VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Distribuovaný algoritmus diferenciální evoluce v platformě nezávislém jazyce

A Platform Independent Distributed DE Algorithm

2015 Bc. Petr Sklenička

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Petr Sklenička**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Distribuovaný algoritmus diferenciální evoluce v platformě nezávislém jazyce
A Platform Independent Distributed DE Algorithm

Zásady pro vypracování:

Cílem práce je vytvoření algoritmu DE v jazyce, jehož použití je nezávislé na používané výpočetní platformě v paralelním provedení. Jádro bude zajišťovat rozdělení úlohy a komunikaci mezi serverem a terminály provádějícími dílčí výpočty. V rámci laboratoře oboru navrhněte praktickou ukázku využívající produkt (např. optimalizovaný matematický model nebo naučenou neuronovou síť). Charakter práce: hardware, software, praktická realizace.

1. Seznámení se s problematikou evolučních algoritmů.
2. Vytvoření distribuovaného DE algoritmu.
3. Provedení testování na vybraných problémech.
4. Vzhodnocen9 v7sledk;.
5. Závěr.

Seznam doporučené odborné literatury:

[1] Koza J.R. 1998, Genetic Programming, MIT Press, ISBN 0-262-11189-6, 1998
[2] Koza J.R.,Bennet F.H., Andre D., Keane M. 1999, Genetic Programming III, Morgan Kaufnamm pub., ISBN 1-55860-543-6, 1999
[3] Lampinen Jouni, Zelinka, Ivan, New Ideas in Optimization & Mechanical Engineering Design Optimization by Differential Evolution. Volume 1. London: McGraw-Hill, 1999. 20 p. ISBN 007-709506-5
[4] Kvasnička V., Pospíchal J., Tiňo P., Evolučné algoritmy, STU Bratislava, ISBN 85-246-2000, 2000
[5] Zelinka I.: Analytic Programming by Means of Soma Algorithm. ICICIS'02, First International Conference on Intelligent Computing and Information Systems, Egypt, Cairo, 2002
[6] Zelinka Ivan, Evoluční výpočetní techniky - principy a alikace, BEN, Praha, 2008

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **prof. Ing. Ivan Zelinka, Ph.D.**

Datum zadání: 01.09.2014

Datum odevzdání: 07.05.2015

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 4. května 2015 .....

Jsem rád, že na tomto místě mohu poděkovat vedoucímu této práce, prof. Ing. Ivanu Zelinkovi, Ph.D. Děkuji mu za jeho odborný a zároveň lidský přístup a dále za jeho rady a doporučení, které mi dal během tvorby této práce. V neposlední řadě bych mu také velmi rád poděkoval za jeho výborné přednášky z předmětu Biologicky inspirované výpočty, během kterých jsem získal mnoho informací a zároveň jsem objevil zajímavé odvětví v oblasti informačních technologií.
Další osobou, které bych rád vyjádřil poděkování, je má přítelkyně. Děkuji ji za její podporu a především za pochopení, které měla, když jsem vytvářel tuto práci.

Abstrakt

Tato práce pojednává o evolučním algoritmu diferenciální evoluce. Je zde vysvětlen základní koncept evolučních algoritmů, po němž následuje podrobný popis a vysvětlení principu činnosti diferenciální evoluce. Dále je v této práci uvedeno několik modifikací tohoto algoritmu, přičemž důraz je kladen na popsání způsobů, jakými lze vytvořit distribuovaný algoritmus diferenciální evoluce. Poté následuje stručné představení neuronových sítí, které byly použity během provádění experimentů. Součástí práce je také popis implementace, popis provedených experimentů a dále prezentace dosažených výsledků.

Klíčová slova: diferenciální evoluce, distribuovaná diferenciální evoluce, paralelní diferenciální evoluce, evoluční algoritmy, neuronové sítě

Abstract

Thesis deals with evolutionary algorithm called differential evolution. A basic concept of evolutionary algorithms is explained here, after that there is followed detailed description and explanation of the differential evolution principles.Then there are mentioned several modifications of the algorithm, with the stress in description of the way how to create differential evolution distributed algorithm. Afterwards it is introduced the neural networks, that were used during performing experiments. Thesis also contains description of implementation, performed experiments and demonstration of achieved results.

Keywords: differential evolution, distributed differential evolution, parallel differential evolution, evolutionary algorithms, neural networks

Seznam použitých zkratek a symbolů

ACO	–	Ant Colony Optimization
DE	–	Differential Evolution
DDE	–	Distributed Differential Evolution
GUI	–	Graphic User Interface
IBDDE	–	Island Based Distributed Differential Evolution
ICEO	–	International Contest on Evolutionary Optimization
JRE	–	Java Runtime Environment
JVM	–	Java Virtual Machine
LS	–	Local Search
LSDE	–	Local Search Differential Evolution
OBDE	–	Opposition-Based Differential Evolution
PC	–	Personal Computer
PDE	–	Parallel Differential Evolution
PVM	–	Parallel Virtual Machine
SOMA	–	Self Organizing Migrating Algorithm
TCP	–	Transmission Control Protocol
TSP	–	Travelling Salesman Problem
UDP	–	User Datagram Protocol
XML	–	Extensible Markup Language

Obsah

Seznam tabulek

Seznam obrázků

Seznam pseudokódů

1 Úvod

Pokud bychom dostali za úkol najít odpovědi na otázky týkající se vzniku života na Zemi, zjistili bychom, že je to úkol více než obtížný, ne-li nadlidský. Celá řada vědců, filosofů ale i obyčejných laiků pátrala po staletí po původu života a přinesla spoustu teorií, ať už těch méně pravděpodobných (průnik života na Zemi z vesmíru; božský zásah; neměnný stav organismů od doby jejich stvoření) či více pravděpodobnějších, jako například vývoj života přímo na Zemi z neživých organismů. Stále jde ale jen o hypotézy a domněnky. [5] [7]

Daleko snadnější by pro nás bylo najít odpovědi na otázky týkající se vývoje lidského rodu. Kdo je člověk a od jakého momentu rozlišujeme lidský druh od ostatních živých organismů? Pomineme-li opět nepravděpodobné teorie založené na náboženských představách, existuje již velká řada vědecky podložených faktů, díky kterým můžeme s určitou mírou „přesnosti" určit moment, kdy se po Zemi pohyboval první živočišný druh rodu Homo – *Homo habilis* (člověk zručný). Stejně tak můžeme s určitostí tvrdit, že od doby, kdy se po planetě Zemi pohyboval člověk zručný, uplynulo již více než milion let, po který se druh druh Homo postupně vyvíjel až do současné podoby *Homo sapiens sapiens* (člověk moudrý). [14] Tento dlouhý proces vývoje lze označit jedním slovem – evoluce.

Pojem evoluce určitě není neznámý a není třeba ho detailněji vysvětlovat. Stejně tak není třeba zdůrazňovat důležitost průkopníků evoluční teorie, mezi které bezpochyby patří Charles Darwin, britský přírodovědec, který je považován za zakladatele teorie evoluce. Základním stavebním kamenem jeho teorie je předávání rodičovského genomu novým potomkům s následným uvolněním prostoru pro tyto nové potomky, čímž vznikne nová generace, která je vyspělejší než ta předchozí. Tuto velmi jednoduchou teorii lze uplatnit i v jiné oblasti, což bude vysvětleno dále v této kapitole.

Žijeme ve 21. století, jsme obklopeni velmi moderními technologiemi a doba, kdy jsme používali primitivní nástroje a žili v jeskyni, je dávnou historií. Za celou dobu naší existence jsme udělali nesmírně velké pokroky v různých oblastech - jmenujme například medicínu, astronomii, biologii, matematiku, chemii či fyziku. Jsme rovněž schopni řešit celou řadu různých problémů, od jednoduchých až po velmi složité. Lze tedy jistě tvrdit, že hranice našich možností se velmi rozšířily. Navzdory tomu však stále existuje velká řada témat a objektů k prozkoumání a neustále se potýkáme s nepřeberným množstvím problémů, které nejsme schopni řešit ani s pomocí nejrychlejších počítačů. Dnešní počítače sice pracují velmi rychle, ne však nekonečně rychle. Stále tak před námi stojí úlohy, jež nedokážeme řešit žádným deterministickým algoritmem, který by byl uspokojivě rychlý.[1]

Jako příklad takové úlohy uveďme problém obchodního cestujícího, rovněž známý pod názvem TSP (Travelling Salesman Problem). Zadání této úlohy je velmi prosté. Existuje n měst, mezi nimiž vedou cesty o známých délkách a úkolem obchodního cestujícího je najít nejkratší možnou cestu, při níž navštíví všechna města.

[1] Pod termínem deterministický algoritmus, který vrací výsledek v uspokojivém čase, chápejme algoritmus s polynomiální časovou složitostí. Problémy, které umíme takovým algoritmem řešit, patří do skupiny zvané PTIME.

Zcela nepochybně existuje deterministický algoritmus řešící tento problém. Stačí vzít v potaz všechny možné existující cesty, zjistit, jak jsou dlouhé a poté vybrat tu nejlepší, resp. nejkratší. Budeme-li předpokládat, že mezi všemi městy existuje přímá cesta, pak v případě pěti měst existuje celkem 5! možností, v jakém pořadí je lze navštívit. Stačí tedy ze 120 kombinací vybrat tu nejlepší. Co se ale stane v případě, že měst bude například 50? Celkový počet možností je pak 50!, což je přibližně $3,0414 \cdot 10^{64}$ možných řešení. Projít postupně všechna tato řešení a vybrat z nich to nejlepší je již úkol, který by i na velmi rychlém počítači trval velmi dlouhou dobu. Není těžké si nyní uvědomit, že časová složitost námi navrženého algoritmu je $n!$, kde n je počet měst, která musí obchodní cestující navštívit. Zatím se nikomu nepodařilo najít deterministický algoritmus s polynomiální časovou složitostí, který by úlohu TSP dokázal řešit. Jinými slovy to znamená, že TSP nepatří do třídy problémů zvané PTIME.

Fakt, že TSP není ve třídě PTIME, ještě ale neznamená, že nejsme schopni takovou úlohu vyřešit. TSP je optimalizační problém a pro takové problémy existuje mnoho kategorií a typů algoritmů, jež můžeme použít. Jedna z těchto kategorií obsahuje algoritmy, jejichž základní princip se opírá o již zmíněnou Darwinovu evoluční teorii. Jedná se o kategorii evolučních algoritmů.

Hlavní myšlenka evolučních algoritmů spočívá v napodobení biologického evolučního procesu. Tyto algoritmy pracují s populací, která je tvořena množinou jedinců, jejichž kombinací, resp. jejich křížením a mutací, poté vznikají noví jedinci, kteří vytvářejí novou, vyspělejší populaci. Celý tento proces se periodicky opakuje, stejně jako je tomu u evolučního biologického procesu. Z tohoto velmi stručného nastínění základního principu činnosti evolučních algoritmů je snad již dostatečně vidět silná podobnost s Darwinovou evoluční teorií. Jedním z algoritmů, který patří do této kategorie, je diferenciální evoluce, která je zároveň náplní této práce.

Následující text je možno pomyslně rozdělit na dvě části, a to na část teoretickou a praktickou. V teoretické části je nejprve obecně nastíněn koncept evolučních algoritmů společně s uvedením kategorie problémů, pro něž jsou tyto algoritmy vhodné. Dále je teoretická část tvořena podrobným popisem algoritmu diferenciální evoluce včetně stručné historie, rovněž je popsáno přesné fungování algoritmu, za kterým následuje popis jeho možných vylepšení, přičemž důraz je kladen na způsoby, jakými lze vytvořit distribuovaný algoritmus diferenciální evoluce. Poslední kapitolu teoretické části tvoří zevrubný popis neuronových sítí. Poté již následuje praktická část, v níž je popsán způsob implementace vytvořených projektů. Dále je část tvořena kapitolou s přehledem použitých testovacích funkcí, za kterou následuje kapitola, v níž jsou popsány provedené experimenty společně s prezentací dosažených výsledků.

2 Evoluční algoritmy

Předtím, než bude čtenář seznámen s algoritmem diferenciální evoluce, je potřeba nastínit oblast použití evolučních algoritmů, základní koncept jejich činnosti, a dále je nutné uvést a vysvětlit základní pojmy, které se v rámci evolučních algoritmů používají.

2.1 Oblast použití

Evoluční algoritmy jsou obecně vhodné k řešení optimalizačních problémů, které mají svůj původ v dávné historii a kterých v současné době existuje celé spektrum pokrývající různé oblasti lidské činnosti. Jmenujme například úlohy z ekonomie, kam se obecně řadí problémy optimalizace výrobních programů, přičemž cílem může být například maximalizovat zisk během výroby, nebo naopak minimalizovat čas potřebný k výrobě požadovaného množství produktů. Dále jmenujme problémy z oblasti dopravy. Zde si představme situaci, kdy se hledá plán rozvozu zboží tak, aby byly minimalizovány náklady nebo aby byl minimalizován čas, který bude potřeba pro splnění plánu rozvozu. Obecně lze tvrdit, že optimalizačních problémů stále přibývá a jsou stále obtížnější. Jejich společným prvkem je fakt, že řešení pomocí deterministických algoritmů by ve většině případů spotřebovalo extrémně velké množství času. Proto se k jejich řešení nabízí použít evoluční algoritmy, které dokáží v poměrně krátkém čase najít optimální řešení. Optimálním řešením se rozumí řešení takové, které nejvíce vyhovuje stanoveným požadavkům (maximální výrobní zisk, minimální ztráty, apod.). Pokud bychom na tyto problémy nahlíželi jako na geometrické problémy, lze optimálním řešením rozumět globální minimum nebo maximum na N rozměrné ploše.

V současné době je známo relativně velké množství evolučních algoritmů, přičemž platí, že žádný z nich není vhodný pro řešení všech druhů problémů, tedy každý evoluční algoritmus vykazuje dobré výsledky pouze na omezené množině problémů. Jako příklad uveďme algoritmus ACO (*Ant Colony Optimization*) [2], který podává velmi dobré výsledky při použití na kombinatorických optimalizačních problémech, jakým je například již zmíněný problém obchodního cestujícího. Oproti tomu existuje celá řada problémů, u kterých by použití ACO přineslo horší výsledky, a proto je vhodnější aplikovat algoritmus jiný. Při použití evolučních algoritmů je tak tedy skutečně důležité vědět, jaký typ algoritmu na řešení daného problému aplikovat. [19]

2.2 Základní princip činnosti

I navzdory faktu, že evolučních algoritmů existuje celá řada a způsoby jejich činnosti nejsou stejné, existuje společná myšlenka, z níž tyto algoritmy vychází. Tato myšlenka již byla velmi stručně nastíněna v úvodu, nicméně pro úplnost ji zopakujeme a rozvedeme do větších detailů.

Typicky jsou evoluční algoritmy založeny na práci s populací, která se při běhu algoritmu vyvíjí. Populace se skládá z množiny jedinců, přičemž každý jedinec reprezentuje

[2] ACO není typickým zástupcem evolučních algoritmů, nicméně patří k evolučním výpočetním technikám. [19]

jedno řešení daného problému. Jedinec se skládá z parametrů a dále z tzv. vhodnosti, též označované jako fitness hodnota. Parametry jedince jsou v podstatě argumenty tzv. účelové[3] (nebo také optimalizační) funkce, což je funkce, jejíž hodnota představuje pro zvolené argumenty onu zmíněnou fitness hodnotu.

Pro lepší vysvětlení opět uvedeme příklad problému obchodního cestujícího. Počet parametrů jedince bude roven počtu měst a argumenty účelové funkce budou v tomto případě indexy měst, které reprezentují pořadí, v jakém má cestující města navštívit. Účelová funkce je pak funkce, která pro zadané pořadí měst vydá na svém výstupu celkovou délku trasy. Délka trasy představuje fitness hodnotu jedince a cílem je tedy najít kombinaci argumentů takovou, aby délka trasy byla co nejmenší.

Jak jsme již zmínili, populace se při běhu algoritmu vyvíjí. To znamená, že z populace jsou postupně odstraňování nevyhovující jedinci, kteří jsou nahrazování novými a lepšími. To se děje tak, že se z populace vybírají jedinci, kteří představují rodiče, a jejichž křížením vznikají potomci, kteří jsou následně zmutováni (obdoba biologické mutace genů). Tím vznikají noví jedinci. Do nové populace se pak dostanou pouze nejlepší jedinci, nevyhovující jsou z populace vyloučeni (umírají). To, jak přesně probíhá výběr rodičů, křížení a mutace, je již odvislé od konkrétního typu evolučního algoritmu. Celý proces vytváření nových jedinců se periodicky opakuje do doby, než je splněno ukončovací kritérium. Různým ukončovacím kriteriím se budeme věnovat v kapitole 3.3. Pro úplnost ještě odkazujeme čtenáře na obrázek 1, který obecně zachycuje běh evolučního algoritmu. [19]

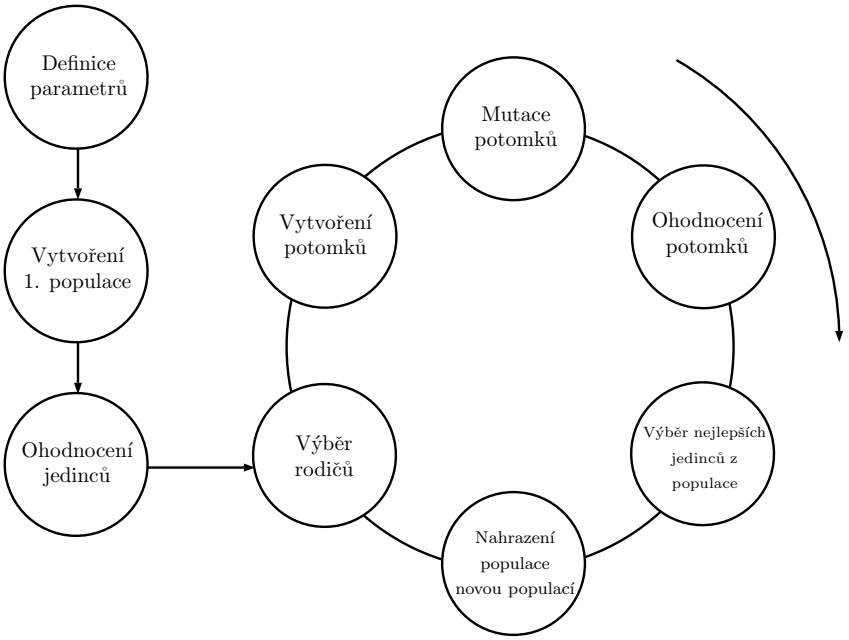
2.3 Vzorový jedinec

Ve většině případů je u evolučních algoritmů prvotní populace tvořena náhodně, tedy s použitím generátoru pseudonáhodných čísel. Obvykle je potřeba hledat řešení pouze na nějaké omezené N rozměrné ploše, což jinými slovy znamená, že parametry jedinců musí být generovány v předem zadaném intervalu. Tím se dostáváme k pojmu vzorový jedinec, který se mnohdy v literatuře označuje též jako specimen.

Specimen definuje, v jakém rozsahu se mají nacházet jednotlivé parametry jedinců. Obecně platí, že parametry jedince mohou být trojího typu:

1. reálné - hodnota parametru je reálné číslo
2. celočíselné - hodnota parametru je celé číslo
3. diskrétní - hodnota parametru je vybrána z předem určené množiny hodnot

[3] V angličtině se můžeme setkat s názvy *objective function*, *cost function*, *fitness function* nebo také *error function*. [12]



Obrázek 1: Obecný průběh evolučního algoritmu [19]

Obrázek neobsahuje ukončení algoritmu po splnění ukončovacího kritéria a výběr nejlepšího jedince z poslední populace.

Můžeme tedy klidně pracovat s populací jedinců, jejichž parametry jsou různé. Některý z parametrů jedince tak může být například celé číslo ze zadaného intervalu, jiný parametr může mít hodnotu vybranou z nějaké předem definované množiny - například $\{true, false, 20, 30, Monday\}$. Abychom tedy mohli jednoznačně určit, z jakých parametrů budou jedinci složeni, stačí nadefinovat vzorového jedince. Jeho obecný zápis vypadá takto:

$$Specimen = \{\{Real, \{Lo, Hi\}\}, \{Integer, \{Lo, Hi\}\}, ..., \{Real\{Lo, Hi\}\}\}$$

Konstanty Lo a Hi určují dolní, resp. horní hranici daného parametru. Jakým způsobem se pracuje s celočíselnými a diskrétními parametry bude popsáno v jedné z částí následující kapitoly.

3 Diferenciální evoluce

V předchozí kapitole jsme obecně nastínili základní myšlenku evolučních algoritmů. Dále jsme zmínili, že jednotlivé operace v rámci evolučního cyklu, jakými jsou křížení, mutace a způsob výběrů rodičů, jsou již závislé na konkrétním typu algoritmu. V kapitole 3 bude vysvětleno, jak jsou tyto operace realizovány u algoritmu diferenciální evoluce. Nejprve ale bude představena stručná historie algoritmu. V závěru kapitoly se pak čtenář seznámí s různými variantami diferenciální evoluce a s dalšími možnými modifikacemi, které lze provádět za účelem zlepšení výkonu algoritmu.

3.1 Historie

Diferenciální evoluce (DE) je algoritmus, jehož základní myšlenka je podobná genetickým algoritmům. Autory DE jsou Kenneth Price a Rainer Storn. Vzniku DE předcházelo vytvoření algoritmu zvaného genetické žíhání, který roku 1994 publikoval Kenneth Price v magazínu *Dr. Dobb's Journal*, což je populární magazín pro programátory. Po uveřejnění genetického žíhání se Rainer Storm rozhodl spojit s Pricem za účelem použití genetického žíhání na složitější problém, v angličtině označovaný jako *Chebyshev polynomial fitting problem*. Ukázalo se, že je velmi obtížné pro tento problém určit řídící parametry algoritmu, a proto Price začal algoritmus genetického žíhání modifikovat, což vedlo k vytvoření operátoru diferenciální mutace, na kterém je DE založena. Na základě další vzájemné spolupráce obou výše zmíněných autorů byla v roce 1995 poprvé představena a použita diferenciální evoluce. Tento úspěch vedl oba autory k použití diferenciální evoluce na první ICEO (International Contest on Evolutionary Optimization) konferenci, která se konala v květnu roku 1996 v japonském městě Nagoya. Ukázalo se, že DE dobře fungovala na testovacích funkcích, nicméně na řešení širokého spektra optimalizačních problémů byla nedostačující. Tento neúspěch vedl k vytvoření další verze diferenciální evoluce a rovněž k napsání článku *Differential Evolution - A Simple Evolution Strategy for Fast Optimization*, který byl publikován v květnu roku 1997 v již zmíněném magazínu *Dr. Dobb's Journal*. Tento článek byl odborníky poměrně dobře přijat a diferenciální evoluce tak byla poprvé představena velkému mezinárodnímu publiku. [12] [19]

V prosinci 1997 byl v magazínu *The Journal of Global Optimization* publikován další článek, po jehož přečtení si mnoho dalších odborníků a výzkumníků v této oblasti začalo uvědomovat potenciál diferenciální evoluce. Článek poskytl rozsáhlé důkazy o robustnosti a výkonnosti algoritmu na široké škále testovacích funkcí.

Téhož roku se v Indianopolis konala i druhá ICEO konference. Z představených optimalizačních technik se diferenciální evoluce ukázala jako nejlepší. Na této konferenci se Kenneth Price setkal s Dr. Davidem Cornem, který jej přizval k napsání části publikace *New Ideas in Optimization*. Spolu s ním byli osloveni i další odborníci - jmenovat můžeme například prof. Jouniho Lampinena nebo prof. Ivana Zelinku. [12] [19]

V dnešní době je již diferenciální evoluce dlouhodobě považována za velmi efektivní algoritmus, který lze úspěšně použít na širokou škálu optimalizačních problémů. [4]

3.2 Popis činnosti diferenciální evoluce

3.2.1 Řídící parametry

Hlavním faktorem, který výrazně ovlivňuje běh a výkonnost diferenciální evoluce, jsou tzv. řídící parametry. Tyto parametry typicky nastavuje uživatel před spuštěním algoritmu. Je nutné podotknout, že jejich špatným nastavením může dojít k tomu, že optimální řešení nebude nikdy nalezeno, nebo jeho nalezení zabere zbytečně velké množství času. Doporučeným hodnotám parametrů se budeme v této kapitole také věnovat, nejprve ale uvedeme, o jaké parametry se jedná a jaký je jejich význam.

Pokud v souvislosti s diferenciální evolucí mluvíme o řídících parametrech, jedná se především o parametry CR, F, dále velikost populace (v literatuře se objevuje pod zkratkou NP) a počet parametrů jedince. Samozřejmě je nutné definovat i ukončovací kritérium, při jehož splnění se algoritmus zastaví a vrátí nejlepší nalezený výsledek. Velmi často používaným ukončovacím kritériem je předem stanovený počet evolučních cyklů (generací). V rámci této práce budeme stanovený počet generací označovat symbolem G.

Přehled parametrů, společně s jejich významem, je uveden v tabulce 1.

Parametr	**Význam**
CR	Práh křížení
F	Mutační konstanta
NP	Velikost populace
D	Počet parametrů jedince (dimenze problému)
G	Počet generačních cyklů

Tabulka 1: Parametry diferenciální evoluce

Již samotný význam parametrů CR a F napovídá, že práh křížení se využívá v průběhu procesu křížení a mutační konstanta je využita během mutace. Oba tyto parametry mají svůj povolený rozsah hodnot. V případě CR jde o interval $\langle 0, 1 \rangle$, zatímco parametr F může nabývat hodnot z intervalu $\langle 0, 2 \rangle$. Jak přesně se tyto parametry používají je vysvětleno dále v této kapitole.

[4] Pro úplnost ještě uvádíme, že tím rozhodně nechceme tvrdit, že je diferenciální evoluce nejlepší algoritmus, který spolehlivě řeší všechny problémy. Důvodem je především fakt, že takový algoritmus neexistuje a také to, že existují i jiné velmi výkonné algoritmy.

3.2.2 Vytvoření počáteční populace

Po nadefinování řídících parametrů evoluce lze přistoupit k vytvoření první populace, aby bylo možno začít provádět evoluční cykly. Počáteční populace je tvořena náhodně. Všem jedincům z populace se tedy přiřadí náhodné hodnoty jejich parametrů. Matematicky můžeme vytvoření populace zachytit vztahem uvedeným níže.

$$P^{(0)} = x_{i,j}^{(0)} = r_{i,j}(x_j^{(U)} - x_j^{(L)}) + x_j^{(L)}$$

Hodnota i zde představuje index jedince v populaci, j označuje pořadí parametru a r je náhodné číslo v rozsahu [0, 1]. Hodnoty $x_j^{(U)}$ a $x_j^{(L)}$ představují horní, resp. dolní hranici hodnoty daného parametru.

3.2.3 Mutace

Poznámka 3.1 V průběhu mutace se v diferenciální evoluci využívají základní vektorové operace. Pokud si u libovolného jedince odmyslíme jeho fitness hodnotu, zůstanou nám jen jeho parametry, které ve své podstatě tvoří vektor. Proto tedy dále v textu, nebude-li řečeno jinak, budeme pod pojmem vektor rozumět jednoho jedince.

Ve své základní verzi vytváří diferenciální evoluce nové potomky celkem ze čtyř rodičů. Nejprve je nutno vytvořit váhový diferenční vektor, který vznikne jako rozdíl dvou náhodných vektorů z populace vynásobený mutační konstantou F. Tento vektor se poté přičte ke třetímu náhodnému vektoru, čímž vznikne tzv. šumový vektor (v angličtině označován jako *noisy vector*). Proces vytvoření šumového vektoru nazýváme mutace. Pro formálnější vyjádření opět použijme matematický vztah:

$$v_{i,G+1} = x_{r_1,G} + F \cdot (x_{r_2,G} - x_{r_3,G})$$

Symboly r_1, r_2, r_3 představují náhodné indexy, pro které platí $r_1, r_2, r_3 \in \{1, 2, ..., NP\} \wedge r_1 \neq r_2 \neq r_3 \neq i$.

Pro úplnost je potřeba dodat, že vlivem mutace může vzniknout jedinec s parametry, které se nemusí nacházet v předem povoleném rozsahu hodnot. U problémů, které nemají kladena omezení na argumenty účelové funkce, se může jednat o vhodnou vlastnost. Je tak totiž možné najít optimální řešení i za hranicí prostoru, do kterého byla náhodně vygenerována prvotní populace. [10]

U problémů, kde je potřeba prohledávat pouze omezený prostor, je však nutné zajistit, aby všechny parametry jedinců ležely v námi definovaném intervalu. K zajištění splnění této podmínky existuje několik možností.

Patrně nejtriviálnějším řešením je zastavení jedince na hranici prohledávaného prostoru. V takovém případě však velmi lehce může dojít k tomu, že se jedinci budou postupně shlukovat na hranicích, čímž se bude snižovat diverzibilita populace, což může v konečném důsledku vést k faktu, že nalezení globálního extrému bude trvat velmi dlouho, nebo v horším případě nebude nalezen vůbec. [19]

Dalším řešením u parametrů, které překročí povolený rozsah hodnot, je opakování procesu mutace tak dlouho, jak bude potřeba. Tedy do doby, než bude vytvořen parametr ležící v povoleném intervalu. [10]

Poslední možností, kterou uvedeme, je prosté nahrazení nevyhovujících parametrů náhodnými hodnotami uvnitř povolené hranice, což lze opět zachytit pomocí matematického vztahu, v němž hodnota i představuje index jedince v populaci a j index parametru daného jedince.

$$x_{i,j}^{(G+1)} = \begin{cases} r_{i,j}(x_j^{(U)} - x_j^{(L)}) + x_j^{(L)} \text{ pokud } x_{i,j}^{(G+1)} < x_j^{(L)} \vee x_{i,j}^{(G+1)} > x_j^{(U)} \\ \mathrm{x}_{i,j}^{(G+1)} \text{ v opačném případě} \end{cases}$$

Při použití této techniky nedochází ke snižování diverzibility populace, ale naopak k jejímu zvyšování. Z geometrického hlediska dojde v případě náhodného nahrazení parametru k přesunutí jedince na novou pozici, která nevznikla křížením jedinců. [19]. Tato technika byla použita i v implementační části této práce.

3.2.4 Křížení

Po mutaci následuje v diferenciální evoluci proces křížení, ke kterému jsou zapotřebí dva vektory. Jedná se o čtvrtého, dosud nepoužitého rodiče (aktivní jedinec) a šumový vektor vzniklý při mutaci. Výsledkem křížení je zkušební vektor (*trial vector*), který poté soutěží o postup do nové populace. [19] Faktorem, který rozhoduje o postupu do dalšího evolučního cyklu, je již zmíněná fitness hodnota.

Proces vytvoření zkušebního vektoru je závislý na typu použitého křížení. V klasické variantě DE (DE/rand/1/bin) se využívá binomické křížení, existuje však i křížení zvané exponenciální. V obou případech je vždy zkušební vektor vytvořen kombinací parametrů z šumového a rodičovského vektoru. Jak přesně probíhají oba zmíněné procesy křížení zachycují pseudokódy 1 a 2, které jsou převzaty z článku [18]. Význam symbolů x, y a z je následující:

- x - aktivní vektor (čtvrtý rodič)
- y - šumový vektor
- z - zkušební vektor

3.2.5 Evoluční cykly

Nyní, když bylo popsáno, jakým způsobem probíhá proces mutace a křížení, je již snadné popsat celý průběh jednoho evolučního cyklu, který se opakuje tak dlouho, dokud není splněna podmínka pro ukončení algoritmu.

Evoluční cyklus tedy probíhá tak, že postupně procházíme celou populaci a k aktuálnímu jedinci vždy náhodně vybereme další tři, čímž získáme čtyři rodiče pro nového potomka (jedince). Poté proběhne proces mutace a křížení, čímž vznikne zkušební jedinec,

Pseudokód 1 Binomické křížení

```
1: crossoverBin (x,y)
2: k ← irand({1, ..., n})
3: for j = 1, n do
4:    if rand(0,1) < CR or j = k then
5:       z^j ← y^j
6:    else
7:       z^j ← x^j
8:    end if
9: end for
10: return z
```

Pseudokód 2 Exponenciální křížení

```
1: crossoverExp (x,y)
2: z ← x; k ← irand({1, ..., n}); j ← k; L ← 0
3: repeat
4:    z^j ← y^j; j ← ⟨j + 1⟩_n; L ← L + 1
5: until rand(0, 1) > CR or L = n
6: return z
```

který na základě své vhodnosti postoupí či nepostoupí do nové populace. Po dokončení iterace je vytvořena nová populace, která nahradí populaci starou. Tím je dokončen jeden evoluční cyklus a celý tento jednoduchý proces se může začít znovu opakovat. V momentě, kdy dojde ke splnění ukončovací podmínky, je z aktuální populace vybrán nejlepší jedinec, který je poté výstupem algoritmu.

Výše popsané kroky jsou znázorněny v pseudokódu 3, který byl opět převzat z článku [18].

3.3 Ukončovací kritéria

Jak již bylo v předchozích částech uvedeno, ukončovací kritérium určuje, kdy se má algoritmus DE zastavit a vybrat z aktuální populace nejlepšího jedince, který reprezentuje nejlepší řešení, jež algoritmus dokázal najít. Do této chvíle jsme zmínili pouze jednu možnou podmínku ukončení, a to předem stanovený počet generačních cyklů.

Nastavením přesného počtu generačních cyklů můžeme značně ovlivnit celkovou dobu běhu algoritmu. Je pochopitelné, že se zvyšující se hodnotou evolučních cyklů se zvyšuje i čas potřebný pro provedení algoritmu. V případě, že nastavíme hodnotu G příliš velkou, může dojít k situaci, že nejlepší jedinec bude nalezen mnohem dříve, než proběhne G evolučních cyklů, tudíž se zbytečně prodlouží doba běhu algoritmu, která samotný výkon algoritmu již nezlepší. V opačném případě, nastavíme-li G příliš malé, nemusí být nejlepší řešení vůbec nalezeno.

Mírnou modifikací výše uvedené podmínky pro ukončení dostaneme další možnost, kdy zastavit běh algoritmu. V průběhu evolučních cyklů si stačí vždy uložit nejlepšího

Pseudokód 3 Průběh diferenciální evoluce

```
1: Population initialization X(0) ← {x_1(0),...,x_m(0)}
2: g ← 0
3: Compute {f(x_1(g)),...,f(x_m(g))}
4: while the stopping condition is false do
5:    for i = 1, m do
6:       y_i ← generateMutant(X(g)) {Vytvoření šumového vektoru}
7:       z_i ← crossover(x_i(g), y_i) {Vytvoření zkušebního vektoru}
8:       if f(z_i) < f(x_i(g)) then
9:          x_i(g + 1) ← z_i
10:      else
11:         x_i(g + 1) ← x_i(g)
12:      end if
13:   end for
14:   g ← g + 1
15:   Compute {f(x_1(g)),...,f(x_m(g))}
16: end while
```

jedince v dané populaci a algoritmus ukončit ve chvíli, kdy se fitness hodnota nejlepšího jedince nezmění během stanoveného počtu evolučních cyklů (Δg_{max}). Je ovšem důležité dbát na to, aby tato hodnota nebyla příliš nízká, jelikož delší periody bez zlepšení nejlepšího jedince jsou u DE mnohem častější než u ostatních evolučních algoritmů. [12]

Níže jsou uvedeny další možnosti ukončovacích kritérií. Je nutné zmínit, že uvedený výčet zdaleka není kompletní, neboť stále lze vymýšlet nové podmínky.

3.3.1 Limitovaný čas doby běhu

V některých případech je pro optimalizaci dostupné pouze omezené množství času. V takovýchto případech musí být algoritmus ukončen po uplynutí zadaného času, a to bez ohledu na stav populace nebo počet provedených evolučních cyklů. [12]

3.3.2 Dosažení hledané hodnoty

Ve většině optimalizačních úloh z reálného světa neznáme předem hledanou minimální či maximální hodnotu účelové funkce. V případě, že je ale hodnota hledaného extrému předem známa (např. u testovacích funkcí), můžeme algoritmus ukončit ve chvíli, kdy je nalezena hodnota extrému(nebo hodnota velmi blízká). [12] V případě použití tohoto ukončovacího kritéria je ale nutné si uvědomit, že například špatným nastavením řídících parametrů může dojít k tomu, že algoritmus hledaný extrém nikdy nenajde. K tomu může dojít například tak, že populace uvázne v lokálním extrému. Vinou této skutečnosti by tak došlo k tomu, že by se algoritmus nikdy nezastavil a evoluční cykly by probíhaly nekonečně dlouho. Abychom takové situaci předešli, můžeme použití tohoto ukončo-

vacího kritéria zkombinovat s jiným kritériem, například s předem stanoveným časem maximální doby běhu nebo maximálním počtem generačních cyklů.

3.3.3 Statistika populace

Další možnost ukončení běhu algoritmu je založena na pozorování vývoje populace. Můžeme například sledovat rozdíl mezi fitness hodnotou nejlepšího a nejhoršího jedince v dané populaci. K ukončení algoritmu pak dojde ve chvíli, kdy se tato hodnota bude nacházet v předem stanoveném intervalu. Použití tohoto ukončovacího kritéria by mělo být aplikováno s opatrností, neboť by mohlo dojít k předčasnému zastavení algoritmu. Jinou statistickou hodnotou, kterou můžeme u populace pozorovat, je směrodatná odchylka vhodnosti jedinců, nebo například největší vzdálenost mezi dvěma vektory v populaci. [12]

3.4 Varianty diferenciální evoluce

Až do této chvíle jsme uvedli pouze jednu variantu (strategii) diferenciální evoluce, kterou je DE/rand/1/bin. Tato strategie je považována za „klasickou“ a je velmi často využívána. Nicméně, existují i další strategie, které se liší ve způsobu vytváření šumového vektoru a také v použitém typu křížení.

Obecně lze libovolnou strategii diferenciální evoluce zapsat jako DE/x/y/z, přičemž:

- DE je zkratka pro diferenciální evoluci
- x označuje způsob, jakým bude vybírán aktivní jedinec
- y je počet diferenčních vektorů
- z určuje typ křížení - exp pro exponenciální a bin pro binomické

V tabulce 2 je uvedeno celkem 10 různých strategií diferenciální evoluce společně se vztahem, podle něhož se tvoří šumový vektor.

U strategií DE/rand-to-best/1/exp a DE/rand-to-best/1/bin je možné si všimnout výskytu parametru λ. Za účelem redukování počtu řídících parametrů se obvykle u těchto strategií používá $\lambda = F$. [15] Pro úplnost ještě doplníme, že x_{best} představuje nejlepšího jedince z populace.

3.5 Optimální nastavení algoritmu

Nastavení parametrů diferenciální evoluce je klíčový faktor, který ovlivňuje běh celého algoritmu. Hraje tak důležitou roli při tom, zda bude nalezeno optimální řešení daného problému, což nás jasně vede k závěru, že nastavení hodnot parametrů nelze podcenit. Při špatných hodnotách může velmi lehce dojít k situaci, že hledaný globální extrém nebude vůbec nalezen. Populace jedinců může například uváznout v lokálním extrému funkce, což způsobí, že žádní lepší jedinci již nebudou vytvořeni. Dalším nežádoucím chováním může být například situace, kdy sice dojde k nalezení globálního extrému, avšak celková

doba běhu algoritmu bude velmi dlouhá, což poukazuje na to, že se populace vyvíjela velmi pomalu.

Chceme-li výše zmíněným situacím předejít nebo jejich počet výskytů minimalizovat, je správné nastavení parametrů nutnou podmínkou. Jedná se však o úkol, který nemusí být vždy úplně snadný, jelikož neexistují žádné univerzální hodnoty parametrů, které bychom ve všech případech mohli označit jako správné. Vždy záleží na typu řešeného problému, lépe řečeno na účelové funkci.

Jak již bylo v rámci této práce naznačeno, nastavení parametrů většinou provádí uživatel před samotným spuštěním algoritmu. Jedná se o parametry CR, F, NP, ukončovací kritérium a případně lze vybrat i variantu DE (viz kapitola 3.4). O možných způsobech ukončení algoritmu pojednávala kapitola 3.3, proto se jimi nebudeme dále zabývat. Zaměříme se pouze na výběr vhodné varianty, velikost populace a hodnoty parametrů CR a F.

3.5.1 Výběr vhodné varianty

V tabulce 2 je uvedeno celkem deset základních variant diferenciální evoluce. Patrně nejčastěji používanou variantou je DE/rand/1/bin, což je v podstatě klasická varianta diferenciální evoluce. Dále se rovněž používají varianty DE/rand/1/exp a DE/best/1/z.[5] Typ použitého křížení není až tak důležitý, ačkoli Kenneth Price tvrdí, že binomické křížení nebude nikdy horší než exponenciální. [4]

Strategie	**Výpočet šumového vektoru**
DE/best/1/exp	$v = x^G_{best,j} + F \cdot (x^G_{r_2,j} - x^G_{r_3,j})$
DE/rand/1/exp	$v = x^G_{r_1,j} + F \cdot (x^G_{r_2,j} - x^G_{r_3,j})$
DE/rand-to-best/1/exp	$v = x^G_{i,j} + \lambda \cdot (x^G_{best,j} - x^G_{i,j}) + F \cdot (x^G_{r_1,j} - x^G_{r_2,j})$
DE/best/2/exp	$v = x^G_{best,j} + F \cdot (x^G_{r_1,j} + x^G_{r_2,j} - x^G_{r_3,j} - x^G_{r_4,j})$
DE/rand/2/exp	$v = x^G_{r_5,j} + F \cdot (x^G_{r_1,j} + x^G_{r_2,j} - x^G_{r_3,j} - x^G_{r_4,j})$
DE/best/1/bin	$v = x^G_{best,j} + F \cdot (x^G_{r_2,j} - x^G_{r_3,j})$
DE/rand/1/bin	$v = x^G_{r_1,j} + F \cdot (x^G_{r_2,j} - x^G_{r_3,j})$
DE/rand-to-best/1/bin	$v = x^G_{i,j} + \lambda \cdot (x^G_{best,j} - x^G_{i,j}) + F \cdot (x^G_{r_1,j} - x^G_{r_2,j})$
DE/best/2/bin	$v = x^G_{best,j} + F \cdot (x^G_{r_1,j} + x^G_{r_2,j} - x^G_{r_3,j} - x^G_{r_4,j})$
DE/rand/2/bin	$v = x^G_{r_5,j} + F \cdot (x^G_{r_1,j} + x^G_{r_2,j} - x^G_{r_3,j} - x^G_{r_4,j})$

Tabulka 2: Strategie diferenciální evoluce

[5] Symbol z zde představuje typ použitého křížení, tedy exponenciální či binomické.

Obecně můžeme tedy říci, že pokud nemáme žádný specifický důvod pro použití konkrétní varianty DE a nevíme tedy, jakou variantu přesně zvolit, nabízí se jako vhodná první volba DE/rand/1/bin. [4]

3.5.2 Velikost populace

Velikost populace bývá obvykle volena s ohledem na dimenzi problému, jinak řečeno s ohledem na počet parametrů jedince. Doporučená hodnota parametru NP se pohybuje v rozmezí $10D$ až $100D$, nicméně $100D$ se používá v případech, kdy je účelová funkce vysoce multimodální (obsahuje velké množství lokálních extrémů). [19] Rainer Storn ve svém článku [15] uvádí, že pro mnohé problémy je NP = $10D$ dobrou volbou. Samozřejmě ale existují případy, kdy byla hodnota NP $< 10D$ a i přesto diferenciální evoluce pracovala správně a nalezla optimální řešení. Pro úplnost je ještě potřeba zmínit, že velikost populace nesmí být nikdy menší než 4, jelikož pro vytváření potomků je u většiny variant zapotřebí čtyř rodičů.

3.5.3 Řídící parametry CR a F

Parametry CR a F představují práh křížení a mutační konstantu. Povolené hodnoty parametru CR se pohybují v rozsahu 0 až 1, přičemž ale není příliš vhodné, aby CR nabývalo těchto hraničních hodnot. Doporučené hodnoty pro tento parametr se pohybují v rozsahu 0,8 - 0,9 [19], nicméně bylo zjištěno, že pokud je optimalizovaná funkce separabilní[6], je vhodné nastavit CR na nižší hodnotu blízkou nule - např. 0,2. [4]

U mutační konstanty je k dispozici rozsah hodnot 0 - 2 [19], přičemž obvykle se hodnota volí v rozsahu 0,5 - 1 [15]. Rainer Storn dále v článku [15] uvádí, že čím vyšší je velikost populace, tím menší by měla být hodnota parametru F.

Lze tedy tvrdit, že pokud nemáme o účelové funkci mnoho informací, je vhodné vyzkoušet klasickou variantu DE/rand/1/bin s NP = 10D, F = 0,8 a CR = 0,9. [4] Rovněž je vhodné sledovat, jak se mění parametry nejlepšího jedince z populace. Dobrým ukazatelem správně fungujícího algoritmu může být vysoká proměnlivost parametrů nejlepšího jedince v průběhu evolučních cyklů (zejména zpočátku běhu algoritmu), a to i v případě, že fitness hodnota nejlepšího jedince klesá pomalu. [15]

3.6 Práce s celočíselnými a diskrétními parametry

Ve své základní podobě je diferenciální evoluce schopna pracovat pouze s reálnými parametry. V kapitole 2.3 však bylo uvedeno, že parametry jedince mohou být trojího typu - reálné, celočíselné a diskrétní. Rozšíření algoritmu diferenciální evoluce tak, aby byla schopna pracovat se všemi typy parametrů, je poměrně snadné, stačí pouze pár jednoduchých modifikací. [10]

V případě, že potřebujeme pracovat s celočíselným parametrem, stačí pouze daný parametr před dosazením do účelové funkce zaokrouhlit na celé číslo. Díky tomu může

[6] Separabilní funkcí se rozumí taková funkce, která může být vyjádřena jako součin funkcí o jedné proměnné. Formálněji zapsáno musí platit následující: $f(x_1, x_2, ...x_n) = f_1(x_1)f_2(x_2)...f_n(x_n)$

algoritmus stále pracovat s reálnými parametry a teprve v momentě, kdy je nutné vypočítat fitness hodnotu jedince, dojde k zaokrouhlení potřebných parametrů. [10]

Stejně jednoduchý je postup při práci s diskrétními parametry. Předpokládejme, že máme množinu obsahující povolené hodnoty parametrů. Počet prvků v množině je n. Jednotlivým hodnotám tak můžeme přiřadit indexy v rozsahu 1 až n. Diferenciální evoluce tak může používat pouze tyto indexy, s nimiž bude pracovat jako s celočíselným parametrem. Jinými slovy to znamená, že namísto toho, abychom optimalizovali přímo hodnotu diskrétního parametru, optimalizujeme hodnotu indexu. Teprve ve chvíli, kdy je potřeba získat hodnotu účelové funkce, dojde k použití skutečné hodnoty. [10]

3.7 Přehled možných vylepšení diferenciální evoluce

Od doby prvního představení diferenciální evoluce až do současnosti se mnoho odborníků, výzkumníků a vědců zabývá možnostmi, jak lze algoritmus diferenciální evoluce modifikovat, případně zkombinovat s jinými technikami, za účelem dosažení ještě lepších výsledků. Jednou z těchto možností je paralelní či distribuované zpracování algoritmu. Vzhledem k tomu, že hlavním tématem práce je distribuovaná DE, bude této problematice věnována celá kapitola 4. V této části kapitoly se pouze velmi stručně seznámíme s některými dalšími možnostmi, nebudeme je však s ohledem na hlavní cíle této práce rozebírat do hlubších detailů.

3.7.1 Technika zvaná Dither a Jitter

V části 3.5.3 bylo uvedeno doporučení, podle něhož lze nastavit hodnoty parametrů CR a F. Předpokládali jsme, že hodnota mutační konstanty se v průběhu algoritmu nemění, nicméně nikde v textu nebylo uvedeno, že tato hodnota musí zůstat konstantní.

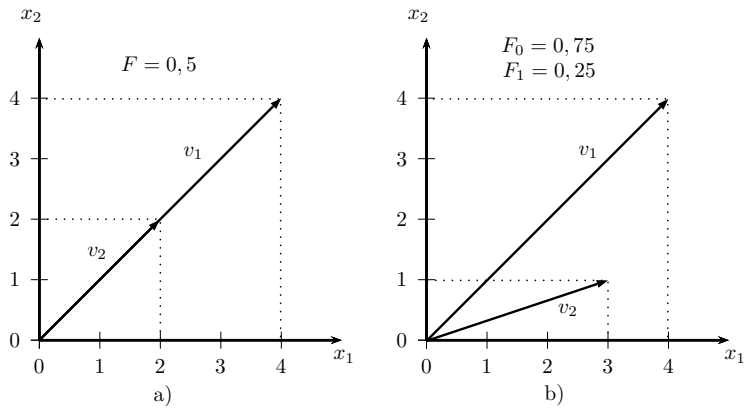
Jednou z možných jednoduchých modifikací algoritmu je tedy dynamická změna hodnoty tohoto parametru. Můžeme například sledovat vývoj fitness hodnoty nejlepšího jedince v populaci a pokud se během n generačních cyklů hodnota nezmění, je možné změnit hodnotu řídících parametrů. [19] Speciálními případy u kterých je jasně dáno, kdy se má změnit hodnota parametru F, jsou techniky zvané *dither* a *jitter*.[7]

V případě techniky *jitter* se hodnota F mění během mutačního procesu pro každý parametr a označuje se F_j, kde j představuje index daného parametru. Pokud se hodnota mění pro každý diferenční vektor, pak jde o techniku *dither*. Hodnota F se v tomto případě značí F_i, kde i označuje index aktuálního jedince v populaci.

Na první pohled by se možná mohlo zdát, že obě výše uvedené techniky se od sebe příliš neliší, nicméně z pohledu vektorů a vektorových operací je rozdíl celkem patrný. V případě, že všechny souřadnice diferenčního vektoru násobíme stejným číslem (*dither*), jedná se z hlediska vektorových operací o násobení vektoru reálným číslem. Je jasné, že taková operace pouze změní velikost násobeného vektoru - směr vektoru zůstane stejný.

V případě metody *jitter* je každá souřadnice diferenčního vektoru násobena jiným číslem F_j. Následkem toho vznikne nový vektor (v terminologii DE označován jako

[7] Vzhledem k nehodícím se českým překladům slov *dither* a *jitter* bylo v textu ponecháno anglické označení.



Obrázek 2: Porovnání technik *dither* a *jitter*

Obrázek a) ukazuje techniku *dither*. Je vidět, že se mění pouze velikost vektoru, směr zůstává stejný. U techniky *jitter* na obrázku b) je patrné, že došlo nejen ke změně velikosti, ale také ke změně směru.

váhový diferenční vektor), který oproti vektoru diferenčnímu má nejen jinou velikost, ale i směr. Celé toto vysvětlení lépe objasní obrázek 2, kde vektor v_1 představuje diferenční vektor a v_2 váhový diferenční vektor.

3.7.2 Opposition-Based Differential Evolution

Algoritmus Opposition-Based Differential Evolution (OBDE) byl publikován v článku [13]. Ve stručnosti se jedná o standardní algoritmus DE, u něhož jsou provedeny mírné modifikace během generování prvotní populace a při dokončení každého generačního cyklu, kdy je stará populace nahrazována novou. Využívají se zde tzv. *„protilehlá“* řešení (odtud název Opposition-Based DE), která se tvoří podle definice 3.1. Pro podrobnější informace a popis OBDE odkazujeme čtenáře na článek [13].

Definice 3.1 *Nechť $P = (x_1, x_2, ...x_D)$ je bod v D-rozměrném prostoru, kde $x_1, x_2, ...x_D \in \mathbb{R}$ a $x_i \in [a_i, b_i]\ \forall i \in \{1, 2, ...D\}$. „Protilehlý“ bod $\check{P} = (\check{x}_1, \check{x}_2, ...\check{x}_D)$ je pak definován souřadnicemi, pro něž platí:*

$$\check{x}_i = a_i + b_i - x_i$$

3.7.3 Rozšíření DE o algoritmus Local Search

Poslední modifikací algoritmu DE, kterou si v této kapitole uvedeme, je jeho rozšíření o algoritmus Local Search (LS) neboli lokální hledání. Jedná se o velmi jednoduchý algoritmus, který při svém startu vytvoří počáteční řešení problému, které je typicky generováno náhodně. K tomuto řešení se následně vygeneruje množina sousedních řešení. Mezi sousedy se najde nejlepší řešení a v případě, že je lepší než řešení výchozí, vezme se jako nové aktuální nejlepší řešení. Poté se celý postup generování sousedních řešení opakuje

do doby, než nastane situace, kdy mezi sousedy nebude existovat žádné zlepšující řešení. [19]

Jakým způsobem využít LS v algoritmu DE pak již záleží na konkrétním návrhu a implementaci celého algoritmu.

V článku [6] byla například uvedena varianta Local Search Differential Evolution (LSDE), u níž je pro každého jedince s pravděpodobností p generován sousední jedinec X_i' podle vztahu

$$X_i' = r_1 \cdot X_i + r_2 \cdot (X_i - X_{best}),$$

kde X_i představuje jedince v populaci na pozici i, X_{best} je nejlepší jedinec z populace a r_1, r_2 jsou náhodná čísla z rozsahu 0 - 1, přičemž platí $r_1 + r_2 = 1$. Více informací o LSDE je možno získat v článku [6].

4 Distribuované zpracování diferenciální evoluce

V dnešní pokročilé době, kdy máme k dispozici již velmi rychlé a výkonné počítače, není příliš velkým divem, že mnoho aplikací vykonává více operací najednou. Celá řada takových aplikací je vytvořena tak, že pracují ve více vláknech, přičemž každé vlákno obvykle provádí odlišnou činnost. Takové zpracování nazýváme paralelní. Pokud máme k dispozici více počítačů, nemusíme se omezovat pouze na zpracování ve více vláknech, ale můžeme algoritmus řešící zadaný problém spustit na větším množství počítačů - pak hovoříme o distribuovaném algoritmu.

J.R.Koza ve své publikaci [9] uvádí, že existují dva základní přístupy, jak lze vytvořit distribuovanou verzi genetických algoritmů. První přístup, v němž je celková populace rozdělena na několik menších subpopulací, lze uplatnit i na algoritmus DE. Základní koncept je zde takový, že se každá subpopulace vyvíjí samostatně a po určitém počtu generačních cyklů dochází k migraci některých jedinců z jedné subpopulace do jiné. [9]

V kapitole 4 se seznámíme s konkrétními možnostmi, jak lze vytvořit distribuovaný algoritmus diferenciální evoluce. V rámci této práce byly vytvořeny dvě verze distribuované DE, které jsou popsány v následujících dvou podkapitolách 4.1 a 4.2. V dalších dvou podkapitolách jsou pak navíc představeny další dvě varianty distribuované diferenciální evoluce.

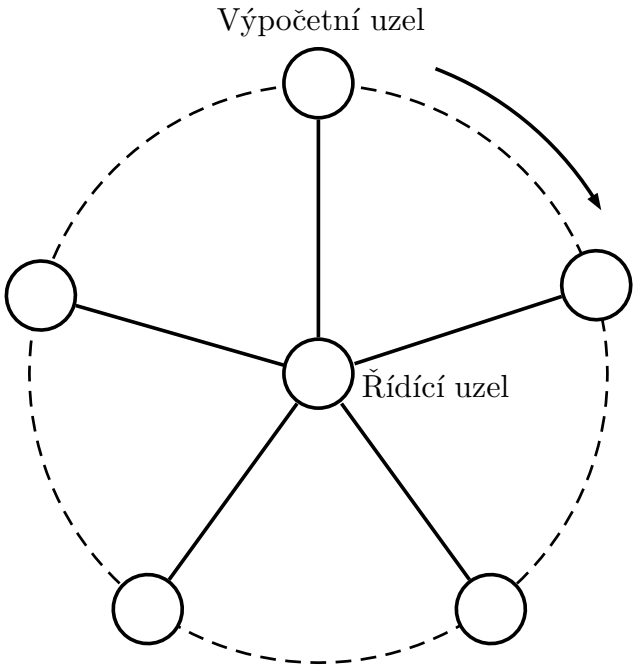
4.1 Single Slave Single Core

Předtím, než přejdeme k samotnému popisu, je třeba podotknout, že termín „Single Slave Slave Core“ není obecně používaným pojmem a byl zvolen autorem pouze pro pojmenování níže popsaného způsobu. Ve vytvořené řídící aplikaci je tento termín použit pro vybrání způsobu provedení algoritmu DE, tudíž pro zachování jednotnosti pojmenování byl termín ponechán i v textové části práce.

Jedná se patrně o nejjednodušší způsob, pomocí kterého lze algoritmus diferenciální evoluce učinit distribuovaným. Princip je prostý - na každém počítači spustíme diferenciální evoluci a poté ze získaných výsledků vybereme ten nejlepší. V tomto případě tedy neprobíhá žádná spolupráce mezi jednotlivými počítači, které zpracovávají algoritmus. Na začátku jim pouze řídící počítač přes síť rozešle parametry a další informace potřebné pro provedení algoritmu (účelová funkce, specimen, atd...). Každý počítač poté provede samostatnou diferenciální evoluci a svůj výsledek odešle zpět na řídící počítač. Ten nakonec ze všech došlých výsledků vybere ten nejlepší, který představuje nejlepší nalezené řešení problému.

4.2 Parallel Differential Evolution

Druhou variantou distribuované diferenciální evoluce, která byla v rámci této práce implementována, je tzv. paralelní diferenciální evoluce (PDE) publikovaná v článku [16]. Název se může na první pohled zdát mírně matoucí, neboť by se dalo předpokládat, že se jedná o vícevláknové zpracování na jednom počítači. Autoři použili tzv. *parallel virtual machine* (PVM), což je sada softwarových nástrojů a knihoven, přičemž hlavním účelem



Obrázek 3: Topologie u paralelní diferenciální evoluce [17]

PVM je propojit počítače tak, aby se jevily jako jeden výkonný virtuální počítač. [16] Zřejmě díky použití PVM autoři zvolili název paralelní diferenciální evoluce.

Hlavním rozdílem u PDE oproti způsobu popsanému v kapitole 4.1 je fakt, že jednotlivé počítače (výpočetní uzly) při provádění algoritmu spolupracují. Celková populace je rozdělena na menší subpopulace, přičemž každý výpočetní uzel pracuje se svou jednou subpopulací. Velikost subpopulace je rovna $\frac{NP}{m}$, kde m představuje počet výpočetních uzlů. Tyto uzly jsou pomyslně uspořádány do kruhu, uvnitř kterého se nachází řídící uzel (*Master*), který zajišťuje výměnu dat mezi výpočetními uzly, které o sobě navzájem neví. Skutečná topologie propojení je tedy hvězda, jak je znázorněno na obrázku 3.

Hlavním úkolem řídícího uzlu je zajišťovat tzv. migraci jedinců. Každý výpočetní uzel provádí klasickou diferenciální evoluci se svou subpopulací, avšak po každém generačním cyklu může dojít k migraci jedince z jedné subpopulace do druhé. Proces migrace probíhá tak, že se vytvoří kopie nejlepšího jedince z dané subpopulace, která se poté skrze řídící uzel odešle sousednímu výpočetnímu uzlu. Jakmile uzel obdrží nového jedince, vybere náhodně jednoho jedince ze své subpopulace (kromě nejlepšího), kterého nahradí nově příchozím jedincem. [17]

Klíčovým faktorem, který rozhoduje o tom, zda bude či nebude migrace mezi dvěma subpopulacemi provedena, je parametr ϕ, který může nabývat hodnot v intervalu $\langle 0, 1 \rangle$. Po každém generačním cyklu je pro každou subpopulaci generováno náhodné číslo v rozsahu 0 až 1 a pokud toto číslo je menší než hodnota parametru ϕ, bude provedena migrace, v opačném případě nikoli. [17]

Celý výše uvedený princip činnosti algoritmu zachycují pseudokódy 4 a 5 převzaté z [17].

Pseudokód 4 Paralelní diferenciální evoluce - řídící uzel

```
1: Spawn N sub-populations, each one on a different processor
2: for each generation do
3:    Receive an individual from each sub-population
4:    for each received individual do
5:       if rand(0,1) < φ then
6:          Send the individual to the next sub-population in the ring
7:       end if
8:    end for
9:    if the stop criterion for the objective function is met then
10:      Send a termination signal to all the sub-populations
11:   end if
12: end for
```

Pseudokód 5 Paralelní diferenciální evoluce - výpočetní uzel

```
1: for each generation do
2:    Perform a DE generation
3:    Send a copy of the best individual to the master node
4:    if a migrated individual has been received then
5:       Replace a random individual, different from the best, by this migrated individual
6:    end if
7:    if a termination signal has been received then
8:       Terminate the execution
9:    end if
10: end for
```

4.3 Island Based Distributed Differential Evolution

Island Based Distributed Differential Evolution (IBDDE) představuje další způsob, jakým lze vytvořit distribuovaný algoritmus diferenciální evoluce. Jedná se o lehce modifikovanou verzi PDE, která byla představena v předchozí části této kapitoly. Opět platí, že celková populace je rozdělena na m subpopulací, kde každá z nich obsahuje $\frac{NP}{m}$ jedinců. Rozdíl oproti PDE spočívá pouze v migraci, která je zde kontrolována pětící $M = (\gamma, \rho, \phi_s, \phi_r, \tau)$. Význam jednotlivých parametrů je následující: [17]

- γ - počet generací mezi dvěma migracemi ($\gamma \in \mathbb{N}$)
- ρ - počet jedinů, kteří migrují z jedné subpopulace během jedné migrace ($\rho \in \mathbb{N}$)
- ϕ_s - funkce, která z dané subpopulace vrací jedince určené k migraci

- ϕ_r - funkce, která ze subpopulace vybere jedince, kteří budou nahrazeni migrujícími jedinci
- τ - pravidlo, které vybere cílovou subpopulaci (tedy subpopulaci, do níž budou umístěni migrující jedinci)

Je tedy vidět, že oproti PDE se IBDDE liší především v tom, že migrace nemusí nutně probíhat po každém generačním cyklu, nemusí být vždy migrován pouze jeden jedinec a migrace nemusí být provedena pouze mezi sousedními subpopulacemi.

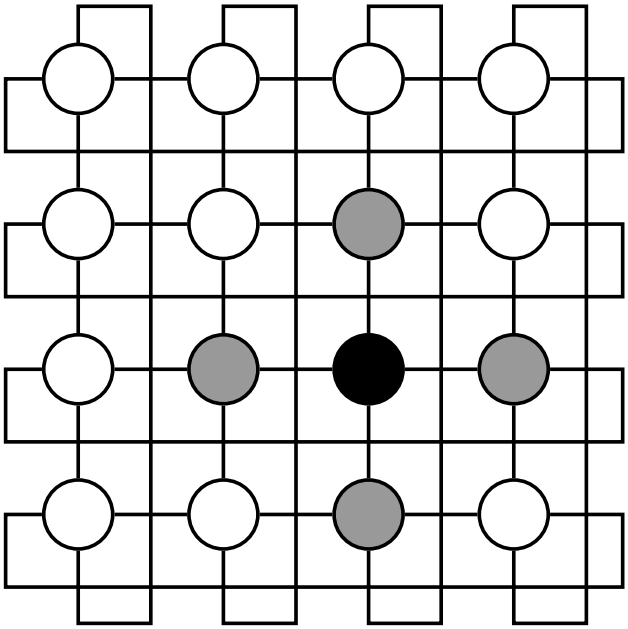
4.4 Distributed Differential Evolution

Posledním způsobem distribuovaného zpracování diferenciální evoluce, který si v této kapitole představíme, je Distributed Differential Evolution (DDE). U této varianty se opět nachází několik výpočetních uzlů, které provádějí diferenciální evoluci se svou subpopulací. V porovnání s BDE či IBDDE, kde jsou ale výpočetní uzly pomyslně uspořádány do kruhu, je u DDE každý výpočetní uzel propojen s dalšími μ uzly. Jinými slovy, každý výpočetní uzel má přesně μ sousedů. Na obrázku 4 je zachyceno schéma propojení, kde $\mu = 4$. Černě je vyznačen jeden výpočetní uzel a šedou barvou jsou označeny jeho čtyři sousední uzly. O ostatních uzlech, vyznačených bílou barvou, černý výpočetní uzel neví a komunikuje s nimi pouze nepřímo skrze své sousedy. Žádný výpočetní uzel tedy neví, kolik uzlů celkem se podílí na provádění algoritmu.

Vždy po určitém počtu generačních cyklů, který se zde označuje M_I (migration interval; migrační interval), dochází mezi sousedními výpočetními uzly k výměně jedinců. Počet jedinců, kteří budou odesláni do sousedních populací, je zde řízen parametrem M_R (migration rate; míra migrace). Všichni jedinci v rámci jednoho výpočetního uzlu, kteří jsou určení k migraci (označme jejich počet S_I), jsou poté posláni všem sousedním uzlům, což znamená, že každý uzel přijme celkem $S_I \cdot \mu$ nových jedinců. [3] Přijatí jedinci poté mohou nahradit nejhorší jedince v dané subpopulaci, nebo je mohou nahradit pouze v případě, že mají nižší fitness hodnotu, případně mohou nahradit náhodně vybrané jedince s výjimkou nejlepšího jedince.[2]

V literatuře se uvádí, že příliš častá migrace mnoha jedinců značně degraduje výkonnost algoritmu, stejně tak jako migrace, které jsou prováděny velmi zřídka s malým množstvím jedinců. [1] V případě, že migrace probíhají příliš často, je proces hledání globálního extrému v rámci jedné subpopulace příliš často narušován nově příchozími jedinci, což má ve výsledku negativní dopad na výkonnost celého algoritmu. [3]

Na závěr ještě dodejme, že v DDE není bezpodmínečně nutné, aby byl přítomen řídící uzel. Pokud bychom ale takový uzel přidali, jeho hlavním úkolem by bylo shromažďovat nejlepší nalezená řešení v jednotlivých subpopulacích, která by poté mohl prezentovat uživateli.



Obrázek 4: Topologie u distribuované diferenciální evoluce [17]

Šedé výpočetní uzly představují čtyři sousedy černě zvýrazněného uzlu. Každý uzel má přesně čtyři sousedy, proto $\mu = 4$.

5 Neuronové sítě

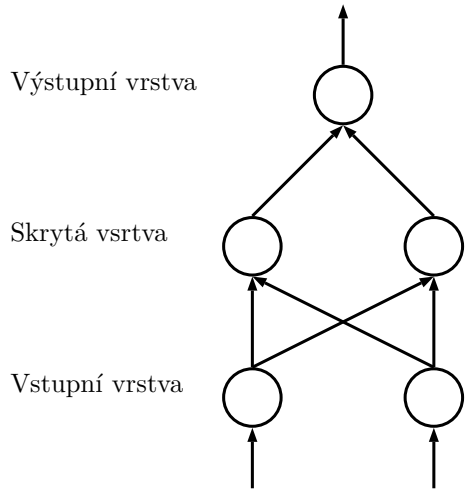
I navzdory faktu, že hlavním tématem této práce je algoritmus diferenciální evoluce, je potřeba čtenáře alespoň zevrubně seznámit s problematikou neuronových sítí, jelikož při provádění experimentů byla společně s testovacími funkcemi použita i jednoduchá neuronová síť pro řešení problému XOR. Cílem této kapitoly je tedy podat základní informace o tom, jak neuronové sítě vznikly, jakým způsobem pracují a jaké je jejich využití.

5.1 Popis fungování neuronových sítí

Historie vzniku neuronových sítí se datuje do první poloviny 20. století, kdy W. S. McCulloch publikoval první práci o neuronech a jejich modelech. Později pak McCulloch společně se svým studentem vypracoval model neuronu, který je v podstatě používán dodnes. Inspirací ke vzniku neuronových sítí byla snaha napodobit lidské myšlení, resp. činnost lidského mozku, v němž se nachází obrovské množství neuronů, přibližně 10^{10}. [21] Každý takový neuron v lidském mozku dostává zprávy od jiného neuronu pomocí dendritů, což jsou úpony rašící z jednoho konce neuronu. Na jeho druhém konci se nachází dlouhé vlákno zvané axon. Ten se může připojit až k deseti tisícům jiných neuronů pomocí jejich dendritů. Spojení mezi nimi nazýváme synapse, jež slouží jako brány regulující tok informací v mozku. Do synapsí jsou uvolňovány chemické látky nazývané neurotransmitery, které mění pohyb impulzů miliardami možných cest mozkem. [8]

Tento velmi zjednodušený popis činnosti mozku se snaží napodobit technické neuronové sítě. Jak již jejich název napovídá, základním stavebním kamenem je neuron podobný tomu, který se nachází v lidském mozku. Samozřejmostí je, že podoba neuronu je velmi zjednodušená v porovnání s neuronem biologickým, nicméně i přesto je pro použití v neuronových sítích dostačující. Model neuronu je v podstatě jednotka, která má několik vstupů a pouze jeden výstup. Každý ze vstupů má přiřazenou svoji váhu, což je bezrozměrné číslo, které určuje, jaký význam má daný vstup pro konkrétní neuron. [21] Uvnitř těla modelu neuronu se nachází tzv. přenosová funkce. S těmito znalostmi je již možné přistoupit k jednoduchému vysvětlení činnosti modelu neuronu. Hodnoty vstupů neuronu jsou vynásobeny odpovídajícími vahami, získané součiny jsou poté sečteny a výsledek je dosazen do přenosové funkce. Výstup funkce pak tvoří hodnotu výstupu daného neuronu. [20]

Neuronové sítě se skládají z několika výše popsaných neuronů, které jsou organizovány do vrstev. Podle počtu vrstev se pak sítě mohou dělit na sítě s jednou vrstvou či sítě s více vrstvami. Pro topologii vícevrstvých sítí obvykle platí, že všechny neurony v rámci jedné vrstvy jsou spojeny se všemi neurony ve vyšší vrstvě. [21] Jeden z mnoha příkladů, jak může jednoduchá vícevrstvá neuronová síť vypadat, je zobrazen na obrázku 5. Obrázek představuje neuronovou síť s celkem pěti neurony uspořádanými do tří vrstev. U takových sítí platí, že neurony v první vrstvě pouze distribuují vstupní hodnoty do další vrstvy. Díky faktu, že se typicky jedná o vícebodový vstup do sítě, hovoří se o vstupních (příp. výstupních) vektorech informací. Síť na obrázku obsahuje ve výstupní vrstvě pouze jeden neuron, což ovšem není žádným pravidlem. Větší neuronové sítě mohou



Obrázek 5: Neuronová síť s jednou skrytou vrstvou

mít (a typicky také mívají) ve výstupní vrstvě více neuronů. Obrázek 5 byl zvolen pouze pro svou jednoduchost. Jednotlivé šipky v obrázku představují spojení mezi neurony, přičemž je zde vidět obvyklé (ne však nutné) výše uvedené pravidlo, že všechny neurony v jedné vrstvě jsou spojeny se všemi neurony ve vyšší vrstvě.

5.1.1 Přenosové funkce

Jak již bylo uvedeno, v těle modelu neuronu se nachází přenosová funkce, která transformuje vstupní signál na signál výstupní. Její volba hraje důležitou roli při tvorbě sítě a závisí na typu problému, který má síť řešit. Pokud například chceme provádět klasifikaci pouze mezi dvěma třídami (např. dobrý vs. špatný výrobek), pak bude dostačující binární funkce. Přenosových funkcí existuje více, mezi nejpoužívanější však patří: [20]

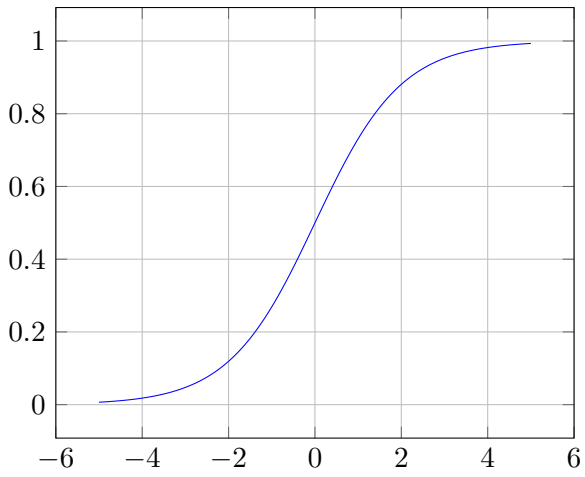
- perceptron
- binární
- logistická (nebo též sigmoida)
- hyperbolický tangens

V této práci byly použity dvě z výše uvedených funkcí, konkrétně binární a logistická. Omezíme se proto pouze na popis těchto dvou funkcí.

V případě binární funkce se jedná o jednoduchou dvouhodnotovou funkci, jejímž výstupem může být hodnota 0 nebo 1. Její zápis vypadá následovně:

$$F(B) = 1 \, \forall x > 0 \, \text{jinak} \, x = 0$$

Za zápisu je zřejmé, že se jedná o funkci, která pro všechny vstupy větší než 0 vrátí hodnotu 1 a pro vstupy menší nebo rovny 0 vrací hodnotu 0.



Obrázek 6: Graf logistické funkce

Logistická funkce je jedna z nejpoužívanějších funkcí a byla odvozena jako aproximace přenosové funkce biologického neuronu. Následuje její matematický zápis.

$$F(S) = \frac{1}{1+e^{-x}}$$

Výstupem této funkce je hodnota z intervalu (0; 1), což je dobře vidět z grafu funkce na obrázku 6.

5.2 Proces učení

Využití neuronových sítí je velmi široké, neboť existuje rozsáhlé spektrum disciplín, v nichž neuronové sítě nacházejí své uplatnění. Lze je například používat pro rozpoznávání tváří, symbolů, objektů apod. Obecně lze říci, že neuronové sítě slouží k řešení klasifikačních problémů. Jsou tedy schopny pro daný vstupní vektor určit, do jaké skupiny tříd vektor patří. Jako příklad lze uvést rozpoznávání čísel na obrázku. Pokud by byl daný obrázek vhodně reprezentován číselným způsobem ve formě vektoru, lze vytvořit síť takovou, která bude schopna pro jednotlivé obrázky vydat na svém výstupu číslo, které se na daném obrázku nachází. Tento příklad je samozřejmě jen jeden z mnoha problémů, které jsou neuronové sítě schopny řešit.

Důležitým krokem k tomu, aby neuronová síť byla schopna správně klasifikovat jednotlivé vstupní vektory, je její naučení čili adaptace na řešený problém. To se provádí tak, že se upravují váhy vstupů jednotlivých neuronů v síti. Byly vyvinuty různé učící algoritmy, které lze obecně rozdělit do dvou základních skupin:

- učení s učitelem
- učení bez učitele

Pro naučení sítě je potřebná tzv. trénovací množina, která obsahuje skupinu vektorů obsahujících informace o daném problému. V případě učení sítě s učitelem je množina tvořena dvojicemi vektorů reprezentující vstup a k němu požadovaný výstup. Při učení bez učitele obsahuje trénovací množina pouze vstupní vektory. [21]

Proces učení sítě je složen ze dvou fází, a to z adaptační a aktivační. V aktivační fázi se síti předloží vektor z trénovací množiny, síť tento vektor zpracuje a vydá vektor výstupní. Ten se porovná s požadovaným výstupním vektorem, čímž vznikne rozdíl (lokální chyba), který se uloží. Poté následuje adaptační fáze, jejímž cílem je minimalizovat lokální chybu, což se provádí tak, že se přenastavují jednotlivé váhy neuronů. Následně se opět opakuje aktivační fáze, znovu se získá lokální chyba, která se přičte k předchozí chybě, proběhne adaptační fáze a celý tento proces se opakuje tak dlouho, než se projde celá trénovací množina. Tím dojde k dokončení tzv. jedné epochy. Součet všech získaných lokálních chyb se nazývá globální chyba. Je-li splněna podmínka, že hodnota globální chyby je menší než požadovaná chyba, proces učení končí. [21]

Jinými slovy, proces učení není nic jiného než nalezení optimální kombinace vah sítě tak, aby globální chyba byla pokud možno co nejmenší. Jedná se tak v podstatě o optimalizační problém, který lze řešit např. pomocí diferenciální evoluce, což bylo provedeno během experimentů, které jsou popsány v kapitole 8.

6 Implementace

Jak jsme již zmínili, v rámci této práce byly vytvořeny dva způsoby distribuované diferenciální evoluce, které byly popsány v kapitolách 4.1 a 4.2. V této části práce se blíže podíváme na způsob, jakým byl celý projekt implementován.

Jedním ze stanovených cílů byl požadavek, aby vytvořená aplikace nebyla závislá na použité platformě. Jinak řečeno, aplikaci musí být možné spustit na všech běžně používaných operačních systémech, kterými jsou Windows, Linux a OS X. Z tohoto důvodu byl veškerý zdrojový kód vytvořen v jazyce Java. Kód jazyku Java je kompilován do tzv. bajtového kódu, který je uložen v souborech s příponou `.class`. Bajtový kód je následně interpretován JVM (Java Virtual Machine), který kód přeloží do strojového jazyka. JVM je dnes dostupný pro většinu druhů počítačů. Díky tomu je tak aplikace napsaná v Javě nezávislá na typu použité platformy.

Veškerá implementace byla provedena ve vývojovém prostředí IntelliJ IDEA ve verzi 14.0.2 a vývoj probíhal na platformě Windows 7 Professional 64-bit. Během implementace byly vytvořeny celkem tři samostatné projekty. Jedná se o dvě spustitelné aplikace a jednu knihovnu, kterou obě aplikace využívají ke své činnosti. Vytvořené aplikace představují řídící aplikaci (tzv. *Master*) a aplikaci provádějící algoritmus diferenciální evoluce (tzv. *Slave*). Obě tyto aplikace si společně se sdílenou knihovnou popíšeme v následujících částech této kapitoly.

Vzhledem k tomu, že všechny projekty obsahují relativně velké množství tříd a rozhraní, nebudeme v této kapitole zacházet až do úplných implementačních detailů jednotlivých tříd a omezíme se pouze na popis projektů a představení funkcí a úkolů, za něž jsou jednotlivé projekty zodpovědné, společně s vysvětlením, jak je zdrojový kód strukturován. Dále také zmíníme některé stěžejní třídy u nichž uvedeme, jakou funkci zajišťují. Bližší popis jak lze aplikaci spustit, ovládat a konfigurovat, se nachází v příloze na konci této práce. Detailní popis celého zdrojového kódu pak poskytne dokumentace, která je dostupná na přiloženém CD ve formátech PDF a HTML.

6.1 Řídící aplikace (Master)

Jak již samotný název naznačuje, jedná se o aplikaci, která podle terminologie použité v kapitole 4 představuje řídící uzel. Jejím klíčovým úkolem je zajišťovat komunikaci s jednotlivými výpočetními uzly a řídit tak celý průběh diferenciální evoluce. Pro snadnější ovládání uživatelem obsahuje grafické uživatelské rozhraní (GUI - Graphic User Interface), díky kterému může uživatel provádět nastavení parametrů algoritmu, výběr výpočetních uzlů, výběr účelové funkce a další nastavení.

Všechny důležité funkce a možnosti, které tato aplikace nabízí, můžeme shrnout do následujícího přehledu.

- **Nastavení parametrů zadaných uživatelem.** GUI aplikace obsahuje pro nastavení celkem čtyři hlavní grafické panely, které znázorňuje obrázek 7. První panel slouží k výběru jednotlivých výpočetních uzlů, které budou zapojeny do zpracování algoritmu. Na druhém panelu lze nastavit parametry algoritmu a variantu

(DE/rand/1/bin, atd...). Třetí panel je určen k definování vzorového jedince, který může být zadán buď přímo skrze GUI, nebo může být importován ze souboru XML. Poslední panel slouží k výběru typu distribuované DE, která bude použita. V celém GUI nechybí samozřejmě také možnost vybrat účelovou funkci a počet opakování algoritmu.

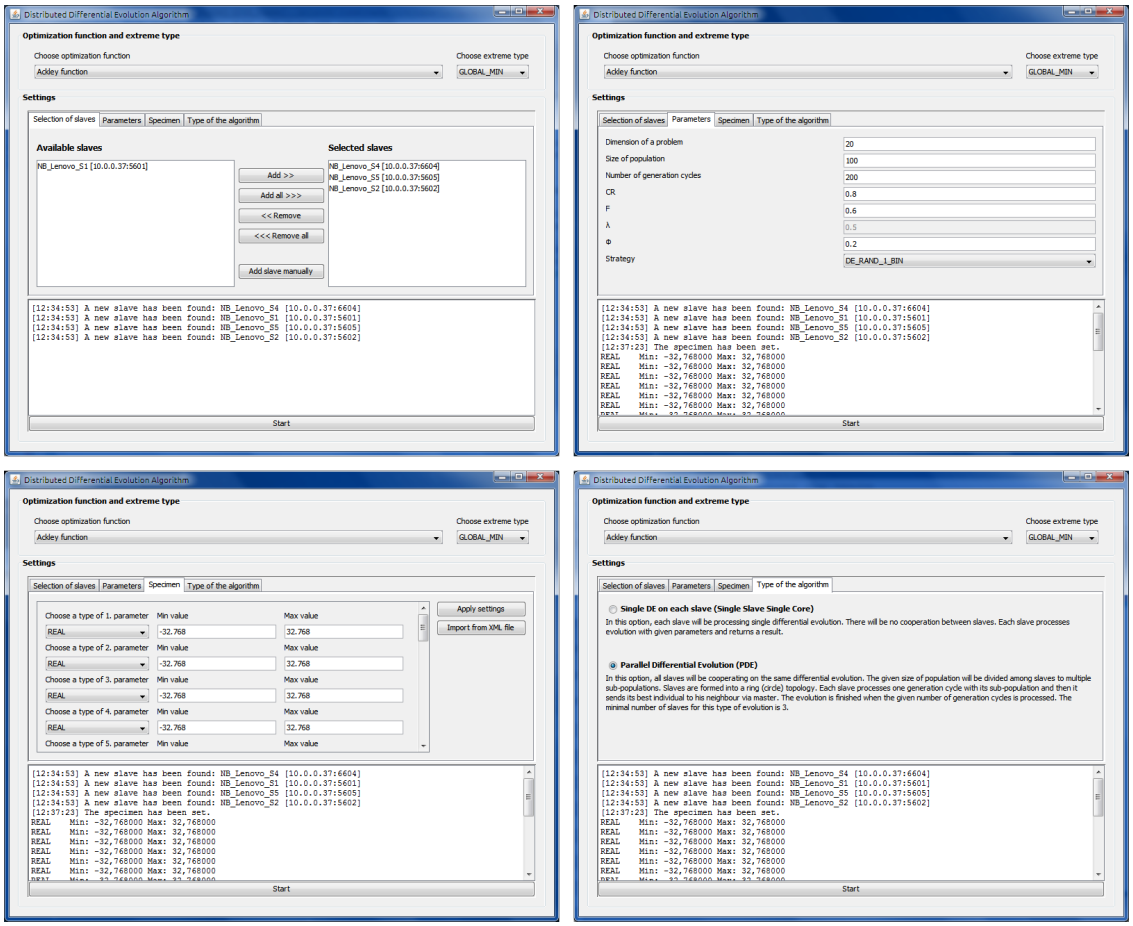
- **Nacházení a přidávání výpočetních uzlů.** Jednotlivé výpočetní uzly odesílají periodicky zprávu o tom, že jsou připraveny k provádění algoritmu. Součástí této zprávy je také číslo portu, na kterém daná výpočetní aplikace naslouchá. Aby nebylo nutné při každém spuštění výpočetní aplikace zadávat IP adresu řídící aplikace, je tento problém vyřešen tak, že jednotlivé výpočetní uzly odesílají datagram skrze multicast. Stačí tedy, aby řídící i výpočetní aplikace měly nastavenu stejnou skupinovou adresu. Multicastové adresy začínají bajtem v rozsahu 224 - 239. [11] Nastavení skupinové adresy aplikace se provádí v konfiguračním souboru.
  V případě, že by skupinové adresy nebyly nastaveny stejně, je zde také možnost přidat výpočetní uzel ručně přímým zadáním IP adresy a čísla portu.

- **Komunikace s výpočetními uzly a řízení průběhu algoritmu.** Kromě výše uvedeného případu s multicastem, kde se pro přenos využívá protokol UDP, probíhá již veškerá další komunikace skrze spolehlivý protokol TCP. Do této komunikace patří odeslání parametrů algoritmu na jednotlivé výpočetní uzly, příjem výsledků od těchto uzlů a v případě paralelní DE také řízení migrace jedinců. IP adresu a port pro komunikaci s daným výpočetním uzlem aplikace zjistí z přijatého datagramu.

- **Zpracování a prezentace výsledků.** Po dokončení provedení algoritmu odesílají výpočetní uzly zprávy s výsledky. Řídící aplikace tyto zprávy přijme a vyhodnotí výsledky, které poté uživateli zobrazí skrze GUI. Uživatel tak může vidět nejlepšího a nejhoršího nalezeného jedince společně s jejich postupným vývojem v rámci všech evolučních cyklů. Tento průběh je pro větší přehlednost zanesen také do grafu, který si uživatel může uložit do souboru. Nechybí zde ani možnost exportu získaných výsledků do souborů TXT nebo CSV. Obrázek 8 prezentuje, jak vypadá okno s přehledem výsledků. Pro tvorbu grafů byla použita knihovna *jmathplot*, která je zdarma dostupná na adrese `https://github.com/yannrichet/jmathplot`.

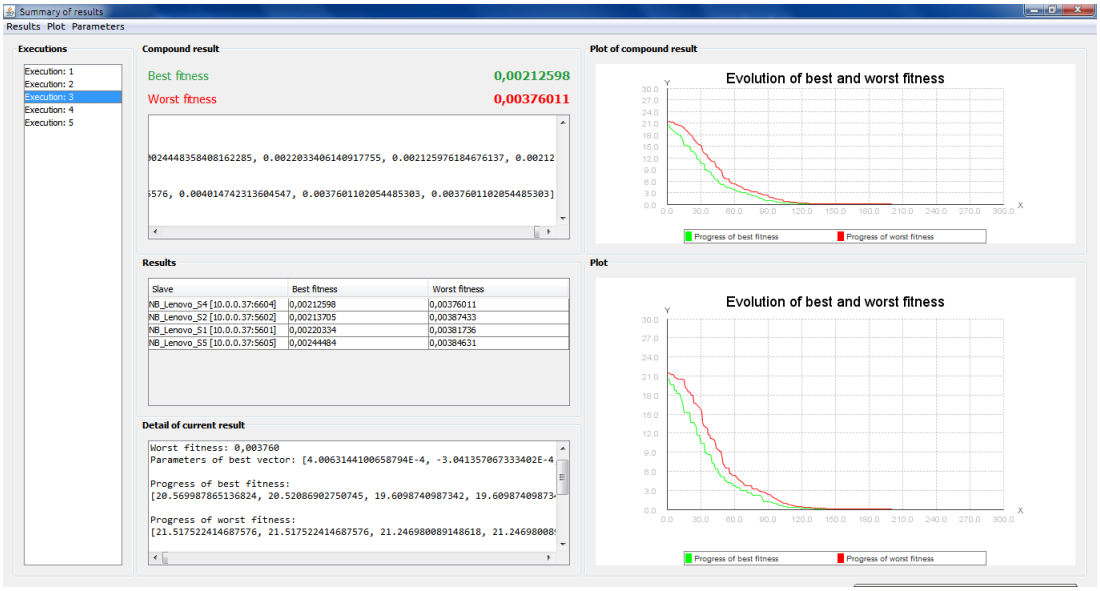
Výše uvedené funkce a možnosti zajišťuje zdrojový kód rozdělený do celkem 66 tříd a rozhraní, které jsou uspořádány do několika balíčků. Každý balíček sdružuje kód, který je zodpovědný za jednu ucelenou logickou část aplikace. Jedná se o následující balíčky:

- **`core`** Sdružuje nejdůležitější třídy celé aplikace. Ty jsou zodpovědné za většinu klíčových funkcí, které řídící uzel provádí. Pro lepší přehlednost a orientaci v kódu je balíček dále dělen do několika menších balíčků.
  - **`core.driver`** V tomto balíčku jsou umístěny třídy, které slouží pro spuštění a řízení průběhu algoritmu.
  - **`core.result`** Obsahuje třídy zodpovědné za zpracování výsledků a jejich možný export do souborů, dále také třídy pro tvorbu grafů.



Obrázek 7: Čtyři panely řídící aplikace umožňující uživatelské nastavení

Levý horní obrázek zobrazuje panel pro výběr jednotlivých výpočetních uzlů. V pravé horní části se vyskytuje panel pro nastavení parametrů algoritmu. V dolní části jsou zleva panely pro nastavení vzorového jedince a pro výběr typu distribuovaného zpracování diferenciální evoluce.



Obrázek 8: Okno s přehledem získaných výsledků

  - **`core.xml`** Zajišťuje načtení vzorového jedince ze souboru XML.

- **`logging`** Tento balíček obsahuje pouze dvě třídy, které poskytují možnost monitorovat běh aplikace. Třídy umožňují, aby veškeré důležité události byly zapisovány do *log* souboru.

- **`network`** Obsah tvoří třídy pro nacházení výpočetních aplikací v síti a také třída umožňující přidat výpočetní uzel zadáním čísla portu a IP adresy. Zbylé třídy, které zajišťují komunikaci během provádění algoritmu, jsou umístěny ve společné knihovně.

- **`ui`** V tomto balíčku jsou umístěny všechny třídy, které vytváří grafické uživatelské rozhraní. Součástí jsou také dva další balíčky dále oddělující některé prvky GUI.

  - **`ui.dialog`** Obsahuje dialogová okna a třídy, které s nimi bezprostředně souvisí.
  - **`ui.validation`** Balíček je tvořen třídami použitými pro validaci dat, která uživatel zadává.

Mezi velmi důležité třídy celého projektu patří abstraktní třída `AlgorithmDriver` a její dva potomci, třídy `BasicDEDriver` a `DistributedDEDriver`. Jak již název samotných tříd napovídá, jde o třídy, které řídí zpracování algoritmu. Obě tyto třídy pochopitelně vykonávají kód v jiném vlákně, aby nebylo narušeno například ovládání aplikace, případně další funkce.

Úkolem třídy `BasicDEDriver` je rozeslat jednotlivým výpočetním uzlům parametry algoritmu zadané uživatelem a dále čekat, až jednotlivé uzly dokončí zpracování. Třída

poté přijme výsledky, které následně předá k dalšímu zpracování nadřazeným třídám. Dalším zpracováním se zde rozumí především grafické zobrazení výsledků uživateli.

Za řízení diferenciální evoluce, u níž dochází k migraci jedinců, je zodpovědná třída s názvem `DistributedDEDriver`. Její hlavní úkoly lze přehledně shrnout v následujícím výčtu:

- Odeslání parametrů jednotlivým výpočetním uzlům. Součástí těchto parametrů je také informace o tom, že bude prováděna paralelní DE, díky čemuž jednotlivé výpočetní uzly ví, že po každém generačním cyklu mají odeslat kopii svého nejlepšího jedince a čekat na zprávu od řídící aplikace.
- Přijetí kopie nejlepších jedinců od všech výpočetních uzlů po každém generačním cyklu.
- Provedení migrace po každém dokončeném evolučním cyklu. Tento proces probíhá tak, že pro každého přijatého jedince z daných subpopulací je generováno náhodné číslo, které je porovnáno s hodnotou parametru ϕ. Je-li náhodné číslo menší než tato hodnota, je patřičnému sousednímu uzlu odeslána zpráva s kopií jedince a s informací o tom, že má být provedena migrace. V opačném případě, tj. když náhodné číslo není menší než hodnota ϕ, je výpočetnímu uzlu rovněž odeslána zpráva, která však obsahuje informaci o tom, že migrace neproběhne. Výpočetní uzel tak ví, že může provést další evoluční cyklus.
- Přijetí výsledků po dokončení zadaného počtu generačních cyklů. Do všech subpopulací je odeslána ukončovací zpráva, po jejímž přijetí výpočetní aplikace ukončí zpracování algoritmu a odešle řídící aplikaci výsledky. Ty jsou stejně jako v případě `BasicDEDriver` předány nadřazeným třídám k dalšímu zpracování.

Výše uvedené třídy, tedy `BasicDEDriver` a `DistributedDEDriver`, jsou také zodpovědné za zpracování chyb. Pokud na některém z výpočetních uzlů dojde k chybě nebo pokud nebude možné odeslat některému z uzlů zprávu, třída tento stav detekuje a odešle ostatním uzlům chybovou zprávu s informací, že mají ukončit provádění algoritmu. K této situaci dojde pouze v případě PDE, jelikož jednotlivé uzly spolupracují a není tak možné ve vykonávání algoritmu pokračovat. V případě, že každý výpočetní uzel zpracovává algoritmus samostatně, dojde pouze k informování uživatele a k vyřazení uzlu, na němž došlo k chybě nebo se kterým nelze dále komunikovat. Ostatní uzly dále pokračují v provádění algoritmu.

Jak již bylo výše uvedeno, obě řídící třídy jsou potomky abstraktní třídy pojmenované `AlgorithmDriver`, která obsahuje několik základních společných metod, které konkrétní řídící třídy využívají. V případě, že by měl být v budoucnu projekt rozšířen o další variantu distribuované DE, měla by být vytvořena třída, která by byla dalším potomkem abstraktní řídící třídy. V té by již nebylo nutné implementovat vše od základu, ale stačilo by přidat pouze logiku související s daným způsobem distribuované DE.

6.2 Výpočetní aplikace (Slave)

Oproti aplikaci řídící neobsahuje výpočetní aplikace grafické uživatelské rozhraní, jelikož není primárně určena pro práci s uživatelem. Jedná se tak o konzolovou aplikaci. Skrze konzoli je možné provádět základní ovládání pouze pomocí několika základních příkazů, které aplikace umí zpracovat.[8] Veškeré další chování aplikace je ovlivněno řídící aplikací. Jediným hlavním úkolem aplikace je přijímat zprávy od řídícího uzlu, provádět algoritmus diferenciální evoluce a výsledky odesílat zpět řídící aplikaci.

Nastavení aplikace se provádí přes konfigurační soubor. V tom lze nastavit číslo portu, na němž bude aplikace naslouchat a dále skupinovou adresu a skupinový port pro periodické odesílání datagramů, které bylo zmíněno v předchozí části této kapitoly.

Zdrojový kód aplikace je opět rozdělen do několika následujících balíčků:

- **`algorithm`** Balíček obsahuje algoritmus diferenciální evoluce společně s třídami, které jsou s vykonáváním algoritmu těsně spjaty.
- **`core`** Zde se nachází třída pro čtení nastavených hodnot v konfiguračním souboru. Dále jsou obsahem následující dva menší balíčky:
  - **`core.command`** Obsahem balíčku jsou třídy pro zpracovávání základních příkazů. Ty mohou být zadány uživatelem přes konzoli, nebo mohou být přijaty z řídící aplikace.
  - **`core.controller`** Zde jsou třídy, které zpracovávají přijaté zprávy týkající se provádění algoritmu. Dále tyto třídy zahajují provádění algoritmu a v případě potřeby také připravují zprávy s migrujícími jedinci. Jinak řečeno, třídy tvoří protějšek k řídícím třídám v řídící aplikaci a ovládají tak průběh celého zpracování algoritmu.
- **`logging`** Podobně jako v řídící aplikaci, i zde je tento balíček zodpovědný za záznam událostí do *log* souboru.
- **`network`** V tomto balíčku je umístěna třída pro periodické odesílání datagramů s informací o tom, že je aplikace připravena provádět algoritmus, společně s číslem portu, na kterém naslouchá. Tato stejná třída také odešle datagram v momentě, kdy je aplikace regulérně ukončena. Obsahem datagramu je pouze informace o ukončení aplikace, díky čemuž dostane řídící uzel oznámení, že s daným výpočetním uzlem již nemůže počítat. Dále balíček obsahuje třídu, která má za úkol odpovídat řídící aplikaci na zprávu, která je odeslána ve chvíli, kdy se řídící aplikace snaží najít výpočetní uzel přímým zadáním IP adresy a čísla portu. Poslední třída, která je zde umístěna, slouží k odesílání zpráv informujících o tom, že došlo k chybě při běhu algoritmu.
- **`main`** Tento balíček obsahuje pouze jednu třídu, která obsahuje i statickou metodu `main(String[] args)`, která tvoří vstupní bod aplikace. Je zde provedena inicializace všech potřebných objektů, které zajišťují výše uvedené funkce a činnosti.

[8] Seznam dostupných příkazů společně s jejich vysvětlením je uveden v příloze této práce.

Nejdůležitější třídou je abstraktní třída s názvem `DifferentialEvolution` nacházející se v balíčku `algorithm`. Tato třída zajišťuje provádění samotného algoritmu DE. V projektu bylo implementováno 10 různých variant algoritmu, které jsou uvedeny v tabulce 2. Vzhledem k tomu, že jednotlivé varianty se od sebe liší pouze ve způsobu, jakými jsou prováděny operace mutace a křížení, abstraktní třída obsahuje prakticky všechen potřebný kód pro zpracování algoritmu (vytvoření populace, ohodnocení jedince, atd.). Pouze metody pro mutaci a křížení jsou abstraktní a jejich konkrétní implementace se nachází v deseti třídách, které dědí z třídy abstraktní. Těchto deset tříd představuje jednotlivé varianty DE. Jelikož všechny tyto varianty používají buď binomické, nebo exponenciální křížení, jsou v abstraktní třídě dvě metody provádějící oba typy křížení. V rámci jednotlivých potomků abstraktní třídy tak není nutné stále psát stejný kód pro křížení, ale stačí použít jednu ze zmíněných metod.

Výhodou tohoto návrhu je snadné přidání nové varianty algoritmu. Za předpokladu, že by se tato varianta lišila pouze ve způsobu křížení či mutace, stačí přidat nového potomka abstraktní třídy a implementovat pouze dvě jednoduché metody.

Pro potřeby generování pseudonáhodných čísel v tomto projektu nebyla použita standardní třída `java.util.Random`. Místo ní byl využit generátor Mersenne Twister, který je součástí knihovny, jež byla v této aplikaci použita. Jedná se o knihovnu Apache Commons Math a lze ji bezplatně stáhnout z adresy `http://commons.apache.org/proper/commons-math/`.

6.3 Sdílená knihovna

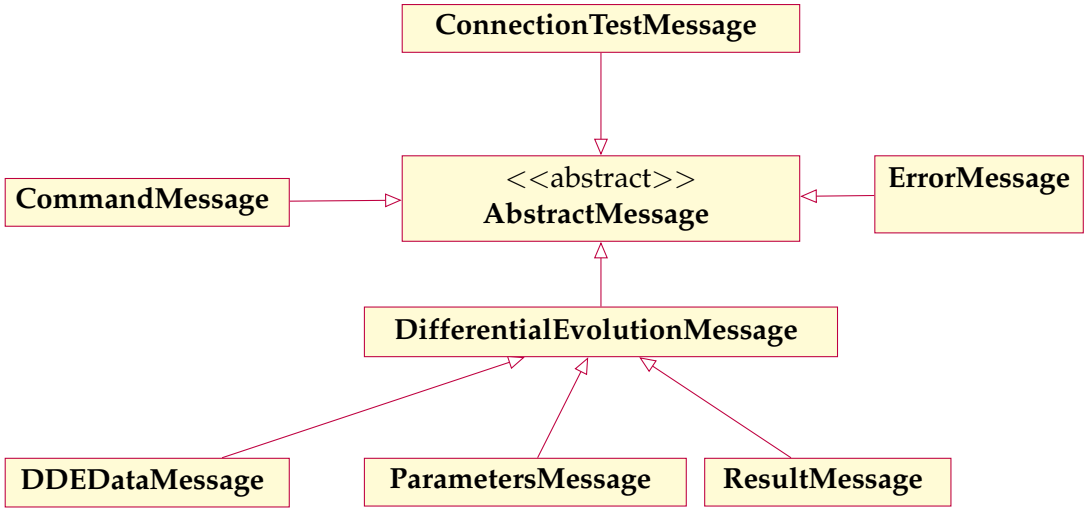
Jedná se o projekt, který sám o sobě není spustitelnou aplikací. Výstupem tohoto projektu s názvem DDE_Common je `jar` soubor, který ke své činnosti využívají obě výše představené aplikace. Základním úkolem knihovny je zpřístupnit oběma aplikacím třídy, které slouží pro jejich vzájemnou komunikaci. Ta probíhá vždy přes protokol TCP a zahajuje ji vždy řídící uzel.

6.3.1 Komunikace mezi aplikacemi

Komunikace probíhá prostřednictvím výměny zpráv, jež jsou reprezentovány pomocí objektů. Tyto objekty jsou poté odesílatelem převedeny na proud bajtů, který je následně odeslán příjemci. Ten si poté zrekonstruuje zpět původní objekt. K zajištění tohoto procesu je nutné, aby všechny třídy, reprezentující zprávy, byly serializovatelné. V programovacím jazyce Java toho lze dosáhnout velmi snadno - stačí, aby patřičná třída implementovala rozhraní `Serializable`.

Základem všech zpráv je třída `AbstractMessage`. Samotný název napovídá, že se jedná o abstraktní třídu, z níž poté jednotlivé třídy, představující konkrétní zprávy, dědí. V průběhu celé komunikace dochází k výměně celkem čtyř základních typů zpráv:

- Zprávy související s průběhem diferenciální evoluce. Tyto zprávy jsou reprezentovány třídou `DifferentialEvolutionMessage`. Může se tak jednat například o zprávy obsahující parametry algoritmu, nebo například zprávy s výsledkem.



Obrázek 9: Diagram znázorňující třídy použité pro komunikaci

- Chybové zprávy. Ty jsou posílány v případě, kdy dojde k nějaké chybě během provádění algoritmu. Reprezentuje je třída `ErrorMessage`.
- Řídící zprávy, pomocí nichž může řídící uzel odesílat základní příkazy jednotlivým výpočetním uzlům. Jedná se o objekty třídy `CommandMessage`.
- Zprávy, které jsou posílány ve chvíli, kdy se řídící aplikace snaží nalézt výpočetní uzel přímým zadáním jeho IP adresy a čísla naslouchajícího portu. Odpovídající třídou pro tyto zprávy je `ConnectionTestMessage`.

Výhodou tohoto návrhu je v případě nutnosti snadné přidání nového typu zprávy. Stačí pouze vytvořit novou třídu rozšiřující `AbstractMessage`. Třídy starající se o samotné odesílání zpráv již není třeba nijak modifikovat.

Pro úplnost a detailní přehled jednotlivých zpráv uvádíme obrázek 9, který zachycuje všechny aktuálně používané komunikační zprávy.

Všechny třídy zajišťující komunikaci, společně s třídami reprezentující jednotlivé komunikační zprávy, jsou umístěny v balíčku `communication`. O samotné odesílání a přijímaní zpráv se starají třídy `SenderThread` a `ReceiverThread`, které vykonávají svou činnost ve vedlejších vláknech, aby nebyly ovlivněny další funkce.

6.3.2 Účelové funkce

Druhou skupinou tříd, které sdílená aplikace obsahuje, jsou třídy představující účelové funkce. Podobně jako u zpráv, i zde je vytvořena abstraktní třída představující obecnou účelovou funkci. Jde o třídu `OptimizationFunction`, která obsahuje jednu abstraktní metodu zvanou `evaluate(Object[] parameters)`. Konkrétní účelové funkce pak rozšiřují výše uvedenou obecnou třídu a poskytují vlastní implementaci abstraktní metody. Díky tomuto návrhu je opět velmi snadné přidat novou účelovou funkci. Není

potřeba zasahovat do zdrojových kódů výpočetní ani řídící aplikace, stačí pouze do sdílené knihovny přidat nového potomka třídy `OptimizationFunction`. Třídám je také možno nastavit jejich jméno skrze instanční proměnnou `name`, které se poté zobrazí v řídící aplikaci u výběru účelové funkce. Vzhledem k tomu, že jsou účelové funkce umístěny ve sdílené knihovně, řídící aplikace vždy ví, jaké funkce jsou k dispozici, čímž odpadá výpočetním aplikacím nutnost informovat řídící uzel o dostupných účelových funkcích.

Balíček s názvem `optimizationfunction` sdružuje všechny implementované účelové funkce. Součástí tohoto balíčku jsou i dvě jednoduché neuronové sítě, které byly použity během experimentů. Jsou také potomky abstraktní třídy představující obecnou účelovou funkci a implementují metodu pro výpočet fitness hodnoty jedince. Tato hodnota zde představuje globální chybu dané neuronové sítě.

7 Testovací funkce

V následující kapitole se seznámíme s několika testovacími funkcemi, které byly v rámci této práce použity k ověření, zda vytvořený algoritmus diferenciální evoluce funguje správně. Obecně lze říci, že potřebujeme-li otestovat výkonnost námi vytvořeného optimalizačního algoritmu, můžeme použít existující problémy, které již byly vyřešeny pomocí jiných algoritmů. Výsledky testovaného algoritmu pak stačí jen porovnat s již existujícími výsledky. Druhým přístupem je použít již zmíněné testovací funkce, u nichž je obvykle známa pozice globálního extrému. Tyto funkce navíc disponují mnoha vlastnostmi, které celý proces hledání extrému značně ztěžují. Funkce mohou být nelineární, mohou obsahovat velké množství lokálních extrémů či mohou trpět různými patologiemi, například rovinou okolo pozice globálního extrému. [19]. V této práci byly experimenty prováděny celkem na šesti testovacích funkcích. Jejich výčet je následující:

- Ackleyho funkce
- Griewangkova funkce
- Rastriginova funkce
- Salomonova funkce
- Schwefelova funkce
- Sférická funkce (Sphere function)

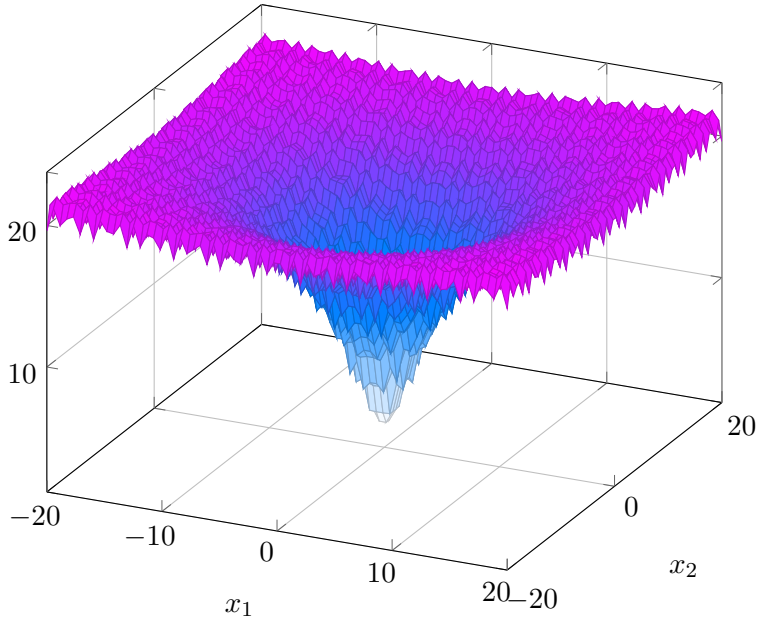
Všechny výše zmíněné funkce si dále v této kapitole představíme. Společně s grafem dané funkce pro dva argumenty uvedeme vždy i matematický zápis konkrétní funkce, interval, v němž leží argumenty funkce a samozřejmě také pozici globálního minima.

7.1 Ackleyho funkce

Pro argumenty této testovací funkce obvykle platí $x_i \in \langle -32,768; 32,768 \rangle$ pro všechna $i = 1, ..., D$. Jedná se o multimodální funkci a její matematický zápis je:

$$f(\mathbf{x}) = -a \exp\left(-b\sqrt{\frac{1}{D}\sum_{i=1}^{D} x_i^2}\right) - \exp\left(\frac{1}{D}\sum_{i=1}^{D}\cos(cx_i)\right) + a + \exp(1)$$

Doporučené hodnoty proměnných a, b a c jsou $a = 20$, $b = 0,2$ a $c = 2\pi$. Pro globální minimum funkce platí $f(x^*) = 0$, kde $x^* = (0, ..., 0)$.



Obrázek 10: Ackleyho funkce

7.2 Griewangkova funkce

Jedná se stejně jako v případě Ackleyho funkce o multimodální funkci, pro jejíž argumenty platí $x_i \in \langle -600; 600 \rangle$ pro všechna $i = 1, ..., D$. Matematický zápis této funkce je následující:

$$f(\mathbf{x}) = \sum_{i=1}^{D} \frac{x_i^2}{4000} - \prod_{i=1}^{D} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$$

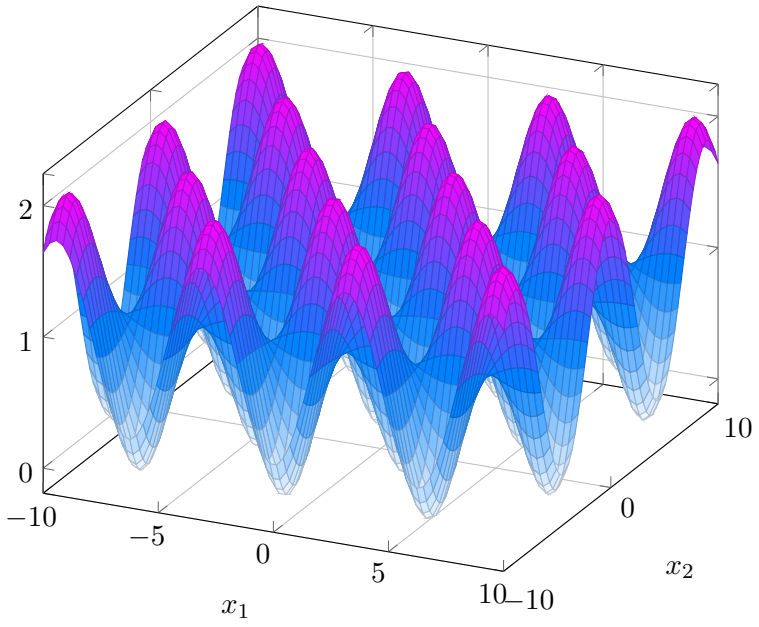
Pro globální minimum funkce platí $f(x^*) = 0$, kde $x^* = (0, ..., 0)$.

7.3 Rastriginova funkce

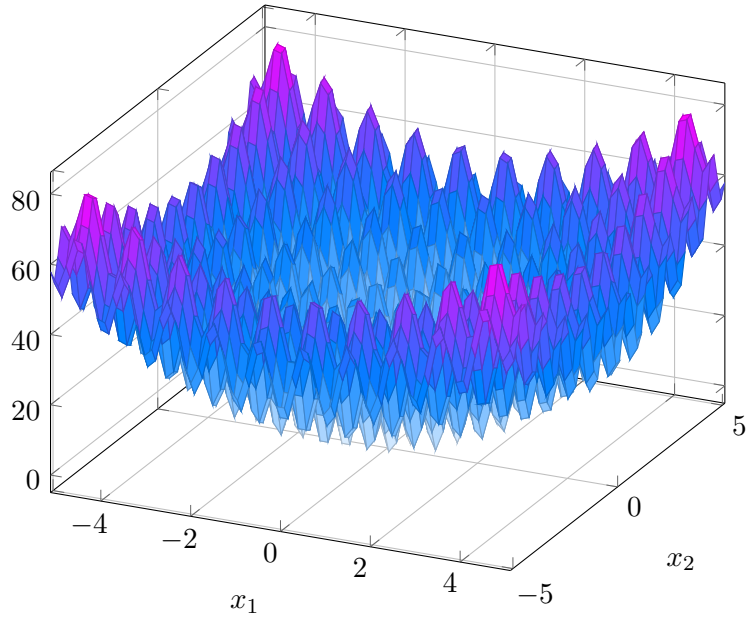
Rastriginova funkce obsahuje velké množství lokálních extrémů, je tak vysoce multimodální. Všechny argumenty funkce leží v intervalu $\langle -5.12; 5.12 \rangle$ a je definována následujícím předpisem:

$$f(\mathbf{x}) = 10D + \sum_{i=1}^{D} (x_i^2 - 10\cos(2\pi x_i))$$

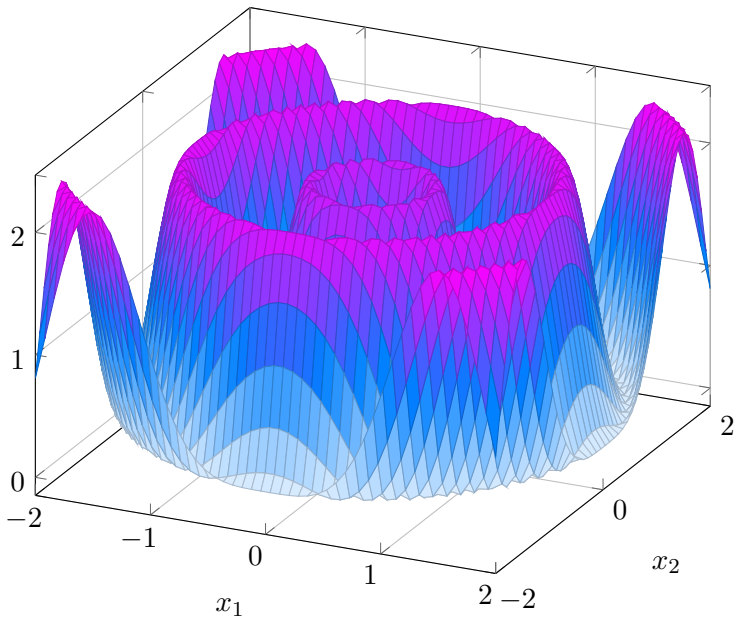
Hodnota globálního minima je stejná jako u předchozích dvou funkcí, tedy $f(x^*) = 0$, kde $x^* = (0, ..., 0)$.



Obrázek 11: Griewangkova funkce



Obrázek 12: Rastriginova funkce



Obrázek 13: Salomonova funkce

7.4 Salomonova funkce

Tato funkce je symetrická a je dalším ze zástupců multimodálních funkcí. Argumenty se nachází v intervalu $\langle -100; 100 \rangle$. Matematické vyjádření je následující:

$$f(\mathbf{x}) = \text{-cos}(2\pi \cdot \|x\|) + 0,1 \cdot \|x\| + 1$$

$$\|x\| = \sqrt{\sum_{i}^{D} x_i^2}$$

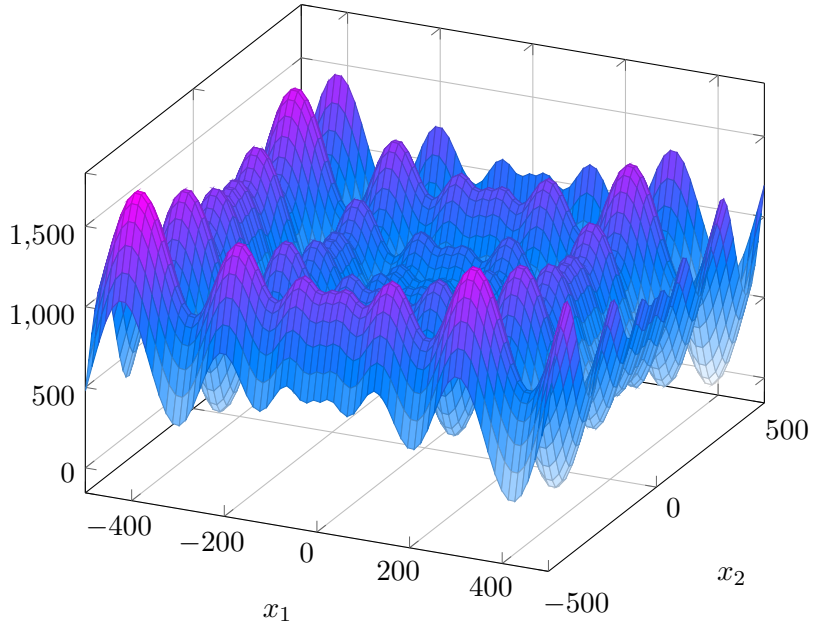
Hodnota globálního minima je rovna nule a nachází se v bodě (0, ..., 0). Opět tedy můžeme psát $f(x^*) = 0$, kde $x^* = (0, ..., 0)$.

7.5 Schwefelova funkce

Předposlední funkcí, kterou si představíme, je Schwefelova funkce. Obsahuje mnoho lokálních extrémů a její matematický zápis vypadá následovně:

$$f(\mathbf{x}) = 418,9829 \cdot D - \sum_{i=1}^{D} x_i \text{sin}(\sqrt{|x_i|})$$

Argumenty funkce se pohybují v intervalu $\langle -500; 500 \rangle$. Globální minimum se nachází v bodě $(420,9687; ...; 420,9687)$ a jeho hodnota je 0.



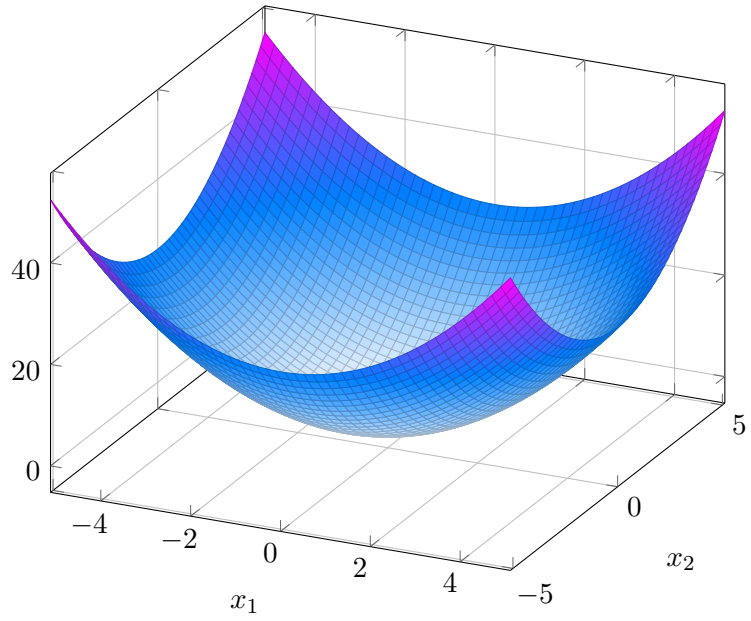
Obrázek 14: Schwefelova funkce

7.6 Sférická funkce (Sphere function)

Jedná se o poměrně jednoduchou unimodální funkci. Robustnější optimalizační algoritmy by tak s touto funkcí neměly mít problém a měly by snadno najít globální extrém. Sférickou funkci lze matematicky zapsat velmi snadno:

$$f(\mathbf{x}) = \sum_{i=1}^{D} x_i^2$$

Pro argumenty sférické funkce platí $x_i \in \langle -5.12, 5.12 \rangle$ pro všechna $i = 1, ..., D$. Globální minimum lze vyjádřit: $f(x^*) = 0$, kde $x^* = (0, ..., 0)$



Obrázek 15: Sférická funkce

8 Experimenty

Po dokončení implementační části práce byly provedeny experimenty, které můžeme rozdělit na dvě části. Do první části patří experimenty, u nichž bylo použito šest testovacích funkcí představených v kapitole 7. Druhá část experimentů byla zaměřena na praktické použití diferenciální evoluce, konkrétně na nalezení optimální kombinace vah neuronové sítě. V této kapitole budou popsány obě části provedených experimentů a dále zde budou prezentovány dosažené výsledky.

8.1 Experimenty na testovacích funkcích

Experimenty byly prováděny na celkem šesti testovacích funkcích, které jsou uvedeny v kapitole 7. U každé funkce byly experimenty prováděny pro pět různých dimenzí, jmenovitě 2, 5, 10, 20 a 30. Cílem těchto experimentů bylo v první řadě ověřit, zda vytvořený algoritmus diferenciální evoluce pracuje správně, a dále byly experimenty provedeny za účelem srovnání výkonnosti klasické diferenciální evoluce s paralelní DE, jejíž vysvětlení je podáno v kapitole 4.2. Obě varianty algoritmu byly pro každou testovací funkci a každou její dimenzi spuštěny stokrát a ze získaných výsledků poté byly vypočítány základní statistické hodnoty, které si dále v textu pro každou použitou testovací funkci uvedeme. Konkrétně se jedná o následující hodnoty:

- minimální a maximální fitness hodnota nejlepšího nalezeného jedince (best_{min} a best_{max})
- minimální a maximální fitness hodnota nejhoršího nalezeného jedince (worst_{min} a worst_{max})
- průměrná fitness hodnota nejlepšího a nejhoršího jedince (μ_{best} a μ_{worst})
- směrodatná odchylka fitness hodnoty nejlepšího a nejhoršího jedince (σ_{best} a σ_{worst})
- minimální a maximální čas doby zpracování algoritmu v sekundách (time_{min} a time_{max})
- průměrná hodnota doby zpracování algoritmu v sekundách (μ_{time})

Parametry algoritmu byly nastaveny pro každou funkci a každou její dimenzi rozdílně, avšak vždy stejně pro DE a PDE. Nastavení parametrů zachycuje tabulka 3, v níž jsou pro každou funkci a dimenzi uvedeny parametry NP, G, CR a F. V případě PDE zde NP představuje velikost jedné subpopulace. Migrační parametr ϕ byl u PDE vždy nastaven na hodnotu 0,2. Společným rysem pro všechny experimenty je varianta DE - byla zvolena klasická varianta DE/rand/1/bin. U PDE bylo použito vždy 5 výpočetních uzlů, které spolupracovaly na výpočtu. Všechny výpočetní uzly byly z důvodu nedostatečného množství PC spuštěny na jednom počítači, což ale na dosažené výsledky nemá vliv. V případě, že by byl každý výpočetní uzel spuštěn na jiném PC, lišil by se pouze čas potřebný k provedení algoritmu, který by závisel především na výpočetním výkonu jednotlivých PC a na přenosové rychlosti mezi řídící aplikací a výpočetními uzly.

Dosažené výsledky jsou dále v textu prezentovány prostřednictvím tabulek 4 - 15. Každá tabulka obsahuje výše uvedené statistické hodnoty pro všechny testované dimenze dané funkce pro jednu variantu algoritmu, tedy DE nebo PDE.

Z tabulek s výsledky lze vyčíst, že ve většině případů se oběma typům algoritmu podařilo nalézt hodnotu globálního extrému, nebo hodnotu tomuto extrému velmi blízkou. Ukázalo se, že mezi obtížnější funkce na optimalizaci patří funkce Rastriginova, Salomonova a Schwefelova. U Schwefelovy funkce je vidět, že klasická varianta DE ne vždy našla globální extrém - u dimenzí 5, 20 a 30 došlo v některých případech k tomu, že se evoluce zastavila v lokálním extrému s hodnotou přibližně 118,5. Svědčí o tom i vyšší průměrná fitness hodnota nejlepších nalezených jedinců. U paralelní DE k uváznutí u Schwefelovy funkce v lokálním extrému nikdy nedošlo, populace se vždy dostala velmi blízko ke globálnímu minimu.

	2	**5**	**10**	**20**	**30**
Ackley	NP = 10 G = 100 CR = 0,8 F = 0,5	NP = 25 G = 100 CR = 0,7 F = 0,3	NP = 50 G = 100 CR = 0,7 F = 0,2	NP = 100 G = 200 CR = 0,8 F = 0,2	NP = 150 G = 200 CR = 0,5 F = 0,15
Griewangk	NP = 20 G = 100 CR = 0,9 F = 0,5	NP = 50 G = 100 CR = 0,4 F = 0,1	NP = 100 G = 150 CR = 0,4 F = 0,1	NP = 100 G = 200 CR = 0,6 F = 0,2	NP = 150 G = 300 CR = 0,6 F = 0,2
Rastrigin	NP = 20 G = 100 CR = 0,2 F = 0,2	NP = 50 G = 100 CR = 0,3 F = 0,1	NP = 100 G = 200 CR = 0,3 F = 0,1	NP = 150 G = 300 CR = 0,3 F = 0,1	NP = 150 G = 450 CR = 0,3 F = 0,1
Salomon	NP = 20 G = 100 CR = 0,75 F = 0,2	NP = 50 G = 100 CR = 0,75 F = 0,2	NP = 100 G = 200 CR = 0,75 F = 0,2	NP = 150 G = 300 CR = 0,75 F = 0,2	NP = 150 G = 350 CR = 0,75 F = 0,2
Schwefel	NP = 20 G = 100 CR = 0,85 F = 0,75	NP = 50 G = 100 CR = 0,5 F = 0,2	NP = 100 G = 150 CR = 0,4 F = 0,1	NP = 100 G = 300 CR = 0,4 F = 0,1	NP = 150 G = 450 CR = 0,5 F = 0,1
Sphere	NP = 10 G = 100 CR = 0,8 F = 0,3	NP = 25 G = 100 CR = 0,7 F = 0,2	NP = 50 G = 100 CR = 0,7 F = 0,2	NP = 100 G = 100 CR = 0,6 F = 0,2	NP = 150 G = 200 CR = 0,6 F = 0,15

Tabulka 3: Nastavení parametrů algoritmu během testování

V případě PDE zde NP představuje velikost jedné subpopulace. Parametr ϕ byl u PDE vždy nastaven na hodnotu 0,2.

Dimenze	2	5	10	20	30
best_{min}	0	0,00001005	0,00355854	0,00061866	0,00603074
best_{max}	2,57992756	4,66199529	1,64878326	0,79379635	0,01206165
μ_{best}	0,05270297	0,06356811	0,12478513	0,03509959	0,00914862
σ_{best}	0,36293917	0,49278219	0,36626456	0,15563035	0,00132972
worst_{min}	0	0,00003835	0,00799764	0,0011874	0,01180078
worst_{max}	2,57992756	4,66199604	1,65879103	0,79386187	0,02229622
μ_{worst}	0,05270305	0,06377359	0,13717879	0,03601723	0,01656323
σ_{worst}	0,36293916	0,4927558	0,36372253	0,15546858	0,00225081
time_{min}	0,003	0,011	0,033	0,165	0,377
time_{max}	0,055	0,049	0,177	0,777	1,326
μ_{time}	0,00696	0,01783	0,06252	0,28728	0,5503

Tabulka 4: Výsledky DE na Ackleyho funkci

Dimenze	2	5	10	20	30
best_{min}	0	0,00000264	0,00147407	0,00028857	0,00481511
best_{max}	0	0,00003108	0,00669048	0,00065939	0,00834165
μ_{best}	0	0,00001239	0,00344509	0,00047555	0,0062684
σ_{best}	0	0,00000583	0,0010042	0,00007975	0,00059191
worst_{min}	0	0,0000183	0,00501074	0,00051348	0,01010451
worst_{max}	0	0,00008472	0,01886466	0,00130539	0,01546127
μ_{worst}	0	0,00004644	0,00875782	0,00089517	0,01158592
σ_{worst}	0	0,00001689	0,00235825	0,00014408	0,00096464
time_{min}	0,284	0,274	0,302	0,81	1,16
time_{max}	2,034	1,001	1,022	2,148	2,71
μ_{time}	0,38801	0,3164	0,33716	0,90646	1,24648

Tabulka 5: Výsledky PDE na Ackleyho funkci

Obecně lze tvrdit, že obě varianty algoritmu pracovaly na testovacích funkcích správně, nicméně PDE v drtivé většině případů nebyla nikdy horší než klasická varianta.

8.2 Hledání optimální kombinace vah neuronové sítě

Druhá část provádění experimentů byla zaměřena na demonstraci praktického využití diferenciální evoluce. Cílem bylo najít optimální kombinaci vah jednoduché neuronové sítě pro řešení problému XOR. XOR, nebo také exkluzivní disjunkce, je logická operace, jejíž hodnota je pravda v případě, když každá vstupní hodnota má unikátní hodnotu v porovnání s ostatními vstupy. Pravdivostní tabulku funkce XOR zachycuje tabulka 16.

Pro řešení problému byla vytvořena neuronová síť s jednou skrytou vrstvou, která se skládala ze čtyř neuronů. Výstupní vrstva obsahovala jeden neuron a vrstva vstupní dva neurony. Každý neuron používal binární přenosovou funkci. Celá topologie sítě je

Dimenze	**2**	**5**	**10**	**20**	**30**
best_{min}	0	0,00494095	0,00000488	0,00001484	0,00000124
best_{max}	0,04683502	0,27752851	0,07973418	0,03032969	0,00128971
μ_{best}	0,00093509	0,04167801	0,02186513	0,00260928	0,00001907
σ_{best}	0,00499614	0,04149177	0,01548071	0,00664573	0,00013179
worst_{min}	0	0,08289114	0,00015349	0,00007936	0,00000331
worst_{max}	0,04683502	0,62359731	0,38835129	0,11862972	0,01007806
μ_{worst}	0,00143089	0,28037157	0,14959327	0,00859049	0,00012379
σ_{worst}	0,00532407	0,11640664	0,07907837	0,02322802	0,0010106
time_{min}	0,006	0,028	0,106	0,181	0,616
time_{max}	0,363	0,126	0,427	0,743	1,719
μ_{time}	0,02929	0,04186	0,18901	0,31005	0,83518

Tabulka 6: Výsledky DE na Griewangkově funkci

Dimenze	**2**	**5**	**10**	**20**	**30**
best_{min}	0	0,0006081	0,0000563	0,00001237	0,00000057
best_{max}	0	0,03556587	0,03459107	0,00110614	0,00000184
μ_{best}	0	0,01547066	0,00881975	0,00004847	0,00000101
σ_{best}	0	0,00737177	0,00686173	0,00010953	0,00000023
worst_{min}	0	0,04888057	0,00135489	0,00005227	0,00000168
worst_{max}	0	0,34145619	0,19312679	0,00531951	0,00000554
μ_{worst}	0	0,16932108	0,10342168	0,00019517	0,00000316
σ_{worst}	0	0,05745143	0,04786898	0,00052468	0,00000072
time_{min}	0,306	0,324	0,56	0,828	1,822
time_{max}	2,112	1,065	1,602	2,737	3,829
μ_{time}	0,44841	0,36635	0,61008	1,00739	1,92412

Tabulka 7: Výsledky PDE na Griewangkově funkci

znázorněna na obrázku 16, z něhož je patrné, že síť obsahuje celkem dvanáct vah, které jsou označeny w_{xy}, kde x představuje index neuronu a y index vstupu pro daný neuron.[9]

Hledání optimální kombinace vah poté probíhalo pomocí diferenciální evoluce. Každý jedinec obsahoval dvanáct parametrů, přičemž hranice parametrů byla nastavena v intervalu $\langle -20, 20 \rangle$. Fitness hodnota jedince reprezentovala globální chybu sítě. Trénovací množina pro neuronovou síť byla složena ze čtyř vstupních dvojic hodnot a čtyř odpovídajících výstupních hodnot (viz tabulka 16).

Po několika provedených experimentech byl nalezen jedinec, jehož fitness hodnota se rovnala nule. Tím se podařilo minimalizovat globální chybu neuronové sítě. Váhy v síti tedy byly nastaveny podle parametrů tohoto nejlepšího jedince a byla provedena druhá část experimentu, jehož úkolem bylo ověřit, zda síť dokáže správně klasifikovat i

[9] Dva neurony ve vstupní vrstvě nejsou do tohoto indexování zahrnuty, jelikož vstupní vrstva pouze distribuuje vstup do skryté vrstvy.

Dimenze	**2**	**5**	**10**	**20**	**30**
best_{min}	0	0	0	0,00000582	0,00000049
best_{max}	1,17736473	2,00152311	0,99738322	1,0024264	1,01119491
μ_{best}	0,19716061	0,30982342	0,07964275	0,07999662	0,09980012
σ_{best}	0,3922205	0,56095639	0,27136228	0,27011806	0,29905594
worst_{min}	0	0,00000008	0,00000003	0,00002452	0,00000163
worst_{max}	1,17736495	2,17573839	0,9981172	1,03367205	1,04355319
μ_{worst}	0,19716539	0,31264757	0,0796611	0,08171991	0,10049401
σ_{worst}	0,3922298	0,56648109	0,2713919	0,27095305	0,30025839
time_{min}	0,006	0,025	0,116	0,43	0,835
time_{max}	0,041	0,182	0,521	1,497	2,649
μ_{time}	0,00977	0,04604	0,21576	0,61585	1,15032

Tabulka 8: Výsledky DE na Rastriginově funkci

Dimenze	**2**	**5**	**10**	**20**	**30**
best_{min}	0	0	0	0,00000097	0,00000017
best_{max}	0	0,00000065	0,00000012	0,00004598	0,00003284
μ_{best}	0	0,00000004	0,00000001	0,0000104	0,0000045
σ_{best}	0	0,00000009	0,00000002	0,00000873	0,00000529
worst_{min}	0	0,00000001	0,00000001	0,00000443	0,00000065
worst_{max}	0	0,00001682	0,00000122	0,00024804	0,00010826
μ_{worst}	0	0,00000138	0,00000018	0,00005677	0,0000158
σ_{worst}	0	0,00000256	0,0000002	0,00004719	0,00001808
time_{min}	0,278	0,299	0,737	1,463	2,467
time_{max}	0,852	1,035	1,954	3,451	7,424
μ_{time}	0,30892	0,34979	0,79836	1,65699	2,77838

Tabulka 9: Výsledky PDE na Rastriginově funkci

neoriginální vstupní vektory, např. (0,8; 0,2).

Tato část experimentu probíhala tak, že byly postupně vyzkoušeny všechny vektory s hodnotami v rozsahu $\langle 0, 1 \rangle$ s velikostí kroku 0,05 - (0; 0), (0,05; 0), ..., (1; 1). Jednotlivé výstupy sítě byly porovnány s odpovídajícími požadovanými výstupy, čímž bylo možné stanovit procentuální hodnotu, která určuje úspěšnost, s jakou síť klasifikuje správně. Z celkem 441 vstupních vektorů síť klasifikovala 347 vektorů správně, což dává úspěšnost klasifikace 78,68%. Tuto hodnotu jistě nelze považovat za špatnou, nicméně zároveň nám tato hodnota ponechává dostatek prostoru pro další vylepšení.

Za účelem dosažení vyšší procentuální úspěšnosti klasifikace byl proveden další experiment, během něhož byla mírně pozměněna neuronová síť. Topologie sítě zůstala stejná, avšak přenosová funkce v neuronech skryté vrstvy byla změněna z binární na logistickou. Binární funkce tak zůstala pouze v neuronu výstupní vrstvy. Po této úpravě byl celý proces hledání optimální kombinace vah zopakován. Algoritmu se poté opět podařilo nalézt jedince s fitness hodnotou rovnou nule a parametry tohoto jedince byly znovu použity

Dimenze	**2**	**5**	**10**	**20**	**30**
best_{min}	0	0,09987335	0,09987335	0,10007712	0,19988042
best_{max}	1,39987335	0,19987421	0,19987335	0,19988356	0,30003593
μ_{best}	0,11737623	0,10094367	0,101054	0,1936491	0,25913445
σ_{best}	0,20325322	0,00999858	0,01001905	0,02254536	0,04635188
worst_{min}	0	0,10450231	0,10761608	0,20623105	0,31675879
worst_{max}	1,39987335	0,40128066	0,23439648	0,35860055	0,50174656
μ_{worst}	0,11738709	0,17658301	0,18315359	0,27991868	0,40672209
σ_{worst}	0,20325055	0,06183099	0,04086113	0,03880582	0,03676076
time_{min}	0,002	0,008	0,054	0,233	0,355
time_{max}	0,015	0,018	0,14	0,418	0,606
μ_{time}	0,00378	0,01296	0,09143	0,32848	0,45446

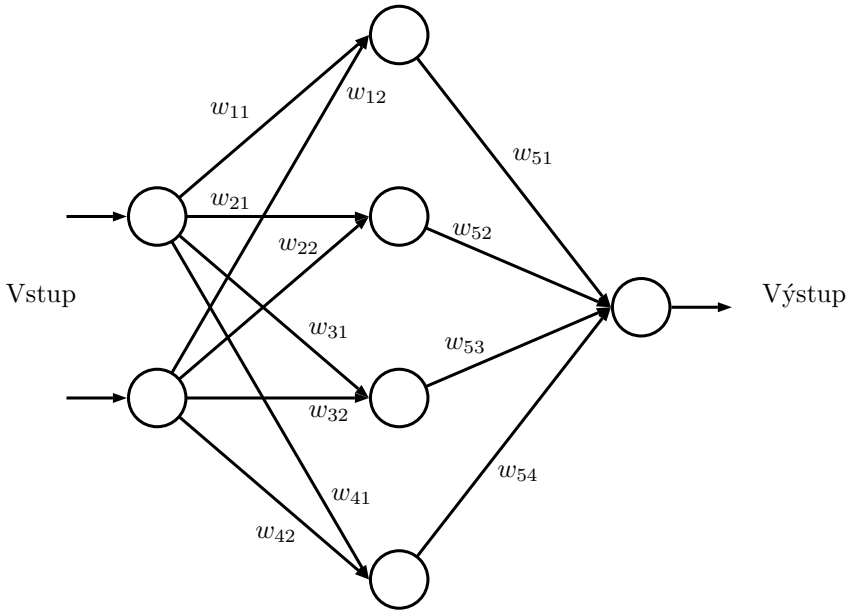
Tabulka 10: Výsledky DE na Salomonově funkci

Dimenze	**2**	**5**	**10**	**20**	**30**
best_{min}	0	0,09987335	0,09987335	0,09987335	0,10002345
best_{max}	0,09987335	0,09987372	0,09987336	0,19987335	0,19988435
μ_{best}	0,02609644	0,09987337	0,09987335	0,14293911	0,19887508
σ_{best}	0,04396895	0,00000006	0	0,04863226	0,00998501
worst_{min}	0	0,10039118	0,10023274	0,20014557	0,20563604
worst_{max}	0,09987335	0,12019293	0,11003788	0,21138671	0,31983668
μ_{worst}	0,02610163	0,10524778	0,10268712	0,20404871	0,2928173
σ_{worst}	0,04396607	0,00359626	0,00171663	0,00167683	0,02874689
time_{min}	0,24	0,256	0,596	1,159	1,448
time_{max}	0,357	0,499	0,773	1,497	4,674
μ_{time}	0,2542	0,2696	0,63101	1,21648	1,63323

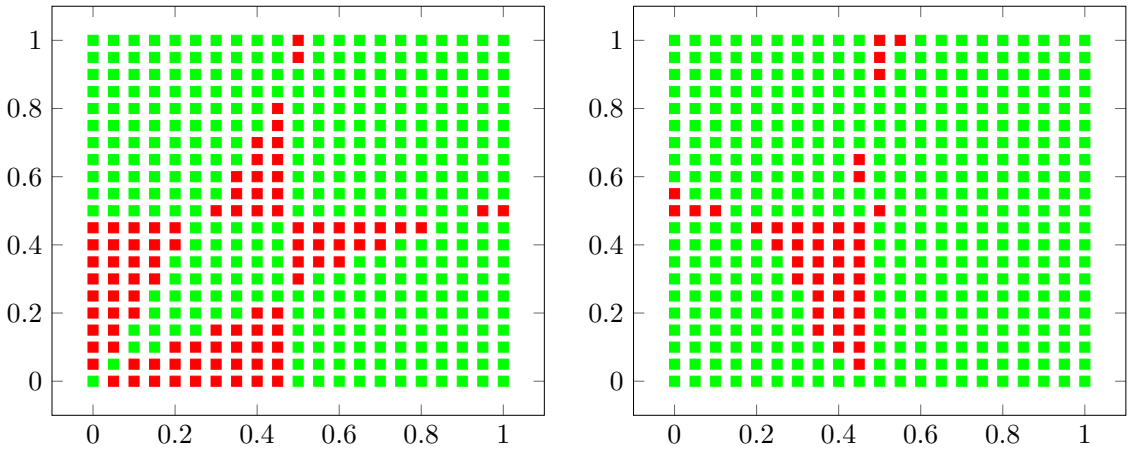
Tabulka 11: Výsledky PDE na Salomonově funkci

jako váhy upravené neuronové sítě. Byl zopakován experiment s neoriginálními vektory a v tomto případě již sít klasifikovala správně 399 vektorů. Jinými slovy, v porovnání s předchozím experimentem se podařilo snížit počet chybně klasifikovaných vektorů přibližně o jednu polovinu. Procentuální úspěšnost sítě během tohoto experimentu byla 90,48%, což již lze považovat za dostačující.

Grafické znázornění úspěšnosti klasifikace obou neuronových sítí je znázorněno na obrázku 17. V tabulce 17 jsou uvedeny nalezené hodnoty vah pro obě sítě.



Obrázek 16: Topologie neuronové sítě použité pro řešení problému XOR



Obrázek 17: Grafické znázornění úspěšnosti klasifikace neuronové sítě

Zelená barva představuje vektory, které síť klasifikovala správně, oproti tomu barva červená představuje ty vektory, které síť klasifikovala špatně. V levé části obrázku je grafické znázornění pro síť, v níž všechny neurony používaly binární přenosovou funkci. Pravá část obrázku graficky znázorňuje úspěšnost klasifikace sítě, jejíž neurony ve skryté vrstvě používaly logistickou funkci.

Dimenze	**2**	**5**	**10**	**20**	**30**
best_{min}	0,00002546	0,00006389	0,00012909	0,00025461	0,00038183
best_{max}	0,00002546	118,4384022	0,11141916	118,51053595	118,43871648
μ_{best}	0,00002546	2,37172841	0,00234379	4,75753926	3,56147093
σ_{best}	0	16,66450355	0,0148729	23,32676367	20,30454511
worst_{min}	0,00002546	0,00006795	0,00015145	0,00025474	0,00038183
worst_{max}	0,00002559	118,43889604	0,11595619	118,51056045	118,43871662
μ_{worst}	0,00002546	2,37581345	0,00274891	4,75755433	3,56147119
σ_{worst}	0,00000002	16,66398852	0,01608367	23,32676242	20,30454508
time_{min}	0,005	0,018	0,097	0,202	0,659
time_{max}	0,019	0,085	0,347	0,839	2,37
μ_{time}	0,00904	0,03594	0,16303	0,32271	0,90732

Tabulka 12: Výsledky DE na Schwefelově funkci

Dimenze	**2**	**5**	**10**	**20**	**30**
best_{min}	0,00002546	0,00006364	0,00012748	0,00025456	0,00038183
best_{max}	0,00002546	0,00006845	0,00014769	0,00025516	0,00038184
μ_{best}	0,00002546	0,000064	0,00013012	0,00025465	0,00038183
σ_{best}	0	0,00000065	0,00000273	0,00000008	0
worst_{min}	0,00002546	0,00006409	0,00013132	0,00025461	0,00038183
worst_{max}	0,00002546	0,00016469	0,00028448	0,00025655	0,00038186
μ_{worst}	0,00002546	0,00007423	0,00015841	0,00025495	0,00038183
σ_{worst}	0	0,00001418	0,00002494	0,00000032	0,00000001
time_{min}	0,288	0,305	0,517	1,08	2,168
time_{max}	0,955	0,953	1,458	2,441	6,21
μ_{time}	0,3328	0,33503	0,56174	1,14556	2,36431

Tabulka 13: Výsledky PDE na Schwefelově funkci

Dimenze	**2**	**5**	**10**	**20**	**30**
best_{min}	0	0	0,00000044	0,00087891	0,00000163
best_{max}	0,73456967	0,51664242	0,03437771	0,00367143	0,00000931
μ_{best}	0,01708687	0,01630136	0,00057249	0,00212623	0,00000322
σ_{best}	0,07946012	0,06876463	0,00355948	0,00068837	0,00000119
worst_{min}	0	0	0,00000174	0,00418238	0,00000478
worst_{max}	0,73456973	0,51665211	0,03451359	0,01466698	0,00001632
μ_{worst}	0,01708687	0,0163033	0,00059198	0,00773918	0,00000927
σ_{worst}	0,07946013	0,06876652	0,00357515	0,0021948	0,00000265
time_{min}	0,002	0,01	0,032	0,088	0,3
time_{max}	0,032	0,021	0,137	0,318	1,332
μ_{time}	0,00415	0,01395	0,04917	0,15199	0,45902

Tabulka 14: Výsledky DE na sférické funkci

Dimenze	**2**	**5**	**10**	**20**	**30**
best_{min}	0	0	0,00000008	0,00048206	0,0000007
best_{max}	0	0	0,00000076	0,00142828	0,00000212
μ_{best}	0	0	0,00000027	0,00090213	0,00000133
σ_{best}	0	0	0,00000014	0,00020488	0,00000024
worst_{min}	0	0	0,00000048	0,0026155	0,00000228
worst_{max}	0	0	0,00000374	0,00558673	0,00000637
μ_{worst}	0	0	0,00000156	0,0035983	0,0000041
σ_{worst}	0	0	0,00000071	0,0006114	0,00000069
time_{min}	0,331	0,328	0,339	0,413	1,059
time_{max}	0,965	0,969	1,011	1,425	2,524
μ_{time}	0,36788	0,35898	0,37959	0,52621	1,20184

Tabulka 15: Výsledky PDE na sférické funkci

x_1	x_2	Výstup
0	0	0
0	1	1
1	0	1
1	1	0

Tabulka 16: Pravdivostní tabulka funkce XOR

	Experiment 1	**Experiment 2**
w_{11}	3,1809	-1,8861
w_{12}	-5,7916	1,1426
w_{21}	-2,0998	-5,3073
w_{22}	5,7639	-4,1488
w_{31}	-8,8388	-2,5822
w_{32}	4,8468	4,4775
w_{41}	-10,7771	-7,6232
w_{42}	4,6131	-18,6276
w_{51}	7,1296	9,9899
w_{52}	-7,004	-16,8894
w_{53}	7,7793	-5,438
w_{54}	1,6682	-9,7928

Tabulka 17: Nalezené hodnoty vah pro obě použité neuronové sítě

Při experimentu 1 byla použita síť, jejíž všechny neurony používaly binární funkci. Při druhém experimentu byla použita síť, jejíž neurony ve skryté vrstvě používaly logistickou funkci. Všechny hodnoty vah v této tabulce byly zaokrouhleny na čtyři desetinná místa.

9 Závěr

Cílem této práce bylo vytvořit platformě nezávislý distribuovaný algoritmus diferenciální evoluce s jeho následným otestováním a ukázkou praktického využití. Pro tento účel byl zvolen problém naučení neuronové sítě neboli nalezení optimální kombinace vah. Dalším cílem bylo seznámit čtenáře prostřednictvím tohoto textu s obecným principem evolučních algoritmů a dále podat podrobné vysvětlení algoritmu diferenciální evoluce společně se zmínkou o možných vylepšeních, přičemž důraz byl kladen na způsoby, jimiž lze učinit algoritmus distribuovaným. Text navíc seznamuje čtenáře velmi stručně s problematikou neuronových sítí, z důvodu jejich použití během provádění experimentů.

Popis provedených experimentů společně s prezentací dosažených výsledků je uveden v kapitole 8. Z první fáze experimentů, které byly prováděny na šesti testovacích funkcích, lze vyvodit závěr, že ve většině případů bylo paralelní zpracování DE úspěšnější než klasická varianta. Při paralelním zpracování byly nalezeny pozice bližší globálním extrémům, nicméně rozdíl ve výsledcích nelze považovat za nikterak velký. Obě varianty algoritmu tak pracují správně.

Druhou část experimentů tvořila praktická ukázka využití algoritmu DE. Cílem zde bylo vytvořit neuronovou síť řešící problém XOR a následně najít optimální kombinaci vah. V první části tohoto experimentu se podařilo u neuronové sítě dosáhnout stavu, kdy síť klasifikovala správně v 78,68%. případů. Při snaze dosáhnout ještě lepších výsledků byla vytvořená síť mírně upravena a následně se již podařilo najít takovou kombinaci vah, při které síť klasifikovala s úspěšností 90,48%.

Pokud bychom se na vytvořené aplikace podívali z hlediska jejich možného dalšího rozšíření, zjistíme, že je zde spousta prostoru pro další úpravy a přidávání nové funkcionality. Z hlediska samotného algoritmu diferenciální evoluce se nabízí hned několik dalších úprav, které je možno provést. Jmenujme například techniku *dither* a *jither* uvedenou v části 3.7.1. Přidáním této úpravy by tak vznikl algoritmus paralelní diferenciální evoluce, v níž by jednotlivé subpopulace náhodně měnily svou hodnotu mutační konstanty.

Další možnou úpravou je přidání nové ukončovací podmínky algoritmu, jelikož v současné verzi je implementováno ukončení pouze po zadaném počtu generačních cyklů. Jako inspirace pro různé možnosti ukončení algoritmu slouží kapitola 3.3.

V neposlední řadě nelze opomenout možnost implementace dalšího způsobu distribuovaného provedení algoritmu, o nichž jsme se zmínili v kapitole 4. Aplikace by samozřejmě mohla být rozšířena o některý z těchto způsobů, případně i o další varianty, které nejsou v této práci uvedeny.

Pokud se oprostíme od samotného algoritmu diferenciální evoluce, je zde také prostor pro přidání zcela jiného algoritmu. Jmenovat můžeme například algoritmus SOMA (Self Organizing Migrating Algorithm), ale zajisté by se našla celá řada dalších.

Opustíme-li přehled možných vylepšení vytvořených aplikací, můžeme se na algoritmus diferenciální evoluce podívat z hlediska jeho uplatnění v každodenních problémech lidské činnosti. Není třeba zdůrazňovat, že potenciál využití algoritmu je skutečně velký. S jistou mírou nadsázky lze tvrdit, že diferenciální evoluci lze použít všude tam, kde klasické deterministické algoritmy selhávají nebo jsou z hlediska času velmi pomalé. Jedná

se o složité optimalizační úlohy, které se dnes vyskytují v celé řadě oborů a kterých stále přibývá a mnohé z nich jsou čím dál složitější z pohledu jejich řešení. Vzhledem ke zvyšující se složitosti ale rozhodně není ztrátou času zabývat se dalšími způsoby, jimiž lze algoritmus upravit s cílem dosáhnout jeho lepší výkonnosti. Mezi tyto způsoby rozhodně patří metody zajišťující paralelní zpracování algoritmu, díky kterému jsme schopni dosahovat lepších výsledků, nebo stejných výsledků v kratším čase v porovnání s klasickou variantou algoritmu. Navíc v dnešním světě velmi moderních technologií není nedostatek výpočetního výkonu, tudíž otázka paralelního zpracování nepředstavuje z hlediska dostupného hardwaru a technologií žádný problém.

Obecně lze tvrdit, že evoluční algoritmy jsou v dnešním světě velmi potřebné pro řešení širokého spektra problémů. Pokud toto tvrzení ještě více zobecníme, můžeme dojít k závěru, že biologicky inspirované metody a výpočty představují velmi zajímavou a potřebnou oblast informačních technologií, přičemž i s přihlédnutím k faktu, že v této oblasti došlo k velkému vývoji a mnoho technik a principů je již objevených, stále nám toto odvětví ponechává dostatek prostoru pro další výzkum a nacházení nových postupů, a proto je zcela jistě vhodné se touto problematikou dále zabývat.

10 Reference

[1] CANTÚ-PAZ, Erick. A summary of research on parallel genetic algorithms. 1995.

[2] DE FALCO, I., U. SCAFURI, E. TARANTINO, A. DELLA CIOPPA a A. DELLA CIOPPA. A Distributed Differential Evolution Approach for Mapping in a Grid Environment. 15th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing (PDP'07). IEEE, 2007, s. 442-449. DOI: 10.1109/PDP.2007.6. Dostupné z: `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4135309`

[3] DE FALCO, Ivanoe, et al. Distributed differential evolution for the registration of remotely sensed images. In: *Parallel, Distributed and Network-Based Processing, 2007. PDP'07. 15th EUROMICRO International Conference on*. IEEE, 2007. p. 358-362.

[4] Differential Evolution (DE): for Continuous Function Optimization (an algorithm by Kenneth Price and Rainer Storn). [online]. [cit. 2015-03-12]. Dostupné z: `http://www1.icsi.berkeley.edu/~storn/code.html`

[5] FLEGR, Jaroslav. *Evoluční biologie.* 2., opr. a rozš. vyd. Praha: Academia, 2009, 569 s. ISBN 978-80-200-1767-3.

[6] GU, Jirong; GU, Guojun. Differential Evolution with a local search operator. In: *Informatics in Control, Automation and Robotics (CAR), 2010 2nd International Asia Conference on*. IEEE, 2010. p. 480-483.

[7] CHALUPOVÁ-KARLOVSKÁ, Vlastimila. *Obecná biologie: středoškolská učebnice : evoluce, biologie buňky, genetika : s 558 řešenými testovými otázkami.* 1. vyd. Olomouc: Nakladatelství Olomouc, 2004, 206 s. ISBN 80-7182-174-8.

[8] KAKU, Michio. *Budoucnost mysli: fascinující průvodce světem technologií, které umožňují realizovat sny tvůrců sci-fi*. 1. vyd. Brno: BizBooks, 2015, 390 s. ISBN 978-80-265-0316-3.

[9] KOZA, John R. *Genetic programming: on the programming of computers by means of natural selection*. Cambridge, Mass.: MIT Press, c1992, xiv, 819 p. ISBN 0262111705.

[10] LAMPINEN, Jouni, ZELINKA, Ivan. Mixed integer-discrete-continuous optimization by differential evolution. In: *Proceedings of the 5th International Conference on Soft Computing*. 1999. p. 71-76.

[11] NEMETH, Evi, Garth SNYDER a Trent R HEIN. *Linux: kompletní příručka administrátora : 2. aktulizované vydání*. Vyd. 1. Brno: Computer Press, 2008, 984 s. Administrace (Computer Press). ISBN 978-80-251-2410-9.

[12] PRICE, Kenneth V, Rainer M STORN a Jouni A LAMPINEN. *Differential evolution: a practical approach to global optimization*. New York: Springer, 2005, xix, 538 p. ISBN 3540209506.

[13] RAHNAMAYAN, Shahryar; TIZHOOSH, Hamid R.; SALAMA, Magdy MA. Opposition-based differential evolution. *Evolutionary Computation, IEEE Transactions on*, 2008, 12.1: 64-79.

[14] SOUKUP, Václav. *Dějiny antropologie: (encyklopedický přehled dějin fyzické antropologie, paleoantropologie, sociální a kulturní antropologie).* Praha: Karolinum, 2004, 667 s., [32] s. barev. obr. příl. ISBN 80-246-0337-3.

[15] STORN, Rainer. On the usage of differential evolution for function optimization. In: *Fuzzy Information Processing Society, 1996. NAFIPS., 1996 Biennial Conference of the North American*. IEEE, 1996. p. 519-523.

[16] TASOULIS, D.K., N.G. PAVLIDIS, V.P. PLAGIANAKOS a M.N. VRAHATIS. Parallel differential evolution. In: *Proceedings of the 2004 Congress on Evolutionary Computation* (IEEE Cat. No.04TH8753). 2004. DOI: 10.1109/cec.2004.1331145.

[17] WEBER, Matthieu; TIRRONEN, Ville; NERI, Ferrante. Scale factor inheritance mechanism in distributed differential evolution. *Soft Computing*, 2010, 14.11: 1187-1207.

[18] ZAHARIE, Daniela. A comparative analysis of crossover variants in differential evolution. *Proceedings of IMCSIT 2007*, 2007, 171-181.

[19] ZELINKA, Ivan. *Biologicky inspirované výpočty: vybrané statě z evolučních algoritmů*. VŠB TU Ostrava.

[20] ZELINKA, Ivan. *Umělá inteligence I.: Skripta k předmětu Metody umělé inteligence*. Zlín, 2005.

[21] ZELINKA, Ivan. *Umělá inteligence: hrozba nebo naděje?*. 1. vyd. Praha: BEN - technická literatura, 2003, 142 s. ISBN 80-7300-068-7.

A Uživatelská příručka

A.1 Výpočetní aplikace

A.1.1 Spuštění aplikace

Pro spuštění aplikace je potřeba nejprve spustit příkazový řádek (příp. terminál) a pomocí příkazu `cd` změnit aktuální pracovní adresář na adresář, v němž je umístěn soubor `DDE_Slave.jar`. Poté je nutné zadat následující příkaz:

```
java -jar DDE_Slave.jar [name]
```

Nepovinný parametr `name` představuje jméno výpočetního uzlu a v případě jeho zadání se jedná o jméno, které bude zobrazeno v GUI řídící aplikace.

Za předpokladu, že je na daném počítači nainstalováno Java SE Runtime Environment ve verzi alespoň 1.7 (JRE), dojde po zadání příkazu ke spuštění aplikace a objeví se text `The slave is running`. V případě, že JRE není na daném počítači k dispozici, je možné tento software bezplatně stáhnout pro celou řadu platforem ze stránek `http://www.oracle.com/technetwork/java/javase/downloads/jre7-downloads-1880261.html`.

A.1.2 Seznam dostupných příkazů

Je-li výpočetní aplikace spuštěna, je možné ji ovládat pomocí několika základních příkazů. Tyto příkazy jsou společně s jejich vysvětlením uvedeny v tabulce 18. Dva z těchto příkazů, konkrétně `restart` a `quit`, vyžadují po jejich zadání dodatečné potvrzení uživatele, které se provádí zadáním a potvrzením znaku `Y` (Yes). K zamítnutí se používá znak `N` (No).

Příkaz	**Vysvětlení**
`port`	Vypíše port, na němž aplikace naslouchá.
`g-address`	Vypíše používanou skupinovou adresu pro odesílání datagramů skrze multicast.
`g-port`	Vypíše používaný skupinový port pro odesílání datagramů.
`r-period`	Zobrazí časový interval v jednotkách ms, který určuje, jak často aplikace odesílá datagram s informací o tom, že je připravena k provádění algoritmu.
`help`	Vypíše nápovědu.
`restart`	Probíhá-li algoritmus, dojde k jeho přerušení. Aplikace se tak dostane do stejného stavu jako při spuštění.
`quit`	Ukončí aplikaci.

Tabulka 18: Příkazy pro ovládání výpočetní aplikace

A.2 Řídící aplikace

Ke spuštění řídící aplikace je stejně jako v případě výpočetní aplikace nutné, aby na daném počítači bylo k dispozici JRE alespoň ve verzi 1.7. Dále je nutné opět spustit příkazový řádek či terminál a přejít do adresáře, v němž se nachází soubor `DDE_Master.jar`. Nakonec je potřeba zadat příkaz:

```
java -jar DDE_Master.jar
```

Zadáním výše uvedeného příkazu dojde ke spuštění aplikace.

A.2.1 Ovládání řídící aplikace

Po úspěšném spuštění se objeví nové okno s názvem Distributed Differential Evolution Algorithm. V rámci tohoto okna jsou k dispozici čtyři různé panely, přičemž ve výchozím stavu je zobrazen panel pro výběr výpočetních uzlů. Obrázky 18 až 21 zobrazují všechny čtyři panely. U každého obrázku jsou detailně popsány všechny ovládací prvky v GUI.

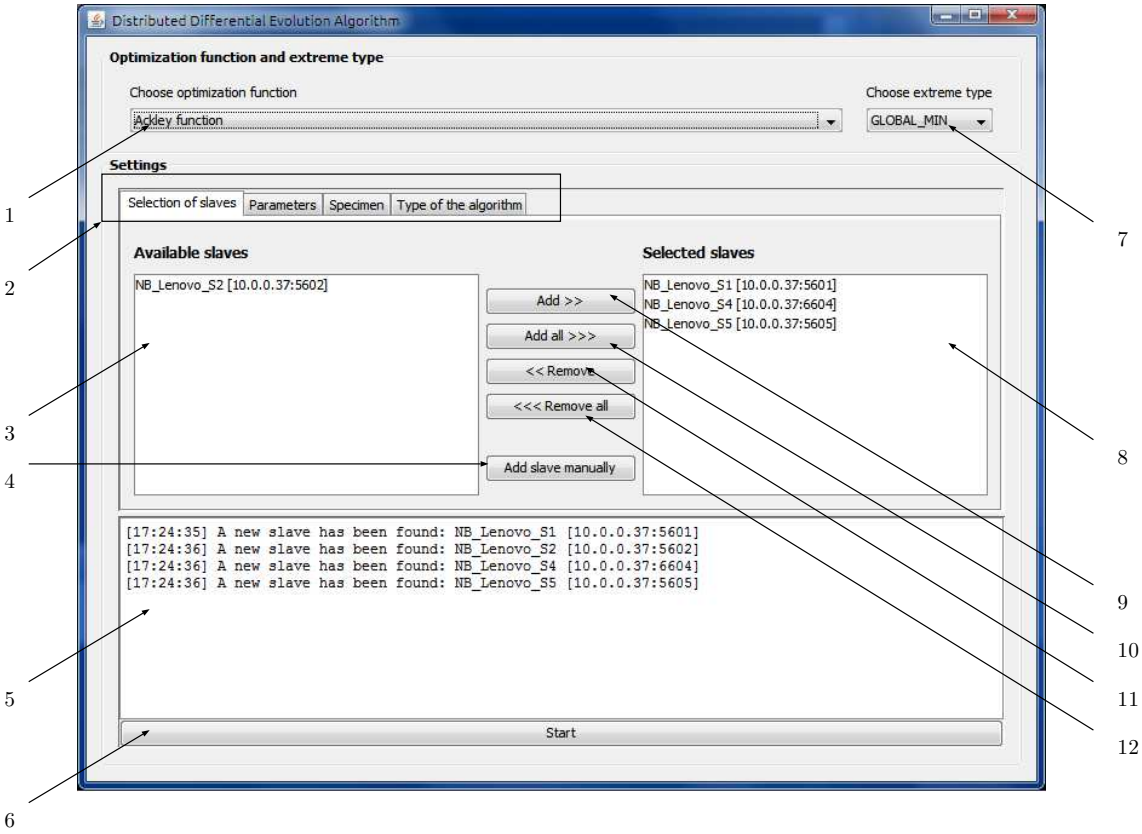
Po nastavení parametrů, vzorového jedince, typu DE a po vybrání výpočetních uzlů je možné zahájit algoritmus tlačítkem `Start` umístěném v dolní části GUI. Pokud je vše nastaveno správně, zobrazí se dialog s otázkou, kolikrát chceme algoritmus provést. Zde zadáme číslo v rozsahu 1 - 500 a potvrdíme spuštění algoritmu. Poté se zobrazí informační dialog informující uživatele o aktuálním průběhu vykonávání algoritmu. V případě, že během provádění algoritmu dojde k odpojení některého z výpočetních uzlů, mohou nastat dvě situace.

- provádění algoritmu se okamžitě zastaví - tato situace nastane v případě, že byla vybrána PDE
- provádění algoritmu bude pokračovat s menším počtem výpočetních uzlů - tato situace nastane v případě, že byla vybrána základní jednoduchá varianta distribuované DE (v případě, že se ale odpojí všechny výpočetní uzly, dojde k ukončení algoritmu)

Po dokončení vykonávání algoritmu se zobrazí okno s přehledem získaných výsledků. V případě, že byl algoritmus z nějakého důvodu přerušen a je-li k dispozici alespoň část výsledků, je uživatel dotázán, zda chce tyto výsledky zobrazit. Pokud zvolí zobrazení těchto nekompletních výsledků, dojde také k zobrazení okna s jejich přehledem.

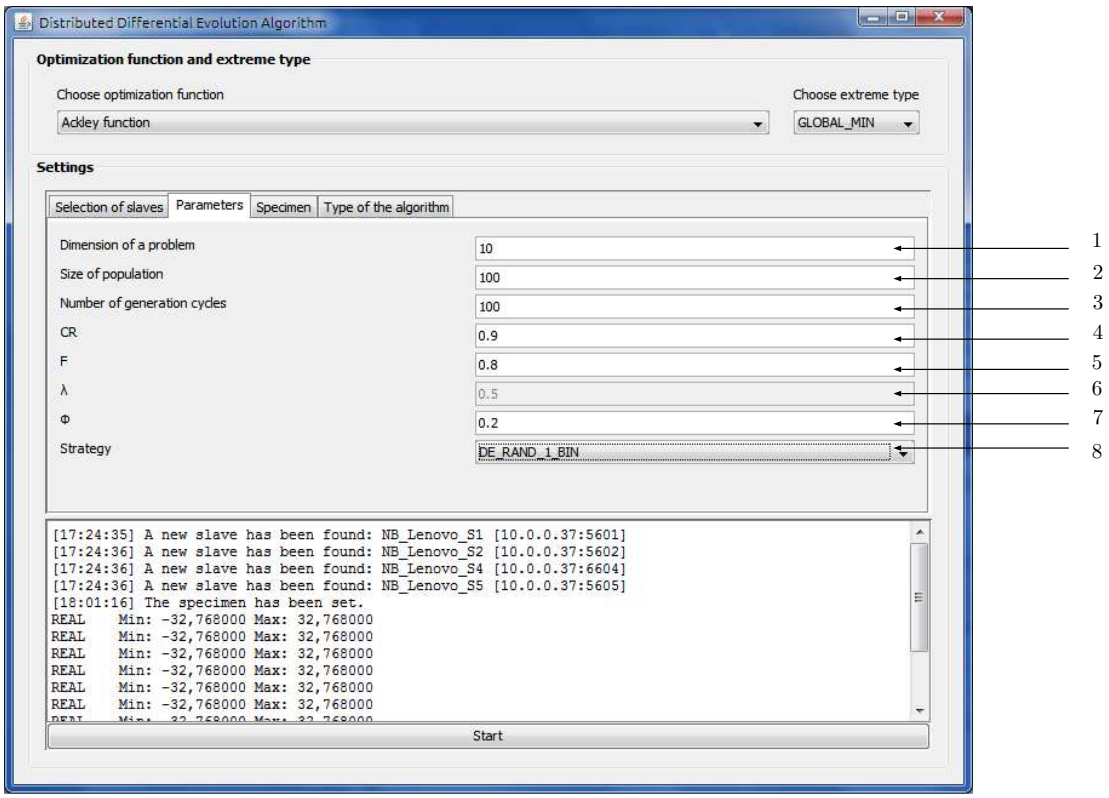
Vzhled okna s výsledky je závislý na typu vybrané varianty distribuované DE. Obrázek 22 znázorňuje okno s výsledky při použití základní varianty DE, obrázek 23 pak zachycuje výsledky paralelní diferenciální evoluce.

V případě jednoduché distribuované varianty DE je na obrázku 22 vidět, že algoritmus byl spuštěn celkem pětkrát, což lze vyčíst z levého panelu (2). Další panel (1) zachycuje seznam všech výpočetních uzlů, které se na zpracování algoritmu podílely. Vybereme-li některý z těchto uzlů, v panelu 3 se zobrazí detailnější informace o výsledku, který tento uzel odeslal řídící aplikaci. Z tohoto panelu lze vyčíst nejlepší a nejhorší nalezenou fitness hodnotu, dále parametry nejlepšího jedince a také průběh vývoje nejlepšího a nejhoršího



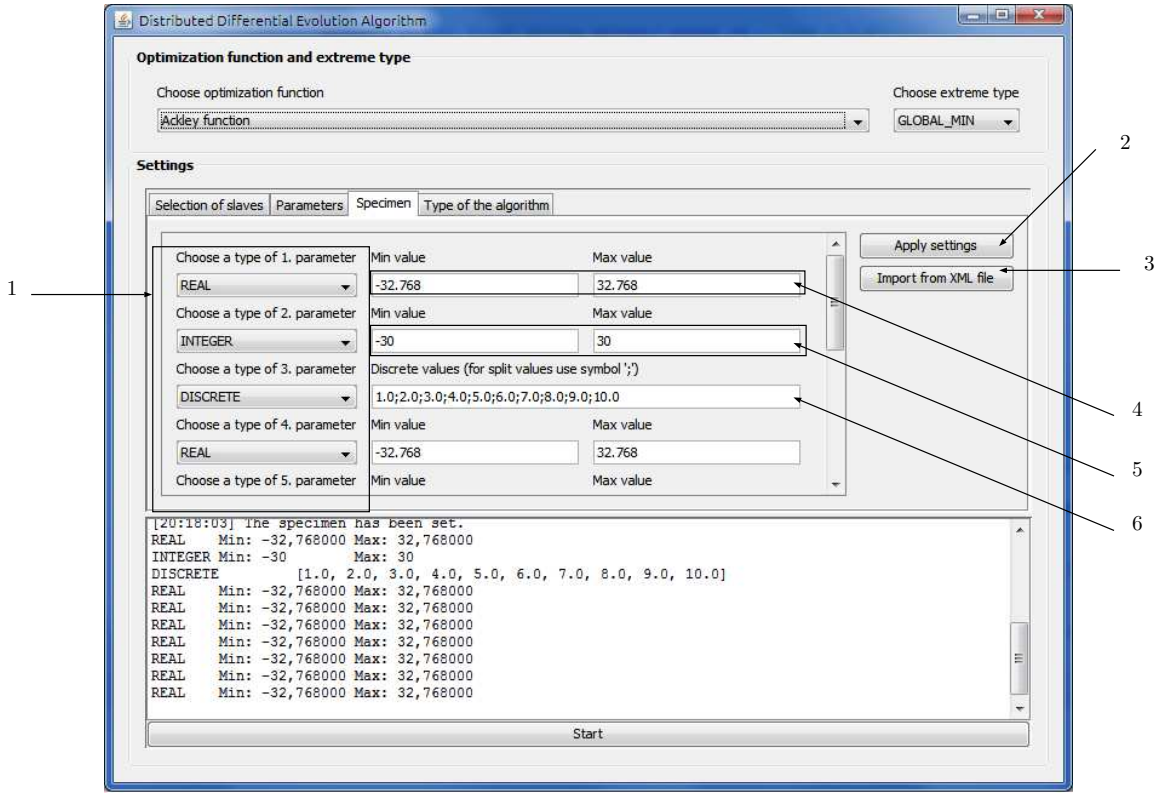
Obrázek 18: Řídící aplikace - panel pro výběr výpočetních uzlů

1 - výběr účelové funkce; 2 - přepínání čtyř základních panelů; 3 - seznam dostupných výpočetních uzlů; 4 - manuální přidání výpočetního uzlu (je třeba zadat IP adresu a číslo portu); 5 - informační výpisy pro uživatele; 6 - spuštění algoritmu; 7 - výběr, zda bude hledáno globální minimum či maximum; 8 - seznam vybraných výpočetních uzlů; 9 - tlačítko pro přesun aktuálně vybraného výpočetního uzlu do seznamu vybraných uzlů; 10 - tlačítko pro přesun všech dostupných výpočetních uzlů do seznamu vybraných uzlů; 11 - tlačítko pro přesun aktuálně vybraného výpočetního uzlu zpět do seznamu dostupných uzlů; 12 - tlačítko pro přesun všech výpočetních uzlů zpět do seznamu dostupných uzlů;



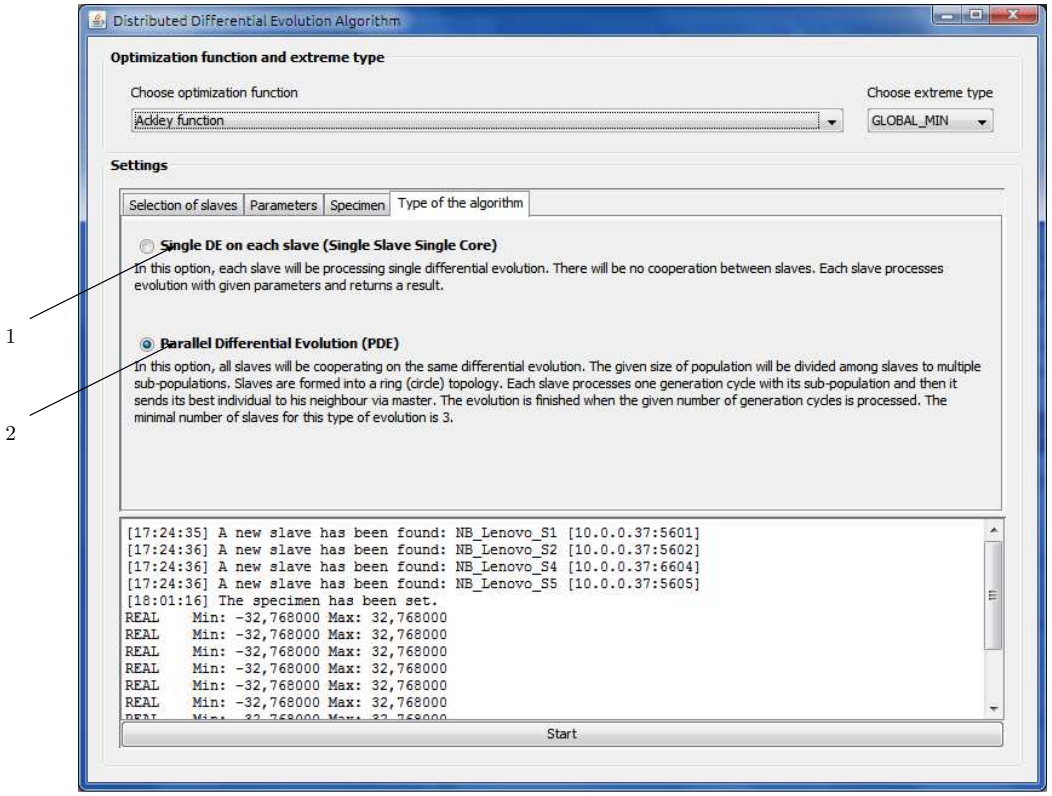
Obrázek 19: Řídící aplikace - panel pro nastavení parametrů

1 - dimenze problému (počet parametrů jedince); 2 - velikost populace (v případě PDE představuje celkovou populaci, velikost sub-populace se určí automaticky); 3 - počet generačních cyklů; 4 - práh křížení CR; 5 - mutační konstanta F; 6 - parametr λ (pole je dostupné jen pro varianty DE/rand-to-best/1/exp a DE/rand-to-best/1/bin); 7 - parametr ϕ pro nastavení migrace (dostupné jen pro PDE); 8 - varianta DE;



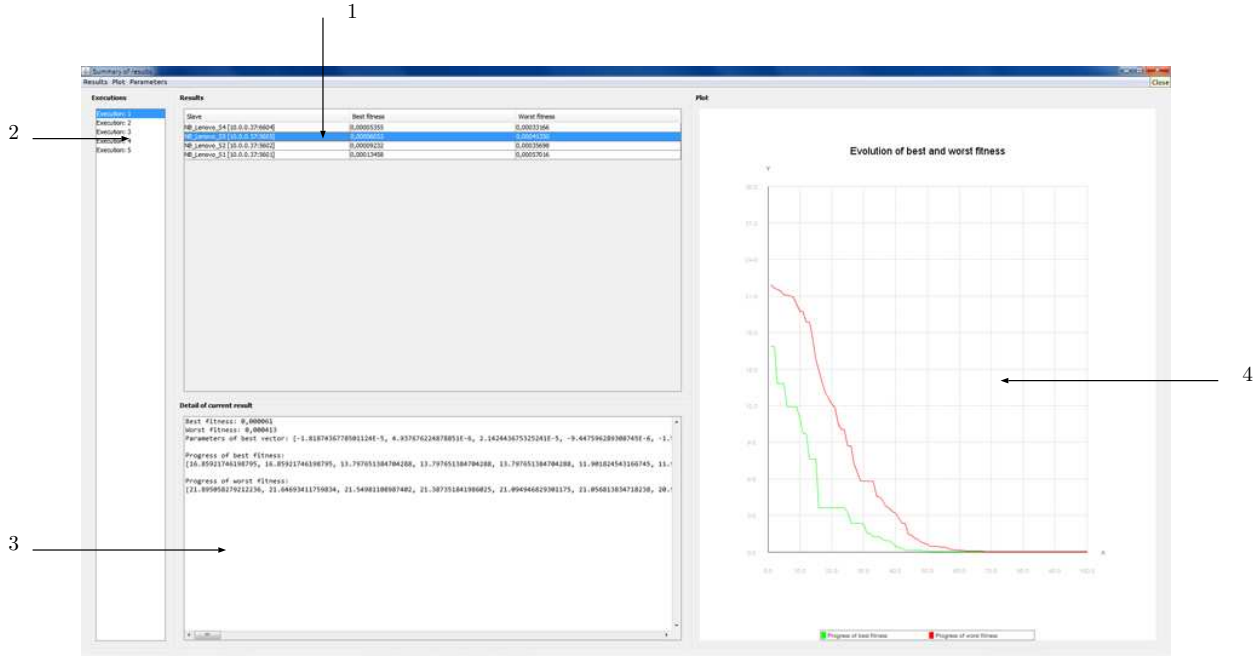
Obrázek 20: Řídící aplikace - panel pro nastavení vzorového jedince

1 - výběr typu parametru (reálný, celočíselný, diskrétní); 2 - potvrzení aktuálního nastavení vzorového jedince; 3 - načtení vzorového jedince ze souboru XML; 4 - minimální a maximální hodnota reálného parametru; 5 - minimální a maximální hodnota celočíselného parametru; 6 - výčet povolených hodnot diskrétního parametru, jednotlivé hodnoty se oddělují středníkem;



Obrázek 21: Řídící aplikace - panel pro výběr typu distribuované DE

1 - každý výpočetní uzel bude samostatně zpracovávat algoritmus DE; 2 - paralelní diferenciální evoluce s migrací jedinců;



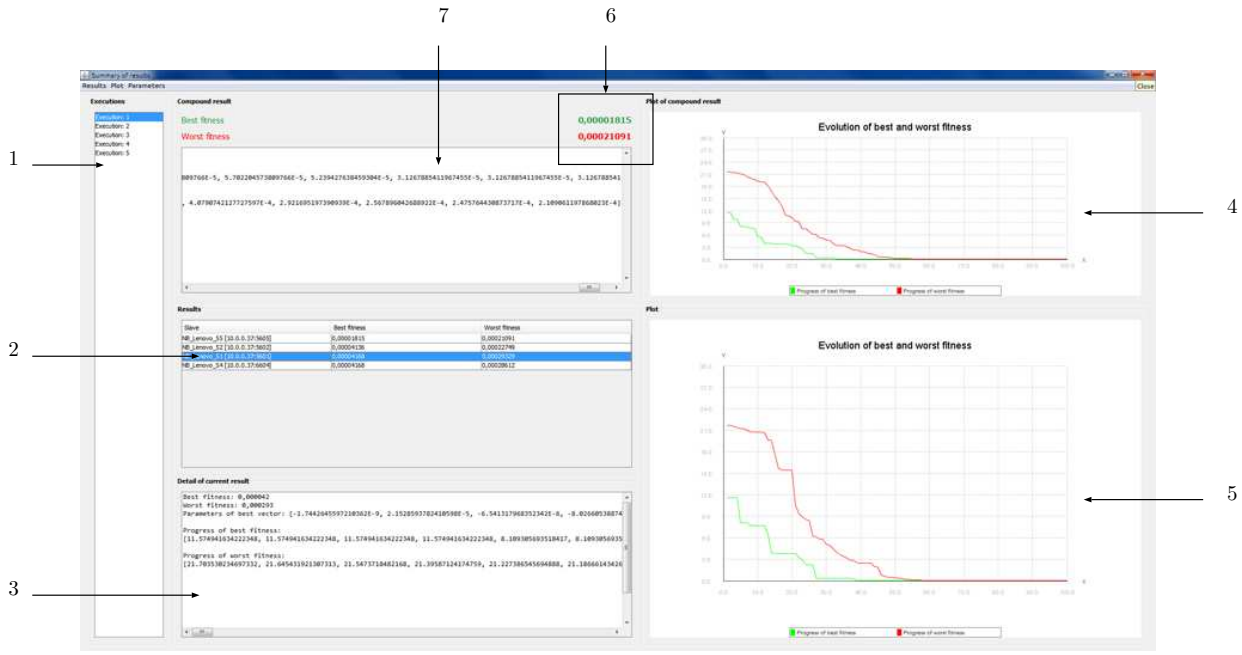
Obrázek 22: Řídící aplikace - panel s výsledky základní verze DE

jedince. Oba tyto průběhy jsou poté pro přehlednější zobrazení zaneseny do grafu, který je umístěn v panelu 4. Na obrázku 23 jsou zobrazeny výsledky získané pomocí PDE. V levé části (1) lze opět vidět, že algoritmus byl spuštěn pětkrát. Vzhledem k tomu, že jednotlivé výpočetní uzly spolupracovaly při vykonávání algoritmu, je k dispozici celkový složený výsledek, který je v aplikaci označován jako *Compound result*. Tento výsledek je poté prezentován v panelech 4, 6 a 7. Na panelu 6 je nejlepší a nejhorší získaná fitness hodnota. Pod těmito hodnotami se nachází panel 7, v němž je zapsán celkový průběh vývoje nejlepší a nejhorší hodnoty účelové funkce. Tyto hodnoty jsou získány tak, že po každém generačním cyklu a po dokončení migrace je vybrán vždy jeden nejlepší a jeden nejhorší výsledek ze všech subpopulací. V rámci panelu 4 jsou pak tyto hodnoty převedeny do grafu, aby byl pohled na výsledky přehlednější. Panely 2, 3 a 5 pak zachycují totéž jako panely 7, 6 a 4, jen s tím rozdílem, že se nejedná o celkový výsledek, ale o výsledky v rámci jedné subpopulace.

A.2.2 Načtení vzorového jedince z XML souboru

Definice vzorového jedince se provádí na panelu s označením *Specimen* a existují dvě možnosti, jak může být jedinec vytvořen.

1. přímé zadání parametrů prostřednictvím GUI aplikace
2. načtení vzorového jedince ze souboru



Obrázek 23: Řídící aplikace - panel s výsledky PDE

První možnost je vhodná pouze pro případy, kdy definujeme jedince s menším počtem parametrů. V případě, že potřebuje jedince s mnoha parametry, je zadávání skrze GUI poměrně zdlouhavé. Stejně tak pokud používáme daného vzorového jedince velmi často, je z uživatelského hlediska celkem nepříjemné vkládat stejné parametry znovu při každém spuštění aplikace.

Z těchto důvodů je možno vzorového jedince načíst z XML souboru. Několik takových XML souborů je již vytvořeno a umístěno na CD přiloženém k této práci. Pro potřeby vytvoření nového vzorového jedince je ale nutné popsat strukturu XML souboru, ze kterého bude aplikace schopna jedince načíst.

Kořenovým elementem v souboru musí být vždy `<specimen>`. Dále musí následovat vnořené elementy `<parameter>` s atributem, který určuje, o jaký typ parametru se bude jednat. Povolené hodnoty jsou `real, int, discrete`. Celočíselný a reálný parametr dále vyžaduje dva vnořené elementy `<min>` a `<max>`, pomocí nichž se určuje minimální a maximální hodnota parametru. Diskrétní parametr vyžaduje několik vnořených elementů `<value>`, jimiž se specifikují povolené hodnoty. Následují výpisy 1, 2, a 3, které znázorňují zápis jednotlivých typů parametrů.

```
<parameter type = ”real”>
    <min>-10.5</min>
    <max>10.5</max>
</parameter>
```

Výpis 1: Ukázka definice reálného parametru s rozsahem -10,5 až 10,5

```
<parameter type = ”int”>
    <min>-5</min>
    <max>5</max>
</parameter>
```

Výpis 2: Ukázka definice celočíselného parametru s rozsahem -5 až 5

```
<parameter type = ”discrete”>
    <value>1</value>
    <value>2</value>
    <value>3</value>
</parameter>
```

Výpis 3: Ukázka definice diskrétního parametru s hodnotami $\{1, 2, 3\}$

Nyní předpokládejme, že chceme definovat vzorového jedince se třemi parametry, přičemž první bude reálný v rozsahu -10,5 až 10,5, druhý celočíselný v rozsahu -5 až 5 a poslední parametr bude diskrétní. Povolené hodnoty budou $\{true, false, 1, 2, 3, January\}$. Celý XML soubor by pak vypadal stejně jako ve výpisu 4.

```
<specimen>
  <parameter type = ”real”>
    <min>-10.5</min>
    <max>10.5</max>
  </parameter>
  <parameter type = ”int”>
    <min>-5</min>
    <max>5</max>
  </parameter>
  <parameter type = ”discrete”>
    <value>true</value>
    <value>false</value>
    <value>1</value>
    <value>2</value>
    <value>3</value>
    <value>January</value>
  </parameter>
</specimen>
```

Výpis 4: Ukázka XML souboru definujícího vzorového jedince

A.3 Nastavení aplikací pomocí konfiguračních souborů

Základní nastavení obou aplikací se provádí skrze jejich konfigurační soubory, které musí být vždy umístěny ve stejném adresáři jako soubor pro spuštění dané aplikace.

Konfigurační soubory mají název `config.properties`. V případě výpočetní aplikace vypadá konfigurační soubor obdobně jako na výpisu 5.

```
1 GROUP_ADDRESS = 239.2.3.4
2 GROUP_PORT = 5454
3
4 REPORT_PERIOD = 10000
5 ENABLE_REPORTING = true
6
7 TCP_RECEIVING_PORT = 6604
```

Výpis 5: Ukázka konfiguračního souboru výpočetní aplikace

`GROUP_ADDRESS` a `GROUP_PORT` představují IP adresu a číslo portu pro zasílání datagramů skrze multicast. `REPORT_PERIOD` určuje v jednotkách milisekund, jak často bude tento datagram odesílán, což se ale bude dít jen v případě, že `ENABLE_REPORTING` bude nastaveno na hodnotu `true`. Hodnota `TCP_RECEIVING_PORT` definuje číslo portu, přes který je aplikace připravena přijímat zprávy od řídící aplikace.

Konfigurační soubor řídící aplikace vypadá stejně jako ve výpisu 5, jen s tím rozdílem, že neobsahuje hodnoty `REPORT_PERIOD` a `ENABLE_REPORTING`. Význam zbylých hodnot zůstává obdobný jako v případě výpočetní aplikace.

B Přidání nové účelové funkce

V kapitole zabývající se implementací bylo uvedeno, že všechny účelové funkce jsou umístěny ve sdílené knihovně. Pokud tedy chceme přidat novou účelovou funkci, je třeba ji přidat do sdíleného projektu. Ve vývojovém prostředí tedy otevřeme projekt s názvem DDE_Common a do balíčku s názvem `optimizationfunction` přidáme novou třídu, která bude reprezentovat novou účelovou funkci. Tato třída musí být potomkem třídy `OptimizationFunction`, která obsahuje jednu abstraktní metodu s názvem `evaluate`. Tato metoda vrací typ `double` a očekává pole třídy `Object`, která byla použita záměrně, neboť ne všechny funkce musí nutně vyžadovat pouze číselné argumenty. Z toho vyplývá, že konkrétní funkce si musí přijaté argumenty přetypovat na vhodný datový typ.

Zkusme nyní vytvořit novou účelovou funkci, která bude očekávat číselné argumenty a její matematický předpis bude následující:

$$f(x) = \sum_{i=1}^{D} x_i^3$$

Z matematického zápisu je zřejmé, že se jedná o jednoduchou funkci, která pouze sečte třetí mocniny všech svých argumentů. Zdrojový kód v Jazyku Java ve výpisu 6 ukazuje, jak by třída definující výše uvedenou funkci mohla vypadat.

```
1  package skl0013.dde.common.optimizationfunction;
2
3  public class BasicFunction extends OptimizationFunction {
4
5      // Zadani nazvu funkce v tele konstruktoru neni povinne
6      public BasicFunction() {
7          name = "Our basic function";
8      }
9
10     @Override
11     public double evaluate(Object[] parameters) {
12         int lenght = parameters.length;
13
14         double output = 0;
15         for (int i = 0; i < lenght; i++) {
16             output += Math.pow((double)parameters[i], 3);
17         }
18         return output;
19     }
20 }
```

Výpis 6: Ukázka kódu jednoduché účelové funkce

Máme-li takto vytvořenou třídu, je třeba provést ještě jednu mírnou úpravu ve zdrojovém kódu knihovny. V balíčku `optimizationfunction` se nachází třída `FunctionProvider`, což je velmi triviální třída s jednou statickou metodou. Metoda se jmenuje `getAllFunctions()`, nemá žádné vstupní argumenty a jejím návratovým typem je pole účelových funkcí. Již

samotný název metody napovídá, že nedělá nic jiného, než že vrátí pole všech dostupných účelových funkcí. Tělo této metody je tedy potřeba pouze upravit tak, aby vracela i naši nově vytvořenou účelovou funkci. Úprava je velmi snadná, ale pro úplnost uvádíme ve výpisu 7 i ukázku zdrojového kódu. Na řádku 17 tohoto výpisu je vidět přidání nově vytvořené účelové funkce.

```
1  package skl0013.dde.common.optimizationfunction;
2
3  import java.util.Arrays;
4
5  public class FunctionProvider {
6
7     public static OptimizationFunction[] getAllFunctions() {
8        OptimizationFunction[] functions = new OptimizationFunction[]{
9              new AckleyFunction(),
10             new GriewangkFunction(),
11             new RastriginFunction(),
12             new SchwefelFunction(),
13             new SphereFunction(),
14             new SalomonFunction(),
15             new XorNeuralNetworkBinary(2),
16             new XorNeuralNetworkLogistic(2),
17             new BasicFunction()
18       };
19       Arrays.sort(functions);
20       return functions;
21    }
22 }
```

Výpis 7: Ukázka metody, která vrací pole účelových funkcí

Po přidání nové funkce do metody `getAllFunctions()` je veškerá potřebná implementace dokončena a je nutné celý projekt znovu přeložit a vytvořit tak novou verzi výstupního `.jar` souboru. Tento soubor je poté potřeba přidat k projektům `DDE_Master` a `DDE_Slave`, které následně stačí znovu zkompilovat a vytvořit jejich nové spustitelné `.jar` soubory. Zdrojový kód obou aplikací není třeba žádným způsobem modifikovat. Po vytvoření nových spustitelných souborů je celý proces přidání nové účelové funkce dokončen. Obě aplikace mají k dispozici stejnou novou verzi sdílené knihovny, takže jsou schopny pracovat s nově vytvořenou účelovou funkcí.

C Obsah přiloženého CD

Součástí této práce je přiložené CD, na němž se nachází několik adresářů. Popis adresářové struktury společně s vysvětlením, co dané adresáře obsahují, je následující:

- **documentation**
  - **pdf** - dokumentace ke zdrojovému kódu ve formátu PDF
  - **html** - dokumentace ke zdrojovému kódu ve formátu HTML
- **source_codes** - obsahuje další tři adresáře, v nichž jsou umístěny zdrojové kódy; jedná se o projekty vývojového prostředí IntelliJ IDEA
  - **master** - zdrojový kód řídící aplikace
  - **slave** - zdrojový kód výpočetní aplikace
  - **common** - zdrojový kód sdílené knihovny
- **results** - výsledky získané během provádění experimentů
- **specimens** - XML soubory se vzorovými jedinci
- **text** - text této práce ve formátu PDF
- **applications**
  - **master** - obsahuje spustitelný `jar` soubor řídící aplikace
  - **slave** - obsahuje spustitelný `jar` soubor výpočetní aplikace