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.

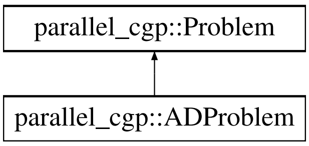
The documentation for this struct was generated from the following file:

- 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.

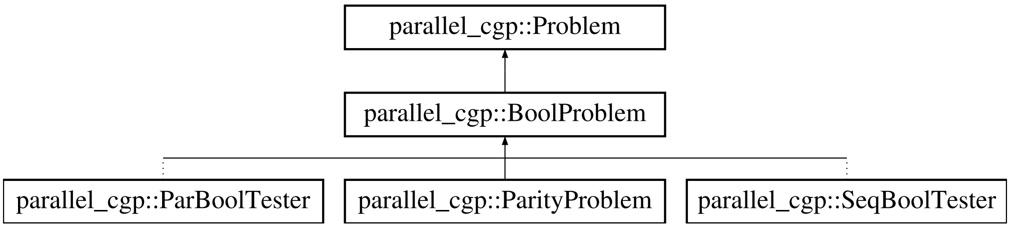
The documentation for this struct was generated from the following file:

- boolProblem/BoolTester.hpp

5.4 parallel_cgp::BoolProblem Class Reference

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

Inheritance diagram for parallel_cgp::BoolProblem:


parallel_cgp::Problem
parallel_cgp::BoolProblem
parallel_cgp::ParBoolTester
parallel_cgp::ParityProblem
parallel_cgp::SeqBoolTester

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.

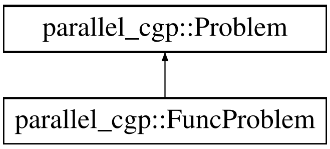
The documentation for this struct was generated from the following file:

- 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.

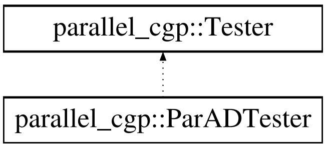
The documentation for this class was generated from the following files:

- 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.

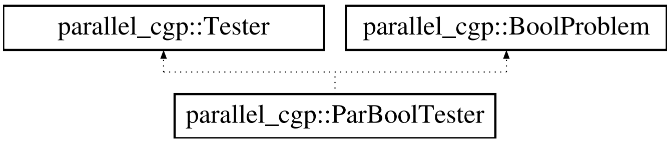
The documentation for this class was generated from the following file:

- 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.

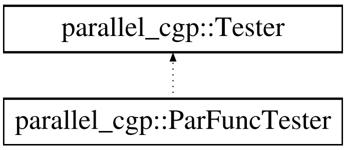
The documentation for this class was generated from the following file:

- 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.

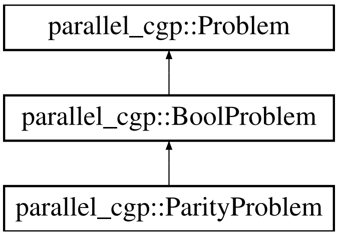
The documentation for this class was generated from the following file:

- 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.

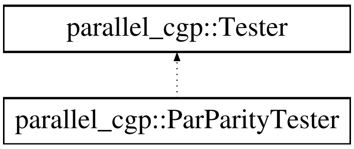
The documentation for this class was generated from the following file:

- 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.

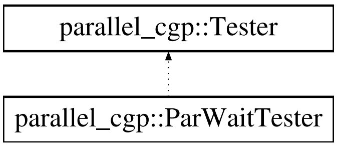
The documentation for this class was generated from the following file:

- 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.

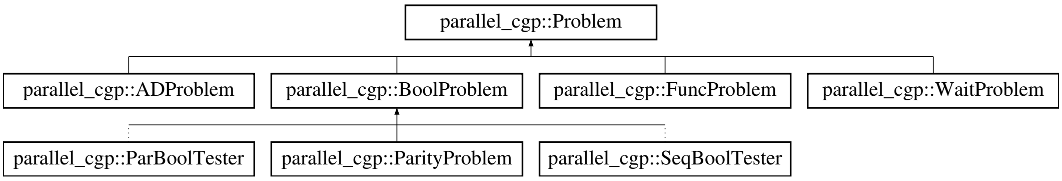
Definition at line 100 of file WaitTester.hpp.

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:


parallel_cgp::Problem
parallel_cgp::ADProblem
parallel_cgp::BoolProblem
parallel_cgp::FuncProblem
parallel_cgp::WaitProblem
parallel_cgp::ParBoolTester
parallel_cgp::ParityProblem
parallel_cgp::SeqBoolTester

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.

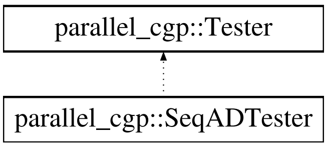
The documentation for this class was generated from the following file:

- 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.

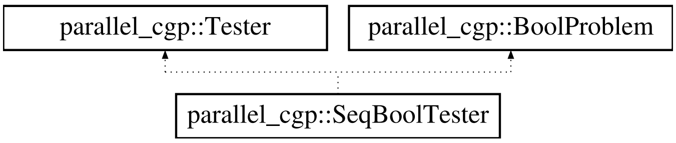
The documentation for this class was generated from the following file:

- 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.

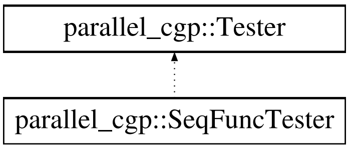
The documentation for this class was generated from the following file:

- 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.

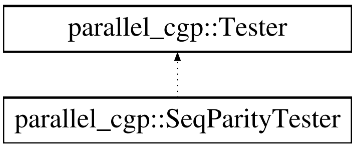
The documentation for this class was generated from the following file:

- 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.

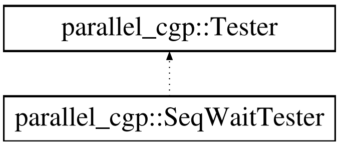
The documentation for this class was generated from the following file:

- 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.

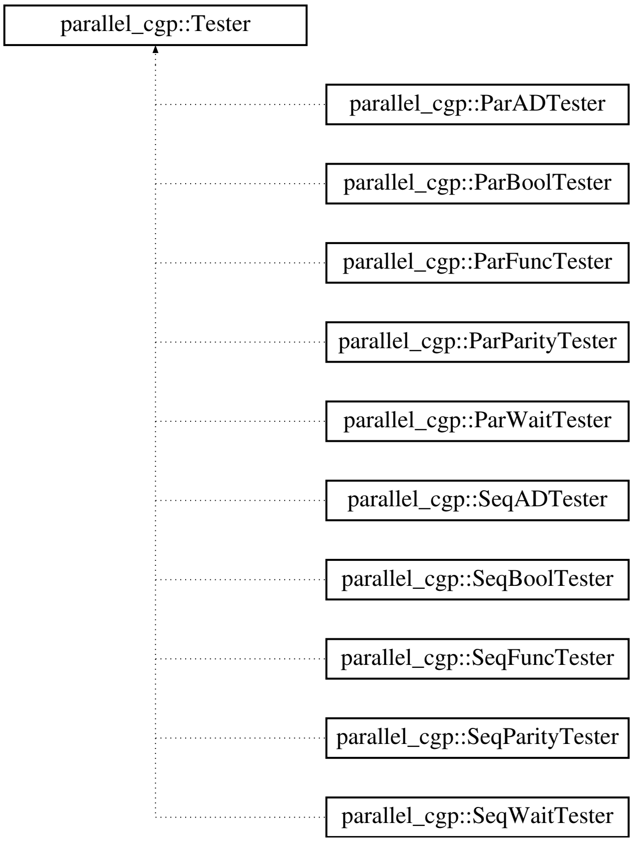
The documentation for this class was generated from the following file:

- waitProblem/WaitTester.hpp

5.23 parallel_cgp::Tester Class Reference

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

Inheritance diagram for parallel_cgp::Tester:


parallel_cgp::Tester
parallel_cgp::ParADTester
parallel_cgp::ParBoolTester
parallel_cgp::ParFuncTester
parallel_cgp::ParParityTester
parallel_cgp::ParWaitTester
parallel_cgp::SeqADTester
parallel_cgp::SeqBoolTester
parallel_cgp::SeqFuncTester
parallel_cgp::SeqParityTester
parallel_cgp::SeqWaitTester

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.

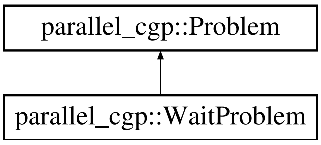
The documentation for this struct was generated from the following file:

- 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

```
#include "ADProblem.hpp"

using namespace std;
using namespace parallel_cgp;

TYPE ADProblem::computeNode(int operand, TYPE value1, TYPE value2) {
    switch (operand) {
    case 1:
        return value1 + value2;
    case 2:
        return value1 - value2;
    case 3:
        return value1 * value2;
    case 4:
        return -value1;
    default:
        return 0;
    }
}

double ADProblem::fitness(TYPE cash, TYPE maxCash, double avgCash) {
    double score = avgCash;

    if (maxCash >= STARTING_CASH * 2)
        score += 50;
    if (cash <= 0)
        score -= 100;
    if (maxCash == MAX_CASH)
        score += 150;

    return score;
}

void ADProblem::printFunction() {
    if (isSimulated)
        cout « "Funkcija: " « evalFunction(bestI->outputGene[0].connection) « endl;
    else
        cout « "Problem nije simuliran." « endl;
}

string ADProblem::evalFunction(int CGPNodeNum) {
    ostringstream oss;

    if (CGPNodeNum < INPUTS) {
        switch (CGPNodeNum) {
        case 0:
            oss « "card1";
            return oss.str();
        case 1:
            oss « "card2";
            return oss.str();
        }
    }

    switch (bestI->genes[CGPNodeNum].operand) {
    case 1:
        oss « "(" « evalFunction(bestI->genes[CGPNodeNum].connection1) « " + " «
evalFunction(bestI->genes[CGPNodeNum].connection2) « ")";
```

```
        return oss.str();
    case 2:
        oss « "(" « evalFunction(bestI->genes[CGPNodeNum].connection1) « " - " « evalFunction(bestI->genes[CGPNodeNum].connection2) « ")";
        return oss.str();
    case 3:
        oss « "(" « evalFunction(bestI->genes[CGPNodeNum].connection1) « " * " « evalFunction(bestI->genes[CGPNodeNum].connection2) « ")";
        return oss.str();
    case 4:
        oss « "-" « evalFunction(bestI->genes[CGPNodeNum].connection1);
        return oss.str();
    }

    return "";
}

void ADProblem::problemSimulator(CGPIndividual& individual, double& fit) {
    Timer probSimTime("problemSimulatorTimer");

    function<double(int op, double v1, double v2)> compNode =
        [&](int op, double v1, double v2) { return computeNode(op, static_cast<TYPE>(v1), static_cast<TYPE>(v2)); };

    int card, win;
    int cash = STARTING_CASH, maxCash = STARTING_CASH;
    double avgCash = 0;

    for (int i = 0; i < CARD_SETS; i++) {
        card = static_cast<int>(sets[i].back());

        if (card > sets[i].at(0) && card < sets[i].at(1))
            win = 1;
        else if (card == sets[i].at(0) || card == sets[i].at(1))
            win = -1;
        else
            win = 0;

        individual.evaluateValue(sets[i], compNode);

        if (individual.outputGene[0].value > 1) {
            if (win == 1)
                cash += 10;
            else if (win == 0)
                cash -= 10;
            else if (win == -1)
                cash -= 20;
        }

        if (cash > maxCash)
            maxCash = cash;

        avgCash += cash;
    }

    avgCash /= static_cast<double>(CARD_SETS);
    fit = fitness(cash, maxCash, avgCash);

    probSimTime.endTimer();
}

void ADProblem::problemRunner() {
    Timer probRunTime("problemRunnerTimer");

    CGP cgp(ROWS, COLUMNS, LEVELS_BACK, INPUTS, OUTPUTS, NUM_OPERANDS, BI_OPERANDS, POPULATION_SIZE);

    vector<CGPIndividual> population(POPULATION_SIZE);
    int bestInd = 0, generacija = 0;

    cgp.generatePopulation(population);

    boost::random::mt19937 gen(chrono::duration_cast<std::chrono::nanoseconds>(chrono::system_clock::now().time_since_epoch()).count() * (omp_get_thread_num() + 1));

    boost::random::uniform_int_distribution<> cardDis(1, 13);

    for (int j = 0; j < CARD_SETS; j++) {
        vector<double> set;
        for (int i = 0; i < 3; i++)
            set.push_back(static_cast<double>(cardDis(gen)));

        double card = set.back();
        set.pop_back();
        sort(set.begin(), set.end());
        set.push_back(card);

        sets.push_back(set);
    }

    for (generacija = 0; generacija < GENERATIONS; generacija++) {
        double bestFit = DBL_MIN;
        bestInd = 0;
        vector<int> bestInds;

        for (int clan = 0; clan < POPULATION_SIZE; clan++) {

            double fit = 0;
            problemSimulator(population[clan], fit);

            if (fit > bestFit) {
                bestFit = fit;
                bestInds.clear();
                bestInds.push_back(clan);
            }
            else if (fit == bestFit)
                bestInds.push_back(clan);
        }

        if (bestInds.size() > 1)
            bestInds.erase(bestInds.begin());
        if (bestInds.size() == 0)
            bestInds.push_back(0);

        boost::random::uniform_int_distribution<> bestDis(0, static_cast<int>(bestInds.size() - 1));

        bestInd = bestInds[bestDis(gen)];

        if(printGens)
            cout « "Gen: " « generacija « "; Fitness: " « bestFit « "; Indeks: " « bestInd « endl;

        if (bestFit >= THRESHOLD)
            break;
        if (generacija != GENERATIONS - 1)
            cgp.goldMutate(population[bestInd], population);
    }

    bestI = &population[bestInd];

    isSimulated = true;

    printFunction();

    probRunTime.endTimer();

    playGame();
}

void ADProblem::playGame() {
    function<double(int op, double v1, double v2)> compNode =
        [&](int op, double v1, double v2) { return computeNode(op, static_cast<TYPE>(v1), static_cast<TYPE>(v2)); };

    boost::random::mt19937 gen(chrono::duration_cast<std::chrono::nanoseconds>(chrono::system_clock::now().time_since_epoch()).count() * (omp_get_thread_num() + 1));

    boost::random::uniform_int_distribution<> cardDis(1, 13);

    int steps = 0;
    int cash = STARTING_CASH, maxCash = STARTING_CASH;

    while (cash && steps < 100 && cash < MAX_CASH) {
        vector<double> input;
        int card, win;
        for (int i = 0; i < 3; i++)
            input.push_back(static_cast<TYPE>(cardDis(gen)));

        card = card = static_cast<int>(input.back());
        input.pop_back();

        sort(input.begin(), input.end());

        if (card > input.at(0) && card < input.at(1))
            win = 1;
        else if (card == input.at(0) || card == input.at(1))
            win = -1;
        else
            win = 0;

        bestI->evaluateValue(input, compNode);

        cout « "Cash: " « cash « "; Cards: " « input[0] « ", " « input[1] « "; Bet: " « ((bestI->outputGene[0].value > 1) ? "YES" : "NO")
```

```
            « "; Third card: " « card « ((win == 1) ? " | WIN!" : " | LOST!") « endl;

        if (bestI->outputGene[0].value > 1) {
            if (win == 1)
                cash += 10;
            else if (win == 0)
                cash -= 10;
            else if (win == -1)
                cash -= 20;
        }

        if (cash > maxCash)
            maxCash = cash;

        steps++;
    }
}
```

6.2 ADProblem.hpp

```
#ifndef ADPROBLEM_HPP
#define ADPROBLEM_HPP

#include "../Problem.hpp"
#include "../cgp/CGP.hpp"

#undef TYPE
#define TYPE int

namespace parallel_cgp {
    class ADProblem : public Problem {
    private:
        const static int NUM_OPERANDS = 4;
        const static int BI_OPERANDS = 4;
        const static int INPUTS = 2;
        const static int OUTPUTS = 1;
        const static int MAX_CASH = 1000;
        const static int STARTING_CASH = 100;
        const static int CARD_SETS = 500;
        const static int THRESHOLD = STARTING_CASH * 3;

        int GENERATIONS = 200;
        int ROWS = 8;
        int COLUMNS = 8;
        int LEVELS_BACK = 3;
        int POPULATION_SIZE = 15;

        std::vector<std::vector<double» sets;

        bool isSimulated = false;

        TYPE computeNode(int operand, TYPE value1, TYPE value2);
        double fitness(TYPE cash, TYPE maxCash, double avgCash);
        void problemSimulator(parallel_cgp::CGPIndividual& individual, double& fit) override;
        std::string evalFunction(int CGPNodeNum) override;
    public:
        ADProblem() {};
        ADProblem(int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int POPULATION_SIZE)
            : GENERATIONS(GENERATIONS), ROWS(ROWS), COLUMNS(COLUMNS), LEVELS_BACK(LEVELS_BACK),
POPULATION_SIZE(POPULATION_SIZE) {};

        void problemRunner() override;
        void printFunction() override;
        void playGame();
    };
}

#endif
```

6.3 ADTester.hpp

```
#ifndef ADTESTER_HPP
#define ADTESTER_HPP

#include "../Tester.hpp"
#include "../Timer.hpp"
#include "ADProblem.hpp"

```

```
namespace parallel_cgp {
    struct ADParam {
        ADParam() {}
        ADParam(int gens, int rows, int cols, int levels, int pop) : gens(gens), rows(rows), cols(cols), levels(levels), pop(pop) {}
        int gens;
        int rows;
        int cols;
        int levels;
        int pop;
    };

    class SeqADTester : private Tester
    {
    private:
        std::string funcs[4] = { "smallSeqADTest", "mediumSeqADTest", "largeSeqADTest", "specialSeqADTest" };
        ADParam params[4] = { ADParam(GENERATIONS, SMALL_ROWS, SMALL_COLUMNS, SMALL_LEVELS, SMALL_POP_SIZE),
            ADParam(GENERATIONS, MEDIUM_ROWS, MEDIUM_COLUMNS, MEDIUM_LEVELS, MEDIUM_POP_SIZE),
            ADParam(GENERATIONS, LARGE_ROWS, LARGE_COLUMNS, LARGE_LEVELS, LARGE_POP_SIZE),
            ADParam(GENERATIONS, SPECIAL_ROWS, SPECIAL_COLUMNS, SPECIAL_LEVELS, SPECIAL_POP_SIZE) };

        void test(std::string testName, int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int POPULATION_SIZE) {
            Timer testTimer("adTestTimer");

            ADProblem problem(GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE);
            problem.problemRunner();

            testTimer.endTimer();

            saveResults(testName, GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE);
        }
    public:
        SeqADTester(ADParam customParams) : Tester((customParams.pop == 0) ? "SeqADTest" : "CustomSeqADTest") {
            if(customParams.pop != 0) {
                for(int i = 0; i < ROUNDS; i++)
                    test("CustomSeqADTest", customParams.gens, customParams.rows, customParams.cols, customParams.levels, customParams.pop);
                return;
            }

            for (int f = 0; f < (sizeof(funcs) / sizeof(*funcs)); f++) {
                for (int i = 0; i < ROUNDS; i++) {
                    test(funcs[f], params[f].gens, params[f].rows, params[f].cols, params[f].levels, params[f].pop);
                }
            }
        }
    };

    class ParADTester : private Tester
    {
    private:
        std::string funcs[4] = { "smallParADTest", "mediumParADTest", "largeParADTest", "specialParADTest" };
        ADParam params[4] = { ADParam(GENERATIONS, SMALL_ROWS, SMALL_COLUMNS, SMALL_LEVELS, SMALL_POP_SIZE),
            ADParam(GENERATIONS, MEDIUM_ROWS, MEDIUM_COLUMNS, MEDIUM_LEVELS, MEDIUM_POP_SIZE),
            ADParam(GENERATIONS, LARGE_ROWS, LARGE_COLUMNS, LARGE_LEVELS, LARGE_POP_SIZE),
            ADParam(GENERATIONS, SPECIAL_ROWS, SPECIAL_COLUMNS, SPECIAL_LEVELS, SPECIAL_POP_SIZE) };

        void test(std::string testName, int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int POPULATION_SIZE, int THREAD_NUM) {
            Timer testTimer("adTestTimer");

            omp_set_num_threads(THREAD_NUM);

            ADProblem problem(GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE);
            problem.problemRunner();

            testTimer.endTimer();

            saveResults(testName, GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE);
        }
    public:
        ParADTester(ADParam customParams) : Tester((customParams.pop == 0) ? "ParADTest" : "CustomParADTest") {
            if(customParams.pop != 0) {
                for (int t = 0; t < threadNums.size(); t++) {
                    for(int i = 0; i < ROUNDS; i++)
                        test("CustomParADTest", customParams.gens, customParams.rows, customParams.cols, customParams.levels, customParams.pop, threadNums[t]);
                    return;
                }
            }

            for (int f = 0; f < (sizeof(funcs) / sizeof(*funcs)); f++) {
                for (int t = 0; t < threadNums.size(); t++) {
                    for (int i = 0; i < ROUNDS; i++) {
                        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]);
                    }
                }
            }
        }
    };
}

#endif
```

6.4 BoolProblem.cpp

```
#include "BoolProblem.hpp"

using namespace std;
using namespace parallel_cgp;

TYPE BoolProblem::computeNode(int operand, TYPE value1, TYPE value2) {
    switch (operand) {
    case 1:
        return value1 | value2;
    case 2:
        return value1 & value2;
    case 3:
        return value1 ^ value2;
    case 4:
        return ~value1;
    default:
        return 0;
    }
}

TYPE BoolProblem::fitness(bitset<INPUTS> in, TYPE res) {
    if (useFunc)
        return boolFunc(in) == res;

    return parityFunc(in) == res;
}

void BoolProblem::printFunction() {
    if (isSimulated)
        cout « "Funkcija: " « evalFunction(bestI->outputGene[0].connection) « endl;
    else
        cout « "Problem nije simuliran." « endl;
}

string BoolProblem::evalFunction(int CGPNodeNum) {
    ostringstream oss;

    if (CGPNodeNum < INPUTS) {
        oss « "bit[" « CGPNodeNum « "]";
        return oss.str();
    }

    switch (bestI->genes[CGPNodeNum].operand) {
    case 1:
        oss « "(" « evalFunction(bestI->genes[CGPNodeNum].connection1) « " | " « evalFunction(bestI->genes[CGPNodeNum].connection2) « ")";
        return oss.str();
    case 2:
        oss « "(" « evalFunction(bestI->genes[CGPNodeNum].connection1) « " & " « evalFunction(bestI->genes[CGPNodeNum].connection2) « ")";
        return oss.str();
    case 3:
        oss « "(" « evalFunction(bestI->genes[CGPNodeNum].connection1) « " ^ " « evalFunction(bestI->genes[CGPNodeNum].connection2) « ")";
        return oss.str();
    case 4:
        oss « "~" « evalFunction(bestI->genes[CGPNodeNum].connection1);
        return oss.str();
    }

    return "";
}

void BoolProblem::problemSimulator(CGPIndividual& individual, TYPE &fit) {
    Timer probSimTime("problemSimulatorTimer");

    function<double(int op, double v1, double v2)> compNode =
        [&](int op, double v1, double v2) { return computeNode(op, static_cast<TYPE>(v1), static_cast<TYPE>(v2)); };

    for (int perm = 0; perm < static_cast<int>(pow(2, INPUTS)); ++perm) {
        bitset<INPUTS> bits(perm);
        vector<double> input;

        for (int i = 0; i < bits.size(); ++i)
            input.push_back(static_cast<double>(bits[i]));

        individual.evaluateValue(input, compNode);
        fit += fitness(bits, static_cast<int>(individual.outputGene[0].value));
    }

    probSimTime.endTimer();
}

void BoolProblem::problemRunner() {
    Timer probRunTime("problemRunnerTimer");

    CGP cgp(ROWS, COLUMNS, LEVELS_BACK, INPUTS, OUTPUTS, NUM_OPERANDS, BI_OPERANDS, POPULATION_SIZE);

    vector<CGPIndividual> population(POPULATION_SIZE);
    int bestInd = 0, generacija = 0;

    cgp.generatePopulation(population);

    for (generacija = 0; generacija < GENERATIONS; generacija++) {
        TYPE bestFit = INT_MIN;
        bestInd = 0;
        vector<int> bestInds;
        boost::random::mt19937 gen(chrono::duration_cast<std::chrono::nanoseconds>(chrono::system_clock::now().time_since_epoch()).count() * (omp_get_thread_num() + 1));

        for (int clan = 0; clan < POPULATION_SIZE; clan++) {

            TYPE fit = 0;
            problemSimulator(population[clan], fit);

            if (fit > bestFit) {
                bestFit = fit;
                bestInds.clear();
                bestInds.push_back(clan);
            }
            else if (fit == bestFit)
                bestInds.push_back(clan);
        }

        if (bestInds.size() > 1)
            bestInds.erase(bestInds.begin());
        if (bestInds.size() == 0)
            bestInds.push_back(0);

        boost::random::uniform_int_distribution<> bestDis(0, static_cast<int>(bestInds.size() - 1));

        bestInd = bestInds[bestDis(gen)];

        if (printGens)
            cout « "Gen: " « generacija « "; Fitness: " « bestFit « "; Indeks: " « bestInd « endl;

        if (bestFit == pow(2, INPUTS))
            break;
        if (generacija != GENERATIONS - 1)
            cgp.goldMutate(population[bestInd], population);
    }

    bestI = &population[bestInd];

    isSimulated = true;

    printFunction();

    probRunTime.endTimer();
}
```

6.5 BoolProblem.hpp

```
#ifndef BOOLPROBLEM_HPP
#define BOOLPROBLEM_HPP

```

```
#include "../Problem.hpp"
#include "../cgp/CGP.hpp"
#include <bitset>

#undef TYPE
#define TYPE int

namespace parallel_cgp {
    class BoolProblem : public Problem {
    protected:
        const static int NUM_OPERANDS = 4;
        const static int BI_OPERANDS = 4;
        const static int INPUTS = 7;
        const static int OUTPUTS = 1;

        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::bitset<INPUTS> in) { return (in[0] | ~in[1]) & ((in[0] ^ in[4]) | (in[3] & ~in[2])); };
        std::function<int(std::bitset<INPUTS> in)> parityFunc =
            [](std::bitset<INPUTS> in) { return (in.count() % 2 == 0) ? 0 : 1; };

        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;
    public:
        BoolProblem() {};
        BoolProblem(int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int POPULATION_SIZE)
            : GENERATIONS(GENERATIONS), ROWS(ROWS), COLUMNS(COLUMNS), LEVELS_BACK(LEVELS_BACK), POPULATION_SIZE(POPULATION_SIZE) {
        };
        BoolProblem(int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int POPULATION_SIZE, std::function<int(std::bitset<INPUTS> in)> boolFunc)
            : GENERATIONS(GENERATIONS), ROWS(ROWS), COLUMNS(COLUMNS), LEVELS_BACK(LEVELS_BACK), POPULATION_SIZE(POPULATION_SIZE), boolFunc(boolFunc) {};

        void problemRunner() override;
        void printFunction() override;
    };

    class ParityProblem : public BoolProblem {
    public:
        ParityProblem() : BoolProblem() { useFunc = false; };
        ParityProblem(int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int POPULATION_SIZE)
            : BoolProblem(GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE) { useFunc = false; };
    };
}

#endif
```

6.6 BoolTester.hpp

```
#ifndef BOOLTESTER_HPP
#define BOOLTESTER_HPP

#include "../Tester.hpp"
#include "../Timer.hpp"
#include "BoolProblem.hpp"

namespace parallel_cgp {
    struct BoolParam {
        BoolParam() {}
        BoolParam(int gens, int rows, int cols, int levels, int pop) : gens(gens), rows(rows), cols(cols), levels(levels), pop(pop) {}
        int gens;
        int rows;
        int cols;
        int levels;
        int pop;
    };

    class SeqBoolTester : private Tester, private BoolProblem
    {
```

```
    private:
        std::string boolFuncs[8] = { "smallSimpleSeqBoolTest", "mediumSimpleSeqBoolTest",
"largeSimpleSeqBoolTest", "specialSimpleSeqBoolTest", "smallComplexSeqBoolTest",
"mediumComplexSeqBoolTest", "largeComplexSeqBoolTest", "specialComplexSeqBoolTest" };
        BoolParam params[8] = { BoolParam(Tester::GENERATIONS, SMALL_ROWS, SMALL_COLUMNS,
SMALL_LEVELS, SMALL_POP_SIZE),
            BoolParam(Tester::GENERATIONS, MEDIUM_ROWS, MEDIUM_COLUMNS, MEDIUM_LEVELS,
MEDIUM_POP_SIZE),
            BoolParam(Tester::GENERATIONS, LARGE_ROWS, LARGE_COLUMNS, LARGE_LEVELS, LARGE_POP_SIZE),
            BoolParam(Tester::GENERATIONS, SPECIAL_ROWS, SPECIAL_COLUMNS, SPECIAL_LEVELS,
SPECIAL_POP_SIZE),
            BoolParam(Tester::GENERATIONS, SMALL_ROWS, SMALL_COLUMNS, SMALL_LEVELS, SMALL_POP_SIZE),
            BoolParam(Tester::GENERATIONS, MEDIUM_ROWS, MEDIUM_COLUMNS, MEDIUM_LEVELS,
MEDIUM_POP_SIZE),
            BoolParam(Tester::GENERATIONS, LARGE_ROWS, LARGE_COLUMNS, LARGE_LEVELS, LARGE_POP_SIZE),
            BoolParam(Tester::GENERATIONS, SPECIAL_ROWS, SPECIAL_COLUMNS, SPECIAL_LEVELS,
SPECIAL_POP_SIZE) };
        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]))); } };

        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) {
            Timer testTimer("boolTestTimer");

            BoolProblem problem(GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE, boolFunc);
            problem.problemRunner();

            testTimer.endTimer();

            saveResults(testName, GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE);
        }
    public:
        SeqBoolTester(BoolParam customParams) : Tester((customParams.pop == 0) ? "SeqBoolTest" :
"CustomSeqBoolTest") {
            if(customParams.pop != 0) {
                for(int i = 0; i < ROUNDS; i++)
                    test("CustomSeqBoolTest", customParams.gens, customParams.rows, customParams.cols,
customParams.levels, customParams.pop, func[0]);
                return;
            }

            for (int f = 0; f < (sizeof(boolFuncs) / sizeof(*boolFuncs)); f++) {
                for (int i = 0; i < ROUNDS; i++) {
                    if (f < 3)
                        test(boolFuncs[f], params[f].gens, params[f].rows, params[f].cols,
params[f].levels, params[f].pop, func[0]);
                    else
                        test(boolFuncs[f], params[f].gens, params[f].rows, params[f].cols,
params[f].levels, params[f].pop, func[1]);
                }
            }
        }
    };

    class ParBoolTester : private Tester, private BoolProblem
    {
    private:
        std::string boolFuncs[8] = { "smallSimpleParBoolTest", "mediumSimpleParBoolTest",
"largeSimpleParBoolTest", "specialSimpleParBoolTest", "smallComplexParBoolTest",
"mediumComplexParBoolTest", "largeComplexParBoolTest", "specialComplexParBoolTest" };
        BoolParam params[8] = { BoolParam(Tester::GENERATIONS, SMALL_ROWS, SMALL_COLUMNS,
SMALL_LEVELS, SMALL_POP_SIZE),
            BoolParam(Tester::GENERATIONS, MEDIUM_ROWS, MEDIUM_COLUMNS, MEDIUM_LEVELS,
MEDIUM_POP_SIZE),
            BoolParam(Tester::GENERATIONS, LARGE_ROWS, LARGE_COLUMNS, LARGE_LEVELS, LARGE_POP_SIZE),
            BoolParam(Tester::GENERATIONS, SPECIAL_ROWS, SPECIAL_COLUMNS, SPECIAL_LEVELS,
SPECIAL_POP_SIZE),
            BoolParam(Tester::GENERATIONS, SMALL_ROWS, SMALL_COLUMNS, SMALL_LEVELS, SMALL_POP_SIZE),
            BoolParam(Tester::GENERATIONS, MEDIUM_ROWS, MEDIUM_COLUMNS, MEDIUM_LEVELS,
MEDIUM_POP_SIZE),
            BoolParam(Tester::GENERATIONS, LARGE_ROWS, LARGE_COLUMNS, LARGE_LEVELS, LARGE_POP_SIZE),
            BoolParam(Tester::GENERATIONS, SPECIAL_ROWS, SPECIAL_COLUMNS, SPECIAL_LEVELS,
SPECIAL_POP_SIZE) };
        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]))); } };

        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) {
            Timer testTimer("boolTestTimer");

            omp_set_num_threads(THREAD_NUM);

            BoolProblem problem(GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE, boolFunc);
            problem.problemRunner();

            testTimer.endTimer();

            saveResults(testName, GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE);
        }
    public:
        ParBoolTester(BoolParam customParams) : Tester((customParams.pop == 0) ? "ParBoolTest" : "CustomParBoolTest") {
            if(customParams.pop != 0) {
                for (int t = 0; t < threadNums.size(); t++) {
                    for(int i = 0; i < ROUNDS; i++)
                        test("CustomParBoolTest", customParams.gens, customParams.rows, customParams.cols, customParams.levels, customParams.pop, func[0], threadNums[t]);
                    return;
                }
            }

            for (int f = 0; f < (sizeof(boolFuncs) / sizeof(*boolFuncs)); f++) {
                for (int t = 0; t < threadNums.size(); t++) {
                    for (int i = 0; i < ROUNDS; i++) {
                        if (f < 3)
                            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]);
                        else
                            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]);
                    }
                }
            }
        }
    };

    class SeqParityTester : private Tester
    {
    private:
        std::string parityFuncs[4] = { "smallSeqParityTest", "mediumSeqParityTest", "largeSeqParityTest", "specialSeqParityTest" };
        BoolParam params[4] = { BoolParam(GENERATIONS, SMALL_ROWS, SMALL_COLUMNS, SMALL_LEVELS, SMALL_POP_SIZE),
            BoolParam(GENERATIONS, MEDIUM_ROWS, MEDIUM_COLUMNS, MEDIUM_LEVELS, MEDIUM_POP_SIZE),
            BoolParam(GENERATIONS, LARGE_ROWS, LARGE_COLUMNS, LARGE_LEVELS, LARGE_POP_SIZE),
            BoolParam(GENERATIONS, SPECIAL_ROWS, SPECIAL_COLUMNS, SPECIAL_LEVELS, SPECIAL_POP_SIZE) };

        void test(std::string testName, int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int POPULATION_SIZE) {
            Timer testTimer("parityTestTimer");

            ParityProblem problem(GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE);
            problem.problemRunner();

            testTimer.endTimer();

            saveResults(testName, GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE);
        }
    public:
        SeqParityTester(BoolParam customParams) : Tester((customParams.pop == 0) ? "SeqParityTest" : "CustomSeqParityTest") {
            if(customParams.pop != 0) {
                for(int i = 0; i < ROUNDS; i++)
                    test("CustomSeqParityTest", customParams.gens, customParams.rows, customParams.cols, customParams.levels, customParams.pop);
                return;
            }

            for (int f = 0; f < (sizeof(parityFuncs) / sizeof(*parityFuncs)); f++)
                for (int i = 0; i < ROUNDS; i++)
                    test(parityFuncs[f], params[f].gens, params[f].rows, params[f].cols, params[f].levels, params[f].pop);
        }
    };

    class ParParityTester : private Tester
    {
    private:
        std::string parityFuncs[4] = { "smallParParityTest", "mediumParParityTest", "largeParParityTest", "specialParParityTest" };
        BoolParam params[4] = { BoolParam(GENERATIONS, SMALL_ROWS, SMALL_COLUMNS, SMALL_LEVELS, SMALL_POP_SIZE),
            BoolParam(GENERATIONS, MEDIUM_ROWS, MEDIUM_COLUMNS, MEDIUM_LEVELS, MEDIUM_POP_SIZE),
            BoolParam(GENERATIONS, LARGE_ROWS, LARGE_COLUMNS, LARGE_LEVELS, LARGE_POP_SIZE),
            BoolParam(GENERATIONS, SPECIAL_ROWS, SPECIAL_COLUMNS, SPECIAL_LEVELS, SPECIAL_POP_SIZE) };

        void test(std::string testName, int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int POPULATION_SIZE, int THREAD_NUM) {
            Timer testTimer("parityTestTimer");

            omp_set_num_threads(THREAD_NUM);

            ParityProblem problem(GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE);
            problem.problemRunner();

            testTimer.endTimer();

            saveResults(testName, GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE);
        }
    public:
        ParParityTester(BoolParam customParams) : Tester((customParams.pop == 0) ? "ParParityTest" : "CustomParParityTest") {
            if(customParams.pop != 0) {
                for (int t = 0; t < threadNums.size(); t++) {
                    for(int i = 0; i < ROUNDS; i++)
                        test("CustomParParityTest", customParams.gens, customParams.rows, customParams.cols, customParams.levels, customParams.pop, threadNums[t]);
                    return;
                }
            }

            for (int f = 0; f < (sizeof(parityFuncs) / sizeof(*parityFuncs)); f++)
                for (int t = 0; t < threadNums.size(); t++)
                    for (int i = 0; i < ROUNDS; i++)
                        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]);
        }
    };
}

#endif
```

6.7 CGP.cpp

```
#include "CGP.hpp"

using namespace std;
using namespace parallel_cgp;

void CGP::generatePopulation(vector<CGPIndividual> &population) {
    // vrijeme za izvodenje cijele funkcije
    Timer genTime("generatePopulationTimer");

    boost::random::mt19937 gen(chrono::duration_cast<std::chrono::nanoseconds>(chrono::system_clock::now().time_since_epoch()).count() * (omp_get_thread_num() + 1));

    for (int i = 0; i < populationSize; i++) {
        boost::random::uniform_int_distribution<> operandDis(1, operands);
        boost::random::uniform_int_distribution<> connectionDis(0, rows * columns + inputs - 1);
        boost::random::uniform_int_distribution<> outputDis(0, rows * columns + inputs - 1);

        vector<CGPNode> genes;
        vector<CGPOutput> outputGene;

        for (int k = 0; k < inputs; k++) {
            CGPNode node;
            node.used = false;
            node.connection1 = -1;
            node.connection2 = -1;
            node.operand = -1;
            genes.push_back(node);
        }

        for (int j = inputs; j < rows * columns + inputs; j++) {
            CGPNode node;
            node.used = false;
            node.operand = operandDis(gen);
            node.connection1 = connectionDis(gen);
            node.outValue = NAN;

            while (true) {
                if (node.connection1 < inputs)
                    break;
                if ((node.connection1 % columns) == (j % columns))
                    node.connection1 = connectionDis(gen);
                else if (((node.connection1 - inputs) % columns) > (((j - inputs) % columns) + levelsBack))
                    node.connection1 = connectionDis(gen);
                else if(genes.size() > node.connection1 && (genes[node.connection1].connection1 == j || genes[node.connection1].connection2 == j))
                    node.connection1 = connectionDis(gen);
                else
```

```
                    break;
            }

            node.connection2 = (node.operand >= biOperands) ? -1 : connectionDis(gen);

            while (true) {
                if (node.connection2 < inputs)
                    break;
                if ((node.connection2 % columns) == (j % columns))
                    node.connection2 = connectionDis(gen);
                else if (((node.connection2 - inputs) % columns) > (((j - inputs) % columns) + levelsBack))
                    node.connection2 = connectionDis(gen);
                else if (genes.size() > node.connection2 && (genes[node.connection2].connection1 == j || genes[node.connection2].connection2 == j))
                    node.connection2 = connectionDis(gen);
                else
                    break;
            }
            genes.push_back(node);
        }

        for (int k = 0; k < outputs; k++) {
            CGPOutput output;

            output.connection = outputDis(gen);
            outputGene.push_back(output);
        }

        CGPIndividual individual(genes, outputGene, rows, columns, levelsBack, inputs, outputs);

        population[i] = individual;
        population[i].resolveLoops();
    }

    genTime.endTimer();
}

void CGP::goldMutate(CGPIndividual parent, vector<CGPIndividual> &population) {
    Timer mutTime("mutatePopulationTimer");

    if (!parent.evalDone)
        parent.evaluateUsed();
    population[0] = parent;

    boost::random::mt19937 gen(chrono::duration_cast<std::chrono::nanoseconds>(chrono::system_clock::now().time_since_epoch()).count() * (omp_get_thread_num() + 1));

    for (int n = 1; n < populationSize; n++) {
        boost::random::uniform_int_distribution<> nodDis(parent.inputs, static_cast<int>(parent.genes.size()));
        boost::random::uniform_int_distribution<> geneDis(0, 2);
        boost::random::uniform_int_distribution<> connectionDis(0, static_cast<int>(parent.genes.size()) - 1);
        boost::random::uniform_int_distribution<> operandDis(1, operands);
        boost::random::uniform_int_distribution<> outputDis(0, parent.outputs - 1);

        vector<CGPNode> genes = parent.genes;
        vector<CGPOutput> outputGene = parent.outputGene;
        bool isActive = false;

        while (!isActive) {
            int mut = geneDis(gen);
            int cell = nodDis(gen);
            if (cell == parent.genes.size()) {
                outputGene[outputDis(gen)].connection = connectionDis(gen);
                break;
            }
            if (mut == 0) {
                genes[cell].operand = operandDis(gen);

                if (genes[cell].operand >= biOperands && genes[cell].connection2 != -1)
                    genes[cell].connection2 = -1;
                else if (genes[cell].operand < biOperands && genes[cell].connection2 == -1)
                    genes[cell].connection2 = connectionDis(gen);
            }
            else if (mut == 1)
                genes[cell].connection1 = connectionDis(gen);
            else if (mut == 2 && genes[cell].operand >= biOperands)
                continue;
            else if (mut == 2)
                genes[cell].connection2 = connectionDis(gen);

            while (true) {
                if (genes[cell].connection1 < parent.inputs)
                    break;
                if ((genes[cell].connection1 % parent.columns) == (cell % parent.columns))
                    genes[cell].connection1 = connectionDis(gen);
                else if (((genes[cell].connection1 - parent.inputs) % parent.columns) > (((cell - parent.inputs) % parent.columns) + parent.levelsBack))
                    genes[cell].connection1 = connectionDis(gen);
                else
                    break;
            }

            while (true) {
                if (genes[cell].connection2 < parent.inputs)
                    break;
                if ((genes[cell].connection2 % parent.columns) == (cell % parent.columns))
                    genes[cell].connection2 = connectionDis(gen);
                else if (((genes[cell].connection2 - parent.inputs) % parent.columns) > (((cell - parent.inputs) % parent.columns) + parent.levelsBack))
                    genes[cell].connection2 = connectionDis(gen);
                else
                    break;
            }

            isActive = genes[cell].used;
        }

        for (size_t z = parent.inputs; z < genes.size(); z++)
            genes[z].used = false;

        CGPIndividual individual(genes, outputGene, parent.rows, parent.columns, parent.levelsBack, parent.inputs, parent.outputs);

        population[n] = individual;
        population[n].resolveLoops();
    }

    mutTime.endTimer();
}
```

6.8 CGP.hpp

```
#ifndef CGP_HPP
#define CGP_HPP
#define TYPE double

#include "CGPIndividual.hpp"
#include "../Timer.hpp"
#include <iostream>
#include <chrono>
#include <thread>
#include <cmath>
#include <random>
#include <fstream>
#include <string>
#include <sstream>
#include <vector>
#include <omp.h>
#include <boost/random.hpp>

namespace parallel_cgp {
    class CGP {
    private:
        int rows, columns, levelsBack, inputs, outputs, operands, biOperands, populationSize;
    public:
        CGP(int rows, int columns, int levelsBack, int inputs, int outputs, int operands, int biOperands, int populationSize)
            : rows(rows), columns(columns), levelsBack(levelsBack), inputs(inputs), outputs(outputs),
                operands(operands), biOperands(biOperands), populationSize(populationSize) {};

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

        void goldMutate(CGPIndividual parent, std::vector<CGPIndividual> &population);
    };
}

#endif
```

6.9 CGPIndividual.cpp

```
#include "CGPIndividual.hpp"
```

```

using namespace std;
using namespace parallel_cgp;

CGPIndividual::CGPIndividual() {
    vector<vector<int» branches;
    this->branches = branches;
    this->rows = 0;
    this->columns = 0;
    this->levelsBack = 0;
    this->inputs = 0;
    this->outputs = 0;
    this->evalDone = false;
}

CGPIndividual::CGPIndividual(vector<CGPNode> genes, vector<CGPOutput> outputGene, int rows, int columns, int levelsBack, int inputs, int outputs) {
    vector<vector<int» branches;
    this->branches = branches;
    this->genes = genes;
    this->outputGene = outputGene;
    this->rows = rows;
    this->columns = columns;
    this->levelsBack = levelsBack;
    this->inputs = inputs;
    this->outputs = outputs;
    this->evalDone = false;
}

CGPIndividual::CGPIndividual(vector<CGPNode> genes, vector<CGPOutput> outputGene, int rows, int columns, int levelsBack, int inputs, int outputs, bool evalDone) {
    vector<vector<int» branches;
    this->branches = branches;
    this->genes = genes;
    this->outputGene = outputGene;
    this->rows = rows;
    this->columns = columns;
    this->levelsBack = levelsBack;
    this->inputs = inputs;
    this->outputs = outputs;
    this->evalDone = evalDone;
}

void CGPIndividual::printNodes() {
    for (size_t i = 0; i < rows * columns + inputs; i++)
        cout « i « " " « genes[i].operand « " " « genes[i].connection1 « " " « genes[i].connection2 « endl;

    for (size_t j = 0; j < outputs; j++)
        cout « outputGene[j].connection « " ";

    cout « endl « endl;
}

void CGPIndividual::evaluateUsed() {
    for (int m = 0; m < outputs; m++)
        isUsed(outputGene[m].connection);

    evalDone = true;
}

void CGPIndividual::isUsed(int CGPNodeNum) {
    genes[CGPNodeNum].used = true;

    if (genes[CGPNodeNum].connection1 >= 0)
        isUsed(genes[CGPNodeNum].connection1);

    if (genes[CGPNodeNum].connection2 >= 0)
        isUsed(genes[CGPNodeNum].connection2);
}

void CGPIndividual::evaluateValue(vector<TYPE> input, function<TYPE(int, TYPE, TYPE)> &computeNode) {
    clearInd();

    for (int l = 0; l < inputs; l++)
        genes[l].outValue = input[l];

    for (int m = 0; m < outputs; m++)
        outputGene[m].value = evalNode(outputGene[m].connection, computeNode);
}

TYPE CGPIndividual::evalNode(int CGPNodeNum, function<TYPE(int, TYPE, TYPE)> &computeNode) {

    if (isnan(genes[CGPNodeNum].outValue)) {
        TYPE value1 = evalNode(genes[CGPNodeNum].connection1, computeNode);
        TYPE value2 = genes[CGPNodeNum].connection2 < 0 ? 0 : evalNode(genes[CGPNodeNum].connection2, computeNode);

        genes[CGPNodeNum].outValue = computeNode(genes[CGPNodeNum].operand, value1, value2);
    }

    return genes[CGPNodeNum].outValue;
}

void CGPIndividual::clearInd() {
    for (int i = inputs; i < genes.size(); i++)
        genes[i].outValue = NAN;
}

bool CGPIndividual::findLoops(int CGPNodeNum) {
    branches.clear();

    vector<int> CGPNodeSet;

    return loopFinder(CGPNodeNum, CGPNodeSet);
}

bool CGPIndividual::loopFinder(int CGPNodeNum, vector<int> CGPNodeSet) {

    for (int i = 0; i < CGPNodeSet.size(); i++)
        if (CGPNodeSet[i] == CGPNodeNum) {
            CGPNodeSet.push_back(CGPNodeNum);
            branches.push_back(CGPNodeSet);
            return true;
        }

    CGPNodeSet.push_back(CGPNodeNum);

    if (CGPNodeNum < inputs) {
        return false;
    }

    bool conn1 = loopFinder(genes[CGPNodeNum].connection1, CGPNodeSet);
    bool conn2 = genes[CGPNodeNum].connection2 == -1 ? false : loopFinder(genes[CGPNodeNum].connection2, CGPNodeSet);

    return conn1 || conn2;
}

void CGPIndividual::resolveLoops() {

    Timer resLoopTime("resolveLoopsTimer");

    boost::random::mt19937 gen(chrono::duration_cast<std::chrono::nanoseconds>(chrono::system_clock::now().time_since_epoch()).count() * (omp_get_thread_num() + 1));

    for (int m = 0; m < outputs; m++) {
        while (findLoops(outputGene[m].connection)) {
            for (int i = 0; i < branches.size(); i++) {
                boost::random::uniform_int_distribution<> connectionDis(0, static_cast<int>(genes.size()) - 1);
                int cell1 = branches[i][branches[i].size() - 2];
                int cell2 = branches[i][branches[i].size() - 1];

                if (genes[cell1].connection1 == cell2) {
                    genes[cell1].connection1 = connectionDis(gen);

                    while (true) {
                        if (genes[cell1].connection1 < inputs)
                            break;
                        if ((genes[cell1].connection1 % columns) == (cell1 % columns))
                            genes[cell1].connection1 = connectionDis(gen);
                        else if (((genes[cell1].connection1 - inputs) % columns) > (((cell1 - inputs) % columns) + levelsBack))
                            genes[cell1].connection1 = connectionDis(gen);
                        else
                            break;
                    }
                }
                else if (genes[cell1].connection2 == cell2) {
                    genes[cell1].connection2 = connectionDis(gen);

                    while (true) {
                        if (genes[cell1].connection2 < inputs)
                            break;
                        if ((genes[cell1].connection2 % columns) == (cell1 % columns))
                            genes[cell1].connection2 = connectionDis(gen);
                        else if (((genes[cell1].connection2 - inputs) % columns) > (((cell1 - inputs) % columns) + levelsBack))
                            genes[cell1].connection2 = connectionDis(gen);
                        else
                            break;
                    }
                }
            }
        }
    }

    resLoopTime.endTimer();
}
```

6.10 CGPIndividual.hpp

```
#ifndef CGPINDIVIDUAL_HPP
#define CGPINDIVIDUAL_HPP
#define TYPE double

#include "CGPNode.hpp"
#include "CGPOutput.hpp"
#include "../Timer.hpp"
#include <vector>
#include <sstream>
#include <functional>
#include <omp.h>
#include <iostream>
#include <chrono>
#include <thread>
#include <boost/random.hpp>

namespace parallel_cgp {
    class CGPIndividual {
    private:
        void isUsed(int nodeNum);
        bool loopFinder(int nodeNum, std::vector<int> nodeSet);
        TYPE evalNode(int nodeNum, std::function<TYPE(int, TYPE, TYPE)> &computeNode);
        void clearInd();
    public:
        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;

        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();
    };
}

#endif
```

6.11 CGPNode.hpp

```
#ifndef CGPNODE_HPP
#define CGPNODE_HPP
#include <iostream>
#include <fstream>
#include <string>
#define TYPE double

namespace parallel_cgp {
    struct CGPNode {
        int operand;
        int connection1;
        int connection2;
        bool used;
        TYPE outValue;
    };
}

#endif
```

6.12 CGPOutput.hpp

```
#ifndef CGPOUTPUT_HPP
#define CGPOUTPUT_HPP
#include <iostream>
#include <fstream>
#include <string>
#define TYPE double

namespace parallel_cgp {
    struct CGPOutput {
        int connection;
        TYPE value;
    };
}

#endif
```

6.13 FuncProblem.cpp

```
#include "FuncProblem.hpp"

using namespace std;
using namespace parallel_cgp;

TYPE FuncProblem::computeNode(int operand, TYPE value1, TYPE value2) {
    switch (operand) {
    case 1:
        return value1 + value2;
    case 2:
        return value1 - value2;
    case 3:
        return value1 * value2;
    case 4:
        return (value2 == 0) ? 0 : value1 / value2;
    case 5:
        return sin(value1);
    case 6:
        return cos(value1);
    case 7:
        return value1 > 0 ? sqrt(value1) : value1;
    case 8:
        return pow(value1, 2);
    case 9:
        return pow(2, value1);
    default:
        return 0;
    }
}

TYPE FuncProblem::fitness(TYPE x, TYPE y, TYPE res) {
    return func(x, y) - res;
}

void FuncProblem::printFunction() {
    if (isSimulated)
        cout « "Funkcija: " « evalFunction(bestI->outputGene[0].connection) « endl;
    else
        cout « "Problem nije simuliran." « endl;
}

string FuncProblem::evalFunction(int CGPNodeNum) {
    ostringstream oss;

    if (CGPNodeNum < INPUTS) {
        switch (CGPNodeNum) {
        case 0:
            oss « "x";
            return oss.str();
        case 1:
            oss « "y";
            return oss.str();
        }
    }

    switch (bestI->genes[CGPNodeNum].operand) {
    case 1:
        oss « "(" « evalFunction(bestI->genes[CGPNodeNum].connection1) « " + " « evalFunction(bestI->genes[CGPNodeNum].connection2) « ")";
        return oss.str();
    case 2:
        oss « "(" « evalFunction(bestI->genes[CGPNodeNum].connection1) « " - " « evalFunction(bestI->genes[CGPNodeNum].connection2) « ")";
        return oss.str();
    case 3:
        oss « "(" « evalFunction(bestI->genes[CGPNodeNum].connection1) « " * " « evalFunction(bestI->genes[CGPNodeNum].connection2) « ")";
        return oss.str();
    case 4:
        oss « "(" « evalFunction(bestI->genes[CGPNodeNum].connection1) « " / " « evalFunction(bestI->genes[CGPNodeNum].connection2) « ")";
        return oss.str();
    case 5:
        oss « "sin(" « evalFunction(bestI->genes[CGPNodeNum].connection1) « ")";
        return oss.str();
    case 6:
        oss « "cos(" « evalFunction(bestI->genes[CGPNodeNum].connection1) « ")";
        return oss.str();
    case 7:
        oss « "sqrt(" « evalFunction(bestI->genes[CGPNodeNum].connection1) « ")";
        return oss.str();
    case 8:
        oss « evalFunction(bestI->genes[CGPNodeNum].connection1) « "^2";
        return oss.str();
    case 9:
        oss « "2^" « evalFunction(bestI->genes[CGPNodeNum].connection1);
        return oss.str();
    }

    return "";
}

void FuncProblem::problemSimulator(CGPIndividual& individual, TYPE& fit) {
    Timer probSimTime("problemSimulatorTimer");

    function<TYPE(int op, TYPE v1, TYPE v2)> compNode =
        [&](int op, TYPE v1, TYPE v2) { return computeNode(op, v1, v2); };

    TYPE N = 0;

    for (TYPE x = -10; x < 10; x += 0.5) {
        for (TYPE y = -10; y < 10; y += 0.5) {
            vector<TYPE> input;
            input.push_back(x);
            input.push_back(y);

            individual.evaluateValue(input, compNode);
            fit += pow(fitness(x, y, individual.outputGene[0].value), 2);
            N++;
        }
    }

    fit /= N;
    fit = sqrt(fit);

    probSimTime.endTimer();
}

void FuncProblem::problemRunner() {
    Timer probRunTime("problemRunnerTimer");

    CGP cgp(ROWS, COLUMNS, LEVELS_BACK, INPUTS, OUTPUTS, NUM_OPERANDS, BI_OPERANDS, POPULATION_SIZE);

    vector<CGPIndividual> population(POPULATION_SIZE);
    int bestInd = 0, generacija = 0;

    cgp.generatePopulation(population);

    for (generacija = 0; generacija < GENERATIONS; generacija++) {
        TYPE bestFit = DBL_MAX;
        bestInd = 0;
        vector<int> bestInds;
        boost::random::mt19937 gen(chrono::duration_cast<std::chrono::nanoseconds>(chrono::system_clock::now().time_since_epoch()).count() * (omp_get_thread_num() + 1));

        for (int clan = 0; clan < POPULATION_SIZE; clan++) {

            TYPE fit = 0;
            problemSimulator(population[clan], fit);

            if (fit < bestFit) {
                bestFit = fit;
                bestInds.clear();
                bestInds.push_back(clan);
            }
            else if (fit == bestFit)
                bestInds.push_back(clan);
        }

        if (bestInds.size() > 1)
            bestInds.erase(bestInds.begin());
        if (bestInds.size() == 0)
            bestInds.push_back(0);

        boost::random::uniform_int_distribution<> bestDis(0, static_cast<int>(bestInds.size()) - 1);

        bestInd = bestInds[bestDis(gen)];

        if(printGens)
            cout « "Gen: " « generacija « "; Fitness: " « bestFit « "; Indeks: " « bestInd « endl;

        if (bestFit <= THRESHOLD)
            break;
        if (generacija != GENERATIONS - 1)
            cgp.goldMutate(population[bestInd], population);
    }

    bestI = &population[bestInd];

    isSimulated = true;

    printFunction();

    probRunTime.endTimer();
}
```

6.14 FuncProblem.hpp

```
#ifndef FUNCPROBLEM_HPP
#define FUNCPROBLEM_HPP

#include "../Problem.hpp"
#include "../cgp/CGP.hpp"

#undef TYPE
#define TYPE double

namespace parallel_cgp {
    class FuncProblem : public Problem {
    private:
        const static int NUM_OPERANDS = 9;
        const static int BI_OPERANDS = 5;
        const static int INPUTS = 2;
        const static int OUTPUTS = 1;

        int GENERATIONS = 5000;
        int ROWS = 8;
        int COLUMNS = 8;
        int LEVELS_BACK = 1;
        int POPULATION_SIZE = 15;
        int THRESHOLD = 0;

        bool isSimulated = false;

        std::function<TYPE(TYPE x, TYPE y)> func =
            [](TYPE x, TYPE y) { return (pow(x, 2) + 2 * x * y + y); };

        TYPE computeNode(int operand, TYPE value1, TYPE value2) override;
        TYPE fitness(TYPE x, TYPE y, TYPE res);
        void problemSimulator(parallel_cgp::CGPIndividual& individual, TYPE& fit) override;
        std::string evalFunction(int CGPNodeNum) override;
    public:
        FuncProblem() {};
        FuncProblem(int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int POPULATION_SIZE, int THRESHOLD, std::function<TYPE(TYPE x, TYPE y)> func)
            : GENERATIONS(GENERATIONS), ROWS(ROWS), COLUMNS(COLUMNS), LEVELS_BACK(LEVELS_BACK), POPULATION_SIZE(POPULATION_SIZE), THRESHOLD(THRESHOLD), func(func) {
        };

        void problemRunner() override;
        void printFunction() override;
    };
}
```

```

#endif
```

6.15 FuncTester.hpp

```
#ifndef FUNCTESTER_HPP
#define FUNCTESTER_HPP

#include "../Tester.hpp"
#include "../Timer.hpp"
#include "FuncProblem.hpp"

namespace parallel_cgp {
    struct FuncParam {
        FuncParam() {}
        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) {}
        int gens;
        int rows;
        int cols;
        int levels;
        int pop;
        int thresh;
    };

    class SeqFuncTester : private Tester
    {
    private:
        std::string funcs[8] = { "smallSimpleSeqFuncTest", "mediumSimpleSeqFuncTest", "largeSimpleSeqFuncTest", "specialSimpleSeqFuncTest", "smallComplexSeqFuncTest", "mediumComplexSeqFuncTest", "largeComplexSeqFuncTest", "specialComplexSeqFuncTest" };
        FuncParam params[8] = { FuncParam(GENERATIONS, SMALL_ROWS, SMALL_COLUMNS, SMALL_LEVELS, SMALL_POP_SIZE, -1),
            FuncParam(GENERATIONS, MEDIUM_ROWS, MEDIUM_COLUMNS, MEDIUM_LEVELS, MEDIUM_POP_SIZE, -1),
            FuncParam(GENERATIONS, LARGE_ROWS, LARGE_COLUMNS, LARGE_LEVELS, LARGE_POP_SIZE, -1),
            FuncParam(GENERATIONS, SPECIAL_ROWS, SPECIAL_COLUMNS, SPECIAL_LEVELS, SPECIAL_POP_SIZE, -1),
            FuncParam(GENERATIONS, SMALL_ROWS, SMALL_COLUMNS, SMALL_LEVELS, SMALL_POP_SIZE, -1),
            FuncParam(GENERATIONS, MEDIUM_ROWS, MEDIUM_COLUMNS, MEDIUM_LEVELS, MEDIUM_POP_SIZE, -1),
            FuncParam(GENERATIONS, LARGE_ROWS, LARGE_COLUMNS, LARGE_LEVELS, LARGE_POP_SIZE, -1),
            FuncParam(GENERATIONS, SPECIAL_ROWS, SPECIAL_COLUMNS, SPECIAL_LEVELS, SPECIAL_POP_SIZE, -1) };
        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)); } };

        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) {
            Timer testTimer("funcTestTimer");

            FuncProblem problem(GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE, THRESHOLD, func);
            problem.problemRunner();

            testTimer.endTimer();

            saveResults(testName, GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE);
        }
    public:
        SeqFuncTester(FuncParam customParams) : Tester((customParams.pop == 0) ? "SeqFuncTest" : "CustomSeqFuncTest") {
            if(customParams.pop != 0) {
                for(int i = 0; i < ROUNDS; i++)
                    test("CustomSeqFuncTest", customParams.gens, customParams.rows, customParams.cols, customParams.levels, customParams.pop, customParams.thresh, func[0]);
                return;
            }

            for (int f = 0; f < (sizeof(funcs) / sizeof(*funcs)); f++) {
                for (int i = 0; i < ROUNDS; i++) {
                    if (f < 3)
                        test(funcs[f], params[f].gens, params[f].rows, params[f].cols, params[f].levels, params[f].pop, params[f].thresh, func[0]);
                    else
                        test(funcs[f], params[f].gens, params[f].rows, params[f].cols, params[f].levels, params[f].pop, params[f].thresh, func[1]);
                }
            }
        }
    };

    class ParFuncTester : private Tester
    {
```

```
    private:
        std::string funcs[8] = { "smallSimpleParFuncTest", "mediumSimpleParFuncTest", "largeSimpleParFuncTest", "specialSimpleParFuncTest", "smallComplexParFuncTest", "mediumComplexParFuncTest", "largeComplexParFuncTest", "specialComplexParFuncTest" };
        FuncParam params[8] = { FuncParam(GENERATIONS, SMALL_ROWS, SMALL_COLUMNS, SMALL_LEVELS, SMALL_POP_SIZE, -1),
            FuncParam(GENERATIONS, MEDIUM_ROWS, MEDIUM_COLUMNS, MEDIUM_LEVELS, MEDIUM_POP_SIZE, -1),
            FuncParam(GENERATIONS, LARGE_ROWS, LARGE_COLUMNS, LARGE_LEVELS, LARGE_POP_SIZE, -1),
            FuncParam(GENERATIONS, SPECIAL_ROWS, SPECIAL_COLUMNS, SPECIAL_LEVELS, SPECIAL_POP_SIZE, -1),
            FuncParam(GENERATIONS, SMALL_ROWS, SMALL_COLUMNS, SMALL_LEVELS, SMALL_POP_SIZE, -1),
            FuncParam(GENERATIONS, MEDIUM_ROWS, MEDIUM_COLUMNS, MEDIUM_LEVELS, MEDIUM_POP_SIZE, -1),
            FuncParam(GENERATIONS, LARGE_ROWS, LARGE_COLUMNS, LARGE_LEVELS, LARGE_POP_SIZE, -1),
            FuncParam(GENERATIONS, SPECIAL_ROWS, SPECIAL_COLUMNS, SPECIAL_LEVELS, SPECIAL_POP_SIZE, -1) };
        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)); } };

        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) {
            Timer testTimer("funcTestTimer");

            omp_set_num_threads(THREAD_NUM);

            FuncProblem problem(GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE, THRESHOLD, func);
            problem.problemRunner();

            testTimer.endTimer();

            saveResults(testName, GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE);
        }
    public:
        ParFuncTester(FuncParam customParams) : Tester((customParams.pop == 0) ? "ParFuncTest" : "CustomParFuncTest") {
            if(customParams.pop != 0) {
                for (int t = 0; t < threadNums.size(); t++) {
                    for(int i = 0; i < ROUNDS; i++)
                        test("CustomParFuncTest", customParams.gens, customParams.rows, customParams.cols, customParams.levels, customParams.pop, customParams.thresh, func[0], threadNums[t]);
                    return;
                }
            }

            for (int f = 0; f < (sizeof(funcs) / sizeof(*funcs)); f++) {
                for (int t = 0; t < threadNums.size(); t++) {
                    for (int i = 0; i < ROUNDS; i++) {
                        if (f < 3)
                            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]);
                        else
                            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]);
                    }
                }
            }
        }
    };
}

#endif
```

6.16 main.cpp

```
#include "Problem.hpp"
#include "Timer.hpp"
#include "boolProblem/BoolTester.hpp"
#include "funcProblem/FuncTester.hpp"
#include "waitProblem/WaitTester.hpp"
#include "adProblem/ADTester.hpp"
#include "boolProblem/BoolProblem.hpp"
#include "funcProblem/FuncProblem.hpp"
#include "waitProblem/WaitProblem.hpp"
#include "adProblem/ADProblem.hpp"

#include <iostream>
#include <omp.h>
#include <boost/program_options.hpp>

```

```
#define PARAM_COUNT 5

using namespace std;
using namespace parallel_cgp;
namespace po = boost::program_options;

#if (defined(_OPENMP) && (defined(OMPCGP) || defined(OMPSIM) || defined(OMPRUN)))
#define BoolTester ParBoolTester
#define ParityTester ParParityTester
#define FuncTester ParFuncTester
#define ADTester ParADTester
#define WaitTester ParWaitTester
#define PARALLEL_TESTER 1
#else
#define BoolTester SeqBoolTester
#define ParityTester SeqParityTester
#define FuncTester SeqFuncTester
#define ADTester SeqADTester
#define WaitTester SeqWaitTester
#define PARALLEL_TESTER 0
#endif

int main(int ac, char** av) {
    try {
        int threads = 1;

        po::options_description desc("Allowed options");
        desc.add_options()
            ("help,h", "produce help message")
            ("test,t", "enable testing")
            ("bool,b", "enable bool problem")
            ("parity,p", "enable parity problem")
            ("func,f", "enable func problem")
            ("acey,a", "enable acey problem")
            ("wait,w", "enable wait problem")
            ("custom,c", po::value<std::vector<int»()->multitoken(), "custom test values (number of generations, rows, columns, levels, population size)")
            ("threads,T", po::value<int>(), "number of threads to use in parallel version")
            ("version,v", "print version information")
            ;

        po::variables_map vm;
        po::store(po::parse_command_line(ac, av, desc), vm);
        po::notify(vm);

        vector<int> params(PARAM_COUNT, 0);
        if (!vm["custom"].empty() && !((params = vm["custom"].as<vector<int> >()).size() == PARAM_COUNT))
            throw invalid_argument("Not the right amount of custom parameters");

        if (vm.count("help")) {
            cout « desc « endl;
            return 1;
        }

        if (vm.count("version")) {
            cout « "ParallelCGP version 1.0 Sequential" « endl;
            cout « "Author: Andrija Macek" « endl;
            return 2;
        }

        if (vm.count("threads")) {
            if (!PARALLEL_TESTER)
                throw invalid_argument("Threads are not supported in the sequential version of the program");
            threads = vm["threads"].as<int>();
            if (threads < 1)
                throw invalid_argument("Number of threads must be greater than 0");
            Tester::threadNums.clear();
            Tester::threadNums.push_back(threads);
        }

        Problem* problem = nullptr;

        if (vm.count("bool"))
            if (vm.count("test"))
                BoolTester boolTest = BoolTester(BoolParam(params[0], params[1], params[2], params[3], params[4]));
            else {
                omp_set_num_threads(threads);
                problem = new BoolProblem;
                problem->printGens = true;
                problem->problemRunner();
            }
        if (vm.count("parity"))
            if (vm.count("test"))
                ParityTester parityTest = ParityTester(BoolParam(params[0], params[1], params[2],
params[3], params[4]));
            else {
                omp_set_num_threads(threads);
                problem = new ParityProblem;
                problem->printGens = true;
                problem->problemRunner();
            }
        if (vm.count("func"))
            if (vm.count("test"))
                FuncTester funcTest = FuncTester(FuncParam(params[0], params[1], params[2], params[3],
params[4], -1));
            else {
                omp_set_num_threads(threads);
                problem = new FuncProblem;
                problem->printGens = true;
                problem->problemRunner();
            }
        if (vm.count("acey"))
            if (vm.count("test"))
                ADTester adTest = ADTester(ADParam(params[0], params[1], params[2], params[3],
params[4]));
            else {
                omp_set_num_threads(threads);
                problem = new ADProblem;
                problem->printGens = true;
                problem->problemRunner();
            }
        if (vm.count("wait"))
            if (vm.count("test"))
                WaitTester waitTest = WaitTester(WaitParam(params[0], params[1], params[2], params[3],
params[4], 1));
            else {
                omp_set_num_threads(threads);
                problem = new WaitProblem;
                problem->printGens = true;
                problem->problemRunner();
            }

        if (vm.count("test"))
            delete(problem);
    }
    catch(exception& e) {
        cerr « "error: " « e.what() « endl;
        return 1;
    }
    catch(...) {
        cerr « "Exception of unknown type!" « endl;
    }

    return 0;
}
```

6.17 Problem.hpp

```
#ifndef PROBLEM_HPP
#define PROBLEM_HPP
#define TYPE double

#include "Timer.hpp"
#include "cgp/CGPIndividual.hpp"
#include <cmath>
#include <random>
#include <cfloat>
#include <climits>
#include <chrono>
#include <boost/random.hpp>

namespace parallel_cgp {
    class Problem {
    private:
        virtual void problemSimulator(parallel_cgp::CGPIndividual &individual, TYPE &fit) {}
        virtual std::string evalFunction(int CGPNodeNum) = 0;
    public:
        virtual ~Problem() = default;
        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;

        virtual TYPE computeNode(int operand, TYPE value1, TYPE value2) {
            switch (operand) {
            case 1:
                return value1 + value2;
            case 2:
                return value1 - value2;
            case 3:
                return value1 * value2;
            case 4:
                return (value2 == 0) ? 0 : value1 / value2;
            case 5:
                return sin(value1);
            case 6:
                return cos(value1);
            case 7:
                return value1 > 0 ? sqrt(value1) : value1;
            case 8:
                return pow(value1, 2);
            case 9:
                return pow(2, value1);
            default:
                return 0;
            }
        }
        virtual TYPE fitness(TYPE fit) { return fit; }

        virtual void problemRunner() = 0;
        virtual void printFunction() = 0;
    };
}

#endif
```

6.18 Tester.hpp

```
#ifndef TESTER_HPP
#define TESTER_HPP

#include "Timer.hpp"
#include <omp.h>
#include <string>
#include <iostream>
#include <fstream>
#include <vector>

#ifndef _OPENMP
#define omp_set_num_threads(threads) 0
#endif

namespace parallel_cgp {
    class Tester
    {
    private:
        std::string testerName;
        std::string filename;
    public:
        inline static std::string VERSION_NAME = "";

        const static int ROUNDS = 10;
        const static int GENERATIONS = 1000;
        const static int SMALL_ROWS = 4;
        const static int MEDIUM_ROWS = 8;
        const static int LARGE_ROWS = 10;
        const static int SPECIAL_ROWS = 1;
        const static int SMALL_COLUMNS = 4;
        const static int MEDIUM_COLUMNS = 8;
        const static int LARGE_COLUMNS = 10;
        const static int SPECIAL_COLUMNS = 100;
        const static int SMALL_LEVELS = 0;
        const static int MEDIUM_LEVELS = 1;
        const static int LARGE_LEVELS = 3;
        const static int SPECIAL_LEVELS = 10;
        const static int SMALL_POP_SIZE = 5;
        const static int MEDIUM_POP_SIZE = 8;
        const static int LARGE_POP_SIZE = 16;
        const static int SPECIAL_POP_SIZE = 5;
        inline static std::vector<int> threadNums = { 1, 2, 4, 8, 16 };

        Tester(std::string testerName) : testerName(testerName), filename(testerName) {
            filename.append(VERSION_NAME);
            filename.append(".csv");
            std::ofstream myFile;
            myFile.open(filename);
            myFile.close();
        }

        void saveResults(std::string testName, int gens, int rows, int cols, int levels, int pop) {
            Timer::saveTimes(filename, testName, gens, rows, cols, levels, pop);

            std::cout « "----------------------------------------" « std::endl;
            std::cout « "TEST NAME: " « testName « std::endl;
            std::cout « "----------------------------------------" « std::endl;
            std::cout « "GENS: " « gens « ", ROWS: " « rows « ", COLUMNS: " « cols
                « ", LEVELS BACK: " « levels « ", POP SIZE: " « pop « std::endl;
            std::cout « "----------------------------------------" « std::endl;
            Timer::clearTimes();
        }
    };
}

#endif
```

6.19 Timer.hpp

```
#ifndef TIMER_HPP
#define TIMER_HPP

#include <omp.h>
#include <chrono>
#include <map>
#include <string>
#include <functional>
#include <iostream>
#include <fstream>

#ifdef _OPENMP
#define timerFunc() omp_get_wtime()
#define timerDiff(startTime, endTime) (endTime - startTime)
#define TIME_UNIT double
#else
#define timerFunc() std::chrono::steady_clock::now()
#define timerDiff(startTime, endTime) (std::chrono::duration_cast<std::chrono::microseconds>(endTime - startTime).count() / 1000000.0)
#define TIME_UNIT std::chrono::steady_clock::time_point
#endif

namespace parallel_cgp {

    class Timer
    {
    private:
        inline static std::map<std::string, std::vector<double» mapa;

        std::string funcName;
        TIME_UNIT start;
        double end;
    public:
        Timer(std::string funcName) : funcName(funcName), start(timerFunc()), end(0) {}

        void endTimer() {
            end = timerDiff(start, timerFunc());

            #pragma omp critical
            parallel_cgp::Timer::mapa[funcName].push_back(end);
        }

        static void printTimes() {
            for (const auto& [key, value] : parallel_cgp::Timer::mapa)
                for (const auto& val : value)
                    std::cout « ’[’ « key « "] = " « val « "; " « std::endl;
        }

        static void saveTimes(std::string filename, std::string testName, int gens, int rows, int cols, int levels, int pop) {
            std::ofstream myFile;
            myFile.open(filename, std::ios_base::app);
            myFile « "TEST NAME: " « testName;
            myFile « ", GENS: " « gens « ", ROWS: " « rows « ", COLUMNS: " « cols
                « ", LEVELS BACK: " « levels « ", POP SIZE: " « pop « std::endl;

            for (const auto& [key, value] : parallel_cgp::Timer::mapa) {
                myFile « '[' « key « "],";
                for (const auto& val : value)
                    myFile « val « ',';
                myFile « std::endl;
            }
            myFile.close();
        }

        static void clearTimes() {
            parallel_cgp::Timer::mapa.clear();
        }
    };
}

#endif

```

6.20 WaitProblem.cpp

```
#include "WaitProblem.hpp"

using namespace std;
using namespace parallel_cgp;

TYPE WaitProblem::fitness(TYPE prev) {
    return ++prev;
}

void WaitProblem::printFunction() {
    if (isSimulated)
        cout « "Funkcija: " « evalFunction(0) « endl;
    else
        cout « "Problem nije simuliran." « endl;
}

string WaitProblem::evalFunction(int CGPNodeNum) {
    ostringstream oss;

    if (!CGPNodeNum) {
        oss « "Wait time: " « WAIT_TIME « "ns";
        return oss.str();
    }

    return "";
}

void WaitProblem::problemSimulator(CGPIndividual& individual, TYPE& fit) {
    Timer probSimTime("problemSimulatorTimer");

    function<TYPE(int op, TYPE v1, TYPE v2)> compNode =
        [&](int op, TYPE v1, TYPE v2) { return computeNode(op, v1, v2); };

    for (int iter = 0; iter < 10; iter++) {
        vector<TYPE> input;
        input.push_back(iter);

        individual.evaluateValue(input, compNode);
        waitFunc();
    }
    fit = fitness(fit);

    probSimTime.endTimer();
}

void WaitProblem::problemRunner() {
    Timer probRunTime("problemRunnerTimer");

    CGP cgp(ROWS, COLUMNS, LEVELS_BACK, INPUTS, OUTPUTS, NUM_OPERANDS, BI_OPERANDS, POPULATION_SIZE);

    vector<CGPIndividual> population(POPULATION_SIZE);
    int bestInd = 0, generacija = 0;

    cgp.generatePopulation(population);

    for (generacija = 0; generacija < GENERATIONS; generacija++) {
        TYPE bestFit = 0;
        bestInd = 0;
        vector<int> bestInds;
        boost::random::mt19937
gen(chrono::duration_cast<std::chrono::nanoseconds>(chrono::system_clock::now().time_since_epoch()).count()
* (omp_get_thread_num() + 1));

        for (int clan = 0; clan < POPULATION_SIZE; clan++) {

            TYPE fit = generacija;
            problemSimulator(population[clan], fit);

            if (fit > bestFit) {
                bestFit = fit;
                bestInds.clear();
                bestInds.push_back(clan);
            }
            else if (fit == bestFit)
                bestInds.push_back(clan);
        }

        if (bestInds.size() > 1)
            bestInds.erase(bestInds.begin());
        if (bestInds.size() == 0)
            bestInds.push_back(0);

        boost::random::uniform_int_distribution<> bestDis(0, static_cast<int>(bestInds.size()) - 1);

        bestInd = bestInds[bestDis(gen)];

        if(printGens)
            cout « "Gen: " « generacija « "; Fitness: " « bestFit « "; Indeks: " « bestInd « endl;

        if (generacija != GENERATIONS - 1)
            cgp.goldMutate(population[bestInd], population);
    }

    bestI = &population[bestInd];

    isSimulated = true;

    printFunction();

    probRunTime.endTimer();
}
```

6.21 WaitProblem.hpp

```
#ifndef WAITPROBLEM_HPP
#define WAITPROBLEM_HPP

#include "../Problem.hpp"
#include "../cgp/CGP.hpp"
#include <chrono>
#include <thread>

#undef TYPE
#define TYPE double

namespace parallel_cgp {
    class WaitProblem : public Problem {
    private:
        int GENERATIONS = 200;
        int ROWS = 8;
        int COLUMNS = 8;
        int LEVELS_BACK = 3;
        int POPULATION_SIZE = 15;
        int INPUTS = 1;
        int OUTPUTS = 1;

        int WAIT_TIME = 50;

        bool isSimulated = false;

        const std::function<void()> waitFunc =
            [&]() { std::this_thread::sleep_for(std::chrono::nanoseconds(WAIT_TIME)); };

        TYPE fitness(TYPE prev) override;
        void problemSimulator(CGPIndividual& individual, TYPE& fit) override;
        std::string evalFunction(int CGPNodeNum) override;
    public:
        WaitProblem() {};
        WaitProblem(int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int POPULATION_SIZE, int WAIT_TIME)
            : GENERATIONS(GENERATIONS), ROWS(ROWS), COLUMNS(COLUMNS), LEVELS_BACK(LEVELS_BACK), POPULATION_SIZE(POPULATION_SIZE), WAIT_TIME(WAIT_TIME) {};

        void problemRunner() override;
        void printFunction() override;
    };
}
```

```

#endif
```

6.22 WaitTester.hpp

```
#ifndef WAITTESTER_HPP
#define WAITTESTER_HPP

#include "../Tester.hpp"
#include "../Timer.hpp"
#include "WaitProblem.hpp"

namespace parallel_cgp {
    struct WaitParam {
        WaitParam() {}
        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) {}
        int gens;
        int rows;
        int cols;
        int levels;
        int pop;
        int time;
    };

    class SeqWaitTester : private Tester
    {
    private:
        std::string funcs[4] = { "smallSeqWaitTest", "mediumSeqWaitTest", "largeSeqWaitTest", "specialSeqWaitTest" };
        WaitParam params[4] = { WaitParam(GENERATIONS, SMALL_ROWS, SMALL_COLUMNS, SMALL_LEVELS, SMALL_POP_SIZE, 1),
            WaitParam(GENERATIONS, MEDIUM_ROWS, MEDIUM_COLUMNS, MEDIUM_LEVELS, MEDIUM_POP_SIZE, 1),
            WaitParam(GENERATIONS, LARGE_ROWS, LARGE_COLUMNS, LARGE_LEVELS, LARGE_POP_SIZE, 1),
            WaitParam(GENERATIONS, SPECIAL_ROWS, SPECIAL_COLUMNS, SPECIAL_LEVELS, SPECIAL_POP_SIZE, 1) };

        void test(std::string testName, int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int POPULATION_SIZE, int WAIT_TIME) {
            Timer testTimer("waitTestTimer");

            WaitProblem problem(GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE, WAIT_TIME);
            problem.problemRunner();

            testTimer.endTimer();

            saveResults(testName, GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE);
        }
    public:
        SeqWaitTester(WaitParam customParams) : Tester((customParams.pop == 0) ? "SeqWaitTest" : "CustomSeqWaitTest") {
            if(customParams.pop != 0) {
                for(int i = 0; i < ROUNDS; i++)
                    test("CustomSeqWaitTest", customParams.gens, customParams.rows, customParams.cols, customParams.levels, customParams.pop, customParams.time);
                return;
            }

            for (int f = 0; f < (sizeof(funcs) / sizeof(*funcs)); f++) {
                for (int i = 0; i < ROUNDS; i++) {
                    test(funcs[f], params[f].gens, params[f].rows, params[f].cols, params[f].levels, params[f].pop, params[f].time);
                }
            }
        }
    };

    class ParWaitTester : private Tester
    {
    private:
        std::string funcs[4] = { "smallParWaitTest", "mediumParWaitTest", "largeParWaitTest", "specialParWaitTest" };
        WaitParam params[4] = { WaitParam(GENERATIONS, SMALL_ROWS, SMALL_COLUMNS, SMALL_LEVELS, SMALL_POP_SIZE, 1),
            WaitParam(GENERATIONS, MEDIUM_ROWS, MEDIUM_COLUMNS, MEDIUM_LEVELS, MEDIUM_POP_SIZE, 1),
            WaitParam(GENERATIONS, LARGE_ROWS, LARGE_COLUMNS, LARGE_LEVELS, LARGE_POP_SIZE, 1),
            WaitParam(GENERATIONS, SPECIAL_ROWS, SPECIAL_COLUMNS, SPECIAL_LEVELS, SPECIAL_POP_SIZE, 1) };

        void test(std::string testName, int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int POPULATION_SIZE, int WAIT_TIME, int THREAD_NUM) {
            Timer testTimer("waitTestTimer");

            omp_set_num_threads(THREAD_NUM);

            WaitProblem problem(GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE, WAIT_TIME);
            problem.problemRunner();

            testTimer.endTimer();

            saveResults(testName, GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, POPULATION_SIZE);
        }
    public:
        ParWaitTester(WaitParam customParams) : Tester((customParams.pop == 0) ? "ParWaitTest" : "CustomParWaitTest") {
            if(customParams.pop != 0) {
                for (int t = 0; t < threadNums.size(); t++) {
                    for(int i = 0; i < ROUNDS; i++)
                        test("CustomParWaitTest", customParams.gens, customParams.rows, customParams.cols, customParams.levels, customParams.pop, customParams.time, threadNums[t]);
                    return;
                }
            }

            for (int f = 0; f < (sizeof(funcs) / sizeof(*funcs)); f++) {
                for (int t = 0; t < threadNums.size(); t++) {
                    for (int i = 0; i < ROUNDS; i++) {
                        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]);
                    }
                }
            }
        }
    };
}

#endif
```

Index