ParallelCGP

1.0.0

Generated by Doxygen 1.13.2

Chapter 1

ParallelCGP

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

1.1 Pokretanje

1.1.1 Unix

clang++ 18.1.0

```
cmake -S ../ -B . -DCMAKE_CXX_COMPILER=clang++
```

1.1.2 Windows

g++ 11.2.0

```
cmake -S ../ -B . -DCMAKE_CXX_COMPILER=g++
```

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.

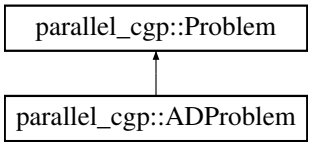
The documentation for this struct was generated from the following file:

- 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 191 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.

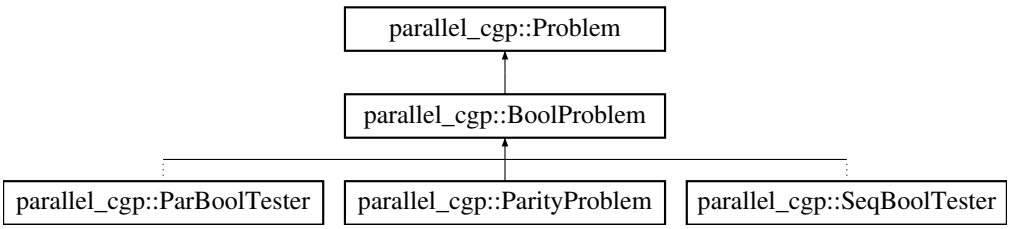
The documentation for this struct was generated from the following file:

- 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 23 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 38 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 82 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.

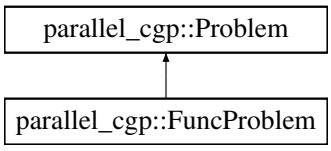
The documentation for this struct was generated from the following file:

- 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.

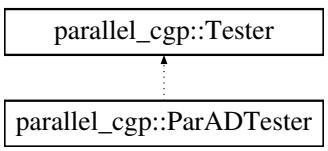
The documentation for this class was generated from the following files:

- 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 (ADParam customParams)

5.11.1 Detailed Description

Klasa koja opisuje paralelni tester Acey Deucey problema.

Definition at line 72 of file ADTester.hpp.

5.11.2 Constructor & Destructor Documentation

5.11.2.1 ParADTester()

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

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

Definition at line 98 of file ADTester.hpp.

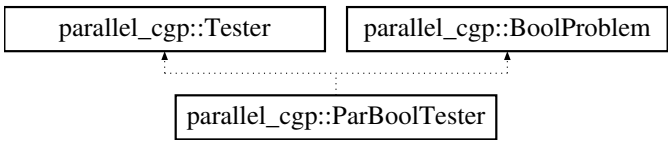
The documentation for this class was generated from the following file:

- adProblem/ADTester.hpp

5.12 parallel_cgp::ParBoolTester Class Reference

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

Inheritance diagram for parallel_cgp::ParBoolTester:



Public Member Functions

- ParBoolTester (BoolParam customParams)

5.12.1 Detailed Description

Klasa koja opisuje paralelni tester Bool problema.

Definition at line 79 of file BoolTester.hpp.

5.12.2 Constructor & Destructor Documentation

5.12.2.1 ParBoolTester()

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

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

Definition at line 110 of file BoolTester.hpp.

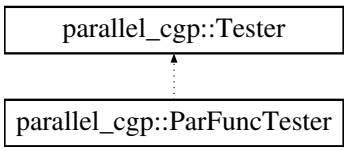
The documentation for this class was generated from the following file:

- boolProblem/BoolTester.hpp

5.13 parallel_cgp::ParFuncTester Class Reference

`#include <FuncTester.hpp>`

Inheritance diagram for parallel_cgp::ParFuncTester:



Public Member Functions

- ParFuncTester (FuncParam customParams)

5.13.1 Detailed Description

Klasa koja opisuje sekvencijski tester Func problema.

Definition at line 82 of file FuncTester.hpp.

5.13.2 Constructor & Destructor Documentation

5.13.2.1 ParFuncTester()

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

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

Definition at line 113 of file FuncTester.hpp.

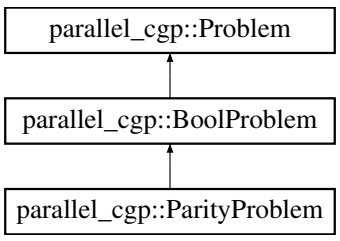
The documentation for this class was generated from the following file:

- 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.

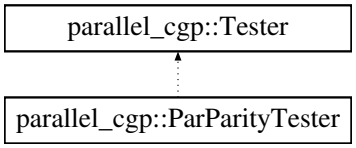
The documentation for this class was generated from the following file:

- boolProblem/BoolProblem.hpp

5.15 parallel_cgp::ParParityTester Class Reference

`#include <BoolTester.hpp>`

Inheritance diagram for parallel_cgp::ParParityTester:



Public Member Functions

- ParParityTester (BoolParam customParams)

5.15.1 Detailed Description

Klasa koja opisuje paralelni tester Parity problema.

Definition at line 175 of file BoolTester.hpp.

5.15.2 Constructor & Destructor Documentation

5.15.2.1 ParParityTester()

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

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

Definition at line 201 of file BoolTester.hpp.

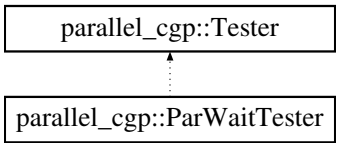
The documentation for this class was generated from the following file:

- boolProblem/BoolTester.hpp

5.16 parallel_cgp::ParWaitTester Class Reference

`#include <WaitTester.hpp>`

Inheritance diagram for parallel_cgp::ParWaitTester:



Public Member Functions

- ParWaitTester (WaitParam customParams)

5.16.1 Detailed Description

Klasa koja opisuje paralelni tester Wait problema.

Definition at line 74 of file WaitTester.hpp.

5.16.2 Constructor & Destructor Documentation

5.16.2.1 ParWaitTester()

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

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

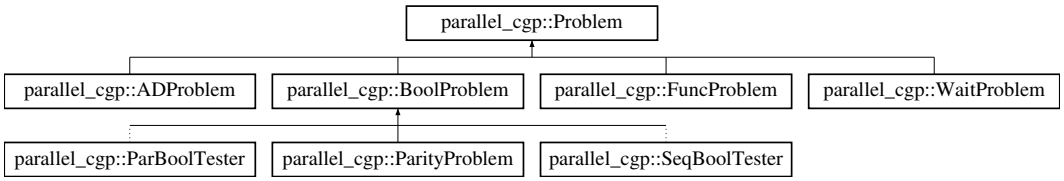
Definition at line 100 of file WaitTester.hpp.

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 15 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 74 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 101 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 36 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 51 of file Problem.hpp.

5.17.4.3 COLUMNS

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

Broj stupaca CGP mreze.

Definition at line 57 of file Problem.hpp.

5.17.4.4 GENERATIONS

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

Broj generacija koji se vrti.

Definition at line 53 of file Problem.hpp.

5.17.4.5 INPUTS

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

Broj ulaza u CGP mrezu.

Definition at line 61 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 59 of file Problem.hpp.

5.17.4.7 NUM_OPERANDS

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

Ukupni broj operanada.

Definition at line 49 of file Problem.hpp.

5.17.4.8 OUTPUTS

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

Broj izlaza iz CGP mrezu.

Definition at line 63 of file Problem.hpp.

5.17.4.9 POPULATION_SIZE

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

Velicina populacije.

Definition at line 65 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 41 of file Problem.hpp.

5.17.4.11 ROWS

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

Broj redova CGP mreze.

Definition at line 55 of file Problem.hpp.

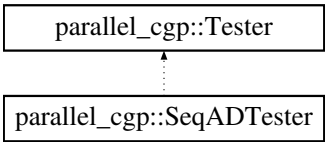
The documentation for this class was generated from the following file:

- Problem.hpp

5.18 parallel_cgp::SeqADTester Class Reference

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

Inheritance diagram for parallel_cgp::SeqADTester:



Public Member Functions

- SeqADTester (ADParam customParams)

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 (
            ADParam customParams)  [inline]
```

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

Definition at line 54 of file ADTester.hpp.

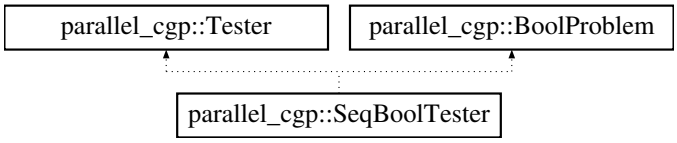
The documentation for this class was generated from the following file:

- adProblem/ADTester.hpp

5.19 parallel_cgp::SeqBoolTester Class Reference

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

Inheritance diagram for parallel_cgp::SeqBoolTester:



Public Member Functions

- SeqBoolTester (BoolParam customParams)

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 (
            BoolParam customParams)  [inline]
```

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

Definition at line 58 of file BoolTester.hpp.

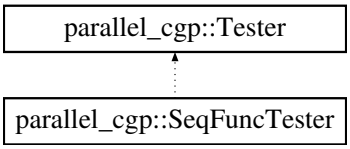
The documentation for this class was generated from the following file:

- boolProblem/BoolTester.hpp

5.20 parallel_cgp::SeqFuncTester Class Reference

`#include <FuncTester.hpp>`

Inheritance diagram for parallel_cgp::SeqFuncTester:



Public Member Functions

- SeqFuncTester (FuncParam customParams)

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 (
            FuncParam customParams)  [inline]
```

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

Definition at line 61 of file FuncTester.hpp.

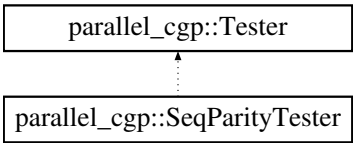
The documentation for this class was generated from the following file:

- funcProblem/FuncTester.hpp

5.21 parallel_cgp::SeqParityTester Class Reference

`#include <BoolTester.hpp>`

Inheritance diagram for parallel_cgp::SeqParityTester:



Public Member Functions

- SeqParityTester (BoolParam customParams)

5.21.1 Detailed Description

Klasa koja opisuje sekvencijski tester Parity problema.

Definition at line 135 of file BoolTester.hpp.

5.21.2 Constructor & Destructor Documentation

5.21.2.1 SeqParityTester()

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

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

Definition at line 159 of file BoolTester.hpp.

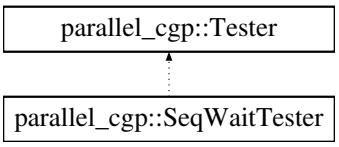
The documentation for this class was generated from the following file:

- boolProblem/BoolTester.hpp

5.22 parallel_cgp::SeqWaitTester Class Reference

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

Inheritance diagram for parallel_cgp::SeqWaitTester:



Public Member Functions

- SeqWaitTester (WaitParam customParams)

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 (
            WaitParam customParams)  [inline]
```

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

Definition at line 56 of file WaitTester.hpp.

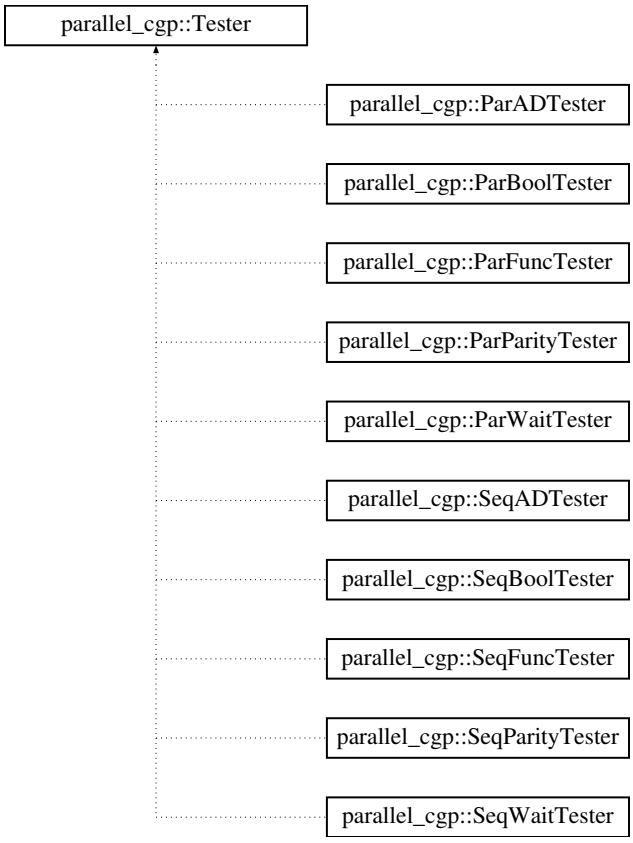
The documentation for this class was generated from the following file:

- 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

- static std::string VERSION_NAME = ""

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 SPECIAL_ROWS = 1
- static const int SMALL_COLUMNS = 4
- static const int MEDIUM_COLUMNS = 8
- static const int LARGE_COLUMNS = 10
- static const int SPECIAL_COLUMNS = 100
- static const int SMALL_LEVELS = 0
- static const int MEDIUM_LEVELS = 1
- static const int LARGE_LEVELS = 3
- static const int SPECIAL_LEVELS = 10
- static const int SMALL_POP_SIZE = 5
- static const int MEDIUM_POP_SIZE = 8
- static const int LARGE_POP_SIZE = 16
- static const int SPECIAL_POP_SIZE = 5
- static std::vector< int > threadNums = { 1, 2, 4, 8, 16 }

5.23.1 Detailed Description

Klasa koja opisuje jedan Tester problema.

Definition at line 19 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 79 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 91 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 38 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 52 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 60 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 68 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 44 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 50 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 58 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 66 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 42 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 36 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 48 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 56 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 64 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 40 of file Tester.hpp.

5.23.4.15 SPECIAL_COLUMNS

```
const int parallel_cgp::Tester::SPECIAL_COLUMNS = 100  [static]
```

Broj CGP stupaca za poseban test.

Definition at line 54 of file Tester.hpp.

5.23.4.16 SPECIAL_LEVELS

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

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

Definition at line 62 of file Tester.hpp.

5.23.4.17 SPECIAL_POP_SIZE

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

Velicina populacije za poseban test.

Definition at line 70 of file Tester.hpp.

5.23.4.18 SPECIAL_ROWS

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

Broj CGP stupaca za poseban test.

Definition at line 46 of file Tester.hpp.

5.23.4.19 threadNums

```
std::vector<int> parallel_cgp::Tester::threadNums = { 1, 2, 4, 8, 16 }  [inline], [static]
```

Koje ce se sve kolicine dretvi koristiti u testovima.

Definition at line 72 of file Tester.hpp.

5.23.4.20 VERSION_NAME

```
std::string parallel_cgp::Tester::VERSION_NAME = ""  [inline], [static]
```

Naziv verzije programa.

Note

Ova varijabla se koristi za naziv datoteke koja se stvara za svaki tester.

Definition at line 29 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.

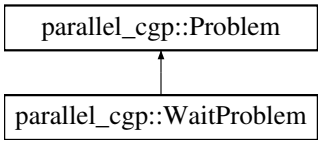
The documentation for this struct was generated from the following file:

- 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     boost::random::mt19937
      gen(chrono::duration_cast<std::chrono::nanoseconds>(chrono::system_clock::now().time_since_epoch()).count()
      * (omp_get_thread_num() + 1));
00127
00128     boost::random::uniform_int_distribution<> cardDis(1, 13);
00129
00130     for (int j = 0; j < CARD_SETS; j++) {
00131         vector<double> set;
00132         for (int i = 0; i < 3; i++)
00133             set.push_back(static_cast<double>(cardDis(gen)));
00134
00135         double card = set.back();
00136         set.pop_back();
00137         sort(set.begin(), set.end());
00138         set.push_back(card);
00139
```

```
00140         sets.push_back(set);
00141     }
00142
00143     for (generacija = 0; generacija < GENERATIONS; generacija++) {
00144         double bestFit = DBL_MIN;
00145         bestInd = 0;
00146         vector<int> bestInds;
00147
00148         for (int clan = 0; clan < POPULATION_SIZE; clan++) {
00149
00150             double fit = 0;
00151             problemSimulator(population[clan], fit);
00152
00153             if (fit > bestFit) {
00154                 bestFit = fit;
00155                 bestInds.clear();
00156                 bestInds.push_back(clan);
00157             }
00158             else if (fit == bestFit)
00159                 bestInds.push_back(clan);
00160         }
00161
00162         if (bestInds.size() > 1)
00163             bestInds.erase(bestInds.begin());
00164         if (bestInds.size() == 0)
00165             bestInds.push_back(0);
00166
00167         boost::random::uniform_int_distribution<> bestDis(0, static_cast<int>(bestInds.size() - 1));
00168
00169         bestInd = bestInds[bestDis(gen)];
00170
00171         if(printGens)
00172             cout « "Gen: " « generacija « "; Fitness: " « bestFit « "; Indeks: " « bestInd « endl;
00173
00174         if (bestFit >= THRESHOLD)
00175             break;
00176         if (generacija != GENERATIONS - 1)
00177             cgp.goldMutate(population[bestInd], population);
00178     }
00179
00180     bestI = &population[bestInd];
00181
00182     isSimulated = true;
00183
00184     printFunction();
00185
00186     probRunTime.endTimer();
00187
00188     playGame();
00189 }
00190
00191 void ADProblem::playGame() {
00192     function<double(int op, double v1, double v2)> compNode =
00193         [&](int op, double v1, double v2) { return computeNode(op, static_cast<TYPE>(v1),
    static_cast<TYPE>(v2)); };
00194
00195     boost::random::mt19937
    gen(chrono::duration_cast<std::chrono::nanoseconds>(chrono::system_clock::now().time_since_epoch()).count()
    * (omp_get_thread_num() + 1));
00196
00197     boost::random::uniform_int_distribution<> cardDis(1, 13);
00198
00199     int steps = 0;
00200     int cash = STARTING_CASH, maxCash = STARTING_CASH;
00201
00202     while (cash && steps < 100 && cash < MAX_CASH) {
00203         vector<double> input;
00204         int card, win;
00205         for (int i = 0; i < 3; i++)
00206             input.push_back(static_cast<TYPE>(cardDis(gen)));
00207
00208         card = card = static_cast<int>(input.back());
00209         input.pop_back();
00210
00211         sort(input.begin(), input.end());
00212
00213         if (card > input.at(0) && card < input.at(1))
00214             win = 1;
00215         else if (card == input.at(0) || card == input.at(1))
00216             win = -1;
00217         else
00218             win = 0;
00219
00220         bestI->evaluateValue(input, compNode);
00221
00222         cout « "Cash: " « cash « "; Cards: " « input[0] « ", " « input[1] « "; Bet: " «
    ((bestI->outputGene[0].value > 1) ? "YES" : "NO")
```

```
00223             « "; Third card: " « card « ((win == 1) ? " | WIN!" : " | LOST!") « endl;
00224
00225         if (bestI->outputGene[0].value > 1) {
00226             if (win == 1)
00227                 cash += 10;
00228             else if (win == 0)
00229                 cash -= 10;
00230             else if (win == -1)
00231                 cash -= 20;
00232         }
00233
00234         if (cash > maxCash)
00235             maxCash = cash;
00236
00237         steps++;
00238     }
00239 }
```

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[4] = { "smallSeqADTest", "mediumSeqADTest", "largeSeqADTest",
      "specialSeqADTest" };
00034         ADParam params[4] = { 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             ADParam(GENERATIONS, SPECIAL_ROWS, SPECIAL_COLUMNS, SPECIAL_LEVELS, SPECIAL_POP_SIZE) };
00038
00039         void test(std::string testName, int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int
      POPULATION_SIZE) {
00040             Timer testTimer("adTestTimer");
00041
00042             ADProblem problem(GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE);
00043             problem.problemRunner();
00044
00045             testTimer.endTimer();
00046
00047             saveResults(testName, GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE);
00048         }
00049     public:
00054         SeqADTester(ADParam customParams) : Tester((customParams.pop == 0) ? "SeqADTest" :
      "CustomSeqADTest") {
00055             if(customParams.pop != 0) {
00056                 for(int i = 0; i < ROUNDS; i++)
00057                     test("CustomSeqADTest", customParams.gens, customParams.rows, customParams.cols,
      customParams.levels, customParams.pop);
00058                 return;
00059             }
00060
00061             for (int f = 0; f < (sizeof(funcs) / sizeof(*funcs)); f++) {
00062                 for (int i = 0; i < ROUNDS; i++) {
00063                     test(funcs[f], params[f].gens, params[f].rows, params[f].cols, params[f].levels,
      params[f].pop);
00064                 }
00065             }
00066         }
00067     };
00068
00072     class ParADTester : private Tester
00073     {
00074     private:
00075         std::string funcs[4] = { "smallParADTest", "mediumParADTest", "largeParADTest",
      "specialParADTest" };
00076         ADParam params[4] = { ADParam(GENERATIONS, SMALL_ROWS, SMALL_COLUMNS, SMALL_LEVELS,
      SMALL_POP_SIZE),
00077             ADParam(GENERATIONS, MEDIUM_ROWS, MEDIUM_COLUMNS, MEDIUM_LEVELS, MEDIUM_POP_SIZE),
00078             ADParam(GENERATIONS, LARGE_ROWS, LARGE_COLUMNS, LARGE_LEVELS, LARGE_POP_SIZE),
00079             ADParam(GENERATIONS, SPECIAL_ROWS, SPECIAL_COLUMNS, SPECIAL_LEVELS, SPECIAL_POP_SIZE) };
00080
00081         void test(std::string testName, int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int
      POPULATION_SIZE, int THREAD_NUM) {
00082             Timer testTimer("adTestTimer");
00083
00084             omp_set_num_threads(THREAD_NUM);
00085
00086             ADProblem problem(GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE);
00087             problem.problemRunner();
00088
00089             testTimer.endTimer();
00090
00091             saveResults(testName, GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE);
00092         }
00093     public:
00098         ParADTester(ADParam customParams) : Tester((customParams.pop == 0) ? "ParADTest" :
      "CustomParADTest") {
00099             if(customParams.pop != 0) {
00100                 for (int t = 0; t < threadNums.size(); t++) {
00101                     for(int i = 0; i < ROUNDS; i++)
00102                         test("CustomParADTest", customParams.gens, customParams.rows,
      customParams.cols, customParams.levels, customParams.pop, threadNums[t]);
00103                     return;
00104                 }
```

```
00105             }
00106
00107             for (int f = 0; f < (sizeof(funcs) / sizeof(*funcs)); f++) {
00108                 for (int t = 0; t < threadNums.size(); t++) {
00109                     for (int i = 0; i < ROUNDS; i++) {
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, threadNums[t]);
00111                     }
00112                 }
00113             }
00114         }
00115     };
00116 }
00117
00118 #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         boost::random::mt19937
      gen(chrono::duration_cast<std::chrono::nanoseconds>(chrono::system_clock::now().time_since_epoch()).count()
      * (omp_get_thread_num() + 1));
00096
00097         for (int clan = 0; clan < POPULATION_SIZE; clan++) {
00098
00099             TYPE fit = 0;
00100             problemSimulator(population[clan], fit);
00101
00102             if (fit > bestFit) {
00103                 bestFit = fit;
00104                 bestInds.clear();
00105                 bestInds.push_back(clan);
00106             }
00107             else if (fit == bestFit)
00108                 bestInds.push_back(clan);
00109         }
00110
00111         if (bestInds.size() > 1)
00112             bestInds.erase(bestInds.begin());
00113         if (bestInds.size() == 0)
00114             bestInds.push_back(0);
00115
00116         boost::random::uniform_int_distribution<> bestDis(0, static_cast<int>(bestInds.size() - 1));
00117
00118         bestInd = bestInds[bestDis(gen)];
00119
00120         if (printGens)
00121             cout « "Gen: " « generacija « "; Fitness: " « bestFit « "; Indeks: " « bestInd « endl;
00122
00123         if (bestFit == pow(2, INPUTS))
00124             break;
00125         if (generacija != GENERATIONS - 1)
00126             cgp.goldMutate(population[bestInd], population);
00127     }
00128
00129     bestI = &population[bestInd];
00130
00131     isSimulated = true;
00132
00133     printFunction();
00134
00135     probRunTime.endTimer();
00136 }
```

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[8] = { "smallSimpleSeqBoolTest", "mediumSimpleSeqBoolTest",
      "largeSimpleSeqBoolTest", "specialSimpleSeqBoolTest", "smallComplexSeqBoolTest",
      "mediumComplexSeqBoolTest", "largeComplexSeqBoolTest", "specialComplexSeqBoolTest" };
00033         BoolParam params[8] = { 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, SPECIAL_ROWS, SPECIAL_COLUMNS, SPECIAL_LEVELS,
      SPECIAL_POP_SIZE),
00037             BoolParam(Tester::GENERATIONS, SMALL_ROWS, SMALL_COLUMNS, SMALL_LEVELS, SMALL_POP_SIZE),
00038             BoolParam(Tester::GENERATIONS, MEDIUM_ROWS, MEDIUM_COLUMNS, MEDIUM_LEVELS,
      MEDIUM_POP_SIZE),
00039             BoolParam(Tester::GENERATIONS, LARGE_ROWS, LARGE_COLUMNS, LARGE_LEVELS, LARGE_POP_SIZE),
00040             BoolParam(Tester::GENERATIONS, SPECIAL_ROWS, SPECIAL_COLUMNS, SPECIAL_LEVELS,
      SPECIAL_POP_SIZE) };
00041         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]))); } };
00042
00043         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) {
00044             Timer testTimer("boolTestTimer");
00045
00046             BoolProblem problem(GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE, boolFunc);
00047             problem.problemRunner();
00048
00049             testTimer.endTimer();
00050
00051             saveResults(testName, GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE);
00052         }
00053     public:
00058         SeqBoolTester(BoolParam customParams) : Tester((customParams.pop == 0) ? "SeqBoolTest" :
      "CustomSeqBoolTest") {
00059             if(customParams.pop != 0) {
00060                 for(int i = 0; i < ROUNDS; i++)
00061                     test("CustomSeqBoolTest", customParams.gens, customParams.rows, customParams.cols,
      customParams.levels, customParams.pop, func[0]);
00062                 return;
00063             }
00064
00065             for (int f = 0; f < (sizeof(boolFuncs) / sizeof(*boolFuncs)); f++) {
00066                 for (int i = 0; i < ROUNDS; i++) {
00067                     if (f < 3)
00068                         test(boolFuncs[f], params[f].gens, params[f].rows, params[f].cols,
      params[f].levels, params[f].pop, func[0]);
00069                     else
00070                         test(boolFuncs[f], params[f].gens, params[f].rows, params[f].cols,
      params[f].levels, params[f].pop, func[1]);
00071                 }
00072             }
00073         }
00074     };
00075
00079     class ParBoolTester : private Tester, private BoolProblem
00080     {
00081     private:
00082         std::string boolFuncs[8] = { "smallSimpleParBoolTest", "mediumSimpleParBoolTest",
      "largeSimpleParBoolTest", "specialSimpleParBoolTest", "smallComplexParBoolTest",
      "mediumComplexParBoolTest", "largeComplexParBoolTest", "specialComplexParBoolTest" };
00083         BoolParam params[8] = { BoolParam(Tester::GENERATIONS, SMALL_ROWS, SMALL_COLUMNS,
      SMALL_LEVELS, SMALL_POP_SIZE),
00084             BoolParam(Tester::GENERATIONS, MEDIUM_ROWS, MEDIUM_COLUMNS, MEDIUM_LEVELS,
      MEDIUM_POP_SIZE),
00085             BoolParam(Tester::GENERATIONS, LARGE_ROWS, LARGE_COLUMNS, LARGE_LEVELS, LARGE_POP_SIZE),
00086             BoolParam(Tester::GENERATIONS, SPECIAL_ROWS, SPECIAL_COLUMNS, SPECIAL_LEVELS,
      SPECIAL_POP_SIZE),
00087             BoolParam(Tester::GENERATIONS, SMALL_ROWS, SMALL_COLUMNS, SMALL_LEVELS, SMALL_POP_SIZE),
00088             BoolParam(Tester::GENERATIONS, MEDIUM_ROWS, MEDIUM_COLUMNS, MEDIUM_LEVELS,
      MEDIUM_POP_SIZE),
00089             BoolParam(Tester::GENERATIONS, LARGE_ROWS, LARGE_COLUMNS, LARGE_LEVELS, LARGE_POP_SIZE),
00090             BoolParam(Tester::GENERATIONS, SPECIAL_ROWS, SPECIAL_COLUMNS, SPECIAL_LEVELS,
      SPECIAL_POP_SIZE) };
00091         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]))); } };
00092
00093         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) {
00094             Timer testTimer("boolTestTimer");
00095
00096             omp_set_num_threads(THREAD_NUM);
00097
00098             BoolProblem problem(GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE, boolFunc);
```

```
00099             problem.problemRunner();
00100
00101             testTimer.endTimer();
00102
00103             saveResults(testName, GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE);
00104         }
00105     public:
00110         ParBoolTester(BoolParam customParams) : Tester((customParams.pop == 0) ? "ParBoolTest" :
    "CustomParBoolTest") {
00111             if(customParams.pop != 0) {
00112                 for (int t = 0; t < threadNums.size(); t++) {
00113                     for(int i = 0; i < ROUNDS; i++)
00114                         test("CustomParBoolTest", customParams.gens, customParams.rows,
    customParams.cols, customParams.levels, customParams.pop, func[0], threadNums[t]);
00115                     return;
00116                 }
00117             }
00118
00119             for (int f = 0; f < (sizeof(boolFuncs) / sizeof(*boolFuncs)); f++) {
00120                 for (int t = 0; t < threadNums.size(); t++) {
00121                     for (int i = 0; i < ROUNDS; i++) {
00122                         if (f < 3)
00123                             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]);
00124                         else
00125                             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]);
00126                     }
00127                 }
00128             }
00129         }
00130     };
00131
00135     class SeqParityTester : private Tester
00136     {
00137     private:
00138         std::string parityFuncs[4] = { "smallSeqParityTest", "mediumSeqParityTest",
    "largeSeqParityTest", "specialSeqParityTest" };
00139         BoolParam params[4] = { BoolParam(GENERATIONS, SMALL_ROWS, SMALL_COLUMNS, SMALL_LEVELS,
    SMALL_POP_SIZE),
00140             BoolParam(GENERATIONS, MEDIUM_ROWS, MEDIUM_COLUMNS, MEDIUM_LEVELS, MEDIUM_POP_SIZE),
00141             BoolParam(GENERATIONS, LARGE_ROWS, LARGE_COLUMNS, LARGE_LEVELS, LARGE_POP_SIZE),
00142             BoolParam(GENERATIONS, SPECIAL_ROWS, SPECIAL_COLUMNS, SPECIAL_LEVELS, SPECIAL_POP_SIZE) };
00143
00144         void test(std::string testName, int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int
    POPULATION_SIZE) {
00145             Timer testTimer("parityTestTimer");
00146
00147             ParityProblem problem(GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE);
00148             problem.problemRunner();
00149
00150             testTimer.endTimer();
00151
00152             saveResults(testName, GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE);
00153         }
00154     public:
00159         SeqParityTester(BoolParam customParams) : Tester((customParams.pop == 0) ? "SeqParityTest" :
    "CustomSeqParityTest") {
00160             if(customParams.pop != 0) {
00161                 for(int i = 0; i < ROUNDS; i++)
00162                     test("CustomSeqParityTest", customParams.gens, customParams.rows,
    customParams.cols, customParams.levels, customParams.pop);
00163                 return;
00164             }
00165
00166             for (int f = 0; f < (sizeof(parityFuncs) / sizeof(*parityFuncs)); f++)
00167                 for (int i = 0; i < ROUNDS; i++)
00168                     test(parityFuncs[f], params[f].gens, params[f].rows, params[f].cols,
    params[f].levels, params[f].pop);
00169         }
00170     };
00171
00175     class ParParityTester : private Tester
00176     {
00177     private:
00178         std::string parityFuncs[4] = { "smallParParityTest", "mediumParParityTest",
    "largeParParityTest", "specialParParityTest" };
00179         BoolParam params[4] = { BoolParam(GENERATIONS, SMALL_ROWS, SMALL_COLUMNS, SMALL_LEVELS,
    SMALL_POP_SIZE),
00180             BoolParam(GENERATIONS, MEDIUM_ROWS, MEDIUM_COLUMNS, MEDIUM_LEVELS, MEDIUM_POP_SIZE),
00181             BoolParam(GENERATIONS, LARGE_ROWS, LARGE_COLUMNS, LARGE_LEVELS, LARGE_POP_SIZE),
00182             BoolParam(GENERATIONS, SPECIAL_ROWS, SPECIAL_COLUMNS, SPECIAL_LEVELS, SPECIAL_POP_SIZE) };
00183
00184         void test(std::string testName, int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int
    POPULATION_SIZE, int THREAD_NUM) {
00185             Timer testTimer("parityTestTimer");
00186
```

```
00187             omp_set_num_threads(THREAD_NUM);
00188
00189             ParityProblem problem(GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE);
00190             problem.problemRunner();
00191
00192             testTimer.endTimer();
00193
00194             saveResults(testName, GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE);
00195         }
00196     public:
00201         ParParityTester(BoolParam customParams) : Tester((customParams.pop == 0) ? "ParParityTest" :
      "CustomParParityTest") {
00202             if(customParams.pop != 0) {
00203                 for (int t = 0; t < threadNums.size(); t++) {
00204                     for(int i = 0; i < ROUNDS; i++)
00205                         test("CustomParParityTest", customParams.gens, customParams.rows,
      customParams.cols, customParams.levels, customParams.pop, threadNums[t]);
00206                     return;
00207                 }
00208             }
00209
00210             for (int f = 0; f < (sizeof(parityFuncs) / sizeof(*parityFuncs)); f++)
00211                 for (int t = 0; t < threadNums.size(); t++)
00212                     for (int i = 0; i < ROUNDS; i++)
00213                         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]);
00214         }
00215     };
00216 }
00217
00218 #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     boost::random::mt19937
      gen(chrono::duration_cast<std::chrono::nanoseconds>(chrono::system_clock::now().time_since_epoch()).count()
      * (omp_get_thread_num() + 1));
00011
00012     for (int i = 0; i < populationSize; i++) {
00013         boost::random::uniform_int_distribution<> operandDis(1, operands);
00014         boost::random::uniform_int_distribution<> connectionDis(0, rows * columns + inputs - 1);
00015         boost::random::uniform_int_distribution<> outputDis(0, rows * columns + inputs - 1);
00016
00017         vector<CGPNode> genes;
00018         vector<CGPOutput> outputGene;
00019
00020         for (int k = 0; k < inputs; k++) {
00021             CGPNode node;
00022             node.used = false;
00023             node.connection1 = -1;
00024             node.connection2 = -1;
00025             node.operand = -1;
00026             genes.push_back(node);
00027         }
00028
00029         for (int j = inputs; j < rows * columns + inputs; j++) {
00030             CGPNode node;
00031             node.used = false;
00032             node.operand = operandDis(gen);
00033             node.connection1 = connectionDis(gen);
00034             node.outValue = NAN;
00035
00036             while (true) {
00037                 if (node.connection1 < inputs)
00038                     break;
00039                 if ((node.connection1 % columns) == (j % columns))
00040                     node.connection1 = connectionDis(gen);
00041                 else if (((node.connection1 - inputs) % columns) > (((j - inputs) % columns) +
      levelsBack))
00042                     node.connection1 = connectionDis(gen);
00043                 else if(genes.size() > node.connection1 && (genes[node.connection1].connection1 == j
      || genes[node.connection1].connection2 == j))
00044                     node.connection1 = connectionDis(gen);
00045                 else
```

```
00046                     break;
00047             }
00048
00049             node.connection2 = (node.operand >= biOperands) ? -1 : connectionDis(gen);
00050
00051             while (true) {
00052                 if (node.connection2 < inputs)
00053                     break;
00054                 if ((node.connection2 % columns) == (j % columns))
00055                     node.connection2 = connectionDis(gen);
00056                 else if (((node.connection2 - inputs) % columns) > (((j - inputs) % columns) +
      levelsBack))
00057                     node.connection2 = connectionDis(gen);
00058                 else if (genes.size() > node.connection2 && (genes[node.connection2].connection1 == j
      || genes[node.connection2].connection2 == j))
00059                     node.connection2 = connectionDis(gen);
00060                 else
00061                     break;
00062             }
00063             genes.push_back(node);
00064         }
00065
00066         for (int k = 0; k < outputs; k++) {
00067             CGPOutput output;
00068
00069             output.connection = outputDis(gen);
00070             outputGene.push_back(output);
00071         }
00072
00073         CGPIndividual individual(genes, outputGene, rows, columns, levelsBack, inputs, outputs);
00074
00075         population[i] = individual;
00076         population[i].resolveLoops();
00077     }
00078
00079     genTime.endTimer();
00080 }
00081
00082 void CGP::goldMutate(CGPIndividual parent, vector<CGPIndividual> &population) {
00083     Timer mutTime("mutatePopulationTimer");
00084
00085     if (!parent.evalDone)
00086         parent.evaluateUsed();
00087     population[0] = parent;
00088
00089     boost::random::mt19937
      gen(chrono::duration_cast<std::chrono::nanoseconds>(chrono::system_clock::now().time_since_epoch()).count()
      * (omp_get_thread_num() + 1));
00090
00091     for (int n = 1; n < populationSize; n++) {
00092         boost::random::uniform_int_distribution<> nodDis(parent.inputs,
      static_cast<int>(parent.genes.size()));
00093         boost::random::uniform_int_distribution<> geneDis(0, 2);
00094         boost::random::uniform_int_distribution<> connectionDis(0,
      static_cast<int>(parent.genes.size()) - 1);
00095         boost::random::uniform_int_distribution<> operandDis(1, operands);
00096         boost::random::uniform_int_distribution<> outputDis(0, parent.outputs - 1);
00097
00098         vector<CGPNode> genes = parent.genes;
00099         vector<CGPOutput> outputGene = parent.outputGene;
00100         bool isActive = false;
00101
00102         while (!isActive) {
00103             int mut = geneDis(gen);
00104             int cell = nodDis(gen);
00105             if (cell == parent.genes.size()) {
00106                 outputGene[outputDis(gen)].connection = connectionDis(gen);
00107                 break;
00108             }
00109             if (mut == 0) {
00110                 genes[cell].operand = operandDis(gen);
00111
00112                 if (genes[cell].operand >= biOperands && genes[cell].connection2 != -1)
00113                     genes[cell].connection2 = -1;
00114                 else if (genes[cell].operand < biOperands && genes[cell].connection2 == -1)
00115                     genes[cell].connection2 = connectionDis(gen);
00116             }
00117             else if (mut == 1)
00118                 genes[cell].connection1 = connectionDis(gen);
00119             else if (mut == 2 && genes[cell].operand >= biOperands)
00120                 continue;
00121             else if (mut == 2)
00122                 genes[cell].connection2 = connectionDis(gen);
00123
00124             while (true) {
00125                 if (genes[cell].connection1 < parent.inputs)
00126                     break;
```

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

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 #include <boost/random.hpp>
00018 
00019 namespace parallel_cgp {
00023     class CGP {
00024     private:
00025         int rows, columns, levelsBack, inputs, outputs, operands, biOperands, populationSize;
00026     public:
00038         CGP(int rows, int columns, int levelsBack, int inputs, int outputs, int operands, int
      biOperands, int populationSize)
00039             : rows(rows), columns(columns), levelsBack(levelsBack), inputs(inputs), outputs(outputs),
00040                 operands(operands), biOperands(biOperands), populationSize(populationSize) {};
00041 
00048         void generatePopulation(std::vector<CGPIndividual> &population);
00049 
00058         void goldMutate(CGPIndividual parent, std::vector<CGPIndividual> &population);
00059     };
00060 }
00061 
00062 #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     boost::random::mt19937
    gen(chrono::duration_cast<std::chrono::nanoseconds>(chrono::system_clock::now().time_since_epoch()).count()
    * (omp_get_thread_num() + 1));
00131
00132     for (int m = 0; m < outputs; m++) {
00133         while (findLoops(outputGene[m].connection)) {
00134             for (int i = 0; i < branches.size(); i++) {
00135                 boost::random::uniform_int_distribution<> connectionDis(0,
    static_cast<int>(genes.size()) - 1);
00136                 int cell1 = branches[i][branches[i].size() - 2];
00137                 int cell2 = branches[i][branches[i].size() - 1];
00138
00139                 if (genes[cell1].connection1 == cell2) {
00140                     genes[cell1].connection1 = connectionDis(gen);
00141
00142                     while (true) {
00143                         if (genes[cell1].connection1 < inputs)
00144                             break;
00145                         if ((genes[cell1].connection1 % columns) == (cell1 % columns))
00146                             genes[cell1].connection1 = connectionDis(gen);
00147                         else if (((genes[cell1].connection1 - inputs) % columns) > (((cell1 - inputs)
    % columns) + levelsBack))
00148                             genes[cell1].connection1 = connectionDis(gen);
00149                         else
00150                             break;
00151                     }
00152                 }
00153                 else if (genes[cell1].connection2 == cell2) {
00154                     genes[cell1].connection2 = connectionDis(gen);
00155
00156                     while (true) {
00157                         if (genes[cell1].connection2 < inputs)
00158                             break;
00159                         if ((genes[cell1].connection2 % columns) == (cell1 % columns))
00160                             genes[cell1].connection2 = connectionDis(gen);
00161                         else if (((genes[cell1].connection2 - inputs) % columns) > (((cell1 - inputs)
    % columns) + levelsBack))
00162                             genes[cell1].connection2 = connectionDis(gen);
00163                         else
00164                             break;
00165                     }
```

```
00166                 }
00167             }
00168         }
00169     }
00170
00171     resLoopTime.endTimer();
00172 }
```

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 <boost/random.hpp>
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         boost::random::mt19937
      gen(chrono::duration_cast<std::chrono::nanoseconds>(chrono::system_clock::now().time_since_epoch()).count()
      * (omp_get_thread_num() + 1));
00130
00131         for (int clan = 0; clan < POPULATION_SIZE; clan++) {
00132
00133             TYPE fit = 0;
00134             problemSimulator(population[clan], fit);
00135
```

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

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[8] = { "smallSimpleSeqFuncTest", "mediumSimpleSeqFuncTest",
      "largeSimpleSeqFuncTest", "specialSimpleSeqFuncTest", "smallComplexSeqFuncTest",
      "mediumComplexSeqFuncTest", "largeComplexSeqFuncTest", "specialComplexSeqFuncTest" };
00036         FuncParam params[8] = { 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, SPECIAL_ROWS, SPECIAL_COLUMNS, SPECIAL_LEVELS, SPECIAL_POP_SIZE,
      -1),
00040             FuncParam(GENERATIONS, SMALL_ROWS, SMALL_COLUMNS, SMALL_LEVELS, SMALL_POP_SIZE, -1),
00041             FuncParam(GENERATIONS, MEDIUM_ROWS, MEDIUM_COLUMNS, MEDIUM_LEVELS, MEDIUM_POP_SIZE, -1),
00042             FuncParam(GENERATIONS, LARGE_ROWS, LARGE_COLUMNS, LARGE_LEVELS, LARGE_POP_SIZE, -1),
00043             FuncParam(GENERATIONS, SPECIAL_ROWS, SPECIAL_COLUMNS, SPECIAL_LEVELS, SPECIAL_POP_SIZE,
      -1) };
00044         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)); } };
00045
00046         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) {
00047             Timer testTimer("funcTestTimer");
00048
00049             FuncProblem problem(GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE, THRESHOLD,
      func);
00050             problem.problemRunner();
00051
00052             testTimer.endTimer();
00053
00054             saveResults(testName, GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE);
00055         }
00056     public:
00061         SeqFuncTester(FuncParam customParams) : Tester((customParams.pop == 0) ? "SeqFuncTest" :
      "CustomSeqFuncTest") {
00062             if(customParams.pop != 0) {
00063                 for(int i = 0; i < ROUNDS; i++)
00064                     test("CustomSeqFuncTest", customParams.gens, customParams.rows, customParams.cols,
      customParams.levels, customParams.pop, customParams.thresh, func[0]);
00065                 return;
00066             }
00067
00068             for (int f = 0; f < (sizeof(funcs) / sizeof(*funcs)); f++) {
00069                 for (int i = 0; i < ROUNDS; i++) {
00070                     if (f < 3)
00071                         test(funcs[f], params[f].gens, params[f].rows, params[f].cols,
      params[f].levels, params[f].pop, params[f].thresh, func[0]);
00072                     else
00073                         test(funcs[f], params[f].gens, params[f].rows, params[f].cols,
      params[f].levels, params[f].pop, params[f].thresh, func[1]);
00074                 }
00075             }
00076         }
00077     };
00078
00082     class ParFuncTester : private Tester
00083     {
```

```
00084     private:
00085         std::string funcs[8] = { "smallSimpleParFuncTest", "mediumSimpleParFuncTest",
      "largeSimpleParFuncTest", "specialSimpleParFuncTest", "smallComplexParFuncTest",
      "mediumComplexParFuncTest", "largeComplexParFuncTest", "specialComplexParFuncTest" };
00086         FuncParam params[8] = { FuncParam(GENERATIONS, SMALL_ROWS, SMALL_COLUMNS, SMALL_LEVELS,
      SMALL_POP_SIZE, -1),
00087             FuncParam(GENERATIONS, MEDIUM_ROWS, MEDIUM_COLUMNS, MEDIUM_LEVELS, MEDIUM_POP_SIZE, -1),
00088             FuncParam(GENERATIONS, LARGE_ROWS, LARGE_COLUMNS, LARGE_LEVELS, LARGE_POP_SIZE, -1),
00089             FuncParam(GENERATIONS, SPECIAL_ROWS, SPECIAL_COLUMNS, SPECIAL_LEVELS, SPECIAL_POP_SIZE,
      -1),
00090             FuncParam(GENERATIONS, SMALL_ROWS, SMALL_COLUMNS, SMALL_LEVELS, SMALL_POP_SIZE, -1),
00091             FuncParam(GENERATIONS, MEDIUM_ROWS, MEDIUM_COLUMNS, MEDIUM_LEVELS, MEDIUM_POP_SIZE, -1),
00092             FuncParam(GENERATIONS, LARGE_ROWS, LARGE_COLUMNS, LARGE_LEVELS, LARGE_POP_SIZE, -1),
00093             FuncParam(GENERATIONS, SPECIAL_ROWS, SPECIAL_COLUMNS, SPECIAL_LEVELS, SPECIAL_POP_SIZE,
      -1) };
00094         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)); } };
00095
00096         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) {
00097             Timer testTimer("funcTestTimer");
00098
00099             omp_set_num_threads(THREAD_NUM);
00100
00101             FuncProblem problem(GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE, THRESHOLD,
      func);
00102             problem.problemRunner();
00103
00104             testTimer.endTimer();
00105
00106             saveResults(testName, GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE);
00107         }
00108     public:
00113         ParFuncTester(FuncParam customParams) : Tester((customParams.pop == 0) ? "ParFuncTest" :
      "CustomParFuncTest") {
00114             if(customParams.pop != 0) {
00115                 for (int t = 0; t < threadNums.size(); t++) {
00116                     for(int i = 0; i < ROUNDS; i++)
00117                         test("CustomParFuncTest", customParams.gens, customParams.rows,
      customParams.cols, customParams.levels, customParams.pop, customParams.thresh, func[0],
      threadNums[t]);
00118                     return;
00119                 }
00120             }
00121
00122             for (int f = 0; f < (sizeof(funcs) / sizeof(*funcs)); f++) {
00123                 for (int t = 0; t < threadNums.size(); t++) {
00124                     for (int i = 0; i < ROUNDS; i++) {
00125                         if (f < 3)
00126                             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]);
00127                         else
00128                             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]);
00129                     }
00130                 }
00131             }
00132         }
00133     };
00134 }
00135
00136 #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 #include <boost/program_options.hpp>
00015
```

```
00016 #define PARAM_COUNT 5
00017
00018 using namespace std;
00019 using namespace parallel_cgp;
00020 namespace po = boost::program_options;
00021
00022 #if (defined(_OPENMP) && (defined(OMPCGP) || defined(OMPSIM) || defined(OMPRUN)))
00023 #define BoolTester ParBoolTester
00024 #define ParityTester ParParityTester
00025 #define FuncTester ParFuncTester
00026 #define ADTester ParADTester
00027 #define WaitTester ParWaitTester
00028 #define PARALLEL_TESTER 1
00029 #else
00030 #define BoolTester SeqBoolTester
00031 #define ParityTester SeqParityTester
00032 #define FuncTester SeqFuncTester
00033 #define ADTester SeqADTester
00034 #define WaitTester SeqWaitTester
00035 #define PARALLEL_TESTER 0
00036 #endif
00037
00038 int main(int ac, char** av) {
00039     try {
00040         int threads = 1;
00041
00042         po::options_description desc("Allowed options");
00043         desc.add_options()
00044             ("help,h", "produce help message")
00045             ("test,t", "enable testing")
00046             ("bool,b", "enable bool problem")
00047             ("parity,p", "enable parity problem")
00048             ("func,f", "enable func problem")
00049             ("acey,a", "enable acey problem")
00050             ("wait,w", "enable wait problem")
00051             ("custom,c", po::value<std::vector<int»()->multitoken(), "custom test values (number of
      generations, rows, columns, levels, population size)")
00052             ("threads,T", po::value<int>(), "number of threads to use in parallel version")
00053             ("version,v", "print version information")
00054             ;
00055
00056         po::variables_map vm;
00057         po::store(po::parse_command_line(ac, av, desc), vm);
00058         po::notify(vm);
00059
00060         vector<int> params(PARAM_COUNT, 0);
00061         if (!vm["custom"].empty() && !((params = vm["custom"].as<vector<int> >()).size() ==
      PARAM_COUNT))
00062             throw invalid_argument("Not the right amount of custom parameters");
00063
00064         if (vm.count("help")) {
00065             cout « desc « endl;
00066             return 1;
00067         }
00068
00069         if (vm.count("version")) {
00070             cout « "ParallelCGP version 1.0 Sequential" « endl;
00071             cout « "Author: Andrija Macek" « endl;
00072             return 2;
00073         }
00074
00075         if (vm.count("threads")) {
00076             if (!PARALLEL_TESTER)
00077                 throw invalid_argument("Threads are not supported in the sequential version of the
      program");
00078             threads = vm["threads"].as<int>();
00079             if (threads < 1)
00080                 throw invalid_argument("Number of threads must be greater than 0");
00081             Tester::threadNums.clear();
00082             Tester::threadNums.push_back(threads);
00083         }
00084
00085         Problem* problem = nullptr;
00086
00087         if (vm.count("bool"))
00088             if (vm.count("test"))
00089                 BoolTester boolTest = BoolTester(BoolParam(params[0], params[1], params[2], params[3],
      params[4]));
00090             else {
00091                 omp_set_num_threads(threads);
00092                 problem = new BoolProblem;
00093                 problem->printGens = true;
00094                 problem->problemRunner();
00095             }
00096         if (vm.count("parity"))
00097             if (vm.count("test"))
00098                 ParityTester parityTest = ParityTester(BoolParam(params[0], params[1], params[2],
```

```
      params[3], params[4]));
00099             else {
00100                 omp_set_num_threads(threads);
00101                 problem = new ParityProblem;
00102                 problem->printGens = true;
00103                 problem->problemRunner();
00104             }
00105         if (vm.count("func"))
00106             if (vm.count("test"))
00107                 FuncTester funcTest = FuncTester(FuncParam(params[0], params[1], params[2], params[3],
      params[4], -1));
00108             else {
00109                 omp_set_num_threads(threads);
00110                 problem = new FuncProblem;
00111                 problem->printGens = true;
00112                 problem->problemRunner();
00113             }
00114         if (vm.count("acey"))
00115             if (vm.count("test"))
00116                 ADTester adTest = ADTester(ADParam(params[0], params[1], params[2], params[3],
      params[4]));
00117             else {
00118                 omp_set_num_threads(threads);
00119                 problem = new ADProblem;
00120                 problem->printGens = true;
00121                 problem->problemRunner();
00122             }
00123         if (vm.count("wait"))
00124             if (vm.count("test"))
00125                 WaitTester waitTest = WaitTester(WaitParam(params[0], params[1], params[2], params[3],
      params[4], 1));
00126             else {
00127                 omp_set_num_threads(threads);
00128                 problem = new WaitProblem;
00129                 problem->printGens = true;
00130                 problem->problemRunner();
00131             }
00132
00133         if (vm.count("test"))
00134             delete(problem);
00135     }
00136     catch(exception& e) {
00137         cerr « "error: " « e.what() « endl;
00138         return 1;
00139     }
00140     catch(...) {
00141         cerr « "Exception of unknown type!" « endl;
00142     }
00143
00144     return 0;
00145 }
```

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 #include <chrono>
00012 #include <boost/random.hpp>
00013
00014 namespace parallel_cgp {
00015     class Problem {
00016     private:
00022         virtual void problemSimulator(parallel_cgp::CGPIndividual &individual, TYPE &fit) {}
00027         virtual std::string evalFunction(int CGPNodeNum) = 0;
00028     public:
00032         virtual ~Problem() = default;
00036         CGPIndividual *bestI;
00037
00041         bool printGens = false;
00042
00049         int NUM_OPERANDS = 9;
00051         int BI_OPERANDS = 5;
00053         int GENERATIONS = 5000;
00055         int ROWS = 8;
00057         int COLUMNS = 8;
```

```
00059         int LEVELS_BACK = 3;
00061         int INPUTS = 6;
00063         int OUTPUTS = 1;
00065         int POPULATION_SIZE = 20;
00067
00074         virtual TYPE computeNode(int operand, TYPE value1, TYPE value2) {
00075             switch (operand) {
00076             case 1:
00077                 return value1 + value2;
00078             case 2:
00079                 return value1 - value2;
00080             case 3:
00081                 return value1 * value2;
00082             case 4:
00083                 return (value2 == 0) ? 0 : value1 / value2;
00084             case 5:
00085                 return sin(value1);
00086             case 6:
00087                 return cos(value1);
00088             case 7:
00089                 return value1 > 0 ? sqrt(value1) : value1;
00090             case 8:
00091                 return pow(value1, 2);
00092             case 9:
00093                 return pow(2, value1);
00094             default:
00095                 return 0;
00096             }
00097         }
00101         virtual TYPE fitness(TYPE fit) { return fit; }
00102
00106         virtual void problemRunner() = 0;
00110         virtual void printFunction() = 0;
00111     };
00112 }
00113
00114 #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 #include <vector>
00010
00011 #ifndef _OPENMP
00012 #define omp_set_num_threads(threads) 0
00013 #endif
00014
00015 namespace parallel_cgp {
00019     class Tester
00020     {
00021     private:
00022         std::string testerName;
00023         std::string filename;
00024     public:
00029         inline static std::string VERSION_NAME = "";
00030
00036         const static int ROUNDS = 10;
00038         const static int GENERATIONS = 1000;
00040         const static int SMALL_ROWS = 4;
00042         const static int MEDIUM_ROWS = 8;
00044         const static int LARGE_ROWS = 10;
00046         const static int SPECIAL_ROWS = 1;
00048         const static int SMALL_COLUMNS = 4;
00050         const static int MEDIUM_COLUMNS = 8;
00052         const static int LARGE_COLUMNS = 10;
00054         const static int SPECIAL_COLUMNS = 100;
00056         const static int SMALL_LEVELS = 0;
00058         const static int MEDIUM_LEVELS = 1;
00060         const static int LARGE_LEVELS = 3;
00062         const static int SPECIAL_LEVELS = 10;
00064         const static int SMALL_POP_SIZE = 5;
00066         const static int MEDIUM_POP_SIZE = 8;
00068         const static int LARGE_POP_SIZE = 16;
00070         const static int SPECIAL_POP_SIZE = 5;
00072         inline static std::vector<int> threadNums = { 1, 2, 4, 8, 16 };
00074
```

```
00079         Tester(std::string testerName) : testerName(testerName), filename(testerName) {
00080             filename.append(VERSION_NAME);
00081             filename.append(".csv");
00082             std::ofstream myFile;
00083             myFile.open(filename);
00084             myFile.close();
00085         }
00086 
00091         void saveResults(std::string testName, int gens, int rows, int cols, int levels, int pop) {
00092             Timer::saveTimes(filename, testName, gens, rows, cols, levels, pop);
00093 
00094             std::cout « "--------------------------------------" « std::endl;
00095             std::cout « "TEST NAME: " « testName « std::endl;
00096             std::cout « "--------------------------------------" « std::endl;
00097             std::cout « "GENS: " « gens « ", ROWS: " « rows « ", COLUMNS: " « cols
00098                 « ", LEVELS BACK: " « levels « ", POP SIZE: " « pop « std::endl;
00099             std::cout « "--------------------------------------" « std::endl;
00100             Timer::clearTimes();
00101         }
00102     };
00103 }
00104 
00105 #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 « "ns";
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         boost::random::mt19937
      gen(chrono::duration_cast<std::chrono::nanoseconds>(chrono::system_clock::now().time_since_epoch()).count()
      * (omp_get_thread_num() + 1));
00061
00062         for (int clan = 0; clan < POPULATION_SIZE; clan++) {
```

```
00063
00064             TYPE fit = generacija;
00065             problemSimulator(population[clan], fit);
00066
00067             if (fit > bestFit) {
00068                 bestFit = fit;
00069                 bestInds.clear();
00070                 bestInds.push_back(clan);
00071             }
00072             else if (fit == bestFit)
00073                 bestInds.push_back(clan);
00074         }
00075
00076         if (bestInds.size() > 1)
00077             bestInds.erase(bestInds.begin());
00078         if (bestInds.size() == 0)
00079             bestInds.push_back(0);
00080
00081         boost::random::uniform_int_distribution<> bestDis(0, static_cast<int>(bestInds.size()) - 1);
00082
00083         bestInd = bestInds[bestDis(gen)];
00084
00085         if(printGens)
00086             cout « "Gen: " « generacija « "; Fitness: " « bestFit « "; Indeks: " « bestInd « endl;
00087
00088         if (generacija != GENERATIONS - 1)
00089             cgp.goldMutate(population[bestInd], population);
00090     }
00091
00092     bestI = &population[bestInd];
00093
00094     isSimulated = true;
00095
00096     printFunction();
00097
00098     probRunTime.endTimer();
00099 }
```

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[4] = { "smallSeqWaitTest", "mediumSeqWaitTest", "largeSeqWaitTest",
      "specialSeqWaitTest" };
00036         WaitParam params[4] = { 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             WaitParam(GENERATIONS, SPECIAL_ROWS, SPECIAL_COLUMNS, SPECIAL_LEVELS, SPECIAL_POP_SIZE, 1)
      };
00040
00041         void test(std::string testName, int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int
      POPULATION_SIZE, int WAIT_TIME) {
00042             Timer testTimer("waitTestTimer");
00043
00044             WaitProblem problem(GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE, WAIT_TIME);
00045             problem.problemRunner();
00046
00047             testTimer.endTimer();
00048
00049             saveResults(testName, GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE);
00050         }
00051     public:
00056         SeqWaitTester(WaitParam customParams) : Tester((customParams.pop == 0) ? "SeqWaitTest" :
      "CustomSeqWaitTest") {
00057             if(customParams.pop != 0) {
00058                 for(int i = 0; i < ROUNDS; i++)
00059                     test("CustomSeqWaitTest", customParams.gens, customParams.rows, customParams.cols,
      customParams.levels, customParams.pop, customParams.time);
00060                 return;
00061             }
00062
00063             for (int f = 0; f < (sizeof(funcs) / sizeof(*funcs)); f++) {
00064                 for (int i = 0; i < ROUNDS; i++) {
00065                     test(funcs[f], params[f].gens, params[f].rows, params[f].cols, params[f].levels,
      params[f].pop, params[f].time);
00066                 }
00067             }
00068         }
00069     };
00070
00074     class ParWaitTester : private Tester
00075     {
00076     private:
00077         std::string funcs[4] = { "smallParWaitTest", "mediumParWaitTest", "largeParWaitTest",
      "specialParWaitTest" };
00078         WaitParam params[4] = { WaitParam(GENERATIONS, SMALL_ROWS, SMALL_COLUMNS, SMALL_LEVELS,
      SMALL_POP_SIZE, 1),
00079             WaitParam(GENERATIONS, MEDIUM_ROWS, MEDIUM_COLUMNS, MEDIUM_LEVELS, MEDIUM_POP_SIZE, 1),
00080             WaitParam(GENERATIONS, LARGE_ROWS, LARGE_COLUMNS, LARGE_LEVELS, LARGE_POP_SIZE, 1),
00081             WaitParam(GENERATIONS, SPECIAL_ROWS, SPECIAL_COLUMNS, SPECIAL_LEVELS, SPECIAL_POP_SIZE, 1)
      };
00082
00083         void test(std::string testName, int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int
      POPULATION_SIZE, int WAIT_TIME, int THREAD_NUM) {
00084             Timer testTimer("waitTestTimer");
00085
```

```
00086             omp_set_num_threads(THREAD_NUM);
00087
00088             WaitProblem problem(GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE, WAIT_TIME);
00089             problem.problemRunner();
00090
00091             testTimer.endTimer();
00092
00093             saveResults(testName, GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE);
00094         }
00095     public:
00100         ParWaitTester(WaitParam customParams) : Tester((customParams.pop == 0) ? "ParWaitTest" :
      "CustomParWaitTest") {
00101             if(customParams.pop != 0) {
00102                 for (int t = 0; t < threadNums.size(); t++) {
00103                     for(int i = 0; i < ROUNDS; i++)
00104                         test("CustomParWaitTest", customParams.gens, customParams.rows,
      customParams.cols, customParams.levels, customParams.pop, customParams.time, threadNums[t]);
00105                     return;
00106                 }
00107             }
00108
00109             for (int f = 0; f < (sizeof(funcs) / sizeof(*funcs)); f++) {
00110                 for (int t = 0; t < threadNums.size(); t++) {
00111                     for (int i = 0; i < ROUNDS; i++) {
00112                         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]);
00113                     }
00114                 }
00115             }
00116         }
00117     };
00118 }
00119
00120 #endif
```

Index