Srovnání výkonu algoritmů PSO, SOMA a DE na problému predikce dodávaného tepla v systému centrálního zásobování teplem

A comparison of PSO, SOMA and DE regarding the issue of predicting Delivered Heat from Central District Heating Systems

Roman Mikala

Bakalářská práce
2013

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
akademický rok: 2012/2013

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Roman MIKALA**
Osobní číslo:	**A10143**
Studijní program:	**B3902 Inženýrská informatika**
Studijní obor:	**Informační a řídicí technologie**
Forma studia:	**prezenční**
Téma práce:	**Srovnání výkonu algoritmů Particle Swarm Optimization, Self Organizing Migrating Algorithm a Differencial Evolution na problému predikce dodávaného tepla v systému centrálního zásobování teplem**

Zásady pro vypracování:

1. **Zpracujte literární rešerši na dané téma, zaměřte se na algoritmy Particle Swarm Optimization, Self Organizing Migrating Algorithm a Differencial Evolution.**
2. **Popište možnosti implementace algoritmů pomocí platformy .NET.**
3. **Implementujte a otestujte dané algoritmy.**
4. **Srovnejte výkon algoritmů pro problém predikce dodávaného tepla v systému centrálního zásobování teplem.**
5. **Demonstrujte výsledky a proveďte zhodnocení.**

Rozsah bakalářské práce:

Rozsah příloh:

Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam odborné literatury:

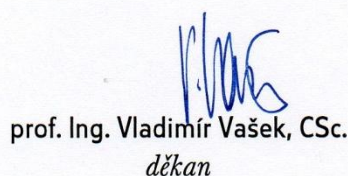
1. **ZELINKA, Ivan. Umělá inteligence v problémech globální optimalizace. 1. vyd. Praha: BEN – technická literatura, 2002, 189 s. ISBN 80-730-0069-5.**
2. **YU, Xinjie a Mitsuo GEN. Introduction to evolutionary algorithms. New York: Springer, c2010. Decision engineering. ISBN 18-499-6129-8.**
3. **WEISE, Thomas. Global Optimization Algorithms: Theory and Aplication [online]. 2009, 2009-06-26 [cit. 2013-02-01]. 2. Dostupné z: http://www.it-weise.de/projects/book.pdf**
4. **BRATTON, D. Defining a Standard for Particle Swarm Optimization. In: 2007 IEEE Swarm Intelligence Symposium: Honolulu, HI, 1-5 April 2007. Piscataway, NJ: IEEE, c2007, s. 120-127. DOI: 1-4244-0708-7.**
5. **KRAMPL J. Implementace vybraných evolučních algoritmů v prostředí .NET. Zlín, 2011. Bachelor thesis (Bc.). Tomas Bata University in Zlín, Faculty of Applied Informatics**
6. **KRÁL, Tomáš. Knihovna evolučních optimalizačních algoritmů v prostředí Java. Zlín, 2011. diplomová práce (Ing.). Univerzita Tomáše Bati ve Zlíně. Fakulta aplikované informatiky**

Vedoucí bakalářské práce: **Ing. Erik Král**
Ústav počítačových a komunikačních systémů

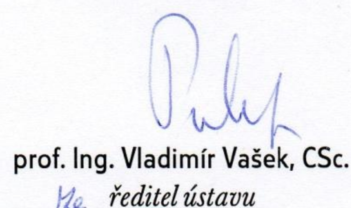
Datum zadání bakalářské práce: **24. února 2013**

Termín odevzdání bakalářské práce: **14. června 2013**

Ve Zlíně dne 24. února 2013



L.S.



prof. Ing. Vladimír Vašek, CSc.
děkan

prof. Ing. Vladimír Vašek, CSc.
ředitel ústavu

ABSTRAKT

Tato práce poskytuje přehled vybraných evolučních algoritmů, konkrétně PSO (Particle Swarm Optimization), SOMA (Self-Organizing Migrating Algorithm) a DE (Differential Evolution). Popisuje jednotlivé parametry, varianty a samotné algoritmy. V praktické části je popsána implementace zmíněných algoritmů v prostředí :NET Framework a srovnání jednotlivých algoritmů z hlediska přesnosti optimalizace při dané délce výpočtu.

Klíčová slova: optimalizace, evoluční algoritmy, účelová funkce, PSO, SOMA, DE, .NET Framework, kvalita algoritmů

ABSTRACT

Thesis provides an overview of selected evolutionary algorithms, namely PSO (Particle Swarm Optimization), SOMA (Self-Organizing Migrating Algorithm) and DE (Differential Evolution). It describes parameters, variants and algorithms themselves. The practical part describes the implementation of these evolutionary algorithms in the environment: .NET Framework and comparison of individual algorithms in terms of accuracy of optimization for a given length of calculation.

Keywords: optimization, evolutionary algorithms, cost function, PSO, SOMA, DE, .NET Framework, quality of algorithms

Chtěl bych poděkovat vedoucímu mé bakalářské práce panu Ing. Eriku Královi za seznámení s evolučními algoritmy a cenné rady při zpracování těchto algoritmů.

Prohlašuji, že

- beru na vědomí, že odevzdáním bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – bakalářskou práci nebo poskytnout licenci k jejímu využití jen s předchozím písemným souhlasem Univerzity Tomáše Bati ve Zlíně, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše);
- beru na vědomí, že pokud bylo k vypracování bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na bakalářské práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze bakalářské práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně ……………………….

podpis diplomanta

OBSAH

ÚVOD ... **9**
I TEORETICKÁ ČÁST ... **10**
1 EVOLUČNÍ ALGORITMY ... **11**
1.1 POPULACE ... 11
1.2 SPECIMEN ... 11
2 OPTIMALIZACE ROJENÍ ČÁSTIC (PARTICLE SWARM OPTIMIZATION) ... **13**
2.1 PARAMETRY A TERMINOLOGIE ... 13
2.2 PRINCIP ALGORITMU ... 14
2.2.1 Stanovení parametrů ... 14
2.2.2 Evoluční cyklus ... 15
2.2.3 Testování naplnění ukončovacích podmínek ... 15
2.2.4 Vyhodnocení ... 15
2.3 VARIANTY PSO ... 16
2.4 ALGORITMUS ... 17
3 SAMO-ORGANIZUJÍCÍ SE MIGRAČNÍ ALGORITMUS (SELF-ORGANISING MIGRATING ALGORITHM) ... **19**
3.1 PARAMETRY A TERMINOLOGIE ... 19
3.2 PRINCIP ALGORITMU ... 21
3.2.1 Stanovení parametrů ... 21
3.2.2 Migrační kola ... 22
3.2.3 Testování naplnění ukončovacích podmínek ... 23
3.2.4 Vyhodnocení ... 23
3.3 STRATEGIE SOMA ... 23
3.4 ALGORITMUS ... 24
4 DIFERENCIÁLNÍ EVOLUCE (DIFFERENTIAL EVOLUTION) ... **27**
4.1 PARAMETRY A TERMINOLOGIE ... 27
4.2 PRINCIP ALGORITMU ... 29
4.2.1 Stanovení parametrů ... 29
4.2.2 Započetí cyklu generace ... 29
4.2.3 Evoluční cyklus ... 29
4.2.4 Testování naplnění ukončovacích podmínek ... 30
4.2.5 Vyhodnocení ... 30
4.3 VARIANTY DE ... 30
4.3.1 Varianty křížení ... 30
4.3.2 Varianty DE ... 31
4.4 ALGORITMUS ... 33
5 FRAMEWORK .NET ... **35**
II PRAKTICKÁ ČÁST ... **36**
6 OPTIMALIZACE FUNKCE ... **37**

6.1 ÚČELOVÁ FUNKCE 37
6.2 APROXIMAČNÍ FUNKCE 37
7 INTERFACE IEVOLUTIONALGORITHM 39
7.1 VLASTNOSTI (PROPERTIES) 39
7.2 METODY (METHODS) 40
8 OPTIMALIZACE ROJENÍ ČÁSTIC (PARTICLE SWARM OPTIMIZATION) 41
8.1 ČLENSKÉ PROMĚNNÉ (FIELDS) 42
8.2 VLASTNOSTI (PROPERTIES) 43
8.3 METODY (METHODS) 45
8.3.1 Konstruktor EvolutionAlgorithmSPSO 45
8.3.2 GetPositionToEvaluate 47
8.3.3 Provedení algoritmu – UpdatePositions 47
8.4 VÝVOJOVÝ DIAGRAM ALGORITMU 50
9 SAMO-ORGANIZUJÍCÍ SE MIGRAČNÍ ALGORITMUS (SELF-ORGANISING MIGRATING ALGORITHM) 51
9.1 ČLENSKÉ PROMĚNNÉ (FIELDS) 52
9.2 VLASTNOSTI (PROPERTIES) 53
9.3 METODY (METHODS) 55
9.3.1 Konstruktor EvolutionAlgorithmSOMA 55
9.3.2 GetPositionToEvaluate 57
9.3.3 Provedení algoritmu - UpdatePositions 57
9.4 VÝVOJOVÝ DIAGRAM 59
10 DIFERENCIÁLNÍ EVOLUCE (DIFFERENTIAL EVOLUTION) 60
10.1 ČLENSKÉ PROMĚNNÉ (FIELDS) 61
10.2 VLASTNOSTI (PROPERTIES) 62
10.3 METODY (METHODS) 64
10.3.1 Konstruktor EvolutionAlgorithmDE 64
10.3.2 GetPositionToEvaluate 66
10.3.3 Provedení algoritmu – UpdatePositions 66
10.4 VÝVOJOVÝ DIAGRAM 68
11 POROVNÁNÍ ALGORITMŮ 69
ZÁVĚR 71
ZÁVĚR V ANGLIČTINĚ 72
SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK 75
SEZNAM OBRÁZKŮ 76
SEZNAM TABULEK 77
SEZNAM PŘÍLOH 78

ÚVOD

Optimalizační algoritmy slouží k vyhledání extrému optimalizovaných funkcí. Evoluční algoritmy jsou speciální verzí těchto algoritmů, které využívají evolučních principů, jako například křížení, genetické mutace a dalších metod, k postupnému vyvinutí nejlepšího řešení.

Hlavní výhodou je schopnost nalézt řešení i u velmi komplexních optimalizačních úloh hledáním globálních extrémů těchto funkcí i za předpokladu, že jich daný problém má více. Nevýhodou pak je jejich závislost na náhodě, čímž dělají obtížné dopředu předvídat, zdali nalezené řešení je to nejlepší možné.

Tato práce poskytuje popis vybraných evolučních algoritmů, konkrétně optimalizaci rojení částic (PSO – Particle Swarm Optimization), samo-organizující se migrační algoritmus (SOMA – Self-Organizing Migrating Algorithm) a diferenciální evoluci (DE – Differential Evolution). Práce popisuje jednotlivé řídící parametry a různé varianty těchto algoritmů, které ovlivňují chování a účinnost vybraných algoritmů.

Cílem mé práce je srovnání výkonu těchto algoritmů na problému predikce dodávaného tepla v systému centrálního zásobování teplem. Porovnávám účinnost algoritmu po přesně zadanou dobu, určenou počtem vyhodnocení účelové funkce. Účelová a aproximační funkce jsou předem zadané. Úkolem je rozhodnout, který ze zpracovaných algoritmů je na daný problém nejúčinnější.

I. TEORETICKÁ ČÁST

1 EVOLUČNÍ ALGORITMY

Evoluční algoritmy jsou silným nástrojem v hledání optimálního řešení tam, kde uplatnění analytických metod je příliš složité až téměř nemožné. Jak název napovídá, evoluční algoritmy využívají mechanismy evoluce, kde se pomocí křížení, mutačních procesů a dalších metod snaží postupně vyvinout nejlepší řešení. Největší výhoda těchto algoritmů spočívá v tom, že jsou schopné řešit i velmi komplexní optimalizační problémy hledáním globálních extrémů. Nevýhodou je spoléhání na náhodu, díky čemuž nelze předem odhadnout, zda je dané řešení nejlepším možným.

Smyslem evolučních algoritmů je cyklické vytváření nových populací a náhrada populací starých těmi novými. To se děje pomocí přesně definovaných matematických pravidel. To jaká pravidla a reprezentace jedinců v populaci jsou použita, pak určuje, o jaký evoluční algoritmus se jedná. [1][2][3]

1.1 Populace

Evoluční algoritmy jsou založeny na práci se skupinou jedinců, které se říká populace. Populace může být zobrazena jako matice počet jedinců × počet parametrů jedince ($NP \times D$). Každý jedinec zde představuje aktuální řešení daného problému. Dá se říct, že se jedná o množinu argumentů účelové funkce, jejíž optimální číselná kombinace je hledána. Každý jedinec tak má i svou vlastní hodnotu účelové funkce – CV („cost value“), která určuje, jak je jedinec vhodný pro další vývoj populace. Tato hodnota se neúčastní vlastního evolučního procesu, nese pouze informaci o kvalitě daného jedince. [1][3]

Tab. 1. Příklad populace

	Jedinec 1	**Jedinec 2**	**Jedinec 3**	**Jedinec 4**	**Jedinec 5**
CV	*-14,82*	*-5,82*	*1,41*	*0,34*	*-1,05*
Parametr 1	1,00	0,05	1,98	1,55	1,79
Parametr 2	1,66	0,88	1,85	1,32	0,19
Parametr 3	0,19	3,04	2,01	0,79	0,99
Parametr 4	3,62	3,78	1,77	0,97	2,28

1.2 Specimen

K vytvoření populace je třeba nadefinovat vzor („Specimen“), podle kterého se celá populace vygeneruje. Ve vzorovém jedinci jsou pro každý parametr konkrétního jedince definovány tři parametry. Jsou to typ proměnné, spodní a horní hranice intervalu. Typ

proměnné určuje, zda je daný jedinec celočíselný, reálný, diskrétní apod. Spodní a horní hranice pak hlídají, že je daný jedinec v intervalu, v němž se hodnota každého parametru jedince nachází. Správné definování hranic je důležité, protože jinak se může stát, že úloha nebude mít fyzicky proveditelné řešení (např. letadlo bez křídel, záporná tloušťka stěny, apod.). Hranice jsou také důležité z hlediska evolučních algoritmů. Může se stát, že daný problém bude reprezentován plochou, na níž lokální extrémy nabývají větších hodnot, čím jsou dále od počátku. To může způsobit, že evoluce bude probíhat do nekonečna v případě, že nebude definováno ukončení v závislosti na počtu evolučních kol. [1][3]

Počáteční populace jedinců je generována z rovnice (1), ve které rnd značí náhodné číslo v rozsahu <0; 1>; i je index jedince a j je index parametru jedince. Hi označuje horní hranici a Lo spodní hranici intervalu. [3]

$$x_{i,j} = rnd \cdot \left(x_{i,j}^{Hi} - x_{i,j}^{Lo}\right) + x_{i,j}^{Lo} \tag{1}$$

2 OPTIMALIZACE ROJENÍ ČÁSTIC (PARTICLE SWARM OPTIMIZATION)

Optimalizaci rojení částic (PSO – particle swarm Optimization) vyvinuli v roce 1995 Dr. Eberhart a Dr. Kennedy. Inspirovali se při tom chováním hejna ptáků a rybích školek. PSO sdílí mnoho věcí s ostatními evolučními algoritmy – systém je vytvořen s populací náhodných jedinců vytvořených pomocí vzoru (Specimen) a hledá optimální řešení vylepšováním jednotlivých generací. Na rozdíl od genetických algoritmů ale nevytváří nové jedince (zde označovaní jako částice) pomocí křížení a mutací, ale částice „prolétávají" plochou problému směrem k nejlepšímu řešení. Zde popsaná varianta algoritmu se nazývá Standard for Particle Swarm Optimization (SPSO) a byla zveřejněna v roce 2007 v [4].[5]

2.1 Parametry a terminologie

Činnost a kvalita tohoto algoritmu ovlivněna řídícími parametry. Proto je nutné je správně nastavit. Jejich označení a význam je:

Tab. 2. Přehled parametrů algoritmu PSO. Převzato z [3]

Řídící parametr	**Interval**	**Poznámka**
PopSize	<20; 40>	Velikost populace
D	určeno problémem	Dimenze problému
MaxEvaluations	uživatel	Maximální počet vyhodnocení účelové funkce
C1		Kognitivní parametr
C2		Sociální parametr

Dimenze problému – D

Udává počet parametrů účelové funkce. Velikost je závislá na definovaném problému, popsaném účelovou funkcí. Změnu lze provést pouze reformulací celého problému. [3]

Velikost populace – $PopSize$

Tento parametr určuje počet jedinců (zde pojmenovaných jako částice) v populaci. Doporučený počet částic je v intervalu $< 20; 40 >$ [3]

Učící parametry - $C1, C2$

Tyto parametry ovlivňují směr cestování jednotlivých částic. Parametr $C1$ se také nazývá „kognitivní parametr" a určuje směr pohybu k dosud nejlepší pozici aktuální částice ($pBest$). Parametr $C2$ je nazýván „sociální parametr" a ovlivňuje pohyb částice k celkové nejlepší pozici ($gBest$). Podle [4] je ideální nastavení těchto parametrů na hodnotu 2,05. [3][4][6]

Faktor zúžení – χ (chi)

Tento parametr byl zaveden k zlepšení výsledků algoritmu. Počítá se z rovnice

$$\chi = \frac{2}{\left|2 - \varphi - \sqrt{\varphi^2 - 4\varphi}\right|} \tag{2}$$

za předpokladu, že

$$\varphi = C1 + C2 > 4.0 \tag{3}$$

Bylo zjištěno v [4], že ideální nastavení pro parametry $C1$ a $C2$ je

$$C1 = C2 = 2{,}05 \tag{4}$$

Maximální počet evaluací účelové funkce – $MaxEvaluation$

Tento parametr závisí pouze na uživateli a řídí maximální počet vyhodnocení účelové funkce. Tento parametr nahrazuje původní ukončovací parametr $Generations$, který označoval maximální počet evolučních cyklů. K záměně došlo z důvodu porovnání tohoto algoritmu s algoritmy DE a SOMA v rámci této práce. Vzhledem k tomu, že ne všechny algoritmy mají stejný počet vyhodnocení během jednoho evolučního/migračního kola, muselo dojít k této změně, aby se při porovnání algoritmů vycházelo ze stejného časového úseku (zde tedy určeného počtem vyhodnocení účelové funkce). Jedná se o ukončovací parametr. Musí splňovat podmínku $MaxEvaluation > 0$.

2.2 Princip algoritmu

2.2.1 Stanovení parametrů

Před započetím provádění algoritmu je nutné stanovit řídící a ukončovací parametry. Jde o parametry, které zajišťují a upravují chod celé evoluce. Tyto parametry jsou popsány v tabulce výše (*Tab. 2*) a jsou to učící parametry $C1$ a $C2$,χ (chi), $MaxEvaluation$, $PopSize$ – velikost populace a D – rozměr jedince (odvozen z rozměru daného problému). Nutné je také nadefinovat účelovou funkci, která bude optimalizována. Tato funkce by

měla vracet skalár, který bude použit jako měřítko kvality daného jedince (pro hledání maxima bude nejlepší jedinec ten, který bude mít tento skalár největší a pro minimum nejmenší). [4]

2.2.2 Evoluční cyklus

Na začátku každého evolučního cyklu je nutné určit pro každou částici její nejlepší hodnotu účelové funkce. Tato hodnota se určuje z celé historie průchodu a označuje se jako $pBest$ („particle best“). Jako $gBest$ („global best“) je potom pro každou částici vybrána ta částice, která leží v okolí aktivní částice a má nejlepší hodnotu účelové funkce. Toto okolí tvoří částice, které mají index $i \pm 1$, za předpokladu, že aktivní částice je označená indexem i. V případě, že se jedná o první ($i = 0$), nebo poslední částici ($i = PopSize - 1$), pak se bere jako vedlejší částice ta poslední, respektive první. Po té, co známe $gBest$ a $pBest$, můžeme vypočítat rychlost částice podle rovnice

$$v_{i,j} = \chi\left(v_{i,j} + C1 \cdot rnd \cdot \left(pBest_{i,j} - X_{i,j}\right) + C2 \cdot rnd \cdot \left(gBest_j - X_{i,j}\right)\right) \tag{5}$$

$X_{i,j}$ je pozice aktuální částice. $C1$ a $C2$ jsou řídící konstanty a χ je faktor zúžení.

Z rychlosti částice a její aktuální pozice se vypočítá nová pozice pomocí rovnice

$$X_{i,j} = X_{i,j} + v_{i,j} \tag{6}$$

Poté se provede ohodnocení nové částice. Pokud je účelová funkce nové částice větší než u staré, pak se z této částice stane její nové $pBest$. Pokud se tak nestane, bude i v příštím evolučním kole počítáno se starým $pBest$. Pokud nedojde k ukončení díky naplnění ukončovacích podmínek, pokračuje celý evoluční cyklus opět výběrem $gBest$ z okolí prvního jedince. [2][3][4][7]

2.2.3 Testování naplnění ukončovacích podmínek

Algoritmus PSO je standardně ukončen naplněním maximálního počtu evolučních cyklů. Pro tuto práci jsem musel tuto ukončovací podmínku modifikovat na naplnění maximálního počtu provedených evaluací, z důvodu porovnání tohoto algoritmu s algoritmy SOMA a DE.

2.2.4 Vyhodnocení

Při splnění jedné z ukončovacích podmínek je algoritmus zastaven a jako jeho výsledná hodnota je brána hodnota nejlepší hodnota ze všech částic.

2.3 Varianty PSO

Existuje několik variant algoritmu PSO, které dále upravují chování částic. Jedná se o kombinace parametru hybnosti (w) s faktorem zúžení χ.[3][4][6]

Faktor zúžení, jak již bylo napsáno v popisu algoritmu, se počítá z rovnice (2)

Parametr hybnosti – w (w_{start}, w_{end})

Parametr hybnosti byl zaveden k snížení rychlosti částic v pokročilé fázi algoritmu, čímž je dosaženo přesnějšího výsledku. Pro použití je potřeba zavést dva nové řídící parametry, w_{start}, w_{end}, kde w_{start} značí hybnost částice na začátku vykonávání algoritmu a w_{end} hybnost při ukončení algoritmu.[3]

Tento parametr se potom počítá z rovnice

$$w = w_{start} - \frac{(w_{start} - w_{end}) \cdot iterations}{MaxIterations} \tag{7}$$

Původní verze PSO

Původní verze algoritmu PSO nepoužívala ani parametr hybnosti, ani faktoru zúžení. Tato verze používala parametr V_{max} pro omezení maximální rychlosti částice. [4]

$$v_{i,j} = v_{i,j} + C1 \cdot rnd \cdot \left(pBest_{i,j} - X_{i,j}\right) + C2 \cdot rnd \cdot \left(gBest_j - X_{i,j}\right) \tag{8}$$

Varianta s použitím hybnosti

Po uveřejnění původní verze PSO bylo snaha odstranit parametr V_{max}, protože tato metoda působila uměle a byla složitá na vybalancování. Hodnota maximální rychlosti totiž nepůsobila stejně na velkých a malých plochách. Proto se zavedl tzv. parametr hybnosti, který upravuje rychlost v závislosti na počtu provedených evolučních cyklů, a ne konstantní hodnotou, jako tomu bylo u V_{max}.[4]

$$v_{i,j} = w \cdot v_{i,j} + C1 \cdot rnd \cdot \left(pBest_{i,j} - X_{i,j}\right) + C2 \cdot rnd \cdot \left(gBest_j - X_{i,j}\right) \tag{9}$$

Varianta s použitím faktoru zúžení

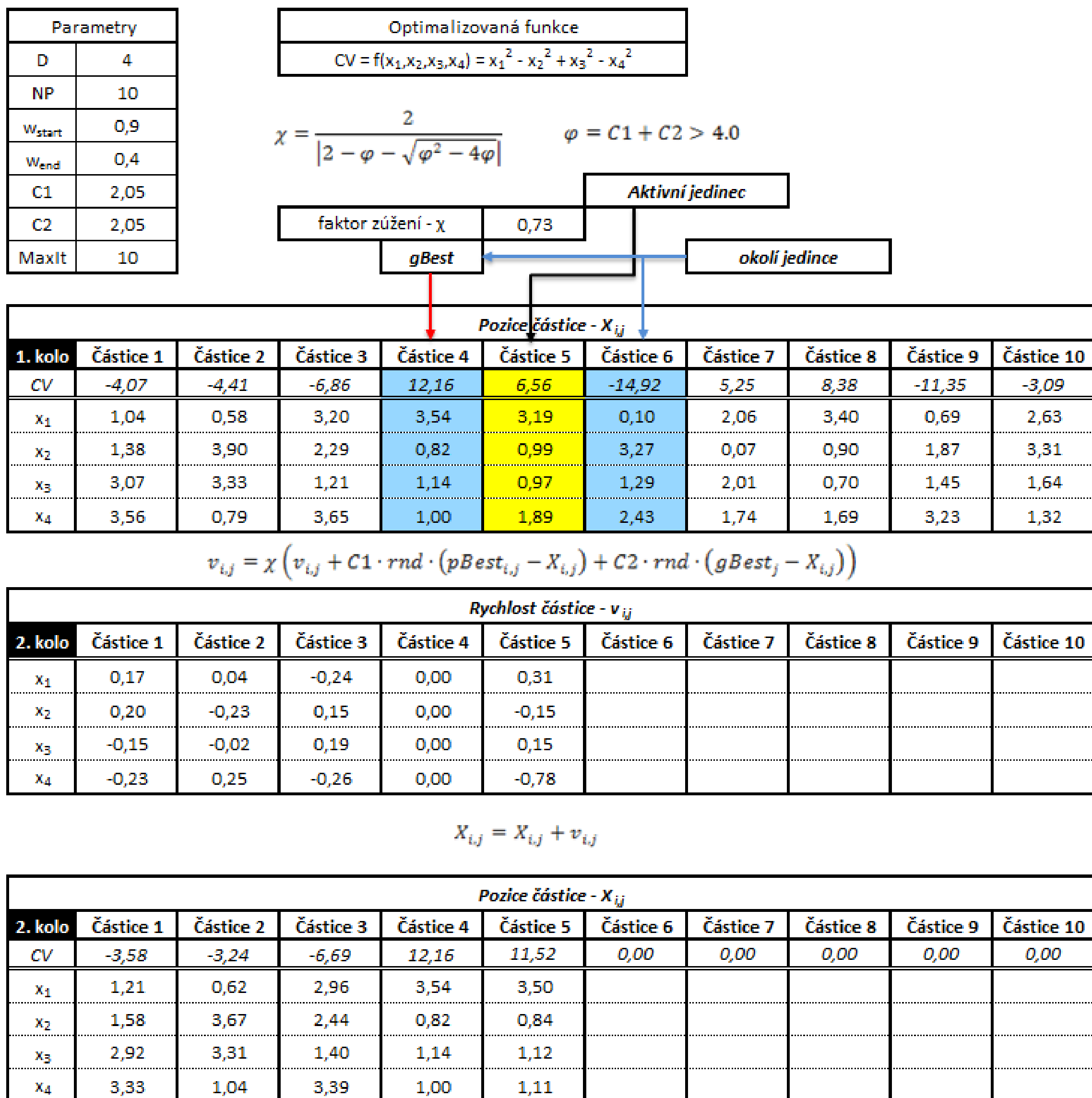
Současně s variantou s použitím hybnosti byla vyvíjena také varianta s použitím faktoru zúžení. Ukázalo se, že tato varianta je speciální verzí varianty s použitím hybnosti, která má řídící parametry vypočítané analyticky. Tato varianta byla pojmenována jako SPSO (Standard for Particle swarm Optimization) a byla uvedena v roce 2007 v [4]. Tuto

variantu zpracovávám v praktické části a je také popsána zde Využívá rovnici (5). Ukázalo se, že ideální hodnoty parametrů C1 a C2 je hodnota 2,05, jak popsáno v rovnici (4). [4]

2.4 Algoritmus

Použitá strategie algoritmu je SPSO popsaná v [4].

1. Nastavení řídících parametrů $C1$ a $C2$**.** Alokování polí pro populaci (X), nejlepší jedince (P), rychlost jedinců (V) a nejlepší hodnoty účelové funkce (f).
2. Naplnění f maximální hodnotou ($double.maxValue$) pro každého jedince.
3. Nastavení výchozí rychlosti pro všechny jedince a všechny parametry na 0.
4. Porovnání, zda je počet již provedených evaluací ($NumberOfEvaluations$) větší než maximální povolený ($MaxEvaluation$). Pokud ano, ukončit algoritmus.
5. Ohodnocení jedinců účelovou funkcí.
6. Zvednutí počtu ohodnocení ($NumberOfEvaluations$) o právě provedené.
7. Porovnání hodnot účelových funkcí nejlepších jedinců (f) s nově ohodnocenýma (nf). V případě že jsou horší, tak pokračovat **8**. Jinak pokračovat **9**.
8. Překopírovat lepšího jedince do P a jeho hodnotu do f.
9. Pro každého jedince (index i): Nalezení předchozího (iB) a následujícího (iF) jedince. Populace je zde chápána jako nekonečný kruh (z posledního se přeskakuje na prvního).
10. Z těchto tří jedinců vybrat toho nejlepšího (g)
11. Pro každý parametr jedince (index d): Vypočítá se nová rychlost jedince pomocí vzorce (5) a nová pozice pomocí vzorce (6).
12. Zvedne se index parametru (d). Pokud se nejedná o poslední parametr, pokračuje se **11**.
13. Zvedne se index jedince (i). Pokud se nejednalo o posledního jedince, pokračuje se **9**. Pokud ano, pokračuje se **4**.



Parametry	
D	4
NP	10
w_{start}	0,9
w_{end}	0,4
C1	2,05
C2	2,05
MaxIt	10

Optimalizovaná funkce
$CV = f(x_1,x_2,x_3,x_4) = x_1^2 - x_2^2 + x_3^2 - x_4^2$

$$\chi = \frac{2}{\left|2 - \varphi - \sqrt{\varphi^2 - 4\varphi}\right|} \qquad \varphi = C1 + C2 > 4.0$$

faktor zúžení - χ	0,73

Pozice částice - $X_{i,j}$										
1. kolo	Částice 1	Částice 2	Částice 3	Částice 4	Částice 5	Částice 6	Částice 7	Částice 8	Částice 9	Částice 10
CV	*-4,07*	*-4,41*	*-6,86*	*12,16*	*6,56*	*-14,92*	*5,25*	*8,38*	*-11,35*	*-3,09*
x_1	1,04	0,58	3,20	3,54	3,19	0,10	2,06	3,40	0,69	2,63
x_2	1,38	3,90	2,29	0,82	0,99	3,27	0,07	0,90	1,87	3,31
x_3	3,07	3,33	1,21	1,14	0,97	1,29	2,01	0,70	1,45	1,64
x_4	3,56	0,79	3,65	1,00	1,89	2,43	1,74	1,69	3,23	1,32

$$v_{i,j} = \chi\left(v_{i,j} + C1 \cdot rnd \cdot \left(pBest_{i,j} - X_{i,j}\right) + C2 \cdot rnd \cdot \left(gBest_j - X_{i,j}\right)\right)$$

Rychlost částice - $v_{i,j}$										
2. kolo	Částice 1	Částice 2	Částice 3	Částice 4	Částice 5	Částice 6	Částice 7	Částice 8	Částice 9	Částice 10
x_1	0,17	0,04	-0,24	0,00	0,31					
x_2	0,20	-0,23	0,15	0,00	-0,15					
x_3	-0,15	-0,02	0,19	0,00	0,15					
x_4	-0,23	0,25	-0,26	0,00	-0,78					

$$X_{i,j} = X_{i,j} + v_{i,j}$$

Pozice částice - $X_{i,j}$										
2. kolo	Částice 1	Částice 2	Částice 3	Částice 4	Částice 5	Částice 6	Částice 7	Částice 8	Částice 9	Částice 10
CV	*-3,58*	*-3,24*	*-6,69*	*12,16*	*11,52*	*0,00*	*0,00*	*0,00*	*0,00*	*0,00*
x_1	1,21	0,62	2,96	3,54	3,50					
x_2	1,58	3,67	2,44	0,82	0,84					
x_3	2,92	3,31	1,40	1,14	1,12					
x_4	3,33	1,04	3,39	1,00	1,11					

Obr. 1. Ukázka provedení algoritmu SPSO.

3 SAMO-ORGANIZUJÍCÍ SE MIGRAČNÍ ALGORITMUS (SELF-ORGANISING MIGRATING ALGORITHM)

Samo-organizující se migrační algoritmus existuje od roku 1999. Jeho činnost je založena na geometrických principech. SOMA se řadí mezi evoluční algoritmy, navzdory faktu, že během jeho průběhu nejsou vytvářeni další potomci. Mezi evoluční algoritmy je řazen z toho důvodu, že pracuje s populacemi, podobně jako diferenciální evoluce, či jiné genetické algoritmy. Původní myšlenka, která vedla ke stvoření tohoto algoritmu, spočívá v napodobení chování skupiny inteligentních jedinců, kteří spolupracují při řešení společného problému (např. hledání potravy ve stádu). [1]

Vzhledem k tomu, že hlavní myšlenka SOMA není založena na principech evoluce, ale spíše na již zmíněných principech „smečky“, neklasifikuje se mezi algoritmy evoluční, ale tzv. memetické. [1]

Vlastnost samo-organizace u SOMA plyne z faktu, že se jedinci ovlivňují navzájem při hledání nejlepšího možného řešení, což mnohdy vede k tomu, v prostoru možných řešení vznikají skupiny jedinců, které se rozpadají či spojují a při tom putují přes prohledávaný prostor. Lze to popsat i tak, že si skupina jedinců (populace) sama organizuje vzájemný pohyb. [1]

3.1 Parametry a terminologie

U tohoto algoritmu nevznikají noví jedinci. Místo toho jedinci cestují prostorem řešení. Proto se nedá u tohoto algoritmu mluvit o generacích (jako v případě ostatních evolučních algoritmů), ale tento cyklus byl přejmenován na migrace. Nevýhodou SOMA, stejně jako ostatních evolučních algoritmů, je závislost kvality fungování algoritmu na nastavení řídících parametrů. Oproti [1] zde proběhla změna názvosloví u některých parametrů, jak byla popsaná v [8]: [1][8]

Tab. 3. Přehled parametrů algoritmu SOMA. Převzato z [1]

Řídící parametr	Interval	Poznámka
PopSize	<10*D; uživatel>	Velikost populace - Při složitějších funkcí 0,2*D - 0,5*D
D	určeno problémem	Dimenze problému
MaxEvaluations	uživatel	Maximální počet vyhodnocení účelové funkce
PathLength	<1,1; 3>	Velikost cesty jedince
Step	<0,11; PathLength>	Velikost kroku jedince. Pokud je nastavena nízka hodnota, je provádění algoritmu výkonově náročnější
PRT	<0; 1>	Určuje směr pohybu nového jedince

Dimenze problému – D

Udává počet parametrů účelové funkce. Velikost je závislá na definovaném problému, popsaném účelovou funkcí. Změnu lze provést pouze reformulací celého problému. [1][2]

Velikost populace – $PopSize$

Tento parametr určuje počet jedinců v populaci. Velikost tohoto parametru by neměla být menší než 4. To je minimální velikost populace, při které diferenciální evoluce ještě pracuje. [1][8]

$PathLength$

Tento parametr určuje, jak daleko se aktivní jedinec zastaví od jedince vedoucího. Pokud je $PathLength = 1$, pak se aktivní jedinec zastaví na pozici vedoucího, při $PathLength = 2$ za ním ve stejné vzdálenosti, ze které startoval. Pokud je $PathLength < 1$ pak se aktivní jedinec zastaví před vedoucím, což povede k degeneraci migračního procesu. Z toho vyplývá, že by se tento parametr měl nacházet v intervalu <1.1; 3> [1][8]

$Step$

Parametr Step určuje, jak moc bude rozdělena cesta jedince v jednotlivém migračním kole. Pro jednoduché problémy je možné použít velkou hodnotu pro urychlení algoritmu. Pro složitější problémy se doporučuje nastavit tento parametr na nízkou hodnotu. Důležité také nastavit $Step$ tak, aby vzdálenost mezi vedoucím a aktivním jedincem nebyla celočíselným násobkem parametru tohoto parametru. Tím by došlo ke snížení rozmanitosti populace. Doporučuje se nastavovat v intervalu <0.11; $PathLength$> [1]

PRT

Podle tohoto parametru se tvoří tzv. perturbační vektor ($PRTVector$), který ovlivňuje směr pohybu jedince. Jeho optimální hodnota je 0.1. Nastavení se pohybuje v intervalu <0; 1> [1]

Minimální rozmanitost (minimal diversity) - MinDiv

Ukončovací parametr. Tento parametr znázorňuje maximální možný rozdíl mezi nejlepším a nejhorším jedincem populace. Rozdíl mezi těmito hodnotami se počítá na konci každého migračního kola, kde také dojde k porovnání s parametrem MinDiv. Pokud je vypočítaná hodnota menší než řídící parametr, dojde k ukončení algoritmu. V rámci této práce byl tento způsob ukončení potlačen, neboť bylo důležité změřit chování všech algoritmů po stejném časovém (výpočtovém) úseku. Proto jako jediný ukončovací parametr slouží $MaxEvaluation$.

Maximální počet evaluací účelové funkce – MaxEvaluation

Tento parametr závisí pouze na uživateli a řídí maximální počet vyhodnocení účelové funkce. Tento parametr nahrazuje původní ukončovací parametr Migrations (popsaný v [1]), který označoval maximální počet migračních cyklů. K záměně došlo z důvodu porovnání tohoto algoritmu s algoritmy DE a SPSO v rámci této práce. Vzhledem k tomu, že ne všechny algoritmy mají stejný počet vyhodnocení během jednoho evolučního/migračního kola, muselo dojít k této změně, aby se při porovnání algoritmů vycházelo ze stejného časového úseku (zde tedy určeného počtem vyhodnocení účelové funkce). Jedná se o ukončovací parametr. Musí splňovat podmínku $MaxEvaluation > 0$. [1]

3.2 Princip algoritmu

3.2.1 Stanovení parametrů

Před započetím provádění algoritmu je nutné stanovit řídící a ukončovací parametry. Jde o parametry, které zajišťují a upravují chod celé evoluce. Tyto parametry jsou popsány v tabulce (*Tab. 3*) a jsou to $PathLength$, $Step$, PRT, $MaxEvaluation$, $PopSize$ – velikost populace a D – rozměr jedince (odvozen z rozměru daného problému). Nutné je také nadefinovat účelovou funkci, která bude optimalizována. Tato funkce by měla vracet

skalár, který bude použit jako měřítko kvality daného jedince (pro hledání maxima bude nejlepší jedinec ten, který bude mít tento skalár největší a pro minimum nejmenší). [1]

3.2.2 Migrační kola

Pro každého jedince je učiněno ohodnocení účelovou funkcí a pro následující migrační kolo je vybrán Leader (Jedinec s nejlepší hodnotou (pro maximum je to největší a pro minimum nejmenší) účelové funkce), ke kterému ostatní jedinci migrují, pomocí skoků, jejichž velikost je určena parametrem $Step$. Po každém skoku si každý jedinec na nově získané pozici přepočítá hodnotu své účelové funkce, a pokud je lepší než předchozí, zapamatuje si ji. Skokový pohyb jedince směrem k Leaderovi pokračuje do té doby, dokud není dosaženo pozice, jež je dána parametrem $PathLength$. Každá nová pozice putujícího jedince je vypočítána z rovnic

$$\vec{r} = \vec{r_0} + \vec{m}\, t\, \overrightarrow{PRTVector}$$
$$kde\ t \in < 0, po\ Step\ až\ po, Mass > \tag{10}$$

kde $\vec{r}, \vec{r_0}$ reprezentují jedince (vektory).

nebo podrobněji

$$X_{i,j}^{ML+1} = X_{i,j,start}^{ML} + (X_{L,j}^{ML} - X_{i,j,start}^{ML})\, t\, PRTVector_j$$
$$kde\ t \in < 0, po\ Step\ až\ po, Mass > \tag{11}$$

kde

$X_{i,j}^{ML+1}$ j-tý prvek i-tého jedince v migračním kole ML+1

$X_{i,j,start}^{ML}$ j-tý prvek i-tého jedince v migračním kole ML ve startovní pozici

$X_{L,j}^{ML}$ j-tý prvek vedoucího jedince (Leadera) v migračním kole ML

Po ukončení běhu se jedinec vrací na jeho nejlepší pozici (tam kde jeho účelová funkce dosahovala nejlepšího výsledku během jeho cesty). To má za následek to, že po ukončení migračního kola se všichni jedinci, kromě vedoucího (Leadera), přemístili.[1]

Předtím než jedinec započne putovat směrem k Leaderovi, je vygenerován prázdný vektor o dimenzi D, včetně sekvence náhodných čísel, jejichž počet je také roven D. Tento vektor se nazývá $PRTVector$**.** Náhodné čísla v $PRTVectoru$ jsou porovnány s parametrem PRT. jestliže je n-té vygenerované číslo větší než parametr PRT, pak je n-tý prvek $PRTVectoru$

nastaven na 0, jinak je nastaven na 1. $PRTVector$ tedy po dosazení do rovnice uvedené výše (11) ovlivňuje směr pohybu jedince a tím má vliv na výsledný pohyb jedince. Díky tomu se jedinec pohybuje v N-k rozměrném podprostoru. Ty prvky $PRTVectoru$, které mají hodnotu 0, se tedy nepřepočítávají – zůstávají na svých aktuálních hodnotách. Toto se děje pro každého jedince a tedy parametry, které jsou „zmraženy“ nyní, nemusí být v tomto stavu v příštích migračních kolech. [1]

3.2.3 Testování naplnění ukončovacích podmínek

Normálně může být celý algoritmus ukončen dvěma metodami. Buď je dosaženo předem určeného počtu migračních kol, nebo je rozdíl mezi nejhorším jedincem a nejlepším jedincem (Leaderem) menší, než parametr $MinDiv$. Pokud není splněna ani jedna z těchto podmínek, vrací se algoritmus zpět k bodu 2. V této práci byl algoritmus upraven na ukončení pouze po dosažení požadovaného počtu evaluací. [1]

3.2.4 Vyhodnocení

Výsledkem celého algoritmu je nejlepší nalezené řešení po posledním migračním kole (ukončené naplněním maximálního počtu vyhodnocení (popř. migrací)). [1]

3.3 Strategie SOMA

V současné době existuje několik různých variací základního algoritmu SOMA. Kvůli zdůraznění faktu kooperace jedinců a geometrického přesouvání celé populace se používá také termín „strategie“ na místo „variace“ nebo „verze“. [1]

AllToOne – Všichni k jednomu

Tato strategie byla popsána v sekci Princip Algoritmu. Všichni jedinci migrují k jednomu Leaderovi (vedoucímu jedinci), kromě jeho samotného. [1]

AllToAll – Všichni ke všem

Zde neexistuje Leader. Všichni jedinci migrují ke všem ostatním, tak jako ve strategii „AllToOne“ s tím rozdílem, že po ukončení migrací aktuálního jedince se daný jedinec vrací na pozici svého nejlepšího výsledku účelové funkce, nalezeného během jeho $PopSize - 1$ migračních cest vykonaných v daném migračním kole. Během tohoto algoritmu se prohledá větší část prostoru možných řešení pro každého jedince, tudíž je výsledek přesnější. To je ovšem vykoupeno jeho větší výpočetní náročností. [1]

AllToAllAdaptive – Adaptivně všichni ke všem

Totožná strategie se strategií „AllToAll“ s tím rozdílem, že aktuální jedinec se nepřesouvá do své nejlepší pozice až po migraci všech ostatních jedinců, ale hned po ukončení každé migrace ke každému z $PopSize - 1$ jedinců. Migraci k dalším jedincům pak provádí z tohoto místa. [1]

AllToOneRand – Všichni k jednomu náhodně

V této strategii se jedinci pohybují pouze k jednomu Leaderovi. Tento jedinec není určen nejlepší hodnotou účelové funkce, ale je po migraci každého jedince náhodně vybrán. [1]

Clusters - Svazky

Tato strategie se dá použít s kteroukoliv z předchozích. V podstatě se jedná o rozdělení všech jedinců do skupin – svazků, ve kterých následně probíhá samostatný algoritmus SOMA. Díky pohybu jedinců je možné, že se svazky budou spojovat nebo rozpadat. Pro každého jedince se testuje, do kterého svazku patří. To se děje podle vztahu

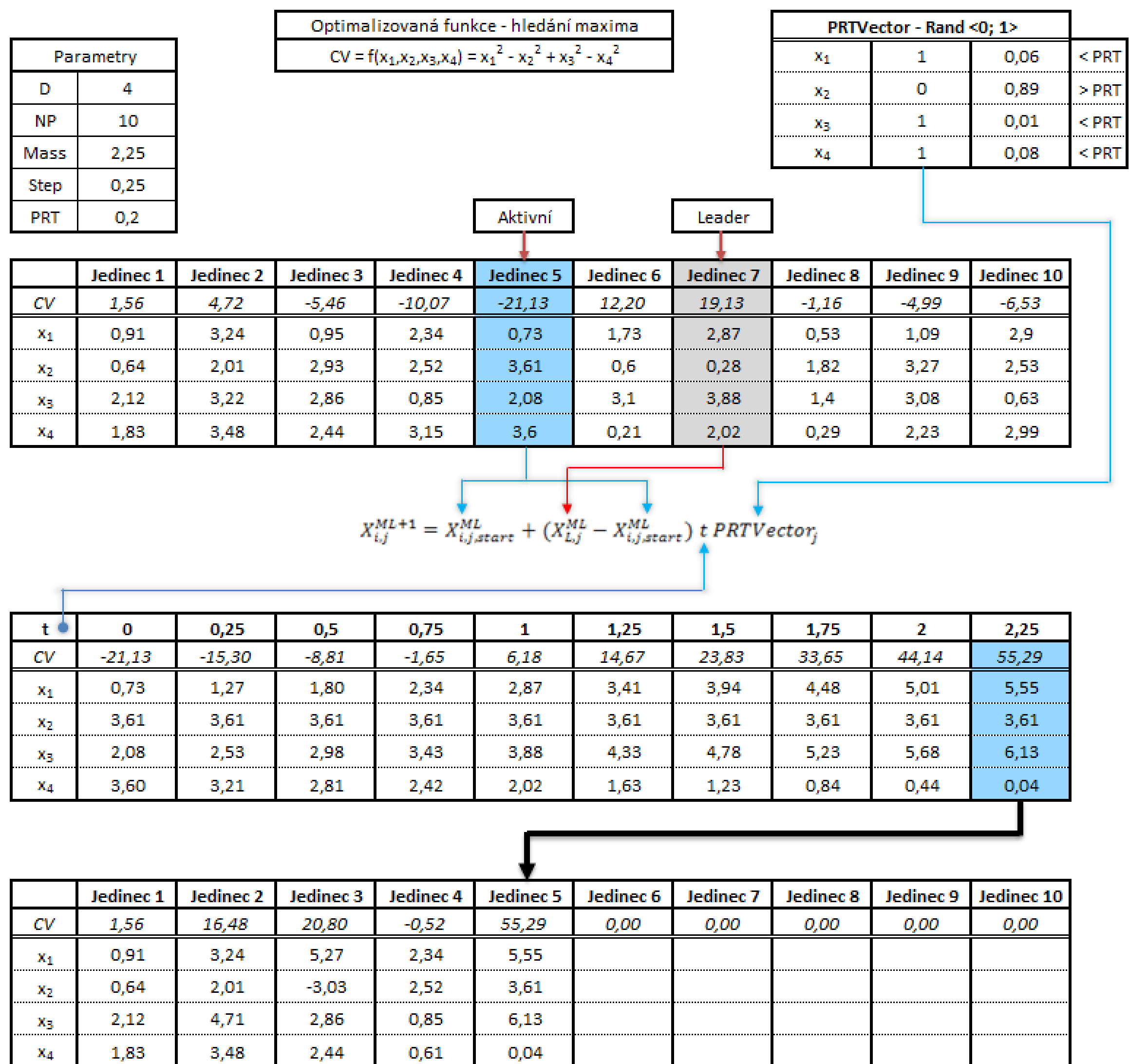
$$CD > \sqrt{\sum_{i=1}^{dim} \left(\frac{IND_i - CC_i}{HB_i - LB_i}\right)^2} \tag{12}$$

Kde parametr IND_i je i-tý parametr jedince, CC_i je i-tý parametr Leadera (centrum svazku), HB_i a LB_i jsou povolené hranice jednotlivých parametrů (viz Specimen). CD je velikost svazku daná uživatelem. Výsledkem je vytvoření svazků obsahující jedince populace. Pokud je nějaký svazek příliš daleko od ostatních a tudíž nemůže být zahrnut do již existujících svazků, pak je svazkem sám sobě a migruje ke všem ostatním jako ve strategii „AllToAll“. [1]

3.4 Algoritmus

1. Nastavení řídících parametrů $PathLength$, $Step$, PRT. Nastavení t na hodnotu větší, než je hodnota $PathLength$. Alokování polí pro populaci (X), nejlepší jedince (P) a nejlepší hodnoty účelové funkce ($BestCostValues$).
2. Naplnění $BestCostValues$ maximální hodnotou ($double.maxValue$) pro každého jedince.
3. Porovnání, zda je počet již provedených evaluací ($NumberOfEvaluations$) větší než maximální povolený ($MaxEvaluation$). Pokud ano, ukončit algoritmus.

4. Ohodnocení jedinců účelovou funkcí.
5. Zvednutí počtu ohodnocení ($NumberOfEvaluations$) o právě provedené.
6. Porovnání hodnot účelových funkcí nejlepších jedinců ($BestCostValues$) s nově ohodnocenýma ($fitnesses$). V případě že jsou horší, tak pokračovat **8**.
7. Překopírovat lepšího jedince do ***P*** a jeho hodnotu do $BestCostValues$.
8. V případě, že je t větší než $PathLength$, pak pokračovat **9**, jinak **12**.
9. Nalezení nové pozice vedoucího jedince ($leader$) pomocí „property“ $BestIndex$.
10. Nakopírovat nejlepší jedince (P) do X.
11. Nastavit t na hodnotu $step$.
12. Pro každého jedince (index i): Kontrola zda se jedná o vedoucího jedince. Pokud ano, pak **16**.
13. Pro každý parametr jedince (index d): Vygeneruje se náhodné číslo z intervalu <0; 1>. Pokud je toto číslo menší než hodnota PRT, tak se do proměnné $PRTvalue$ uloží 1. V opačném případě se uloží 0.
14. Vypočítá se nová pozice jedince pomocí vzorce (11)
15. Zvedne se index parametru (d). Pokud se nejednalo o poslední parametr, pokračuje se **13**.
16. Zvedne se index jedince (i). Pokud se nejednalo o posledního jedince, pokračuje se **12.**
17. Zvedne se hodnota t o hodnotu $Step$. Pokračuje se **3**.



Parametry	
D	4
NP	10
Mass	2,25
Step	0,25
PRT	0,2

Optimalizovaná funkce - hledání maxima
$CV = f(x_1,x_2,x_3,x_4) = x_1^2 - x_2^2 + x_3^2 - x_4^2$

PRTVector - Rand <0; 1>			
x_1	1	0,06	< PRT
x_2	0	0,89	> PRT
x_3	1	0,01	< PRT
x_4	1	0,08	< PRT

	Jedinec 1	Jedinec 2	Jedinec 3	Jedinec 4	Jedinec 5	Jedinec 6	Jedinec 7	Jedinec 8	Jedinec 9	Jedinec 10
CV	*1,56*	*4,72*	*-5,46*	*-10,07*	*-21,13*	*12,20*	*19,13*	*-1,16*	*-4,99*	*-6,53*
x_1	0,91	3,24	0,95	2,34	0,73	1,73	2,87	0,53	1,09	2,9
x_2	0,64	2,01	2,93	2,52	3,61	0,6	0,28	1,82	3,27	2,53
x_3	2,12	3,22	2,86	0,85	2,08	3,1	3,88	1,4	3,08	0,63
x_4	1,83	3,48	2,44	3,15	3,6	0,21	2,02	0,29	2,23	2,99

$$X_{i,j}^{ML+1} = X_{i,j,start}^{ML} + (X_{L,j}^{ML} - X_{i,j,start}^{ML})\ t\ PRTVector_j$$

t	0	0,25	0,5	0,75	1	1,25	1,5	1,75	2	2,25
CV	*-21,13*	*-15,30*	*-8,81*	*-1,65*	*6,18*	*14,67*	*23,83*	*33,65*	*44,14*	*55,29*
x_1	0,73	1,27	1,80	2,34	2,87	3,41	3,94	4,48	5,01	5,55
x_2	3,61	3,61	3,61	3,61	3,61	3,61	3,61	3,61	3,61	3,61
x_3	2,08	2,53	2,98	3,43	3,88	4,33	4,78	5,23	5,68	6,13
x_4	3,60	3,21	2,81	2,42	2,02	1,63	1,23	0,84	0,44	0,04

	Jedinec 1	Jedinec 2	Jedinec 3	Jedinec 4	Jedinec 5	Jedinec 6	Jedinec 7	Jedinec 8	Jedinec 9	Jedinec 10
CV	*1,56*	*16,48*	*20,80*	*-0,52*	*55,29*	*0,00*	*0,00*	*0,00*	*0,00*	*0,00*
x_1	0,91	3,24	5,27	2,34	5,55					
x_2	0,64	2,01	-3,03	2,52	3,61					
x_3	2,12	4,71	2,86	0,85	6,13					
x_4	1,83	3,48	2,44	0,61	0,04					

Obr. 2. Ukázka provedení algoritmu SOMA

4 DIFERENCIÁLNÍ EVOLUCE (DIFFERENTIAL EVOLUTION)

Historie diferenciální evoluce se datuje do roku 1995, kdy ji vyvinuli a poprvé použili pánové Ken Price a Rainer Storn. Diferenciální evoluce je podobná algoritmům genetickým, s nimiž sdílí společné prvky, jako například tvorbu potomků (na rozdíl od genetických algoritmů, kde se používají 2 rodiče, zde má každý jedinec rodiče 4), nebo využití generací.[1][2]

Vychází z genetického žíhání (autor Ken Price – 1994), jehož binární reprezentaci převádí na dekadické a úměrně tomu logické operace na vektorové. Tyto změny udělaly z genetického žíhání algoritmus vhodný pro numerickou optimalizaci. Po těchto změnách byla K. Pricem objevena tzv. diferenciální mutace, která spočívá v generování tzv. zkušebního řešení přičtením diference dvou náhodně vybraných vektorů (jedinců) ke třetímu jedinci z populace. Po zkombinování diferenciální mutace s metodou selekce z genetického žíhání vzniklo to, co se dnes označuje jako první verze diferenciální evoluce.[1]

Ukázalo se, že principy žíhání aplikované v genetickém algoritmu jsou nadbytečné a tak byly z diferenciální evoluce vypuštěny. Zároveň R. Storn navrhl vytváření populace potomků ve zvláštní populaci, jenž se účastní souboje o místo v nové populaci až po naplnění této zvláštní populace. Tato verze sice velmi dobře fungovala na testovacích funkcích, ale ukázala se jako nedostatečná pro řešení široké množiny optimalizačních problémů a proto byla vyvinuta třetí verze tohoto algoritmu.[1]

4.1 Parametry a terminologie

Činnost a kvalita diferenciální evoluce je vysoce ovlivněna řídícími parametry. Proto je nutné je správně nastavit. Jejich označení a význam je:

Tab. 4. Přehled parametrů algoritmu DE. Převzato z [1]

Řídící parametr	Interval	Optimum?	Poznámka
PopSize	<2*D; 100*D?>	10*D	Velikost populace - 100*D pokud je funkce multimodální
D	určeno problémem		Dimenze problému
MaxEvaluation	uživatel		Maximální počet vyhodnocení účelové funkce
F	<0; 2>	0.3 - 0.9	Mutační kostanta
CR	<0; 1>	0.8 - 0.9	Práh křížení

Dimenze problému – D

Udává počet parametrů účelové funkce. Velikost je závislá na definovaném problému, popsaném účelovou funkcí. Změnu lze provést pouze reformulací celého problému. [1][2]

Velikost populace – $PopSize$

Tento parametr určuje počet jedinců v populaci. Velikost tohoto parametru by neměla být menší než 4. To je minimální velikost populace, při které diferenciální evoluce ještě pracuje. [1]

Práh křížení – CR

V případě, že je tento parametr nastaven na 0, pak se mutace nedostane do zkušebního jedince, který díky tomu bude čistou kopii aktuálního rodiče. Naopak pokud bude CR nastaven na 1, pak budou všechny parametry zkušebního jedince tvořeny pouze ze třech náhodných rodičů (aktuální jedinec bude vynechán). Proto je vhodné, aby CR nikdy nenabývalo těchto hodnot. Doporučená hodnota by se měla v případě reparabilních úloh blížit 0, v opačném případě 1. [1]

Mutační konstanta – F

Touto konstantou se násobí diferenční vektor, čímž vznikne diferenční váhový vektor (viz Princip algoritmu). Tato konstanta by se měla pohybovat v intervalu <0; 2> [1]

Maximální počet evaluací účelové funkce – $MaxEvaluation$

Tento parametr závisí pouze na uživateli a řídí maximální počet vyhodnocení účelové funkce. Tento parametr nahrazuje původní ukončovací parametr $Generations$ (popsaný v [1]), který označoval maximální počet evolučních cyklů. K záměně došlo z důvodu porovnání tohoto algoritmu s algoritmy SOMA a SPSO v rámci této práce. Vzhledem

k tomu, že ne všechny algoritmy mají stejný počet vyhodnocení během jednoho evolučního kola, muselo dojít k této změně, aby se při porovnání algoritmů vycházelo ze stejného časového úseku (zde tedy určeného počtem vyhodnocení účelové funkce). Jedná se o ukončovací parametr. Musí splňovat podmínku $MaxEvaluation > 0$. [1]

4.2 Princip algoritmu

4.2.1 Stanovení parametrů

Před započetím provádění algoritmu je nutné stanovit řídící parametry. Jde o parametry, které zajišťují a upravují chod celé evoluce. Tyto parametry jsou popsány v tabulce uvedené výše (*Tab. 4*). Jsou to ***F*** – mutační konstanta, ***CR*** – práh křížení, ***PopSize*** – velikost populace a ***D*** – rozměr jedince (odvozen z rozměru daného problému).

Dále je nutné nadefinovat prototyp jedince – tj. v jakém rozmezí a z jakých typů (celočíselné – Integer, s desetinou čárkou – real) se budou daní jedinci generovat. (Viz Specimen)

4.2.2 Započetí cyklu generace

Cyklus se provádí během každé generace a zabezpečuje postupné evoluční šlechtění každého jedince z populace. V každém cyklu se postupně vybírá jeden jedinec (aktivní jedinec, cílový vektor) za druhým a pro každého z nich je proveden daný evoluční cyklus. [1]

4.2.3 Evoluční cyklus

Zde je prováděna mutace a křížení. Z celé populace jedinců se náhodně vyberou 3 další různí jedinci (vektory). První dva se od sebe odečtou. Tím se získá tzv. „diferenční vektor“. Ten je vynásoben mutační konstantou F, která jej změní (zmutuje). Výsledný vektor se nazývá „váhovaný diferenční vektor“. Ten je následně přičten ke třetímu vybranému vektoru (jedinci) a tím vznikne „šumový vektor“.

Dále se spojením šumového vektoru s cílovým (aktivní jedinec) vytvoří „zkušební vektor“. Toto spojení probíhá tak, že se náhodně vybere parametr jedince, který se automaticky zapíše do „zkušebního vektoru“ a následně se z každého cílového a šumového vektoru postupně bere jeden prvek za druhým (ve směru od vygenerovaného náhodného, v případě že se dosáhne posledního parametru, pokračuje se od prvního až do doby, než se dosáhne

toho parametru, který byl náhodně vybrán) a pro každou tuto dvojici se vygeneruje náhodné číslo v rozsahu <0; 1>. Pokud je toto číslo menší než řídící konstanta CR – práh křížení, použije se do příslušné pozice „zkušebního vektoru“ prvek z vektoru šumového, v opačném případě se použije prvek z vektoru cílového. Po získání celého „zkušebního vektoru“ se porovná účelová hodnota funkce tohoto vektoru s účelovou hodnotou funkce cílového vektoru. Na pozici cílového vektoru v nové populaci je pak vybrán takový jedinec, pro kterého je účelová hodnota lepší (pro maximum je to větší z hodnot, pro minimum menší). Tím je zajištěno, že se do nové populace dostanou pouze jedinci s lepšími vlastnostmi.

Poté se vybere další jedinec – cílový vektor a celý cyklus se opakuje až do vyčerpání populace. Tak vznikne nová generace potomků (jedinců). [1][9]

4.2.4 Testování naplnění ukončovacích podmínek

Diferenciální evoluce je ukončena pouze tehdy, provede-li se uživatelem zadaný počet Vyhodnocení účelové funkce (ve standardním diferenciální evoluce by se ukončovala počtem provedených evolučních kol). Jiný ukončovací parametr tento algoritmus nemá.

Pro neupravený algoritmus by bylo možné při vyhodnocování účelové funkce přidat algoritmus, který evoluci ukončí, pokud se například při deseti posledních generacích nejlepší hodnota účelové funkce nezlepšila. [1]

4.2.5 Vyhodnocení

Celý proces se opakuje, dokud není vyčerpán zadaný počet generací (popřípadě evaluací). Během každé generace se hodnota účelové funkce nejlepšího jedince uschová do vektoru historie, který po ukončení znázorňuje průběh evolučního cyklu.

4.3 VARIANTY DE

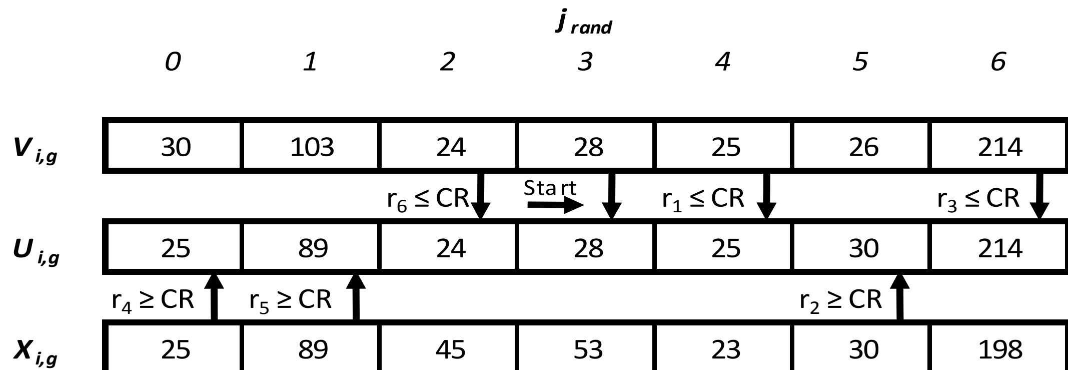
Existuje několik variant diferenciální evoluce, které se liší především ve výpočtu „šumového vektoru“.

4.3.1 Varianty křížení

Binomiální

Náhodně je vybrán jeden prvek j_{rand} z „šumového vektoru“ $V_{i,g}$, který se přesune do vektoru zkušebního $U_{i,g}$ a dále pak pro každou dvojici parametrů „šumového vektoru“ a

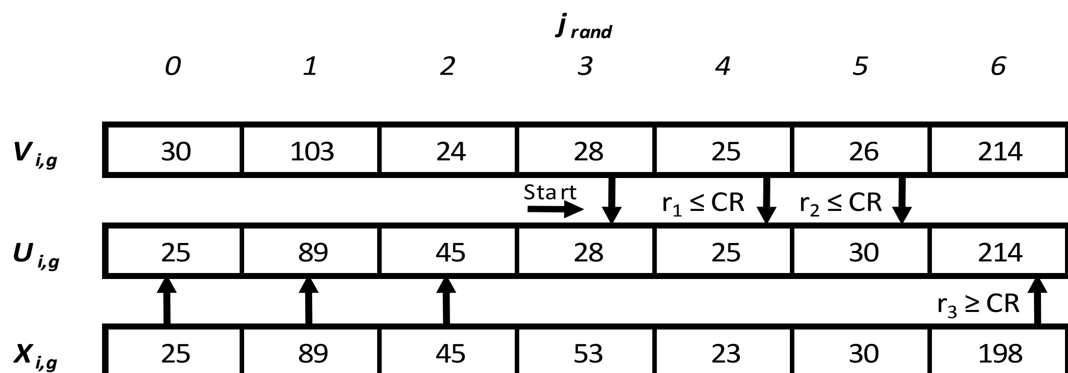
cílového (aktivního) jedince $X_{i,g}$ se generuje náhodné číslo R_j z intervalu <0; 1>. Pokud je toto číslo větší než CR, do zkušebního vektoru se na příslušnou pozici vloží prvek cílového (aktivního) jedince, v opačném případě se použije parametr ze „šumového vektoru“**.** [1][2]



Obr. 3. Ukázka binomiálního křížení.

Exponenciální

Náhodně se zvolí prvek j_{rand} z „šumového vektoru“ $V_{i,g}$, jenž se přesune na odpovídající pozici zkušebního vektoru $U_{i,g}$. Dále se pro každý další prvek (v kruhovém smyslu) generuje náhodné číslo R_j z intervalu <0; 1>. Pokud je toto číslo menší než CR, pak se použije parametr ze „šumového vektoru“. Jakmile je však číslo větší, pro všechny následující volné pozice se volí prvky cílového (aktivního) jedince. [1][2]



Obr. 4. Ukázka exponenciálního křížení

4.3.2 Varianty DE

Značení jednotlivých variant je ve tvaru: DE/x/y/z. DE zde označuje použití diferenciální evoluce, x potom reprezentuje metodu výběru rodiče, který bude použit jako základní vektor. Pro DE/rand/y/z to tedy bude základní vektor náhodně z celé populace, DE/best/y/z

bude nejlepší vektor. y udává počet různých vektorů, jenž potrubují základní vektor, z je typ použitého křížení (exp – exponenciální, bin – binomiální). [1][2]

DE/best/1/exp

$$v_j = x_{best,j}^G + F \cdot (x_{r2,j}^G - x_{r3,j}^G) \tag{13}$$

DE/rand/1/exp

$$v_j = x_{r1,j}^G + F \cdot (x_{r2,j}^G - x_{r3,j}^G) \tag{14}$$

DE/rand-to-best/1/exp

$$v_j = x_{i,j}^G + F \cdot \left(x_{best,j}^G - x_{i,j}^G\right) + F \cdot (x_{r1,j}^G - x_{r2,j}^G) \tag{15}$$

DE/best/2/exp

$$v_j = x_{best,j}^G + F \cdot \left(x_{r1,j}^G + x_{r2,j}^G - x_{r3,j}^G - x_{r4,j}^G\right) \tag{16}$$

DE/rand/2/exp

$$v_j = x_{r5,j}^G + F \cdot \left(x_{r1,j}^G + x_{r2,j}^G - x_{r3,j}^G - x_{r4,j}^G\right) \tag{17}$$

DE/best/1/bin

$$v_j = x_{best,j}^G + F \cdot \left(x_{r2,j}^G - x_{r3,j}^G\right) \tag{18}$$

DE/rand/1/bin

$$v_j = x_{r1,j}^G + F \cdot \left(x_{r2,j}^G - x_{r3,j}^G\right) \tag{19}$$

DE/rand-to-best/1/bin

$$v_j = x_{i,j}^G + \lambda \cdot \left(x_{best,j}^G - x_{i,j}^G\right) + F \cdot (x_{r1,j}^G - x_{r2,j}^G) \tag{20}$$

DE/best/2/bin

$$v_j = x_{best,j}^G + F \cdot \left(x_{r1,j}^G + x_{r2,j}^G - x_{r3,j}^G - x_{r4,j}^G\right) \tag{21}$$

DE/rand/2/bin

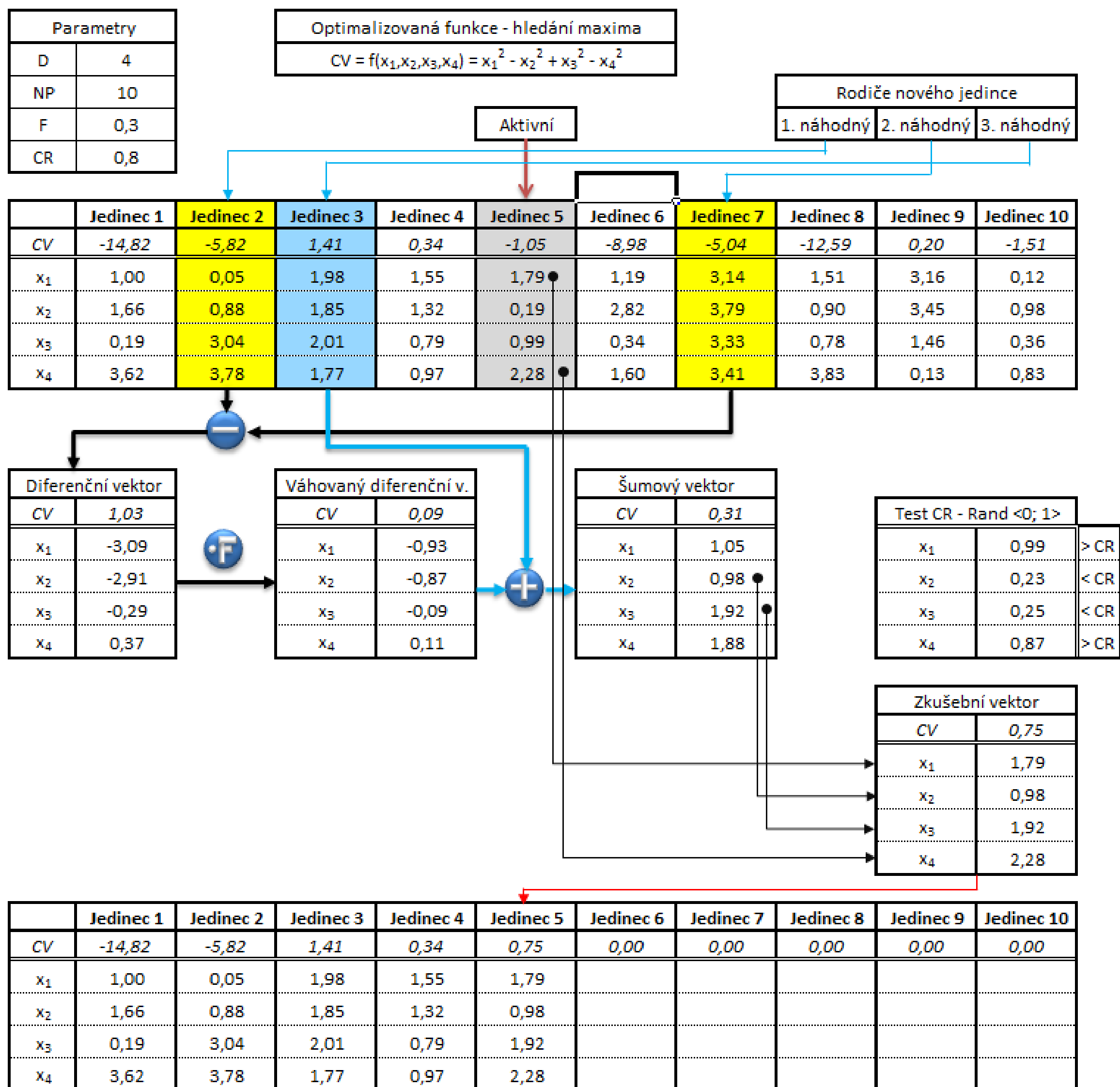
$$v_j = x_{r5,j}^G + F \cdot \left(x_{r1,j}^G + x_{r2,j}^G - x_{r3,j}^G - x_{r4,j}^G\right) \tag{22}$$

Některé z uvedených metod se na první pohled zdají být stejné (např. DE/best/1/exp a DE/best/1/bin). Tyto metody se liší v dalším provádění algoritmu diferenciální evoluce. [1]

4.4 Algoritmus

Použita varianta algoritmu diferenciální evoluce označená jako **DE/rand/1/bin** popsaná v [9].

1. Nastavení řídících parametrů CR, F. Alokování polí pro populaci ($trial$), nejlepší jedince ($x1$) a nejlepší hodnoty účelové funkce ($cost$).
2. Naplnění $cost$ maximální hodnotou ($double.maxValue$) pro každého jedince.
3. Porovnání, zda je počet již provedených evaluací ($NumberOfEvaluations$) větší než maximální povolený ($MaxEvaluation$). Pokud ano, ukončit algoritmus.
4. Ohodnocení jedinců účelovou funkcí.
5. Zvednutí počtu ohodnocení ($NumberOfEvaluations$) o právě provedené.
6. Porovnání hodnot účelových funkcí nejlepších jedinců ($cost$) s nově ohodnocenýma ($trialScores$). V případě že jsou horší, tak pokračovat **8**.
7. Překopírovat lepšího jedince do $x1$ a jeho hodnotu do $cost$.
8. Pro každého jedince (index i): Nalezení 3 náhodných různých jedinců.
9. Nalezení náhodné startovací pozice pro procházení parametrů jedince.
10. Pro každý parametr (index j): Vygenerování náhodného čísla z intervalu <0; 1> a porovnání tohoto s hodnotou PRT. Pokud je menší (popřípadě pokud se jedná o poslední parametr jedince), pak pokračovat **11**, jinak **12**.
11. Vypočítání nové pozice jedince pomocí vzorce (19) a výsledek uložit do $trial$. Pokračování **13**.
12. Do $trial$ uložit parametr z matice nejlepších jedinců ($x1$). Pokračování **13**.
13. Zvednout index parametru (j). Pokud se nejedná o poslední parametr, pak pokračovat na **10**.
14. Zvednou index jedince (i). Pokud se nejedná o posledního jedince, pak pokračovat **8**, jinak pokračovat **3**.



Obr. 5. Ukázka provedení algoritmu DE

5 FRAMEWORK .NET

Framework .NET je vyvíjen společností Microsoft pro operační systémy Windows (pro kompatibilitu s ostatními operačními systémy je nutné použít nějakou open source alternativu, např. Mono). Tento framework poskytuje podporu tzv. jazykovou interoperabilitu (každý programovací jazyk může využít část kódu napsanou v jiném jazyce). To umožňuje softwarové prostředí, nazvané CLR (Common Language Runtime), které se chová jako virtuální stroj aplikace a ve kterém jsou programy tohoto frameworku vykonávány a které poskytuje programu služby jako, ochranu, správu paměti, správu výjimek, apod. Rozhraní .NET obsahuje, kromě tohoto prostředí CLR, ještě knihovnu tříd .NET. [2][10][11]

Knihovna tříd .NET je objektově orientovaná knihovna, která poskytuje nástroje pro vývoj široké škály aplikací, jako např.:

- Konzolové aplikace
- Aplikace GUI (Graphical User Interface – grafické uživatelské rozhraní) systému Windows – Windows Forms
- Aplikace typu WPF – Windows Presentation Foundation
- Aplikace ASP.NET
- a další

Windows Forms nabízí mnoho ovládacích prvků a dialogových oken pro jednodušší práci s GUI systému Windows. Windows Presentation Foundation oproti tomu poskytuje velkou volnost v úpravě grafického vzhledu aplikace pomocí programovacího jazyku XAML.[2][10]

Aplikace rozhraní .NET se kvůli kompatibilitě a možnosti interoperability překládají do zprostředkujícího kódu (jazyku MSIL). [2][11]

PRAKTICKÁ ČÁST

6 OPTIMALIZACE FUNKCE

Účelová i aproximační funkce byli předem zadané.

6.1 ÚČELOVÁ FUNKCE

Jako účelová funkce byl zvolen Root Mean Squared Error (RMSE), což je odmocnina střední čtvercové chyby (Mean Squared Error – MSE). Hledáme parametry funkce $f_{Q_{load}}(t_i)$, které minimalizují odmocninu střední čtvercové chyby vůči určené naměřené hodnotě $Q_{load}(t_i)$:

$$MSE_{Q_{load}} = \frac{\sum_{i=1}^{n} \left(f_{Q_{load}}(t_i) - Q_{load}(t_i) \right)^2}{b}$$
$$RMSE_{Q_{load}} = \sqrt{MSE_{Q_{load}}} \qquad (23)$$

kde n je počet vzorků a $i = 1,2, \dots, n$.

6.2 APROXIMAČNÍ FUNKCE

$$f_{Q_{load}}(h, \vartheta_{ex}) = A + \frac{K - A}{(1 + Qe^{-B(-\vartheta_{ex} - M)})^{\frac{1}{v}}} + \sum_{h=1}^{24} p_h + \sum_{h=1}^{24} q_h \qquad (24)$$

Pro aproximaci složky výkonu závislého na venkovní teplotě vzduchu je využita Richardsova křivka, což je růstová křivka, která je vhodná pro modelování fyzikálního procesu i když tento proces není znám [12]. Vzhledem k tomu, že tepelná zátěž roste s poklesem venkovní teploty vzduchu, tak je ve vztahu (24) použitá záporná hodnota $-\vartheta_{ex}$.

Druhou komponentou predikce tepelného zatížení je komponenta závislosti na hodině ve dni. V tomto případě je pro každý typ dne určeno 24 hodinových hodnot tepelného zatížení p_h a q_h, kde h je hodina ve dni $h = 1,2, \dots, 24$.

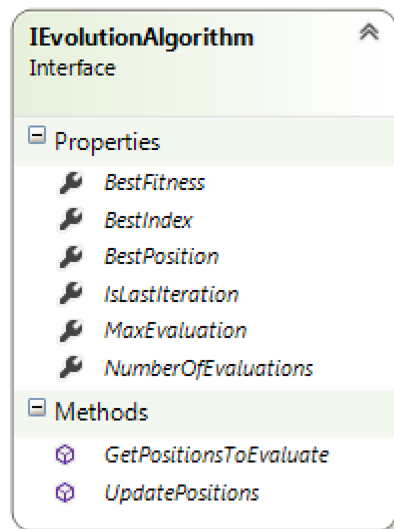
Meze pro generování vzoru (Specimen) jsou poté nastaveny následovně

Tab. 5. Parametry a meze aproximační funkce

Parametr	Dolní mez	Horní mez
A	-40000	40000
K	40000	200000
Q	0,04	0,5
B	0	0,5
M	-15	15
v	0	2
H	0	80000

7 INTERFACE IEVOLUTIONALGORITHM

Během programování bylo důležité, aby bylo možné rychle a jednoduše volit mezi použitými algoritmy SOMA, DE a PSO. Proto byl použit interface *IEvolutionAlgorithm*, který všechny algoritmy využívají (dědí z něho). Díky použití tohoto interface došlo k oddělení části evolučních algoritmů od části výpočtu hodnoty účelové funkce. Tento výpočet tak probíhá mimo daný algoritmus a tak hodnoty vyhrazené algoritmem pro vyhodnocení musí být posílány pomocí metody *GetPositionToEvaluate()*.



Obr. 6. Přehled IEvolutionAlgorithm

Na následujících řádcích popíši jednotlivé metody tohoto interface:

7.1 Vlastnosti (Properties)

int BestIndex { get; }

Vrací index nejlepšího jedince. Typ Integer.

double BestFitness { get; }

Vrací nejlepší hodnotu účelové funkce.

double[] BestPosition { get; }

Vrací vektor (jednorozměrné pole typu double) parametrů nejlepšího jedince.

int NumberOfEvaluations{ get;}

Vrací počet dosud provedených vyhodnocení účelové funkce.

int MaxEvaluation { get;}

Vrací maximální počet evaluací před ukončením evolučního algoritmu.

bool IsLastIteration { get; }

Kontroluje, zda se jedná o poslední iteraci. V tom případě vrátí hodnotu True. Jinak vrací False.

7.2 Metody (Methods)

double[][] GetPositionsToEvaluate();

V této metodě vrací daná třída data (pole jedinců), u kterých mají být vyhodnoceny hodnoty účelových funkcí. Tyto data musí být ve tvaru dvourozměrného pole typu double (s plovoucí desetinou čárkou).

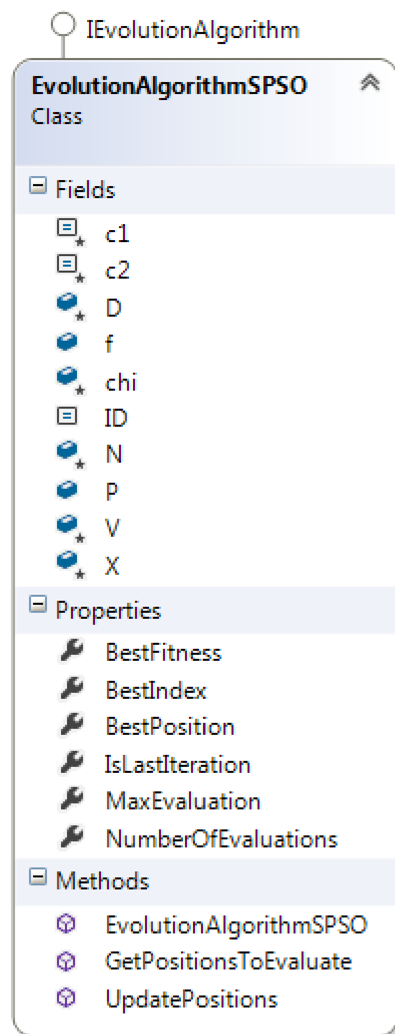
void UpdatePositions(double[] fitnesses);

Tato metoda slouží k provedení evolučního algoritmu. Přijímá jednorozměrné pole typu double, ve kterém jsou uloženy hodnoty účelové funkce všech jedinců.

```
1.    public interface IEvolutionAlgorithm
2.    {
3.        int BestIndex { get; }
4.        double BestFitness { get; }
5.        double[] BestPosition { get; }
6.        int NumberOfEvaluations { get;}
7.        int MaxEvaluation { get;}
8.        bool IsLastIteration { get; }
9.        double[][] GetPositionsToEvaluate();
10.       void UpdatePositions(double[] fitnesses);
11.   }
```

8 OPTIMALIZACE ROJENÍ ČÁSTIC (PARTICLE SWARM OPTIMIZATION)

Pro provedení algoritmu jsem zvolil verzi z roku 2007 popsanou v [6]. Samotný algoritmus je popsán v teoretické části této práce (2.4). Algoritmus využívá interface *IEvolutionAlgorithm*, ze kterého dědí vlastnosti *BestFitness, BestIndex, BestPosition, IsLastIteration, MaxEvaulation, NumberOfEvaulations* a metody *GetPositionsToEvaulate* a *UpdatePositions*. Podrobnější popis tohoto interface se nachází v části (7). Při popisu algoritmu budu jedince označovat jako částice. (viz Parametry a terminologie).



Obr. 7. Přehled EvolutionAlgorithmSPSO

8.1 Členské proměnné (Fields)

double c1, c2

Řídící parametry. Nastaveny na hodnotu 2,05, jak je uvedeno v [6].

int D

Dimenze problému – Tento parametr uživatel nenastavuje, je určen přímo problémem. Hodnota se získá z matice populace X.

double[] f

Vektor nejlepších hodnot účelové funkce.

double chi

Proměnná χ, neboli „chi" označuje faktor zúžení. V této proměnné je uložená výsledná hodnota paramertu χ, vypočítaná pomocí rovnice (2) za předpokladu, že je (3) splněno.

const string ID

Identifikační název třídy. Pro tuto třídu je tento string nastaven na „SPSO" (Standard for Particle Swarm Optimization)

int PopSize

Velikost populace. Tento parametr je určen velikostí populace předané konstruktoru v proměnné X.

double[][] P

Matice nejlepších parametrů částic. Rozměr matice je určen vztahem $PopSize \times D$.

double[][] V

Matice rychlosti částic. Rozměr matice je určen vztahem $PopSize \times D$.

double[][] X

Matice parametrů částic v populaci. Rozměr matice je určen vztahem $PopSize \times D$.

```
1.  public class EvolutionAlgorithmSPSO : IEvolutionAlgorithm
2.  {
3.      public const string ID = "SPSO";
4.
5.      protected int D;
6.      protected int PopSize;
7.      protected const double c1 = 2.05;
8.      protected const double c2 = 2.05;
9.      protected double chi;
10.
11.     protected double[][] X = null;
12.     public double[][] P = null;
13.     protected double[][] V = null;
14.     public double[] f = null;
```

8.2 Vlastnosti (Properties)

double BestFitness

Navrací nejlepší hodnotu (zde nejmenší) z vektoru nejlepších hodnot cílové funkce (**f**).

```
15. public double BestFitness
16. {
17.     get
18.     {
19.         double min = f.Min();
20.         return min;
21.     }
22. }
```

int BestIndex

Vrací index nejlepší hodnoty z vektoru nejlepších hodnot účelové funkce (**f**).

```
23. public int BestIndex
24. {
25.     get
26.     {
27.         int bestIndex = Array.IndexOf(f, f.Min());
28.         return bestIndex;
29.     }
30. }
```

double[] BestPosition

Vrací vektor nejlepšího částice. K jeho nalezení se používá BestIndex (viz výše).

```
31. public double[] BestPosition
32. {
33.     get
34.     {
35.         var values = P[BestIndex];
36.         return values;
37.     }
38. }
```

bool IsLastIteration

Ukončovací kritérium. Kontroluje, zda počet vyhodnocení účelové funkce již přesáhl maximální možný. Ten je nastavený uživatelem při volání konstruktoru. V případě, že maximum překročeno není, tak vrací hodnotu *false*.

```
39. public bool IsLastIteration
40. {
41.     get
42.     {
43.         if (NumberOfEvaluations < MaxEvaluation)
44.            return false;
45.
46.         return true;
47.     }
48. }
```

MaxEvaulation

Zde je uložen uživatelem nastavený maximální počet vyhodnocení účelovou funkcí.

NumberOfEvaulations

Počítá, kolik vyhodnocení účelové funkce již proběhlo.

```
49. public int NumberOfEvaluations { get; protected set;}
50. public int MaxEvaluation { get;  protected set;}
```

8.3 Metody (Methods)

8.3.1 Konstruktor EvolutionAlgorithmSPSO

EvolutionAlgorithmSPSO (double[][] X, int maxEvaluations = 30000)

Konstruktor třídy EvolutionAlgorithmSPSO. Slouží k inicializaci všech proměnných algoritmu, k nastavení řídících parametrů a alokování paměti pro používaná data. Konstruktory jsou zvláštní typy metod, které jsou volané pouze při vytváření nové instance třídy. Tento konstruktor přijímá dva parametry.

double[][] X – zde je uložená populace, se kterou se pracuje. Jedná se o dvourozměrné pole typu double ([index částice][index parametru]).

int maxEvaluations – tento parametr určuje maximální počet evaluací, které se provedou před ukončením algoritmu. Pokud ho uživatel nezavolá, tak je nastaven na výchozí hodnotu 30000 evaluací.

```
1.  public EvolutionAlgorithmSPSO(double[][] X, int
    maxEvaluations = 30000)
2.  {
```

Jako první věc, která se provede po zavolání konstruktoru, je otestování pole populace ***X***. V případě, že je toto pole prázdné, je „vyhozena“ výjimka. Následně je vypočítána „lambda“ (φ) podle rovnice (3) a „chi“ (2).

Následně je z pole částic ***X***, které bylo předáno konstruktorem, určena velikost populace (pomocí *X.length* – vrátí velikost pole) a dimenze problému (*X[0].length*). Tyto hodnoty jsou uloženy do proměnných ***PopSize*** a ***D***. Maximální počet evaluací je z proměnné předané konstruktoru (*maxEvaluations*) uložen do členské proměnné (this.*MaxEvaluation*) a počet již provedených vyhodnocení v proměnné ***NumberOfEvaluations*** je vynulován.

```
3.  if ((X == null) || (X.Length == 0))
4.     throw new Exception("min.Count() != max.Count()");
5.
6.  double lambda = c1 + c2; // 4.1
7.  this.chi = 2 / (Math.Abs(2 - lambda - Math.Sqrt(lambda *
    lambda - 4*lambda)));
8.
9.  this.PopSize = X.Length;
10. this.D = X[0].Length;
11.
12. this.MaxEvaluation = maxEvaluations;
13.
14. this.NumberOfEvaluations = 0;
```

Do členské proměnné *this.**X*** je překopírována celá populace z parametru konstruktoru ***X.*** Po té dochází k alokaci paměti pro pole rychlostí částic (***V***), pole nejlepších hodnot částic (***P***) a pole nejlepších hodnot účelové funkce (***f***). Pole nejlepších hodnot účelové funkce je jednorozměrné a je alokováno na velikost populace ***PopSize***. Matice rychlostí částic a nejlepších hodnot částic jsou dvourozměrné a jejich první rozměr je alokován na velikost populace ***PopSize***.

Ve *for* cyklu poté dochází k naplnění každého prvku vektoru nejlepších hodnot účelové funkce (***f***) na maximální hodnotu proměnné typu double (hledáme minimum, tedy bude zajištěno, že jakákoliv hodnota bude lepší) a zároveň dochází k alokaci druhých rozměrů polí pro rychlost částic a pro nejlepší parametry částic na velikost dimenze problému (***D***) , až do zpracování všech částic (velikost dána členskou proměnnou ***PopSize***). Pro každého částice v cyklu ještě dochází k naplnění všech parametrů rychlosti částic nulovou výchozí hodnotou a k překopírování částice z populačního pole (***X***) do pole nejlepších částic (***P***).

```
15.    this.X = X;
16.    this.P = new double[this.PopSize][];
17.    this.V = new double[this.PopSize][];
18.    this.f = new double[this.PopSize];
19.
20.    // i - index jedince
21.    for (int i = 0; i < this.PopSize; ++i)
22.    {
23.       P[i] = new double[this.D];
24.       V[i] = new double[this.D];
25.       f[i] = double.MaxValue;
26.
27.       // j - index dimenze (parametru)
28.       for (int d = 0; d < this.D; ++d)
29.       {
30.          P[i][d] = X[i][d];
31.          V[i][d] = 0; // nulová výchozí rychlost
32.       }
33.    }
34. }
```

8.3.2 GetPositionToEvaluate

double[][] GetPositionToEvaluate()

V této metodě se vrací pole všech částic (***X***). Tato metoda slouží k navrácení hodnot, které se mají ohodnotit účelovou funkcí.

```
1.  public double[][] GetPositionsToEvaluate()
2.  {
3.     return X;
4.  }
```

8.3.3 Provedení algoritmu – UpdatePositions

void UpdatePositions(double[] nf)

Tato metoda slouží k provedení celého evolučního algoritmu. Její parametr je vektor účelových hodnot částic, které byli poslání na vyhodnocení metodou ***GetPositionToEvaluate()***. Tento vektor se nazývá ***nf*** (new fitnesses).

Prvním krokem algoritmu je navýšení počtu provedených vyhodnocení v členské proměnné ***NumberOfEvaluations*** o právě provedené. Po té dojde k inicializaci náhodných čísel. Následuje aktualizace částic. U každé částice dojde k porovnání nových hodnot funkce (uložených v ***nf***) se starýma (uložené ve vektoru ***f***). Pokud jsou nové hodnoty lepší

(v tomto případě menší) dojde k přepsání staré hodnoty účelové funkce ve vektoru ***f*** hodnotou novou. Zároveň dojde k nakopírování daného jedince do pole nejlepších jedinců ***P***.

```
1.  public void UpdatePositions(double[] nf)
2.  {
3.      this.NumberOfEvaluations += nf.Count();
4.
5.      Random rnd = new Random();
6.
7.      // projdu všechny částice
8.      for (int i = 0; i < this.PopSize; i++)
9.      {
10.         // jestliže je Cost Function (CF) je lepší než
            předchozí
11.         if (nf[i] < f[i])
12.         {
13.             // přepíšu nejlepší parametry částice
14.             f[i] = nf[i];
15.             X[i].CopyTo(P[i], 0);
16.         }
17.     }
```

Po této aktualizace dojde k samotnému provedení PSO. V cyklu se pro každého jedince (částici) z populace nejdříve vypočítá pozice předcházející a následující částice. V případě že se jedná o poslední částici v populaci, vybere se jako následující první částice. V případě, že je aktivní první částice, tak jako předešlá je vybraná ta poslední. Po té se porovnají hodnoty účelových funkcí aktivního jedince s předešlým a následujícím. Vybere se ta nejlepší (nejmenší) hodnota, která je následně použita jako nejlepší částice ze svazku. Index této částice je uložen do proměnné ***g***.

```
10.    for (int i = 0; i < this.PopSize; ++i)
11.    {
12.       int iB = i - 1; // Předchozí (Backward)
13.       int iF = i + 1; // Následující (Forward)
14.
15.       // když je první, tak předchozí je poslední
16.       if (i == 0)
17.       {
18.          iB = PopSize - 1;
19.       }
20.       // kdy je poslední, tak následující je první
21.       else if (i == (PopSize - 1))
22.       {
23.          iF = 0;
24.       }
25.
26.       int g = i;


27.       if (f[iB] < f[i])
28.       {
29.          g = iB;
30.       }
31.       else if (f[iF] < f[g])
32.       {
33.          g = iF;
34.       }
```

Pro každý parametr částic se následně provede výpočet rychlosti podle vzorce (5) a nové pozice podle (6). Tímto se získají nové pozice všech částic, které se uloží do matice ***X***, ze které jsou pomocí metody ***GetPositionToEvaluate()*** odeslány na vyhodnocení.

```
35.       // pro všechny dimenze (parametry)
36.       for (int d = 0; d < this.D; d++)
37.       {
38.          double r1 = rnd.NextDouble();
39.          double r2 = rnd.NextDouble();

40.          double p1 = c1 * r1 * (P[i][d] - X[i][d]);
41.          double p2 = c2 * r2 * (P[g][d] - X[i][d]);
42.          V[i][d] = chi * (V[i][d] + p1 + p2);
43.          X[i][d] = X[i][d] + V[i][d];
44.       }
45.    }
46. }
```

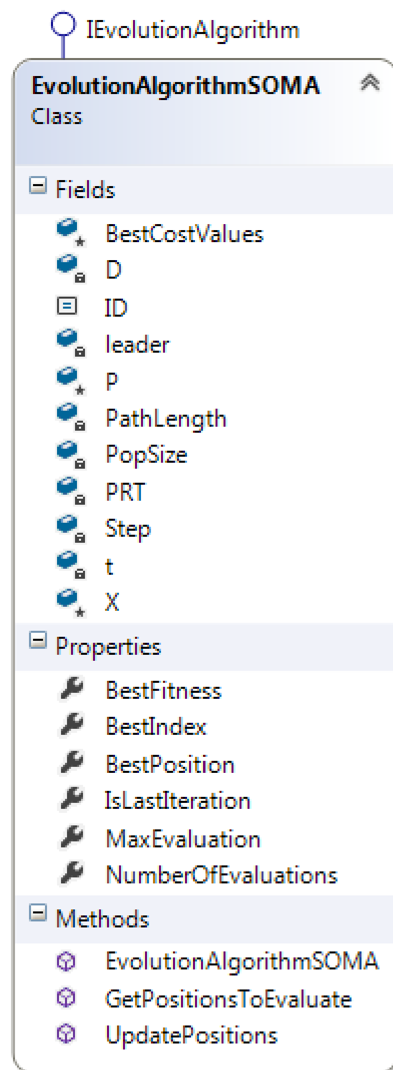
Před provedením každého evolučního kola je testována ukončovací podmínka v „property“ ***IsLastIteration***. Algoritmus se opakuje do té doby, dokud tato „property“ nevrátí hodnotu *True*.

8.4 Vývojový diagram algoritmu

Vývojový diagram zpracovaného algoritmu SPSO se nachází v příloze Příloha P I.

9 SAMO-ORGANIZUJÍCÍ SE MIGRAČNÍ ALGORITMUS (SELF-ORGANISING MIGRATING ALGORITHM)

Je použita verze algoritmu SOMA nazvaná AllToOne. Samotný algoritmus je popsán v teoretické části této práce (3.4). Algoritmus využívá interface *IEvolutionAlgorithm*, ze kterého dědí vlastnosti *BestFitness, BestIndex, BestPosition, IsLastIteration, MaxEvaulation, NumberOfEvaulations* a metody *GetPositionsToEvaulate* a *UpdatePositions*. Podrobnější popis tohoto interface se nachází v části (7). Při popisu algoritmu budu generace označovat jako migrace. (viz Parametry a terminologie).



Obr. 8. Přehled EvolutionAlgorithmSOMA

9.1 Členské proměnné (Fields)

double PathLength

Řídící parametr. Udává, jak daleko se aktivní jedinec při svém putování zastaví od vedoucího jedince.

double Step

Další z řídících parametrů. Tento ovlivňuje, jak podrobně bude daná funkce prohledána. Hodnota parametru by měla ležet v intervalu <0,11; PathLength>

double PRT

Tento parametr ovlivňuje, jak se bude daný jedinec pohybovat směrem k vedoucímu. Jedná se o jeden z řídících parametrů, které uživatel zadává při volání konstruktoru. Může nabývat hodnot v intervalu <0; 1>

double t

V této proměnné je uchován zrovna používaný násobek t. Ten může nabývat hodnot <0; PathLength> ve kterých roste po krocích o velikosti parametru Step.

int leader

Index jedince s nejlepší hodnotou účelové funkce (nejmenší).

int D

Dimenze problému – Tento parametr uživatel nenastavuje, je určen přímo problémem. Hodnota se získá z matice populace *X*.

double[] BestCostValues

Vektor nejlepších hodnot účelové funkce.

const string ID

Identifikační název třídy. Pro tuto třídu je tento string nastaven na „SOMA" (Self-Organising Migrating Algorithm)

int PopSize

Velikost populace. Tento parametr je určen velikostí populace předané konstruktoru v proměnné *X*.

double[][] P

Matice parametrů nejlepších jedinců v populaci. Rozměr matice je určen vztahem $PopSize \times D$.

double[][] X

Matice parametrů jedinců v populaci. Tato matice slouží k výpočtu nových jedinců v migračním kole. Rozměr matice je určen vztahem $PopSize \times D$.

```
1.  public class EvolutionAlgorithmSOMA : IEvolutionAlgorithm
2.  {
3.     public const string ID = "SOMA";
4.
5.     private int D;
6.     private int PopSize;
7.
8.     private double PathLength;
9.
10.    private double Step;
11.
12.    private double PRT;
13.
14.    private double t = 0;
15.    private int leader;
16.
17.    protected double[][] X = null;
18.    protected double[][] P = null;
19.    protected double[] BestCostValues = null;
```

9.2 Vlastnosti (Properties)

double BestFitness

Navrací nejlepší hodnotu (zde nejmenší) z vektoru nejlepších hodnot cílové funkce (**BestCostValues**).

```
20.    public double BestFitness
21.    {
22.       get
23.       {
24.          double bestFitness = BestCostValues.Min();
25.          return bestFitness;
26.       }
27.    }
```

int BestIndex

Vrací index nejlepší hodnoty z vektoru nejlepších hodnot účelové funkce (***BestCostValues***).

```
28.     public int BestIndex
29.     {
30.        get
31.        {
32.           int bestIndex = Array.IndexOf(BestCostValues,
              BestCostValues.Min());
33.           return bestIndex;
34.        }
35.     }
```

double[] BestPosition

Vrací vektor nejlepšího částice. K jeho nalezení se používá „property“ BestIndex (viz výše).

```
36.     public double[] BestPosition
37.     {
38.        get
39.        {
40.           var values = this.P[BestIndex];
41.           return values;
42.        }
43.     }
```

bool IsLastIteration

Ukončovací kritérium. Kontroluje, zda počet vyhodnocení účelové funkce již přesáhl maximální možný. Ten je nastavený uživatelem při volání konstruktoru (inicializaci instance třídy). V případě, že maximum překročeno není, tak vrací hodnotu false.

```
44.     public bool IsLastIteration
45.     {
46.        get
47.        {
48.           if (NumberOfEvaluations < MaxEvaluation)
49.              return false;
50.
51.           return true;
52.        }
53.     }
```

MaxEvaulation

Zde je uložen uživatelem nastavený maximální počet vyhodnocení účelovou funkcí.

NumberOfEvaulations

Počítá, kolik vyhodnocení účelové funkce již proběhlo.

```
54.     public int NumberOfEvaluations { get; private set; }
55.     public int MaxEvaluation { get; private set; }
```

9.3 Metody (Methods)

9.3.1 Konstruktor EvolutionAlgorithmSOMA

EvolutionAlgorithmSOMA(double[][] X, int maxevaluations = 30000, double pathLength = 3, double step = 0.75, double PRT = 0.25)

Konstruktor třídy EvolutionAlgorithmSOMA. Slouží k inicializaci všech proměnných algoritmu, k nastavení řídících parametrů a alokování paměti pro používaná data. Konstruktory jsou zvláštní typy metod, které jsou volané pouze při vytváření nové instance třídy. Tento konstruktor přijímá pět parametrů.

double[][] X – zde je uložená populace, se kterou se pracuje. Jedná se o dvourozměrné pole typu double ([index jedince][index parametru]).

int maxEvaluations – tento parametr určuje maximální počet evaluací, které se provedou před ukončením algoritmu. Pokud ho uživatel nezavolá, je nastaven na výchozí hodnotu 30000 evaluací.

double pathLength – zde uživatel nastavuje maximální vzdálenost, jakou jedinec během jednoho migračního kola vykoná. Výchozí hodnota je nastavena na 3.

double step – Další z řídících parametrů – délka kroku. Výchozí hodnota tohoto parametru je 0,75. Čím menší je tato hodnota (minimální hodnota by se měla pohybovat kolem 0,11), tím přesnější hledání je, ale zároveň je náročnější na výpočet (v každém migračním kole se provede více výpočtů nové pozice).

double PRT – Tento parametr ovlivňuje směr pohybu jedince v každém migračním kole. Hodnota leží v intervalu <0; 1> a je porovnávána s náhodně vygenerovaným číslem pro každý parametr každého jedince. Výchozí hodnota je 0,25.

```
1. public EvolutionAlgorithmSOMA(double[][] X, int
   maxevaluations = 30000, double pathLength = 3, double step
   = 0.75, double PRT = 0.25)
2. {
```

Stejně jako u obou dalších algoritmů, nejdříve dojde ke kontrole, zda je správně načtená populace a zda parametr ***maxEvaluations*** je v kladných hodnotách. Pokud je vše v pořádku, dojde k nahrání těchto hodnot do příslušných členských proměnných (***X*** pro populaci jedinců, ***MaxEvaluation*** pro maximální počet vyhodnocení účelové funkce) a vypočtení velikosti populace a dimenze problému z matice populace ***X***. Také řídící proměnné jsou uložené do příslušných členských a parametr ***t*** je nastaven na hodnotu větší, než je její povolená (větší než ***PathLength***) z důvodu hledání vedoucího jedince v prvním kole (viz ***UpdatePositions(double[] trialScores)***). Dojde také k vynulování počítadla provedených evaluací (***NumberOfEvaluations***).

```
3.     if ((X == null) || (X.Length == 0))
4.        throw new Exception("min.Count() != max.Count()");
5.
6.     if (maxevaluations <= 0)
7.        throw new Exception("maxiterations must be greater
          than 0");
8.
9.     this.X = X;
10.    this.MaxEvaluation = maxevaluations;
11.
12.    this.PopSize = X.Length;
13.    this.D = X[0].Length;
14.
15.    this.PathLength = pathLength;
16.    this.Step = step;
17.    this.t = this.PathLength + this.Step;
18.    this.PRT = PRT;
19.
20.    this.NumberOfEvaluations = 0;
```

Dále je alokován prostor pro nejlepší jedince (dvourozměrná matice ***P***) a pro nejlepší hodnoty účelové funkce (vektor ***BestCostValues***). První rozměry obou polí jsou alokovány pro velikost populace. Dále je pro každého jedince vymezen prostor o velikosti všech jeho parametrů (dimenze ***D***) a ***BestCostValues*** je nastaven na maximální možnou hodnotu.

```
21.     this.P = new double[this.PopSize][];
22.     this.BestCostValues = new double[this.PopSize];
23.
24.     for (int i = 0; i < this.PopSize; i++)
25.     {
26.        P[i] = new double[this.D];
27.        BestCostValues[i] = double.MaxValue;
28.     }
29.  }
```

9.3.2 GetPositionToEvaluate

double[][] GetPositionToEvaluate()

V této metodě se vrací pole všech nových pozic jedinců (***X***), kteří na tyto doputovali v tomto migračním kole. Tato metoda slouží k navrácení hodnot, které se mají ohodnotit účelovou funkcí.

```
1.  public double[][] GetPositionsToEvaluate()
2.  {
3.     return this.X;
4.  }
```

9.3.3 Provedení algoritmu - UpdatePositions

void UpdatePositions(double[] fitnesses)

V této metodě se provádí změna pozice jednotlivých jedinců podle algoritmu SOMA. Metoda přijímá jako parametr vektor ***fitnesses***, ve kterém je uložený výsledek vyhodnocení účelové funkce pro jedince, poslané metodou ***GetPositionEvaluate()***.

Před provedení vlastního algoritmu se zvedne počet již provedených evaluací o právě provedené a inicializuje se funkce *Random*. Poté se pro každého jedince porovná, zda hodnota jeho nové pozice (vrácená ve ***fitnesses***) je lepší než hodnota pozice staré (***BestCostValues***). Pokud tomu tak je, tak se tato hodnota nahraje do ***BestCostValues*** a zároveň se do matice nejlepších jedinců ***P*** tento jedinec nahraje.

```
1.  public void UpdatePositions(double[] fitnesses)
2.  {
3.      this.NumberOfEvaluations += fitnesses.Count();
4.
5.      Random rnd = new Random();
6.
7.      // projdu všechny jedince
8.      for (int i = 0; i < this.PopSize; i++)
9.      {
10.         if (fitnesses[i] < this.BestCostValues[i])
11.         {
12.             // přepíšu nejlepší parametry jedince
13.
14.             this.BestCostValues[i] = fitnesses[i];
15.             X[i].CopyTo(P[i], 0);
16.         }
17.     }
```

Následně se provede kontrola, zda je ***t*** větší než parametr ***PathLength.*** Pokud ano (k tomu dojde na konci každého migračního kola a při úplně prvním průchodu), tak se vyhledá nová pozice vedoucího jedince (***leader***) pomocí volání „property“ ***BestIndex*** a matice nejlepších jedinců (***P***) nakopíruje do matice jedinců, se kterou se počítá (***X***). Proměnná ***t*** se nastaví na hodnotu ***Step*** (Pokud by byla nastavena na 0, pak by bylo jedno evaluační kolo zbytečné, protože by se vyhodnocovali jedinci, kteří byli právě nakopírováni z matice nejlepších jedinců – již jsou vyhodnoceni).

```
18.     if (this.t > this.PathLength)
19.     {
20.         this.leader = this.BestIndex;
21.         for (int i = 0; i < this.PopSize; ++i)
22.         {
23.             P[i].CopyTo(X[i], 0);
24.         }
25.
26.         this.t = this.Step;
27.     }
```

Pro všechny jedince, kromě vedoucího (***leader***) se následně provede výpočet nové pozice. To začne vygenerováním náhodného čísla pro každý parametr jedince, které se porovná s řídícím parametrem ***PRT***. Pokud je toto číslo menší, tak se do PRTvektoru (viz Parametry a terminologie)(***PRTvalue***) dosadí 1, jinak 0. Následně se provede výpočet pomocí vzorce (11). Po vypočítání nové pozice všech jedinců dojde k navýšení proměnné ***t*** o hodnotu řídící proměnné ***Step***.

```
28.     for (int i = 0; i < this.PopSize; ++i)
29.     {
30.        if (i != leader)
31.        {
32.            for (int d = 0; d < this.D; ++d)
33.            {
34.               double PRTvalue = (rnd.NextDouble() < PRT)
                  ? 1 : 0;
35.
36.               X[i][d] = X[i][d] + (P[leader][d] -
                  X[i][d]) * this.t * PRTvalue;
37.            }
38.        }
39.     }
40.     this.t = this.t + this.Step;
41. }
```

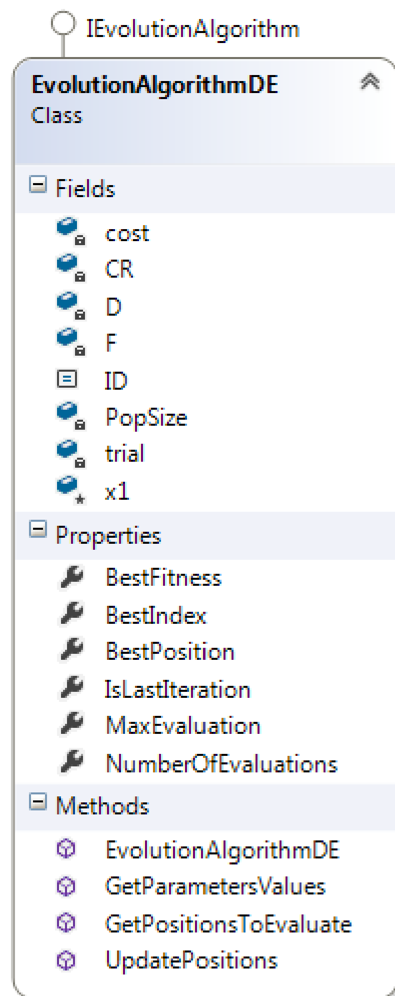
Před provedením každého evolučního kola je testována ukončovací podmínka v „property“ ***IsLastIteration***. Algoritmus se opakuje do té doby, dokud tato „property“ nevrátí hodnotu *True*.

9.4 Vývojový diagram

Vývojový diagram zpracovaného algoritmu SOMA se nachází v příloze (Příloha P II).

10 DIFERENCIÁLNÍ EVOLUCE (DIFFERENTIAL EVOLUTION)

Pro provedení algoritmu jsem zvolil verzi popsanou v [1][9]. Použitá varianta je označená jako **DE/rand/1/bin.** Samotný algoritmus je popsán v teoretické části této práce (4.4). Algoritmus využívá interface *IEvolutionAlgorithm*, ze kterého dědí vlastnosti („Properties“) *BestFitness, BestIndex, BestPosition, IsLastIteration, MaxEvaulation, NumberOfEvaulations* a metody *GetPositionsToEvaulate* a *UpdatePositions.* Podrobnější popis tohoto interface se nachází v části (7). Tento algoritmus využívá 2 řídící parametry – Práh křížení (***CR***) a mutační konstanta (***F***). Popis těchto parametrů se nachází v teoretické části (viz Parametry a terminologie).



Obr. 9. Přehled EvolutionAlgorithmDE

10.1 Členské proměnné (Fields)

double CR

Zde je uloženo nastavení řídícího parametru CR – práh křížení.

double F

Proměnná pro uchování mutační konstanty.

int D

Dimenze problému – Tento parametr uživatel nenastavuje, je určen přímo problémem. Hodnota se získá z matice populace $x1$.

double[] cost

Vektor nejlepších hodnot účelové funkce.

const string ID

Identifikační název třídy. Pro tuto třídu je tento string nastaven na „DE“ (Differential Evolution)

int PopSize

Velikost populace. Tento parametr je určen velikostí populace předané konstruktoru v proměnné $x1$.

double[][] trial

Matice parametrů jedinců v populaci. Tato matice slouží k výpočtu nových jedinců v každém evolučním cyklu. Rozměr matice je určen vztahem $PopSize \times D$.

double[][] x1

Matice parametrů jedinců v populaci. Tato matice slouží k uchování nejlepších jedinců. Rozměr matice je určen vztahem $PopSize \times D$.

```
1.  public class EvolutionAlgorithmDE : IEvolutionAlgorithm
2.  {
3.      public const string ID = "DE";
4.
5.      private int D;
6.      private int PopSize;
7.
8.      private double CR;
9.      private double F;
10.
11.     protected double[][] x1 = null;
12.     private double[][] trial = null;
13.     private double[] cost = null;
```

10.2 Vlastnosti (Properties)

double BestFitness

Navrací nejlepší hodnotu (zde nejmenší) z vektoru nejlepších hodnot cílové funkce (***cost***).

```
14.     public double BestFitness
15.     {
16.        get
17.        {
18.           double bestFitness = this.cost.Min();
19.           return bestFitness;
20.        }
21.     }
```

int BestIndex

Vrací index nejlepší hodnoty z vektoru nejlepších hodnot účelové funkce (***cost***).

```
22.     public int BestIndex
23.     {
24.        get
25.        {
26.           // Vrací index nejlepší hodnoty z vektoru cost
27.           int bestIndex = Array.IndexOf(cost, cost.Min());
28.           return bestIndex;
29.        }
30.     }
```

double[] BestPosition

Vrací vektor nejlepšího částice. K jeho nalezení se používá „property“ *BestIndex* (viz výše).

```
31.     public double[] BestPosition
32.     {
33.        get
34.        {
35.           var values = this.x1[BestIndex];
36.           return values;
37.        }
38.     }
```

bool IsLastIteration

Ukončovací kritérium. Kontroluje, zda počet vyhodnocení účelové funkce již přesáhl maximální možný. Ten je nastavený uživatelem při volání konstruktoru. V případě, že maximum překročeno není, tak vrací hodnotu *false*.

```
39.     public bool IsLastIteration
40.     {
41.        get
42.        {
43.           if (NumberOfEvaluations < MaxEvaluation)
44.              return false;
45.
46.           return true;
47.        }
48.     }
```

MaxEvaulation

Zde je uložen uživatelem nastavený maximální počet vyhodnocení účelovou funkcí.

NumberOfEvaulations

Počítá, kolik vyhodnocení účelové funkce již proběhlo.

```
49.     public int NumberOfEvaluations { get; private set; }
50.     public int MaxEvaluation { get; private set; }
```

10.3 Metody (Methods)

10.3.1 Konstruktor EvolutionAlgorithmDE

EvolutionAlgorithmDE (double[][] x1, int maxEvaluations = 30000, double CR = 0.8, double f = 0.3)

Konstruktor třídy EvolutionAlgorithmDE. Slouží k inicializaci všech proměnných algoritmu, k nastavení řídících parametrů a alokování paměti pro používaná data. Konstruktory jsou zvláštní typy metod, které jsou volané pouze při vytváření nové instance třídy. Tento konstruktor přijímá čtyři parametry.

double[][] x1 – zde je uložená populace, se kterou se pracuje. Jedná se o dvourozměrné pole typu double ([index jedince][index parametru]).

int maxEvaluations – tento parametr určuje maximální počet evaluací, které se provedou před ukončením algoritmu. Pokud ho uživatel nezavolá, tak je nastaven na výchozí hodnotu 30000 evaluací.

double CR – zde uživatel nastavuje práh křížení pro algoritmus. Výchozí hodnota je nastavena na 0,8.

Double f – uživatelem nastavený další řídící parametr – mutační konstanta F. Výchozí hodnota tohoto parametru je 0,3.

```
1.  public EvolutionAlgorithmDE(double[][] x1, int
    maxEvaluations = 30000, double CR = 0.8, double f = 0.3)
2.  {
```

Nejprve je nutné ověřit, zda matice populace (***x1***) byla předána správně. Pokud je tato matice prázdná, program „vyhodí“ výjimku. Výjimka bude „vyhozena“ i v případě, že ***maxEvaluations*** nesplňuje podmínku > 0.

Pokud není žádná výjimka detekována, je vynulováno počítadlo vyhodnocení (***NumberOfEvaluations***) a může se přejít k uložení hodnot do členských proměnných. Nejdříve se uloží práh křížení z parametru konstruktoru (***CR***) do proměnné (*this.***CR**) a poté mutační konstanta (***f***) do (*this.**F***). Stejně tak maximální počet vyhodnocení účelové funkce (***maxEvaluations*** do *this.**MaxEvaluation***) a matice populace (z ***x1*** do členského proměnné označené *this.**x1***). Z velikosti prvního rozměru matice ***x1*** je vyčtena velikost

celé populace a tato hodnota je uložena do ***PopSize*** a z velikosti druhého rozměru je vyčtena dimenze problému (počet parametrů každého jedince) uložena do ***D***.

```
3.     if ((x1 == null) || (x1.Length == 0))
4.        throw new Exception("min.Count() != max.Count()");
5.
6.     if (maxEvaluations <= 0)
7.        throw new Exception("maxiterations must be greater
          than 0");
8.
9.     this.NumberOfEvaluations = 0;
10.
11.    this.CR = CR;
12.    this.F = f;
13.
14.    this.MaxEvaluation = maxEvaluations;
15.    this.x1 = x1;
16.
17.    this.PopSize = x1.Length;
18.    this.D = x1[0].Length;
```

Dojde k alokaci paměti pro dvourozměrné pole typu double ***trial*** a jednorozměrné ***cost***. U obou dvou polí je první rozměr velikosti populace (***PopSize***). Poté se pro každého jedince alokuje i druhý rozměr matice ***trial*** a to o velikosti dimenze optimalizovaného problému (***D***). Zároveň se všichni jedinci z matice ***x1*** překopírují do matice ***trial***, aby mohli být předání k evaluaci jejich účelové funkce. Nakonec se celá matice nejlepších vyhodnocených hodnot (***cost***) nastaví na maximální hodnotu (protože hledáme minimum) typu double.

```
19.    this.trial = new double[this.PopSize][];
20.    this.cost = new double[this.PopSize];
21.
22.    // i - index jedince
23.    for (int i = 0; i < this.PopSize; ++i)
24.    {
25.       this.trial[i] = new double[this.D];
26.       this.x1[i].CopyTo(trial[i], 0);
27.       cost[i] = double.MaxValue;
28.    }
29. }
```

10.3.2 GetPositionToEvaluate

double[][] GetPositionToEvaluate()

V této metodě se vrací pole všech jedinců, kteří vznikli v tomto evolučním cyklu (***trial***). Tato metoda slouží k navrácení hodnot, které se mají ohodnotit účelovou funkcí.

```
1.  public double[][] GetPositionsToEvaluate()
2.  {
3.     return this.trial;
4.  }
```

10.3.3 Provedení algoritmu – UpdatePositions

void UpdatePositions(double[] trialScores)

Zde se vykonává celý algoritmus diferenciální evoluce. Tato metoda přijímá jako parametr vektor vypočtených hodnot účelové funkce jedinců, kteří byli posláni k evaluaci metodou ***GetPositionToEvaluate()***.

Ještě před samotným prováděním evolučního algoritmu se zvedne počet dosud vyhodnocených funkcí (***NumberOfEvaluations***) o počet právě vyhodnocených (***trialScores.Count()***). Jako prvním krokem samotného algoritmu je vyhodnocení, zda některý z nových jedinců (uložené v matici ***trial***, hodnoty ve vektoru ***trialScores***) má lepší hodnotu jak jedinec starý (uložené v matici ***x1***, hodnoty ve vektoru ***cost***). Pokud je nová hodnota lepší dojde k jejímu překopírování do vektoru nejlepších výsledků (***cost***), zároveň dojde i k přesunu jedince do matice nejlepších jedinců (***x1***).

```
1.  public void UpdatePositions(double[] trialScores)
2.  {
3.     this.NumberOfEvaluations += trialScores.Count();
4.
5.     for (int i = 0; i < this.PopSize; i++)
6.     {
7.        double score = trialScores[i];
8.
9.        if (score < cost[i])
10.       {
11.          trial[i].CopyTo(x1[i], 0);
12.          cost[i] = score;
13.       }
14.    }
```

Dalším krokem algoritmu je vybrání tří náhodných jedinců (provádí se verze algoritmu označená jako **DE/rand/1/bin** viz VARIANTY DE)

To se děje pomocí funkce *rnd.Next(PopSize)*, která generuje náhodné číslo typu integer v intervalu <0; PopSize>. Je zajištěno, aby nebyli náhodně vybráni stejní jedinci vícekrát.

```
15.     Random rnd = new Random();
16.
17.     //započetí cyklu generace
18.     for (int i = 0; i < this.PopSize; i++)
19.     {
20.        int a = 0;
21.        int b = 0;
22.        int c = 0;
23.
24.        do
25.        {
26.           a = rnd.Next(PopSize);
27.        } while (a == i);
28.
29.        do
30.        {
31.           b = rnd.Next(PopSize);
32.        } while ((b == i) || (b == a));
33.
34.        do
35.        {
36.           c = rnd.Next(PopSize);
37.        } while ((c == i) || (c == a) || (c == b));
```

Pro každého jedince je následně náhodně určen parametr, od kterého se má další cyklus provádět. Tím se zajistí, že poslední parametr nového vektoru bude vždy na jiné pozici. Tento poslední parametr bude vždy ze tzv. *šumového vektoru* (viz Evoluční cyklus). To zajistí, že v každém evolučním cyklu dojde ke změně všech jedinců (žádný z jedinců nebude stejný, jako jeho varianta v předchozím cyklu. Lišit se budou alespoň v jednom parametru).

Pro každý parametr jedince se následně vygeneruje náhodné číslo v rozsahu <0; 1>, které se porovná s prahem křížení (***CR***). Pokud je náhodné číslo menší (nebo pokud se jedná o poslední parametr jedince), pak se počítá nová pozice daného parametru jedince pomocí vzorce (19) (výpočet tzv. *šumového vektoru*).

V opačném případě, je použit parametr z původního jedince. Nový jedinec je uložen do matice ***trial***. Po vykonání této operace pro všechny jedince je matice ***trial*** předána pomocí metody ***GetPositionToEvaluate()*** k vyhodnocení hodnot účelové funkce.

```
38.         // jednotlivé parametry jedince
39.
40.         int j = rnd.Next(D);
41.
42.         for (int k = 1; k <= D; k++)
43.         {
44.            if (rnd.NextDouble() < CR || k == D)
45.            {
46.               this.trial[i][j] = x1[c][j] + ( F * (
                  x1[a][j]  - x1[b][j] ) );
47.            }
48.            else
49.            {
50.               this.trial[i][j] = x1[i][j];
51.            }
52.
53.            j = (j + 1) % D;
54.         }
55.      }
56.  }
```

Před provedením každého evolučního kola je testována ukončovací podmínka v „property“ ***IsLastIteration***. Algoritmus se opakuje do té doby, dokud tato „property“ nevrátí hodnotu *True*.

10.4 Vývojový diagram

Vývojový diagram zpracovaného algoritmu SOMA se nachází v příloze Příloha P III.

11 POROVNÁNÍ ALGORITMŮ

Vybrané algoritmy byly spuštěny 15 krát pro každý zdroj dat (CHP_A a CHP_B). Každé spuštění odpovídá běhu algoritmu do ukončovací podmínky 300000 vyhodnocení účelové funkce.

Tab. 6. Porovnání algoritmů pro CHP_A

CHP_A

Měření	SOMA	DE	SPSO
1.	4894,68	4893,05	5063,82
2.	4904,45	4892,47	4894,27
3.	4899,30	4893,26	5145,88
4.	4895,46	4895,60	4914,28
5.	4893,11	4893,03	4892,99
6.	4893,13	4892,04	5030,34
7.	4937,90	4893,45	4902,26
8.	4892,83	4892,46	4900,54
9.	4898,28	4896,07	4903,77
10.	4893,53	4892,20	4948,46
11.	4908,24	4897,60	4897,58
12.	4894,68	4891,94	4907,61
13.	4902,72	4893,37	4902,14
14.	4894,25	4900,37	4904,99
15.	4894,26	4897,28	4958,25
Průměr	**4899,79**	**4894,28**	**4944,48**
SMODCH	**11,15**	**2,44**	**73,37**

Tab. 7. *Porovnání algoritmů pro CHP_B*

CHP_B

Měření	SOMA	DE	SPSO
1.	2582,25	2576,12	2582,33
2.	2581,54	2575,21	2660,05
3.	2600,02	2576,66	2594,52
4.	2597,47	2573,23	2592,31
5.	2578,38	2573,68	2640,02
6.	2576,57	2574,99	2589,19
7.	2584,34	2578,84	2581,74
8.	2624,44	2582,65	2955,50
9.	2603,36	2573,34	2611,86
10.	2617,00	2591,80	2583,05
11.	2609,78	2574,80	2592,34
12.	2651,33	2575,51	2694,72
13.	2634,79	2582,67	2581,66
14.	2584,00	2575,01	2661,10
15.	2620,31	2580,78	2603,41
Průměr	**2603,04**	**2577,69**	**2634,92**
SMODCH	**22,05**	**4,84**	**92,26**

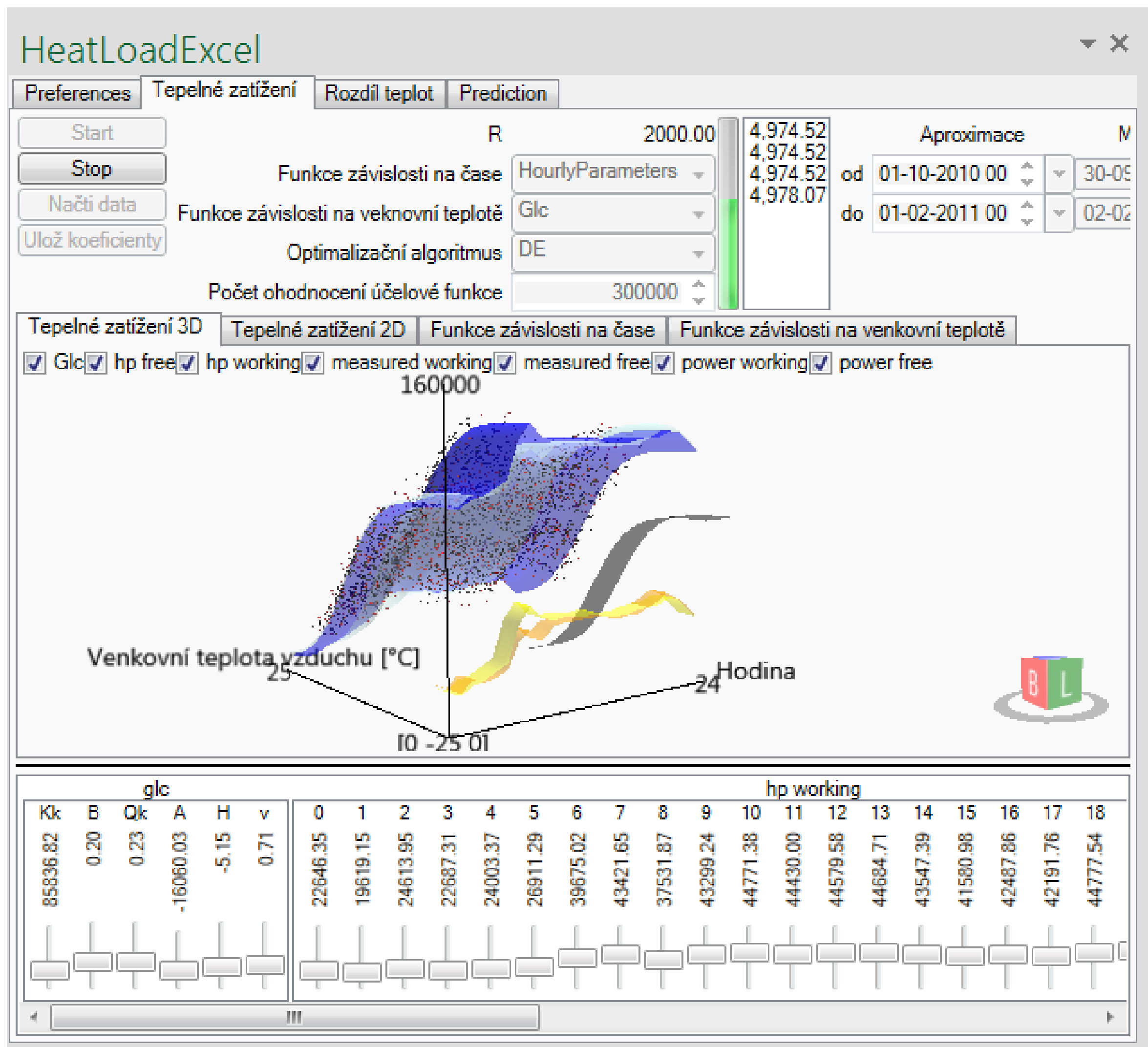
V tabulkách se nachází porovnání vyhodnocení účelových funkcí pro všechny tři algoritmy. Bylo provedeno 15 měření a z výsledných hodnot následně vypočítán průměr a směrodatná odchylka základního souboru (***SMODCH***). Ta se počítá pomocí vzorce (25) a určuje míru rozptýlení od průměrné střední hodnoty.

$$\sqrt{\frac{\sum(x-\bar{x})^2}{n}} \tag{25}$$

Kde x je střední hodnota průměru a n je velikost hodnoty.[16]

Z měření vyplývá, že nejúspěšnější algoritmus pro daný problém je diferenciální evoluce (DE). Naopak nejhorší z vybraných algoritmů je optimalizace rojení částic (PSO).

Hodnoty byly získány simulací v grafickém rozhraní, do kterého byly mé algoritmy implementovány pomocí Interface *IEvolutionAlgorithm.* Toto grafické rozhraní bylo vytvořeno panem Ing. Erikem Králem.



Obr. 10. Ukázka grafického interface pro porovnání algoritmů.

ZÁVĚR

Pro zpracování algoritmů bylo důležité, aby byla oddělená část, ve které se provádí ohodnocení účelové funkce od části, která provádí vlastní algoritmus. Toto oddělení bylo nutné z důvodu bezproblémového přidání, nebo nahrazení, jak evolučního algoritmu, tak účelové funkce. Toho se dokázalo dosáhnout pomocí implementace interface IEvolutionAlgorithm, které zpracovává dané evoluční algoritmy, ale výpočty hodnot účelové funkce probíhají mimo tento interface. K předání hodnot k vyhodnocení tak slouží jedna z metod interface, a sice GetPositionToEvaluate.

Z výsledků porovnání vyplývá, že pro daný optimalizační problém vyšel nejlépe algoritmus diferenciální evoluce (DE). Nejen, že pro obě testovací skupiny dosáhl nejlepších výsledků, ale také byl v těchto výsledcích nejvíce konzistentní. Naopak nejhůře dopadl algoritmus PSO, který nevykazoval tak dobré výsledky. Je možné, že by tento algoritmus fungoval lépe, kdyby každá částice cestovala k celkové nejlepší částici, na místo cestování k nejlepší částici v okolí. Tím by se asi snížila schopnost nalézt více extrémů, ale v konečném výsledku by to mohlo přinést lepší výsledek pro tento konkrétní problém.

Práci je možné v budoucnu rozšířit o implementaci dalších variant a strategií zpracovaných evolučních algoritmů a tím získat kompletní přehled o účinnosti těchto algoritmů na daném problému. Také by bylo vhodné provést v budoucnu rozsáhlejší měření výkonu algoritmů, což by už ovšem vyžadovalo specializovaný a výkonný výpočetní hardware.

ZÁVĚR V ANGLIČTINĚ

It was important to separate part where is done evaluation of the cost function, from part which performs evolutionary algorithm. This separation was necessary for being able to smoothly add or replace both evolutionary algorithms and the cost function. This was achieved through the implementation of IEvolutionAlgorithm interface that handles the evolutionary algorithms, but the calculations of cost function values are outside of this interface. To send data to evaluate, I am using one of the methods of the interface, namely GetPositionToEvaluate.

From the comparison results implies that for a given optimization problem came preferably algorithm Differential Evolution (DE). Not only that, for both test groups achieved the best results, but also had the most consistent results. The worst ended PSO algorithm, which did not show such good results. It is possible that this algorithm would work better if each particle traveled to the “global best” particle, instead of travelling to the best particle in its vicinity. That would probably reduce the ability to find more extremes, but in the end it could bring a better result for this particular problem.

It is possible to expand the work in the future by implementation of other variations and strategies of evolutionary algorithms and gain a complete overview of the effectiveness of these algorithms. It would also be appropriate to make more extensive performance measurement of these algorithms in the future, which, however, require specialized and powerful computing hardware.

SEZNAM POUŽITÉ LITERATURY

[1] ZELINKA, Ivan. Umělá inteligence v problémech globální optimalizace. 1. vyd. Praha: BEN - technická literatura, 2002, 189 s. ISBN 80-730-0069-5.

[2] KRAMPL J. Implementace vybraných evolučních algoritmů v prostředí .NET. Zlín, 2011. Bachelor thesis (Bc.). Tomas Bata University in Zlín, Faculty of Applied Informatics

[3] VAŘACHA, P., M. POSPÍŠILÍK, I. MOTÝL, M. BLIŽŇÁK, D. SLOVÁK, J. KRAMPL a J. KOLEK. *Comparison of Evolutionary Algorithms SOMA, DE, PSO*. s. 431-435. ISBN 978-1-61804-108-1.

[4] BRATTON, Daniel a KENNEDY. 2007 IEEE Swarm Intelligence Symposium: Honolulu, HI, 1-5 April 2007. *Defining a Standard for Particle Swarm Optimization*. 2007. DOI: 1-4244-0708-7.

[5] HU, Xiaohui. Particle Swarm Optimization: Introduction. HU, Xiaohui. *Particle Swarm Optimization* [online]. 2006 [cit. 2013-06-10]. Dostupné z: http://www.swarmintelligence.org/index.php

[6] Constriction factors and Parameters. UNIVERSITY CARLOS III OF MADRID. *PSO: Parameters* [online]. 2005 [cit. 2013-06-10]. Dostupné z: http://tracer.uc3m.es/tws/pso/parameters.html

[7] HU, Xiaohui. PSO Tutorial. HU, Xiaohui. *Particle Swarm Optimization* [online]. 2006 [cit. 2013-06-10]. Dostupné z: http://www.swarmintelligence.org/tutorials.php

[8] Book Support. *SOMA* [online]. 2005 [cit. 2013-06-10]. Dostupné z: http://www.ft.utb.cz/people/zelinka/soma/

[9] STORN, Rainer a Kenneth PRICE. Differential Evolution: A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces. *Journal of Global Optimization*. 1997, č. 11, 341–359.

[10] Overview of the .NET Framework. *Microsoft Developer Network* [online]. 2013 [cit. 2013-06-10]. Dostupné z: http://msdn.microsoft.com/en-us/library/zw4w595w.aspx

[11] .NET Framework. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001-2013, 2013-06-03 [cit. 2013-06-10]. Dostupné z: http://en.wikipedia.org/wiki/.NET_Framework

[12] HLUBINKA, Daniel. Metody pro prokládání křivek s použitím na reálných datech. In: *Robust*. 1998. p. 55-76.

[13] YU, Xinjie a Mitsuo GEN. Introduction to evolutionary algorithms. New York: Springer, c2010. Decision engineering. ISBN 18-499-6129-8.

[14] WEISE, Thomas. Global Optimization Algorithms: Theory and Aplication [online]. 2009, 2009-06-26 [cit. 2013-02-01]. 2. Dostupné z: http://www.it-weise.de/projects/book.pdf

[15] KRÁL, Tomáš. Knihovna evolučních optimalizačních algoritmů v prostředí Java. Zlín, 2011. diplomová práce (Ing.). Univerzita Tomáše Bati ve Zlíně. Fakulta aplikované informatiky

[16] SMODCH. MICROSOFT. *Excel - Office.com* [online]. 2013 [cit. 2013-06-12]. Dostupné z: http://office.microsoft.com/cs-cz/excel-help/smodch-HP005209281.aspx

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

DE	Differential Evolution
PSO	Particle Swarm Optimization.
SOMA	Self-Organizing Migrating Algorithm
SPSO	Standard for Particle Swarm Optimization

SEZNAM OBRÁZKŮ

Obr. 1. Ukázka provedení algoritmu SPSO. ... 18
Obr. 2. Ukázka provedení algoritmu SOMA ... 26
Obr. 3. Ukázka binomiálního křížení. ... 31
Obr. 4. Ukázka exponenciálního křížení ... 31
Obr. 5. Ukázka provedení algoritmu DE ... 34
Obr. 6. Přehled IEvolutionAlgorithm ... 39
Obr. 7. Přehled EvolutionAlgorithmSPSO ... 41
Obr. 8. Přehled EvolutionAlgorithmSOMA ... 51
Obr. 9. Přehled EvolutionAlgorithmDE ... 60
Obr. 10. Ukázka grafického interface pro porovnání algoritmů. ... 70

SEZNAM TABULEK

Tab. 1. Příklad populace 11
Tab. 2. Přehled parametrů algoritmu PSO. Převzato z [3]... 13
Tab. 3. Přehled parametrů algoritmu SOMA. Převzato z [1] 20
Tab. 4. Přehled parametrů algoritmu DE. Převzato z [1] 28
Tab. 5. Parametry a meze aproximační funkce... 38
Tab. 6. Porovnání algoritmů pro CHP_A 69
Tab. 7. *Porovnání algoritmů pro CHP_B* 69

SEZNAM PŘÍLOH

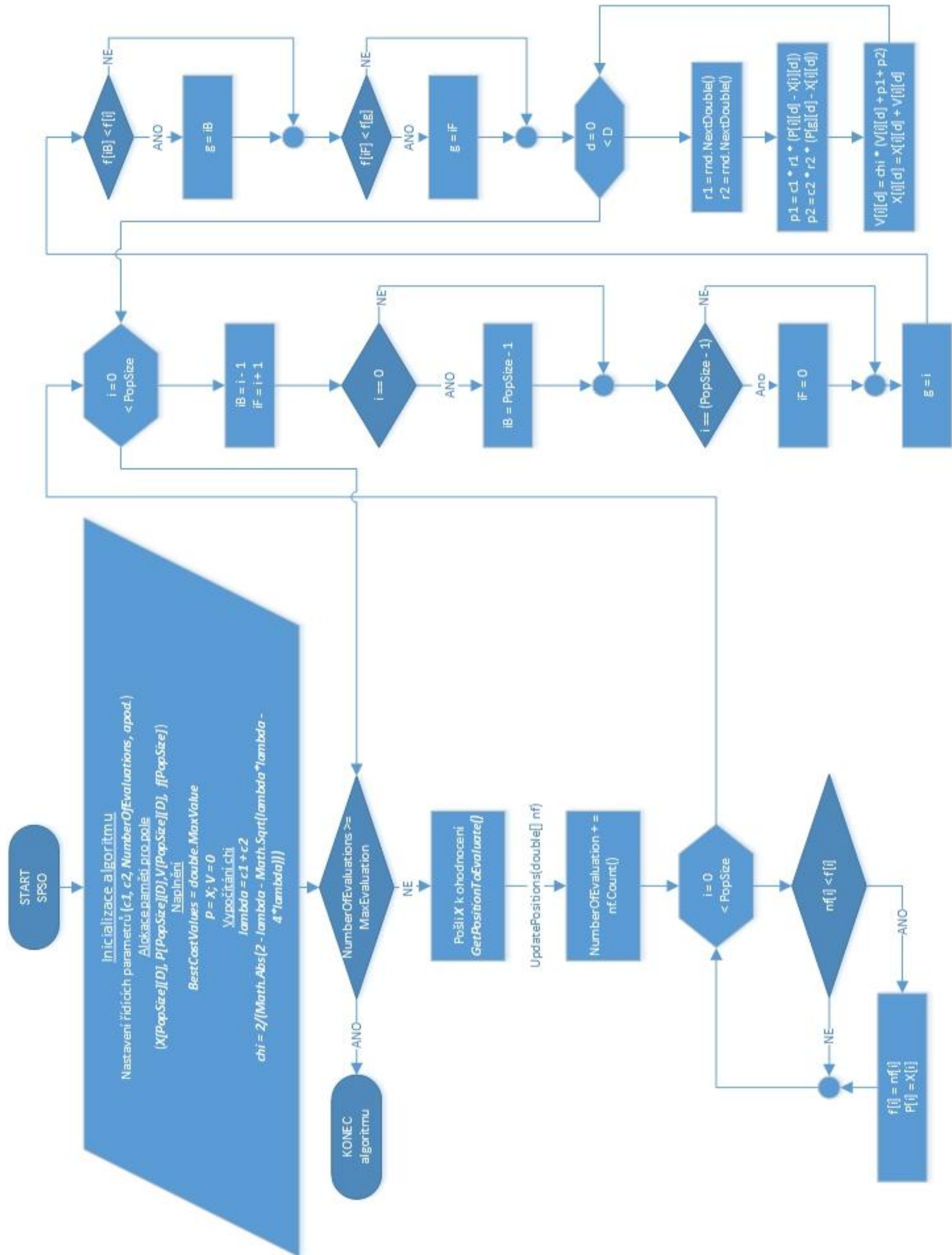
P I Vývojový diagram SPSO

P II Vývojový diagram SOMA

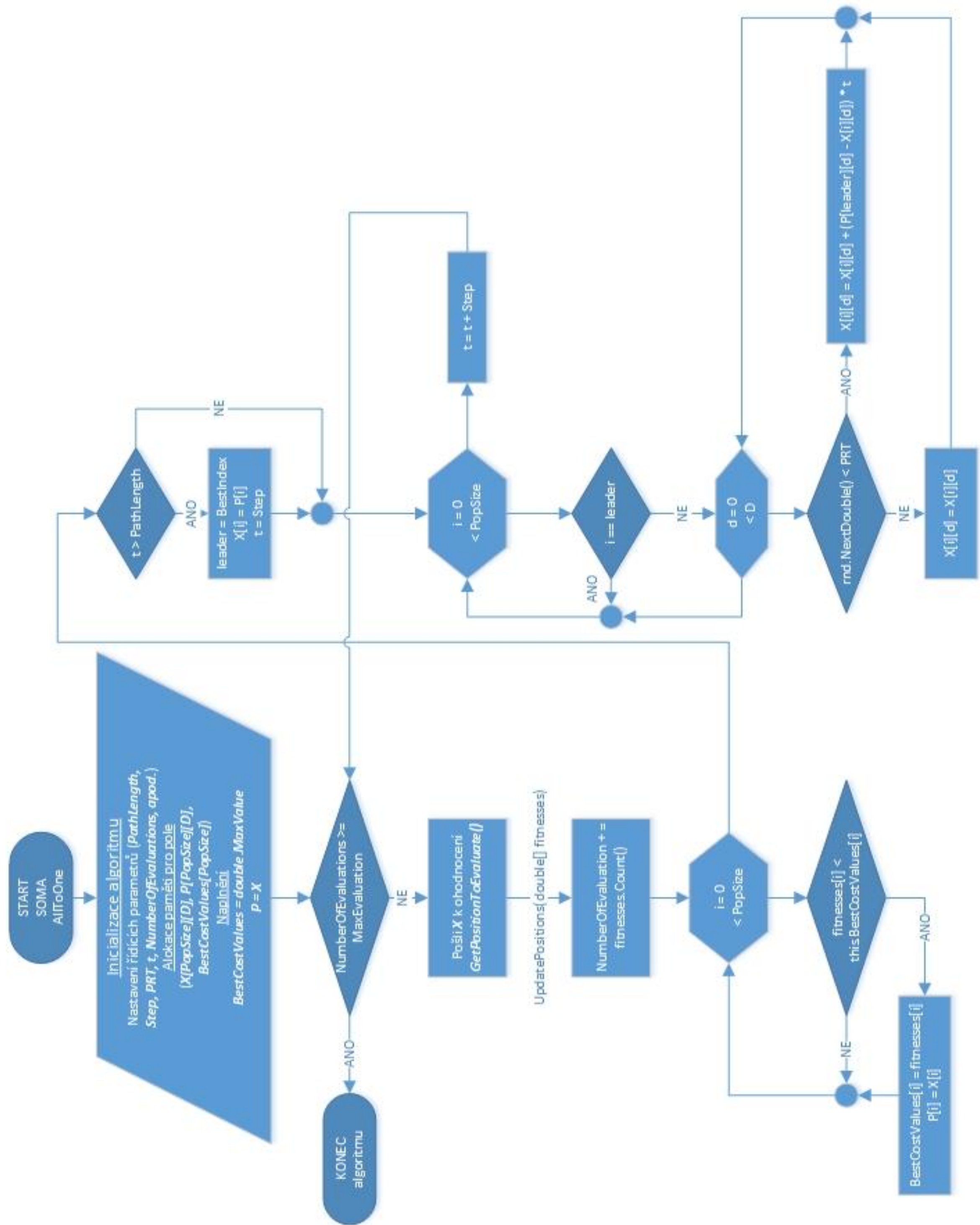
P III Vývojový diagram DE

P IV CD

PŘÍLOHA P I: VÝVOJOVÝ DIAGRAM SPSO



PŘÍLOHA P II: VÝVOJOVÝ DIAGRAM SOMA



PŘÍLOHA P III: VÝVOJOVÝ DIAGRAM DE