ParallelCGP

1.0.0

Generated by Doxygen 1.13.2

Chapter 1

ParallelCGP

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

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

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

Chapter 3

Class Index

3.1 Class List

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

Chapter 4

File Index

4.1 File List

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

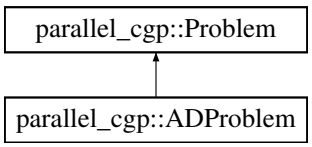
Chapter 5

Class Documentation

5.1 parallel_cgp::ADProblem Class Reference

`#include <ADProblem.h>`

Inheritance diagram for parallel_cgp::ADProblem:



Public Member Functions

- void problemRunner () override
- void printFunction () override
- void playGame ()

Public Member Functions inherited from parallel_cgp::Problem

- virtual TYPE fitness (TYPE fit)

Additional Inherited Members

Public Attributes inherited from parallel_cgp::Problem

- std::string bestFile = "problem_best.txt"
- int NUM_OPERANDS = 9
- int BI_OPERANDS = 5
- int GENERATIONS = 5000
- int ROWS = 20
- int COLUMNS = 20
- int LEVELS_BACK = 3
- int INPUTS = 6
- int OUTPUTS = 1
- int MUTATIONS = 6
- int POPULATION_SIZE = 20

5.1.1 Detailed Description

Klasa koja predstavlja problem igranja Acey Deucey igre.

Definition at line 14 of file ADProblem.h.

5.1.2 Member Function Documentation

5.1.2.1 playGame()

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

Metoda prikaze kako najbolja jedinka igra jednu partiju igre.

Definition at line 183 of file ADProblem.cpp.

5.1.2.2 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.

5.1.2.3 problemRunner()

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

Metoda za pokretanje problema.

Implements parallel_cgp::Problem.

Definition at line 112 of file ADProblem.cpp.

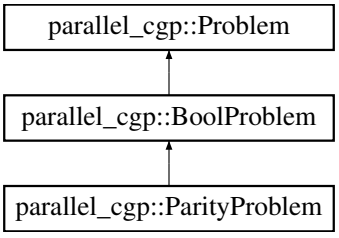
The documentation for this class was generated from the following files:

- adProblem/ADProblem.h
- adProblem/ADProblem.cpp

5.2 parallel_cgp::BoolProblem Class Reference

```
#include <BoolProblem.h>
```

Inheritance diagram for parallel_cgp::BoolProblem:



Public Member Functions

- BoolProblem ()
- BoolProblem (int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int MUTATIONS, int POPULATION_SIZE)
- void problemRunner () override
- void printFunction () override

Public Member Functions inherited from parallel_cgp::Problem

- 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

- CGPIndividual bestI
- const std::string bestFile = "bool_best.txt"
- int GENERATIONS = 5000
- int ROWS = 100
- int COLUMNS = 20
- int LEVELS_BACK = 0
- int MUTATIONS = 0
- int POPULATION_SIZE = 20
- bool isSimulated = false
- bool useFunc = true
- const std::function< int(std::bitset< INPUTS > in)> boolFunc
- const 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

- std::string bestFile = "problem_best.txt"
- int NUM_OPERANDS = 9
- int BI_OPERANDS = 5
- int GENERATIONS = 5000
- int ROWS = 20
- int COLUMNS = 20
- int LEVELS_BACK = 3
- int INPUTS = 6
- int OUTPUTS = 1
- int MUTATIONS = 6
- int POPULATION_SIZE = 20

5.2.1 Detailed Description

Klasa koja opisuje problem pronalaska boolean funkcije.
Moze se koristiti i za paritetni problem

Definition at line 16 of file BoolProblem.h.

5.2.2 Constructor & Destructor Documentation

5.2.2.1 BoolProblem() [1/2]

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

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

Definition at line 76 of file BoolProblem.h.

5.2.2.2 BoolProblem() [2/2]

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

Konstruktor koji prima sve promjenjive vrijednosti za bool problem.

Definition at line 80 of file BoolProblem.h.

5.2.3 Member Function Documentation

5.2.3.1 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.

5.2.3.2 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.

5.2.3.3 fitness()

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

Definition at line 21 of file BoolProblem.cpp.

5.2.3.4 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.

5.2.3.5 problemRunner()

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

Metoda za pokretanje problema.

Implements parallel_cgp::Problem.

Definition at line 77 of file BoolProblem.cpp.

5.2.3.6 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.
in	*fit*	Referenca na varijablu u koju se pohranjuje fitness.

Reimplemented from parallel_cgp::Problem.

Definition at line 61 of file BoolProblem.cpp.

5.2.4 Member Data Documentation

5.2.4.1 bestFile

```
const std::string parallel_cgp::BoolProblem::bestFile = "bool_best.txt"  [protected]
```

Naziv datoteke koja sadrzi najbolju jedinku.

Definition at line 25 of file BoolProblem.h.

5.2.4.2 bestI

```
CGPIndividual parallel_cgp::BoolProblem::bestI  [protected]
```

Najbolja jedinka nakon pokretanja problem simulatora.

Definition at line 21 of file BoolProblem.h.

5.2.4.3 BI_OPERANDS

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

Definition at line 33 of file BoolProblem.h.

5.2.4.4 boolFunc

```
const 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 60 of file BoolProblem.h.

5.2.4.5 COLUMNS

```
int parallel_cgp::BoolProblem::COLUMNS = 20  [protected]
```

Definition at line 43 of file BoolProblem.h.

5.2.4.6 GENERATIONS

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

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

Definition at line 41 of file BoolProblem.h.

5.2.4.7 INPUTS

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

Definition at line 34 of file BoolProblem.h.

5.2.4.8 isSimulated

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

Parametar koji oznacava je li simulacija obavljena.

Definition at line 51 of file BoolProblem.h.

5.2.4.9 LEVELS_BACK

```
int parallel_cgp::BoolProblem::LEVELS_BACK = 0  [protected]
```

Definition at line 44 of file BoolProblem.h.

5.2.4.10 MUTATIONS

```
int parallel_cgp::BoolProblem::MUTATIONS = 0  [protected]
```

Definition at line 45 of file BoolProblem.h.

5.2.4.11 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 32 of file BoolProblem.h.

5.2.4.12 OUTPUTS

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

Definition at line 35 of file BoolProblem.h.

5.2.4.13 parityFunc

```
const 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 65 of file BoolProblem.h.

5.2.4.14 POPULATION_SIZE

```
int parallel_cgp::BoolProblem::POPULATION_SIZE = 20  [protected]
```

Definition at line 46 of file BoolProblem.h.

5.2.4.15 ROWS

```
int parallel_cgp::BoolProblem::ROWS = 100  [protected]
```

Definition at line 42 of file BoolProblem.h.

5.2.4.16 useFunc

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

Parametar koji oznacava koristi li se funkcija ili partiet.

Definition at line 55 of file BoolProblem.h.

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

- boolProblem/BoolProblem.h
- boolProblem/BoolProblem.cpp

5.3 parallel_cgp::CGP Class Reference

```
#include <CGP.h>
```

Public Member Functions

- CGP (int generations, int rows, int columns, int levelsBack, int inputs, int outputs, int mutations, int operands, int biOperands, int populationSize)
- std::vector< CGPIndividual > generatePopulation ()
- std::vector< CGPIndividual > pointMutate (CGPIndividual parent)
- std::vector< CGPIndividual > goldMutate (CGPIndividual parent)

5.3.1 Detailed Description

Klasa koja opisuje CGP instancu.

Definition at line 13 of file CGP.h.

5.3.2 Constructor & Destructor Documentation

5.3.2.1 CGP()

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

Konstruktor za CGP klasu.

Parameters

in	*generations*	Broj generacija koji ce se izvrtiti pri ucenju.
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	*mutations*	Broj mutacija genoma po jedinki.
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 30 of file CGP.h.

5.3.3 Member Function Documentation

5.3.3.1 generatePopulation()

```
vector< CGPIndividual > CGP::generatePopulation ()
```

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

Definition at line 14 of file CGP.cpp.

5.3.3.2 goldMutate()

```
vector< CGPIndividual > CGP::goldMutate (
            CGPIndividual parent)
```

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.

Definition at line 165 of file CGP.cpp.

5.3.3.3 pointMutate()

```
vector< CGPIndividual > CGP::pointMutate (
            CGPIndividual parent)
```

Funkcija za kreiranje nove generacije populacije na bazi roditeljske jedinke.
Koristi se **Point Mutacija** kojom se u roditeljskoj jedinci mutira dani broj gena kako bi se kreirala nova jedinka.

Parameters

in	*parent*	Najbolja jedinka iz prosle generacija, roditelj za novu.

Definition at line 95 of file CGP.cpp.

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

- cgp/CGP.h
- cgp/CGP.cpp

5.4 parallel_cgp::CGPIndividual Class Reference

`#include <CGPIndividual.h>`

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, std::vector< int > nodeSet)
- void resolveLoops ()

Static Public Member Functions

- static CGPIndividual deserialize (std::istream &is)

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

Friends

- std::ostream & operator<< (std::ostream &os, const CGPIndividual &ind)
- std::istream & operator>> (std::istream &is, CGPIndividual &ind)

5.4.1 Detailed Description

Klasa koja reprezentira jednog CGP pojedinca.

Definition at line 15 of file CGPIndividual.h.

5.4.2 Constructor & Destructor Documentation

5.4.2.1 CGPIndividual() [1/3]

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

Osnovni kostruktor koji kreira praznu jedinku.

Definition at line 10 of file CGPIndividual.cpp.

5.4.2.2 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.

5.4.2.3 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.

5.4.3 Member Function Documentation

5.4.3.1 deserialize()

```
CGPIndividual CGPIndividual::deserialize (
            std::istream & is)  [static]
```

Staticka metoda za ucitavanje jedinke iz datoteke.

Parameters

in	*is*	Istream za ulaznu datoteku.

Definition at line 93 of file CGPIndividual.cpp.

5.4.3.2 evaluateUsed()

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

Metoda za oznacavanje koristenih gena u mrezi.

Definition at line 49 of file CGPIndividual.cpp.

5.4.3.3 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 ulazniih vrijednosti tipa double.

Definition at line 66 of file CGPIndividual.cpp.

5.4.3.4 findLoops()

```
bool CGPIndividual::findLoops (
            int nodeNum,
            std::vector< int > nodeSet)
```

Rekurzivna funkcija za pronalazak petlji u mrezi.

Parameters

in	*nodeNum*	Broj trenutnog noda.
in	*nodeSet*	Vector za sad prodjenih nodeova.

Returns

True ako je pronadjena petlja, inace false.

Definition at line 121 of file CGPIndividual.cpp.

5.4.3.5 printNodes()

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

Metoda za ispis svih nodova na standardni izlaz.

Definition at line 39 of file CGPIndividual.cpp.

5.4.3.6 resolveLoops()

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

Metoda za razrjesavanje petlji u mrezi.

Definition at line 148 of file CGPIndividual.cpp.

5.4.4 Friends And Related Symbol Documentation

5.4.4.1 operator<<

```
std::ostream & operator<< (
            std::ostream & os,
            const CGPIndividual & ind)  [friend]
```

Operator overloading za pisanje najbolje jedinke u datoteku.

Definition at line 116 of file CGPIndividual.h.

5.4.4.2 operator>>

```
std::istream & operator>> (
            std::istream & is,
            CGPIndividual & ind)  [friend]
```

Operator overloading za citanje najbolje jedinke iz datoteke.

Definition at line 133 of file CGPIndividual.h.

5.4.5 Member Data Documentation

5.4.5.1 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 34 of file CGPIndividual.h.

5.4.5.2 columns

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

Broj stupaca u mrezi.

Definition at line 42 of file CGPIndividual.h.

5.4.5.3 evalDone

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

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

Definition at line 58 of file CGPIndividual.h.

5.4.5.4 genes

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

Vector CGPNode koji reprezentira sve ulazne i gene mreze.

Definition at line 25 of file CGPIndividual.h.

5.4.5.5 inputs

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

Broj ulaznih gena.

Definition at line 50 of file CGPIndividual.h.

5.4.5.6 levelsBack

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

Broj stupaca ispred noda na koje se moze spojiti.

Definition at line 46 of file CGPIndividual.h.

5.4.5.7 outputGene

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

Vector CGPOutput koji reprezentira sve izlazne gene.

Definition at line 29 of file CGPIndividual.h.

5.4.5.8 outputs

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

Broj izlaznih gena.

Definition at line 54 of file CGPIndividual.h.

5.4.5.9 rows

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

Broj redova u mrezi.

Definition at line 38 of file CGPIndividual.h.

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

- cgp/CGPIndividual.h
- cgp/CGPIndividual.cpp

5.5 parallel_cgp::CGPNode Struct Reference

```
#include <CGPNode.h>
```

Public Attributes

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

Friends

- std::ostream & operator<< (std::ostream &os, const CGPNode &node)
- std::istream & operator>> (std::istream &is, CGPNode &node)

5.5.1 Detailed Description

Struktura koja opisuje gene mreze CGP jedinke.

Definition at line 12 of file CGPNode.h.

5.5.2 Friends And Related Symbol Documentation

5.5.2.1 operator<<

```
std::ostream & operator<< (
            std::ostream & os,
            const CGPNode & node)  [friend]
```

Operator overloading za pisanje gena u datoteku.

Definition at line 37 of file CGPNode.h.

5.5.2.2 operator>>

```
std::istream & operator>> (
            std::istream & is,
            CGPNode & node)  [friend]
```

Operator overloading za citanje gena iz datoteke.

Definition at line 44 of file CGPNode.h.

5.5.3 Member Data Documentation

5.5.3.1 connection1

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

Prva konekcija nodea na drugi node.

Definition at line 20 of file CGPNode.h.

5.5.3.2 connection2

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

Druga konekcija nodea na drugi node.

Definition at line 24 of file CGPNode.h.

5.5.3.3 operand

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

Vrijednost koja oznacava koji se operand koristi u nodeu.

Definition at line 16 of file CGPNode.h.

5.5.3.4 outValue

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

Izlazna vrijednost nakon racunanja vrijednosti.

Definition at line 32 of file CGPNode.h.

5.5.3.5 used

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

Vrijednost koja oznacava koristi li se node.

Definition at line 28 of file CGPNode.h.

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

- cgp/CGPNode.h

5.6 parallel_cgp::CGPOutput Struct Reference

```
#include <CGPOutput.h>
```

Public Attributes

- int connection
- TYPE value

Friends

- std::ostream & operator<< (std::ostream &os, const CGPOutput &output)
- std::istream & operator>> (std::istream &is, CGPOutput &output)

5.6.1 Detailed Description

Struktura koja opisuje izlazne gene CGP jedinke.

Definition at line 12 of file CGPOutput.h.

5.6.2 Friends And Related Symbol Documentation

5.6.2.1 operator<<

```
std::ostream & operator<< (
            std::ostream & os,
            const CGPOutput & output)  [friend]
```

Operator overloading za pisanje izlaznog gena u datoteku.

Definition at line 25 of file CGPOutput.h.

5.6.2.2 operator>>

```
std::istream & operator>> (
            std::istream & is,
            CGPOutput & output)  [friend]
```

Operator overloading za citanje izlaznog gena iz datoteke.

Definition at line 32 of file CGPOutput.h.

5.6.3 Member Data Documentation

5.6.3.1 connection

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

Broj koji reprezentira na koji gen je spojen izlazni gen.

Definition at line 16 of file CGPOutput.h.

5.6.3.2 value

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

Izlazna vrijednost gena nakon izracuna.

Definition at line 20 of file CGPOutput.h.

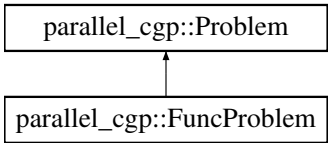
The documentation for this struct was generated from the following file:

- cgp/CGPOutput.h

5.7 parallel_cgp::FuncProblem Class Reference

```
#include <FuncProblem.h>
```

Inheritance diagram for parallel_cgp::FuncProblem:



Public Member Functions

- FuncProblem ()
- FuncProblem (int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int MUTATIONS, int POPULATION_SIZE)
- void problemRunner () override
- void printFunction () override

Public Member Functions inherited from parallel_cgp::Problem

- virtual TYPE fitness (TYPE fit)

Additional Inherited Members

Public Attributes inherited from parallel_cgp::Problem

- std::string bestFile = "problem_best.txt"
- int NUM_OPERANDS = 9
- int BI_OPERANDS = 5
- int GENERATIONS = 5000
- int ROWS = 20
- int COLUMNS = 20
- int LEVELS_BACK = 3
- int INPUTS = 6
- int OUTPUTS = 1
- int MUTATIONS = 6
- int POPULATION_SIZE = 20

5.7.1 Detailed Description

Klasa koja opisuje problem pronalaska funkcije.

Definition at line 14 of file FuncProblem.h.

5.7.2 Constructor & Destructor Documentation

5.7.2.1 FuncProblem() [1/2]

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

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

Definition at line 65 of file FuncProblem.h.

5.7.2.2 FuncProblem() [2/2]

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

Konstruktor koji prima sve promjenjive vrijednosti za func problem.

Definition at line 69 of file FuncProblem.h.

5.7.3 Member Function Documentation

5.7.3.1 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.

5.7.3.2 problemRunner()

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

Metoda za pokretanje problema.

Implements parallel_cgp::Problem.

Definition at line 111 of file FuncProblem.cpp.

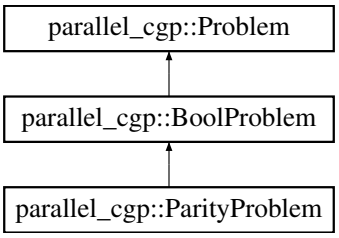
The documentation for this class was generated from the following files:

- funcProblem/FuncProblem.h
- funcProblem/FuncProblem.cpp

5.8 parallel_cgp::ParityProblem Class Reference

`#include <BoolProblem.h>`

Inheritance diagram for parallel_cgp::ParityProblem:



Public Member Functions

- ParityProblem ()

Public Member Functions inherited from parallel_cgp::BoolProblem

- BoolProblem ()
- BoolProblem (int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int MUTATIONS, int POPULATION_SIZE)
- void problemRunner () override
- void printFunction () override

Public Member Functions inherited from parallel_cgp::Problem

- virtual TYPE fitness (TYPE fit)

Additional Inherited Members

Public Attributes inherited from parallel_cgp::Problem

- std::string bestFile = "problem_best.txt"
- int NUM_OPERANDS = 9
- int BI_OPERANDS = 5
- int GENERATIONS = 5000
- int ROWS = 20
- int COLUMNS = 20
- int LEVELS_BACK = 3
- int INPUTS = 6
- int OUTPUTS = 1
- int MUTATIONS = 6
- 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

- CGPIndividual bestI
- const std::string bestFile = "bool_best.txt"
- int GENERATIONS = 5000
- int ROWS = 100
- int COLUMNS = 20
- int LEVELS_BACK = 0
- int MUTATIONS = 0
- int POPULATION_SIZE = 20
- bool isSimulated = false
- bool useFunc = true
- const std::function< int(std::bitset< INPUTS > in)> boolFunc
- const 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

5.8.1 Detailed Description

Klasa koja opisuje problema pariteta.

Definition at line 96 of file BoolProblem.h.

5.8.2 Constructor & Destructor Documentation

5.8.2.1 ParityProblem()

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

Konstruktor koji samo mijenja koja se funkcija koristi.

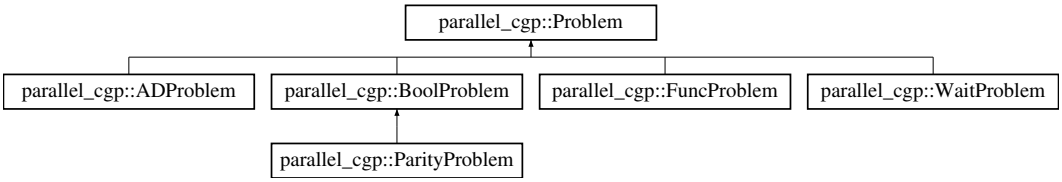
Definition at line 101 of file BoolProblem.h.

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

- boolProblem/BoolProblem.h

5.9 parallel_cgp::Problem Class Reference

Inheritance diagram for parallel_cgp::Problem:



Public Member Functions

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

Public Attributes

- std::string bestFile = "problem_best.txt"
- int NUM_OPERANDS = 9
- int BI_OPERANDS = 5
- int GENERATIONS = 5000
- int ROWS = 20
- int COLUMNS = 20
- int LEVELS_BACK = 3
- int INPUTS = 6
- int OUTPUTS = 1
- int MUTATIONS = 6
- int POPULATION_SIZE = 20

5.9.1 Detailed Description

Definition at line 10 of file Problem.h.

5.9.2 Member Function Documentation

5.9.2.1 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 50 of file Problem.h.

5.9.2.2 fitness()

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

Funkcija koja se koristi za izracun fitnessa za odredenu jedinku.

Definition at line 77 of file Problem.h.

5.9.2.3 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.

5.9.2.4 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.

5.9.3 Member Data Documentation

5.9.3.1 bestFile

```
std::string parallel_cgp::Problem::bestFile = "problem_best.txt"
```

Naziv datoteke koja sadrzi najbolju jedinku.

Definition at line 27 of file Problem.h.

5.9.3.2 BI_OPERANDS

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

Definition at line 34 of file Problem.h.

5.9.3.3 COLUMNS

```
int parallel_cgp::Problem::COLUMNS = 20
```

Definition at line 37 of file Problem.h.

5.9.3.4 GENERATIONS

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

Definition at line 35 of file Problem.h.

5.9.3.5 INPUTS

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

Definition at line 39 of file Problem.h.

5.9.3.6 LEVELS_BACK

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

Definition at line 38 of file Problem.h.

5.9.3.7 MUTATIONS

```
int parallel_cgp::Problem::MUTATIONS = 6
```

Definition at line 41 of file Problem.h.

5.9.3.8 NUM_OPERANDS

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

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

Definition at line 33 of file Problem.h.

5.9.3.9 OUTPUTS

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

Definition at line 40 of file Problem.h.

5.9.3.10 POPULATION_SIZE

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

Definition at line 42 of file Problem.h.

5.9.3.11 ROWS

```
int parallel_cgp::Problem::ROWS = 20
```

Definition at line 36 of file Problem.h.

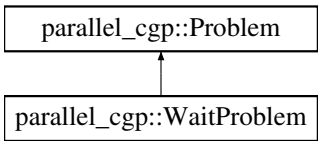
The documentation for this class was generated from the following file:

- Problem.h

5.10 parallel_cgp::WaitProblem Class Reference

```
#include <WaitProblem.h>
```

Inheritance diagram for parallel_cgp::WaitProblem:



Public Member Functions

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

Public Member Functions inherited from parallel_cgp::Problem

- virtual TYPE computeNode (int operand, TYPE value1, TYPE value2)

Additional Inherited Members

Public Attributes inherited from parallel_cgp::Problem

- std::string bestFile = "problem_best.txt"
- int NUM_OPERANDS = 9
- int BI_OPERANDS = 5
- int GENERATIONS = 5000
- int ROWS = 20
- int COLUMNS = 20
- int LEVELS_BACK = 3
- int INPUTS = 6
- int OUTPUTS = 1
- int MUTATIONS = 6
- int POPULATION_SIZE = 20

5.10.1 Detailed Description

Klasa koja opisuje problem koji ceka odredeno vrijeme.

Definition at line 16 of file WaitProblem.h.

5.10.2 Constructor & Destructor Documentation

5.10.2.1 WaitProblem() [1/2]

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

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

Definition at line 63 of file WaitProblem.h.

5.10.2.2 WaitProblem() [2/2]

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

Konstruktor koji prima sve promjenjive vrijednosti za wait problem.

Definition at line 67 of file WaitProblem.h.

5.10.3 Member Function Documentation

5.10.3.1 printFunction()

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

Metoda za ispis na kraju dobivene funkcije.

Implements parallel_cgp::Problem.

Definition at line 11 of file WaitProblem.cpp.

5.10.3.2 problemRunner()

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

Metoda za pokretanje problema.

Implements parallel_cgp::Problem.

Definition at line 42 of file WaitProblem.cpp.

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

- waitProblem/WaitProblem.h
- waitProblem/WaitProblem.cpp

Chapter 6

File Documentation

6.1 ADProblem.cpp

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

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

```
00141         vector<int> bestInds;
00142         random_device rd;
00143         mt19937 gen(rd());
00144 
00145         for (int clan = 0; clan < POPULATION_SIZE; clan++) {
00146 
00147             double fit = 0;
00148             problemSimulator(population[clan], fit);
00149 
00150             if (fit > bestFit) {
00151                 bestFit = fit;
00152                 bestInds.clear();
00153                 bestInds.push_back(clan);
00154             }
00155             else if (fit == bestFit)
00156                 bestInds.push_back(clan);
00157         }
00158 
00159         if (bestInds.size() > 1)
00160             bestInds.erase(bestInds.begin());
00161 
00162         uniform_int_distribution<> bestDis(0, static_cast<int>(bestInds.size() - 1));
00163 
00164         bestInd = bestInds[bestDis(gen)];
00165 
00166         cout « "Gen: " « generacija « "; Fitness: " « bestFit « "; Indeks: " « bestInd « endl;
00167 
00168         if (bestFit >= STARTING_CASH * 3)
00169             break;
00170         if (generacija != GENERATIONS - 1)
00171             population = cgp.goldMutate(population[bestInd]);
00172     }
00173 
00174     bestI = population[bestInd];
00175 
00176     isSimulated = true;
00177 
00178     printFunction();
00179 
00180     playGame();
00181 }
00182 
00183 void ADProblem::playGame() {
00184     function<double(int op, double v1, double v2)> compNode =
00185         [&](int op, double v1, double v2) { return computeNode(op, static_cast<TYPE>(v1),
      static_cast<TYPE>(v2)); };
00186 
00187     random_device rd;
00188     mt19937 gen(rd());
00189 
00190     uniform_int_distribution<> cardDis(1, 13);
00191 
00192     int steps = 0;
00193     int cash = STARTING_CASH, maxCash = STARTING_CASH;
00194 
00195     while (cash && steps < 100 && cash < MAX_CASH) {
00196         vector<double> input;
00197         int card, win;
00198         for (int i = 0; i < 3; i++)
00199             input.push_back(static_cast<TYPE>(cardDis(gen)));
00200 
00201         card = input.back();
00202         input.pop_back();
00203 
00204         sort(input.begin(), input.end());
00205 
00206         if (card > input.at(0) && card < input.at(1))
00207             win = 1;
00208         else if (card == input.at(0) || card == input.at(1))
00209             win = -1;
00210         else
00211             win = 0;
00212 
00213         bestI.evaluateValue(input, compNode);
00214 
00215         cout « "Cash: " « cash « "; Cards: " « input[0] « ", " « input[1] « "; Bet: " «
      ((bestI.outputGene[0].value > 1) ? "YES" : "NO")
00216             « "; Third card: " « card « ((win == 1) ? " | WIN!" : " | LOST!") « endl;
00217 
00218         if (bestI.outputGene[0].value > 1) {
00219             if (win == 1)
00220                 cash += 10;
00221             else if (win == 0)
00222                 cash -= 10;
00223             else if (win == -1)
00224                 cash -= 20;
00225         }
```

```
00226
00227         if (cash > maxCash)
00228             maxCash = cash;
00229
00230         steps++;
00231     }
00232 }
```

6.2 ADProblem.h

```
00001 #ifndef ADPROBLEM_H
00002 #define ADPROBLEM_H
00003
00004 #include "../Problem.h"
00005 #include "../cgp/CGP.h"
00006
00007 #undef TYPE
00008 #define TYPE int
00009
00010 namespace parallel_cgp {
00014     class ADProblem : public Problem {
00015     private:
00019         CGPIndividual bestI;
00023         const std::string bestFile = "ad_best.txt";
00024
00030         const static int NUM_OPERANDS = 4;
00031         const static int BI_OPERANDS = 4;
00032         const static int INPUTS = 2;
00033         const static int OUTPUTS = 1;
00034         const static int MAX_CASH = 1000;
00035         const static int STARTING_CASH = 100;
00036         double CARD_SETS = 500;
00037
00042         int GENERATIONS = 5000;
00043         int ROWS = 8;
00044         int COLUMNS = 8;
00045         int LEVELS_BACK = 1;
00046         int MUTATIONS = 0;
00047         int POPULATION_SIZE = 20;
00048
00052         std::vector<std::vector<double» sets;
00053
00057         bool isSimulated = false;
00058
00059         TYPE computeNode(int operand, TYPE value1, TYPE value2);
00060         double fitness(TYPE cash, TYPE maxCash, double avgCash);
00061         void problemSimulator(parallel_cgp::CGPIndividual& individual, double& fit) override;
00062         std::string evalFunction(int CGPNodeNum) override;
00063     public:
00067         void problemRunner() override;
00071         void printFunction() override;
00075         void playGame();
00076     };
00077 }
00078
00079 #endif
```

6.3 BoolProblem.cpp

```
00001 #include "BoolProblem.h"
00002
00003 using namespace std;
00004 using namespace parallel_cgp;
00005
00006 TYPE BoolProblem::computeNode(int operand, TYPE value1, TYPE value2) {
00007     switch (operand) {
00008     case 1:
00009         return value1 | value2;
00010     case 2:
00011         return value1 & value2;
00012     case 3:
00013         return value1 ^ value2;
00014     case 4:
00015         return ~value1;
00016     default:
00017         return 0;
00018     }
00019 }
```

```
00020
00021 TYPE BoolProblem::fitness(bitset<INPUTS> in, TYPE res) {
00022     if (useFunc)
00023         return boolFunc(in) == res;
00024
00025     return parityFunc(in) == res;
00026 }
00027
00028 void BoolProblem::printFunction() {
00029     if (isSimulated)
00030         cout « "Funkcija: " « evalFunction(bestI.outputGene[0].connection) « endl;
00031     else
00032         cout « "Problem nije simuliran." « endl;
00033 }
00034
00035 string BoolProblem::evalFunction(int CGPNodeNum) {
00036     ostringstream oss;
00037
00038     if (CGPNodeNum < INPUTS) {
00039         oss « "bit[" « CGPNodeNum « "]";
00040         return oss.str();
00041     }
00042
00043     switch (bestI.genes[CGPNodeNum].operand) {
00044     case 1:
00045         oss « "(" « evalFunction(bestI.genes[CGPNodeNum].connection1) « " | " «
    evalFunction(bestI.genes[CGPNodeNum].connection2) « ")";
00046         return oss.str();
00047     case 2:
00048         oss « "(" « evalFunction(bestI.genes[CGPNodeNum].connection1) « " & " «
    evalFunction(bestI.genes[CGPNodeNum].connection2) « ")";
00049         return oss.str();
00050     case 3:
00051         oss « "(" « evalFunction(bestI.genes[CGPNodeNum].connection1) « " ^ " «
    evalFunction(bestI.genes[CGPNodeNum].connection2) « ")";
00052         return oss.str();
00053     case 4:
00054         oss « "~" « evalFunction(bestI.genes[CGPNodeNum].connection1);
00055         return oss.str();
00056     }
00057
00058     return "";
00059 }
00060
00061 void BoolProblem::problemSimulator(CGPIndividual& individual, TYPE &fit) {
00062     function<double(int op, double v1, double v2)> compNode =
00063         [&](int op, double v1, double v2) { return computeNode(op, static_cast<TYPE>(v1),
    static_cast<TYPE>(v2)); };
00064
00065     for (int perm = 0; perm < pow(2, INPUTS); ++perm) {
00066         bitset<INPUTS> bits(perm);
00067         vector<double> input;
00068
00069         for (int i = 0; i < bits.size(); ++i)
00070             input.push_back(static_cast<double>(bits[i]));
00071
00072         individual.evaluateValue(input, compNode);
00073         fit += fitness(bits, static_cast<int>(individual.outputGene[0].value));
00074     }
00075 }
00076
00077 void BoolProblem::problemRunner() {
00078     CGP cgp(GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, INPUTS, OUTPUTS, MUTATIONS, NUM_OPERANDS,
    BI_OPERANDS, POPULATION_SIZE);
00079
00080     vector<CGPIndividual> population;
00081     int bestInd = 0, generacija = 0;
00082
00083     population = cgp.generatePopulation();
00084
00085     for (generacija = 0; generacija < GENERATIONS; generacija++) {
00086         TYPE bestFit = -1;
00087         bestInd = 0;
00088         vector<int> bestInds;
00089         random_device rd;
00090         mt19937 gen(rd());
00091
00092         for (int clan = 0; clan < POPULATION_SIZE; clan++) {
00093
00094             TYPE fit = 0;
00095             problemSimulator(population[clan], fit);
00096
00097             if (fit > bestFit) {
00098                 bestFit = fit;
00099                 bestInds.clear();
00100                 bestInds.push_back(clan);
00101             }
```

```
00102             else if (fit == bestFit)
00103                 bestInds.push_back(clan);
00104         }
00105
00106         if (bestInds.size() > 1)
00107             bestInds.erase(bestInds.begin());
00108
00109         uniform_int_distribution<> bestDis(0, static_cast<int>(bestInds.size() - 1));
00110
00111         bestInd = bestInds[bestDis(gen)];
00112
00113         cout « "Gen: " « generacija « "; Fitness: " « bestFit « "; Indeks: " « bestInd « endl;
00114
00115         if (bestFit == pow(2, INPUTS))
00116             break;
00117         if (generacija != GENERATIONS - 1)
00118             population = cgp.goldMutate(population[bestInd]);
00119     }
00120
00121     bestI = population[bestInd];
00122
00123     isSimulated = true;
00124
00125     printFunction();
00126 }
```

6.4 BoolProblem.h

```
00001 #ifndef BOOLPROBLEM_H
00002 #define BOOLPROBLEM_H
00003
00004 #include "../Problem.h"
00005 #include "../cgp/CGP.h"
00006 #include <bitset>
00007
00008 #undef TYPE
00009 #define TYPE int
00010
00011 namespace parallel_cgp {
00016     class BoolProblem : public Problem {
00017     protected:
00021         CGPIndividual bestI;
00025         const std::string bestFile = "bool_best.txt";
00026
00032         const static int NUM_OPERANDS = 4;
00033         const static int BI_OPERANDS = 4;
00034         const static int INPUTS = 7;
00035         const static int OUTPUTS = 1;
00036
00041         int GENERATIONS = 5000;
00042         int ROWS = 100;
00043         int COLUMNS = 20;
00044         int LEVELS_BACK = 0;
00045         int MUTATIONS = 0;
00046         int POPULATION_SIZE = 20;
00047
00051         bool isSimulated = false;
00055         bool useFunc = true;
00056
00060         const std::function<int(std::bitset<INPUTS> in)> boolFunc =
00061             [](std::bitset<INPUTS> in) { return (in[0] | ~in[1]) & (in[0] ^ in[4] | (in[3] & ~in[2]));
      };
00065         const std::function<int(std::bitset<INPUTS> in)> parityFunc =
00066             [](std::bitset<INPUTS> in) { return (in.count() % 2 == 0) ? 0 : 1; };
00067
00068         TYPE computeNode(int operand, TYPE value1, TYPE value2);
00069         TYPE fitness(std::bitset<INPUTS> input, TYPE res);
00070         void problemSimulator(CGPIndividual &individual, TYPE &fit);
00071         std::string evalFunction(int CGPNodeNum) override;
00072     public:
00076         BoolProblem() {};
00080         BoolProblem(int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int MUTATIONS, int
      POPULATION_SIZE)
00081             : GENERATIONS(GENERATIONS), ROWS(ROWS), COLUMNS(COLUMNS), LEVELS_BACK(LEVELS_BACK),
      MUTATIONS(MUTATIONS), POPULATION_SIZE(POPULATION_SIZE) {};
00082
00086         void problemRunner() override;
00090         void printFunction() override;
00091     };
00092
00096     class ParityProblem : public BoolProblem {
00097     public:
00101         ParityProblem() { useFunc = false; };
```

```
00102     };
00103 }
00104
00105 #endif
```

6.5 CGP.cpp

```
00001 #include "CGP.h"
00002 #include <iostream>
00003 #include <chrono>
00004 #include <thread>
00005 #include <cmath>
00006 #include <random>
00007 #include <fstream>
00008 #include <string>
00009 #include <sstream>
00010
00011 using namespace std;
00012 using namespace parallel_cgp;
00013
00014 vector<CGPIndividual> CGP::generatePopulation() {
00015     vector<CGPIndividual> population;
00016
00017     for (int i = 0; i < populationSize; i++) {
00018         random_device rd;
00019         mt19937 gen(rd());
00020
00021         uniform_int_distribution<> operandDis(1, operands);
00022         uniform_int_distribution<> connectionDis(0, rows * columns + inputs - 1);
00023         uniform_int_distribution<> outputDis(0, rows * columns + inputs - 1);
00024
00025         vector<CGPNode> genes;
00026         vector<CGPOutput> outputGene;
00027
00028         for (size_t k = 0; k < inputs; k++) {
00029             CGPNode node;
00030             node.used = false;
00031             node.connection1 = -1;
00032             node.connection2 = -1;
00033             node.operand = -1;
00034             genes.push_back(node);
00035         }
00036
00037         for (int j = inputs; j < rows * columns + inputs; j++) {
00038             CGPNode node;
00039             node.used = false;
00040             node.operand = operandDis(gen);
00041             node.connection1 = connectionDis(gen);
00042             node.outValue = NAN;
00043
00044             while (true) {
00045                 if (node.connection1 < inputs)
00046                     break;
00047                 if ((node.connection1 % columns) == (j % columns))
00048                     node.connection1 = connectionDis(gen);
00049                 else if (((node.connection1 - inputs) % columns) > (((j - inputs) % columns) +
    levelsBack))
00050                     node.connection1 = connectionDis(gen);
00051                 else if(genes.size() > node.connection1 && (genes[node.connection1].connection1 == j
    || genes[node.connection1].connection2 == j))
00052                     node.connection1 = connectionDis(gen);
00053                 else
00054                     break;
00055             }
00056
00057             node.connection2 = (node.operand >= biOperands) ? -1 : connectionDis(gen);
00058
00059             while (true) {
00060                 if (node.connection2 < inputs)
00061                     break;
00062                 if ((node.connection2 % columns) == (j % columns))
00063                     node.connection2 = connectionDis(gen);
00064                 else if (((node.connection2 - inputs) % columns) > (((j - inputs) % columns) +
    levelsBack))
00065                     node.connection2 = connectionDis(gen);
00066                 else if (genes.size() > node.connection2 && (genes[node.connection2].connection1 == j
    || genes[node.connection2].connection2 == j))
00067                     node.connection2 = connectionDis(gen);
00068                 else
00069                     break;
00070             }
00071
00072             genes.push_back(node);
```

```
00073         }
00074
00075         for (size_t k = 0; k < outputs; k++) {
00076             CGPOutput output;
00077
00078             output.connection = outputDis(gen);
00079
00080             outputGene.push_back(output);
00081         }
00082
00083         CGPIndividual individual(genes, outputGene, rows, columns, levelsBack, inputs, outputs);
00084         individual.resolveLoops();
00085         population.push_back(individual);
00086
00087         cout « "|";
00088     }
00089     cout « endl;
00090
00091     return population;
00092 }
00093
00094 // point mutacija
00095 vector<CGPIndividual> CGP::pointMutate(CGPIndividual parent) {
00096     vector<CGPIndividual> population;
00097     if (!parent.evalDone)
00098         parent.evaluateUsed();
00099     population.push_back(parent);
00100
00101     random_device rd;
00102     mt19937 gen(rd());
00103
00104     uniform_int_distribution<> nodDis(parent.inputs, static_cast<int>(parent.genes.size()));
00105     uniform_int_distribution<> geneDis(0, 2);
00106     uniform_int_distribution<> connectionDis(0, static_cast<int>(parent.genes.size()) - 1);
00107     uniform_int_distribution<> operandDis(1, operands);
00108     uniform_int_distribution<> outputDis(0, parent.outputs - 1);
00109
00110     for (int n = 0; n < populationSize - 1; n++) {
00111         vector<CGPNode> genes = parent.genes;
00112         vector<CGPOutput> outputGene = parent.outputGene;
00113
00114         for (int z = parent.inputs; z < genes.size(); z++)
00115             genes[z].used = false;
00116
00117         for (int i = 0; i < mutations; i++) {
00118             int mut = geneDis(gen);
00119             int cell = nodDis(gen);
00120             if (cell == parent.genes.size()) {
00121                 outputGene[outputDis(gen)].connection = connectionDis(gen);
00122                 continue;
00123             }
00124             if (mut == 0)
00125                 genes[cell].operand = operandDis(gen);
00126             else if (mut == 1)
00127                 genes[cell].connection1 = connectionDis(gen);
00128             else if (mut == 2)
00129                 genes[cell].connection2 = connectionDis(gen);
00130
00131             genes[cell].connection2 = (genes[cell].operand >= biOperands) ? -1 : connectionDis(gen);
00132
00133             while (true) {
00134                 if (genes[cell].connection1 < parent.inputs)
00135                     break;
00136                 if ((genes[cell].connection1 % parent.columns) == (cell % parent.columns))
00137                     genes[cell].connection1 = connectionDis(gen);
00138                 else if (((genes[cell].connection1 - parent.inputs) % parent.columns) > (((cell -
      parent.inputs) % parent.columns) + parent.levelsBack))
00139                     genes[cell].connection1 = connectionDis(gen);
00140                 else
00141                     break;
00142             }
00143
00144             while (true) {
00145                 if (genes[cell].connection2 < parent.inputs)
00146                     break;
00147                 if ((genes[cell].connection2 % parent.columns) == (cell % parent.columns))
00148                     genes[cell].connection2 = connectionDis(gen);
00149                 else if (((genes[cell].connection2 - parent.inputs) % parent.columns) > (((cell -
      parent.inputs) % parent.columns) + parent.levelsBack))
00150                     genes[cell].connection2 = connectionDis(gen);
00151                 else
00152                     break;
00153             }
00154         }
00155
00156         CGPIndividual individual(genes, outputGene, parent.rows, parent.columns, parent.levelsBack,
      parent.inputs, parent.outputs);
```

```
00157         individual.resolveLoops();
00158         population.push_back(individual);
00159     }
00160 
00161     return population;
00162 }
00163 
00164 // goldman mutacija
00165 vector<CGPIndividual> CGP::goldMutate(CGPIndividual parent) {
00166     vector<CGPIndividual> population;
00167     if (!parent.evalDone)
00168         parent.evaluateUsed();
00169     population.push_back(parent);
00170 
00171     random_device rd;
00172     mt19937 gen(rd());
00173 
00174     uniform_int_distribution<> nodDis(parent.inputs, static_cast<int>(parent.genes.size()));
00175     uniform_int_distribution<> geneDis(0, 2);
00176     uniform_int_distribution<> connectionDis(0, static_cast<int>(parent.genes.size()) - 1);
00177     uniform_int_distribution<> operandDis(1, operands);
00178     uniform_int_distribution<> outputDis(0, parent.outputs - 1);
00179 
00180     #pragma omp parallel for
00181     for (int n = 0; n < populationSize - 1; n++) {
00182         vector<CGPNode> genes = parent.genes;
00183         vector<CGPOutput> outputGene = parent.outputGene;
00184         bool isActive = false;
00185 
00186         while (!isActive) {
00187             int mut = geneDis(gen);
00188             int cell = nodDis(gen);
00189             if (cell == parent.genes.size()) {
00190                 outputGene[outputDis(gen)].connection = connectionDis(gen);
00191                 break;
00192             }
00193             if (mut == 0) {
00194                 genes[cell].operand = operandDis(gen);
00195 
00196                 if (genes[cell].operand >= biOperands && genes[cell].connection2 != -1)
00197                     genes[cell].connection2 = -1;
00198                 else if (genes[cell].operand < biOperands && genes[cell].connection2 == -1)
00199                     genes[cell].connection2 = connectionDis(gen);
00200             }
00201             else if (mut == 1)
00202                 genes[cell].connection1 = connectionDis(gen);
00203             else if (mut == 2 && genes[cell].operand >= biOperands)
00204                 continue;
00205             else if (mut == 2)
00206                 genes[cell].connection2 = connectionDis(gen);
00207 
00208             while (true) {
00209                 if (genes[cell].connection1 < parent.inputs)
00210                     break;
00211                 if ((genes[cell].connection1 % parent.columns) == (cell % parent.columns))
00212                     genes[cell].connection1 = connectionDis(gen);
00213                 else if (((genes[cell].connection1 - parent.inputs) % parent.columns) > (((cell -
    parent.inputs) % parent.columns) + parent.levelsBack))
00214                     genes[cell].connection1 = connectionDis(gen);
00215                 else
00216                     break;
00217             }
00218 
00219             while (true) {
00220                 if (genes[cell].connection2 < parent.inputs)
00221                     break;
00222                 if ((genes[cell].connection2 % parent.columns) == (cell % parent.columns))
00223                     genes[cell].connection2 = connectionDis(gen);
00224                 else if (((genes[cell].connection2 - parent.inputs) % parent.columns) > (((cell -
    parent.inputs) % parent.columns) + parent.levelsBack))
00225                     genes[cell].connection2 = connectionDis(gen);
00226                 else
00227                     break;
00228             }
00229 
00230             isActive = genes[cell].used;
00231         }
00232 
00233         for (int z = parent.inputs; z < genes.size(); z++)
00234             genes[z].used = false;
00235 
00236         CGPIndividual individual(genes, outputGene, parent.rows, parent.columns, parent.levelsBack,
    parent.inputs, parent.outputs);
00237         individual.resolveLoops();
00238 
00239         #pragma omp critical
00240         population.push_back(individual);
```

```
00241     }
00242
00243     return population;
00244
00245 }
```

6.6 CGP.h

```
00001 #ifndef CGP_H
00002 #define CGP_H
00003 #define TYPE double
00004
00005 #include <vector>
00006 #include <string>
00007 #include "CGPIndividual.h"
00008
00009 namespace parallel_cgp {
00013     class CGP {
00014     private:
00015         int generations, rows, columns, levelsBack, inputs, outputs, mutations, operands, biOperands,
      populationSize;
00016     public:
00030         CGP(int generations, int rows, int columns, int levelsBack, int inputs, int outputs, int
      mutations, int operands, int biOperands, int populationSize)
00031             : generations(generations), rows(rows), columns(columns), levelsBack(levelsBack),
      inputs(inputs), outputs(outputs), mutations(mutations),
00032               operands(operands), biOperands(biOperands), populationSize(populationSize) {};
00033
00039         std::vector<CGPIndividual> generatePopulation();
00040
00047         std::vector<CGPIndividual> pointMutate(CGPIndividual parent);
00048
00056         std::vector<CGPIndividual> goldMutate(CGPIndividual parent);
00057     };
00058 }
00059
00060 #endif
```

6.7 CGPIndividual.cpp

```
00001 #include "CGPIndividual.h"
00002 #include <iostream>
00003 #include <chrono>
00004 #include <thread>
00005 #include <random>
00006
00007 using namespace std;
00008 using namespace parallel_cgp;
00009
00010 CGPIndividual::CGPIndividual() {
00011 }
00012
00013 CGPIndividual::CGPIndividual(vector<CGPNode> genes, vector<CGPOutput> outputGene, int rows, int
      columns, int levelsBack, int inputs, int outputs) {
00014     vector<vector<int» branches;
00015     this->branches = branches;
00016     this->genes = genes;
00017     this->outputGene = outputGene;
00018     this->rows = rows;
00019     this->columns = columns;
00020     this->levelsBack = levelsBack;
00021     this->inputs = inputs;
00022     this->outputs = outputs;
00023     this->evalDone = false;
00024 }
00025
00026 CGPIndividual::CGPIndividual(vector<CGPNode> genes, vector<CGPOutput> outputGene, int rows, int
      columns, int levelsBack, int inputs, int outputs, bool evalDone) {
00027     vector<vector<int» branches;
00028     this->branches = branches;
00029     this->genes = genes;
00030     this->outputGene = outputGene;
00031     this->rows = rows;
00032     this->columns = columns;
00033     this->levelsBack = levelsBack;
00034     this->inputs = inputs;
00035     this->outputs = outputs;
00036     this->evalDone = evalDone;
```

```
00037 }
00038
00039 void CGPIndividual::printNodes() {
00040     for (size_t i = 0; i < rows * columns + inputs; i++)
00041         cout « i « " " « genes[i].operand « " " « genes[i].connection1 « " " « genes[i].connection2 «
      endl;
00042
00043     for (size_t j = 0; j < outputs; j++)
00044         cout « outputGene[j].connection « " ";
00045
00046     cout « endl « endl;
00047 }
00048
00049 void CGPIndividual::evaluateUsed() {
00050     for (int m = 0; m < outputs; m++)
00051         isUsed(outputGene[m].connection);
00052
00053     evalDone = true;
00054 }
00055
00056 void CGPIndividual::isUsed(int CGPNodeNum) {
00057     genes[CGPNodeNum].used = true;
00058
00059     if (genes[CGPNodeNum].connection1 >= 0)
00060         isUsed(genes[CGPNodeNum].connection1);
00061
00062     if (genes[CGPNodeNum].connection2 >= 0)
00063         isUsed(genes[CGPNodeNum].connection2);
00064 }
00065
00066 void CGPIndividual::evaluateValue(vector<TYPE> input, function<TYPE(int, TYPE, TYPE)> computeNode) {
00067     clearInd();
00068
00069     for (int l = 0; l < inputs; l++)
00070         genes[l].outValue = input[l];
00071
00072     for (int m = 0; m < outputs; m++)
00073         outputGene[m].value = evalNode(outputGene[m].connection, computeNode);
00074 }
00075
00076 TYPE CGPIndividual::evalNode(int CGPNodeNum, function<TYPE(int, TYPE, TYPE)> computeNode) {
00077
00078     if (isnan(genes[CGPNodeNum].outValue)) {
00079         TYPE value1 = evalNode(genes[CGPNodeNum].connection1, computeNode);
00080         TYPE value2 = genes[CGPNodeNum].connection2 < 0 ? 0 : evalNode(genes[CGPNodeNum].connection2,
      computeNode);
00081
00082         genes[CGPNodeNum].outValue = computeNode(genes[CGPNodeNum].operand, value1, value2);
00083     }
00084
00085     return genes[CGPNodeNum].outValue;
00086 }
00087
00088 void CGPIndividual::clearInd() {
00089     for (int i = inputs; i < genes.size(); i++)
00090         genes[i].outValue = NAN;
00091 }
00092
00093 CGPIndividual CGPIndividual::deserialize(istream& is) {
00094     int rows, columns, levelsBack, inputs, outputs, evalDone;
00095
00096     is » rows » columns » levelsBack » inputs » outputs » evalDone;
00097
00098     size_t genesSize;
00099     is » genesSize;
00100     vector<CGPNode> genes;
00101     genes.reserve(genesSize);
00102     for (size_t i = 0; i < genesSize; ++i) {
00103         CGPNode gene;
00104         is » gene;
00105         genes.emplace_back(gene);
00106     }
00107
00108     size_t outputGeneSize;
00109     is » outputGeneSize;
00110     vector<CGPOutput> outputGene;
00111     outputGene.reserve(outputGeneSize);
00112     for (size_t i = 0; i < outputGeneSize; ++i) {
00113         CGPOutput outGene;
00114         is » outGene;
00115         outputGene.emplace_back(outGene);
00116     }
00117
00118     return CGPIndividual(move(genes), move(outputGene), rows, columns, levelsBack, inputs, outputs,
      evalDone);
00119 }
00120
```

```
00121 bool CGPIndividual::findLoops(int CGPNodeNum, vector<int> CGPNodeSet) {
00122     branches.clear();
00123 
00124     return loopFinder(CGPNodeNum, CGPNodeSet);;
00125 }
00126 
00127 bool CGPIndividual::loopFinder(int CGPNodeNum, vector<int> CGPNodeSet) {
00128 
00129     for (int i = 0; i < CGPNodeSet.size(); i++)
00130         if (CGPNodeSet[i] == CGPNodeNum) {
00131             CGPNodeSet.push_back(CGPNodeNum);
00132             branches.push_back(CGPNodeSet);
00133             return true;
00134         }
00135 
00136     CGPNodeSet.push_back(CGPNodeNum);
00137 
00138     if (CGPNodeNum < inputs) {
00139         return false;
00140     }
00141 
00142     bool conn1 = loopFinder(genes[CGPNodeNum].connection1, CGPNodeSet);
00143     bool conn2 = genes[CGPNodeNum].connection2 == -1 ? false :
      loopFinder(genes[CGPNodeNum].connection2, CGPNodeSet);
00144 
00145     return conn1 || conn2;
00146 }
00147 
00148 void CGPIndividual::resolveLoops() {
00149 
00150     random_device rd;
00151     mt19937 gen(rd());
00152     uniform_int_distribution<> connectionDis(0, static_cast<int>(genes.size()) - 1);
00153 
00154     vector<int> CGPNodeSet;
00155 
00156     for (int m = 0; m < outputs; m++) {
00157         while (findLoops(outputGene[m].connection, CGPNodeSet)) {
00158             for (int i = 0; i < branches.size(); i++) {
00159                 int cell1 = branches[i][branches[i].size() - 2];
00160                 int cell2 = branches[i][branches[i].size() - 1];
00161 
00162                 if (genes[cell1].connection1 == cell2) {
00163                     genes[cell1].connection1 = connectionDis(gen);
00164 
00165                     while (true) {
00166                         if (genes[cell1].connection1 < inputs)
00167                             break;
00168                         if ((genes[cell1].connection1 % columns) == (cell1 % columns))
00169                             genes[cell1].connection1 = connectionDis(gen);
00170                         else if (((genes[cell1].connection1 - inputs) % columns) > (((cell1 - inputs)
      % columns) + levelsBack))
00171                             genes[cell1].connection1 = connectionDis(gen);
00172                         else
00173                             break;
00174                     }
00175                 }
00176                 else if (genes[cell1].connection2 == cell2) {
00177                     genes[cell1].connection2 = connectionDis(gen);
00178 
00179                     while (true) {
00180                         if (genes[cell1].connection2 < inputs)
00181                             break;
00182                         if ((genes[cell1].connection2 % columns) == (cell1 % columns))
00183                             genes[cell1].connection2 = connectionDis(gen);
00184                         else if (((genes[cell1].connection2 - inputs) % columns) > (((cell1 - inputs)
      % columns) + levelsBack))
00185                             genes[cell1].connection2 = connectionDis(gen);
00186                         else
00187                             break;
00188                     }
00189                 }
00190             }
00191 
00192             CGPNodeSet.clear();
00193         }
00194     }
00195 }
```

6.8 CGPIndividual.h

```
00001 #ifndef CGPINDIVIDUAL_H
00002 #define CGPINDIVIDUAL_H
```

```
00003 #define TYPE double
00004
00005 #include <vector>
00006 #include <sstream>
00007 #include <functional>
00008 #include "CGPNode.h"
00009 #include "CGPOutput.h"
00010
00011 namespace parallel_cgp {
00015     class CGPIndividual {
00016     private:
00017         void isUsed(int nodeNum);
00018         bool loopFinder(int nodeNum, std::vector<int> nodeSet);
00019         TYPE evalNode(int nodeNum, std::function<TYPE(int, TYPE, TYPE)> computeNode);
00020         void clearInd();
00021     public:
00025         std::vector<CGPNode> genes;
00029         std::vector<CGPOutput> outputGene;
00034         std::vector<std::vector<int» branches;
00038         int rows;
00042         int columns;
00046         int levelsBack;
00050         int inputs;
00054         int outputs;
00058         int evalDone;
00059
00063         CGPIndividual();
00075         CGPIndividual(std::vector<CGPNode> genes, std::vector<CGPOutput> outputGene, int rows, int
      columns, int levelsBack, int inputs, int outputs);
00081         CGPIndividual(std::vector<CGPNode> genes, std::vector<CGPOutput> outputGene, int rows, int
      columns, int levelsBack, int inputs, int outputs, bool evalDone);
00082
00086         void printNodes();
00091         void evaluateValue(std::vector<TYPE> input, std::function<TYPE(int, TYPE, TYPE)> computeNode);
00095         void evaluateUsed();
00100         static CGPIndividual deserialize(std::istream& is);
00107         bool findLoops(int nodeNum, std::vector<int> nodeSet);
00111         void resolveLoops();
00112
00116         friend std::ostream& operator«(std::ostream& os, const CGPIndividual& ind) {
00117             os « ind.rows « " " « ind.columns « " " « ind.levelsBack « " "
00118                 « ind.inputs « " " « ind.outputs « " " « ind.evalDone « "\n";
00119
00120             os « ind.genes.size() « "\n";
00121             for (const auto& gene : ind.genes)
00122                 os « gene « "\n";
00123
00124             os « ind.outputGene.size() « "\n";
00125             for (const auto& output : ind.outputGene)
00126                 os « output « "\n";
00127
00128             return os;
00129         }
00133         friend std::istream& operator»(std::istream& is, CGPIndividual& ind) {
00134             is » ind.rows » ind.columns » ind.levelsBack
00135                 » ind.inputs » ind.outputs » ind.evalDone;
00136
00137             size_t genesSize, outputGeneSize;
00138             is » genesSize;
00139             ind.genes.resize(genesSize);
00140             for (auto& gene : ind.genes)
00141                 is » gene;
00142
00143             is » outputGeneSize;
00144             ind.outputGene.resize(outputGeneSize);
00145             for (auto& output : ind.outputGene)
00146                 is » output;
00147
00148             return is;
00149         }
00150     };
00151 }
00152
00153 #endif
```

6.9 CGPNode.h

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

```
00007
00008 namespace parallel_cgp {
00012     struct CGPNode {
00016         int operand;
00020         int connection1;
00024         int connection2;
00028         bool used;
00032         TYPE outValue;
00033
00037         friend std::ostream& operator«(std::ostream& os, const CGPNode& node) {
00038             os « node.operand « " " « node.connection1 « " " « node.connection2 « " " « node.used « "
      " « 0;
00039             return os;
00040         }
00044         friend std::istream& operator»(std::istream& is, CGPNode& node) {
00045             is » node.operand » node.connection1 » node.connection2 » node.used » node.outValue;
00046             return is;
00047         }
00048     };
00049 }
00050
00051 #endif
```

6.10 CGPOutput.h

```
00001 #ifndef CGPOUTPUT_H
00002 #define CGPOUTPUT_H
00003 #include <iostream>
00004 #include <fstream>
00005 #include <string>
00006 #define TYPE double
00007
00008 namespace parallel_cgp {
00012     struct CGPOutput {
00016         int connection;
00020         TYPE value;
00021
00025         friend std::ostream& operator«(std::ostream& os, const CGPOutput& output) {
00026             os « output.connection « " " « output.value;
00027             return os;
00028         }
00032         friend std::istream& operator»(std::istream& is, CGPOutput& output) {
00033             is » output.connection » output.value;
00034             return is;
00035         }
00036     };
00037 }
00038
00039 #endif
```

6.11 FuncProblem.cpp

```
00001 #include "FuncProblem.h"
00002
00003 using namespace std;
00004 using namespace parallel_cgp;
00005
00006 TYPE FuncProblem::computeNode(int operand, TYPE value1, TYPE value2) {
00007     switch (operand) {
00008     case 1:
00009         return value1 + value2;
00010     case 2:
00011         return value1 - value2;
00012     case 3:
00013         return value1 * value2;
00014     case 4:
00015         return (value2 == 0) ? 0 : value1 / value2;
00016     case 5:
00017         return sin(value1);
00018     case 6:
00019         return cos(value1);
00020     case 7:
00021         return value1 > 0 ? sqrt(value1) : value1;
00022     case 8:
00023         return pow(value1, 2);
00024     case 9:
00025         return pow(2, value1);
00026     default:
```

```
00027         return 0;
00028     }
00029 }
00030
00031 TYPE FuncProblem::fitness(TYPE x, TYPE y, TYPE res) {
00032     return func(x, y) - res;
00033 }
00034
00035 void FuncProblem::printFunction() {
00036     if (isSimulated)
00037         cout << "Funkcija: " << evalFunction(bestI.outputGene[0].connection) << endl;
00038     else
00039         cout << "Problem nije simuliran." << endl;
00040 }
00041
00042 string FuncProblem::evalFunction(int CGPNodeNum) {
00043     ostringstream oss;
00044
00045     if (CGPNodeNum < INPUTS) {
00046         switch (CGPNodeNum) {
00047         case 0:
00048             oss << "x";
00049             return oss.str();
00050         case 1:
00051             oss << "y";
00052             return oss.str();
00053         }
00054     }
00055
00056     switch (bestI.genes[CGPNodeNum].operand) {
00057     case 1:
00058         oss << "(" << evalFunction(bestI.genes[CGPNodeNum].connection1) << " + " <<
      evalFunction(bestI.genes[CGPNodeNum].connection2) << ")";
00059         return oss.str();
00060     case 2:
00061         oss << "(" << evalFunction(bestI.genes[CGPNodeNum].connection1) << " - " <<
      evalFunction(bestI.genes[CGPNodeNum].connection2) << ")";
00062         return oss.str();
00063     case 3:
00064         oss << "(" << evalFunction(bestI.genes[CGPNodeNum].connection1) << " * " <<
      evalFunction(bestI.genes[CGPNodeNum].connection2) << ")";
00065         return oss.str();
00066     case 4:
00067         oss << "(" << evalFunction(bestI.genes[CGPNodeNum].connection1) << " / " <<
      evalFunction(bestI.genes[CGPNodeNum].connection2) << ")";
00068         return oss.str();
00069     case 5:
00070         oss << "sin(" << evalFunction(bestI.genes[CGPNodeNum].connection1) << ")";
00071         return oss.str();
00072     case 6:
00073         oss << "cos(" << evalFunction(bestI.genes[CGPNodeNum].connection1) << ")";
00074         return oss.str();
00075     case 7:
00076         oss << "sqrt(" << evalFunction(bestI.genes[CGPNodeNum].connection1) << ")";
00077         return oss.str();
00078     case 8:
00079         oss << evalFunction(bestI.genes[CGPNodeNum].connection1) << "^2";
00080         return oss.str();
00081     case 9:
00082         oss << "2^" << evalFunction(bestI.genes[CGPNodeNum].connection1);
00083         return oss.str();
00084     }
00085
00086     return "";
00087 }
00088
00089 void FuncProblem::problemSimulator(CGPIndividual& individual, TYPE& fit) {
00090     function<TYPE(int op, TYPE v1, TYPE v2)> compNode =
00091         [&](int op, TYPE v1, TYPE v2) { return computeNode(op, v1, v2); };
00092
00093     TYPE N = 0;
00094
00095     for (TYPE x = -10; x < 10; x += 0.5) {
00096         for (TYPE y = -10; y < 10; y += 0.5) {
00097             vector<TYPE> input;
00098             input.push_back(x);
00099             input.push_back(y);
00100
00101             individual.evaluateValue(input, compNode);
00102             fit += pow(fitness(x, y, individual.outputGene[0].value), 2);
00103             N++;
00104         }
00105     }
00106
00107     fit /= N;
00108     fit = sqrt(fit);
00109 }
```

```
00110
00111 void FuncProblem::problemRunner() {
00112     CGP cgp(GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, INPUTS, OUTPUTS, MUTATIONS, NUM_OPERANDS,
      BI_OPERANDS, POPULATION_SIZE);
00113
00114     vector<CGPIndividual> population;
00115     int bestInd = 0, generacija = 0;
00116
00117     population = cgp.generatePopulation();
00118
00119     for (generacija = 0; generacija < GENERATIONS; generacija++) {
00120         TYPE bestFit = 0;
00121         bestInd = 0;
00122         vector<int> bestInds;
00123         random_device rd;
00124         mt19937 gen(rd());
00125
00126         for (int clan = 0; clan < POPULATION_SIZE; clan++) {
00127
00128             TYPE fit = 0;
00129             problemSimulator(population[clan], fit);
00130
00131             if (clan == 0)
00132                 bestFit = fit;
00133
00134             if (fit < bestFit) {
00135                 bestFit = fit;
00136                 bestInds.clear();
00137                 bestInds.push_back(clan);
00138             }
00139             else if (fit == bestFit)
00140                 bestInds.push_back(clan);
00141         }
00142
00143         if (bestInds.size() > 1)
00144             bestInds.erase(bestInds.begin());
00145
00146         uniform_int_distribution<> bestDis(0, static_cast<int>(bestInds.size()) - 1);
00147
00148         bestInd = bestInds[bestDis(gen)];
00149
00150         cout « "Gen: " « generacija « "; Fitness: " « bestFit « "; Indeks: " « bestInd « endl;
00151
00152         if (bestFit <= 5)
00153             break;
00154         if (generacija != GENERATIONS - 1)
00155             population = cgp.goldMutate(population[bestInd]);
00156     }
00157
00158     bestI = population[bestInd];
00159
00160     isSimulated = true;
00161
00162     printFunction();
00163 }
```

6.12 FuncProblem.h

```
00001 #ifndef FUNCPROBLEM_H
00002 #define FUNCPROBLEM_H
00003
00004 #include "../Problem.h"
00005 #include "../cgp/CGP.h"
00006
00007 #undef TYPE
00008 #define TYPE double
00009
00010 namespace parallel_cgp {
00014     class FuncProblem : public Problem {
00015     private:
00019         CGPIndividual bestI;
00023         const std::string bestFile = "func_best.txt";
00024
00030         const static int NUM_OPERANDS = 9;
00031         const static int BI_OPERANDS = 5;
00032         const static int INPUTS = 2;
00033         const static int OUTPUTS = 1;
00034
00039         int GENERATIONS = 5000;
00040         int ROWS = 8;
00041         int COLUMNS = 8;
00042         int LEVELS_BACK = 1;
00043         int MUTATIONS = 0;
```

```
00044         int POPULATION_SIZE = 20;
00045
00049         bool isSimulated = false;
00050
00054         const std::function<TYPE(TYPE x, TYPE y)> func =
00055             [](TYPE x, TYPE y) { return (pow(x, 2) + 2 * x * y + y); };
00056
00057         TYPE computeNode(int operand, TYPE value1, TYPE value2);
00058         TYPE fitness(TYPE x, TYPE y, TYPE res);
00059         void problemSimulator(parallel_cgp::CGPIndividual& individual, TYPE& fit) override;
00060         std::string evalFunction(int CGPNodeNum) override;
00061     public:
00065         FuncProblem() {};
00069         FuncProblem(int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int MUTATIONS, int
      POPULATION_SIZE)
00070             : GENERATIONS(GENERATIONS), ROWS(ROWS), COLUMNS(COLUMNS), LEVELS_BACK(LEVELS_BACK),
      MUTATIONS(MUTATIONS), POPULATION_SIZE(POPULATION_SIZE) {};
00071
00075         void problemRunner() override;
00079         void printFunction() override;
00080     };
00081 }
00082
00083 #endif
```

6.13 main.cpp

```
00001 #include <iostream>
00002 #include "Problem.h"
00003 #include "boolProblem/BoolProblem.h"
00004 #include "funcProblem/FuncProblem.h"
00005 #include "waitProblem/WaitProblem.h"
00006 #include "adProblem/ADProblem.h"
00007
00008 using namespace std;
00009 using namespace parallel_cgp;
00010
00011 int main() {
00012     int choice;
00013     cout « "Choose which problem to run" « endl « endl;
00014     cout « "1 - Boolean problem" « endl;
00015     cout « "2 - Parity problem" « endl;
00016     cout « "3 - Function problem" « endl;
00017     cout « "4 - Acey Deucey problem" « endl;
00018     cout « "5 - Wait problem" « endl;
00019     cout « endl « "Enter your choice: ";
00020     cin » choice;
00021
00022     Problem* problem = nullptr;
00023
00024     if (choice == 1)
00025         problem = new BoolProblem;
00026     else if (choice == 2)
00027         problem = new ParityProblem;
00028     else if (choice == 3)
00029         problem = new FuncProblem;
00030     else if (choice == 4)
00031         problem = new ADProblem;
00032     else if (choice == 5)
00033         problem = new WaitProblem;
00034     else
00035         cout « "Invalid option" « endl;
00036
00037     problem->problemRunner();
00038
00039     return 0;
00040 }
```

6.14 Problem.h

```
00001 #ifndef PROBLEM_H
00002 #define PROBLEM_H
00003 #define TYPE double
00004
00005 #include "cgp/CGPIndividual.h"
00006 #include <cmath>
00007 #include <random>
00008
```

```
00009 namespace parallel_cgp {
00010     class Problem {
00011     private:
00017         virtual void problemSimulator(parallel_cgp::CGPIndividual &individual, TYPE &fit) {}
00022         virtual std::string evalFunction(int CGPNodeNum) = 0;
00023     public:
00027         std::string bestFile = "problem_best.txt";
00028
00033         int NUM_OPERANDS = 9;
00034         int BI_OPERANDS = 5;
00035         int GENERATIONS = 5000;
00036         int ROWS = 20;
00037         int COLUMNS = 20;
00038         int LEVELS_BACK = 3;
00039         int INPUTS = 6;
00040         int OUTPUTS = 1;
00041         int MUTATIONS = 6;
00042         int POPULATION_SIZE = 20;
00043
00050         virtual TYPE computeNode(int operand, TYPE value1, TYPE value2) {
00051             switch (operand) {
00052             case 1:
00053                 return value1 + value2;
00054             case 2:
00055                 return value1 - value2;
00056             case 3:
00057                 return value1 * value2;
00058             case 4:
00059                 return (value2 == 0) ? 0 : value1 / value2;
00060             case 5:
00061                 return sin(value1);
00062             case 6:
00063                 return cos(value1);
00064             case 7:
00065                 return value1 > 0 ? sqrt(value1) : value1;
00066             case 8:
00067                 return pow(value1, 2);
00068             case 9:
00069                 return pow(2, value1);
00070             default:
00071                 return 0;
00072             }
00073         }
00077         virtual TYPE fitness(TYPE fit) { return fit; }
00078
00082         virtual void problemRunner() = 0;
00086         virtual void printFunction() = 0;
00087     };
00088 }
00089
00090 #endif
```

6.15 WaitProblem.cpp

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

```
00030     function<TYPE(int op, TYPE v1, TYPE v2)> compNode =
00031         [&](int op, TYPE v1, TYPE v2) { return computeNode(op, v1, v2); };
00032
00033     for (int iter = 0; iter < 10; iter++) {
00034         vector<TYPE> input;
00035         input.push_back(iter);
00036
00037         individual.evaluateValue(input, compNode);
00038     }
00039     fit = fitness(fit);
00040 }
00041
00042 void WaitProblem::problemRunner() {
00043     CGP cgp(GENERATIONS, ROWS, COLUMNS, LEVELS_BACK, INPUTS, OUTPUTS, MUTATIONS, NUM_OPERANDS,
      BI_OPERANDS, POPULATION_SIZE);
00044
00045     vector<CGPIndividual> population;
00046     int bestInd = 0, generacija = 0;
00047
00048     population = cgp.generatePopulation();
00049
00050     for (generacija = 0; generacija < GENERATIONS; generacija++) {
00051         TYPE bestFit = 0;
00052         bestInd = 0;
00053         vector<int> bestInds;
00054         random_device rd;
00055         mt19937 gen(rd());
00056
00057         for (int clan = 0; clan < POPULATION_SIZE; clan++) {
00058
00059             TYPE fit = generacija;
00060             problemSimulator(population[clan], fit);
00061
00062             if (fit > bestFit) {
00063                 bestFit = fit;
00064                 bestInds.clear();
00065                 bestInds.push_back(clan);
00066             }
00067             else if (fit == bestFit)
00068                 bestInds.push_back(clan);
00069         }
00070
00071         if (bestInds.size() > 1)
00072             bestInds.erase(bestInds.begin());
00073
00074         uniform_int_distribution<> bestDis(0, static_cast<int>(bestInds.size()) - 1);
00075
00076         bestInd = bestInds[bestDis(gen)];
00077
00078         cout « "Gen: " « generacija « "; Fitness: " « bestFit « "; Indeks: " « bestInd « endl;
00079
00080         if (bestFit == 100)
00081             break;
00082         if (generacija != GENERATIONS - 1)
00083             population = cgp.goldMutate(population[bestInd]);
00084     }
00085
00086     bestI = population[bestInd];
00087
00088     isSimulated = true;
00089
00090     printFunction();
00091 }
```

6.16 WaitProblem.h

```
00001 #ifndef WAITPROBLEM_H
00002 #define WAITPROBLEM_H
00003
00004 #include "../Problem.h"
00005 #include "../cgp/CGP.h"
00006 #include <chrono>
00007 #include <thread>
00008
00009 #undef TYPE
00010 #define TYPE double
00011
00012 namespace parallel_cgp {
00016     class WaitProblem : public Problem {
00017     private:
00021         CGPIndividual bestI;
00025         const std::string bestFile = "wait_best.txt";
00026
```

```
00031         int GENERATIONS = 5000;
00032         int ROWS = 8;
00033         int COLUMNS = 8;
00034         int LEVELS_BACK = 1;
00035         int INPUTS = 1;
00036         int OUTPUTS = 1;
00037         int MUTATIONS = 0;
00038         int POPULATION_SIZE = 20;
00039 
00043         int WAIT_TIME = 10;
00044 
00048         bool isSimulated = false;
00049 
00053         const std::function<void()> waitFunc =
00054             [&]() { std::this_thread::sleep_for(std::chrono::milliseconds(WAIT_TIME)); };
00055 
00056         TYPE fitness(TYPE prev) override;
00057         void problemSimulator(CGPIndividual& individual, TYPE& fit);
00058         std::string evalFunction(int CGPNodeNum) override;
00059     public:
00063         WaitProblem() {};
00067         WaitProblem(int GENERATIONS, int ROWS, int COLUMNS, int LEVELS_BACK, int OUTPUTS, int
      MUTATIONS, int POPULATION_SIZE, int WAIT_TIME)
00068             : GENERATIONS(GENERATIONS), ROWS(ROWS), COLUMNS(COLUMNS), LEVELS_BACK(LEVELS_BACK),
      OUTPUTS(OUTPUTS), MUTATIONS(MUTATIONS),
00069             POPULATION_SIZE(POPULATION_SIZE), WAIT_TIME(WAIT_TIME) {
00070         };
00071 
00075         void problemRunner() override;
00079         void printFunction() override;
00080     };
00081 }
00082 
00083 #endif
```

Index