATION_SIZE = 15
- bool isSimulated = false
- bool useFunc = true
- std::function< int(std::bitset< INPUTS > in)> boolFunc
- std::function< int(std::bitset< INPUTS > in)> parityFunc

### Static Protected Attributes inherited from parallel_cgp::BoolProblem

- static const int NUM_OPERANDS = 4
- static const int BI_OPERANDS = 4
- static const int INPUTS = 7
- static const int OUTPUTS = 1

## 5.14.1 Detailed Description

Klasa koja opisuje problema pariteta.

Definition at line 92 of file BoolProblem.hpp.

## 5.14.2 Constructor & Destructor Documentation

### 5.14.2.1 ParityProblem() [1/2]

```
parallel_cgp::ParityProblem::ParityProblem ()  [inline]
```

Konstruktor koji samo mijenja koja se funkcija koristi.

Definition at line 97 of file BoolProblem.hpp.

### 5.14.2.2 ParityProblem() [2/2]

```
parallel_cgp::ParityProblem::ParityProblem (
            int GENERATIONS,
            int ROWS,
            int COLUMNS,
            int LEVELS_BACK,
            int POPULATION_SIZE)  [inline]
```

Konstruktor koji prima sve promjenjive vrijednosti za parity problem.

Definition at line 101 of file BoolProblem.hpp.

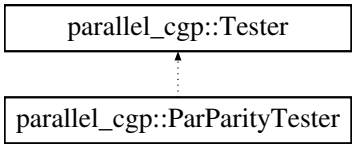
The documentation for this class was generated from the following file:

- boolProblem/BoolProblem.hpp

## 5.15 parallel_cgp::ParParityTester Class Reference

`#include <BoolTester.hpp>`

Inheritance diagram for parallel_cgp::ParParityTester:




**Public Member Functions**

- ParParityTester ()

### 5.15.1 Detailed Description

Klasa koja opisuje paralelni tester Parity problema.

Definition at line 150 of file BoolTester.hpp.

### 5.15.2 Constructor & Destructor Documentation

#### 5.15.2.1 ParParityTester()

`parallel_cgp::ParParityTester::ParParityTester () [inline]`

Konstruktor testera koji odmah i pokrece testiranje.
Parametar ROUNDS je opisan u Tester.

Definition at line 175 of file BoolTester.hpp.

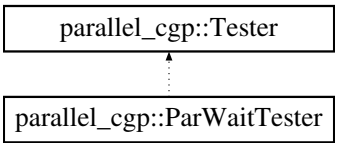
The documentation for this class was generated from the following file:

- boolProblem/BoolTester.hpp

## 5.16 parallel_cgp::ParWaitTester Class Reference

`#include <WaitTester.hpp>`

Inheritance diagram for parallel_cgp::ParWaitTester:



**Public Member Functions**

- ParWaitTester ()

### 5.16.1 Detailed Description

Klasa koja opisuje paralelni tester Wait problema.

Definition at line 67 of file WaitTester.hpp.

### 5.16.2 Constructor & Destructor Documentation

#### 5.16.2.1 ParWaitTester()

```
parallel_cgp::ParWaitTester::ParWaitTester ()  [inline]
```

Konstruktor testera koji odmah i pokrece testiranje.
Parametar ROUNDS je opisan u Tester.

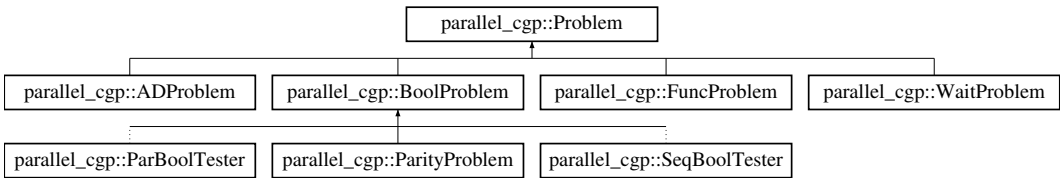
Definition at line 92 of file WaitTester.hpp.

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

- waitProblem/WaitTester.hpp

## 5.17 parallel_cgp::Problem Class Reference

Inheritance diagram for parallel_cgp::Problem:




**Public Member Functions**

- virtual ~Problem ()=default
- virtual TYPE computeNode (int operand, TYPE value1, TYPE value2)
- virtual TYPE fitness (TYPE fit)
- virtual void problemRunner ()=0
- virtual void printFunction ()=0

**Public Attributes**

- CGPIndividual * bestI
- bool printGens = false

**Promjenjivi parametri**

*Parametri koji su na raspolaganju svakom problemu.*
*Mogu se mijenjati po potrebi.*

- int NUM_OPERANDS = 9
- int BI_OPERANDS = 5
- int GENERATIONS = 5000
- int ROWS = 8
- int COLUMNS = 8
- int LEVELS_BACK = 3
- int INPUTS = 6
- int OUTPUTS = 1
- int POPULATION_SIZE = 20

## 5.17.1 Detailed Description

Definition at line 13 of file Problem.hpp.

## 5.17.2 Constructor & Destructor Documentation

### 5.17.2.1 ~Problem()

```
virtual parallel_cgp::Problem::~Problem ()  [virtual], [default]
```

Destruktor Problem objekata.

## 5.17.3 Member Function Documentation

### 5.17.3.1 computeNode()

```
virtual TYPE parallel_cgp::Problem::computeNode (
            int operand,
            TYPE value1,
            TYPE value2)  [inline], [virtual]
```

Funkcija u kojoj su zapisani svi moguci operandi za dani problem.

**Parameters**

| | | |
|---|---|---|
| in | *operand* | Broj operanda. |
| in | *value1* | Prva vrijednost. |
| in | *value2* | Druga vrijednost. |

Reimplemented in parallel_cgp::BoolProblem.

Definition at line 72 of file Problem.hpp.

#### 5.17.3.2 fitness()

```
virtual TYPE parallel_cgp::Problem::fitness (
            TYPE fit)  [inline], [virtual]
```

Funkcija koja se koristi za izracun fitnessa za odredenu jedinku.

Definition at line 99 of file Problem.hpp.

#### 5.17.3.3 printFunction()

```
virtual void parallel_cgp::Problem::printFunction ()  [pure virtual]
```

Metoda za ispis na kraju dobivene funkcije.

Implemented in parallel_cgp::ADProblem, parallel_cgp::BoolProblem, parallel_cgp::FuncProblem, and parallel_cgp::WaitProblem.

#### 5.17.3.4 problemRunner()

```
virtual void parallel_cgp::Problem::problemRunner ()  [pure virtual]
```

Metoda za pokretanje problema.

Implemented in parallel_cgp::ADProblem, parallel_cgp::BoolProblem, parallel_cgp::FuncProblem, and parallel_cgp::WaitProblem.

### 5.17.4 Member Data Documentation

#### 5.17.4.1 bestI

```
CGPIndividual* parallel_cgp::Problem::bestI
```

Najbolja jedinka nakon pokretanja problem simulatora.

Definition at line 34 of file Problem.hpp.

#### 5.17.4.2 BI_OPERANDS

```
int parallel_cgp::Problem::BI_OPERANDS = 5
```

Broj binarnih operanada (+1 iz nekog razloga).

Definition at line 49 of file Problem.hpp.

#### 5.17.4.3 COLUMNS

```
int parallel_cgp::Problem::COLUMNS = 8
```

Broj stupaca CGP mreze.

Definition at line 55 of file Problem.hpp.

#### 5.17.4.4 GENERATIONS

```
int parallel_cgp::Problem::GENERATIONS = 5000
```

Broj generacija koji se vrti.

Definition at line 51 of file Problem.hpp.

#### 5.17.4.5 INPUTS

```
int parallel_cgp::Problem::INPUTS = 6
```

Broj ulaza u CGP mrezu.

Definition at line 59 of file Problem.hpp.

#### 5.17.4.6 LEVELS_BACK

```
int parallel_cgp::Problem::LEVELS_BACK = 3
```

Broj razina unazad na koji se nodeovi mogu spojiti u CGP mrezi.

Definition at line 57 of file Problem.hpp.

#### 5.17.4.7 NUM_OPERANDS

```
int parallel_cgp::Problem::NUM_OPERANDS = 9
```

Ukupni broj operanada.

Definition at line 47 of file Problem.hpp.

#### 5.17.4.8 OUTPUTS

```
int parallel_cgp::Problem::OUTPUTS = 1
```

Broj izlaza iz CGP mrezu.

Definition at line 61 of file Problem.hpp.

#### 5.17.4.9 POPULATION_SIZE

```
int parallel_cgp::Problem::POPULATION_SIZE = 20
```

Velicina populacije.

Definition at line 63 of file Problem.hpp.

#### 5.17.4.10 printGens

```
bool parallel_cgp::Problem::printGens = false
```

Varijabla koja oznacuje hoce li se ispisivati vrijednost fitnesa za svaku generaciju.

Definition at line 39 of file Problem.hpp.

#### 5.17.4.11 ROWS

```
int parallel_cgp::Problem::ROWS = 8
```

Broj redova CGP mreze.

Definition at line 53 of file Problem.hpp.

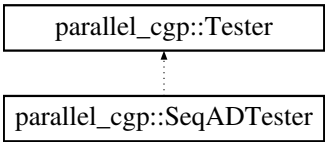
The documentation for this class was generated from the following file:

- Problem.hpp

## 5.18 parallel_cgp::SeqADTester Class Reference

```
#include <ADTester.hpp>
```

Inheritance diagram for parallel_cgp::SeqADTester:




### Public Member Functions

- SeqADTester ()

### 5.18.1 Detailed Description

Klasa koja opisuje sekvencijski tester Acey Deucey problema.

Definition at line 30 of file ADTester.hpp.

### 5.18.2 Constructor & Destructor Documentation

#### 5.18.2.1 SeqADTester()

```
parallel_cgp::SeqADTester::SeqADTester () [inline]
```

Konstruktor testera koji odmah i pokrece testiranje.
Parametar ROUNDS je opisan u Tester.

Definition at line 53 of file ADTester.hpp.

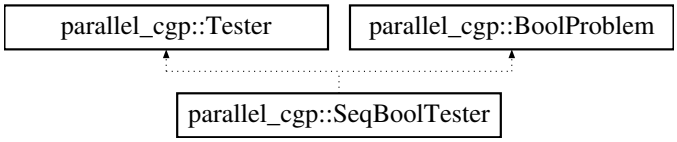
The documentation for this class was generated from the following file:

- adProblem/ADTester.hpp

## 5.19 parallel_cgp::SeqBoolTester Class Reference

```
#include <BoolTester.hpp>
```

Inheritance diagram for parallel_cgp::SeqBoolTester:




#### Public Member Functions

- SeqBoolTester ()

### 5.19.1 Detailed Description

Klasa koja opisuje sekvencijski tester Bool problema.

Definition at line 29 of file BoolTester.hpp.

### 5.19.2 Constructor & Destructor Documentation

#### 5.19.2.1 SeqBoolTester()

```
parallel_cgp::SeqBoolTester::SeqBoolTester () [inline]
```

Konstruktor testera koji odmah i pokrece testiranje.
Parametar ROUNDS je opisan u Tester.

Definition at line 56 of file BoolTester.hpp.

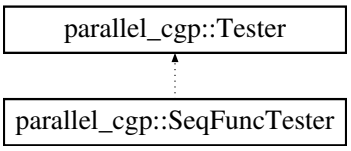
The documentation for this class was generated from the following file:

- boolProblem/BoolTester.hpp

## 5.20 parallel_cgp::SeqFuncTester Class Reference

`#include <FuncTester.hpp>`

Inheritance diagram for parallel_cgp::SeqFuncTester:




**Public Member Functions**

- SeqFuncTester ()

### 5.20.1 Detailed Description

Klasa koja opisuje sekvencijski tester Func problema.

Definition at line 32 of file FuncTester.hpp.

### 5.20.2 Constructor & Destructor Documentation

#### 5.20.2.1 SeqFuncTester()

`parallel_cgp::SeqFuncTester::SeqFuncTester () [inline]`

Konstruktor testera koji odmah i pokrece testiranje.
Parametar ROUNDS je opisan u Tester.

Definition at line 59 of file FuncTester.hpp.

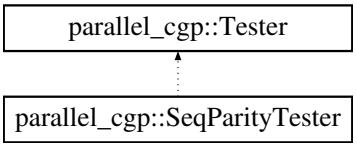
The documentation for this class was generated from the following file:

- funcProblem/FuncTester.hpp

## 5.21 parallel_cgp::SeqParityTester Class Reference

`#include <BoolTester.hpp>`

Inheritance diagram for parallel_cgp::SeqParityTester:



**Public Member Functions**

- SeqParityTester ()

### 5.21.1 Detailed Description

Klasa koja opisuje sekvencijski tester Parity problema.

Definition at line 117 of file BoolTester.hpp.

### 5.21.2 Constructor & Destructor Documentation

#### 5.21.2.1 SeqParityTester()

```
parallel_cgp::SeqParityTester::SeqParityTester ()  [inline]
```

Konstruktor testera koji odmah i pokrece testiranje.
Parametar ROUNDS je opisan u Tester.

Definition at line 140 of file BoolTester.hpp.

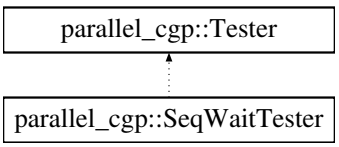
The documentation for this class was generated from the following file:

- boolProblem/BoolTester.hpp

## 5.22 parallel_cgp::SeqWaitTester Class Reference

```
#include <WaitTester.hpp>
```

Inheritance diagram for parallel_cgp::SeqWaitTester:




**Public Member Functions**

- SeqWaitTester ()

### 5.22.1 Detailed Description

Klasa koja opisuje sekvencijski tester Wait problema.

Definition at line 32 of file WaitTester.hpp.

### 5.22.2 Constructor & Destructor Documentation

#### 5.22.2.1 SeqWaitTester()

```
parallel_cgp::SeqWaitTester::SeqWaitTester ()  [inline]
```

Konstruktor testera koji odmah i pokrece testiranje.
Parametar ROUNDS je opisan u Tester.

Definition at line 55 of file WaitTester.hpp.

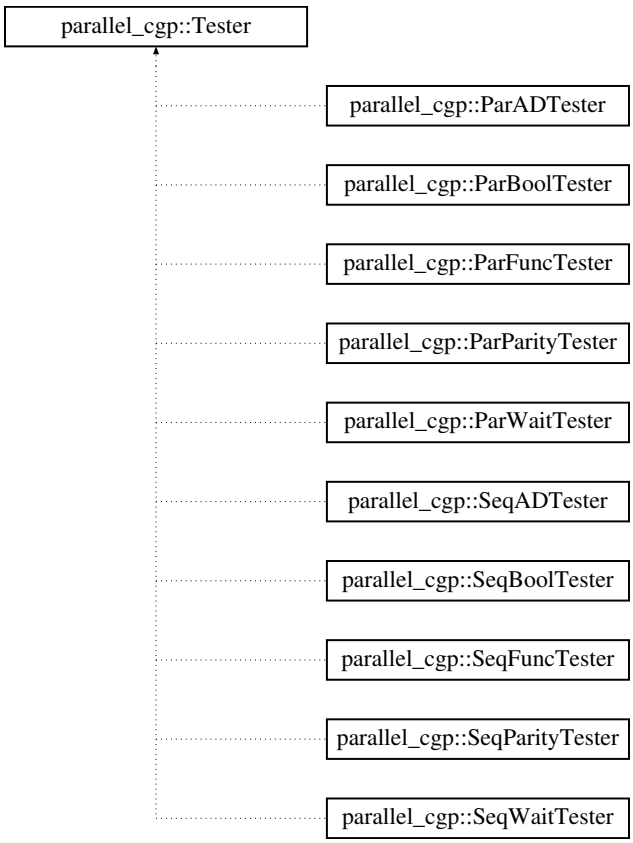
The documentation for this class was generated from the following file:

- waitProblem/WaitTester.hpp

## 5.23 parallel_cgp::Tester Class Reference

```
#include <Tester.hpp>
```

Inheritance diagram for parallel_cgp::Tester:




#### Public Member Functions

- Tester (std::string testerName)
- void saveResults (std::string testName, int gens, int rows, int cols, int levels, int pop)

**Static Public Attributes**

**Vrijednosti testera**

*Vrijednosti koje se koriste kod razlicitih testova.*

- static const int ROUNDS = 10
- static const int GENERATIONS = 1000
- static const int SMALL_ROWS = 4
- static const int MEDIUM_ROWS = 8
- static const int LARGE_ROWS = 10
- static const int SMALL_COLUMNS = 4
- static const int MEDIUM_COLUMNS = 8
- static const int LARGE_COLUMNS = 10
- static const int SMALL_LEVELS = 0
- static const int MEDIUM_LEVELS = 1
- static const int LARGE_LEVELS = 3
- static const int SMALL_POP_SIZE = 5
- static const int MEDIUM_POP_SIZE = 8
- static const int LARGE_POP_SIZE = 16
- static const int threadNums [6] = { 1, 2, 4, 8, 16, 32 }

## 5.23.1 Detailed Description

Klasa koja opisuje jedan Tester problema.

Definition at line 18 of file Tester.hpp.

## 5.23.2 Constructor & Destructor Documentation

### 5.23.2.1 Tester()

```
parallel_cgp::Tester::Tester (
            std::string testerName)  [inline]
```

Konstruktor koji incijalizira varijable i stvara csv datoteku za tu instancu.

**Parameters**

| | | |
|---|---|---|
| in | *testerName* | Naziv test suitea. |

Definition at line 64 of file Tester.hpp.

## 5.23.3 Member Function Documentation

### 5.23.3.1 saveResults()

```
void parallel_cgp::Tester::saveResults (
            std::string testName,
            int gens,
            int rows,
            int cols,
            int levels,
            int pop)  [inline]
```

Funkcija koja sprema sve rezultate u datoteku te ispisuje trenutno stanje testiranja.

**Parameters**

| | | |
|---|---|---|
| in | *testName* | Naziv trenutnog testa. |

Definition at line 75 of file Tester.hpp.

### 5.23.4 Member Data Documentation

#### 5.23.4.1 GENERATIONS

```
const int parallel_cgp::Tester::GENERATIONS = 1000  [static]
```

Broj generacija po testu.

Definition at line 31 of file Tester.hpp.

#### 5.23.4.2 LARGE_COLUMNS

```
const int parallel_cgp::Tester::LARGE_COLUMNS = 10  [static]
```

Broj CGP stupaca za veliki test.

Definition at line 43 of file Tester.hpp.

#### 5.23.4.3 LARGE_LEVELS

```
const int parallel_cgp::Tester::LARGE_LEVELS = 3  [static]
```

Broj CGP razina unatrag za veliki test (CGPIndividual::levelsBack).

Definition at line 49 of file Tester.hpp.

#### 5.23.4.4 LARGE_POP_SIZE

```
const int parallel_cgp::Tester::LARGE_POP_SIZE = 16  [static]
```

Velicina populacije za veliki test.

Definition at line 55 of file Tester.hpp.

#### 5.23.4.5 LARGE_ROWS

```
const int parallel_cgp::Tester::LARGE_ROWS = 10  [static]
```

Broj CGP redova za veliki test.

Definition at line 37 of file Tester.hpp.

#### 5.23.4.6 MEDIUM_COLUMNS

```
const int parallel_cgp::Tester::MEDIUM_COLUMNS = 8  [static]
```

Broj CGP stupaca za srednji test.

Definition at line 41 of file Tester.hpp.

#### 5.23.4.7 MEDIUM_LEVELS

```
const int parallel_cgp::Tester::MEDIUM_LEVELS = 1  [static]
```

Broj CGP razina unatrag za srednji test (CGPIndividual::levelsBack).

Definition at line 47 of file Tester.hpp.

#### 5.23.4.8 MEDIUM_POP_SIZE

```
const int parallel_cgp::Tester::MEDIUM_POP_SIZE = 8  [static]
```

Velicina populacije za srednji test.

Definition at line 53 of file Tester.hpp.

#### 5.23.4.9 MEDIUM_ROWS

```
const int parallel_cgp::Tester::MEDIUM_ROWS = 8  [static]
```

Broj CGP redova za srednji test.

Definition at line 35 of file Tester.hpp.

#### 5.23.4.10 ROUNDS

```
const int parallel_cgp::Tester::ROUNDS = 10  [static]
```

Koliko se puta vrti jedan test.

Definition at line 29 of file Tester.hpp.

#### 5.23.4.11 SMALL_COLUMNS

```
const int parallel_cgp::Tester::SMALL_COLUMNS = 4  [static]
```

Broj CGP stupaca za mali test.

Definition at line 39 of file Tester.hpp.

#### 5.23.4.12 SMALL_LEVELS

```
const int parallel_cgp::Tester::SMALL_LEVELS = 0  [static]
```

Broj CGP razina unatrag za mali test (CGPIndividual::levelsBack).

Definition at line 45 of file Tester.hpp.

#### 5.23.4.13 SMALL_POP_SIZE

```
const int parallel_cgp::Tester::SMALL_POP_SIZE = 5  [static]
```

Velicina populacije za mali test.

Definition at line 51 of file Tester.hpp.

#### 5.23.4.14 SMALL_ROWS

```
const int parallel_cgp::Tester::SMALL_ROWS = 4  [static]
```

Broj CGP redova za mali test.

Definition at line 33 of file Tester.hpp.

#### 5.23.4.15 threadNums

```
const int parallel_cgp::Tester::threadNums[6] = { 1, 2, 4, 8, 16, 32 }  [inline], [static]
```

Koje ce se sve kolicine dretvi koristiti u testovima.

Definition at line 57 of file Tester.hpp.

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

- Tester.hpp

## 5.24 parallel_cgp::Timer Class Reference

### Public Member Functions

- Timer (std::string funcName)
- void endTimer ()

### Static Public Member Functions

- static void printTimes ()
- static void saveTimes (std::string filename, std::string testName, int gens, int rows, int cols, int levels, int pop)
- static void clearTimes ()

### 5.24.1 Detailed Description

Definition at line 25 of file Timer.hpp.

### 5.24.2 Constructor & Destructor Documentation

#### 5.24.2.1 Timer()

```
parallel_cgp::Timer::Timer (
            std::string funcName)  [inline]
```

Osnovni kontruktor koji zapocinje timer za dani naziv funkcije.

**Parameters**

| | | |
|---|---|---|
| in | *funcName* | Naziv funkcije cije se vrijeme mjeri. |

Definition at line 39 of file Timer.hpp.

### 5.24.3 Member Function Documentation

#### 5.24.3.1 clearTimes()

```
static void parallel_cgp::Timer::clearTimes ()  [inline], [static]
```

Funkcija koja prazni mapu.

Definition at line 83 of file Timer.hpp.

#### 5.24.3.2 endTimer()

```
void parallel_cgp::Timer::endTimer ()  [inline]
```

Funkcija koja zavrsava timer te ga pohranjuje u mapu.

Definition at line 44 of file Timer.hpp.

#### 5.24.3.3 printTimes()

```
static void parallel_cgp::Timer::printTimes ()  [inline], [static]
```

Funkcija koja ispisuje sva vremena na standardni izlaz.

Definition at line 54 of file Timer.hpp.

#### 5.24.3.4 saveTimes()

```
static void parallel_cgp::Timer::saveTimes (
            std::string filename,
            std::string testName,
            int gens,
            int rows,
            int cols,
            int levels,
            int pop)  [inline], [static]
```

Funkcija koja sprema sva vremena u csv datoteku.

**Parameters**

| | | |
|---|---|---|
| in | *filename* | Naziv datoteke u koju se spremaju vremena. |

Definition at line 64 of file Timer.hpp.

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

- Timer.hpp

## 5.25 parallel_cgp::WaitParam Struct Reference

`#include <WaitTester.hpp>`

### Public Member Functions

- WaitParam (int gens, int rows, int cols, int levels, int pop, int time)

### Public Attributes

- int gens
- int rows
- int cols
- int levels
- int pop
- int time

### 5.25.1 Detailed Description

Struktura koja se koristi za upravljanje test parametara.

Definition at line 12 of file WaitTester.hpp.

### 5.25.2 Constructor & Destructor Documentation

#### 5.25.2.1 WaitParam() [1/2]

```
parallel_cgp::WaitParam::WaitParam ()  [inline]
```

Definition at line 13 of file WaitTester.hpp.

#### 5.25.2.2 WaitParam() [2/2]

```
parallel_cgp::WaitParam::WaitParam (
            int gens,
            int rows,
            int cols,
            int levels,
            int pop,
            int time)  [inline]
```

Definition at line 14 of file WaitTester.hpp.

### 5.25.3 Member Data Documentation

#### 5.25.3.1 cols

```
int parallel_cgp::WaitParam::cols
```

Broj stupaca za CGP.

Definition at line 20 of file WaitTester.hpp.

#### 5.25.3.2 gens

```
int parallel_cgp::WaitParam::gens
```

Broj generacija po testu.

Definition at line 16 of file WaitTester.hpp.

#### 5.25.3.3 levels

```
int parallel_cgp::WaitParam::levels
```

Broj razina iza na koliko se nodeovi mogu spajati u CGP.

Definition at line 22 of file WaitTester.hpp.

#### 5.25.3.4 pop

```
int parallel_cgp::WaitParam::pop
```

Velicina populacije.

Definition at line 24 of file WaitTester.hpp.

#### 5.25.3.5 rows

```
int parallel_cgp::WaitParam::rows
```

Broj redova za CGP.

Definition at line 18 of file WaitTester.hpp.

#### 5.25.3.6 time

```
int parallel_cgp::WaitParam::time
```

Vrijeme koje se ceka u WaitProblem.

Definition at line 26 of file WaitTester.hpp.

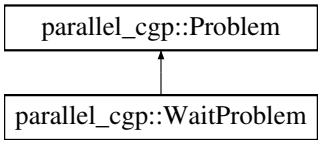
The documentation for this struct was generated from the following file:

- waitProblem/WaitTester.hpp

## 5.26 parallel_cgp::WaitProblem Class Reference

`#include <WaitProblem.hpp>`

Inheritance diagram for parallel_cgp::WaitProblem:




#### Public Member Functions

- WaitProblem ()
- WaitProblem (int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int POPULATION_SIZE, int WAIT_TIME)
- void problemRunner () override
- void printFunction () override

### Public Member Functions inherited from parallel_cgp::Problem

- virtual ~Problem ()=default
- virtual TYPE computeNode (int operand, TYPE value1, TYPE value2)

#### Additional Inherited Members

### Public Attributes inherited from parallel_cgp::Problem

- CGPIndividual * bestI
- bool printGens = false

- int NUM_OPERANDS = 9
- int BI_OPERANDS = 5
- int GENERATIONS = 5000
- int ROWS = 8
- int COLUMNS = 8
- int LEVELS_BACK = 3
- int INPUTS = 6
- int OUTPUTS = 1
- int POPULATION_SIZE = 20

### 5.26.1 Detailed Description

Klasa koja opisuje problem koji ceka odredeno vrijeme.

Definition at line 16 of file WaitProblem.hpp.

## 5.26.2 Constructor & Destructor Documentation

### 5.26.2.1 WaitProblem() [1/2]

```
parallel_cgp::WaitProblem::WaitProblem ()  [inline]
```

Osnovni kostruktor koji kreira osnovnu jedinku na bazi prije zadanih vrijednosti.

Definition at line 53 of file WaitProblem.hpp.

### 5.26.2.2 WaitProblem() [2/2]

```
parallel_cgp::WaitProblem::WaitProblem (
            int GENERATIONS,
            int ROWS,
            int COLUMNS,
            int LEVELS_BACK,
            int POPULATION_SIZE,
            int WAIT_TIME)  [inline]
```

Konstruktor koji prima sve promjenjive vrijednosti za wait problem.

Definition at line 57 of file WaitProblem.hpp.

## 5.26.3 Member Function Documentation

### 5.26.3.1 printFunction()

```
void WaitProblem::printFunction ()  [override], [virtual]
```

Metoda za ispis na kraju dobivene funkcije.

Implements parallel_cgp::Problem.

Definition at line 10 of file WaitProblem.cpp.

### 5.26.3.2 problemRunner()

```
void WaitProblem::problemRunner ()  [override], [virtual]
```

Metoda za pokretanje problema.

Implements parallel_cgp::Problem.

Definition at line 46 of file WaitProblem.cpp.

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

- waitProblem/WaitProblem.hpp
- waitProblem/WaitProblem.cpp

# Chapter 6

# File Documentation

## 6.1 ADProblem.cpp

```
00001 #include "ADProblem.hpp"
00002
00003 using namespace std;
00004 using namespace parallel_cgp;
00005
00006 TYPE ADProblem::computeNode(int operand, TYPE value1, TYPE value2) {
00007     switch (operand) {
00008     case 1:
00009         return value1 + value2;
00010     case 2:
00011         return value1 - value2;
00012     case 3:
00013         return value1 * value2;
00014     case 4:
00015         return -value1;
00016     default:
00017         return 0;
00018     }
00019 }
00020
00021 double ADProblem::fitness(TYPE cash, TYPE maxCash, double avgCash) {
00022     double score = avgCash;
00023
00024     if (maxCash >= STARTING_CASH * 2)
00025         score += 50;
00026     if (cash <= 0)
00027         score -= 100;
00028     if (maxCash == MAX_CASH)
00029         score += 150;
00030
00031     return score;
00032 }
00033
00034 void ADProblem::printFunction() {
00035     if (isSimulated)
00036         cout « "Funkcija: " « evalFunction(bestI->outputGene[0].connection) « endl;
00037     else
00038         cout « "Problem nije simuliran." « endl;
00039 }
00040
00041 string ADProblem::evalFunction(int CGPNodeNum) {
00042     ostringstream oss;
00043
00044     if (CGPNodeNum < INPUTS) {
00045         switch (CGPNodeNum) {
00046         case 0:
00047             oss « "card1";
00048             return oss.str();
00049         case 1:
00050             oss « "card2";
00051             return oss.str();
00052         }
00053     }
00054
00055     switch (bestI->genes[CGPNodeNum].operand) {
00056     case 1:
00057         oss « "(" « evalFunction(bestI->genes[CGPNodeNum].connection1) « " + " «
      evalFunction(bestI->genes[CGPNodeNum].connection2) « ")";
```

```
00058         return oss.str();
00059     case 2:
00060         oss « "(" « evalFunction(bestI->genes[CGPNodeNum].connection1) « " - " «
      evalFunction(bestI->genes[CGPNodeNum].connection2) « ")";
00061         return oss.str();
00062     case 3:
00063         oss « "(" « evalFunction(bestI->genes[CGPNodeNum].connection1) « " * " «
      evalFunction(bestI->genes[CGPNodeNum].connection2) « ")";
00064         return oss.str();
00065     case 4:
00066         oss « "-" « evalFunction(bestI->genes[CGPNodeNum].connection1);
00067         return oss.str();
00068     }
00069
00070     return "";
00071 }
00072
00073 void ADProblem::problemSimulator(CGPIndividual& individual, double& fit) {
00074     Timer probSimTime("problemSimulatorTimer");
00075
00076     function<double(int op, double v1, double v2)> compNode =
00077         [&](int op, double v1, double v2) { return computeNode(op, static_cast<TYPE>(v1),
      static_cast<TYPE>(v2)); };
00078
00079     int card, win;
00080     int cash = STARTING_CASH, maxCash = STARTING_CASH;
00081     double avgCash = 0;
00082
00083     for (int i = 0; i < CARD_SETS; i++) {
00084         card = static_cast<int>(sets[i].back());
00085
00086         if (card > sets[i].at(0) && card < sets[i].at(1))
00087             win = 1;
00088         else if (card == sets[i].at(0) || card == sets[i].at(1))
00089             win = -1;
00090         else
00091             win = 0;
00092
00093         individual.evaluateValue(sets[i], compNode);
00094
00095         if (individual.outputGene[0].value > 1) {
00096             if (win == 1)
00097                 cash += 10;
00098             else if (win == 0)
00099                 cash -= 10;
00100             else if (win == -1)
00101                 cash -= 20;
00102         }
00103
00104         if (cash > maxCash)
00105             maxCash = cash;
00106
00107         avgCash += cash;
00108     }
00109
00110     avgCash /= static_cast<double>(CARD_SETS);
00111     fit = fitness(cash, maxCash, avgCash);
00112
00113     probSimTime.endTimer();
00114 }
00115
00116 void ADProblem::problemRunner() {
00117     Timer probRunTime("problemRunnerTimer");
00118
00119     CGP cgp(ROWS, COLUMNS, LEVELS_BACK, INPUTS, OUTPUTS, NUM_OPERANDS, BI_OPERANDS, POPULATION_SIZE);
00120
00121     vector<CGPIndividual> population(POPULATION_SIZE);
00122     int bestInd = 0, generacija = 0;
00123
00124     cgp.generatePopulation(population);
00125
00126     random_device rd;
00127     mt19937 gen(rd());
00128
00129     uniform_int_distribution<> cardDis(1, 13);
00130
00131     for (int j = 0; j < CARD_SETS; j++) {
00132         vector<double> set;
00133         for (int i = 0; i < 3; i++)
00134             set.push_back(static_cast<double>(cardDis(gen)));
00135
00136         double card = set.back();
00137         set.pop_back();
00138         sort(set.begin(), set.end());
00139         set.push_back(card);
00140
00141         sets.push_back(set);
```

```
00142     }
00143
00144     for (generacija = 0; generacija < GENERATIONS; generacija++) {
00145         double bestFit = DBL_MIN;
00146         bestInd = 0;
00147         vector<int> bestInds;
00148         random_device rd;
00149         mt19937 gen(rd());
00150
00151         for (int clan = 0; clan < POPULATION_SIZE; clan++) {
00152
00153             double fit = 0;
00154             problemSimulator(population[clan], fit);
00155
00156             if (fit > bestFit) {
00157                 bestFit = fit;
00158                 bestInds.clear();
00159                 bestInds.push_back(clan);
00160             }
00161             else if (fit == bestFit)
00162                 bestInds.push_back(clan);
00163         }
00164
00165         if (bestInds.size() > 1)
00166             bestInds.erase(bestInds.begin());
00167         if (bestInds.size() == 0)
00168             bestInds.push_back(0);
00169
00170         uniform_int_distribution<> bestDis(0, static_cast<int>(bestInds.size() - 1));
00171
00172         bestInd = bestInds[bestDis(gen)];
00173
00174         if(printGens)
00175             cout « "Gen: " « generacija « "; Fitness: " « bestFit « "; Indeks: " « bestInd « endl;
00176
00177         if (bestFit >= THRESHOLD)
00178             break;
00179         if (generacija != GENERATIONS - 1)
00180             cgp.goldMutate(population[bestInd], population);
00181     }
00182
00183     bestI = &population[bestInd];
00184
00185     isSimulated = true;
00186
00187     printFunction();
00188
00189     probRunTime.endTimer();
00190
00191     playGame();
00192 }
00193
00194 void ADProblem::playGame() {
00195     function<double(int op, double v1, double v2)> compNode =
00196         [&](int op, double v1, double v2) { return computeNode(op, static_cast<TYPE>(v1),
      static_cast<TYPE>(v2)); };
00197
00198     random_device rd;
00199     mt19937 gen(rd());
00200
00201     uniform_int_distribution<> cardDis(1, 13);
00202
00203     int steps = 0;
00204     int cash = STARTING_CASH, maxCash = STARTING_CASH;
00205
00206     while (cash && steps < 100 && cash < MAX_CASH) {
00207         vector<double> input;
00208         int card, win;
00209         for (int i = 0; i < 3; i++)
00210             input.push_back(static_cast<TYPE>(cardDis(gen)));
00211
00212         card = card = static_cast<int>(input.back());
00213         input.pop_back();
00214
00215         sort(input.begin(), input.end());
00216
00217         if (card > input.at(0) && card < input.at(1))
00218             win = 1;
00219         else if (card == input.at(0) || card == input.at(1))
00220             win = -1;
00221         else
00222             win = 0;
00223
00224         bestI->evaluateValue(input, compNode);
00225
00226         cout « "Cash: " « cash « "; Cards: " « input[0] « ", " « input[1] « "; Bet: " «
      ((bestI->outputGene[0].value > 1) ? "YES" : "NO")
```

```
00227             « "; Third card: " « card « ((win == 1) ? " | WIN!" : " | LOST!") « endl;
00228
00229         if (bestI->outputGene[0].value > 1) {
00230             if (win == 1)
00231                 cash += 10;
00232             else if (win == 0)
00233                 cash -= 10;
00234             else if (win == -1)
00235                 cash -= 20;
00236         }
00237
00238         if (cash > maxCash)
00239             maxCash = cash;
00240
00241         steps++;
00242     }
00243 }
```

## 6.2 ADProblem.hpp

```
00001 #ifndef ADPROBLEM_HPP
00002 #define ADPROBLEM_HPP
00003
00004 #include "../Problem.hpp"
00005 #include "../cgp/CGP.hpp"
00006
00007 #undef TYPE
00008 #define TYPE int
00009
00010 namespace parallel_cgp {
00014     class ADProblem : public Problem {
00015     private:
00024         const static int NUM_OPERANDS = 4;
00025         const static int BI_OPERANDS = 4;
00026         const static int INPUTS = 2;
00027         const static int OUTPUTS = 1;
00028         const static int MAX_CASH = 1000;
00029         const static int STARTING_CASH = 100;
00030         const static int CARD_SETS = 500;
00031         const static int THRESHOLD = STARTING_CASH * 3;
00032
00037         int GENERATIONS = 200;
00038         int ROWS = 8;
00039         int COLUMNS = 8;
00040         int LEVELS_BACK = 3;
00041         int POPULATION_SIZE = 15;
00042
00046         std::vector<std::vector<double» sets;
00047
00051         bool isSimulated = false;
00052
00053         TYPE computeNode(int operand, TYPE value1, TYPE value2);
00054         double fitness(TYPE cash, TYPE maxCash, double avgCash);
00055         void problemSimulator(parallel_cgp::CGPIndividual& individual, double& fit) override;
00056         std::string evalFunction(int CGPNodeNum) override;
00057     public:
00061         ADProblem() {};
00065         ADProblem(int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int POPULATION_SIZE)
00066             : GENERATIONS(GENERATIONS), ROWS(ROWS), COLUMNS(COLUMNS), LEVELS_BACK(LEVELS_BACK),
    POPULATION_SIZE(POPULATION_SIZE) {};
00067
00071         void problemRunner() override;
00075         void printFunction() override;
00079         void playGame();
00080     };
00081 }
00082
00083 #endif
```

## 6.3 ADTester.hpp

```
00001 #ifndef ADTESTER_HPP
00002 #define ADTESTER_HPP
00003
00004 #include "../Tester.hpp"
00005 #include "../Timer.hpp"
00006 #include "ADProblem.hpp"
00007
```

```
00008 namespace parallel_cgp {
00012     struct ADParam {
00013         ADParam() {}
00014         ADParam(int gens, int rows, int cols, int levels, int pop) : gens(gens), rows(rows),
      cols(cols), levels(levels), pop(pop) {}
00016         int gens;
00018         int rows;
00020         int cols;
00022         int levels;
00024         int pop;
00025     };
00026
00030     class SeqADTester : private Tester
00031     {
00032     private:
00033         std::string funcs[3] = { "smallSeqADTest", "mediumSeqADTest", "largeSeqADTest" };
00034         ADParam params[3] = { ADParam(GENERATIONS, SMALL_ROWS, SMALL_COLUMNS, SMALL_LEVELS,
      SMALL_POP_SIZE),
00035             ADParam(GENERATIONS, MEDIUM_ROWS, MEDIUM_COLUMNS, MEDIUM_LEVELS, MEDIUM_POP_SIZE),
00036             ADParam(GENERATIONS, LARGE_ROWS, LARGE_COLUMNS, LARGE_LEVELS, LARGE_POP_SIZE) };
00037
00038         void test(std::string testName, int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int
      POPULATION_SIZE) {
00039             Timer testTimer("adTestTimer");
00040
00041             ADProblem problem(GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE);
00042             problem.problemRunner();
00043
00044             testTimer.endTimer();
00045
00046             saveResults(testName, GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE);
00047         }
00048     public:
00053         SeqADTester() : Tester("SeqADTest") {
00054             for (int f = 0; f < (sizeof(funcs) / sizeof(*funcs)); f++) {
00055                 for (int i = 0; i < ROUNDS; i++) {
00056                     test(funcs[f], params[f].gens, params[f].rows, params[f].cols, params[f].levels,
      params[f].pop);
00057                 }
00058             }
00059         }
00060     };
00061
00065     class ParADTester : private Tester
00066     {
00067     private:
00068         std::string funcs[3] = { "smallParADTest", "mediumParADTest", "largeParADTest" };
00069         ADParam params[3] = { ADParam(GENERATIONS, SMALL_ROWS, SMALL_COLUMNS, SMALL_LEVELS,
      SMALL_POP_SIZE),
00070             ADParam(GENERATIONS, MEDIUM_ROWS, MEDIUM_COLUMNS, MEDIUM_LEVELS, MEDIUM_POP_SIZE),
00071             ADParam(GENERATIONS, LARGE_ROWS, LARGE_COLUMNS, LARGE_LEVELS, LARGE_POP_SIZE) };
00072
00073         void test(std::string testName, int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int
      POPULATION_SIZE, int THREAD_NUM) {
00074             Timer testTimer("adTestTimer");
00075
00076             omp_set_num_threads(THREAD_NUM);
00077
00078             ADProblem problem(GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE);
00079             problem.problemRunner();
00080
00081             testTimer.endTimer();
00082
00083             saveResults(testName, GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE);
00084         }
00085     public:
00090         ParADTester() : Tester("ParADTest") {
00091             for (int f = 0; f < (sizeof(funcs) / sizeof(*funcs)); f++) {
00092                 for (int t = 0; t < (sizeof(threadNums) / sizeof(*threadNums)); t++) {
00093                     for (int i = 0; i < ROUNDS; i++) {
00094                         test(funcs[f] + std::to_string(threadNums[t]) + "T", params[f].gens,
      params[f].rows, params[f].cols, params[f].levels, params[f].pop, threadNums[t]);
00095                     }
00096                 }
00097             }
00098         }
00099     };
00100 }
00101
00102 #endif
```

## 6.4 BoolProblem.cpp

```
00001 #include "BoolProblem.hpp"
```

```
00002
00003 using namespace std;
00004 using namespace parallel_cgp;
00005
00006 TYPE BoolProblem::computeNode(int operand, TYPE value1, TYPE value2) {
00007     switch (operand) {
00008     case 1:
00009         return value1 | value2;
00010     case 2:
00011         return value1 & value2;
00012     case 3:
00013         return value1 ^ value2;
00014     case 4:
00015         return ~value1;
00016     default:
00017         return 0;
00018     }
00019 }
00020
00021 TYPE BoolProblem::fitness(bitset<INPUTS> in, TYPE res) {
00022     if (useFunc)
00023         return boolFunc(in) == res;
00024
00025     return parityFunc(in) == res;
00026 }
00027
00028 void BoolProblem::printFunction() {
00029     if (isSimulated)
00030         cout « "Funkcija: " « evalFunction(bestI->outputGene[0].connection) « endl;
00031     else
00032         cout « "Problem nije simuliran." « endl;
00033 }
00034
00035 string BoolProblem::evalFunction(int CGPNodeNum) {
00036     ostringstream oss;
00037
00038     if (CGPNodeNum < INPUTS) {
00039         oss « "bit[" « CGPNodeNum « "]";
00040         return oss.str();
00041     }
00042
00043     switch (bestI->genes[CGPNodeNum].operand) {
00044     case 1:
00045         oss « "(" « evalFunction(bestI->genes[CGPNodeNum].connection1) « " | " «
      evalFunction(bestI->genes[CGPNodeNum].connection2) « ")";
00046         return oss.str();
00047     case 2:
00048         oss « "(" « evalFunction(bestI->genes[CGPNodeNum].connection1) « " & " «
      evalFunction(bestI->genes[CGPNodeNum].connection2) « ")";
00049         return oss.str();
00050     case 3:
00051         oss « "(" « evalFunction(bestI->genes[CGPNodeNum].connection1) « " ^ " «
      evalFunction(bestI->genes[CGPNodeNum].connection2) « ")";
00052         return oss.str();
00053     case 4:
00054         oss « "~" « evalFunction(bestI->genes[CGPNodeNum].connection1);
00055         return oss.str();
00056     }
00057
00058     return "";
00059 }
00060
00061 void BoolProblem::problemSimulator(CGPIndividual& individual, TYPE &fit) {
00062     Timer probSimTime("problemSimulatorTimer");
00063
00064     function<double(int op, double v1, double v2)> compNode =
00065         [&](int op, double v1, double v2) { return computeNode(op, static_cast<TYPE>(v1),
      static_cast<TYPE>(v2)); };
00066
00067     for (int perm = 0; perm < static_cast<int>(pow(2, INPUTS)); ++perm) {
00068         bitset<INPUTS> bits(perm);
00069         vector<double> input;
00070
00071         for (int i = 0; i < bits.size(); ++i)
00072             input.push_back(static_cast<double>(bits[i]));
00073
00074         individual.evaluateValue(input, compNode);
00075         fit += fitness(bits, static_cast<int>(individual.outputGene[0].value));
00076     }
00077
00078     probSimTime.endTimer();
00079 }
00080
00081 void BoolProblem::problemRunner() {
00082     Timer probRunTime("problemRunnerTimer");
00083
00084     CGP cgp(ROWS, COLUMNS, LEVELS_BACK, INPUTS, OUTPUTS, NUM_OPERANDS, BI_OPERANDS, POPULATION_SIZE);
```

```
00085
00086     vector<CGPIndividual> population(POPULATION_SIZE);
00087     int bestInd = 0, generacija = 0;
00088
00089     cgp.generatePopulation(population);
00090
00091     for (generacija = 0; generacija < GENERATIONS; generacija++) {
00092         TYPE bestFit = INT_MIN;
00093         bestInd = 0;
00094         vector<int> bestInds;
00095         random_device rd;
00096         mt19937 gen(rd());
00097
00098         for (int clan = 0; clan < POPULATION_SIZE; clan++) {
00099
00100             TYPE fit = 0;
00101             problemSimulator(population[clan], fit);
00102
00103             if (fit > bestFit) {
00104                 bestFit = fit;
00105                 bestInds.clear();
00106                 bestInds.push_back(clan);
00107             }
00108             else if (fit == bestFit)
00109                 bestInds.push_back(clan);
00110         }
00111
00112         if (bestInds.size() > 1)
00113             bestInds.erase(bestInds.begin());
00114         if (bestInds.size() == 0)
00115             bestInds.push_back(0);
00116
00117         uniform_int_distribution<> bestDis(0, static_cast<int>(bestInds.size() - 1));
00118
00119         bestInd = bestInds[bestDis(gen)];
00120
00121         if (printGens)
00122             cout « "Gen: " « generacija « "; Fitness: " « bestFit « "; Indeks: " « bestInd « endl;
00123
00124         if (bestFit == pow(2, INPUTS))
00125             break;
00126         if (generacija != GENERATIONS - 1)
00127             cgp.goldMutate(population[bestInd], population);
00128     }
00129
00130     bestI = &population[bestInd];
00131
00132     isSimulated = true;
00133
00134     printFunction();
00135
00136     probRunTime.endTimer();
00137 }
```

## 6.5 BoolProblem.hpp

```
00001 #ifndef BOOLPROBLEM_HPP
00002 #define BOOLPROBLEM_HPP
00003
00004 #include "../Problem.hpp"
00005 #include "../cgp/CGP.hpp"
00006 #include <bitset>
00007
00008 #undef TYPE
00009 #define TYPE int
00010
00011 namespace parallel_cgp {
00015     class BoolProblem : public Problem {
00016     protected:
00022         const static int NUM_OPERANDS = 4;
00023         const static int BI_OPERANDS = 4;
00024         const static int INPUTS = 7;
00025         const static int OUTPUTS = 1;
00026
00031         int GENERATIONS = 5000;
00032         int ROWS = 10;
00033         int COLUMNS = 10;
00034         int LEVELS_BACK = 3;
00035         int POPULATION_SIZE = 15;
00036
00040         bool isSimulated = false;
00044         bool useFunc = true;
00045
```

```
00049         std::function<int(std::bitset<INPUTS> in)> boolFunc =
00050             [](std::bitset<INPUTS> in) { return (in[0] | ~in[1]) & ((in[0] ^ in[4]) | (in[3] &
~in[2])); };
00054         std::function<int(std::bitset<INPUTS> in)> parityFunc =
00055             [](std::bitset<INPUTS> in) { return (in.count() % 2 == 0) ? 0 : 1; };
00056
00057         TYPE computeNode(int operand, TYPE value1, TYPE value2);
00058         TYPE fitness(std::bitset<INPUTS> input, TYPE res);
00059         void problemSimulator(CGPIndividual &individual, TYPE &fit);
00060         std::string evalFunction(int CGPNodeNum) override;
00061     public:
00065         BoolProblem() {};
00070         BoolProblem(int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int POPULATION_SIZE)
00071             : GENERATIONS(GENERATIONS), ROWS(ROWS), COLUMNS(COLUMNS), LEVELS_BACK(LEVELS_BACK),
POPULATION_SIZE(POPULATION_SIZE) {
00072         };
00076         BoolProblem(int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int POPULATION_SIZE,
std::function<int(std::bitset<INPUTS> in)> boolFunc)
00077             : GENERATIONS(GENERATIONS), ROWS(ROWS), COLUMNS(COLUMNS), LEVELS_BACK(LEVELS_BACK),
POPULATION_SIZE(POPULATION_SIZE), boolFunc(boolFunc) {};
00078
00082         void problemRunner() override;
00086         void printFunction() override;
00087     };
00088
00092     class ParityProblem : public BoolProblem {
00093     public:
00097         ParityProblem() : BoolProblem() { useFunc = false; };
00101         ParityProblem(int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int POPULATION_SIZE)
00102             : BoolProblem(GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE) { useFunc = false;
};
00103     };
00104 }
00105
00106 #endif
```

## 6.6 BoolTester.hpp

```
00001 #ifndef BOOLTESTER_HPP
00002 #define BOOLTESTER_HPP
00003
00004 #include "../Tester.hpp"
00005 #include "../Timer.hpp"
00006 #include "BoolProblem.hpp"
00007
00008 namespace parallel_cgp {
00012     struct BoolParam {
00013         BoolParam() {}
00014         BoolParam(int gens, int rows, int cols, int levels, int pop) : gens(gens), rows(rows),
cols(cols), levels(levels), pop(pop) {}
00015         int gens;
00017         int rows;
00019         int cols;
00021         int levels;
00023         int pop;
00024     };
00025
00029     class SeqBoolTester : private Tester, private BoolProblem
00030     {
00031     private:
00032         std::string boolFuncs[6] = { "smallSimpleSeqBoolTest", "mediumSimpleSeqBoolTest",
"largeSimpleSeqBoolTest", "smallComplexSeqBoolTest", "mediumComplexSeqBoolTest",
"largeComplexSeqBoolTest" };
00033         BoolParam params[6] = { BoolParam(Tester::GENERATIONS, SMALL_ROWS, SMALL_COLUMNS,
SMALL_LEVELS, SMALL_POP_SIZE),
00034             BoolParam(Tester::GENERATIONS, MEDIUM_ROWS, MEDIUM_COLUMNS, MEDIUM_LEVELS,
MEDIUM_POP_SIZE),
00035             BoolParam(Tester::GENERATIONS, LARGE_ROWS, LARGE_COLUMNS, LARGE_LEVELS, LARGE_POP_SIZE),
00036             BoolParam(Tester::GENERATIONS, SMALL_ROWS, SMALL_COLUMNS, SMALL_LEVELS, SMALL_POP_SIZE),
00037             BoolParam(Tester::GENERATIONS, MEDIUM_ROWS, MEDIUM_COLUMNS, MEDIUM_LEVELS,
MEDIUM_POP_SIZE),
00038             BoolParam(Tester::GENERATIONS, LARGE_ROWS, LARGE_COLUMNS, LARGE_LEVELS, LARGE_POP_SIZE) };
00039         std::function<int(std::bitset<INPUTS> in)> func[2] = { [](std::bitset<INPUTS> in) { return
(in[0] | ~in[1]) & ((in[0] ^ in[4]) | (in[3] & ~in[2])); }, [](std::bitset<INPUTS> in) { return
(((in[0] & ~in[1]) | (in[2] ^ in[3])) & ((in[4] | in[5]) & (~in[6] | (in[0] & in[1])))) | (((in[2] &
in[3]) | (in[4] ^ in[5])) & ((in[6] | ~in[0]) & (in[1] | in[2])))); } };
00040
00041         void test(std::string testName, int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int
POPULATION_SIZE, std::function<int(std::bitset<INPUTS> in)> boolFunc) {
00042             Timer testTimer("boolTestTimer");
00043
00044             BoolProblem problem(GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE, boolFunc);
00045             problem.problemRunner();
```

```
00046
00047             testTimer.endTimer();
00048
00049             saveResults(testName, GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE);
00050         }
00051     public:
00056         SeqBoolTester() : Tester("SeqBoolTest") {
00057             for (int f = 0; f < (sizeof(boolFuncs) / sizeof(*boolFuncs)); f++) {
00058                 for (int i = 0; i < ROUNDS; i++) {
00059                     if (f < 3)
00060                         test(boolFuncs[f], params[f].gens, params[f].rows, params[f].cols,
      params[f].levels, params[f].pop, func[0]);
00061                     else
00062                         test(boolFuncs[f], params[f].gens, params[f].rows, params[f].cols,
      params[f].levels, params[f].pop, func[1]);
00063                 }
00064             }
00065         }
00066     };
00067
00071     class ParBoolTester : private Tester, private BoolProblem
00072     {
00073     private:
00074         std::string boolFuncs[6] = { "smallSimpleParBoolTest", "mediumSimpleParBoolTest",
      "largeSimpleParBoolTest", "smallComplexParBoolTest", "mediumComplexParBoolTest",
      "largeComplexParBoolTest" };
00075         BoolParam params[6] = { BoolParam(Tester::GENERATIONS, SMALL_ROWS, SMALL_COLUMNS,
      SMALL_LEVELS, SMALL_POP_SIZE),
00076             BoolParam(Tester::GENERATIONS, MEDIUM_ROWS, MEDIUM_COLUMNS, MEDIUM_LEVELS,
      MEDIUM_POP_SIZE),
00077             BoolParam(Tester::GENERATIONS, LARGE_ROWS, LARGE_COLUMNS, LARGE_LEVELS, LARGE_POP_SIZE),
00078             BoolParam(Tester::GENERATIONS, SMALL_ROWS, SMALL_COLUMNS, SMALL_LEVELS, SMALL_POP_SIZE),
00079             BoolParam(Tester::GENERATIONS, MEDIUM_ROWS, MEDIUM_COLUMNS, MEDIUM_LEVELS,
      MEDIUM_POP_SIZE),
00080             BoolParam(Tester::GENERATIONS, LARGE_ROWS, LARGE_COLUMNS, LARGE_LEVELS, LARGE_POP_SIZE) };
00081         std::function<int(std::bitset<INPUTS> in)> func[2] = { [](std::bitset<INPUTS> in) { return
      (in[0] | ~in[1]) & ((in[0] ^ in[4]) | (in[3] & ~in[2])); }, [](std::bitset<INPUTS> in) { return
      (((in[0] & ~in[1]) | (in[2] ^ in[3])) & ((in[4] | in[5]) & (~in[6] | (in[0] & in[1])))) | (((in[2] &
      in[3]) | (in[4] ^ in[5])) & ((in[6] | ~in[0]) & (in[1] | in[2])))); } };
00082
00083         void test(std::string testName, int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int
      POPULATION_SIZE, std::function<int(std::bitset<INPUTS> in)> boolFunc, int THREAD_NUM) {
00084             Timer testTimer("boolTestTimer");
00085
00086             omp_set_num_threads(THREAD_NUM);
00087
00088             BoolProblem problem(GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE, boolFunc);
00089             problem.problemRunner();
00090
00091             testTimer.endTimer();
00092
00093             saveResults(testName, GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE);
00094         }
00095     public:
00100         ParBoolTester() : Tester("ParBoolTest") {
00101             for (int f = 0; f < (sizeof(boolFuncs) / sizeof(*boolFuncs)); f++) {
00102                 for (int t = 0; t < (sizeof(threadNums) / sizeof(*threadNums)); t++) {
00103                     for (int i = 0; i < ROUNDS; i++) {
00104                         if (f < 3)
00105                             test(boolFuncs[f] + std::to_string(threadNums[t]) + "T", params[f].gens,
      params[f].rows, params[f].cols, params[f].levels, params[f].pop, func[0], threadNums[t]);
00106                         else
00107                             test(boolFuncs[f] + std::to_string(threadNums[t]) + "T", params[f].gens,
      params[f].rows, params[f].cols, params[f].levels, params[f].pop, func[1], threadNums[t]);
00108                     }
00109                 }
00110             }
00111         }
00112     };
00113
00117     class SeqParityTester : private Tester
00118     {
00119     private:
00120         std::string parityFuncs[3] = { "smallSeqParityTest", "mediumSeqParityTest",
      "largeSeqParityTest" };
00121         BoolParam params[3] = { BoolParam(GENERATIONS, SMALL_ROWS, SMALL_COLUMNS, SMALL_LEVELS,
      SMALL_POP_SIZE),
00122             BoolParam(GENERATIONS, MEDIUM_ROWS, MEDIUM_COLUMNS, MEDIUM_LEVELS, MEDIUM_POP_SIZE),
00123             BoolParam(GENERATIONS, LARGE_ROWS, LARGE_COLUMNS, LARGE_LEVELS, LARGE_POP_SIZE) };
00124
00125         void test(std::string testName, int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int
      POPULATION_SIZE) {
00126             Timer testTimer("parityTestTimer");
00127
00128             ParityProblem problem(GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE);
00129             problem.problemRunner();
00130
```

```
00131             testTimer.endTimer();
00132
00133             saveResults(testName, GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE);
00134         }
00135     public:
00140         SeqParityTester() : Tester("SeqParityTest") {
00141             for (int f = 0; f < (sizeof(parityFuncs) / sizeof(*parityFuncs)); f++)
00142                 for (int i = 0; i < ROUNDS; i++)
00143                     test(parityFuncs[f], params[f].gens, params[f].rows, params[f].cols,
      params[f].levels, params[f].pop);
00144         }
00145     };
00146
00150     class ParParityTester : private Tester
00151     {
00152     private:
00153         std::string parityFuncs[3] = { "smallParParityTest", "mediumParParityTest",
      "largeParParityTest" };
00154         BoolParam params[3] = { BoolParam(GENERATIONS, SMALL_ROWS, SMALL_COLUMNS, SMALL_LEVELS,
      SMALL_POP_SIZE),
00155             BoolParam(GENERATIONS, MEDIUM_ROWS, MEDIUM_COLUMNS, MEDIUM_LEVELS, MEDIUM_POP_SIZE),
00156             BoolParam(GENERATIONS, LARGE_ROWS, LARGE_COLUMNS, LARGE_LEVELS, LARGE_POP_SIZE) };
00157
00158         void test(std::string testName, int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int
      POPULATION_SIZE, int THREAD_NUM) {
00159             Timer testTimer("parityTestTimer");
00160
00161             omp_set_num_threads(THREAD_NUM);
00162
00163             ParityProblem problem(GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE);
00164             problem.problemRunner();
00165
00166             testTimer.endTimer();
00167
00168             saveResults(testName, GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE);
00169         }
00170     public:
00175         ParParityTester() : Tester("ParParityTest") {
00176             for (int f = 0; f < (sizeof(parityFuncs) / sizeof(*parityFuncs)); f++)
00177                 for (int t = 0; t < (sizeof(threadNums) / sizeof(*threadNums)); t++)
00178                     for (int i = 0; i < ROUNDS; i++)
00179                         test(parityFuncs[f] + std::to_string(threadNums[t]) + "T", params[f].gens,
      params[f].rows, params[f].cols, params[f].levels, params[f].pop, threadNums[t]);
00180         }
00181     };
00182 }
00183
00184 #endif
```

## 6.7 CGP.cpp

```
00001 #include "CGP.hpp"
00002
00003 using namespace std;
00004 using namespace parallel_cgp;
00005
00006 void CGP::generatePopulation(vector<CGPIndividual> &population) {
00007     // vrijeme za izvodenje cijele funkcije
00008     Timer genTime("generatePopulationTimer");
00009
00010     random_device rd;
00011     mt19937 gen(rd());
00012
00013     for (int i = 0; i < populationSize; i++) {
00014         uniform_int_distribution<> operandDis(1, operands);
00015         uniform_int_distribution<> connectionDis(0, rows * columns + inputs - 1);
00016         uniform_int_distribution<> outputDis(0, rows * columns + inputs - 1);
00017
00018         vector<CGPNode> genes;
00019         vector<CGPOutput> outputGene;
00020
00021         for (int k = 0; k < inputs; k++) {
00022             CGPNode node;
00023             node.used = false;
00024             node.connection1 = -1;
00025             node.connection2 = -1;
00026             node.operand = -1;
00027             genes.push_back(node);
00028         }
00029
00030         for (int j = inputs; j < rows * columns + inputs; j++) {
00031             CGPNode node;
00032             node.used = false;
```

```
00033             node.operand = operandDis(gen);
00034             node.connection1 = connectionDis(gen);
00035             node.outValue = NAN;
00036 
00037             while (true) {
00038                 if (node.connection1 < inputs)
00039                     break;
00040                 if ((node.connection1 % columns) == (j % columns))
00041                     node.connection1 = connectionDis(gen);
00042                 else if (((node.connection1 - inputs) % columns) > (((j - inputs) % columns) +
      levelsBack))
00043                     node.connection1 = connectionDis(gen);
00044                 else if(genes.size() > node.connection1 && (genes[node.connection1].connection1 == j
      || genes[node.connection1].connection2 == j))
00045                     node.connection1 = connectionDis(gen);
00046                 else
00047                     break;
00048             }
00049 
00050             node.connection2 = (node.operand >= biOperands) ? -1 : connectionDis(gen);
00051 
00052             while (true) {
00053                 if (node.connection2 < inputs)
00054                     break;
00055                 if ((node.connection2 % columns) == (j % columns))
00056                     node.connection2 = connectionDis(gen);
00057                 else if (((node.connection2 - inputs) % columns) > (((j - inputs) % columns) +
      levelsBack))
00058                     node.connection2 = connectionDis(gen);
00059                 else if (genes.size() > node.connection2 && (genes[node.connection2].connection1 == j
      || genes[node.connection2].connection2 == j))
00060                     node.connection2 = connectionDis(gen);
00061                 else
00062                     break;
00063             }
00064             genes.push_back(node);
00065         }
00066 
00067         for (int k = 0; k < outputs; k++) {
00068             CGPOutput output;
00069 
00070             output.connection = outputDis(gen);
00071             outputGene.push_back(output);
00072         }
00073 
00074         CGPIndividual individual(genes, outputGene, rows, columns, levelsBack, inputs, outputs);
00075 
00076         population[i] = individual;
00077         population[i].resolveLoops();
00078     }
00079 
00080     genTime.endTimer();
00081 }
00082 
00083 void CGP::goldMutate(CGPIndividual parent, vector<CGPIndividual> &population) {
00084     Timer mutTime("mutatePopulationTimer");
00085 
00086     if (!parent.evalDone)
00087         parent.evaluateUsed();
00088     population[0] = parent;
00089 
00090     random_device rd;
00091     mt19937 gen(rd());
00092 
00093     for (int n = 1; n < populationSize; n++) {
00094         uniform_int_distribution<> nodDis(parent.inputs, static_cast<int>(parent.genes.size()));
00095         uniform_int_distribution<> geneDis(0, 2);
00096         uniform_int_distribution<> connectionDis(0, static_cast<int>(parent.genes.size()) - 1);
00097         uniform_int_distribution<> operandDis(1, operands);
00098         uniform_int_distribution<> outputDis(0, parent.outputs - 1);
00099 
00100         vector<CGPNode> genes = parent.genes;
00101         vector<CGPOutput> outputGene = parent.outputGene;
00102         bool isActive = false;
00103 
00104         while (!isActive) {
00105             int mut = geneDis(gen);
00106             int cell = nodDis(gen);
00107             if (cell == parent.genes.size()) {
00108                 outputGene[outputDis(gen)].connection = connectionDis(gen);
00109                 break;
00110             }
00111             if (mut == 0) {
00112                 genes[cell].operand = operandDis(gen);
00113 
00114                 if (genes[cell].operand >= biOperands && genes[cell].connection2 != -1)
00115                     genes[cell].connection2 = -1;
```

```
00116                 else if (genes[cell].operand < biOperands && genes[cell].connection2 == -1)
00117                     genes[cell].connection2 = connectionDis(gen);
00118             }
00119             else if (mut == 1)
00120                 genes[cell].connection1 = connectionDis(gen);
00121             else if (mut == 2 && genes[cell].operand >= biOperands)
00122                 continue;
00123             else if (mut == 2)
00124                 genes[cell].connection2 = connectionDis(gen);
00125 
00126             while (true) {
00127                 if (genes[cell].connection1 < parent.inputs)
00128                     break;
00129                 if ((genes[cell].connection1 % parent.columns) == (cell % parent.columns))
00130                     genes[cell].connection1 = connectionDis(gen);
00131                 else if (((genes[cell].connection1 - parent.inputs) % parent.columns) > (((cell -
      parent.inputs) % parent.columns) + parent.levelsBack))
00132                     genes[cell].connection1 = connectionDis(gen);
00133                 else
00134                     break;
00135             }
00136 
00137             while (true) {
00138                 if (genes[cell].connection2 < parent.inputs)
00139                     break;
00140                 if ((genes[cell].connection2 % parent.columns) == (cell % parent.columns))
00141                     genes[cell].connection2 = connectionDis(gen);
00142                 else if (((genes[cell].connection2 - parent.inputs) % parent.columns) > (((cell -
      parent.inputs) % parent.columns) + parent.levelsBack))
00143                     genes[cell].connection2 = connectionDis(gen);
00144                 else
00145                     break;
00146             }
00147 
00148             isActive = genes[cell].used;
00149         }
00150 
00151         for (size_t z = parent.inputs; z < genes.size(); z++)
00152             genes[z].used = false;
00153 
00154         CGPIndividual individual(genes, outputGene, parent.rows, parent.columns, parent.levelsBack,
      parent.inputs, parent.outputs);
00155 
00156         population[n] = individual;
00157         population[n].resolveLoops();
00158     }
00159 
00160     mutTime.endTimer();
00161 }
```

## 6.8 CGP.hpp

```
00001 #ifndef CGP_HPP
00002 #define CGP_HPP
00003 #define TYPE double
00004 
00005 #include "CGPIndividual.hpp"
00006 #include "../Timer.hpp"
00007 #include <iostream>
00008 #include <chrono>
00009 #include <thread>
00010 #include <cmath>
00011 #include <random>
00012 #include <fstream>
00013 #include <string>
00014 #include <sstream>
00015 #include <vector>
00016 #include <omp.h>
00017 
00018 namespace parallel_cgp {
00022     class CGP {
00023     private:
00024         int rows, columns, levelsBack, inputs, outputs, operands, biOperands, populationSize;
00025     public:
00037         CGP(int rows, int columns, int levelsBack, int inputs, int outputs, int operands, int
      biOperands, int populationSize)
00038             : rows(rows), columns(columns), levelsBack(levelsBack), inputs(inputs), outputs(outputs),
00039                 operands(operands), biOperands(biOperands), populationSize(populationSize) {};
00040 
00047         void generatePopulation(std::vector<CGPIndividual> &population);
00048 
00057         void goldMutate(CGPIndividual parent, std::vector<CGPIndividual> &population);
00058     };
```

```
00059 }
00060
00061 #endif
```

## 6.9 CGPIndividual.cpp

```
00001 #include "CGPIndividual.hpp"
00002
00003 using namespace std;
00004 using namespace parallel_cgp;
00005
00006 CGPIndividual::CGPIndividual() {
00007     vector<vector<int» branches;
00008     this->branches = branches;
00009     this->rows = 0;
00010     this->columns = 0;
00011     this->levelsBack = 0;
00012     this->inputs = 0;
00013     this->outputs = 0;
00014     this->evalDone = false;
00015 }
00016
00017 CGPIndividual::CGPIndividual(vector<CGPNode> genes, vector<CGPOutput> outputGene, int rows, int
      columns, int levelsBack, int inputs, int outputs) {
00018     vector<vector<int» branches;
00019     this->branches = branches;
00020     this->genes = genes;
00021     this->outputGene = outputGene;
00022     this->rows = rows;
00023     this->columns = columns;
00024     this->levelsBack = levelsBack;
00025     this->inputs = inputs;
00026     this->outputs = outputs;
00027     this->evalDone = false;
00028 }
00029
00030 CGPIndividual::CGPIndividual(vector<CGPNode> genes, vector<CGPOutput> outputGene, int rows, int
      columns, int levelsBack, int inputs, int outputs, bool evalDone) {
00031     vector<vector<int» branches;
00032     this->branches = branches;
00033     this->genes = genes;
00034     this->outputGene = outputGene;
00035     this->rows = rows;
00036     this->columns = columns;
00037     this->levelsBack = levelsBack;
00038     this->inputs = inputs;
00039     this->outputs = outputs;
00040     this->evalDone = evalDone;
00041 }
00042
00043 void CGPIndividual::printNodes() {
00044     for (size_t i = 0; i < rows * columns + inputs; i++)
00045         cout « i « " " « genes[i].operand « " " « genes[i].connection1 « " " « genes[i].connection2 «
      endl;
00046
00047     for (size_t j = 0; j < outputs; j++)
00048         cout « outputGene[j].connection « " ";
00049
00050     cout « endl « endl;
00051 }
00052
00053 void CGPIndividual::evaluateUsed() {
00054     for (int m = 0; m < outputs; m++)
00055         isUsed(outputGene[m].connection);
00056
00057     evalDone = true;
00058 }
00059
00060 void CGPIndividual::isUsed(int CGPNodeNum) {
00061     genes[CGPNodeNum].used = true;
00062
00063     if (genes[CGPNodeNum].connection1 >= 0)
00064         isUsed(genes[CGPNodeNum].connection1);
00065
00066     if (genes[CGPNodeNum].connection2 >= 0)
00067         isUsed(genes[CGPNodeNum].connection2);
00068 }
00069
00070 void CGPIndividual::evaluateValue(vector<TYPE> input, function<TYPE(int, TYPE, TYPE)> &computeNode) {
00071     clearInd();
00072
00073     for (int l = 0; l < inputs; l++)
00074         genes[l].outValue = input[l];
```

```
00075
00076     for (int m = 0; m < outputs; m++)
00077         outputGene[m].value = evalNode(outputGene[m].connection, computeNode);
00078 }
00079
00080 TYPE CGPIndividual::evalNode(int CGPNodeNum, function<TYPE(int, TYPE, TYPE)> &computeNode) {
00081
00082     if (isnan(genes[CGPNodeNum].outValue)) {
00083         TYPE value1 = evalNode(genes[CGPNodeNum].connection1, computeNode);
00084         TYPE value2 = genes[CGPNodeNum].connection2 < 0 ? 0 : evalNode(genes[CGPNodeNum].connection2,
      computeNode);
00085
00086         genes[CGPNodeNum].outValue = computeNode(genes[CGPNodeNum].operand, value1, value2);
00087     }
00088
00089     return genes[CGPNodeNum].outValue;
00090 }
00091
00092 void CGPIndividual::clearInd() {
00093     for (int i = inputs; i < genes.size(); i++)
00094         genes[i].outValue = NAN;
00095 }
00096
00097 bool CGPIndividual::findLoops(int CGPNodeNum) {
00098     branches.clear();
00099
00100     vector<int> CGPNodeSet;
00101
00102     return loopFinder(CGPNodeNum, CGPNodeSet);
00103 }
00104
00105 bool CGPIndividual::loopFinder(int CGPNodeNum, vector<int> CGPNodeSet) {
00106
00107     for (int i = 0; i < CGPNodeSet.size(); i++)
00108         if (CGPNodeSet[i] == CGPNodeNum) {
00109             CGPNodeSet.push_back(CGPNodeNum);
00110             branches.push_back(CGPNodeSet);
00111             return true;
00112         }
00113
00114     CGPNodeSet.push_back(CGPNodeNum);
00115
00116     if (CGPNodeNum < inputs) {
00117         return false;
00118     }
00119
00120     bool conn1 = loopFinder(genes[CGPNodeNum].connection1, CGPNodeSet);
00121     bool conn2 = genes[CGPNodeNum].connection2 == -1 ? false :
      loopFinder(genes[CGPNodeNum].connection2, CGPNodeSet);
00122
00123     return conn1 || conn2;
00124 }
00125
00126 void CGPIndividual::resolveLoops() {
00127
00128     Timer resLoopTime("resolveLoopsTimer");
00129
00130     random_device rd;
00131     mt19937 gen(rd());
00132
00133     for (int m = 0; m < outputs; m++) {
00134         while (findLoops(outputGene[m].connection)) {
00135             for (int i = 0; i < branches.size(); i++) {
00136                 uniform_int_distribution<> connectionDis(0, static_cast<int>(genes.size()) - 1);
00137                 int cell1 = branches[i][branches[i].size() - 2];
00138                 int cell2 = branches[i][branches[i].size() - 1];
00139
00140                 if (genes[cell1].connection1 == cell2) {
00141                     genes[cell1].connection1 = connectionDis(gen);
00142
00143                     while (true) {
00144                         if (genes[cell1].connection1 < inputs)
00145                             break;
00146                         if ((genes[cell1].connection1 % columns) == (cell1 % columns))
00147                             genes[cell1].connection1 = connectionDis(gen);
00148                         else if (((genes[cell1].connection1 - inputs) % columns) > (((cell1 - inputs)
      % columns) + levelsBack))
00149                             genes[cell1].connection1 = connectionDis(gen);
00150                         else
00151                             break;
00152                     }
00153                 }
00154                 else if (genes[cell1].connection2 == cell2) {
00155                     genes[cell1].connection2 = connectionDis(gen);
00156
00157                     while (true) {
00158                         if (genes[cell1].connection2 < inputs)
```

```
00159                             break;
00160                         if ((genes[cell1].connection2 % columns) == (cell1 % columns))
00161                             genes[cell1].connection2 = connectionDis(gen);
00162                         else if (((genes[cell1].connection2 - inputs) % columns) > (((cell1 - inputs)
    % columns) + levelsBack))
00163                             genes[cell1].connection2 = connectionDis(gen);
00164                         else
00165                             break;
00166                     }
00167                 }
00168             }
00169         }
00170     }
00171
00172     resLoopTime.endTimer();
00173 }
```

## 6.10 CGPIndividual.hpp

```
00001 #ifndef CGPINDIVIDUAL_HPP
00002 #define CGPINDIVIDUAL_HPP
00003 #define TYPE double
00004
00005 #include "CGPNode.hpp"
00006 #include "CGPOutput.hpp"
00007 #include "../Timer.hpp"
00008 #include <vector>
00009 #include <sstream>
00010 #include <functional>
00011 #include <omp.h>
00012 #include <iostream>
00013 #include <chrono>
00014 #include <thread>
00015 #include <random>
00016
00017 namespace parallel_cgp {
00021     class CGPIndividual {
00022     private:
00023         void isUsed(int nodeNum);
00024         bool loopFinder(int nodeNum, std::vector<int> nodeSet);
00025         TYPE evalNode(int nodeNum, std::function<TYPE(int, TYPE, TYPE)> &computeNode);
00026         void clearInd();
00027     public:
00031         std::vector<CGPNode> genes;
00035         std::vector<CGPOutput> outputGene;
00040         std::vector<std::vector<int» branches;
00042         int rows;
00044         int columns;
00046         int levelsBack;
00048         int inputs;
00050         int outputs;
00052         int evalDone;
00053
00057         CGPIndividual();
00069         CGPIndividual(std::vector<CGPNode> genes, std::vector<CGPOutput> outputGene, int rows, int
    columns, int levelsBack, int inputs, int outputs);
00075         CGPIndividual(std::vector<CGPNode> genes, std::vector<CGPOutput> outputGene, int rows, int
    columns, int levelsBack, int inputs, int outputs, bool evalDone);
00076
00080         void printNodes();
00086         void evaluateValue(std::vector<TYPE> input, std::function<TYPE(int, TYPE, TYPE)>
    &computeNode);
00090         void evaluateUsed();
00096         bool findLoops(int nodeNum);
00100         void resolveLoops();
00101     };
00102 }
00103
00104 #endif
```

## 6.11 CGPNode.hpp

```
00001 #ifndef CGPNODE_HPP
00002 #define CGPNODE_HPP
00003 #include <iostream>
00004 #include <fstream>
00005 #include <string>
00006 #define TYPE double
```

```
00007
00008 namespace parallel_cgp {
00012     struct CGPNode {
00016         int operand;
00020         int connection1;
00024         int connection2;
00028         bool used;
00032         TYPE outValue;
00033     };
00034 }
00035
00036 #endif
```

## 6.12 CGPOutput.hpp

```
00001 #ifndef CGPOUTPUT_HPP
00002 #define CGPOUTPUT_HPP
00003 #include <iostream>
00004 #include <fstream>
00005 #include <string>
00006 #define TYPE double
00007
00008 namespace parallel_cgp {
00012     struct CGPOutput {
00016         int connection;
00020         TYPE value;
00021     };
00022 }
00023
00024 #endif
```

## 6.13 FuncProblem.cpp

```
00001 #include "FuncProblem.hpp"
00002
00003 using namespace std;
00004 using namespace parallel_cgp;
00005
00006 TYPE FuncProblem::computeNode(int operand, TYPE value1, TYPE value2) {
00007     switch (operand) {
00008     case 1:
00009         return value1 + value2;
00010     case 2:
00011         return value1 - value2;
00012     case 3:
00013         return value1 * value2;
00014     case 4:
00015         return (value2 == 0) ? 0 : value1 / value2;
00016     case 5:
00017         return sin(value1);
00018     case 6:
00019         return cos(value1);
00020     case 7:
00021         return value1 > 0 ? sqrt(value1) : value1;
00022     case 8:
00023         return pow(value1, 2);
00024     case 9:
00025         return pow(2, value1);
00026     default:
00027         return 0;
00028     }
00029 }
00030
00031 TYPE FuncProblem::fitness(TYPE x, TYPE y, TYPE res) {
00032     return func(x, y) - res;
00033 }
00034
00035 void FuncProblem::printFunction() {
00036     if (isSimulated)
00037         cout « "Funkcija: " « evalFunction(bestI->outputGene[0].connection) « endl;
00038     else
00039         cout « "Problem nije simuliran." « endl;
00040 }
00041
00042 string FuncProblem::evalFunction(int CGPNodeNum) {
00043     ostringstream oss;
00044
00045     if (CGPNodeNum < INPUTS) {
```

```
00046         switch (CGPNodeNum) {
00047         case 0:
00048             oss « "x";
00049             return oss.str();
00050         case 1:
00051             oss « "y";
00052             return oss.str();
00053         }
00054     }
00055
00056     switch (bestI->genes[CGPNodeNum].operand) {
00057     case 1:
00058         oss « "(" « evalFunction(bestI->genes[CGPNodeNum].connection1) « " + " «
      evalFunction(bestI->genes[CGPNodeNum].connection2) « ")";
00059         return oss.str();
00060     case 2:
00061         oss « "(" « evalFunction(bestI->genes[CGPNodeNum].connection1) « " - " «
      evalFunction(bestI->genes[CGPNodeNum].connection2) « ")";
00062         return oss.str();
00063     case 3:
00064         oss « "(" « evalFunction(bestI->genes[CGPNodeNum].connection1) « " * " «
      evalFunction(bestI->genes[CGPNodeNum].connection2) « ")";
00065         return oss.str();
00066     case 4:
00067         oss « "(" « evalFunction(bestI->genes[CGPNodeNum].connection1) « " / " «
      evalFunction(bestI->genes[CGPNodeNum].connection2) « ")";
00068         return oss.str();
00069     case 5:
00070         oss « "sin(" « evalFunction(bestI->genes[CGPNodeNum].connection1) « ")";
00071         return oss.str();
00072     case 6:
00073         oss « "cos(" « evalFunction(bestI->genes[CGPNodeNum].connection1) « ")";
00074         return oss.str();
00075     case 7:
00076         oss « "sqrt(" « evalFunction(bestI->genes[CGPNodeNum].connection1) « ")";
00077         return oss.str();
00078     case 8:
00079         oss « evalFunction(bestI->genes[CGPNodeNum].connection1) « "^2";
00080         return oss.str();
00081     case 9:
00082         oss « "2^" « evalFunction(bestI->genes[CGPNodeNum].connection1);
00083         return oss.str();
00084     }
00085
00086     return "";
00087 }
00088
00089 void FuncProblem::problemSimulator(CGPIndividual& individual, TYPE& fit) {
00090     Timer probSimTime("problemSimulatorTimer");
00091
00092     function<TYPE(int op, TYPE v1, TYPE v2)> compNode =
00093         [&](int op, TYPE v1, TYPE v2) { return computeNode(op, v1, v2); };
00094
00095     TYPE N = 0;
00096
00097     for (TYPE x = -10; x < 10; x += 0.5) {
00098         for (TYPE y = -10; y < 10; y += 0.5) {
00099             vector<TYPE> input;
00100             input.push_back(x);
00101             input.push_back(y);
00102
00103             individual.evaluateValue(input, compNode);
00104             fit += pow(fitness(x, y, individual.outputGene[0].value), 2);
00105             N++;
00106         }
00107     }
00108
00109     fit /= N;
00110     fit = sqrt(fit);
00111
00112     probSimTime.endTimer();
00113 }
00114
00115 void FuncProblem::problemRunner() {
00116     Timer probRunTime("problemRunnerTimer");
00117
00118     CGP cgp(ROWS, COLUMNS, LEVELS_BACK, INPUTS, OUTPUTS, NUM_OPERANDS, BI_OPERANDS, POPULATION_SIZE);
00119
00120     vector<CGPIndividual> population(POPULATION_SIZE);
00121     int bestInd = 0, generacija = 0;
00122
00123     cgp.generatePopulation(population);
00124
00125     for (generacija = 0; generacija < GENERATIONS; generacija++) {
00126         TYPE bestFit = DBL_MAX;
00127         bestInd = 0;
00128         vector<int> bestInds;
```

```
00129         random_device rd;
00130         mt19937 gen(rd());
00131 
00132         for (int clan = 0; clan < POPULATION_SIZE; clan++) {
00133 
00134             TYPE fit = 0;
00135             problemSimulator(population[clan], fit);
00136 
00137             if (fit < bestFit) {
00138                 bestFit = fit;
00139                 bestInds.clear();
00140                 bestInds.push_back(clan);
00141             }
00142             else if (fit == bestFit)
00143                 bestInds.push_back(clan);
00144         }
00145 
00146         if (bestInds.size() > 1)
00147             bestInds.erase(bestInds.begin());
00148         if (bestInds.size() == 0)
00149             bestInds.push_back(0);
00150 
00151         uniform_int_distribution<> bestDis(0, static_cast<int>(bestInds.size()) - 1);
00152 
00153         bestInd = bestInds[bestDis(gen)];
00154 
00155         if(printGens)
00156             cout « "Gen: " « generacija « "; Fitness: " « bestFit « "; Indeks: " « bestInd « endl;
00157 
00158         if (bestFit <= THRESHOLD)
00159             break;
00160         if (generacija != GENERATIONS - 1)
00161             cgp.goldMutate(population[bestInd], population);
00162     }
00163 
00164     bestI = &population[bestInd];
00165 
00166     isSimulated = true;
00167 
00168     printFunction();
00169 
00170     probRunTime.endTimer();
00171 }
```

## 6.14 FuncProblem.hpp

```
00001 #ifndef FUNCPROBLEM_HPP
00002 #define FUNCPROBLEM_HPP
00003 
00004 #include "../Problem.hpp"
00005 #include "../cgp/CGP.hpp"
00006 
00007 #undef TYPE
00008 #define TYPE double
00009 
00010 namespace parallel_cgp {
00014     class FuncProblem : public Problem {
00015     private:
00021         const static int NUM_OPERANDS = 9;
00022         const static int BI_OPERANDS = 5;
00023         const static int INPUTS = 2;
00024         const static int OUTPUTS = 1;
00025 
00030         int GENERATIONS = 5000;
00031         int ROWS = 8;
00032         int COLUMNS = 8;
00033         int LEVELS_BACK = 1;
00034         int POPULATION_SIZE = 15;
00035         int THRESHOLD = 0;
00036 
00040         bool isSimulated = false;
00041 
00045         std::function<TYPE(TYPE x, TYPE y)> func =
00046             [](TYPE x, TYPE y) { return (pow(x, 2) + 2 * x * y + y); };
00047 
00048         TYPE computeNode(int operand, TYPE value1, TYPE value2) override;
00049         TYPE fitness(TYPE x, TYPE y, TYPE res);
00050         void problemSimulator(parallel_cgp::CGPIndividual& individual, TYPE& fit) override;
00051         std::string evalFunction(int CGPNodeNum) override;
00052     public:
00056         FuncProblem() {};
00060         FuncProblem(int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int POPULATION_SIZE, int
      THRESHOLD, std::function<TYPE(TYPE x, TYPE y)> func)
```

```
00061             : GENERATIONS(GENERATIONS), ROWS(ROWS), COLUMNS(COLUMNS), LEVELS_BACK(LEVELS_BACK),
      POPULATION_SIZE(POPULATION_SIZE), THRESHOLD(THRESHOLD), func(func) {
00062         };
00063
00067         void problemRunner() override;
00071         void printFunction() override;
00072     };
00073 }
00074
00075 #endif
```

## 6.15 FuncTester.hpp

```
00001 #ifndef FUNCTESTER_HPP
00002 #define FUNCTESTER_HPP
00003
00004 #include "../Tester.hpp"
00005 #include "../Timer.hpp"
00006 #include "FuncProblem.hpp"
00007
00008 namespace parallel_cgp {
00012     struct FuncParam {
00013         FuncParam() {}
00014         FuncParam(int gens, int rows, int cols, int levels, int pop, int thresh) : gens(gens),
      rows(rows), cols(cols), levels(levels), pop(pop), thresh(thresh) {}
00016         int gens;
00018         int rows;
00020         int cols;
00022         int levels;
00024         int pop;
00026         int thresh;
00027     };
00028
00032     class SeqFuncTester : private Tester
00033     {
00034     private:
00035         std::string funcs[6] = { "smallSimpleSeqFuncTest", "mediumSimpleSeqFuncTest",
      "largeSimpleSeqFuncTest", "smallComplexSeqFuncTest", "mediumComplexSeqFuncTest",
      "largeComplexSeqFuncTest" };
00036         FuncParam params[6] = { FuncParam(GENERATIONS, SMALL_ROWS, SMALL_COLUMNS, SMALL_LEVELS,
      SMALL_POP_SIZE, -1),
00037             FuncParam(GENERATIONS, MEDIUM_ROWS, MEDIUM_COLUMNS, MEDIUM_LEVELS, MEDIUM_POP_SIZE, -1),
00038             FuncParam(GENERATIONS, LARGE_ROWS, LARGE_COLUMNS, LARGE_LEVELS, LARGE_POP_SIZE, -1),
00039             FuncParam(GENERATIONS, SMALL_ROWS, SMALL_COLUMNS, SMALL_LEVELS, SMALL_POP_SIZE, -1),
00040             FuncParam(GENERATIONS, MEDIUM_ROWS, MEDIUM_COLUMNS, MEDIUM_LEVELS, MEDIUM_POP_SIZE, -1),
00041             FuncParam(GENERATIONS, LARGE_ROWS, LARGE_COLUMNS, LARGE_LEVELS, LARGE_POP_SIZE, -1) };
00042         std::function<TYPE(TYPE x, TYPE y)> func[2] = { [](TYPE x, TYPE y) { return (pow(x, 2) + 2 * x
      * y + y); } , [](TYPE x, TYPE y) { return (pow(x, 3) * sin(y) + 2 * cos(x) * pow(y, 2) + 4 * pow(x, 2)
      * pow(y, 3) - 3 * sin(x) * cos(y)); } };
00043
00044         void test(std::string testName, int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int
      POPULATION_SIZE, int THRESHOLD, std::function<TYPE(TYPE x, TYPE y)> func) {
00045             Timer testTimer("funcTestTimer");
00046
00047             FuncProblem problem(GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE, THRESHOLD,
      func);
00048             problem.problemRunner();
00049
00050             testTimer.endTimer();
00051
00052             saveResults(testName, GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE);
00053         }
00054     public:
00059         SeqFuncTester() : Tester("SeqFuncTest") {
00060             for (int f = 0; f < (sizeof(funcs) / sizeof(*funcs)); f++) {
00061                 for (int i = 0; i < ROUNDS; i++) {
00062                     if (f < 3)
00063                         test(funcs[f], params[f].gens, params[f].rows, params[f].cols,
      params[f].levels, params[f].pop, params[f].thresh, func[0]);
00064                     else
00065                         test(funcs[f], params[f].gens, params[f].rows, params[f].cols,
      params[f].levels, params[f].pop, params[f].thresh, func[1]);
00066                 }
00067             }
00068         }
00069     };
00070
00074     class ParFuncTester : private Tester
00075     {
00076     private:
00077         std::string funcs[6] = { "smallSimpleParFuncTest", "mediumSimpleParFuncTest",
      "largeSimpleParFuncTest", "smallComplexParFuncTest", "mediumComplexParFuncTest",
      "largeComplexParFuncTest" };
```

```
00078         FuncParam params[6] = { FuncParam(GENERATIONS, SMALL_ROWS, SMALL_COLUMNS, SMALL_LEVELS,
      SMALL_POP_SIZE, -1),
00079             FuncParam(GENERATIONS, MEDIUM_ROWS, MEDIUM_COLUMNS, MEDIUM_LEVELS, MEDIUM_POP_SIZE, -1),
00080             FuncParam(GENERATIONS, LARGE_ROWS, LARGE_COLUMNS, LARGE_LEVELS, LARGE_POP_SIZE, -1),
00081             FuncParam(GENERATIONS, SMALL_ROWS, SMALL_COLUMNS, SMALL_LEVELS, SMALL_POP_SIZE, -1),
00082             FuncParam(GENERATIONS, MEDIUM_ROWS, MEDIUM_COLUMNS, MEDIUM_LEVELS, MEDIUM_POP_SIZE, -1),
00083             FuncParam(GENERATIONS, LARGE_ROWS, LARGE_COLUMNS, LARGE_LEVELS, LARGE_POP_SIZE, -1) };
00084         std::function<TYPE(TYPE x, TYPE y)> func[2] = { [](TYPE x, TYPE y) { return (pow(x, 2) + 2 * x
      * y + y); } , [](TYPE x, TYPE y) { return (pow(x, 3) * sin(y) + 2 * cos(x) * pow(y, 2) + 4 * pow(x, 2)
      * pow(y, 3) - 3 * sin(x) * cos(y)); } };
00085
00086         void test(std::string testName, int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int
      POPULATION_SIZE, int THRESHOLD, std::function<TYPE(TYPE x, TYPE y)> func, int THREAD_NUM) {
00087             Timer testTimer("funcTestTimer");
00088
00089             omp_set_num_threads(THREAD_NUM);
00090
00091             FuncProblem problem(GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE, THRESHOLD,
      func);
00092             problem.problemRunner();
00093
00094             testTimer.endTimer();
00095
00096             saveResults(testName, GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE);
00097         }
00098     public:
00103         ParFuncTester() : Tester("ParFuncTest") {
00104             for (int f = 0; f < (sizeof(funcs) / sizeof(*funcs)); f++) {
00105                 for (int t = 0; t < (sizeof(threadNums) / sizeof(*threadNums)); t++) {
00106                     for (int i = 0; i < ROUNDS; i++) {
00107                         if (f < 3)
00108                             test(funcs[f] + std::to_string(threadNums[t]) + "T", params[f].gens,
      params[f].rows, params[f].cols, params[f].levels, params[f].pop, params[f].thresh, func[0],
      threadNums[t]);
00109                         else
00110                             test(funcs[f] + std::to_string(threadNums[t]) + "T", params[f].gens,
      params[f].rows, params[f].cols, params[f].levels, params[f].pop, params[f].thresh, func[1],
      threadNums[t]);
00111                     }
00112                 }
00113             }
00114         }
00115     };
00116 }
00117
00118 #endif
```

## 6.16 main.cpp

```
00001 #include "Problem.hpp"
00002 #include "Timer.hpp"
00003 #include "boolProblem/BoolTester.hpp"
00004 #include "funcProblem/FuncTester.hpp"
00005 #include "waitProblem/WaitTester.hpp"
00006 #include "adProblem/ADTester.hpp"
00007 #include "boolProblem/BoolProblem.hpp"
00008 #include "funcProblem/FuncProblem.hpp"
00009 #include "waitProblem/WaitProblem.hpp"
00010 #include "adProblem/ADProblem.hpp"
00011
00012 #include <iostream>
00013 #include <omp.h>
00014
00015 using namespace std;
00016 using namespace parallel_cgp;
00017
00018 #if (defined(_OPENMP) && (defined(OMPCGP) || defined(OMPSIM) || defined(OMPRUN)))
00019 #define BoolTester ParBoolTester
00020 #define ParityTester ParParityTester
00021 #define FuncTester ParFuncTester
00022 #define ADTester ParADTester
00023 #define WaitTester ParWaitTester
00024 #else
00025 #define BoolTester SeqBoolTester
00026 #define ParityTester SeqParityTester
00027 #define FuncTester SeqFuncTester
00028 #define ADTester SeqADTester
00029 #define WaitTester SeqWaitTester
00030 #endif
00031
00032 int main() {
00033     BoolTester boolTest;
00034     ParityTester parityTest;
```

```
00035     FuncTester funcTest;
00036     ADTester adTest;
00037     WaitTester waitTest;
00038 
00039     return 0;
00040 }
```

## 6.17 Problem.hpp

```
00001 #ifndef PROBLEM_HPP
00002 #define PROBLEM_HPP
00003 #define TYPE double
00004 
00005 #include "Timer.hpp"
00006 #include "cgp/CGPIndividual.hpp"
00007 #include <cmath>
00008 #include <random>
00009 #include <cfloat>
00010 #include <climits>
00011 
00012 namespace parallel_cgp {
00013     class Problem {
00014     private:
00020         virtual void problemSimulator(parallel_cgp::CGPIndividual &individual, TYPE &fit) {}
00025         virtual std::string evalFunction(int CGPNodeNum) = 0;
00026     public:
00030         virtual ~Problem() = default;
00034         CGPIndividual *bestI;
00035 
00039         bool printGens = false;
00040 
00047         int NUM_OPERANDS = 9;
00049         int BI_OPERANDS = 5;
00051         int GENERATIONS = 5000;
00053         int ROWS = 8;
00055         int COLUMNS = 8;
00057         int LEVELS_BACK = 3;
00059         int INPUTS = 6;
00061         int OUTPUTS = 1;
00063         int POPULATION_SIZE = 20;
00065 
00072         virtual TYPE computeNode(int operand, TYPE value1, TYPE value2) {
00073             switch (operand) {
00074             case 1:
00075                 return value1 + value2;
00076             case 2:
00077                 return value1 - value2;
00078             case 3:
00079                 return value1 * value2;
00080             case 4:
00081                 return (value2 == 0) ? 0 : value1 / value2;
00082             case 5:
00083                 return sin(value1);
00084             case 6:
00085                 return cos(value1);
00086             case 7:
00087                 return value1 > 0 ? sqrt(value1) : value1;
00088             case 8:
00089                 return pow(value1, 2);
00090             case 9:
00091                 return pow(2, value1);
00092             default:
00093                 return 0;
00094             }
00095         }
00099         virtual TYPE fitness(TYPE fit) { return fit; }
00100 
00104         virtual void problemRunner() = 0;
00108         virtual void printFunction() = 0;
00109     };
00110 }
00111 
00112 #endif
```

## 6.18 Tester.hpp

```
00001 #ifndef TESTER_HPP
00002 #define TESTER_HPP
```

```
00003
00004 #include "Timer.hpp"
00005 #include <omp.h>
00006 #include <string>
00007 #include <iostream>
00008 #include <fstream>
00009
00010 #ifndef _OPENMP
00011 #define omp_set_num_threads(threads) 0
00012 #endif
00013
00014 namespace parallel_cgp {
00018     class Tester
00019     {
00020     private:
00021         std::string testerName;
00022         std::string filename;
00023     public:
00029         const static int ROUNDS = 10;
00031         const static int GENERATIONS = 1000;
00033         const static int SMALL_ROWS = 4;
00035         const static int MEDIUM_ROWS = 8;
00037         const static int LARGE_ROWS = 10;
00039         const static int SMALL_COLUMNS = 4;
00041         const static int MEDIUM_COLUMNS = 8;
00043         const static int LARGE_COLUMNS = 10;
00045         const static int SMALL_LEVELS = 0;
00047         const static int MEDIUM_LEVELS = 1;
00049         const static int LARGE_LEVELS = 3;
00051         const static int SMALL_POP_SIZE = 5;
00053         const static int MEDIUM_POP_SIZE = 8;
00055         const static int LARGE_POP_SIZE = 16;
00057         inline const static int threadNums[6] = { 1, 2, 4, 8, 16, 32 };
00059
00064         Tester(std::string testerName) : testerName(testerName), filename(testerName) {
00065             filename.append(".csv");
00066             std::ofstream myFile;
00067             myFile.open(filename);
00068             myFile.close();
00069         }
00070
00075         void saveResults(std::string testName, int gens, int rows, int cols, int levels, int pop) {
00076             Timer::saveTimes(filename, testName, gens, rows, cols, levels, pop);
00077
00078             std::cout « "---------------------------------------" « std::endl;
00079             std::cout « "TEST NAME: " « testName « std::endl;
00080             std::cout « "---------------------------------------" « std::endl;
00081             std::cout « "GENS: " « gens « ", ROWS: " « rows « ", COLUMNS: " « cols
00082                 « ", LEVELS BACK: " « levels « ", POP SIZE: " « pop « std::endl;
00083             std::cout « "---------------------------------------" « std::endl;
00084             Timer::clearTimes();
00085         }
00086     };
00087 }
00088
00089 #endif
```

## 6.19 Timer.hpp

```
00001 #ifndef TIMER_HPP
00002 #define TIMER_HPP
00003
00004 #include <omp.h>
00005 #include <chrono>
00006 #include <map>
00007 #include <string>
00008 #include <functional>
00009 #include <iostream>
00010 #include <fstream>
00011
00013 #ifdef _OPENMP
00014 #define timerFunc() omp_get_wtime()
00015 #define timerDiff(startTime, endTime) (endTime - startTime)
00016 #define TIME_UNIT double
00017 #else
00018 #define timerFunc() std::chrono::steady_clock::now()
00019 #define timerDiff(startTime, endTime) (std::chrono::duration_cast<std::chrono::microseconds>(endTime -
      startTime).count() / 1000000.0)
00020 #define TIME_UNIT std::chrono::steady_clock::time_point
00021 #endif
00022
00023 namespace parallel_cgp {
00024
```

```
00025     class Timer
00026     {
00027     private:
00029         inline static std::map<std::string, std::vector<double» mapa;
00030
00031         std::string funcName;
00032         TIME_UNIT start;
00033         double end;
00034     public:
00039         Timer(std::string funcName) : funcName(funcName), start(timerFunc()), end(0) {}
00040
00044         void endTimer() {
00045             end = timerDiff(start, timerFunc());
00046
00047             #pragma omp critical
00048             parallel_cgp::Timer::mapa[funcName].push_back(end);
00049         }
00050
00054         static void printTimes() {
00055             for (const auto& [key, value] : parallel_cgp::Timer::mapa)
00056                 for (const auto& val : value)
00057                     std::cout « '[' « key « "] = " « val « "; " « std::endl;
00058         }
00059
00064         static void saveTimes(std::string filename, std::string testName, int gens, int rows, int
    cols, int levels, int pop) {
00065             std::ofstream myFile;
00066             myFile.open(filename, std::ios_base::app);
00067             myFile « "TEST NAME: " « testName;
00068             myFile « ", GENS: " « gens « ", ROWS: " « rows « ", COLUMNS: " « cols
00069                 « ", LEVELS BACK: " « levels « ", POP SIZE: " « pop « std::endl;
00070
00071             for (const auto& [key, value] : parallel_cgp::Timer::mapa) {
00072                 myFile « '[' « key « "],";
00073                 for (const auto& val : value)
00074                     myFile « val « ',';
00075                 myFile « std::endl;
00076             }
00077             myFile.close();
00078         }
00079
00083         static void clearTimes() {
00084             parallel_cgp::Timer::mapa.clear();
00085         }
00086     };
00087 }
00088
00089 #endif
00090
```

## 6.20 WaitProblem.cpp

```
00001 #include "WaitProblem.hpp"
00002
00003 using namespace std;
00004 using namespace parallel_cgp;
00005
00006 TYPE WaitProblem::fitness(TYPE prev) {
00007     return ++prev;
00008 }
00009
00010 void WaitProblem::printFunction() {
00011     if (isSimulated)
00012         cout « "Funkcija: " « evalFunction(0) « endl;
00013     else
00014         cout « "Problem nije simuliran." « endl;
00015 }
00016
00017 string WaitProblem::evalFunction(int CGPNodeNum) {
00018     ostringstream oss;
00019
00020     if (!CGPNodeNum) {
00021         oss « "Wait time: " « WAIT_TIME « "ms";
00022         return oss.str();
00023     }
00024
00025     return "";
00026 }
00027
00028 void WaitProblem::problemSimulator(CGPIndividual& individual, TYPE& fit) {
00029     Timer probSimTime("problemSimulatorTimer");
00030
00031     function<TYPE(int op, TYPE v1, TYPE v2)> compNode =
```

```
00032         [&](int op, TYPE v1, TYPE v2) { return computeNode(op, v1, v2); };
00033
00034     for (int iter = 0; iter < 10; iter++) {
00035         vector<TYPE> input;
00036         input.push_back(iter);
00037
00038         individual.evaluateValue(input, compNode);
00039         waitFunc();
00040     }
00041     fit = fitness(fit);
00042
00043     probSimTime.endTimer();
00044 }
00045
00046 void WaitProblem::problemRunner() {
00047     Timer probRunTime("problemRunnerTimer");
00048
00049     CGP cgp(ROWS, COLUMNS, LEVELS_BACK, INPUTS, OUTPUTS, NUM_OPERANDS, BI_OPERANDS, POPULATION_SIZE);
00050
00051     vector<CGPIndividual> population(POPULATION_SIZE);
00052     int bestInd = 0, generacija = 0;
00053
00054     cgp.generatePopulation(population);
00055
00056     for (generacija = 0; generacija < GENERATIONS; generacija++) {
00057         TYPE bestFit = 0;
00058         bestInd = 0;
00059         vector<int> bestInds;
00060         random_device rd;
00061         mt19937 gen(rd());
00062
00063         for (int clan = 0; clan < POPULATION_SIZE; clan++) {
00064
00065             TYPE fit = generacija;
00066             problemSimulator(population[clan], fit);
00067
00068             if (fit > bestFit) {
00069                 bestFit = fit;
00070                 bestInds.clear();
00071                 bestInds.push_back(clan);
00072             }
00073             else if (fit == bestFit)
00074                 bestInds.push_back(clan);
00075         }
00076
00077         if (bestInds.size() > 1)
00078             bestInds.erase(bestInds.begin());
00079         if (bestInds.size() == 0)
00080             bestInds.push_back(0);
00081
00082         uniform_int_distribution<> bestDis(0, static_cast<int>(bestInds.size()) - 1);
00083
00084         bestInd = bestInds[bestDis(gen)];
00085
00086         if(printGens)
00087             cout « "Gen: " « generacija « "; Fitness: " « bestFit « "; Indeks: " « bestInd « endl;
00088
00089         if (generacija != GENERATIONS - 1)
00090             cgp.goldMutate(population[bestInd], population);
00091     }
00092
00093     bestI = &population[bestInd];
00094
00095     isSimulated = true;
00096
00097     printFunction();
00098
00099     probRunTime.endTimer();
00100 }
```

## 6.21 WaitProblem.hpp

```
00001 #ifndef WAITPROBLEM_HPP
00002 #define WAITPROBLEM_HPP
00003
00004 #include "../Problem.hpp"
00005 #include "../cgp/CGP.hpp"
00006 #include <chrono>
00007 #include <thread>
00008
00009 #undef TYPE
00010 #define TYPE double
00011
```

```
00012 namespace parallel_cgp {
00016     class WaitProblem : public Problem {
00017     private:
00022         int GENERATIONS = 200;
00023         int ROWS = 8;
00024         int COLUMNS = 8;
00025         int LEVELS_BACK = 3;
00026         int POPULATION_SIZE = 15;
00027         int INPUTS = 1;
00028         int OUTPUTS = 1;
00029 
00033         int WAIT_TIME = 50;
00034 
00038         bool isSimulated = false;
00039 
00043         const std::function<void()> waitFunc =
00044             [&]() { std::this_thread::sleep_for(std::chrono::nanoseconds(WAIT_TIME)); };
00045 
00046         TYPE fitness(TYPE prev) override;
00047         void problemSimulator(CGPIndividual& individual, TYPE& fit) override;
00048         std::string evalFunction(int CGPNodeNum) override;
00049     public:
00053         WaitProblem() {};
00057         WaitProblem(int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int POPULATION_SIZE, int
      WAIT_TIME)
00058             : GENERATIONS(GENERATIONS), ROWS(ROWS), COLUMNS(COLUMNS), LEVELS_BACK(LEVELS_BACK),
      POPULATION_SIZE(POPULATION_SIZE), WAIT_TIME(WAIT_TIME) {};
00059 
00063         void problemRunner() override;
00067         void printFunction() override;
00068     };
00069 }
00070 
00071 #endif
```

## 6.22 WaitTester.hpp

```
00001 #ifndef WAITTESTER_HPP
00002 #define WAITTESTER_HPP
00003 
00004 #include "../Tester.hpp"
00005 #include "../Timer.hpp"
00006 #include "WaitProblem.hpp"
00007 
00008 namespace parallel_cgp {
00012     struct WaitParam {
00013         WaitParam() {}
00014         WaitParam(int gens, int rows, int cols, int levels, int pop, int time) : gens(gens),
      rows(rows), cols(cols), levels(levels), pop(pop), time(time) {}
00016         int gens;
00018         int rows;
00020         int cols;
00022         int levels;
00024         int pop;
00026         int time;
00027     };
00028 
00032     class SeqWaitTester : private Tester
00033     {
00034     private:
00035         std::string funcs[3] = { "smallSeqWaitTest", "mediumSeqWaitTest", "largeSeqWaitTest" };
00036         WaitParam params[3] = { WaitParam(GENERATIONS, SMALL_ROWS, SMALL_COLUMNS, SMALL_LEVELS,
      SMALL_POP_SIZE, 1),
00037             WaitParam(GENERATIONS, MEDIUM_ROWS, MEDIUM_COLUMNS, MEDIUM_LEVELS, MEDIUM_POP_SIZE, 1),
00038             WaitParam(GENERATIONS, LARGE_ROWS, LARGE_COLUMNS, LARGE_LEVELS, LARGE_POP_SIZE, 1) };
00039 
00040         void test(std::string testName, int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int
      POPULATION_SIZE, int WAIT_TIME) {
00041             Timer testTimer("waitTestTimer");
00042 
00043             WaitProblem problem(GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE, WAIT_TIME);
00044             problem.problemRunner();
00045 
00046             testTimer.endTimer();
00047 
00048             saveResults(testName, GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE);
00049         }
00050     public:
00055         SeqWaitTester() : Tester("SeqWaitTest") {
00056             for (int f = 0; f < (sizeof(funcs) / sizeof(*funcs)); f++) {
00057                 for (int i = 0; i < ROUNDS; i++) {
00058                     test(funcs[f], params[f].gens, params[f].rows, params[f].cols, params[f].levels,
      params[f].pop, params[f].time);
```

```
00059                 }
00060             }
00061         }
00062     };
00063 
00067     class ParWaitTester : private Tester
00068     {
00069     private:
00070         std::string funcs[3] = { "smallParWaitTest", "mediumParWaitTest", "largeParWaitTest" };
00071         WaitParam params[3] = { WaitParam(GENERATIONS, SMALL_ROWS, SMALL_COLUMNS, SMALL_LEVELS,
      SMALL_POP_SIZE, 1),
00072             WaitParam(GENERATIONS, MEDIUM_ROWS, MEDIUM_COLUMNS, MEDIUM_LEVELS, MEDIUM_POP_SIZE, 1),
00073             WaitParam(GENERATIONS, LARGE_ROWS, LARGE_COLUMNS, LARGE_LEVELS, LARGE_POP_SIZE, 1) };
00074 
00075         void test(std::string testName, int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int
      POPULATION_SIZE, int WAIT_TIME, int THREAD_NUM) {
00076             Timer testTimer("waitTestTimer");
00077 
00078             omp_set_num_threads(THREAD_NUM);
00079 
00080             WaitProblem problem(GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE, WAIT_TIME);
00081             problem.problemRunner();
00082 
00083             testTimer.endTimer();
00084 
00085             saveResults(testName, GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE);
00086         }
00087     public:
00092         ParWaitTester() : Tester("ParWaitTest") {
00093             for (int f = 0; f < (sizeof(funcs) / sizeof(*funcs)); f++) {
00094                 for (int t = 0; t < (sizeof(threadNums) / sizeof(*threadNums)); t++) {
00095                     for (int i = 0; i < ROUNDS; i++) {
00096                         test(funcs[f] + std::to_string(threadNums[t]) + "T", params[f].gens,
      params[f].rows, params[f].cols, params[f].levels, params[f].pop, params[f].time, threadNums[t]);
00097                     }
00098                 }
00099             }
00100         }
00101     };
00102 }
00103 
00104 #endif
```

# Index