VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA STROJNÍHO INŽENÝRSTVÍ
FACULTY OF MECHANICAL ENGINEERING

ÚSTAV AUTOMATIZACE A INFORMATIKY
INSTITUTE OF AUTOMATION AND COMPUTER SCIENCE

BENCHMARKING PRO HEJNOVÉ OPTIMALIZAČNÍ ALGORITMY
BENCHMARKING OF SWARM OPTIMIZATION ALGORITHMS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. Eduard Mittaš**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. Jakub Kůdela, Ph.D.**
SUPERVISOR

BRNO 2022

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA STROJNÍHO INŽENÝRSTVÍ

Zadání diplomové práce

Ústav:	Ústav automatizace a informatiky
Student:	**Bc. Eduard Mittaš**
Studijní program:	Aplikovaná informatika a řízení
Studijní obor:	bez specializace
Vedoucí práce:	**Ing. Jakub Kůdela, Ph.D.**
Akademický rok:	2021/22

Ředitel ústavu Vám v souladu se zákonem č.111/1998 o vysokých školách a se Studijním a zkušebním řádem VUT v Brně určuje následující téma diplomové práce:

Benchmarking pro hejnové optimalizační algoritmy

Stručná charakteristika problematiky úkolu:

Benchmarking je jedno z důležitých témat při vývoji a srovnávání různých optimalizačních method. V posledních letech se objevily nové nástroje pro benchmarking. Práce se bude zabývat jak popisem těchto nástrojů, tak jejich implementací pro srovnání vybraných hejnových algoritmů.

Cíle diplomové práce:

Rešerše a popis dostupných nástrojů pro benchmarking.
Výběr vhodných hejnových algoritmů a testovacích úloh pro srovnání.
Benchmarking vybraných algoritmů.

Seznam doporučené literatury:

HANSEN, D. et al., Coco: a platform for comparing continuous optimizers in a black-box setting. Optimization Methods and Software, 2021, vol. 36, no. 1, pp. 114–144.

BANSAL, J.C. et al., Evolutionary and Swarm Intelligence Algorithms, 2018, Springer.

KUDELA, J., Novel zigzag-based benchmark functions for bound constrained single objective optimization. IEEE Congress on Evolutionary Computation (CEC), 2021, pp. 857–862.

DOERR, C. et al., IOHprofiler: A benchmarking and profiling tool for iterative optimization heuristics. arXiv e-prints:1810.05281, Oct. 2018.

Termín odevzdání diplomové práce je stanoven časovým plánem akademického roku 2021/22

V Brně, dne

L. S.

doc. Ing. Radomil Matoušek, Ph.D. ředitel ústavu	doc. Ing. Jaroslav Katolický, Ph.D. děkan fakulty

ABSTRAKT

Táto práca sa zaoberá benchmarkingom pre rojové optimalizačné algoritmy. Prvá časť sa zaoberá optimalizačným problémom a jeho významom pri testovaní výkonnosti algoritmov. Ďalšia kapitola pojednáva o samotnom benchmarkingu, jeho nástrojoch a software platformách. Následne sú teoreticky popísané jednotlivé algoritmy, ktoré boli vybrané na implementáciu. Po tejto časti nasleduje popis programovej realizácie riešenia, zvolených algoritmov, zvolených testovacích funkcií, a tiež dát, ktoré program exportuje. Posledná kapitola pojednáva o výsledkoch jednotlivých testov výkonnosti, pri ktorých algoritmy riešili dané testovacie problémy. Napokon sú tieto výsledky zhodnotené a je z nich vyvodený záver o efektivite a výkonnosti algoritmov.

ABSTRACT

This thesis deals with benchmarking of swarm optimization algorithms. First part handles optimization problem and it’s meaning in testing of algorithm’s performances. Next chapter describes the very benchmarking itself, it’s tools and software platforms. Afterwards individual algorithms, which were selected for implementation are described. Following this part is a program realization of solution, selected algorithms, selected testing functions and the data, which is exported by the program. The last chapter deals with results of respective performance tests, in which algorithms solved given testing problems. Eventually these results are evaluated and from them an outcome of efficiency and performance of algorithms is formed.

KĽÚČOVÉ SLOVÁ

Benchmarking, hejnové algoritmy, rojové algoritmy, testovacie funkcie, stochastická optimalizácia, IOHanalyzer

KEYWORDS

Benchmarking, swarm algorithms, test functions, stochastic optimization, IOHanalyzer

ÚSTAV AUTOMATIZACE A INFORMATIKY

2022

BIBLIOGRAFICKÁ CITÁCIA

MITTAŠ, Eduard. *Benchmarking pro hejnové optimalizační algoritmy*. Brno: Vysoké učení technické v Brně, Fakulta strojního inženýrství, Ústav automatizace a informatiky, 2022, 115 s. Diplomová práce. Vedoucí práce: Ing. Jakub Kůdela, Ph.D.

POĎAKOVANIE

Chcel by som sa poďakovať vedúcemu tejto diplomovej práce, pánovi Ing. Jakubovi Kůdelovi, Ph.D. za jeho pomoc, rady, trpezlivosť a priateľský prístup. Taktiež by som sa chcel poďakovať svojej priateľke a svojej rodine – môjmu bratovi, rodičom a starému otcovi, ktorí pri mne vždy stáli a podporovali ma.

ČESTNÉ PREHLÁSENIE

Prehlasujem, že táto diplomová práca je mojím pôvodným dielom, vypracoval som ju samostatne pod vedením vedúceho práce Ing. Jakuba Kůdelu, PhD. a s použitím odbornej literatúry a ďalších informačných zdrojov, ktoré sú všetky citované v práci a uvedené v zozname literatúry .

Ako autor uvedenej práce ďalej prehlasujem, že v súvislosti s vytvorením tejto práce som neporušil autorské práva tretích osôb, najmä som nezasiahol nedovoleným spôsobom do cudzích autorských práv osobnostných a som si plne vedomý následkov porušenia ustanovení § 11 a následného autorského zákona č. 121/2000 Sb, vrátane možných trestnoprávnych dôsledkov.

V Brne dňa 20. 5. 2022 ……

Eduard Mittaš

OBSAH

1. ÚVOD

Vývoj výpočtovej techniky a informačných technológií, ktorý začal v polovici minulého storočia a stále rýchlejším tempom pokračuje až do súčasnosti, postupne viedol k vynájdeniu mnohých moderných algoritmov a heuristík aplikovateľných pri riešení optimalizačných problémov. Algoritmy a heuristiky, ktoré v minulosti nebolo možné aplikovať, či naprogramovať ich software-ové prevedenie kvôli limitom stanoveným technikou vtedajšej doby, sa stali realizovateľnými.

Ďalším pokrokom bolo hľadanie nových inšpirácií v informačných technológiách. Nové algoritmy a heuristiky, ktoré vznikali, boli teda inšpirované napríklad prírodou a človekom. Vznikli preto riešenia ako neurónové siete, evolučné algoritmy či algoritmy inšpirované rojovou inteligenciou, tzv. swarm algoritmy. Táto diplomová práca sa bude zaoberať práve na poslednú zmienenou skupinou.

Veľká variácia algoritmov a prístupov, nehľadiac na inšpiráciu, vedie na problém správnej voľby a ich aplikácie. Každý má totiž svoje kvality, v ktorých dokáže prekonať ostatné alebo sa vyrovnať iným, no taktiež disponuje zápormi, ktoré môžu jeho kvalitu pri určitých optimalizačných problémoch znižovať. Tento problém by mal riešiť benchmarking, čo je metodológia porovnávania algoritmov a heuristík vzhľadom na určité výkonnostné kritériá. Je možné využiť radu kritérií a princípov, ako túto metodológiu realizovať. Jedným z cieľov tejto diplomovej práce preto bude využiť benchmarkingu na posúdenie a analýzu výkonnosti rady swarm algoritmov a následné zhodnotenie výsledkov.

2. OPTIMALIZÁCIA

Optimalizácia je dôležitý nástroj pri analýze systémov. Na využitie tohto nástroja musíme najprv identifikovať nejaký cieľ (objective), čo je kvantitatívna miera výkonnosti študovaného systému (quantitative measure). Týmto cieľom môže byť profit, čas, potenciálna energia alebo akákoľvek kombinácia kvantít, ktorá môže byť reprezentovaná jedným číslom.

Cieľ je závislý na určitých charakteristikách systému, ktoré sa nazývajú premenné (variables) alebo neznáme (unknowns) [1,2,3,4]. Úlohou je nájsť hodnoty týchto premenných, ktoré optimalizujú cieľ. Tieto premenné sú často obmedzené (constrained) v určitom zmysle.

Proces identifikácie cieľa, premenných a obmedzení pre daný problém je známy ako modelovanie. Vytvorenie správneho modelu je prvý a najdôležitejší krok v optimalizačnom procese. Neexistuje univerzálny optimalizačný algoritmus, ale kolekcie algoritmov, z ktorých každý je súci na určitý typ optimalizačného problému.

2.1 Optimalizačný problém

Matematicky je optimalizácia vlastne minimalizácia alebo maximalizácia funkcie, ktorá je podrobená obmedzeniam na jej premenných. Použijeme nasledujúci prepis:

- x je vektor premenných (rozhodovacích alebo nezávislých premenných), tzv. parametrov
- f je účelová funkcia (objective function), je to funkcia x, ktorú chceme maximalizovať alebo minimalizovať. Účelová funkcia je teda aplikovaná na rozhodovacie premenné.
- c_i sú funkcie obmedzení, ktoré sú skalárnymi funkciami x a definujú určité rovnosti a nerovnosti, ktoré musí neznámy vektor x spĺňať.

Pri tomto zápise môžeme formulovať optimalizačný problém nasledovne:

$$\min_{x \in R^n} f(x) \ podrobené\ obmedzeniam \begin{cases} c_i(x) = 0, \ i \in E, \\ c_i(x) \geq 0, \ i \in I. \end{cases} \quad (1)$$

Pričom I a E sú indexy obmedzení nerovnosti (inequality) a rovnosti (equality).

2.1.1 Rozdelenie optimalizačných problémov

Samotná klasifikácia optimalizačných problémov je možná z hľadiska niekoľkých kritérií:

1. V závislosti od charakteru prevedenia účelových funkcií a obmedzení
   - Konvexné problémy – funkcie obsiahnuté v probléme sú konvexné (akékoľvek lokálne optimum musí byť zároveň aj globálnym optimom).

     Nekonvexné problémy – funkcie obsiahnuté v probléme nie sú konvexné

- Lineárne problémy – funkcie obsiahnuté v probléme sú lineárneho charakteru

  Nelineárne problémy – funkcie obsiahnuté v probléme sú nelineárneho charakteru (môžu sem patriť napríklad kvadratické či geometrické programovanie)
- S obmedzením (constrained)

  Bez obmedzenia (unconstrained)

2. V závislosti od povolených hodnôt rozhodovacích premenných:
   - Celočíselné problémy (integer alebo discrete-valued) – jedná sa o diskŕetne optimalizačné problémy (pokiaľ nie sú premenné obmedzené nadobúdať celočíselné alebo binárne hodnoty, nazývajú sa tieto problémy aj MIP – mixed-integer problémy).
   - Reálne problémy (real-valued) – jedná sa o spojité optimalizačné problémy
3. V závislosti od deterministického charakteru premenných“
   - Deterministické problémy - dáta pre daný problém sú presne známe
   - Stochastické problémy – dáta pre daný problém sú náhodné
4. V závislosti od počtu účelových funkcií:
   - Jednokriteriálne problémy (single-objective)
   - Viackriteriálne problémy (multi-objective)

2.2 Testovacie funkcie

V prípade optimalizácie sa na evaluáciu charakteristík optimalizačných algoritmov používajú testovacie funkcie (optimalizačné testovacie problémy) [5,6]. Tradične je cieľom algoritmu nájsť minimum týchto testovacích funkcií. Podľa [7] môžeme deliť tieto problémy nasledovne:

1. Optimalizačné testovacie problémy s veľa lokálnymi minimami:
   - Ackleyho funkcia – je to spojitá, separabilná, multimodálna, nekonvexná testovacia funkcia, najčastejšie evaluovaná na $x_i \in [-32.768, 32.768]$, pre všetky $i = 1, \dots, d$, s globálnym minimom $f(x^*) = 0,$ v $x^* = (0, \dots, 0)$.
   - Rastriginova funkcia – je to spojitá, separabilná, multimodálna, nekonvexná testovacia funkcia, jej minimá sú regulárne rozmiestnené, najčastejšie evaluovaná na $x_i \in [-5.12, 5.12]$, pre všetky $i = 1, \dots, d$, s globálnym minimom $f(x^*) = 0,$ v $x^* = (0, \dots, 0)$.
   - Schwefelova funkcia – je spojitá, separabilná, multimodálna, nekonvexná testovacia funkcia, globálne minimum je geometricky vzdialené v priestore parametrov od najlepších lokálnych miním, čo môže zapríčiniť konvergenciu v zlom smere- Najčastejšie evaluovaná na $x_i \in [-500,\ 500]$ pre $i = 1, \dots, d$, s globálnym minimom v $f(x^*) = 0,$ v $x^* = (420.9687, \dots, 420.9687)$.

- Griewankova funkcia – je podobná Rastriginovej funkcií, s mohými lokálnymi regulárne distribuovanými minimami, evaluovaná je na intervale $x_i \in [-600,\ 600]$ pre všetky $i = 1, \dots, d$, s globálnym minimom $f(x^*) = 0, \text{v } x^* = (0, \dots, 0)$.
- Langermannova funkcia – je spojitá, nekonvexná, neseparabilná, multimodálna funkcia s nerovnomerne distribuovanými lokálnymi minimami. Funkcia je evaluovaná na $x_i \in [0{,}10]$ pre všetky $i = 1, \dots, d$, neobsahuje globálne minimum.
- Funkcia padajúcej vlny (Drop-Wave function) – je spojitá, nekonvexná, neseparabilná, multimodálna a vysoko komplexná, najčastejšie evaluovaná na intervale $x_i \in [-5.12, 5.12]$ pre všetky $i = 1, 2$, s globálnym minimom $f(x^*) = -1, \text{v } x^* = (0, 0)$.

2. Misovito tvarované (bowl-shaped):
   - Sférická funkcia (Sphere function) – je to konvexná, separabilná, unimodálna, spojitá funkcia, evaluovaná na $x_i \in [-5.12, 5.12]$ pre všetky $i = 1, \dots, d$, s globálnym minimom $f(x^*) = 0, \text{v } x^* = (0, \dots, 0)$. Funkcia má d lokálnych miním.
   - Funkcia súčtu štvorcov (Sum squares function) – taktiež nazývaná Osovo paralelná hyper-elipsoidná funkcia (Axis Parallel Hyper-Ellipsoid function), neobsahuje lokálne, ale iba globálne minimá. Je spojitá, konvexná, unimodálna a neseparabilná. Najčastejšie je evaluovaná na $x_i \in [-10, 10]$, no môže byť obmedzená aj na interval $x_i \in [-5.12, 5.12]$ pre všetky $i = 1, \dots, d$, s globálnym minimom $f(x^*) = 0, \text{v } x^* = (0, \dots, 0)$.
   - Bohachevského funkcie – jedná sa o tri spojité funkcie, všetky podobného misovitého tvaru, pričom sú evaluované na intervale $x_i \in [-100, 100]$, pre všetky $i = 1, 2\ \ 2$ s globálnymi minimami $f_j(x^*) = 0, \text{v } x^* = (0, 0)$ pre všetky $j = 1, 2, 3$. Prvá funkcia je unimodálna, konvexná a separabilná, druhá a tretia sú nekonvexné, multimodálne a neseparabilné.
3. Plátovo tvarované (plate-shaped):
   - Zakharova funkcia – je konvexná, spojitá, neseparabilná, unimodálna funkcia. Neobsahuje žiadne lokálne minimá, iba globálne. Najčastejšie je evaluovaná na intervale $x_i \in [-5, 10]$ pre všetky $i = 1, \dots, d$ s globálnym minimom $f(x^*) = 0, \text{v } x^* = (0, \dots, 0)$.
   - Boothova funkcia – je konvexná, spojitá, unimodálna a neseparabilná funkcia, ktorá je najčastejšie evaluovaná na intervale $x_i \in [-10, 10]$ pre všetky $i = 1{,}2$ s globálnym minimom $f(x^*) = 0, \text{v } x^* = (1, 3)$.
4. Dolinovo tvarované (valley-shaped):
   - Rosenbrockova funkcia – nazývaná tiež Rosenbrockova dolina, Banánová funkcia, či druhá funkcia De Jonga, je nekonvexná, unimodálna funkcia, ktorej globálne minimum leží v úzkej, parabolickej doline. Konvergencia k tomuto minimu je náročná, preto sa hodí pre gradientne založené optimalizačné

algoritmy. Najčastejšie býva evaluovaná na intervale $x_i \in [-5, 10]$, no môže byť obmedzená aj na $x_i \in [-2.048, 2.048]$, pre všetky $i = 1, \ldots, d$ s globálnym minimom $f(x^*) = 0$, v $x^* = (1, \ldots, 1)$.

- Funkcia šesťhrbej ťavy (Six-Hump Camel function) – jedná sa o spojitú, nekonvexnú, neseparabilnú a unimodálnu funkciu. Obsahuje šesť lokálnych miním, pričom dve z nich sú aj globálnymi. Evaluovaná býva najčastejšie na intervale $x_1 \in [-3, 3]$, $x_2 \in [-2, 2]$, s hodnotou globálneho minima $f(x^*) = -1.0316$, v $x^* = (0.0898, -0.7126)$ a $(-0.0898, 0.7126)$.

5. Strmé hrebene/spády (Steep Ridges/Drops):

- De Jongova funkcia č.5 – ide o nekonvexnú, spojitú, neseparabilnú a multimodálnu funkciu, ktorá je najčastejšie evaluovaná na intervale $x_i \in [-65.536, 65.536]$, pre všetky $i = 1,2$.
- Michalewiczova funkcia – je multimodálna, nekonvexná, separabilná a spojitá funkcia, ktorá obsahuje *d!* lokálnych miním. Jej strmosť dokážeme riadiť parametrom, čím uskutočníme hľadanie obtiažnejším. Funkcia je najčastejšie evaluovaná na intervale $x_i \in [0, \pi]$ pre všetky $i = 1, \ldots, d$, s globálnym minimom pri *d = 2:* $f(x^*) = -1.8013$, v $x^* = (2.20, 1.57)$,*d = 5:* $f(x^*) = -4.687658$ a *d = 10:* $f(x^*) = -9.66015$.

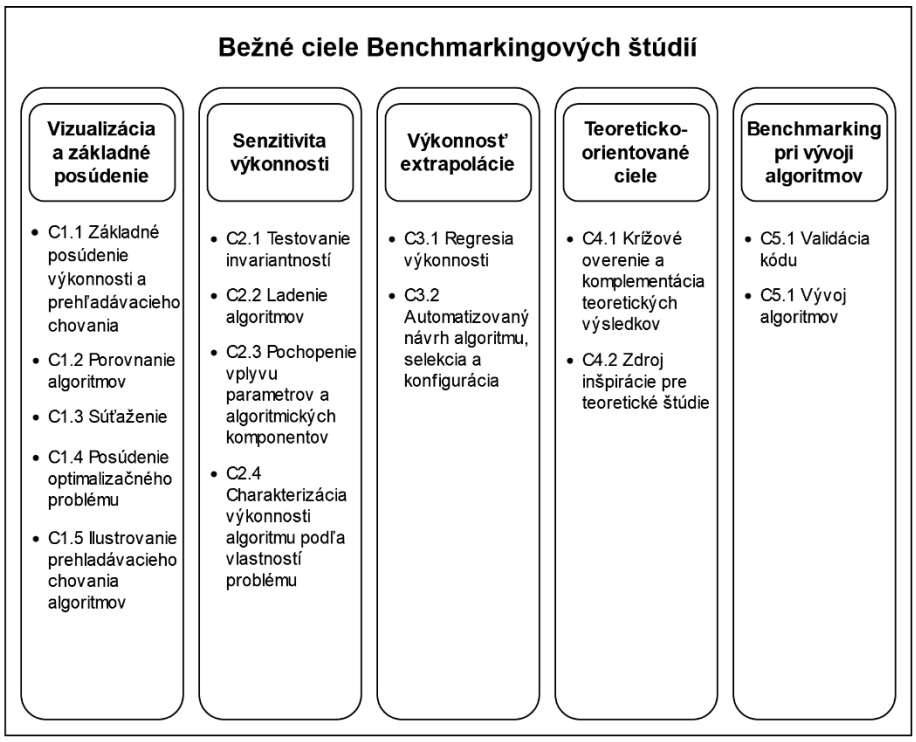
3. BENCHMARKING

Táto kapitola pojednáva o benchmarkingu, jeho cieľoch, vlastnostiach a častiach, z ktorých pozostáva. Zároveň popíše kritériá, poďla ktorých prebieha samotné meranie výkonnosti a predstaví niektoré benchmarkingové software.

3.1 Princíp benchmarkingu

V oblasti optimalizácie metaheuristík je experimentovanie jedným z najdôležitejších nástrojov porovnávania výkonnosti rôznych algoritmov. Benchmarking je metodológia ich porovnávania v závislosti od určitých kritérií [8,9]. Otázky, ktoré sa snaží benchmarking zodpovedať, vo všeobecnosti sú:

- Ako dobre určitý algoritmus pracuje na danom probléme
- Prečo algoritmus uspeje/zlyhá na špecifickom probléme



Obr.1 Ciele Benchmarkingu[8]

3.2 Benchmark

Počas výskumu v oblasti informačných technológií a počítačových vied je vyhodnotenie novej heuristiky alebo algoritmu primárne závislé na jeho výkonnosti v porovnaní s ostatnými možnými prístupmi. Testovanie tejto výkonnosti prebieha na reprezentatívnych dátach – tzv. *benchmark* [8,9]. Tieto dáta sú vo všeobecnosti

predgenerované alebo preddefinované. Benchmark, teda test set, je set využitý na porovnávanie výkonnosti alternatívnych nástrojov, techník, či algoritmov. Každá štúdia benchmarkingu musí pozostávať z troch aspektov:

- Voľba problému
- Voľba kritérií výkonnosti
- Voľba algoritmov

3.2.1 Vlastnosti benchmarku

Benchmark by mal oplývať nasledujúcimi vlastnosťami, ktoré zaisťujú využiteľnosť testovacieho setu pri experimentovaní [8,10]:

- Prístupnosť (Accessibility): Benchmark musí byť verejne prístupný a jednoduchý na aplikáciu, aby ho dokázal každý využiť. Pokiaľ je zrozumiteľný, existuje menšia šanca misinterpretácie.
- Dostupnosť (Affordability): Cena využívania benchmarku musí byť úmerná benefitom. Cenu je možné znížiť automatizáciou benchmarkingu. Pri vysokej cene a nízkych benefitoch je malá šanca využiteľnosti benchmarku.
- Priehľadnosť (Clarity): Špecifikácia benchmarku musí byť jasná a stručná.
- Riešiteľnosť a rôznorodosť (Solvability and diversibility): Musí byť možné dosiahnutie uspokojivého riešenia. Úloha, ktorá je príliš zložitá, neposkytne pre väčšinu nástrojov dáta na podporu porovnávania. Netriviálna a uskutočniteľná úloha dokáže vhodne poukázať aj na nedostatky systémov. Benchmark by teda mal byť dostatočne diverzný a obsahovať problémy v rozsahoch náročností.
- Škálovateľnosť a laditeľnosť (Scalability and tunability): Benchmark by mal poskytovať možnosť ladenia (tuning), teda nastavenia charakteristík problému. Ide napríklad o nastavenie dimenzií problému, úrovne závislosti medzi premennými alebo počet cieľov.

3.3 Benchmark sety

Pred započatím experimentu je nutné zvoliť vhodný dátový set, teda benchmark. Tento krok je jedným z troch aspektov benchmarknigu (voľba problému). Jeho výber závisí od samotného cieľu experimentu [8,12]. V rámci benchmarkingu a optimalizácie sú uskutočňované špeciálne súťaže a konferencie, ktoré sú zároveň výborným zdrojom benchmark setov. Dostupných setov je široké množstvo, príkladmi môžu byť:

- Umelé diskrétne optimalizačné problémy (Artificial discrete optimization problems): Napríklad Booleovská splniteľnosť (Boolean satisfiability), knižnica problémov Obchodného cestujúceho (TSP – Travelling salesperson problem [13] library), či Zmiešané celočíselné programovanie (Mixed integer programming library of problems).
- Umelé problémy s reálnymi parametrami (Artificial real-parameter problems)

- Umelé problémy so zmiešanou reprezentáciou (Artificial mixed representation problems): Napríklad Zmiešané binárne viackriteriálne problémy (Mixed-binary multi-objective problems) alebo Reálne zakódované viackriteriálne problémy (Real encoded multi-objective problems).
- Dynamické jednokriteriálne optimalizačné problémy (Dynamic single-objective optimization problems): Sem môžeme zaradiť napríklad benchmark problémy pre evolučné algoritmy.
- Nákladné optimalizačné problémy (Expensive optimization problems)
- S rušením (Noisy)
- Problémy s nezávislými komponentami (Problems with indenpendent components): Sem patrí napríklad Problém cestujúceho zlodeja (TTP – Travelling thief problem [14])
- Diskrétna optimalizácia z reálneho prostredia (Real-world discrete optimization): napríklad Mazda benchmark problem
- Numerická optimalizácia z reálneho prostredia (Real-world numerical optimization)

3.4 Algoritmus

Voľba algoritmu je ďalším aspektom benchmarkingu. Algoritmy by nemali byť zastarané, čo znamená, že je vhodné využívať aktuálne implementácie. Podľa informácií poskytovaných benchmarkingovým problémom je vhodné aplikovať algoritmy, ktoré tieto informácie dokážu účinne využiť [8]. Napríklad, pokiaľ sú nám známe informácie ako gradienty, môžeme využiť gradientne založené prehľadávanie (gradient-based seach). Pokiaľ prehľadávame zmiešané celočíselné rozhodovacie priestory (mixed-integer), je účelné využiť algoritmy, ktoré nie sú zamerané priamo na numerické a kombinatorické problémy. U algoritmov obsahujúcich hyperparametre je nutné ich správne nastavenie.

Dôležitým prvkom je tiež inicializácia počiatočného bodu pri prehľadávaní. Nie je vhodné využívať náhodnej inicializácie, nakoľko môže viesť k umiestneniu počiatočného bodu v blízkosti lokálneho optima niektorého z problémov, čo by znamenalo výhodu pre niektorý algoritmus. To znamená, že by mali všetky začínať v rovnakom počiatočnom bode. Algoritmy, inak nazývané solvery, môžu pochádzať z rozdielnych skupín:

- Jednorázové optimalizačné algoritmy (One-shot optimization algorithms): Napríklad Náhodné prehľadávanie (Random search).
- Hladové algoritmy s lokálnym prehľadávaním (Greedy local search algorithms): Napríklad Nelder-mead alebo Konjugovaný gradient (Conjugate gradient)
- Nehladové algoritmy lokálneho prehľadávania (Non-greedy local search algorithms): Napríklad Simulované žíhanie (Simulated annealing) či Tabu prehľadávanie (Tabu search).

- Algoritmy jednobodového globálneho prehľadávania (Single-point global search algorithms): Napríklad Evolučné stratégie (Evolution strategies) alebo Variabilné prehľadávanie susedstva (Variable neighbourhood seach).
- Populačne založené algoritmy (Population based algorithms): Napríklad Optimalizácia rojom častíc (Particle swarm optimization) či Optimalizácia mravčou kolóniou (Ant colony optimization) alebo akýkoľvek Swarm algoritmus.
- Algoritmy založené na nahradzovaní (Surrogate based algorithms): Sem môžeme zaradiť napríklad Algoritmus efektívnej globálnej optimalizácie (Efficient global optimization algorithm), Bayesovské optimalizačné algoritmy (Bayesian optimization algorithms)

3.5 Kritéria výkonnosti

Meranie výkonnosti je možné v zmysle niekoľkých kritérií. Dôležitosť metrík spočíva v ich analýze a štúdiu rôznych algoritmov, ktoré sú podstatné pre zistenie kvality ich výstupov a taktiež na porovnanie rôznych prístupov [8]. Dve najhlavnejšie sú kvalita riešenia (Fixed-target) a využitý rozpočet (Fixed-budget).

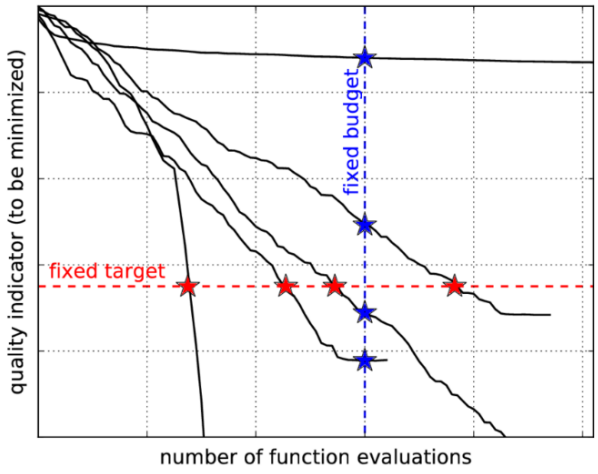
3.5.1 Fixed-target, Fixed-budget

Zjednodušene je možné tieto dve hlavné kritéria popísať nasledovne:

- Ako rýchlo dokáže algoritmus dosiahnuť kvality riešenia
- Akej kvality riešenia dokáže algoritmus dosiahnuť pri danom rozpočte

Jedná sa teda o horizontálne a vertikálne rezy v diagrame výkonnosti [8,9,15], ktoré je možné vidieť na obrázku číslo 2.

Vertikálny rez, inak nazývaný fixovaný rozpočet (fixed-budget), oplýva výhodou dobre definovaných výsledkov, pretože má limitovaný rozpočet. To znamená, že táto stratégia zastaví algoritmus v prípade dosiahnutia určitého počtu evaluácií účelovej funkcie, takže sme limitovaný počtom behov algoritmu.



Obr.2 Horizontálny/Vertikálny rez (Hodnota účelovej funkcie/ Počet evaluácií)[11]

Horizontálny rez, teda fixed-target alebo aj riešenie požadovanej kvality (desired quality solution) pracuje na princípe, pri ktorom stratégia zastaví algoritmus v prípade dosiahnutia určitej hodnoty účelovej funkcie. V tomto prípade môžeme vyvodiť záver v zmysle, že riešenie získané algoritmom *a* je lepšie, ako riešenie získané algoritmom *b.*

3.5.2 Kritérium merania času

Ďalším kritériom je meranie času. Najintuitívnejším prístupom je meranie s hodinami alebo s časom CPU. Inou možnosťou merania času je sledovanie počtu plne evaluovaných kandidátnych riešení [8]. Tento prístup je najpoužívanejší v prípade Kontinuálnej optimalizácie (Continuous optimization). Ide vlastne o meranie času prostredníctvom evaluácie funkcií. Alternatívou je využitie generácií ako meradla Času, napríklad pri genetických algoritmoch.

3.5.3 Kritérium merania kvality riešenia

Meranie kvality je možné popísať prostredníctvom viacerých kritérií. Voľba metriky kvality závisí tiež od samotného druhu problému [8,10,15]. Napríklad pri kontinuálnej optimalizácií je vhodný fitness, pri Probléme obchodného cestujúceho dĺžka cesty (tour length), pri strojovom učení (machine learning) je to presnosť klasifikácie a podobne.

Tento prístup však nie je veľmi praktický, pretože účelové hodnoty (objective values), sú silne špecifické pre jednotlivé druhy problémov. Najlepším riešením je, pokiaľ je známe optimálne riešenie, využiť absolútny alebo relatívny rozdiel riešenia s optimom nájdeným.

Ďalšou alternatívou je využitie spodného obmedzenia na normalizáciu výsledkov. Iný vhodný prístup zase predstavuje využitie heuristiky na normalizáciu.

V prípade fixed-budget je vhodné využiť na určenie kvality riešenia medián alebo geometrický priemer. Okrem spomínaného mediánu je užitočné sledovať aj aritmerický priemer a smerodajnú odchýlku funkčných hodnôt. V prípade fixed-target je nutné agregovať úspešné a neúspešné behy algoritmu. Na agregáciu výkonnosti je tiež možné využiť metriky ako ERT – Očakávaný čas behu (Expected Running Time), ktorý spočíta pomer medzi sumou spotrebovaného budgetu ponad všetky behy a počtom úspešných behov. Takto získame priemerný čas, ktorý algoritmus potrebuje na nájdenie riešenia požadovanej kvality. ERT je možne vyjadriť nasledovne:

$$ERT(A,f,d,B,v) = \frac{\sum_{i=1}^{r} \min\{T(A,f,d,B,v,i),B\}}{r\widehat{p_s}} = \frac{\sum_{i=1}^{r} \min\{T(A,f,d,B,v,i),B\}}{\sum_{i=1}^{r} 1(T(A,f,d,B,v,i)<\infty)} \quad (2)$$

Kde *A* sú algoritmy, *f* je funkcia, *d* je dimenzia, *B* je budget, *v* je požadovaná kvalita riešenia, *i* sú indexy vzoriek a *T* je miera (measure) fixed-target, teda počet evaluácií, ktoré algoritmus potrebuje na dosiahnutie riešenia určitej kvality. Empirická miera úspešnosti, ktorá je značená $\widehat{p_s}$, je zlomok behov algoritmu, v ktorých dosiahol požadovanej kvality riešenia *v*. Získa sa ako:

$$\widehat{p_s} = \sum_{i=1}^{r} \frac{1(T(A,f,d,B,v,i)<\infty)}{r} \quad (3)$$

V prípade TSP alebo Problému splniteľnosti (SAT – Satisfiability problem[16]) je vhodné využiť PAR – Penalizovaný priemerný čas (Penalized Average Runtime). Ten, ako názov napovedá, penalizuje neúspešné behy s násobkami maximálneho budgetu (násobkami dvoch, desiatich...) a následne spočíta aritmetický priemer spotrebovaného budgetu ponad všetky behy. Alternatívou je PQR – Penalizovaný kvantil času (Penalized Quantile Runtime), ktorý pracuje podobne, avšak namiesto aritmetického priemeru pre agregáciu využije medián. PQR je robustnejšia alternatíva k PAR.

$$PAR - c(v) = \frac{1}{r}\sum_{i=1}^{r} \min\{T(A, f, d, B, v, i), cB\} \quad (4)$$

V rovnici pre PAR majú premenné rovnaký význam ako pri ERT, pričom *cB* predstavuje neúspešné behy algoritmu, to sú tie, kde algoritmus *A* nedosiahol požadovanej kvality riešenia.

3.5.4 Kritérium merania robustnosti

Meranie robustnosti je závislé od rady faktorov [8, 10], Výsledky sú totiž vystavené neurčitosti, ako dôsledku troch hlavných príčin:

- Stochastické chovanie uvažovaného algoritmu
- Problémy s prítomnosťou rušenia (Noisy problémy)
- Členitosť povrchu problému (funkcie), čo v stručnosti znamená, či obsahuje veľa lokálnych optím

Prvá príčina spôsobuje rozdielnu výkonnosť v prípade opakovaného experimentu s rovnakým vstupom. Preto je vhodné previesť niekoľko behov, aby sme získali spoľahlivý odhad výkonnosti algoritmu.

V prípade real-world aplikácií je problémom výskyt rušenia (noise), čo rovnako spôsobuje rozdielny výstup pri využití rovnakého vstupu.

Vysoká členitosť funkcie môže byť problematická v prípade optimalizačného problému. Niekedy je teda výhodnejšie hľadať lokálne optimá namiesto globálneho, avšak len také, ktoré sa nachádzajú v jeho tesnej blízkosti.

3.5.5 Ďalšie kritériá

Meranie spoľahlivosti a rozšírenia využíva kritérium pravdepodobnosti odhadovaného úspechu [8,10,17]. To je zlomok behov algoritmu, ktorý dosiahol požadovaný cieľ. Fixed-probability je teda kritérium, ktoré pracuje s pravdepodobnosťou, že algoritmus dosiahol riešenie aspoň tak dobré, ako stanovený limit kvality v rámci daného budgetu evaluacií funckie.

Meranie rozptylu danej výkonnostnej metriky je prevedené skrz štatistické metódy ako smerodajná odchýlka a kvantil.

V prípade optimalizácie s obmedzeniami je možné aplikovať FR – mieru prípustnosti (feasibility rate), nakoľko riešenia sú pri nej iba prípustné a neprípustné. Tá je definovaná ako zlomok behov, ktoré objavia aspoň jedno prípustné riešenie. Je možné

tiež využiť počet obmedzení, ktoré boli prekročené mediánom riešení alebo priemer počtu prekročenia obmedzení skrz všetky najlepšie výsledky všetkých behov algoritmu.
Ďalšie metriky, ktoré je možné zvoliť, zahŕňajú napríklad:

- Meranie evolúcie parametrov
- Krivky pokroku (Progress curves)
- ECDF – Empirická distribučná funkcia (Empirical Cumulative Distribution Function), ktorá pre daný časový úsek (meranie), vráti zlomok behov, ktoré dosiahli daný cieľ chyby (goal-error), značený F_t (tradične sa uvažuje $F_t=0$).

3.6 Benchmarking software

V súčasnej dobe existuje veľké množstvo benchmarking software, ktoré umožňujú vyhodnocovanie výkonnosti prostredníctvom integrovaných kriteriálnych metód. Mnohé z nich zároveň poskytujú nástroje na analýzu či vizualizáciu [15,17,18]. Vďaka využitiu týchto systémov dokážeme znížiť záťaž pri experimentovaní, spôsobenú opakovaným behom algoritmu a generovaním veľkého množstva dát.

Ďalšou výhodou je automatizácia jednotlivých krokov benchmarkingu. Niektoré systémy umožňujú efektívne ladenie (tuning), teda nastavovanie parametrov. Dokážeme použiť napríklad SPOT na zvolenie správnych parametrov algoritmu a následne aplikovať COCO, ktoré nám umožní previesť numerickú optimalizáciu, zautomatizuje proces vyhodnocovania výkonnosti a zníži celkovú záťaž.
Mnohé z týchto algoritmov sú na sebe založené alebo sa využívajú vo vzájomnej kombinácií, aby bolo dosiahnuté čo najpriaznivejšieho výsledku.

3.6.1 Účel a rozdelenie platforiem

V nasledujúcich tabuľkách sú vymenované niektoré z platforiem spolu s ich uplatnením. Tieto systémy sú napísané a aplikovateľné primárne v jazykoch C/C++, Python a R. Niektoré, ako napríklad COCO, sú použiteľné aj pri Matlabe a iné, ako napríklad ECJ alebo MOEA sú špecificky založené na programovacom jazyku Java. IOHprofiler poskytuje internetovú verziu svojej platformy, do ktorej je možné nahrať dáta definovaného tvaru a využiť následne jeho nástroje na ich analýzu. Jednotlivé platformy je možné vidieť na tabuľkách číslo 1 a 2.

Tab.1 Benchmarkingové software

Software	Performance Assessment	Single-Objective	Multi-Objective	Mixed-Integer	Test Problems	Continuos Optimization	Discrete Optimization	Algorithm configuration
COCO[18]	✓	✓	✓	✓	✓	✓		
DEAP[24]	✓	✓	✓		✓	✓	✓	
ECJ[23]		✓	✓			✓		
Ecr[21]	✓	✓	✓			✓	✓	

Tab.1 Pokračovanie

Flacco[31]		✓				✓		
HeuristicLab[22]	✓	✓	✓			✓	✓	
IOHprofiler[10]	✓	✓	✓			✓	✓	
JCLEC[29]		✓	✓					
jMetal[30]			✓		✓	✓		
MOEA[32]			✓			✓		
Nevergrad[25]		✓				✓	✓	
Optproblems[26]		✓	✓		✓			
ParadisEO[33]		✓	✓			✓	✓	
PlatEMO[34]			✓		✓			
Platypus[27]	✓		✓		✓	✓		
Smoof[35]		✓	✓		✓			
ParamILS[36]								✓
Irace[28]								✓
SMAC[37]								✓

Špeciálnejšou kategóriou sú nástroje a systémy určené pre benchmarking v strojovom učení (ML – Machine learningu), poprípade ako už bolo spomenuté, na optimálne nastavenie parametrov, či už to modelu pri ML alebo algoritmov.

Tab.2 Benchmarkingové software pre ML

Software	Machine-learning	Model-based
Mlr[38]	✓	
mlrMBO[39]		✓
OpenML[40]	✓	
scikit-learn[41]	✓	
Shark[42]	✓	
SPOT[43]		✓

3.6.2 COCO

COCO alebo Comparing Continuous Optimizers je platforma, ktorá slúži na porovnávanie optimizerov [18,19,20]. Toto porovnanie sa deje v nastavení Black-box.

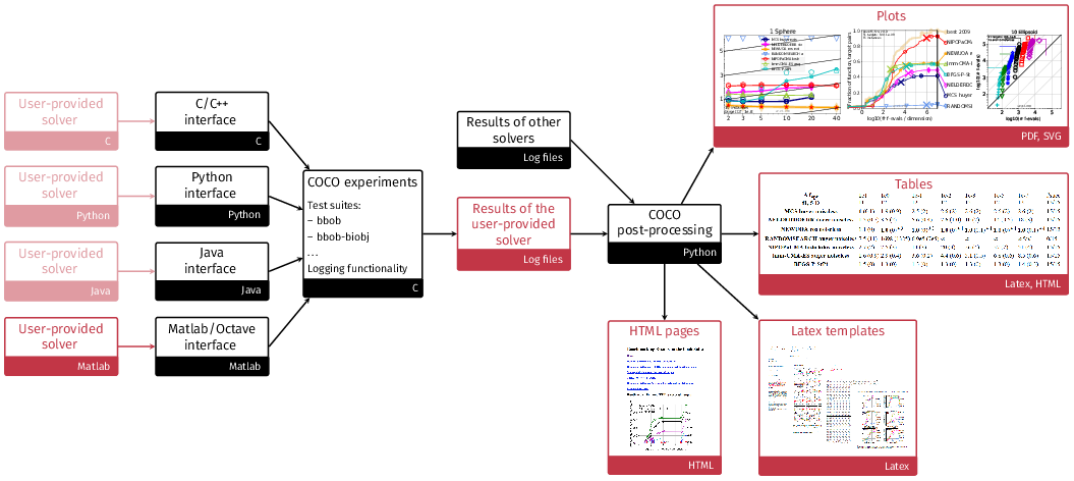
Black-box optimalizácia označujeme problém, pri ktorom optimalizačný algoritmus optimalizuje (napr. hľadá minimum) účelovú funkciu skrz tzv. Black-box

interface. To znamená, že algoritmus sa môže dotázať na hodnotu *f(x)* bodu *x*, ale nezíska informácie napríklad o gradiente a rovnako nevie vyvodiť závery analytickej formy *f* (napr. či je lineárna alebo kvadratická). Táto účelová funkcia je teda „uzavretá v black-boxe (čiernej skrinke)“. Cieľom optimalizácie je teda nájsť čo najlepšie riešenie, hodnotu *f(x)*, v preddefinovanom čase. Ten môže byť daný napríklad počtom prístupných dotazov k black-boxu.

COCO sa snaží čo najviac zautomatizovať repetetívnosť a zdĺhavosť benchmarkingu numerických optimalizačných algoritmov. COCO je budget-free, kritériom je počet volaní účelovej funkcie, za ktoré dosiahneme určitú kvalitu riešenia, označovaný aj *runtime*.

COCO framework poskytuje interface pre niekoľko jazykov, v ktorých môže byť optimizer napísaný. Ide o jazyky C/C++, Java, Matlab/Octave a Python. Zároveň poskytuje benchmarkové sety, ktoré sú napísané v jazyku C. Jedná sa napríklad o multi-objective sety *bbob-biobj*, ktorý obsahuje 92 účelových funkcií alebo o single-objective sety ako *bbob*, či *bbob-noisy*.

Výhodou COCO je, že využíva, ako už bolo spomenuté, iba jedno kritérium výkonnosti, ktorým je *runtime*. Toto kritérium je nezávislé od jazyka, compileru, či výpočtovej platformy. Zároveň je relevantné, ľahko interpretovateľné a dokáže nadobudnúť široké spektrum hodnôt. Ďalšou prednosťou je, že COCO nevyužíva preddefinovaný budget (budget-free). Jeho benchmarkové funkcie sú škálovateľné a zrozumiteľné a napriek tomu, že sú využité ako black-boxy pre algoritmus, sú vo vedeckej komunite explicitne známe.

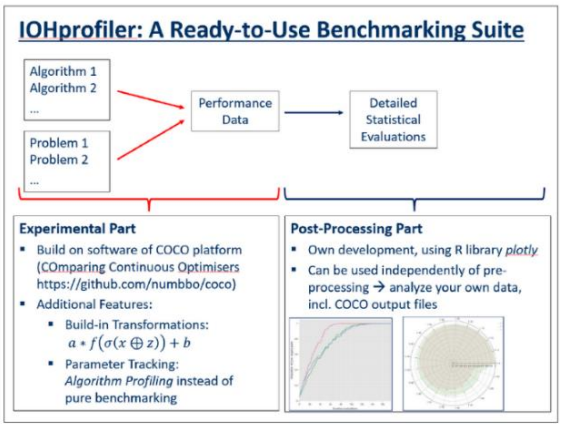


Obr.3 COCO Platforma[18]

3.6.3 IOHprofiler

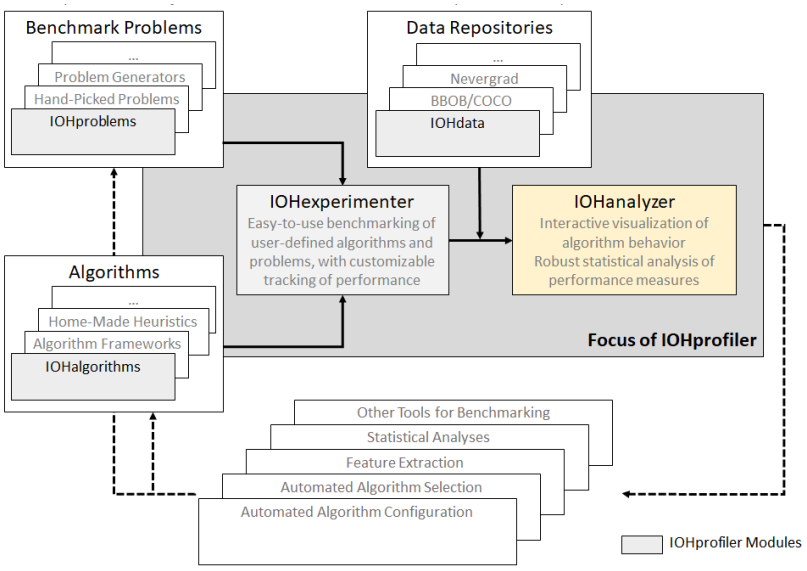
Táto benchmarkingová platforma slúži na vyhodnocovanie výkonnosti iteratívnych optimalizačných heuristík (IOHs – iterative optimization heuristics) ako sú napríklad evolučné alebo swarm algoritmy, ktoré hľadajú riešenia iteratívnym prehľadávaním [10,15]. IOHprofiler sa skladá z niekoľkých hlavných častí, ktorými sú:

- IOHexperimenter: Slúži na generovanie benchmarkových setov, produkuje dáta na experiment.
- IOHanalyzer: Poskytuje štatistickú analýzu a vizualizáciu dát, výkonnosť posudzuje z perspektívy fixed-target, fixed-budget a fixed-probability.
- IOHproblem: Zaisťuje kolekciu testovacích funkcií
- IOHdata: Umožňuje využitie benchmarkových dát z iných setov, ako z vlastného IOHexperimenter *PBO* (Pseudo Boolean Optimization) *benchmark suite* a to napríklad COCO *bbob* set, či nevergrad sety.
- IOHalgorithm: slúži na efektívnu implementáciu rady klasických optimalizačných algoritmov, ako sú Greedy Hill Climber, Randomized Local Search, Fast Genetic Algorithm alebo Evolution Algorithm with Static Mutation Rate.



Obr.4 Princíp IOHprofileru[10]

IOHprofiler pracuje s black-box optimalizačnými problémami alebo s Pseudo Boolean optimalizačnými problémami. Poskytuje tiež grafické rozhranie a interface pre jazyky C++, Python a R.



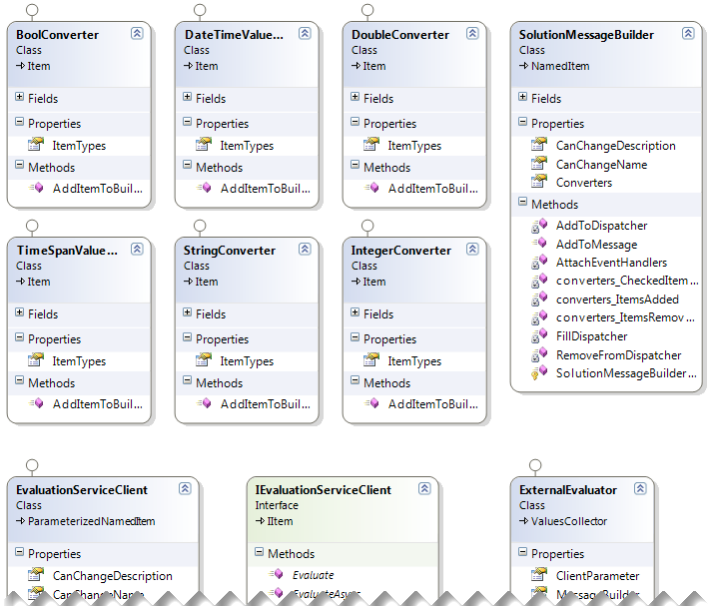
Obr.5 IOHprofiler platforma[15]

3.6.4 Nevergrad, ecr, HeuristicLab

Nevergrad je gradient-free platforma, slúžiaca na automatizáciu rutín benchmarkingu [21,22,24,25]. Umožňuje implementáciu optimalizačných algoritmov napísaných v jazyku Python a tiež parametrizáciu samotnej optimalizácie. Využíva kritéria fixed-budget.

Ecr alebo Evolutionary Computation in R, ako už z názvu plynie, je platforma umožňujúca implementáciu algoritmov v jazyku R. Inšpirovaná je DEAP-om, čo je ďalšia software platforma slúžiaca na benchmarking evolučných algoritmov napísaných v jazyku Python. Jedná sa o white-box framework, čo znamená, že rovnako ako pri DEAP je vývoj evolučného algoritmu transparentný. Obe, samozrejme, umožňujú aj black-box prístup. Výkonnostné kritéria využívané ecr sú napríklad epsilon indicator alebo hyper-volume indicator.

HeuristicLab je framework pre heuristické a evolučné algoritmy. Poskytuje grafické rozhranie GUI, v ktorom je možné algoritmy, ktoré sú reprezentované operátorovými grafmi, meniť priamo v rozhraní platformy bez nutnosti písania kódu. Jeho príklad je vidieť na obrázku číslo 6. Tento prístup je menej štandartný a celkom raritný v prípade benchmarkingových software. Navrhnutý je pre programovací jazyk C#.



Obr.6 Príklad rozhrania HeuristicLab platformy[22]

4. SWARM ALGORITMY

Biologicky inšpirované výpočty a rojová (swarm) inteligencia sa stali v posledných rokoch skutočne populárnymi technikami [44,45]. Vďaka svojej jednoduchosti a flexibilite sa swarm inteligencia využíva v mnohých moderných aplikáciách v rozličných disciplínach, kde tradičné metódy nie sú dostatočne presné alebo neprinášajú uspokojivé výsledky.

Samotná rojová inteligencia sa radí do väčšieho celku, ktorý predstavujú už spomínané biologicky inšpirované algoritmy. Vo všeobecnosti sa jedná o populačne založené (population based) algoritmy. To znamená, že populácia jedincov (potentionálny kandidáti na riešenia), spolupracuje medzi sebou a štatisticky sa zlepšuje počas generácií, čím eventuálne nájdu riešenie. Koncept swarm inteligencie zahŕňa multiplicitu, stochasticitu, náhodnosť a tiež neusporiadanosť.

4.1 Princíp chovania

Rojová inteligencia je súbor prírodou inšpirovaných algoritmov, ktoré sú založené na kolektívnom chovaní jednoduchých agentov. Tí postupujú podľa určitých pravidiel. Samotný agent môže byť považovaný za neinteligentného, avšak celý systém agentov preukazuje prvky samoorganizácie a decentralizácie a teda sa chová ako istý druh kolektívnej inteligencie.

Algoritmy v skupine swarm inteligencie pozostávajú primárne z dvoch fáz, ktoré udržiavajú rovnováhu medzi skúmaním a využívaním alebo vykorisťovaním (exploration a exploitation). Tak nútia celý roj (swarm), teda množinu možných riešení, aby upravovali svoje pozície. Ide o variačnú a selekčnú fázu, pričom variačná fáza skúma rozličné plochy prehľadávaného priestoru, kým selekčná fáza slúži na využívanie predchádzajúcich skúseností.

4.2 Samoorganizácia

Jedná sa o proces, pri ktorom pri interakcií medzi jedincami pôvodne neusporiadanej skupiny nastanú usporiadané pohyby. Je možné ju charakterizovať v štyroch stratégiách:

- Pozitívna spätná väzba (positive feedback): Informácia extrahovaná z výstupu systému je privedená na vstup systému, aby pomáhala formovaniu príslušných štruktúr. Pozitívna spätná väzba zaisťuje diverzitu v swarm inteligencií.
- Negatívna spätná väzba (negative feedback): Zaisťuje rovnováhu pozitívnej spätnej väzby a stabilizáciu kolektívneho vzoru. Negatívna spätná väzba predstavuje vykorisťovanie (exploitation) v rojovej inteligencií.
- Fluktuácie (fluctuations): Prislúchajú náhodnosti v systéme. Fluktuácie poskytujú nové situácie v procese riadenia a pomáhajú odstrániť stagnáciu.

- Viacnásobné interakcie (multiple interactions): Zaisťujú, že celková inteligencia roja (swarmu) je zlepšovaná učením sa z viac ako jedného jedinca v rámci spoločenstva

4.3 Particle Swarm Optimization

Optimalizácia rojom častíc (Particle Swarm Optimization), v skratke PSO, je swarm algoritmus, ktorý je inšpirovaný zhlukovaním vtákov alebo rýb. Táto inteligencia je založená na numerickom optimalizačnom algoritme predstavenom v roku 1995 pánmi Jamesom Kennedym a Russelom Eberhartom [46,47,48]. Sociálne živočíchy sa zhlukujú kvôli rôznym dôvodom, pričom kŕdle či zhluky sú vždy výhodné pre prežitie jedincov z niekoľkých dôvodov:

- Hľadanie potravy (Forgaging): Jedinci swarmu dokážu profitovať z objavov a predchádzajúcich skúseností ostatných jedincov pri hľadaní potravy.
- Obrana pred predátorom: Zhluk živočíchov (napr. kŕdeľ vtákov) sa dokáže účinnejšie brániť proti predátorovi. Viac očí znamená väčšiu šancu spozorovať nebezpečenstvo, množstvo potencionálnej koristi v kŕdli redukuje nebezpečenstvo hroziace jedincovi a skupina živočíchov je schopná zmiasť nepriateľa.

Toto chovanie samozrejme prináša aj nevýhody, ktoré však PSO nesimuluje (napr. slabší jedinci nezahynú, ako pri genetickom algoritme). Pri PSO všetci jedinci prežijú a snažia sa zosilnieť počas procesu hľadania. To znamená, že sa každá generácia pri swarm algoritmoch vylepšuje. Na navrhnutie PSO modelu bolo využitých 5 základných princípov:

1. Princíp blízkosti (Proximity principle): Populácia by mala byť schopná prevádzať jednoduché priestorové a časové výpočty.
2. Princíp kvality (Quality principle): Populácia by mala byť schopná reagovať na faktory kvality v prostredí.
3. Princíp rozdielnej reakcie (Diverse response principle): Populácia by sa nemala dopúšťať rovnakej reakcie.
4. Princíp stability (Stability principle): Populácia by nemala meniť svoj štýl chovania zakaždým, keď prebehne zmena prostredia.
5. Princíp prispôsobivosti (Adaptibility principle): Populácia by mala byť schopná zmeniť svoje chovanie, pokiaľ je to výhodné z hľadiska výpočetnej ceny.

Pri PSO je riešenie získané prostredníctvom náhodného prehľadávania (Random search), ktoré je vybavené swarm inteligenciou. To znamená, že PSO je swarm inteligentný prehľadávací algoritmus. Prehľadávanie je realizované náhodne generovanými potencionálnymi riešeniami. Táto kolekcia potencionálnych riešení sa nazýva roj (swarm) a každé potencionálne riešenia sa nazýva častica (particle).

Prehľadávanie je ovplyvnené dvoma spôsobmi učenia. Každá častica sa učí od inej častice, čo predstavuje tzv. sociálne učenie a zároveň sama zo svojich vlastných skúseností, tzv. kognitívne učenie. Vďaka sociálnemu učeniu teda častica získa najlepšie riešenia navštívené ktoroukoľvek časticou swarmu, nazývané *gbest* (global best).

Kognitívnym učením dospeje k najlepšiemu riešeniu, ktoré doposiaľ našla a navštívila každá častica sama, označovanému *pbest* (personal alebo particle best).

Každá častica má svoju vlastnú trajektóriu, nazývanú pozícia x_i a rýchlosť v_i, pričom sa pohybuje v prehľadávanom priestore postupným aktualizovaním svojej trajektórie. Populácie častíc upravujú ich trajektórie na základe pozícií *pbest* a *gbest*. Všetky častice majú *fitness* hodnotu, ktorá je vyhodnocovaná účelovou funkciou určenou na optimalizáciu. Častice sa pohybujú v priestore riešení nasledovaním optimálnych častíc. Algoritmus nainicializuje skupinu častíc na náhodných pozíciach a následne hľadá optimá aktualizáciou generácií. V každej iterácií je častica aktualizovaná na základe dvoch najlepších pozícií *pbest* a *gbest*. V *D*-dimenzionálnom priestore sú pozícia a rýchlosť *i*-tej častice v časovom kroku *t* reprezentovaná *D*-dimenzionálnym vektorom, $x_i^t = (x_{t1}^t, x_{i2}^t, \dots, x_{iD}^t)^T$ a $v_i^t = (v_{t1}^t, v_{i2}^t, \dots, v_{iD}^t)^T$. Predchádzajúce najlepšie riešenie navštívené *i*-tou časticou, v kroku *t* je vyjadrené vektorom $p_i^t = (p_{t1}^t, p_{i2}^t, \dots, p_{iD}^t)^T$. Najlepšia častica v swarme je označená indexom *g*. Na základe týchto tvrdení je aktualizácia rýchlosti a pozície vyjadrená nasledujúcimi rovnicami:

$$v_{id}^{t+1} = v_{id}^t + c_1 r_1 (p_{id}^t - x_{id}^t) + c_2 r_2 (p_{gd}^t - x_{id}^t) \qquad (5)$$

$$x_{id}^{t+1} = x_{id}^t + v_{id}^{t+1} \qquad (6)$$

Kde $d = 1,2,\dots, D$ reprezentuje dimenziu a $i = 1,2,\dots,S$ index častice, S predstavuje veľkosť swarmu, c_1 a c_2 sú konštanty nazývané kognitívne a sociálne škálovacie parametre alebo zjednodušene akceleračné koeficienty. Platí podmienka $c_1, c_2 > 0$, pričom koeficient c_1 reguluje maximálnu veľkosť kroku v smere osobnej najlepšej pozície častice a koeficient c_2 reguluje maximálnu veľkosť kroku v smere ku globálnemu najlepšiemu riešeniu. Parametre r_1 a r_2 sú náhodné čísla v rozmedzí [0,1] získané rovnomerným náhodným rozdelením (uniform random distribution). Aktualizácia je teda založená na troch záveroch:

1. Rýchlosť – bráni častici drasticky zmeniť jej smer
2. Kognitívna alebo egoistická zložka – súčasná pozícia je priťahovaná ku osobnému najlepšiemu riešeniu, čím bráni blúdeniu častice
3. Sociálna zložka – vďaka zdieľaniu informácií v roji je častica priťahovaná najlepším riešením swarmu.

```
Create and Initialize a D-dimensional swarm, S and corresponding velocity vectors ;
for t= 1 to the maximum bound on the number of iterations do
    for i=1 to S do
        for d=1 to D do
            Apply the velocity update equation 1;
            Apply position update equation 2;
        end
        Compute fitness of updated position;
        If needed, update historical information for pbest and gbest;
    end
    Terminate if gbest meets problem requirements;
end
```

Obr.7 PSO Pseudo kód[47]

Na obrázku 7 je možné vidieť pseudo kód PSO. Pretože sa všetky častice v swarme učia od *gbest* aj pokiaľ je ďaleko od globálneho optima, môžu byť jednoducho pritiahnuté do jeho okolia a uviaznuť v lokálnom optime v prípade multimodálnych problémov. Tento prípad môže nastať aj pokiaľ sú *gbest* umiestnené priamo na lokálnom optime. To znamená, že ak je včasné riešenie suboptimálne, môže swarm jednoducho stagnovať v jeho okolí. Aby sa zabránilo spomínanej situácií, bolo ako vylepšenie zavedené reseedovanie, teda čiastočný reštart. Ten vygeneruje nové častice v rozličných miestach prehľadávaného priestoru.

PSO dokáže nájsť oblasť optima rýchlejšie ako evolučné algoritmy, avšak keď sa do tohto regiónu dostane, jeho postup sa spomalí v dôsledku fixovanej veľkosti kroku rýchlosti. Varianta PSO s lineárne sa znižujúcimi váhami – LDWPSO (Lineary decreasing weigth PSO) efektívne vyvažuje globálne a lokálne schopnosti prehľadávania swarmu. Zavádza parameter nazývaný váha zotrvačnosti (inertia weigth), značený α, ktorý typicky graduálne klesá z hodnoty α_{MAX} po α_{MIN}. Týmto parametrom je násobená okamžitá rýchlosť častice v_{id}^{t} v rovnici (5). Jeho hodnota sa upravuje podľa nasledujúceho predpisu:

$$\alpha(t) = \alpha_{MAX} - (\alpha_{MAX} - \alpha_{MIN})\frac{t}{T} \qquad (7)$$

Kde *T* je maximálny počet iterácií. Ďalšou variantou môže byť CenterPSO, ktorá prináša tzv. stredovú časticu (center particle) do varianty LDWPSO. Tá je aktualizovaná ako centrum roja v každej iterácií. Napriek tomu, že táto častica nemá rýchlosť je zahrnutá v každej operácií rovnako, ako štandartné častice (napr. v evaluácií fitness, súťažení o najlepšiu časticu...) - okrem výpočtu rýchlosti. Všetky častice oscilujú okolo centra roja a graduálne k nemu konvergujú. Stredová Častica sa často stane *gbest* riešením celého swarmu a teda ovplyvňuje výkonnosť. CenterPSO dosahuje lepšie riešenia a aj rýchlejšiu konvergenciu v porovnaní s LDWPSO. Pokiaľ sú v PSO na rýchlosť aplikované medze namiesto zotrvačnosti, vznikne varianta PSO s obmedzením (constriction PSO). Tá aplikuje obmedzujúci faktor χ, ktorý riadi veľkosť rýchlosti:

$$v_{id}^{t+1} = \chi\{v_{id}^{t} + \phi_1 r_1 (p_{id}^{t} - x_{id}^{t}) + \phi_2 r_2 (p_{gd}^{t} - x_{id}^{t})\} \qquad (8)$$

$$\chi = \frac{2}{\left|2 - \varphi - \sqrt{\varphi^2 - 4\varphi}\right|} \qquad (9)$$

V tejto rovnici musí platiť podmienka $\varphi = \varphi_1 + \varphi_2 > 4$. Vďaka tejto formulácií nemusíme limitovať maximálnu rýchlosť častíc a algoritmus dokáže garantovať konvergenciu. PSO s obmedzením poskytuje lepšiu konvergenciu ako LDWPSO, avšak je náchylnejšie na uviaznutie v lokálnom optime pri multimodálnych funkciách. Iné varianty PSO zahŕňajú Bare-Bones PSO, využívajúce Gaussovo rozdelenie alebo FDR-PSO (Fitness-distance-ratio-based), využívajúce dodatočné informácie o blízkej častici s vysokým fitness.

Particle Swarm Optimization je možné aplikovať na prakticky akýkoľvek problém súvisiaci s numerickou optimalizáciou. Ide o algoritmus, ktorý je flexibilný na modifikácie a práve preto z neho, či z jeho variánt čerpá inšpiráciu mnoho ďalších swarm algoritmov.

4.4 Artificial Bee Colony

Algoritmus, skrátene označovaný ABC, bol prvýkrát uverejnený v roku 2005 pánom Dervisom Karabogom. Je založený na modeli kolónie včiel [49,50], ktorý pozostáva z niekoľkých hlavných zložiek:

- Zdroje potravy (Food sources): Aby včela zvolila zdroj potravy, musí vyhodnotiť jeho vlastnosti, ako sú vzdialenosť od úľa, chuť nektáru, výdatnosť energie a miera zložitosti jej extrakcie. Pre zjednodušenie môžeme tieto parametre reprezentovať pod jednou kvantitou.
- Zamestnaní zberači (Employed foragers): Tieto včely sú zamestnané špecifickým zdrojom potravy, ktorý práve využívajú. Nesú o ňom informáciu a zdieľajú ju s ostatnými včelami v úli. Skrátene tieto včely nazývame zberači.
- Nezamestnaní zberači (Unemployed foragers): Tieto včely hľadajú zdroj potravy. Môže ísť o prieskumníka (scout), ktorý prehľadáva okolie náhodne (pomocou náhodného pohybu – random walk) alebo pozorovateľa (onlooker), ktorý sa snaží nájsť zdroj potravy na základe informácií poskytnutých zberačmi.

To znamená, že v modeli sa nachádzajú tri skupiny včiel, pričom výmena informácií medzi nimi je realizovaná prostredníctvom tanca a prebieha na tanečnej ploche. Polovica kolónie je tvorená zo zamestnaných včiel – zberačov, kým druhá polovica je tvorená pozorovateľmi. Na každý zdroj potravy pripadá jedna zamestnaná včela. Takže počet riešení je rovný počtu zdrojov potravy v okolí úľa. Pozícia zdroja potravy predstavuje možné riešenie problému a zodpovedá fitness-u riešenia:

$$fit_i = \frac{1}{1+f_i} \qquad (10)$$

Pričom f_i je hodnota účelovej funkcie. ABC najskôr vygeneruje náhodne rozloženú počiatočnú populáciu *P(C=0) SN* riešení, kde každé riešenie z_i je reprezentované *D*-dimenzionálnym vektorom, pričom *i=1,2,...,SN*. Po inicializácií je populácia vystavená opakovaným cyklom *C=1,2,...,MCM* (maximum cycle number) procesu hľadania zberačmi, pozorovateľmi a prieskumníkmi.

Zberač prevádza zmenu pozície riešenia v jeho pamäti v závislosti od lokálnej informácie a testuje fitness (množstvo nektáru) nového zdroja (nového riešenia). Pokiaľ je fitness nového riešenia vyšší, ako predchádzajúceho uloženého v pamäti, včela si túto pozíciu zapamätá a starú ňou prepíše. V opačnom prípade si pozíciu predchádzajúceho riešenia uchová v pamäti.

Po ukončení procesu prehľadávania všetkými zberačmi zazdieľajú zberači svoje informácie o zdrojoch potravy a ich pozíciách s pozorovateľmi na tanečnej ploche. Pozorovateľ ohodnotí informácie od všetkých zberačov a zvolí čo najvhodnejší zdroj potravy, pričom rovnako ako zberač, prevedie zmenu pozície vo svojej pamäti v závislosti na množstve nektáru (fitness) nového zdroja.

Pozorovateľ volí tento zdroj podľa pravdepodobnostnej hodnoty p_i, prislúchajúcej zdroju potravy. Tá sa získa ako:

$$p_i = \frac{fit_i}{\sum_{n=1}^{SN} fit_n} \qquad (11)$$

V tejto formulácií *SN* predstavuje počet zdrojov potravy a fit_i je fitness riešenia inverzne proporčný f_i, teda hodnote účelovej funkcie problému. Na vytvorenie kandidátnej pozície potravy $v_{i,j}$ s využitím starej pozície, aplikuje ABC vzťah:

$$v_{ij} = z_{ij} + \phi_{ij}(z_{ij} - z_{kj}) \qquad (12)$$

Kde $k \in \{1,2,\dots,SN\}$ a $j \in \{1,2,\dots D\}$ sú náhodne zvolené indexy, pričom platí $k \neq i$, a ϕ_{ij} je náhodné číslo z intervalu [-1,1], ktoré ovláda produkciu susedných zdrojov potravy v okolí zdroja z_i a reprezentuje porovnanie dvoch pozícií potravy, ktoré sú včele viditeľné. Nevyužitý zdroj potravy je nahradený novým, pričom táto nová pozícia je vytvorená náhodne. Za nevyužitú sa považuje pozícia, ktorá nedokáže byť vylepšená počas preddefinovaného počtu cyklov, nazývaného *limit*:

$$z_{min}^j = z_{min}^j + rand(0,1)(z_{max}^j - z_{min}^j) \quad (13)$$

To znamená, že pokiaľ je zdroj z_i nevyužitý a $j \in \{1,2,\dots D\}$, tak prieskumník objaví nový zdroj potravy, ktorým z_i nahradí. Po vytvorení novej kandidátnej pozície v_{ij} je jej výkonnosť porovnávaná so starou.

ABC má tri ovládacie (control) parametre, ktorými sú *SN* (počet včiel – zberačov alebo pozorovateľov), *limit* a maximálny počet cyklov (iterácií) *MCM*.

1: Load training samples
2: Generate the initial population z_i $i = 1 \dots SN$
3: Evaluate the fitness (f_i) of the population
4: set cycle to 1
5: **repeat**
6: **FOR** each employed bee{
Produce new solution v_i by using (6)
Calculate the value f_i
Apply greedy selection process}
7: Calculate the probability values p_i for the solutions (z_i) by (5)
8: **FOR** each onlooker bee{
Select a solution z_i depending on p_i
Produce new solution v_i
Calculate the value f_i
Apply greedy selection process}
9: **If** there is an abandoned solution for the scout **then** replace it with a new solution which will be randomly produced by (7)
10: Memorize the best solution so far
11: cycle=cycle+1
12: **until** cycle=MCN

Obr.8 ABC Pseudo kód [51]

4.5 Antlion Optimization

Tento algoritmus bol prezentovaný po prvýkrát v roku 2015 pánom S. Mirjalilim. Inšpirovaný je mechanizmom, ktorý mrakolevy (antlion) používajú pri love. Larvy tohoto hmyzu vyhrabú pascu – kužeľovitú dieru, do ktorej chytajú korisť [52,53]. Algoritmus ALO sa zameriava na interakciu medzi mrakolevmi a ich korisťou – mravcami. Na realizáciu tejto interakcie je nutné, aby sa mravce pohybovali v prehľadávanom priestore, pričom mrakolevy ich lovia prostredníctvom pascí. Tak sa stávajú vhodnejšími (fittier).

Model je teda založený na niekoľkých princípoch, ktoré sú aplikované počas optimalizácie:

- Mravce sa pohybujú prehľadávaným priestorom s využitím náhodného prechádzania (Random Walk).
- Náhodné prechádzanie je aplikované na všetky dimenzie mravcov
- Náhodné prechádzanie je ovplyvnené pascami mrakolevov
- Mrakolevy budujú svoje pasce proporcionálne ich fitnessu (čím je lepší, tým väčšia je pasca)
- Mrakolevy s väčšími pascami majú vyššiu šancu chytiť mravca
- Každý mravec môže byť chytený mrakolevom v každej iterácií a tiež elitou (najvhodnejšími mrakolevami – s najlepším fitnessom)
- Rozsah náhodného prechádzania sa adaptívne znižuje na simulovanie zosúvania sa (smerom do diery) mravcov ku mrakolevom
- Pokiaľ sa mravec stane vhodnejším (fittier) ako mrakolev, znamená to, že je chytený a stiahnutý mrakolevom pod piesok (do pasce)
- Mrakolev sa presunie na posledné miesto kde chytil korisť a vybuduje novú pascu, aby zlepšil svoje šance chytenia ďalšej koristi po každom love

Mravce sa pohybujú stochasticky pomocou náhodného prechádzania pri hľadaní potravy, pričom tento pohyb dokážeme formulovať prostredníctvo nasledujúcej rovnice:

$$X(t) = [0, cumsum(2r(t_1) - 1), cumsum(2r(t_2) - 1), \dots, cumsum(2r(t_n) - 1)\ ] \tag{14}$$

Kde *cumsum* spočíta kumulatívnu sumu, *n* je maximálny počet iterácií, *t* predstavuje krok náhodného prechádzania a *r(t)* je stochastická funkcia definovaná ako:

$$r(t) = \begin{cases} 1\ if\ rand > 0.5 \\ 0\ if\ rand \leq 0.5 \end{cases} \tag{15}$$

Pričom *t* opäť predstavuje krok náhodného prechádzania a *rand* je náhodné číslo generované rovnomerným rozdelením v intervale [0,1]. Pozície mrakolevov sú uložené a využívané počas optimalizačného procesu prostredníctvom matice:

$$M_{ant} = \begin{bmatrix} ant_{1,1} & \cdots & ant_{1,d} \\ \vdots & \ddots & \vdots \\ ant_{n,1} & \cdots & ant_{n,d} \end{bmatrix} \tag{16}$$

Pozícia mravca predstavuje riešenie. Matica (16) obsahuje pozíciu každého mravca. Účelová funkcia je využitá počas optimalizácie, pričom hodnoty fitness každého mravca sú uložené v nasledujúcej matici:

$$M_{oa} = \begin{bmatrix} F_t([ant_{1,1}, ant_{1,2}, \dots ant_{1,d}]) \\ \vdots \\ F_t([ant_{n,1}, ant_{n,2}, \dots ant_{n,d}]) \end{bmatrix} \tag{17}$$

Mrakolevy sú umiestnené („schované“) v prehľadávanom priestore. Na uloženie ich pozícií a fitness sa využijú matice:

$$M_{antlion} = \begin{bmatrix} antlion_{1,1} & \cdots & antlion_{1,d} \\ \vdots & \ddots & \vdots \\ antlion_{n,1} & \cdots & antlion_{n,d} \end{bmatrix} \tag{18}$$

$$M_{oal} = \begin{bmatrix} F_t([antlion_{1,1}, antlion_{1,2}, \dots antlion_{1,d}]) \\ \vdots \\ F_t([antlion_{n,1}, antlion_{n,2}, \dots antlion_{n,d}]) \end{bmatrix} \quad (19)$$

Mravce menia svoje pozície podľa rovnice (14). Aby bolo ich náhodné prechádzanie udržané v medziach prehľadávaného priestoru, je ich pohyb normalizovaný:

$$X_i^t = \frac{(X_i^t - a_i) \times (d_i^t - c_i^t)}{(b_i - a_i)} + c_i^t \quad (20)$$

Kde a_i, b_i sú minimum a maximum náhodného prechádzania *i*-tej premennej, d_i^t, c_i^t je minimum a maximum *i*-tej premennej na iterácií *t*. Keďže náhodné prechádzanie mravcov je ovplyvnené pascami mrakolevov, je nutné formulovať nasledujúce rovnice:

$$c_i^t = Antlion_j^t + c^t \quad (21) \qquad d_i^t = Antlion_j^t + d^t \quad (22)$$

Kde c^t, d^tpredstavujú minimum a maximum všetkých premenných na *t*-tej iterácií a c_i^t, d_i^t sú minimum a maximum všetkých premenných pre *i*-teho mravca. $Antlion_j^t$ označuje *j*-teho mrakoleva na *t*-tej iterácií. Mrakolevy sú schopné postaviť pasce proporcionálne ich fitness-u. Keď zacítia, že sa mravec chytil do pasce, začnú hádzať piesok smerom k centru pasce (tvaru kužeľa). Toto chovanie je modelované rovnicami:

$$c^t = \frac{c^t}{I} \quad (23) \qquad d^t = \frac{d^t}{I} \quad (24)$$

Pričom *I* je klzný pomer, ktorý je možné určiť ako $I = 10^{\omega} \frac{t}{T}$, kde *t* je súčasná iterácia, *T* je maximálny počet iterácií a ω je konštanta závislá na súčasnej iterácií ($\omega = 2$, keď $t > 0.1T$, $\omega = 2$, keď $t > 0.1T$, $\omega = 3$, keď $t > 0.5T$, $\omega = 4$, keď $t > 0.75T$, $\omega = 5$, keď $t > 0.9T$ a $\omega = 6$, keď $t > 0.95T$).

Antlion Optimization Algorithm:

Input: Fitness function, ants and antlions, maximum iteration number, population size.
Output: The elite antlion position and its fitness value.

1) Initialize antlions' positions.
2) Calculate fitness values of antlions by using objective function.
3) Sort fitness values and save best antlion.
4) **while** $(iteration < Max\ iteration)$ and $(|f_{best} - f_{worst}| < VTR)$
   for every ant
      a) Select antlion by roulette wheel method for building trap.
      b) Slide randomly walking ants in trap.
      c) Create random walk for every ants around elite antlion and selected antlion.
      d) Normalize random walks (Eq.(3)).
      e) Update the ant position (Eq.(9)).
      f) Reposition the ants in case of outside search space.
   end for
   g) Calculate the fitness values of ants.
   h) Concatenate fitness of ants and antlions.
   i) Update antlions' positions.
   j) Save elite antlions' position and fitness value.
5) **end while**

Obr.9 ALO Pseudo kód[54]

Posledné štádium lovu predstavuje stiahnutie mravca do piesku a jeho skonzumovanie. Ak mal mravec lepší fitness ako mrakolev, zmení mrakolev svoju pozíciu na pozíciu, kde mravca ulovil:

$$Antlion_j^t = Ant_j^t \quad if \ f\left(Ant_j^t\right) > f(Antlion_j^t) \quad (25)$$

Pri algoritme je aplikovaný elitizmus, čo je udržiavanie najlepšieho jedinca nažive. Najlepší mrakolev by mal byť udržiavaný ako elita a mal by byť schopný ovplyvniť pohyb všetkých mravcov počas iterácií. To znamená, že platí predpoklad, že každý mravec náhodne prechádza okolo mrakoleva zvoleného prostredníctvom ruletového kolesa (roulette-wheel) a elity:

$$Ant_j^t = \frac{R_A^t + R_E^t}{2} \tag{26}$$

V tejto formulácií je R_A^t náhodné prechádzanie v okolí mrakoleva zvoleného ruletou na *t*-tej iterácií a R_E^t je náhodné prechádzanie v okolí elity na *t*-tej iterácií.

4.6 Cuckoo Search

Tento metaheuristický algoritmus bol vynájdený v roku 2009 pánmi Xin-She Yang-om a Suash Deb-om [55,56]. CS je založený na parazitizme niektorých druhov kukučiek pri kladení vajec do cudzích hniezd. Jeho model pracuje na troch prinípoch:

- Každá kukučka znesie len jedno vajce a uloží ho v náhodne zvolenom hniezde
- Najlepšie hniezda s najvyššou kvalitou vajíčok (riešení) budú prenesené do ďalšej generácie
- Počet hostiteľských hniezd je fixovaný, navyše hostiteľ dokáže objaviť cudzie vajíčko s pravdepodobnosťou $P_a \in [0,1]$. V takom prípade môže hostiteľský vták vajce buď zahodiť alebo hniezdo opustiť a vybudovať nové na inej pozícií.

Posledný princíp môže byť prevedený tak, že časť p_a z n hostiteľských hniezd bude nahradená novými hniezdami (novými náhodnými riešeniami). Kvalita alebo fitness riešenia môže byť proporcionálna hodnote účelovej funkcie.

Základná implementácia pracuje s reprezentáciou, že každé vajce v hniezde predstavuje jedno riešenie a každá kukučka znesie iba jedno vajce. Cieľom je využiť nové a potencionálne lepšie riešenia na nahradenie nevyhovujúcich riešení. Algoritmus môže byť samozrejme rozšírený do varianty, kde každé hniezdo obsahuje viac vajíčok (teda set riešení).

Algoritmus využíva vyváženú kombináciu lokálneho a globálneho náhodného prechádzania (Local Random Walk, Global Random Walk) na prehľadávanie. Táto rovnováha je riadená prepínacím parametrom $p_a \in [0,1]$. Globálne prehľadávanie je realizované prostredníctvom Lévyho letov (Lévy flights [57]), čo je forma náhodného prechádzania. Rovnice popisujúce lokálne (27) a globálne (28) prehľadávanie sú formulované nasledovne:

$$x_i^{t+1} = x_i^t + \alpha s H(p_a - \varepsilon)(x_j^t - x_k^t) \tag{27}$$

$$x_i^{t+1} = x_i^t + \alpha L(s, \lambda) \tag{28}$$

Kde x_j^t a x_k^t sú dve rozdielne riešenia zvolené náhodne prostredníctvom náhodnej permutácie, *H(u)* je Heavisideova funkcia (jednotkový skok) [58], ε je náhodné číslo získané rovnomerným rozdelením (uniform distribution), s je veľkosť kroku a α je škálovací faktor veľkosti kroku, pričom platí podmienka $\alpha > 0$. Jeho hodnota prislúcha

mieram záujmu. V prípade globálneho prehľadávania (28), $L(s,\lambda)$ reprezentuje Lévyho rozdelenie [59], ktoré slúži na zadefinovanie veľkosti kroku náhodného prechádzania. Lévyho let je pre algoritmus matematicky modelovaný prostredníctvom nasledujúcej formulácie:

$$L(s,\lambda) = \frac{\lambda\Gamma(\lambda)\sin(\pi\lambda/2)}{\pi}\frac{1}{s^{1+\lambda}}, \qquad (s \gg s_0 > 0) \qquad (29)$$

Pričom Γ predstavuje gamma rozdelenie a λ je konštanta, pre ktorú platí $1 < \lambda \leq 3$. Vo väčšine prípadov sa volí $\alpha = O(\frac{L}{10})$, kde L značí charakteristickú mieru (scale) problému, avšak v niektorých prípadoch $\alpha = O(\frac{L}{100})$ môže byť efektívnejšou voľbou, ktorá zabráni príliš ďalekému letu (Lévyho). Vďaka využívaniu Lévyho letov namiesto štandardného náhodného prechádzania, dokáže preskúmať priestor riešení veľmi efektívne, nakoľko Lévyho lety majú nekonečný priemer a varianciu. Ich ďalšia výhoda spočíva v tom, že generujú podstatnú časť nových riešení dostatočne ďaleko od pozície súčasného najlepšieho riešenia, čím zabraňujú uviaznutiu v lokálnom optime.

Cuckoo Search Algorithm

```
begin
  Objective function f(x), x=( x_1,..., x_d)^T;
  Initial a population of n host nests x_i (i=1,2,...,n);
    while (t < Maximum Generation) or (stop criterion);
      Get a cuckoo (say i) randomly
                and generate a new solution by Lévy flights;
      Evaluate its quality/fitness; F_i
      Choose a nest among n (say j ) randomly;
      if (F_i > F_j),
          Replace j by the new solution;
      end
      Abandon a fraction (P_a) of worse nests
          [and build new ones at new locations via Lévy flights];
       Keep the best solutions (or nests with quality solutions);
      Rank the solutions and find the current best;
    end while
    Post process results and visualization;
end
```

Obr.10 CS Pseudo kód[60]

4.7 Dragonfly Algorithm

Tento optimalizačný algoritmus bol predstavený v roku 2016 pánom S. Mirjalilim [61,62]. Inšpirovaný je prirodzeným dynamickým (migráciou) a statickým (lovom) rojovým chovaním vážok. Na vytvorenie modelu rojového chovania vážok bolo využitých päť primitívnych princípov:

1. Separácia (Separation): Je stratégia, ktorú vážky využívajú na vyhýbanie sa kolíziám medzi sebou. Matematicky je modelovaná ako:

$$S_i = -\sum_{j-1}^{M} P - P_j \qquad (30)$$

Kde P je pozícia súčasného jedinca, P_j predstavuje pozíciu j-teho susedného jedinca a M počet susediacich jedincov.

2. Usporiadanie (Alignment): Ukazuje, ako agent nastaví svoju rýchlosť vzhľadom na rýchlosť ostatných pridružených susedných vážok. Vyjadrené je rovnicou:

$$A_i = \frac{\sum_{j-1}^{M} V_j}{M} \tag{31}$$

Kde V_j je vektor rýchlosti j-teho jedinca.

3. Súdržnosť (Cohesion): Indikuje inklináciu jedincov k pohybu v smere do centra roja. Formulovaná je ako:

$$C_i = \frac{\sum_{j-1}^{M} P_j}{M} - P \tag{32}$$

4. Príťažlivosť (Attraction): Predstavuje náchylnosť jedincov k pohybu v smere ku zdroju potravy. Modelovaná je ako príťažlivosť medzi zdrojom potravy a i-tym agentom:

$$F_i = F_p - P \tag{33}$$

Pričom F_p je pozícia zdroja potravy.

5. Rozptýlenie (Distraction): Značí tendenciu vážok vyhýbať sa konfliktu s nepriateľom. Matematicky ju formulujeme ako rozptýlenie medzi nepriateľom a i-tou vážkou:

$$E_i = E_p + P \tag{34}$$

Kde E_p označuje pozíciu nepriateľa.

Chovanie vážok je tvorené kombináciou týchto piatich predpokladov. V DA (ďalšia používaná skratka je aj DAO), sú fitness zdroja potravy a vektory pozícií aktualizované na základe doposiaľ najlepšieho agenta. Navyše, fitness a pozícia nepriateľa sú spočítané na základe najhoršej vážky. Vďaka tejto skutočnosti sa DA dokáže vyhýbať nepriaznivej oblasti.

Vektory pozície vážok sú aktualizované na základe dvoch pravidiel – vektor kroku ΔP a vektor pozície P. Vektor kroku vyjadruje smer pohybu vážky. Spočítaný je prostredníctvom rovnice:

$$\Delta P_i^{t+1} = (sS_i + aA_i + cC_i + fF_i + eE_i) + \omega \Delta P_i^t \tag{35}$$

Pričom s je váha separácie, S_i značí separáciu i-teho jedinca, a je váha usporiadania, A_i je usporiadanie i-teho jedinca, c je váha súdržnosti a C_i je súdržnosť i-teho jedinca, f je faktor potravy, F_i je zdroj potravy i-teho jedinca, e je faktor nepriateľa, E_i je pozícia nepriateľa i-teho jedinca, ω je váha zotrvačnosti a t je iterácia. Po spočítaní vektora kroku, sú spočítané vektory pozícií:

$$P_i^{i+1} = P_i^t + \Delta P_i^{t+1} \tag{36}$$

Konvergencia DA môže byť riadená prostredníctvom váh s, a, c, f, e a ω. Na vylepšenie prehľadávania a procesu aktualizácie riešení je možné aplikovať náhodné prechádzanie vo forme Lévyho letov. Kombináciou všetkých popisovaných princípov a techník je možné zostaviť pseudo kód, ktorého príklad je na obrázku 11.

Algorithm 1 The pseudo-code of the Dragonfly Algorithm

Initialize the population of dragonflies $P_i (i = 1, 2, \ldots, m)$ randomly
Initialize the step vectors $\Delta P_i (i = 1, 2, \ldots, m)$
while $(t < Maximum\ number\ of\ iterations)$ **do**
    Calculate the fitness of each dragonfly $f(P_i)$
    Update the source food and enemy
    Update w, s, a, c, f, and e
    Calculate S, A, C, F, and E using equations (1) to (5)
    Update neighboring radius
    if a dragonfly has at least one neighboring dragonfly **then**
        Update the velocity vector ΔP_i^{t+1} using equation (6)
        Update the position vector P_i^{t+1} using equation (7)
    else
        Update P_i^{t+1} using equation (8)
    end if
    Check and correct the new positions based on the boundaries of variables
end while

Obr.11 DA Pseudo kód[61]

4.8 Bat Algorithm

Tento algoritmus bol navrhnutý v roku 2010 pánom Xin-She Yangom na základe pozorovania chovania a charakteristík mikronetopierov [63,64]. Sformulované boli tri základné pravidlá:

1. Všetky netopiere využívajú echolokáciu na vnímanie vzdialenosti a taktiež poznajú rozdiel medzi potravou (korisťou) a prekážkami.
2. Netopiere lietajú náhodne s rýchlosťou v_i na pozícií x_i s frekvenciou pulzov f (alebo vlnovou dĺžkou λ) a hlasitosťou pulzov A_0 aby hľadali potravu. Automaticky dokážu upraviť vlnovú dĺžku (alebo frekvenciu) ich vysielaných pulzov a tiež mieru vysielania pulzov (pulse rate emision), značenú r ($r \in [0, 1]$), v závislosti na proximite ich cieľa.
3. Napriek tomu, že hlasitosť dokáže variovať, predpokladáme, že dosahuje hodnoty od vysokej (pozitívnej) A_0 až po minimálnu konštantnú A_{min}.

Tieto pravidlá je možné popísať nasledujúcimi matematickými formuláciami:

$$f_i = f_{min} + (f_{max} + f_{min}) * \beta \qquad (37)$$

$$v_i^t = v_i^{t-1} + (x_i^t - x_{best}) * f_i \qquad (38)$$

$$x_i^t = x_i^{t-1} + v_i^t \qquad (39)$$

V týchto rovniciach platí, že každý *i*-ty netopier je asociovaný s rýchlosťou v_i^t a pozíciou x_i^t na iterácií *t*, v *d*-dimenzionálnom priestore riešení. Medzi všetkými netopiermi existuje najlepšie riešenie v populácií x_{best}. Frekvencia f je využívaná netopierom pri hľadaní koristi, jej indexy *min* a *max* reprezentujú minimálnu a maximálnu hodnotu. β je náhodný vektor, ktorý je získaný prostredníctvom rovnomerného rozdelenia (uniform distribution), pričom pre jeho hodnotu platí $\beta \in [0, 1]$. Miera vysielania pulzov je značená r_i, kde i je index *i*-teho netopiera. V každej iterácií je vygenerované náhodné číslo, ktoré je porovnávané s r_i. Pokiaľ je väčšie, zahájí sa stratégia lokálneho

prehľadávania (realizovaná náhodným prechádzaním – random walk). Touto stratégiou získame nové riešenie pre netopiera, reprezentované pozíciou generovanou podľa rovnice náhodného prechádzania:

$$x_{new} = x_{old} + \varepsilon A^t \qquad (40)$$

Kde $\varepsilon \in [-1, 1]$ je náhodné číslo a A^t reprezentuje priemernú hlasitosť všetkých netopierov v súčasnej iterácií. Po aktualizácií pozície všetkých netopierov sa hlasitosť A a miera vysielania pulzov r_i aktualizujú iba za podmienky, že je nové najlepšie riešenie aktualizované (teda nová pozícia je výhodnejšia ako stará) a náhodne vygenerované číslo je menšie ako A. To znamená, že pokiaľ netopier našiel jeho korisť, hlasitosť sa bude znižovať a miera vysielania pulzov sa bude zvyšovať. Môžeme teda predpokladať, že $A_{min}=0$, ak netopier našiel zdroj potravy a teda dočasne prestal vysielať akýkoľvek zvuk. Aktualizácia hlasitosti a miery vysielania pulzov je vyjadrená rovnicami:

$$A_i^{t+1} = \alpha * A_i^t \qquad (41)$$

$$r_i^{t+1} = r_i^0[1 - e^{-\gamma t}] \qquad (42)$$

V týchto rovniciach α a γ sú konštanty, pričom α môžeme prirovnať k ochladzovaciemu faktoru pri simulovanom žíhaní (Simulated annealing), teda predstavuje potláčanie hlasitosti. Všeobecne by mali platiť podmienky, že pre akékoľvek $0 < \alpha < 1$ a $\gamma > 0$: $A_i^t \to 0, r_i^t \to r_i^o$ pri $t \to 0$.

Bat Algorithm

Objective function $f(\mathbf{x})$, $\mathbf{x} = (x_1, \ldots, x_d)^T$
Initialize the bat population $\mathbf{x}_i$ $(i = 1,2, \ldots, n)$ *and* $\mathbf{v}_i$
Define pulse frequency f_i *at* $\mathbf{x}_i$
Initialize pulse rates r_i *and the loudness* A_i
while *(t <Max number of iterations)*
  Generate new solutions by adjusting frequency,
  and updating velocities and locations/solutions [equations (2) to (4)]
  if *(rand* $> r_i$*)*
    Select a solution among the best solutions randomly
    Generate a local solution around the selected best solution by a local random walk
  end if
  if *(rand* $< A_i$ *&* $f(\mathbf{x}_i) < f(\mathbf{x}_*)$*)*
    Accept the new solutions
    Increase r_i *and reduce* A_i
  end if
  Rank the bats at each iteration and find their current best $\mathbf{x}_*$
end while
Postprocess results and visualization

Obr.12 Pseudo kód BA[65]

4.9 Elephant Herding Optimization

Algoritmus bol navrhnutý pánom G. Wangom v roku 2015, pričom biologickú inšpiráciu čerpá zo sociálneho chovania slonov, ktoré žijú v spoločenstvách [66,67]. Model EHO je založený na troch idealizovaných pravidlách:

1. Populácia slonov je tvorená klanmi, každý klan má pevne daný počet slonov – samíc a samcov.
2. Pevný počet sloních samcov opustí rodinnú skupinu v každej generácií a žije osamotene, ďaleko od hlavnej skupiny slonov.
3. Slony v každom klane žijú spolu pod vedením matriarchy – hlavnej slonice

Náhodne vytvorená počiatočná populácia slonov je rozdelená do fixovaného počtu klanov a zoradená podľa fitnessu. Následne je každý klan aktualizovaný osobitne v aktualizačnom procese, v ktorom každý slon nesie tzv. aktualizačný operátor. Ten uvádza, že pozícia každého slona v klane je ovplyvnená matriarchou. To znamená, že pozícia *j*-teho slona v klane *ci* môže byť aktualizovaná ako:

$$x_{new,ci,j} = x_{ci,j} + \alpha \left(x_{best,ci} - x_{ci,j}\right) r \qquad (43)$$

Pričom $x_{new,ci,j}$ a $x_{ci,j}$ sú nová a stará pozícia pre slona *j* v klane *ci*, $\alpha \in [0,1]$ je miera, ktorá udáva vplyv matriarchy *ci* na slona $x_{ci,j}$, $x_{best,ci}$ predstavuje matriarchu *ci*, ktorá je najlepším sloním jedincom v klane *ci* a $r \in [0,1]$ môže byť napríklad rovnomerné alebo Gaussovo rozdelenie. Najlepší slon (s najlepším fitness) v každom klane nemôže byť aktualizovaný prostredníctvom rovnice (43). Na jeho úpravu bola sformulovaná nasledujúca rovnica:

$$x_{new,ci,j} = \beta x_{center,ci} \qquad (44)$$

Kde $\beta \in [0,1]$ je faktor určujúci vplyv $x_{center,ci}$ na $x_{new,ci,j}$. Pozícia nového jedinca $x_{new,ci,j}$ v rovnici (44) je generovaná prostredníctvom informácií získaných všetkými slonmi v klane *ci*. Centrum klanu *ci* je značené $x_{center,ci}$ a môže byť spočítané nasledovne:

$$x_{center,ci} = \frac{1}{n_{ci}} \sum_{j=1}^{n_{ci}} x_{ci,j} \qquad (45)$$

V tejto rovnici n_{ci} predstavuje počet slonov v klane *ci*. To znamená, že pozícia matriarchy je aktualizovaná pozíciami všetkých slonov v klane. V procese separácie samce opustia skupinu a ostanú žiť osamotene. V EHO algoritme platí predpoklad, že najhorší slon (s najhorším fitness), v každom klane, je nahradený prostredníctvom separačného operátoru. Ten je formulovaný ako:

$$x_{worst,c} = x_{min} + rand\ (x_{max} - x_{min} + 1) \qquad (46)$$

Kde x_{min} a x_{max} sú dolný a horný limit pozícií jednotlivých sloních jedincov, $x_{worst,c}$ je najhorší slon v klane *ci* a $rand \in [0,1]$ je číslo získané Gaussovským alebo rovnomerným rozdelením.

```
Initialization:
Initialize (Maximum generation, Population size, Boundaries).
Initialize the population.
Calculate elephant's fitness.
Repeat
    Sort all the elephants according to their fitness.
    Clan updating:
    For c= 1 to nClan (for all clans in elephant population) do
        For j= 1 to nci (for all elephants in clan c) do
            If x_{c,j}=x_{best,ci} then
                Update x_{c,j} (old elephant) and generate x_{n,c,j} (new elephant) by Eq. (2).
            Else
                Update x_{c,j} (old elephant) and generate x_{n,c,j} (new elephant) by Eq. (1).
            End if
        End for j
    End for c
    Separating operator:
    For c= 1 to nClan (all the clans in elephant population) do
        Replace the worst elephant in clan c by Eq. (4).
    End for c
    Evaluate population by the newly updated positions.
Until (Maximum number of generation)
```

Obr. 13 Pseudo kód EHO[68]

EHO algoritmus je teda riadený dvoma operátormi:

- Aktualizačný operátor (Updating operator), vyjadrený rovnicami (43),(44),(45)
- Separačný operátor (Separation operator), vyjadrený rovnicou (46)

4.10 Grey Wolf Optimization

Tento algoritmus bol navrhnutý pánmi S. Mirjalilim a A. Lewisom v roku 2014 [69,70]. GWO je založený na základe sociálnej hierarchie vlka obyčajného (*Canis lupus*). Matematicky model uvažuje najlepšie riešenie (optimum) ako alfu (α). To znamená, že druhé a tretie najlepšie riešenie bude podľa vlčej hierarchie beta (β) a delta (δ). Ostatné riešenia sú považované za omegy (ω). Optimalizácia (lov) je v GWO vedená týmito troma vlkmi, teda α, β a δ. Omega vlky ich nasledujú. Algoritmus pracuje na nasledujúcich princípoch:

- Obkľúčenie koristi (Encyrcling prey): Pri love vlky najskôr obkľúčia korisť. Matematicky je toto chovanie modelované rovnicami:

$$\vec{D} = \left|\vec{C} * \overrightarrow{X_p}(t) - \vec{X}(t)\right| \quad (47)$$

$$\vec{X}(t+1) = \overrightarrow{X_p}(t) - \vec{A} * \vec{D} \quad (48)$$

Kde *t* predstavuje súčasnú iteráciu, $\overrightarrow{X_p}$ je vektor pozície koristi, $\vec{X}$ je vektor pozície vlka a $\vec{A}$ a $\vec{C}$ su vektory koeficientov, ktoré sú spočítané nasledovne:

$$\vec{A} = 2\vec{a} * \overrightarrow{r_1} - \vec{a} \quad (49)$$

$$\vec{C} = 2 * \overrightarrow{r_2} \quad (50)$$

Pričom $\vec{a}$ je komponenta lineárne klesajúca od 2 po 0 v rámci iterácií a $\overrightarrow{r_1}$ a $\overrightarrow{r_2}$ sú náhodné vektory (vektory náhodných čísel), v rozsahoch [0,1].

Pozícia vlka môže byť aktualizovaná v závislosti na pozícií koristi. Rozličné pozície v okolí najlepšieho agenta. Rozličné pozície v okolí najlepšieho agenta v zmysle súčasnej pozície je možné získať úpravou hodnôt vektorov $\vec{A}$ a $\vec{C}$. Náhodné vektory zase umožňujú vlkovi dosiahnuť akúkoľvek pozíciu v priestore.

- Lov (Hunting): Vlky alfa, beta a delta majú najlepšie informácie o potencionálnej koristi. To znamená, že sa tieto tri najlepšie riešenia uložia a ostatní agenti (vrátane omega vlkov) sa zaviažu aktualizovať svoje pozície v závislosti na pozícií najlepších agentov. V tomto zmysle sú navrhnuté nasledujúce formulácie:

$$\overrightarrow{D_\alpha} = \left|\overrightarrow{C_1} * \overrightarrow{X_\alpha} - \vec{X}\right|, \overrightarrow{D_\beta} = \left|\overrightarrow{C_2} * \overrightarrow{X_\beta} - \vec{X}\right|, \overrightarrow{D_\delta} = \left|\overrightarrow{C_3} * \overrightarrow{X_\delta} - \vec{X}\right| \quad (51)$$

$$\overrightarrow{X_1} = \overrightarrow{X_\alpha} - \overrightarrow{A_1} * (\overrightarrow{D_\alpha}), \overrightarrow{X_2} = \overrightarrow{X_\beta} - \overrightarrow{A_2} * (\overrightarrow{D_\beta}), \overrightarrow{X_3} = \overrightarrow{X_\delta} - \overrightarrow{A_3} * (\overrightarrow{D_\delta}) \quad (52)$$

$$\vec{X}(t+1) = \frac{\overrightarrow{X_1} + \overrightarrow{X_2} + \overrightarrow{X_3}}{3} \quad (53)$$

Z toho vyplýva, že alfa, beta a delta odhadnú pozíciu koristi a ostatné vlky aktualizujú svoje pozície náhodne okolo nej.

- Zaútočenie na korisť (Attacking prey): Matematicky sa približovanie ku koristi modeluje znižovaním hodnoty $\vec{a}$. Pri jej znižovaní dochádza tiež k znižovaniu fluktuačného rozsahu $\vec{A}$. To znamená, že $\vec{A}$ bude náhodná veličina v intervale [*-2a*, *2a*],

kde a klesá od 2 po 0. Pri hodnote $\vec{A}$ v rozsahu [-1,1] bude ďalšia aktualizovaná pozícia medzi súčasnou pozíciou agenta a pozíciou koristi. Takže pokiaľ platí $|A| < 1$, zaútočia vlky v ústrety koristi.

- Hľadanie koristi (Search for prey): Proces hľadania začne s vytvorením náhodnej populácie vlkov (kandidátnych riešení). Každé kandidátne riešenie aktualizuje svoju vzdialenosť od koristi. Parameter $\vec{a}$ je znižovaný od 2 po 0 v zmysle zdôraznenia útoku a hľadania. Kandidátne riešenia divergujú od koristi v prípade, že $|A| > 1$ a ako už bolo spomenuté v predchádzajúcom princípe, konvergujú k nej pokiaľ $|A| < 1$. Hľadanie je teda realizované konvergenciou/divergenciou vlkov voči sebe, aby zaútočili na korisť. Parameter C prikladá náhodnú váhu koristi. Jeho implementácia je užitočná pri prehľadávaní a zabraňuje stagnácií v lokálnom optime, predovšetkým vo finálnych iteráciách. Na obrázku 14 je možné vidieť popisované princípy vo forme pseudo kódu.

```
Initialize the grey wolf population X_i (i = 1, 2, ..., n)
Initialize a, A, and C
Calculate the fitness of each search agent
X_α=the best search agent
X_β=the second best search agent
X_δ=the third best search agent
while (t < Max number of iterations)
    for each search agent
            Update the position of the current search agent by equation (3.7)
    end for
    Update a, A, and C
    Calculate the fitness of all search agents
    Update X_α, X_β, and X_δ
    t=t+1
end while
return X_α
```

Obr.14 Pseudo kód GWO[69]

4.11 Moth Flame Optimization

Jedná sa o prírodou inšpirovaný optimalizačný algoritmus predstavený v roku 2015 pánom S. Mirjalilim [71,72]. Založený je na priečnej orientácií, ktorú mole aplikujú pri navigácií počas lietania v noci. Tento mechanizmus spočíva v udržiavaní fixovaného uhla voči Mesiacu, čo umožňuje prekonať dlhú vzdialenosť pri zachovávaním priameho smeru. Pokiaľ moľa príde do styku s umelým svetlom, snaží sa aplikovať rovnaký postup voči tomuto zdroju svetla, no blízka vzdialenosť zapríčiní, že uviazne v špirálovej slučke. Algoritmus pozostáva z nasledujúcich krokov:

- Vytvorenie počiatočnej populácie molí: Pri MFO platí predpoklad, že kandidátne riešenia sú mole. Tie môžu lietať v *1D, 2D, 3D* alebo hyperdimenzionálnom priestore. Set molí a ich fitness sú vyjadrené prostredníctvom matíc:

$$M = \begin{bmatrix} m_{1,1} & \cdots & m_{1,d} \\ \vdots & \ddots & \vdots \\ m_{n,1} & \cdots & m_{n,d} \end{bmatrix} \quad (54) \qquad OM = \begin{bmatrix} OM_1 \\ \vdots \\ OM_n \end{bmatrix} \quad (55)$$

Kde (54) predstavuje maticu molí a (55) predstavuje maticu fitness každej mole. Ten získame predaním vektoru pozície mole (riadku z (54)) účelovej funkcií. V rámci týchto matíc *n* predstavuje počet molí a *d* predstavuje počet premenných (dimenzií). Ďalším dôležitým parametrom sú ohne (flames). Ich matice sú podobné maticiam vyjadrujúcim mole:

$$F = \begin{bmatrix} F_{1,1} & \cdots & F_{1,d} \\ \vdots & \ddots & \vdots \\ F_{n,1} & \cdots & F_{n,d} \end{bmatrix} \quad (56) \qquad OF = \begin{bmatrix} OF_1 \\ \vdots \\ OF_n \end{bmatrix} \quad (57)$$

Kde *n* a *d* majú rovnaký význam, ako pri maticiach (54) a (55). Obe mole aj plamene sú riešenia. Rozdiel spočíva v zaobchádzaní s nimi a ich aktualizáciou v každej iterácií. Mole sú agenti, ktorí sa pohybujú priestorom a prehľadávajú, pričom plamene sú doposiaľ najlepšie získané pozície molí. Takže plamene je možné považovať za vlajočky, ktorú moľa zhodí pri prehľadávaní. Každá moľa teda prehľadáva v okolí vlajočky (plameňa) a aktualizuje svoju pozíciu v prípade, že nájde lepšie riešenie. Vďaka tomuto mechanizmu moľa nikdy nestratí jej najlepšie riešenie.

- Moth Flame Optimization využíva tri funkcie na zaistenie konvergencie ku globálnemu optimu:

$$MFO = (I, P, T) \quad (58)$$

Pričom $I: \theta \rightarrow \{M, OM\}$ je funkcia generujúca náhodnú populáciu molí a im prislúchajúcich hodnôt fitness. Funkcia $P: M \rightarrow M$ je hlavná funkcia, zabezpečujúca pohyb molí priestorom. To znamená, že zabezpečuje aktualizáciu matice (54). Napokon funkcia $T: M \rightarrow \{true, false\}$ vráti *pravda* (*true*), pokiaľ je terminujúce kritérium splnené, v opačnom prípade vráti *nepravda* (*false*). Nasledujúca rovnica predstavuje *I* funkciu, ktorá je využitá na implementáciu náhodného rozdelenia:

$$M(i,j) = \big(ub(i) - lb(i)\big) * rand() + lb(i) \quad (59)$$

Kde *lb* a *ub* indikujú dolný a horný limit premenných. Ako už bolo spomenuté na začiatku, mole sa pohybujú priestorom prostredníctvom priečnej orientácie. Ako mechanizmus aktualizácie pozície mole je využitá logaritmická špirála. Pri jej využití musia byť dodržané tri podmienky:

1. Počiatok špirály by mal začínať od mole
2. Koniec špirály by mal byť pozícia plameňa
3. Fluktuácia rozsahu špirály nesmie presiahnuť z prehľadávaného priestoru

Potom bude logaritmická špirála definovaná nasledovne:

$$S\big(M_i, F_j\big) = D_i * e^{bt} * \cos(2\pi t) + F_j \quad (60)$$

V tejto rovnici je *b* konštanta pre definíciu tvaru logaritmickej špirály, *t* je náhodné číslo z intervalu [-1,1] a D_i predstavuje vzdialenosť medzi *i*-tou moľou M_i a *j*-tým plameňom F_j, definovaná ako:

$$D_i = \big|F_j - M_i\big| \quad (61)$$

Uviaznutiu v lokálnom optime je predídené vďaka udržiavaniu optimálnych riešení pri každom opakovaní iterácie a tiež vďaka moliam lietajúcim okolo plameňov s využitím matíc (55) a (57).

- Aktualizácia počtu plameňov: Aktualizácia pozície molí na n rozličných pozíciách v priestore môže zdegradovať využitie najlepších potencionálnych riešení. Na vyriešenie tohto problému je zavedený adaptívny mechanizmus pre znižovanie počtu plameňov počas iterácií:

$$flame\ no = round(N - l * \frac{N-1}{T}) \tag{62}$$

Kde N je maximálny počet plameňov, l je číslo súčasnej iterácie a T značí maximálny počet iterácií. Na obrázku 15 je znázornený pseudo kód algoritmu.

Algorithm 1 Moth-flame optimization algorithm

```
Initialize the parameters for Moth-flame
Initialize Moth position M_i randomly
for i = 1 to n do
   Calculate the fitness function f_i
end for
while iteration ⩽ Max_iterations do
   Update the position of M_i
   Calculate the number of flames using Eq.()
   Evaluate the fitness function f_i
   if iteration ==1 then
      F=sort(M) and OF=sort(OM)
   else
      F=sort(M_{t-1}, M_t) and OF=sort(M_{t-1}, M_t)
   end if
   for i = 1 to n do
      for j = 1 to d do
         Update the values of r and t
         Calculate the value of D respect to its corresponding moth using Eq.()
         Update M(i,j)respect to its corresponding moth using Eq.()
      end for
   end for
end while
Print the best solution
```

Obr.15 Pseudo kód MFO[73]

4.12 Whale Optimization Algorithm

Veľryby (konkrétne keporkaky) využívajú pri love špeciálnu techniku lovu nazývanú útok bublinovou sieťou [74,75]. Algoritmus WOA bol predstavený v roku 2016 pánmi S. Mirjalilim a A. Lewisom, ktorý previedli túto metódu lovu do matematického modelu a optimalizačného algoritmu. Model pozostáva z troch častí:

1. Obkľúčenie koristi: Keporkaky určia polohu koristi a obkľúčia ju. Cieľová korisť je najlepšie kandidátne riešenie. Ostatné veľryby (agenti) sa budú snažiť vylepšiť svoju pozíciu smerom k najlepšiemu agentovi (veľrybe, ktorá našla najlepšiu korisť). Tento princíp je popísaný rovnicami:

$$\vec{D} = \left|\vec{C} * \vec{X}^*(t) - \vec{X}(t)\right| \tag{63}$$

$$\vec{X}(t+1) = \vec{X}^*(t) - \vec{A} * \vec{D} \tag{64}$$

V týchto rovniciach t predstavuje súčasnú iteráciu, $\vec{A}$ a $\vec{C}$ sú vektory koeficientov, $\vec{X}^*$ predstavuje vektor pozície doposiaľ najlepšieho získaného riešenia a $\vec{X}$ je vektor pozície. Je dôležité, aby bolo $\vec{X}^*$ aktualizované v každej iterácií, pokiaľ existuje lepšie riešenie. Vektory $\vec{A}$ a $\vec{C}$ sa spočítavajú podľa nasledujúcich vzťahov:

$$\vec{A} = 2\vec{a} * \vec{r} - \vec{a} \tag{65}$$

$$\vec{C} = 2 * \vec{c} \tag{66}$$

Kde $\vec{a}$ je premenná, ktorá lineárne klesá z hodnoty 2 na hodnotu 0 počas iterácií a $\vec{r}$ je náhodný vektor v rozmedzí [0,1]. Vďaka úprave hodnôt $\vec{A}$ a $\vec{C}$ dokážeme dosiahnuť rozdielne umiestnenie v okolí pozície najlepšieho agenta. Vďaka definícií $\vec{r}$ je možné dosiahnuť akúkoľvek pozíciu v priestore prehľadávania. Každý agent teda aktualizuje svoju pozíciu na základe tej najlepšej.

2. Metóda útoku bublinovou sieťou: V zmysle matematického modelovania tejto metódy boli navrhnuté dva prístupy:

- Mechanizmus zmršťujúceho sa obkľúčenia (Shrinking encircling mechanism): Toto chovanie je dosiahnuté zmenšovaním hodnoty $\vec{a}$, čím tiež znižujeme kolísavý rozsah náhodnej hodnoty $\vec{A}$ z intervalu [-*a, a*], kde *a* klesá y hodnoty 2 na hodnotu 0 počas plynutia iterácií. Nastavením náhodných hodnôt pre $\vec{A}$ v intervale [-1,1] môže byť nová pozícia agenta definovaná kdekoľvek medzi pôvodnou pozíciou agenta a pozíciou najlepšieho súčasného agenta.
- Špirálová aktualizácia pozície (Spiral updating position): Tento prístup najprv spočíta vzdialenosť medzi veľrybou a korisťou. Následne je medzi týmito pozíciami vytvorená špirálová rovnica, ktorá napodobňuje špirálovo tvarovaný pohyb keporkakov podľa:

$$\vec{X}(t+1) = \overrightarrow{D'} * e^{bl} * \cos(2\pi l) + \overrightarrow{X^*}(t) \tag{67}$$

Kde $\overrightarrow{D'} = \left|\overrightarrow{X^*}(t) - \vec{X}(t)\right|$ indikuje vzdialenosť *i*-tej veľryby od koristi (doposiaľ najlepšieho nájdeného riešenia), *b* je konštanta pre definíciu tvaru logaritmickej špirály a *l* je náhodné číslo z intervalu [-1,1].

Model pracuje s predpokladom 50% pravdepodobnosti výberu jedného z týchto dvoch prístupov. Matematicky je tento predpoklad modelovaný nasledovne:

$$\vec{X}(t+1) = \begin{cases} \overrightarrow{X^*}(t) - \vec{A} * \vec{D} & if\ p < 0.5 \\ \overrightarrow{D'} * e^{bl} * \cos(2\pi l) + \overrightarrow{X^*}(t) & if\ p \geq 0.5 \end{cases} \tag{68}$$

Pričom *p* je náhodné číslo v intervale [0,1].

3. Hľadanie potravy: Keporkaky hľadajú potravu náhodne podľa vzájomných pozícií. Aby sme agentovi vnútili pohyb smerom od referencovanej veľryby, je využité vektora $\vec{A}$, pričom pre jeho hodnoty platí $\left|\vec{A}\right| \leq 1$ alebo $\left|\vec{A}\right| > 1$. Výpočet vzdialenosti agentov a aktualizácia pozície prebiehajú pri tomto mechanizme podľa rovníc:

$$\vec{D} = \left|\vec{C} * \vec{X}_{rand} - \vec{X}\right| \tag{69}$$

$$\vec{X}(t+1) = \vec{X}_{rand} - \vec{A} * \vec{D} \tag{70}$$

Kde $\vec{X}_{rand}$ je vektor pozície náhodnej veľryby, zvolenej z aktuálnej populácie.

WOA algoritmus teda začne so skupinou náhodných riešení. V každej iterácií agenti obnovia svoje pozície v závislosti buď na náhodne zvolenom agentovi alebo na najlepšom agentovi (doposiaľ najlepšom nájdenom riešení). Pre aktualizáciu pozície agenta je náhodný agent zvolený za podmienky $\left|\vec{A}\right| > 1$, naopak najlepší agent je zvolený v prípade, že platí $\left|\vec{A}\right| \leq 1$. V závislosti na hodnote parametra *p* je WOA schopné prepínať medzi špirálovým alebo kruhovým pohybom (mechanizmom zmršťujúceho sa obkľúčenia).

```
WOA
Initialize the random population of whales Xi(i= 1,2,…..n)
Initialize parameters a, A, C, l and p
Evaluate the fitness value of all search agents
X*= the best search agents
while(k<max value of iteration)
    for each member in search space
        if (p<0.5)
            if (|A|<1)
                update the current Position with equation no.(1)
            else if (|A|≥1)
            Update the current position of search agents by equation no.(8)
            end if
        else if(p≥0.5)
             update the position of the search agents by equation no.(5)
        end if
     end for
    Calculate the fitness
    Update X*, if the new one is better
    k=k+1
end while
return X*
```

Obr.16 Pseudo kód WOA[75]

4.13 Firefly Algorithm

Tento algoritmus bol sformulovaný v roku 2008 pánom Xin-She Yangom. Inšpirovaný je chovaním svetlušiek, konkrétne ich formou komunikácie – bioluminiscenciou [55,76]. Model algoritmu je založený na nasledujúcich princípoch:

- Všetky svetlušky sa navzájom priťahujú nezávisle na ich pohlaví (sú jednotného pohlavia)
- Príťažlivosť je proporcionálna ich jasu. Svetluška s menším jasom bude priťahovaná svetluškou s jasom väčším. So zvyšujúcou sa vzdialenosťou svetlušiek príťažlivosť klesá. Z toho vyplýva, že príťažlivosť je inverzne proporčná vzdialenosti svetlušiek.
- Pokiaľ je jas oboch svetlušiek rovnaký, budú sa pohybovať náhodne
- Jas svetlušky je ovplyvnený tvarom účelovej funkcie. Pre minimalizačný problém môže byť jas prevrátenou hodnotou účelovej funkcie, To znamená, že svetluška s vyšším jasom má minimálnu hodnotu účelovej funkcie. To znamená, že svetluška s vyšším jasom má minimálnu hodnotu účelovej funkcie. Naopak pre maximalizačný problém má jas priamu spojitosť s účelovou funkciou.

Intenzita žiarenia vyžarovaného svetluškou alebo jas variuje v závislosti na vzdialenosti r_{ij}, pričom je vyjadrená rovnicou:

$$I = I_0 e^{-\gamma r_{ij}^2} \tag{71}$$

Kde I_0 je intenzita v zdroji žiarenia a absorpcia žiarenia je spočítaná prostredníctvom konštantnej hodnoty nazývanej koeficient absorpcie svetla, ktorý je značený γ. Intenzita žiarenia predstavuje fitness hodnotu získanú z účelovej funkcie prostredníctvom pozície svetlušky.

Keďže príťažlivosť β svetlušky je proporcionálna intenzite svetla ňou vyžarovaného, ktoré je videné pridruženými svetluškami, môžeme ju formulovať nasledovne:

$$\beta = \beta_0 e^{-\gamma r_{ij}^2} \tag{72}$$

Kde β_0 je príťažlivosť vo vzdialenosti $r=0$. To znamená, že svetlušky veľmi blízko seba majú medzi sebou maximálnu príťažlivosť. Intenzita žiarenia je využitá svetluškou pri výbere lepšieho agenta, kým príťažlivosť slúži na následný výpočet novej pozície priťahovanej svetlušky. Ďalšie porovnanie môže predstavovať fakt, že intenzita žiarenia definuje mieru svetla vyžarovaného svetluškou, zatiaľ čo príťažlivosť je evaluácia svetla pozorovaného alebo videného iným agentom. Euklidovská vzdialenosť r_{ij} medzi dvoma svetluškami je vyjadrená ako:

$$r_{ij} = Distance(x_i, x_j) = \sqrt{\sum_{k=1}^{d}(x_{i,k} - x_{j,k})^2} \tag{73}$$

V kartézskom súradnicovom systéme sa dvaja agenti i a j nachádzajú na pozíciách x_i a x_j, pričom ich vzdialenosť je spočítaná prostredníctvom rovnice (73). Dimenzia riešenia je označená d. Pokiaľ i-ta svetluška uvidí lepšiu (svetlejšiu) j-tu svetlušku, zmení svoju pozíciu smerom k nej. Tento pohyb je možné vyjadriť rovnicou:

$$x_i^{t+1} = x_i^t + \beta * (x_j^t - x_i^t) + \alpha * \varepsilon \tag{74}$$

Tento pohyb obsahuje tri zložky. Prvou je súčasné riešenie (jeho pozícia) x_i^t, druhou je príťažlivosť β ku jasnejšiemu agentovi a tretia je náhodný pohyb pozostávajúci z dvoch parametrov. Prvým je α, čo je parameter náhodnosti alebo veľkosť kroku, nadobúdajúci hodnoty z intervalu [0,1]. Do určitej miery ním dokážeme riadiť diverzitu riešení. Druhý parameter, značený ε, je vektor náhodných čísel získaných rovnomerným alebo Gaussovským rozdelením, pričom platí vzťah $\varepsilon = (\text{rand} - 0.5)$, kde *rand* vygeneruje náhodné číslo v intervale [0,1]. Nakoniec t označuje súčasnú iteráciu. Náhodnosť ε dokážeme jednoducho nahradiť inými rozdeleniami, napríklad Lévyho letmi (a Lévyho rozdelením).

Firefly Algorithm

Objective function $f(\mathbf{x})$, $\mathbf{x} = (x_1, ..., x_d)^T$
Generate initial population of fireflies $\mathbf{x}_i$ $(i = 1, 2, ..., n)$
Light intensity I_i *at* $\mathbf{x}_i$ *is determined by* $f(\mathbf{x}_i)$
Define light absorption coefficient γ
while *(t <MaxGeneration)*
for $i = 1 : n$ *all n fireflies*
  for $j = 1 : i$ *all n fireflies*
    if $(I_j > I_i)$, *Move firefly i towards j in d-dimension;* **end if**
    Attractiveness varies with distance r via $\exp[-\gamma r]$
    Evaluate new solutions and update light intensity
  end for j
end for i
Rank the fireflies and find the current best
end while
Postprocess results and visualization

Obr.17 Pseudo kód FA[77]

5. REALIZÁCIA A IMPLEMENTÁCIA

Táto kapitola bude pojednávať o štruktúre programu a jeho jednotlivých zložiek, popíše ich funkčnosť a implementáciu. Zároveň bude obsahovať informácie o programovacom jazyku a prostredí, v ktorom bol program vyvíjaný.

5.1 Programovací jazyk a vývojové prostredie

Ako programovací jazyk na implementáciu programového riešenia bol zvolený jazyk Python. Jedná sa o univerzálny programovací jazyk, ktorý bol vyvinutý v rokoch 1985-1990 pánom Guidom Van Rossumom [78]. Python je interpretovaný a interaktívny. Okrem objektovo orientovaného programovania podporuje aj funkcionálne a štruktúrované (predovšetkým procedurálne) programovacie metódy. Môže byť využitý ako skriptovací jazyk alebo byť kompilovaný do byte kódu pre budovanie rozsiahlych aplikácií. Poskytuje prístup k vysoko úrovňovým dynamickým dátovým typom a tiež dynamické písanie. Python obsahuje garbage collector (automaticky uvoľňuje pamäť sám). Je tiež možné ho integrovať s jazykmi C/C++ alebo Java. Jeho standart library je prenosná a multiplatformová, teda kompatabilná s operačnými systémami UNIX, Windows a Macintosh. Ako nevýhodu pri využití tohto jazyka je nutné spomenúť jeho výpočetnú rýchlosť. Oproti práve spomínaným jazykom Java a C/C++ je Python pomalý, nakoľko sa jedná o interpretovaný jazyk. Ďalšiu nevýhodu môže predstavovať spotreba pamäte, ktorá je pri Pythone vysoká, práve kvôli flexibilite dátových typov.

Python bol zvolený primárne z dôvodov jeho širokej ponuky knižníc a to predovšetkým knižnice *numpy*, ktorá poskytuje podporu pre prácu s numerickými dátami, vektormi, maticami a tiež rady užitočných matematických funkcií. Ďalším dôvodom bola knižnica *Matplotlib*, ktorá umožňuje vizualizáciu dát.

Ako vývojové prostredie bol zvolený Pycharm od spoločnosti JetBrains, ktorý je, rovnako ako Python, multiplatformový [79].

5.2 Štruktúra optimalizačných algoritmov

Na implementáciu metód rojovej inteligencie bolo zvolených 11 algoritmov, ktoré boli teoreticky popísané v kapitole 4. V Tabuľke 3. je možné vidieť jednotlivé rojové algoritmy a názvy pyskriptov, v ktorých boli implementované. Na implementáciu bolo využitých techník objektovo orientovaného programovania. Každý zo skriptov obsahuje vždy jednu triedu s názvom samotného algoritmu. Táto trieda slúži ako hlavná. Do jej konštruktoru sa predávajú vstupné parametre pre algoritmus, ako sú počet jedincov, dimenzia problému, počet iterácií, budget, účelová funkcia a jej limity, ktoré sú spoločné pre všetky algoritmy, ale aj špecifické parametre, jedinečné pre každú metaheuristiku. Ďalej skripty obsahujú triedy, ktoré predstavujú jednotlivých agentov. Ich počet závisí od modelu algoritmu a samotnej implementácie. V nasledujúcich podkapitolách budú veľmi

stručne popísané štruktúry a funkcionalita jednotlivých implementácií swarm metaheuristík. Tento popis nebude obsahovať všetky funkcie implementované v jednotlivých skriptoch, práve pre dodržanie spomínanej stručnosti. Je dôležité spomenúť, že všetky skripty využívajú niektoré podobné princípy, no každý v svojej upravenej podobe. Tými sú napríklad princíp prehľadávania agentov, ktorý je realizovaný dvoma vnorenými *for* cyklami, kde prvý predstavuje iteráciu po generáciách rojov, kým vnorený iteruje po jednotlivých agentoch v roji. Ďalší napríklad predstavuje počítanie evaluácií a ukončenie prehľadávania algoritmu v prípade prekročenia budgetu. Iné z týchto spoločných princípov môžu byť spomenuté a popísané v samotných podkapitolách.

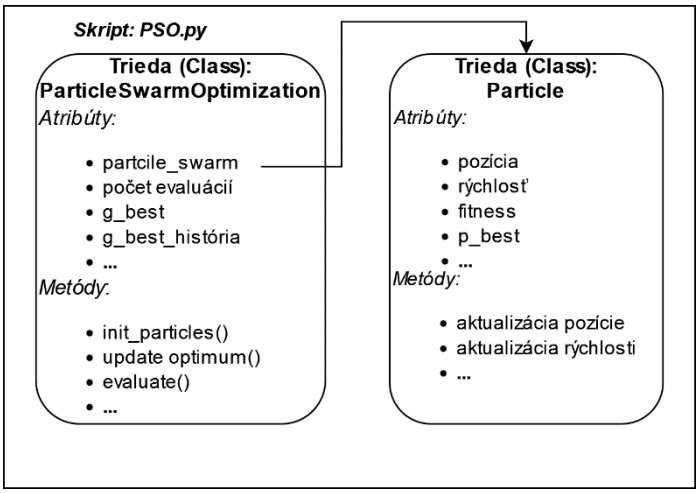
Tab.3 Swarm algoritmy a im prislúchajúce pyskripty

Particle Swarm Optimization	*PSO.py*
Artificial Bee Colony	*ABC.py*
Antlion Optimization	*ALO.py*
Cuckoo Search	*CS.py*
Dragonfly Algorithm	*DAO.py*
Bat Algorithm	*BA.py*
Elephant Herding Optimization	*EHO.py*
Grey Wolf Optimization	*GWO.py*
Moth Flame Optimization	*MFO.py*
Whale Optimization Algorithm	*WOA.py*
Firefly Algorithm	*FA.py*

5.2.1 PSO.py

PSO.py je pyskript obsahujúci implementáciu algoritmu *Particle Swarm Optimization*. Táto metaheuristika bola popísaná v kapitole 4.3. Pyskript obsahuje dve triedy, tými sú *ParticleSwarmOptimization* a *Particle*. Prvá spomínaná slúži ako hlavná a vstupom jej konštruktoru sú okrem všeobecných vstupných parametroch (popísaných v predchádzajúcej kapitole 5.2) aj špecifické parametre *c1*, *c2*, a *w*. Táto trieda ukladá informácie o dôležitých parametroch ako sú globálne najlepšie riešenie a jeho história, počet evaluácií, ktorý slúži ako zastavovacie kritérium v prípade prečerpania budgetu (fixed-budget) a predovšetkým populáciu agentov *particle_swarm*. Každý agent je reprezentovaný triedou *Particle*, ktorej atribútmi sú okrem iných poloha, rýchlosť a fitness častice. Jednotliví agenti sú vytváraní metódou *init_particles()*, v ktorej sú do konštruktoru každej častice predané parametre potrebné pre jej umiestnenie a výpočet hodnoty účelovej funkcie. Na vytvorenie pozície je využité knižnice *numpy* a ňou implementovaného náhodného rovnomerného rozdelenia (*uniform random distribution*).

Tento princíp umiestnenia jedincov je využitý pri každom algoritme, ktorý bol implementovaný. Optimalizácia je realizovaná prostredníctvom metódy *evaluate()*, v ktorej sa v jednotlivých iteráciach *for* cyklu snaží roj častíc iteratívne zmeniť svoju pozíciu podľa princípov definovaných v kapitole 4.3. Po každej generácií je aktualizované najlepšie riešenie skrz metódu *update_optimum()* a história jeho hodnôt.

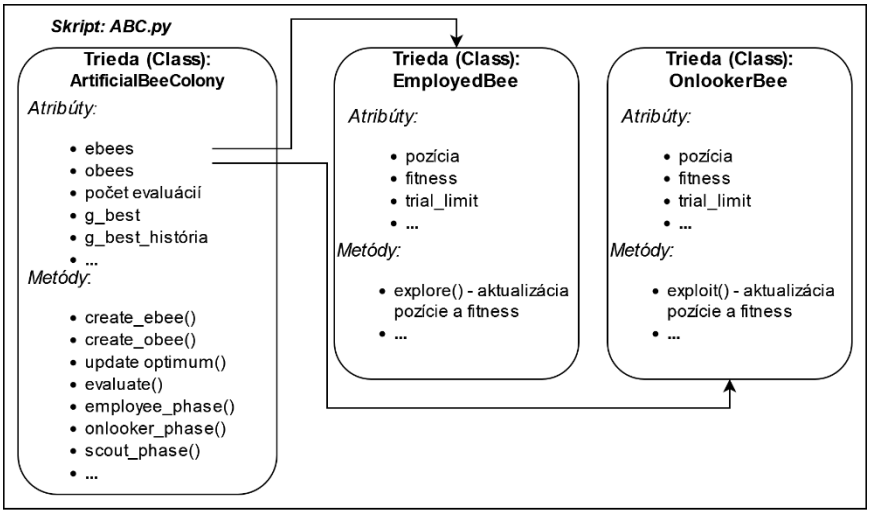


Obr.18 PSO.py štruktúra

5.2.2 ABC.py

ABC.py je pyskript obsahujúci implementáciu *Artificial Bee Colony*. Tento algoritmus bol popísaný v kapitole 4.4. Obsahuje tri triedy, tými sú *ArtificialBeeColony*, *EmployedBee*, *OnlookerBee*. Konštruktor triedy pomenovanej po algoritme preberá všeobecné parametre (popísané v kapitole 5.2) a špecifický parameter, ktorým je *trial_limit*. V prípade ABC.py je swarm agentov rozdelený na dve časti – dva roje. Prvú časť tvoria zamestnané včely. To sú agenti triedy *EmployedBee*, pričom druhá je tvorená agentami triedy *OnlookerBee*, teda pozorovateľmi. Oba typy agentov nesú informácie rovnakého typu, ako je pozícia včely či jej fitness, no aj sebe špecifické, ako je napríklad parameter *prob* v prípade *EmployedBee*. V prípade metód je tento princíp rovnaký. Skupiny agentov sú vytvorené prostredníctvom metód *create_ebee()* a *create_obee()*. V týchto metódach sú konštruktorom tried agentov predané parametre ako napríklad dimenzia, limity problému, či špecifický parameter *trial_limit*. Ich pozícia je generovaná rovnako, ako popisuje kapitola 5.2.1. V metóde *evaluate()* sa včely pokúšajú zlepšiť svoju pozíciu. Táto operácia je rozdelená do fáz. Prvá fáza *employee_phase()* je fáza agentov typu *EmployedBee*, ktorí preskúmavajú prehľadávaný priestor a tým potencionálne menia svoje pozície. Následne v druhej fáze *onlooker_phase()* včely typu *OnlookerBee* využijú informácie získané včelami v prvej fáze na zmenu svojej pozície. Posledná fáza, nazývaná *scout_phase()* simuluje chovanie včiel prieskumníkov podľa kapitoly 4.4. Tí ako samostatná trieda v tejto implementácií neexistujú. V tejto fáze sa včely, ktoré

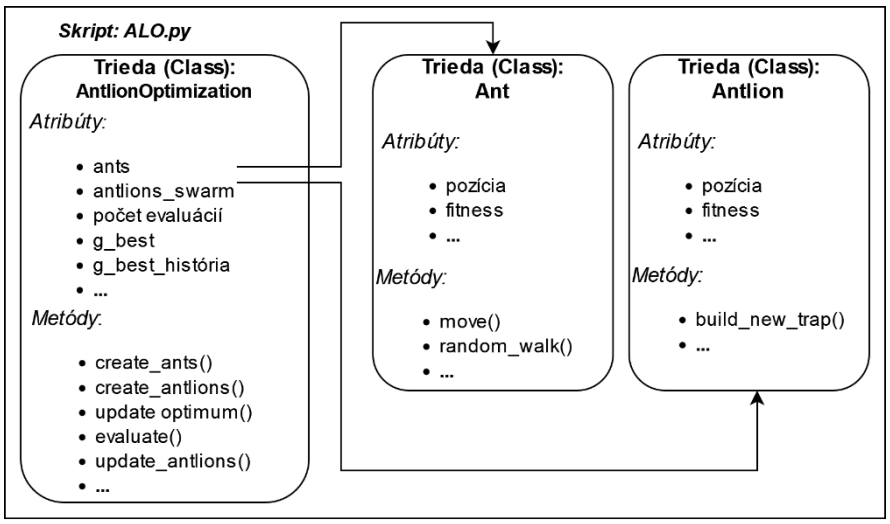
prekročili *trial_limit* nahradia novými riešeniami. Najlepšie riešenie je aktualizované po každej fáze, pričom jeho história hodnôt iba raz za iteráciu.



Obr.19 ABC.py štruktúra

5.2.3 ALO.py

ALO.py je pyskript implementujúci algoritmus *Antlion Optimization* popísaný v kapitole 4.5. Skript obsahuje tri triedy. Tými sú *AntlionOptimization*, *Antlion* a *Ant*. Konštruktor prvej spomínanej triedy pracuje na rovnakom princípe, ako bolo popísané v predchádzajúcich podkapitolách 5.2.1 a 5.2.2.



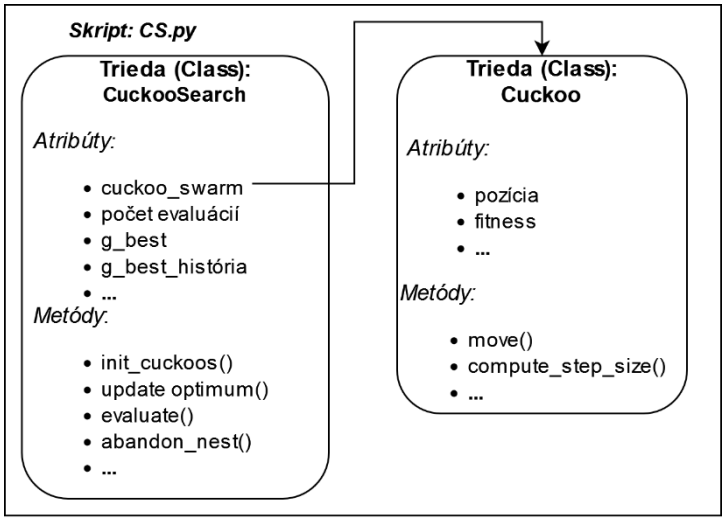
Obr.20 ALO.py štruktúra

Táto metaheuristika využíva dva roje. Jeden je tvorený mrakolevmi triedy *Antlion*, ktoré sú vytvorené metódou *create_antlions()* a druhý mravcami triedy *Ant*, vytvorenými s využitím metódy *create_ants()*. Pozície agentov sú vygenerované podľa princípu popísaného v podkapitole 5.2.1. Prehľadávanie priestoru sa realizuje prostredníctvom

mravcov, ktoré sa náhodne pohybujú okolo mrakolevov. Tento princíp popisuje kapitola 4.5. Na zmenu pozície, ktorú prevádza s využitím metódy *move()*, aplikuje mravec radu pomocných metód slúžiacich na výpočet parametrov potrebných pre realizáciu samotného pohybu. Príkladom takýchto pomocných metód môže byť *random_walk()*, ktorá spočíta krok mravca. O ukončení fáze pohybu mravcov sa ich roj spojí s rojom mrakolevov a na základe fitnessu navzájom prislúchajúcich agentov sa v metóde *update_antlions()* vyhodnotí, či b mravec chytený, alebo nie. To znamená, či mrakolev vybuduje novú pascu na novej pozícií. Túto operáciu realizuje metódou *build_new_trap()*. Optimalizácia prebieha v metóde *evaluate()*, kde sa popísaný postup iteratívne opakuje. Pre každú iteráciu sa aktualizujú optimum a jeho história hodnôt, pričom algoritmus neustále kontroluje, či nedošlo k prekročeniu prideleného rozpočtu na prevádzanie evaluácií.

5.2.4 CS.py

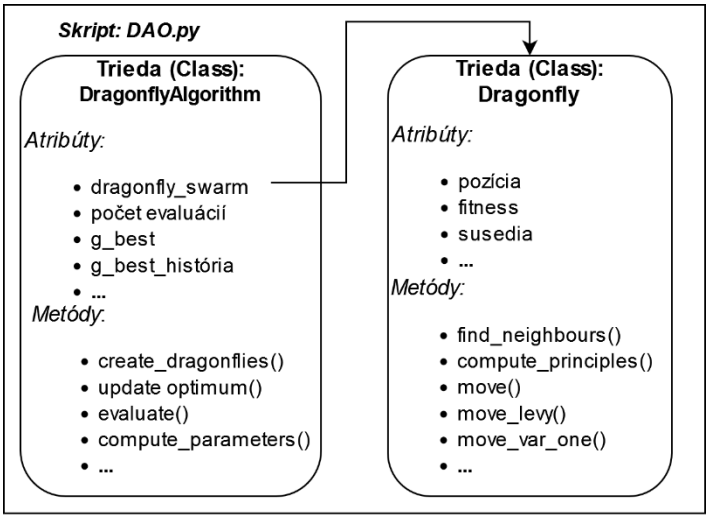
Tento skript implementuje metaheuristiku *Cuckoo Search*, ktorá bola popísaná v kapitole 4.6. Obsahuje dve triedy, *CuckooSearch* a *Cuckoo*. Konštruktor prvej spomínanej triedy pracuje na rovnakom princípe, ako v prípade predchádzajúcich podkapitol (5.2.3, 5.2.2...). Okrem všeobecných preberá aj špecifické parametre algoritmu, ktoré neskôr predáva ďalej jednotlivým agentom pri ich vytváraní prostredníctvom metódy *init_cuckoos()*. Agenti, ktorý tvoria roj *cuckoo_swarm*, sú triedy *Cuckoo*. Algoritmus v metóde *evaluate()* implementuje optimalizáciu účelovej funkcie. V každej iterácií sa kukučka v roji pokúša zlepšiť svoju pozíciu prostredníctvom metódy *move()*. Tá využíva pomocné metódy, ako napríklad *compute_step_size()*, na výpočet kroku Lévyho letu. Po tom, ako sa všetky kukučky pokúsia zmeniť svoju pozíciu, je v iterácií zavolaná metóda *abandon_nest()*, pri ktorej je náhodne zvolené riešenie nahradené novým. Najlepšie doposiaľ nájdené riešenie je samozrejme uložené, aby nedošlo k jeho strate. V každej iterácií (generácií) sa aktualizuje optimum a história jeho hodnôt.



Obr.21 CS.py štruktúra

5.2.5 DAO.py

Tento skript obsahuje implementáciu *Dragonfly Algorithm* metaheuristiky, popísanej v kapitole 4.7. Je tvorený dvoma triedami. Prvou je hlavná trieda *DragonflyAlgorithm*, ktorá preberá do konštruktoru, rovnako ako pri ostatných algoritmoch, všeobecné parametre. Roj je tvorený vážkami triedy *Dragonfly*, ktoré hlavná trieda vytvorí prostredníctvom metódy *create_dragonflies()*. Pozície agentov sú generované rovnako, ako popisuje podkapitola 5.2.1. Špecifické parametre pre tento algoritmus sú automaticky spočítané a aktualizované v každej iterácií (generácií) na základe súčasnej hodnoty iterácie prostredníctvom metódy *compute_parameters()*. Tá je volaná v metóde *evaluate()* triedy *DragonflyAlgorithm*. V každej iterácií sú spočítané parametre predané vážke, ktorá ich využije na zmenu pozície. Každá vážka pred samotným pohybom nájde svojich susedov vďaka metóde *find_neighbours()*. Agent na zmenu pozície využíva tri variácie, ktorými je možné ju realizovať. Prostredníctvom parametrov jej predaným si spočíta princípy potrebné pri výpočte jednotlivých pohybov. Optimum, spolu s históriou jeho hodnôt, je aktualizované pravidelne po generáciách.

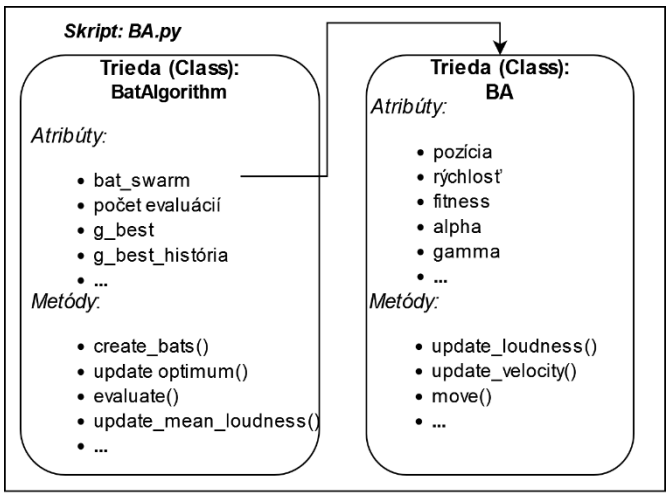


Obr.22 DAO.py štruktúra

5.2.6 BA.py

BA.py pyskript implementuje metaheuristiku *Bat Algorithm*, ktorá bola popísaná v kapitole 4.8. Podobne, ako v predchádzajúcich prípadoch, je skript tvorený dvoma triedami, *BatAlgorithm* a *Bat*. Prvá trieda pracuje rovnako, ako ostatné hlavné triedy popísané v podkapitolách 5.2.1-5.2.5. Vytvorí populáciu netopierov *bat_swarm* triedy *Bat* prostredníctvom metódy *create_bats()* a predá im parametre pôvodne predané jej konštruktoru. Následne v metóde *evaluate()* iteruje skrz všetky netopiere. V každej generácií sa aktualizuje hlasitosť netopierov s využitím metódy *update_loudness()*. Tento parameter bude predaný každému agentovi pri volaní metódy *move()*, slúžiacej na zmenu pozície. V tejto metóde agent zároveň aktualizuje špecifické parametre, ktoré nesie a to

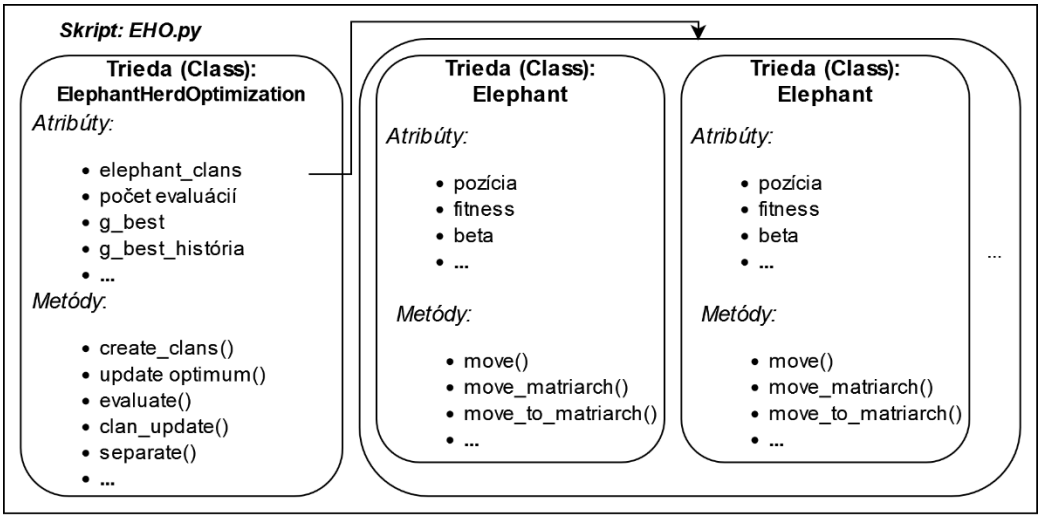
s využitím pomocných metód, ako napríklad *update_ferquency()*. V každej generácií sa samozrejme aktualizuje doposiaľ najlepšie nájdené riešenie a jeho hodnota sa tiež uloží do histórie optím počas iterácií.



Obr.23 BA.py štruktúra

5.2.7 EHO.py

Tento skript implementuje algoritmus *Elephant Herding Optimization*, popísaný v kapitole 4.9. Je tvorený dvoma triedami, *ElephantHerdingOptimization* a *Elephant*.



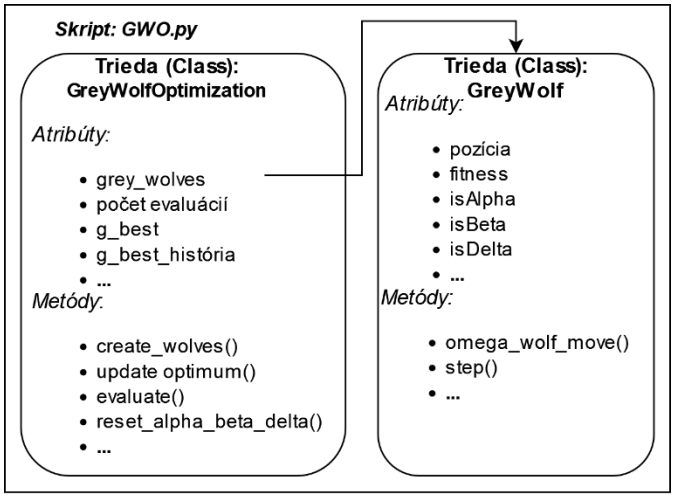
Obr. 24 EHO.py štruktúra

Prvá spomínaná trieda preberá, rovnako ako popisuje podkapitola 5.2.1, všeobecné a špecifické parametre algoritmu prostredníctvom svojho konštruktoru. S ich využitím inicializuje populáciu slonov *elephant_clans*, ktorým časť z nich aj predá. Slony, teda jednotliví agenti, sú triedy *Elephant*. Populácia slonov je však, oproti populáciám agentov pri ostatných algoritmoch, rozdelená na klany. To znamená, že trieda *ElephantHerdingOptimization* využije metódu *create_clans()* na vytvorenie klanov,

každý po rovnaký počet agentov. Optimalizácia je realizovaná metódou *evaluate()*, ktorá volá metódy implementujúce operátory aktualizácie a separácie. Prvý operátor je realizovaný metódou *clan_update()*, kde každý slon v klane zmení svoju pozíciu. Túto operáciu môže previesť prostredníctvom dvoch metód – *move_matriarch()* alebo *move_to_matriarch()*. Tú, ktorú zvolí, záleží na tom, či sa jedná o matriarchu (najlepšieho slona v klane) alebo nie. Zvolenú metódu agent volá v metóde *move()*. Druhý, separačný operátor, je realizovaný implementáciou metódy *separate()*. V každej generácií nahradí najhoršieho slona v klane novým agentom na novej pozícií.

5.2.8 GWO.py

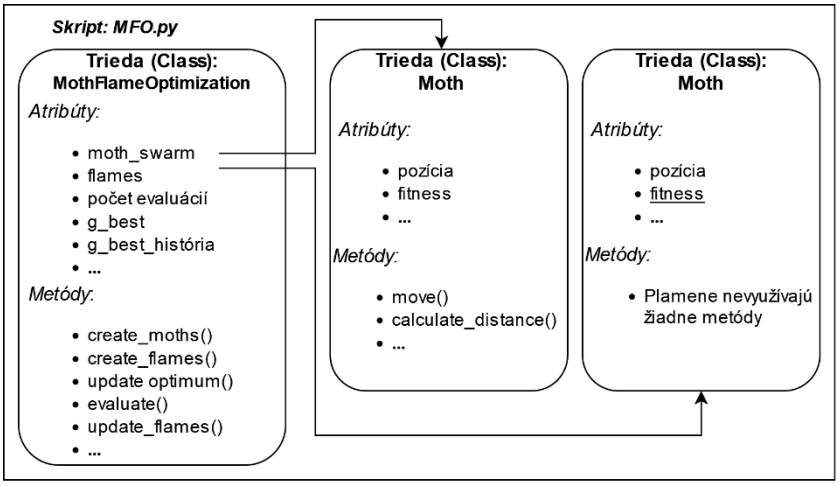
Tento skript obsahuje implementáciu metaheuristiky *Grey Wolf Optimization*, popísanú v kapitole 4.10. Jeho štruktúra je podobná predchádzajúcim algoritmom. Tvorený je dvoma triedami, *GreyWolfOptimization* a *GreyWolf*. Prvá spomínaná preberá do konštruktoru informácie, na základe ktorých vytvorí roj (svorku) vlkov *grey_wolves*, agentov druhej triedy *GreyWolf*. Túto operáciu zrealizuje s využitím metódy *create_wolves()*. Optimalizácia prebieha v metóde *evaluate()*. V nej sa iterujúc skrz populáciu vlkov, menia pozície jednotlivých agentov. Pred samotnou iteráciou sa určí, či je vlk alpha, beta, delta alebo omega. Tento status je realizovaný prostredníctvom vlajočiek s hodnotami *True* a *False*. Vlky, ktoré nemajú nastavenú ani jednu vlajočku, sú omegy. Práve tento typ agenta ako jediný realizuje pohyb s využitím metódy *omega_wolf_move()*, ktorá implementuje rady pomocných metód na potrebné výpočty (napríklad metóda *step()*). Po prejdení všetkých jedincov v generácií sa resetujú vlajočky s využitím metódy *reset_alpha_beta_delta_wolves()* a aktualizuje sa najlepšie riešenie a história hodnôt, ktoré počas generácií nadobúda. Proces sa opakuje opäť novou generáciou.



Obr.25 GWO.py štruktúra

5.2.9 MFO.py

MFO.py je pyskript popísaný v kapitole 4.11. Implementuje algoritmus *Moth Flame Optimization*. Podobne ako ostatné algoritmy, je tvorený dvoma triedami. Trieda *MothFlameOptimization* preberá vo svojom konštruktore špecifické a všeobecné parametre. Na ich základe prostredníctvom metódy *create_moth()* vytvorí roj molí *moth_swarm*. Následne s využitím metódy *create_flames()* vytvorí kópiu *moth_swarm*, čím vznikne „roj" plameňov *flames*. Tieto objekty sú samostatné entity, napriek tomu, že vznikli kopírovaním. Každá moľa tak má svoj prislúchajúci plameň, u ktorého využíva jeho fitness na aktualizáciu svojho riešenia. Prostredníctvom metódy *evaluate()* iteruje trieda *MothFlameOptimization* skrz všetky mole a im prislúchajúce plamene. Pred každým iterovaním (novou generáciou) sa spočítajú parametre pre realizáciu pohybu mole prostredníctvom pomocných metód, ako napríklad *compute_flames_no()*. Zároveň sa aktualizujú plamene s využitím metódy *update_flames()*, s výnimkou iterácie prvej. Agenti menia svoju pozíciu s využitím metódy *move()*, pričom na realizáciu samotného pohybu implementuje trieda *Moth* pomocné metódy, ktoré sú v tejto metóde volané (napríklad *calculate_distance()*). V každej iterácii (generácii) sa napokon, rovnako ako pri ostatných algoritmoch, aktualizuje optimum a história jeho hodnôt.

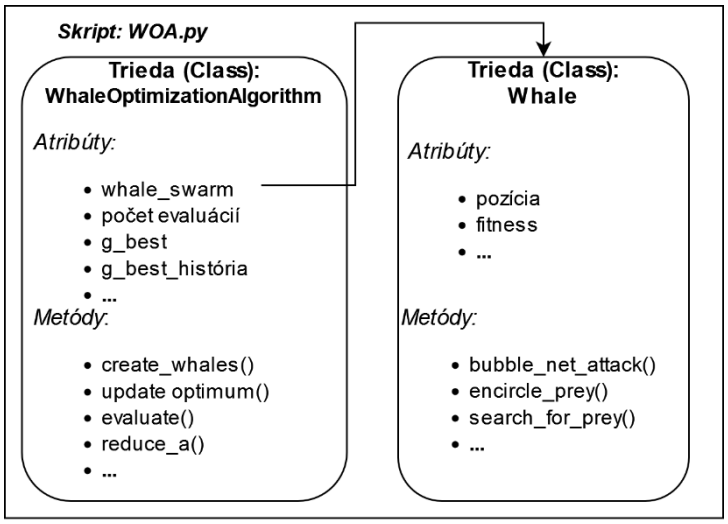


Obr.26 MFO.py štruktúra

5.2.10 WOA.py

Tento pyskript obsahuje implementáciu *Whale Optimization Algorithm* metaheuristiky, ktorá bola popísaná v kapitole 4.12. Algoritmov je realizovaný skrz dve triedy. Trieda *WhaleOptimization* preberá do konštruktoru parametre ako budget, dimenzia, či účelová funkcia (rovnako ako popisuje podkapitola 5.2.1). Prostredníctvom metódy *create_whales()* vytvorí swarm agentov triedy *Whale*. Následne s využitím metódy *evaluate()* algoritmus iteruje skrz všetky veľryby v roji. V každej generácii sa automaticky aktualizuje riadiaci parameter *a* prostredníctvom metódy *reduce_a()*. Ako ďalší krok sa pre každého agenta v swarme spočítajú ďalšie riadiace parametre, ktoré

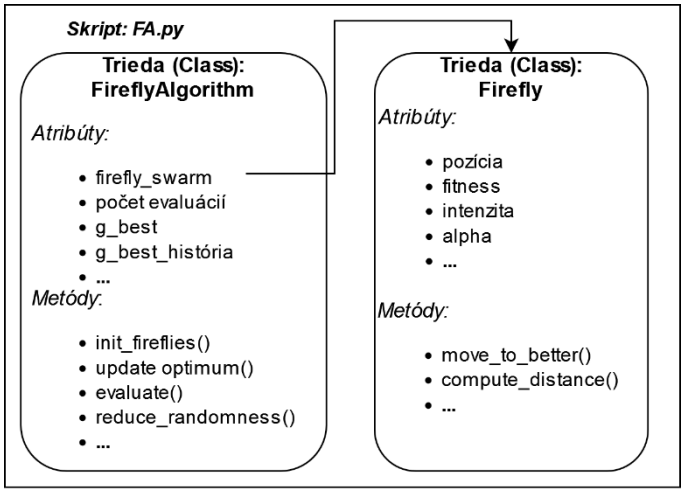
agent využije pri zmene svojej pozície. Tú aktualizuje s využitím jednej z troch implementovaných metód – *encircle_prey()*, *search_for_prey()* a *bubble_net_attack()*. Ktorú zvolí závisí od princípov definovaných modelom metaheuristiky. Každá z nich samozrejme využíva pomocné funkcie upravujúce parametre pri výpočtoch. Po ukončení iterovania generácie je aktualizované optimum spolu s históriou jeho hodnôt a proces sa opakuje.



Obr.27 WOA.py štruktúra

5.2.11 FA.py

FA.py implementuje *Firefly Algorithm* metaheuristiku, ktorá bola popísaná v kapitole 4.13. Tvorený je dvoma triedami. Trieda *FireflyAlgorithm* preberá do svojho konštruktoru parametre, pomocou ktorých zainicializuje roj svetlušiek *firefly_swarm* (rovnako ako popisujú podkapitoly 5.2.1-5.2.10) prostredníctvom metódy *init_firefly()*. Pri ich tvorbe predáva jednotlivým agentom parametre predané jej konštruktoru.



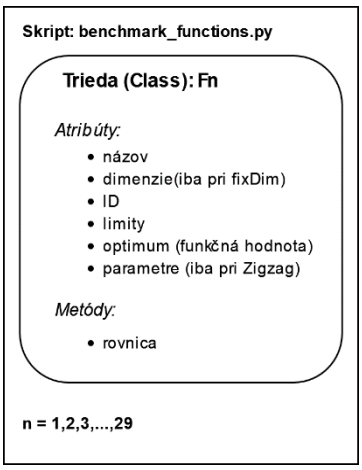
Obr.28 FA.py štruktúra

Agenti sú triedy *Firefly*. Každá svetluška je priťahovaná inou prostredníctvom jej intenzity. Algoritmus preto rozlišuje *fitness* a *intenzitu*, ktorá je jeho prevrátenou

hodnotou. S využitím metódy *evaluate()* sa algoritmus snaží iteratívne zlepšiť pozície svetlušiek. V každej iterácií sa prechádza celý swarm svetlušiek. To znamená, že každá svetluška prehľadá celý swarm, pričom mení svoju pozíciu iba k agentom s lepšou intenzitou (svetlejšiemu). Pohyb je prevedený metódou *move_to_better()*, pričom svetluška využíva pomocné metódy na výpočet parametrov dôležitých pre realizáciu pohybu ako napríklad *compute_distance()*, ktorá spočíta vzdialenosť od lepšieho agenta. Po ukončení iterovania generácie swarmu sa zredukuje parameter náhodnosti *alpha* u každej svetlušky a aktualizuje sa optimum spolu s históriou jeho hodnoty sledovanej počas generácií.

5.3 Zvolené testovacie úlohy

Pre zadanú problematiku bolo na vytvorenie benchmarku vybraných 29 testovacích funkcií. Tieto funkcie, získané z viacerých zdrojov, sme rozdelili na tri skupiny. Prvá skupina bola tvorená z unimodálnych a multimodálnych problémov s nastaviteľnými dimenziami [7,69], pričom boli použité dimenzie *D=2,5,10,15,20*. Druhá skupina sa skladala z multimodálnych problémov, ktorých dimenzie boli pevne definované.[69] Posledná skupina bola tvorená 4 špeciálnymi funkciami, nazývanými *zigzag*, ktoré sú parametrizovateľné (parametre boli nastavené podľa autora) a ich dimenzie boli definované rovnako, ako pri prvej skupine [80]. Všetky funkcie boli, rovnako ako aj algoritmy, napísané v jazyku Python. Všetky boli implementované v pyskripte pod názvom *benchmark_functions.py*. Každý problém bol reprezentovaný objektom, ktorý niesol atribúty ako meno a ID funkcie, dimenziu (iba pokiaľ bola fixovaná), obmezdenia problému a jeho hodnotu globálneho optima. Ďalej každý problém obsahoval metódu implementujúcu matematický prepis funkcie. Do tejto metódy sa ako vstupný parameter predávala pozícia agenta (včely, slona...) a dimenzia, pričom jej výstupom bola funkčná hodnota (fitness) na danej pozícií. Túto štruktúru znázorňuje obrázok 29.



Obr.29 benchmark.py štruktúra

Informácie o funkciách, ako sú ich hodnota optima, obmedzenia, či vykreslenia je možné nájsť v Prílohe 2. Na implementáciu matematických operácií bola využitá knižnica

numpy, na vykreslenie povrchov zase knižnica matplotlib. Jednotlivé skupiny problémov s im prislúchajúcimi názvami a ID je možné vidieť v tabuľkách 4.-6.

Tab.4 Unimodálne (F1-F7) a Multimodálne (F8-F15) testovacie funkcie

F1	*SphereModel*
F2	*SchwefelProblem222*
F3	*SchewelProblem12*
F4	*SchwefelProblem221*
F5	*Rosenbrock Function*
F6	*Step Function*
F7	*Quadratic Function (Noise)*
F8	*Schwefel Function*
F9	*Rastrigin Function*
F10	*Ackley Function*
F11	*Griewank Function*
F12	*Styblinski Tank*
F13	*Salomon Function*
F14	*Levy Function*
F15	*Zakharov Function*

Tab.5 Multimodálne testovacie funkcie s pevne definovanými dimenziami

F16	*M. Shekel Foxholes*
F17	*N. Kowalik Function*
F18	*Six-Hump Camel Function*
F19	*Bohachevsky Function*
F20	*Goldstein-Price Function*
F21	*R. Hartman Family (1)*
F22	*R. Hartman Family (2)*
F23	*Shekel Function (1)*
F24	*Shekel Function (2)*
F25	*Shekel Function (3)*

Tab.6 Zigzag testovacie funkcie

F26	*Zigzag F1*
F27	*Zigzag F2*
F28	*Zigzag F3*
F29	*Zigzag F4*

5.4 Hodnotiace kritériá a sledované parametre

Každý algoritmus popísaný v kapitole 5.2 zbiera a ukladá informácie pri riešení optimalizačných problémov. Tieto informácie sú následne využité pri porovnávaní výkonnosti a analýze riešenia benchmarku jednotlivými algoritmami. Ako kritérium výkonnosti bol zvolený Fixed-budget (kapitola 3.4.1). To znamená, že každý algoritmus dostal pridelený rovnaký rozpočet (budget) na optimalizáciu. Ten predstavoval maximálny počet evaluácií účelovej funkcie, ktoré môže vykonať. Algoritmy počítali jednotlivé evaluácie a pokiaľ došlo k prekročeniu budgetu, nastala terminácia otpimalizačného procesu.

V prípade riešenia problému bola tiež sledovaná hodnota optima (najlepšieho doposiaľ nájdeného riešenia) v každej generácií, ktorá sa ukladala do parametru histórie najlepších riešení. Pri terminácií optimalizácie, či už z hľadiska vyčerpania rozpočtu alebo vykonania definovaného počtu iterácií, sa posledné najlepšie nájdené riešenie uložilo ako globálne optimum, teda finálne riešenie algoritmu. To znamená, že v prípade implementovaného programu algoritmus ukladal históriu hodnoty optima v každej generácií počas riešenia úlohy a po ukončení optimalizácie uložil najlepšiu nájdenú hodnotu ako globálne optimum.

Keďže všetky implementované metaheuristiky sú zo skupiny rojových optimalizačných algoritmov, bolo prevedených niekoľko nezávislých behov na potlačenie ich stochastického chovania. Vďaka tomu bolo zaistené kritérium merania robustnosti. Z každého behu algoritmov boli sledované hodnoty globálnych optím, ktoré dosiahli pre dané spustenie. Napokon sa zo všetkých týchto hodnôt určili štatistické veličiny – aritmetický priemer, medián a smerodajná odchýlka. Vďaka tomuto procesu bolo zaistené sledovanie kvality riešenia. Z priemerov histórie hodnôt, ktoré optimum nadobúdalo počas riešenia problému jednotlivými algoritmami, bola vykreslená ich konvergencia ku globálnemu minimu testovacej funkcie.

Každý beh algoritmu bol sledovaný podľa kritéria času. K tomu poslúžila knižnica *pytictoc,* predstavujúca implementáciu Matlab funkcie *tictoc* v programovacom jazyku Python. Pre jednotlivé behy algoritmov boli z ich časov riešenia problému a počtu evaluácií pri nich prevedených spočítané aritmetické priemery. Na všetky štatistické operácie bola využitá knižnica *numpy*. Ďalšie sledované veličiny boli všeobecné

parametre, ktoré algoritmy zdieľali pri optimalizácií. Jednalo sa o názov problému, jeho ID a dimenziu. Pre každú testovaciu funkciu sa samozrejme sledoval a zaznamenával aj spomínaný počet behov (opakovaných spustení) a budget, pridelený na jej riešenie.

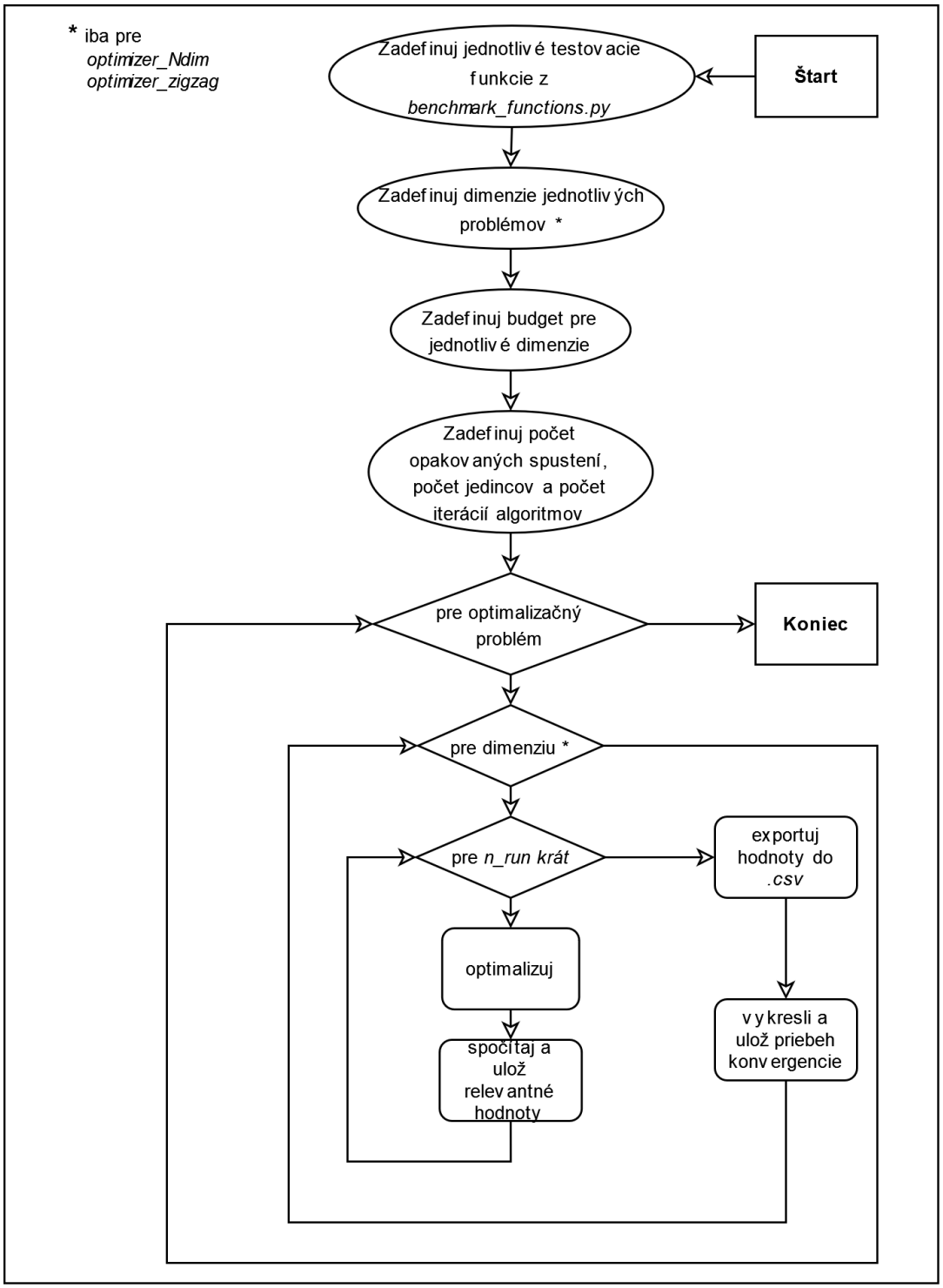
Tab.7 Kritériá a parametre

Kritérium výkonnosti	Fixed-budget, sledovanie budgetu
Kritérium merania času	Meranie CPU času pomocou pytictoc
Kritérium merania kvality riešenia	Mean, median, std najlepšieho nájdeného riešenia a rovnako histórie jeho hodnôt počas jednotlivých iterácii algoritmu, pre všetky opakované spustenia (jednotlivých behov) riešenia problému
Kritérium robustnosti	Opakované behy riešenia problému pri každom algoritme a sledovanie ich počtu
Sledované parametre	Optimalizačný problém, ID optimalizačného problému, dimenzia optimalizačného problému

5.5 Štruktúra hlavných skriptov

Jednotlivé kapitoly doposiaľ popisovali zložky, ktoré boli zhrnuté v hlavnom programovom riešení. Tento hlavný program bol rozdelený do troch častí, pričom každá obsahovala optimalizáciu a benchmarking jednej skupiny problémov popisovaných v kapitole 5.3. Podľa skupiny problému niesol prislúchajúci pyskript aj názov. To znamená, že boli vytvorené tri pyskripty, s názvami *optimizer_Ndim.py*, *optimizer_fixDim.py* a *optimizer_ZigZag.py*. V každom z nich sa nachádza funkcia *main()*, v ktorej sú vytvorené jednotlivé inštancie optimalizačných problémov, zadefinovaný rozpočet pre ich riešenie na základe ich dimenzie , počet opakovaných spustení algoritmov a počet jedincov a počet iterácií pre solvery v každom behu. Program teda riešenie každého problému na každej dimenzií opakuje *n_run* krát, s výnimkou *optimizer_fixDim.py*, ktorý tento postup aplikuje iba na jednu, pevne zadefinovanú dimenziu problému. V každom z týchto behov sú pre každý algoritmus zbierané informácie popisované v kapitole 5.4, ktoré sú následne upravené, spracované a uložené prostredníctvom pomocných funkcií. Ukladanie je realizované exportovaním výsledkov do *.csv* súborov, pričom sú zaznamenávané informácie tabuľky 7. To znamená, že pre každý problém na každej dimenzii sú exportované dva *.csv* súbory. Táto operácia je realizovaná dvoma funkciami - *save_statistics_F_D_R_B()* a *save_statistics_optimum()*. Zároveň je z dát, ako už bolo spomínané v predchádzajúcej kapitole, vykreslený konvergenčný graf pre každý problém a každú jeho dimenziu. Na túto operáciu slúži funkcia *plot_convergence()*. V pyskripte sú teda implementované tri vnorené (dva

v prípade *fixDim*) *for* cykly. Prvý iteruje skrz všetky optimalizačné problémy, druhý pre každú dimenziu problému a napokon tretí opakuje optimalizáciu *n_run* krát. Dáta sú ukladané po ukončení najvnorenejšieho *for* cyklu, teda pre každý problém na príslušnej dimenzií (v prípade *fixDim* sa cez dimenzie neiteruje, iba cez problémy).



Obr.30 Princíp funkcie *main()*

6. INTERPRETÁCIA A FORMA VÝSLEDKOV

Ako bolo spomínané v predchádzajúcej kapitole 5.5, program exportuje výsledky vo forme *.csv* a konvergenčného grafu (*.png* obrázok). Tieto dáta sú ukladané v systéme priečinkov, pričom výsledky sú rozdelené do troch takýchto štruktúr. Všetky tieto dáta je možné nájsť v elektronickej prílohe.

6.1 Adresárová štruktúra

Každá z troch spomínaných štruktúr prislúcha jednej skupine problémov a teda sú podľa nich pomenované – *zig_zag_statistics*, *n_DIM_statistics*, *fix_DIM_statistics*. Jednotlivé štruktúry sú tvorené súborom adresárov, každý obsahujúci informácie o riešení daného problému. Tieto adresáre sú značené vo forme *Fxx_benchmarking_statistics*, pričom *xx* predstavuje ID optimalizačného problému. V týchto priečinkoch sa nachádzajú exportované *.csv* súbory a adresár s konvergenčnými *.png* obrázkami.

6.2 Štruktúra exportovaných dát

Pre každý problém na danej dimenzii existujú dva súbory .csv, z ktorých prvý drží štatistické údaje (aritmetický priemer, smerodajnú odchýlku...) spočítané z parametrov (popísané v kapitole 5.4) všetkých behov algoritmov pre daný problém a príslušnú dimenziu. Pomenovaný je *Fxx_Dyy_statistics.csv*. Druhý typ súboru, pomenovaný *Fxx_Dyy_optimum_data.csv* obsahuje najlepšiu hodnotu optima (výsledok) nájdenú algoritmom v každom behu algoritmu, pre daný problém a dimenziu. V názve týchto súborov *xx* predstavuje ID optimalizačného problému a *yy* predstavuje dimenziu problému. Formu exportovaných dát je možné vidieť na obrázkoch č. 82 a č. 83.

Name	ID	Dim	No. Runs	Budget							
StyblinskiTank	F12	2	25	25000							
no.run	PSO	BA	ABC	ALO	CS	DAO	EHO	GWO	MFO	WOA	FA
1	-78.3323	-77.9598	-78.3323	-78.3323	-78.2542	-78.3166	-78.2613	-78.3288	-78.3323	-78.3322	-75.1729
2	-78.3323	-76.125	-78.3323	-78.3323	-78.2834	-78.2856	-77.9818	-78.3323	-78.3323	-78.3322	-75.7495
3	-78.3323	-77.9086	-78.3323	-78.3323	-78.1218	-78.2926	-77.9115	-78.3322	-78.3323	-78.3323	-78.3322
4	-78.3323	-78.0574	-78.3323	-78.3323	-78.2104	-78.2918	-78.2454	-78.3323	-78.3323	-78.3323	-62.9168
5	-78.3323	-78.2379	-78.3323	-78.3323	-78.1124	-78.3261	-77.7531	-78.332	-78.3323	-78.3322	-78.3319
6	-78.3323	-78.1081	-78.3323	-78.3323	-78.1977	-78.2581	-78.0694	-78.3323	-78.3323	-78.3323	-78.3193
7	-78.3323	-78.1858	-78.3323	-78.3323	-78.3119	-78.1662	-78.2742	-78.3315	-78.3323	-78.3323	-78.3164
8	-78.3323	-72.8785	-78.3323	-78.3323	-78.3054	-78.2911	-78.0564	-78.3244	-78.3323	-78.3323	-66.1972
9	-78.3323	-75.5561	-78.3323	-78.3323	-78.2357	-78.3003	-78.2855	-78.3323	-78.3323	-78.3323	-78.3302
10	-78.3323	-77.9774	-78.3323	-78.3323	-78.3262	-78.327	-77.9731	-78.3323	-78.3323	-78.3323	-77.5458
11	-78.3323	-78.0115	-78.3323	-78.3323	-78.2779	-78.273	-78.2173	-78.3322	-78.3323	-78.3323	-63.9301
12	-78.3323	-78.0015	-78.3323	-78.3323	-78.2705	-78.2898	-78.2889	-78.3271	-78.3323	-78.3323	-78.3323
13	-78.3323	-77.4242	-78.3323	-78.3323	-78.1429	-78.3119	-78.2853	-78.3307	-78.3323	-78.3321	-69.2152
14	-78.3323	-77.1164	-78.3323	-78.3323	-78.2863	-78.3112	-78.1854	-78.3314	-78.3323	-78.3323	-76.7353
15	-78.3323	-77.8379	-78.3323	-78.3323	-78.1973	-78.3265	-77.9766	-78.3321	-78.3323	-78.3323	-78.1963

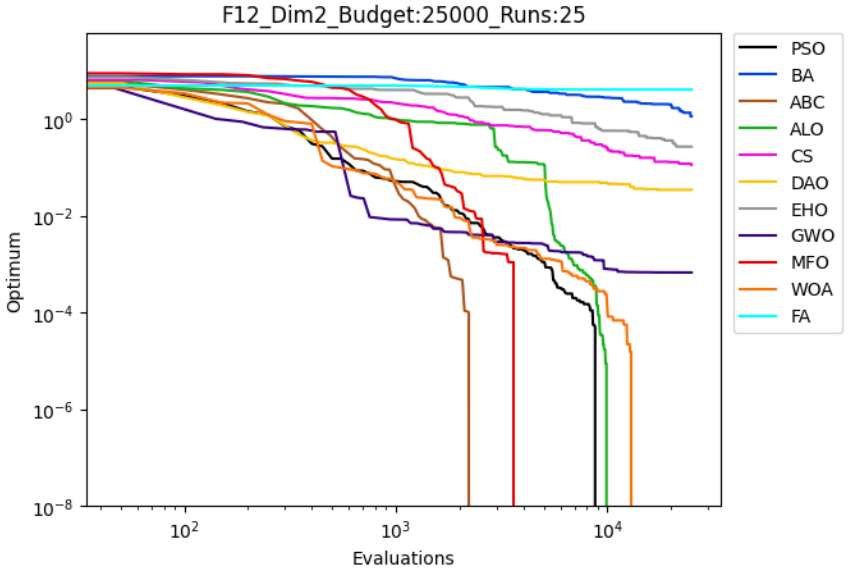
Obr. 31 F12_D2_optimum_data.csv

Name	ID	Dim	No. Runs	Budget							
StyblinskiTank	F12	2	25	25000							
Algorithm	PSO	BA	ABC	ALO	CS	DAO	EHO	GWO	MFO	WOA	FA
Mean	-78.3323	-77.1974	-78.3323	-78.3323	-78.219	-78.2977	-78.0642	-78.3313	-78.3323	-78.3323	-74.2728
Median	-78.3323	-77.9086	-78.3323	-78.3323	-78.2705	-78.3112	-78.1393	-78.3321	-78.3323	-78.3323	-76.7353
std	1.50E-05	1.603938	0	1.13E-05	0.131962	0.035118	0.31356	0.001854	1.58E-12	6.53E-05	5.130579
Min	-78.3323	-78.3241	-78.3323	-78.3323	-78.3307	-78.33	-78.3296	-78.3323	-78.3323	-78.3323	-78.3323
Max	-78.3323	-71.8504	-78.3323	-78.3323	-77.7839	-78.1662	-76.729	-78.3244	-78.3323	-78.3321	-62.9168
MeanTime	0.52527	0.55587	2.296136	6.689969	0.944471	13.08199	0.57247	1.081553	1.525783	1.022645	0.835508
MeanEval	25000	25000	25000	25000	25000	25000	25000	25000	25000	25000	25000

Obr. 32 F12_D2_statistics.csv

6.3 Štruktúra konvergenčných obrázkov

V adresárovej štruktúre sa okrem *.csv* exportovaných súborov nachádza aj priečinok s názvom *plots*. Ten obsahuje vykreslenia konvergencií všetkých algoritmov ku globálnemu optimu optimalizačného problému. Sú ukladané dve formy obrázkov, pričom v druhej sú x-ová a y-ová os prekonvertované do logaritmických súradníc, aby boli informácie lepšie interpretovateľné. Tieto vykreslenia nasledujú názvovú konvenciu exportovaných dát a teda *Fxx_Dyy_convergence.png* a *Fxx_Dyy_convergence_LOG.csv* pre graf v logaritmických súradniciach. Každý obsahuje legendu, v ktorej sú algoritmy označené svojimi skratkami. Horizontálna os znázorňuje počet evaluácií prevedených daným algoritmom a vertikálna nesie informácie o hodnota optima. Názov grafu nesie informácie o probléme, jeho dimenzií, budgete a počte behov (opakovaných spustení), pre ktoré boli dané výsledky získané. Jeho formát je teda v tvare *Fxx_Dimyy_Budget:ZZZZZ_Runs:QQ*, pričom *xx* a *yy* opäť značia *ID a dimenziu problému, ZZZZZ* značí množstvo rozpočtu (maximálneho počtu evaluácií) prideleného na riešenie a *QQ* značí počet opakovaných spustení algoritmov. Na obrázku č. 84 je možné vidieť príklad vykresleného konvergenčného grafu.



Obr. 33 Konvergenčný graf F12_D2_convergence_LOG.png

6.4 Aplikácia IOHanalyzer

IOHanalyzer je jedna z komponent benchmarkingovej platformy *IOHprofiler*. Tento software bol popísaný v podkapitole 3.5.3. Dáta, ktoré exportoval implementovaný program, boli upravené do podoby definovanej pre platformu, aby bolo možné ich nahrať a previesť ich analýzu [15]. Spracované a upravené boli dáta *Fxx_Dyy_optimum_data.csv* zo všetkých skupín, všetkých problémov na všetkých dimenziách. Následne boli skomprimované do *.zip* formátu a nahrané do internetovej verzie benchmarkingovej platformy [81]. Tieto dáta je možné nájsť v elektronickej prílohe, pričom jednotlivé formáty nesú názvy tvaru *IOH_data0_časť*. Časť v tejto konvencií značí buď jednotlivú skupinu (*n_DIM*, *fix_DIM*, *zig_zag*) alebo z nich zlúčený celok (*all*).

Po nahratí dát do *IOHanalyzer* bol na ohodnotenie výsledkov ich spracovania a analýzy využitý rebríček založený na *Glicko-2* klasifikácií. V každom kole pre každú funkciu a každú dimenziu nahratého dátasetu spolu súťažia páry algoritmov. Táto súťaž vzorkuje náhodnú funkčnú hodnotu pre poskytnutý počet behov. Ten algoritmus, ktorý má lepšiu funkčnú hodnotu, vyhrá hru. Z týchto hier je následne určené *Glicko-2* hodnotenie na vytvorenie rebríčku.

6.5 Glicko-2 klasifikácia

Na porovnanie a radenie algoritmov v zmysle, ktorý dosahuje lepšie výsledky existuje rada metód. Niektoré prístupy navrhujú využitie hodnotiaceho systému pri hre šach. To znamená, že je možne dosiahnuť tri výsledky – prvý hráč vyhrá, druhý hráč vyhrá alebo hra skončí remízou [82,83,84]. Výkonnosť hráča je popísaná číslom nazývaným ohodnotenie (rating). Ten je aktualizovaný po každom turnaji, ktorého sa hráč účastní. Existuje viac hodnotiacich systémov, no všetky majú spoločné dve vlastnosti. Ohodnotenie hráča je vždy kladné, celé číslo a hráč s vyšším ohodnotením je vždy lepší.

Pravdepodobne jedným z najznámejších hodnotiacich systémov je *Elo* systém. Jeho nevýhodu však predstavuje pravdepodobnosť, že hráč, ktorý hru vyhrá, stratí ohodnotenie. Môže sa jednať napríklad o situáciu, kde súťažia dvaja hráči, avšak jeden z nich nehral po dlhší čas (napríklad rok), kým druhý hral neprestajne – obaja stratia a získajú rovnaký počet bodov (rovnaké ohodnotenie). Očakáva sa, že pre hráča, ktorý nehral dlhší čas, bude ohodnotenie menej spoľahlivé, ako pri tom, ktorý hral neprestajne. Na popis spoľahlivosti zaviedol pán Glickman nový šachový hodnotiaci systém – *Glicko* systém. Ten pridáva novú hodnotu, ktorá reprezentuje spoľahlivosť hráča – hodnotiacu odchýlku *RD*, ktorá je podobná smerodajnej odchýlke pri štatistike. Tá je nastavená na 360 pri prvom turnaji (rovnako ako hodnotenie) a je aktualizovaná pri konci každého turnaja. Platí, že sa znižuje s každým turnajom, ktorého sa hráč zúčastnil a zvyšuje s každým turnajom, ktorý hráč vynechal. Maximálna hodnota je 350, pričom minimálna je podľa Glickmana navrhovaná 30. Čím je nižšia, tým spoľahlivejšie je ohodnotenie. V roku 2012 Glickman zaviedol vylepšenú formu *Glicko-2* hodnotiaci systém, ktorý

zavádza navyše ďalšiu premennú predstavujúcu spoľahlivosť sily hráča – nestálosť hodnotenia σ_i. Tá predstavuje stupeň očakávanej fluktuácie v hráčovom ohodnotení. Pokiaľ je nízka, hráč si počína konzistentne, pokiaľ je vysoká, je výkonnosť je nestála.

Pri prechode z *Glicko* na *Glicko-2* sa najskôr spočítajú ohodnotenie *R* a hodnotiaca odchýlka *RD*.

$$\mu = \frac{R-1500}{173.7178} \quad (75), \qquad \phi = \frac{RD}{173.7178} \quad (76)$$

Predpokladaná variancia *v* hráčovho hodnotenia založená iba na výsledku hry je získaná prostredníctvom rovnice:

$$v = \left(\sum_{j=1}^{m} g(\phi_j)^2 E(\mu_i, \mu_j, \phi_i)(1 - E(\mu_i, \mu_j, \phi_i))\right)^{-1} \quad (77)$$

Gravitačný faktor g a očakávané skóre E sú spočítané prostredníctvom nasledujúcich formulácií:

$$g(\phi) = \frac{1}{\sqrt{1+3\phi^2/\Pi^2}} \quad (78), \quad E(\mu_i, \mu_j, \phi_i) = \frac{1}{1+10^{-g(\phi_i)(\mu-\mu_i)}} \quad (79)$$

Ako ďalšie je určené očakávané zlepšenie Δ, v ktorého formulácií je ohodnotenie μ_i porovnané s výkonnostným ohodnotením μ_j iba na základe výsledku hry $S_{i,j}$:

$$\Delta = v \sum_{j=1}^{m} g(\phi)(S_{i,j} - E(\mu_i, \mu_j, \phi_i)) \quad (80)$$

Nová nestálosť hodnotenia σ' je nájdená pri využití *Illinoiského algoritmu* pre funkciu tvaru $f(x) = \frac{e^x(\Delta^2-\phi^2-v-e^x)}{2(\phi^2+v+e^x)^2} - \frac{x-\ln(\sigma^2)}{\tau^2}$ s presnosťou na 6 desatinných miest. Táto metóda je použitá na nájdenie núl x_0 tejto funkcie. Keď sú získané, σ'sa nastaví na hodnotu $e^{x_0/2}$ a je spočítaná ϕ^* vyjadrujúca hodnotu pred ohodnotením:

$$\phi^* = \sqrt{\phi^2 + \sigma'^2} \quad (81)$$

Nové hodnoty hodnotiacej odchýlky ϕ' a ohodnotenia μ' sa potom nastavia ako:

$$\phi' = \frac{1}{\sqrt{\frac{1}{(\phi^*)^2} + \frac{1}{v}}} \quad (82)$$

$$\mu' = \mu + \phi' \sum_{i=1}^{m} g(\phi_i)(S_{i,j} - E(\mu_i, \mu_j, \phi_i)) \quad (83)$$

Napokon je nové ohodnotenie R' a nová hodnotiaca odchýlka RD' skonvertované z *Glicko* do *Glicko-2* systému prostredníctvom formuly:

$$R' = 173.7178\mu' \quad (84), \qquad RD' = 173.7178\phi' \quad (85)$$

Miera zmeny *RD* je teda určená nestálosťou hodnotenia. Tá sa mení na základe náhlych zmien vo výkonnosti. Váha vplyvu hry na miesto v rebríčku hodnotenia je proporcionálna funkcii *RD*. Glicko-2

7. BENCHMARKING TESTOVACÍCH FUNKCIÍ

Táto kapitola bude pojednávať o výsledkoch testovania výkonnosti všetkých algoritmov na vybraných testovacích problémoch. Je nutné upozorniť, že výsledky testov sú relevantné pre jednotlivé implementácie algoritmov, vytvorené pre účely tejto diplomovej práce, s nastaveniami, pri ktorých boli púšťané a na hardware, na ktorom boli vykonané. Testy prebiehali na osobnom počítači s nasledujúcimi parametrami:

- Operačná pamäť RAM 24.0 GB
- Procesor Intel®Core™i7-8750H CPU @ 2.20 GHz 2.21 GHz
- Grafická karta NVIDIA GeForce GTX 1060 6GB

Výsledky budú rozdelené do niekoľkých skupín. Najprv budú odprezentované testy pre jednotlivé skupiny popísané v kapitole 5.3. Potom budú odprezentované výsledky pre všetky skupiny zlúčené spolu, rozdelené podľa dimenzií problémov a nakoniec pre všetky skupiny zlúčené spolu pre všetky dimenzie.

Všetky testy prebiehali s rovnakým počtom jedincov a iterácií pre algoritmy(50 a 1000). Pokiaľ obsahoval algoritmus špecifické parametre, boli nastavené podľa autorov [49,51,67,73,76,85,86,87]. Každá optimalizácia problému algoritmom na danej dimenzií bola opakovaná 25 krát (účel a princíp popisujú kapitola 5.4 a 5.5.). Každý algoritmus tiež dostal rovnaký budget (rozpočet) na riešenie problémov. Ten bol obmedzený pre jednotlivé dimenzie na hodnotu, ktorá zaisťovala rozumnú časovú náročnosť pri daných hardware-ových podmienkach.

Výsledky budú prezentované vo forme *Glicko-2* hodnotení získaných z upravených dát exportovaných programom (popísané v kapitole 6.4). Sú zobrazené na grafoch, pričom môžu byť doplnené konvergenčnými grafmi, ktoré generuje programová implementácia. Ostatné výsledky testovania (popísané v kapitole 6.), ktoré je možné nájsť v elektronickej prílohe, budú v zhodnotení testov zohľadnené tiež. Tabuľkové výsledky *Glicko-2* hodnotení je možné nájsť v prílohe 3.

7.1 Výsledky benchmarkingu skupiny nDIM

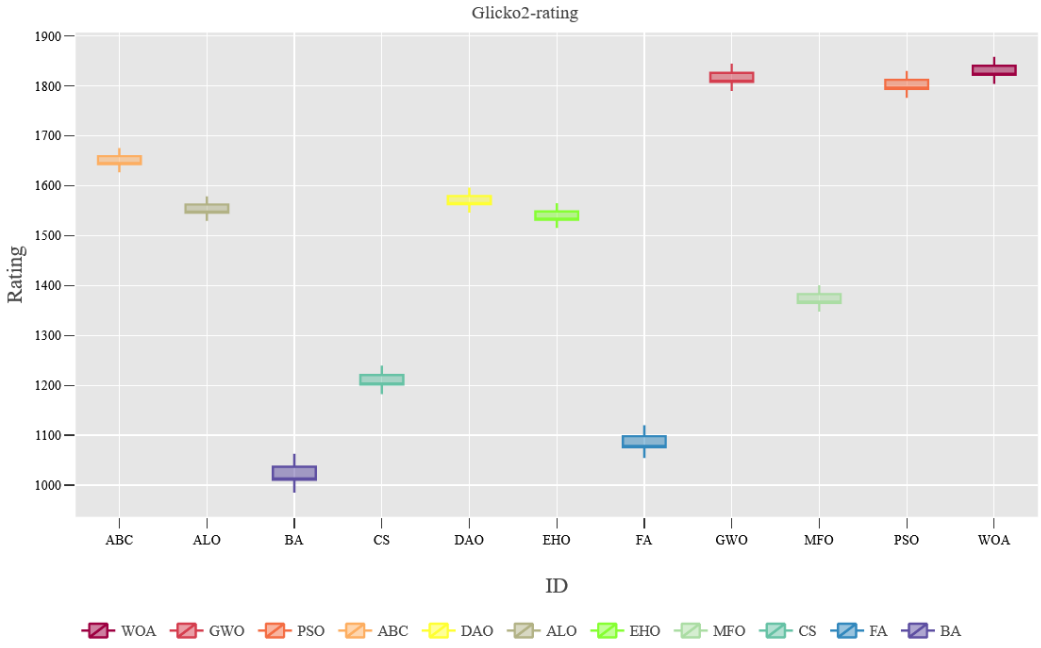
Táto skupina je tvorená unimodálnymi a multimodálnymi funkciami. Prvá spomínaná časť funkcií obsahuje iba jedno globálne optimum. Jedná sa o funkcie F1-F7. Druhá skupina je tvorená problémami s viacerými globálnymi optimami, pričom ich počet narastá s počtom dimenzií. Patria sem funkcie F8-F15.

Z výsledkov *Glicko-2* hodnotenia na obrázku 85. je možné konštatovať, že najväčšiu úspešnosť dosiahli algoritmy Whale Optimization Algorithm, Grey Wolf Optimization, Particle Swarm Optimization a Artificial Bee Colony. Pomerne dobre si viedli tiež algoritmy Antlion Optimization, Dragonfly Optimization a Elephant Herding Optimization. Horšie výsledky dosiahol algoritmus Moth Flame Optimization, s Cuckoo Search umiestneným pod ním v rebríčku. S najnižším ohodnotením skončili Firefly Algoritmy a Bat Algorithm.

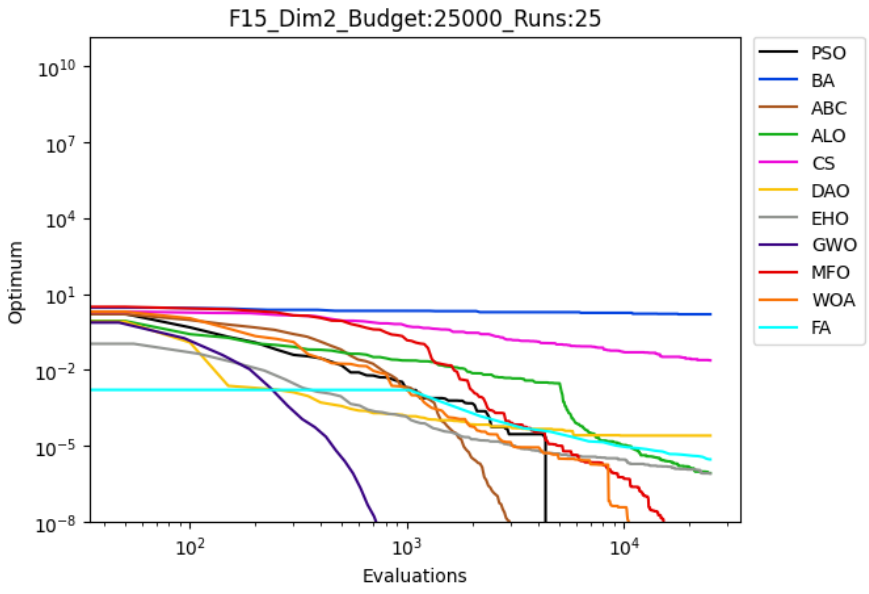
Pri sledovaní konvergencie na obrázku 86. je možné vidieť, že algoritmy s najlepším ratingom tiež najrýchlejšie konvergujú ku globálnemu optimu.

Táto skupina rovnako dosahuje najvyššiu presnosť riešenia. Pri algoritmoch umiestnených v prostriedku môžeme hovoriť o presnosti na jedno desatinné miesto, s výnimkou EHO, ktorý dokáže pri nižších dimenziách poskytnúť presnosť s dvomi desatinnými miestami.

Z hľadiska časovej náročnosti sú najpomalšími Antlion Optimization a Dragonfly Optimization, ktoré silne prevyšujú ostatné, pričom s narastajúcou dimenziou problému rastie aj čas ich riešenia. Najsilnejšie práve pri ALO (z priemerných 7 sekúnd za beh na 70 sekúnd pri 20D), napriek tomu, že problémy dokáže riešiť skutočne úspešne. Ako najrýchlejšie algoritmy je možné označiť EHO a BA.



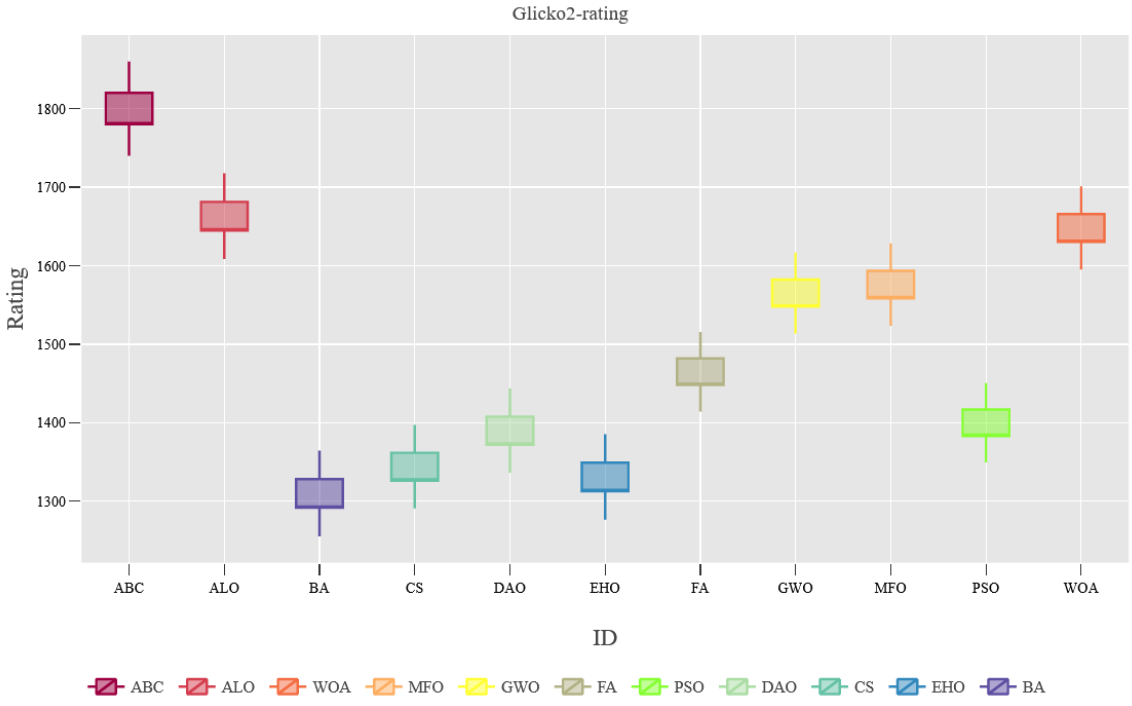
Obr.34 Glicko-2 hodnotenie skupiny nDIM



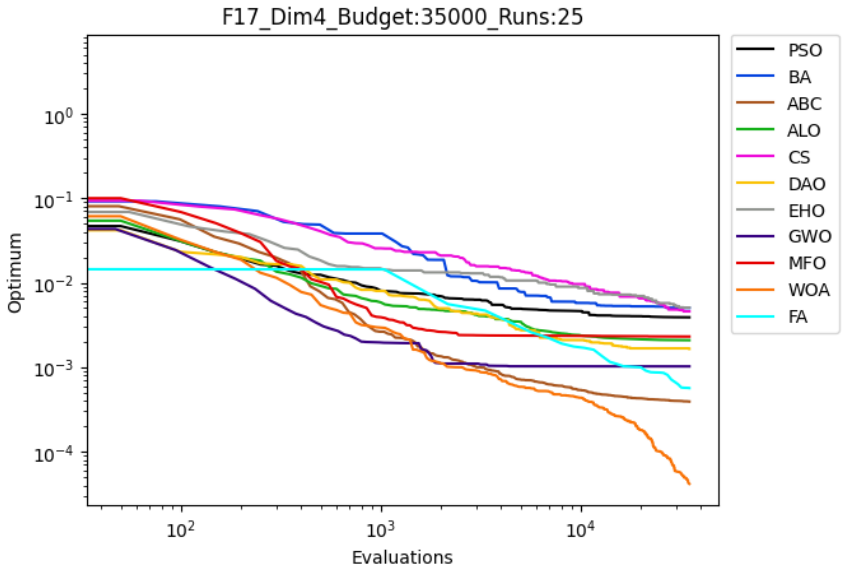
Obr.35 Konvergencia algoritmov pri probléme F15

7.2 Výsledky benchmarkingu skupiny fixDIM

Táto skupina je tvorená problémami s pevne nastaviteľnou dimenziou. Jedná sa o multimodálne funkcie v rozsahu F16-F25.



Obr. 36 Glicko-2 hodnotenie skupiny fixDIM



Obr.37 Konvergencia algoritmov pri probléme F24

Z výsledkov je možné vidieť podobnú úspešnosť s predchádzajúcou skupinou. Medzi najlepšími sa opakovane umiestnili ABC a WOA, pričom u ALO sa rating oproti predchádzajúcej skupiny vylepšil. MFO algoritmus si rovnako polepšil, zato u GWO došlo k miernemu zhoršeniu a umiestnil sa takmer na rovnakej priečke ako MFO. Silné

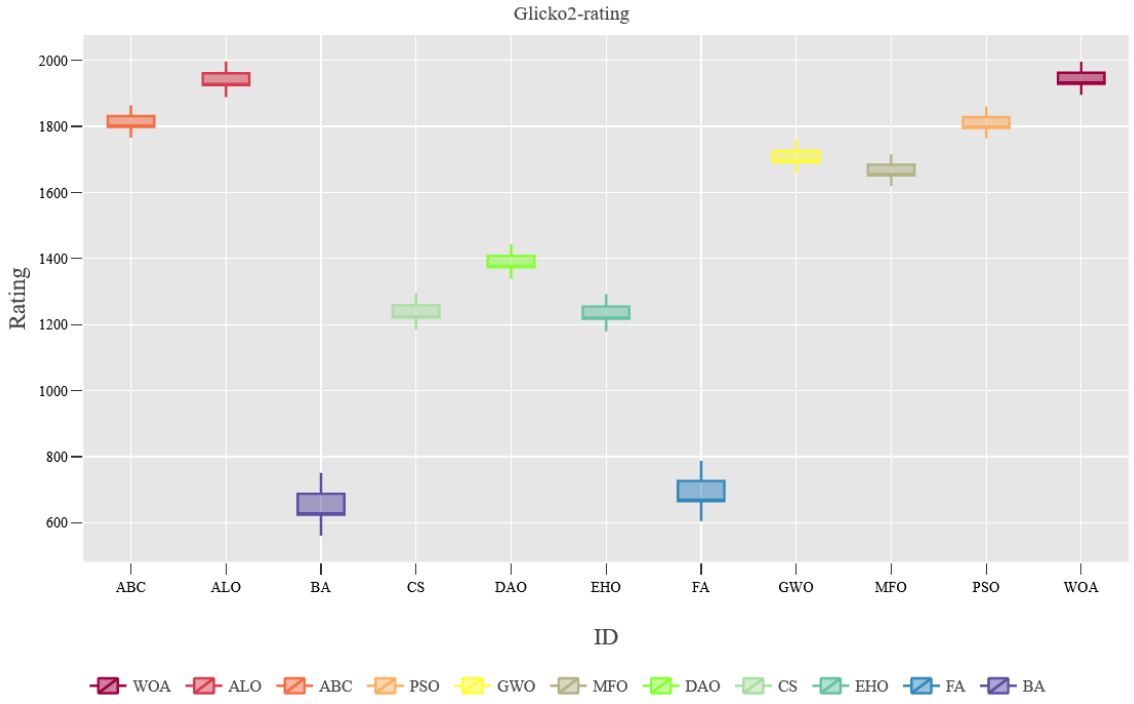
zhoršenie zaznamenal algoritmus PSO, ktorý klesol mierne nad algoritmy umiestnené na posledných miestach rebríčka. Pri bližšej analýze PSO je možné tvrdiť, že jeho zhoršenie je spôsobené vyčerpaním rozpočtu skôr ako u algoritmov umiestnených vo vyšších priečkach. Tam, kde budget vyčerpal neskôr, sa dokázal dostať pomerne blízko k hľadanému optimu, pri niektorých tak presne ako najlepšie algoritmy. Algoritmus DAO sa v rámci hodnotenia umiestnil na približne rovnakom mieste v rebríčku ako PSO. EHO sa umiestnil na pozícii blízko CS. Bližšia analýza však ukázala, že došlo nielen k zhoršeniu výkonnosti EHO ale aj k vylepšeniu CS a algoritmy zaostávali oproti lepším súťažiacim na rovnakých problémoch. BA sa opäť umiestnil na poslednej priečke. Výrazné zlepšenie je možné konštatovať u FA, ktorý ukázal, že pri nižších dimenziách dokáže podávať lepšie výsledky.

Časová náročnosť riešenia nasledovala vzor predchádzajúcej skupiny, teda ALO a DAO môžeme opäť označiť za najpomalšie.

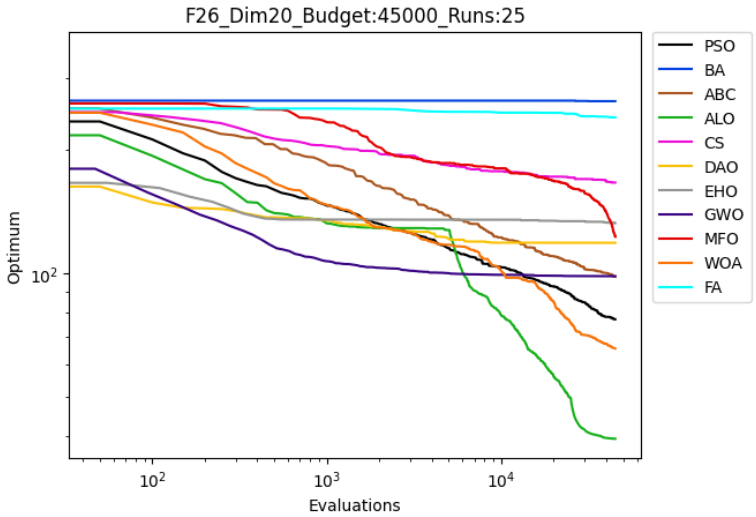
Príklad konvergencie algoritmov pri riešení problémov tejto skupiny je možné vidieť na obrázku 88. V prípade týchto testovacích funkcií riešili niektoré algoritmy jednotlivé problémy nerovnomerne úspešne (čo už bolo v analýze vyššie v texte spomenuté). Pri niektorých teda dosahovali omnoho lepšie výsledky. Práve tu *Glicko-2* hodnotenie ukázalo svoju presnosť a účinnosť v zostavovaní rebríčku.

7.3 Výsledky benchmarkingu skupiny zigzag

Táto skupina problémov je tvorená multimodálnymi parametrizovateľnými funkciami s veľmi vysokou členitosťou, čo robí ich riešenie skutočne obťažným. Náročnosť navyše výrazne narastá so zvyšujúcou sa dimenziou problému.



Obr.38 Glicko-2 hodnotenie skupiny zigzag



Obr.39 Konvergencia algoritmov pri probléme F26

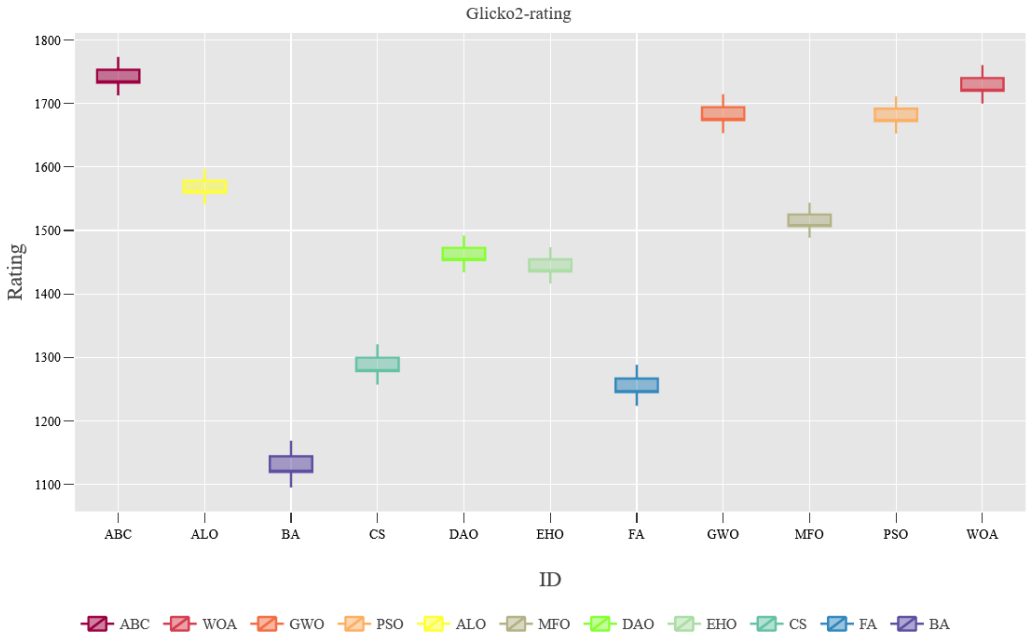
Z výsledkov je zrejmé, že najväčšiu úspešnosť opäť dosahujú favoriti predchádzajúcich skupín. WOA a ALO obsadili najvyššie umiestnenie v rebríčku, s ABC a PSO umiestnenými tesne za nimi. Úspešne si tiež počínali algoritmy GWO a MFO. Pri druhom spomínanom je možné vidieť zlepšenie oproti prvej skupine. Priemerne sa umiestnili DAO, CS a EHO, s prvým v poradí mierne lepším ako zvyšné dva. Na najnižších priečkach sa opäť umiestnili BA a tiež FA, ktorý zaznamenal zhoršenie oproti predchádzajúcej skupine. To znamená, že algoritmus má problémy riešiť problémy vyšších dimenzií.

Algoritmus ALO dosiahol najvyššiu časovú náročnosť, s priemerom 79 sekúnd pri 20D probléme, pričom DAO hneď za ním. Ostatné metaheuristiky však rovnako potrebovali viac času na riešenie problémov v porovnaní s ostatnými skupinami. Je nutné konštatovať, že pri poskytnutom budgete mali algoritmy často problém nájsť presné riešenie aj pri nižších dimenziách, čo dokazuje náročnosť tejto skupiny testovacích problémov. Z konvergencie na obrázku 90 je však zrejmé, že algoritmy konvergujú, avšak pri danom rozpočte a parametroch ich presnosť nie je veľmi vysoká.

7.4 Výsledky benchmarkingu zlúčených skupín pre DIM<6

Tento test bol prevedený na dátasete, ktorý vznikol zlúčením všetkých skupín, pričom sa hodnotili iba problémy s dimenziou menšou ako 6. Výsledky nasledujú koncepciu predchádzajúcich skupín. To znamená, že medzi najlepšími metaheuristikami sa opäť umiestnili ABC a WOA, pričom pri zhrnutí všetkých problémov týchto dimenzií sa GWO umiestnil ako tretí najlepší v rebríčku, čo predstavuje zlepšenie oproti predchádzajúcim prípadom, kedy najlepšie hodnotenia nedosahoval. Mierne za ním sa nachádzal PSO, nasledovaný ALO, ktorý pri tomto teste naopak klesol v hodnotení. Priemerne úspešné výsledky dosahujú MOF, DAO a EHO, čo približne zodpovedá ich umiestneniu v testoch prevedených na jednotlivých skupinách. CS a FA obsadili spodné priečky s nízkym hodnotením, pričom BA sa umiestnil na poslednom mieste.

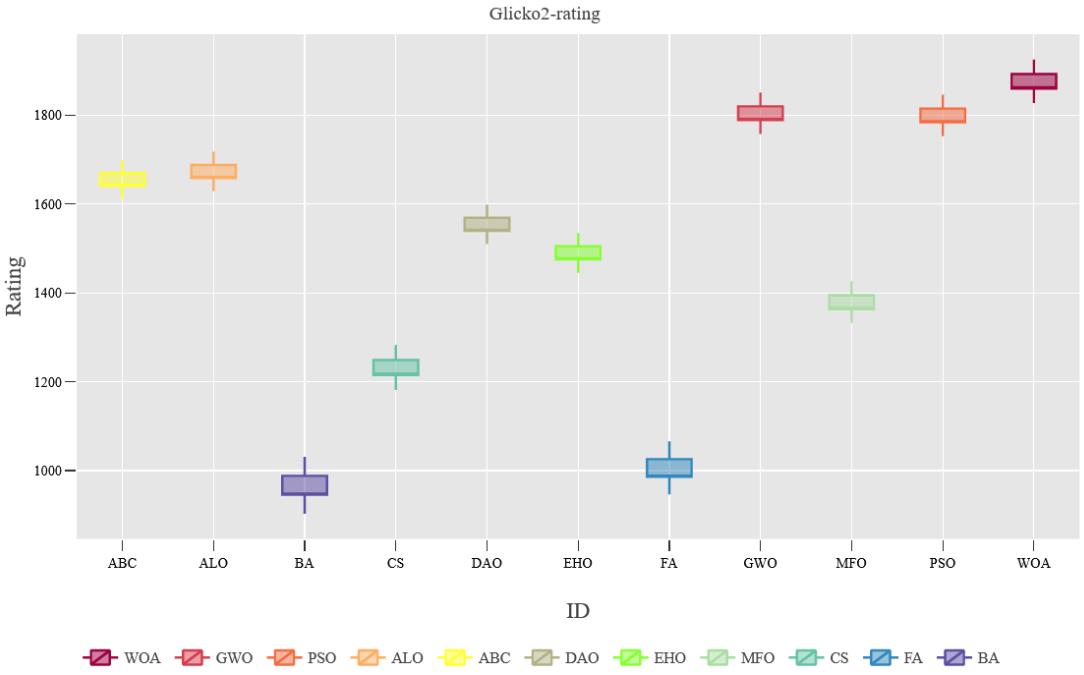
Pri nižších dimenziách dosahovali algoritmy pomerne presné výsledky. Výnimku môže tvoriť set problémov zigzag, ktorý predstavuje výzvu aj pre riešenie problémov nižších dimenzií a teda vyžaduje vyšší rozpočet.



Obr.40 Glicko-2 hodnotenie zlúčeného dátasetu pre DIM<6

7.5 Výsledky benchmarkingu zlúčených skupín pre DIM10

Využitý dátaset vznikol rovnako ako predchádzajúci a teda spojením všetkých skupín, pričom v tomto teste boli hodnotené iba problémy s dimenziou rovnou 10. Glicko-2 hodnotenie je možné vidieť na obrázku 92.



Obr.41 Glicko-2 hodnotenie zlúčeného dátasetu pre DIM10

V tomto prípade môžeme z výsledkov konštatovať, že algoritmy ABC a ALO si pohoršili v celkovom hodnotení, nielen oproti testom na jednotlivých skupinách, ale aj na zlúčenom dátasete s nižšími dimenziami. Prvú priečku opäť obsadil WOA, ktorý potvrdil svoju úspešnosť, nasledovaný algoritmom GWO, ktorý svoju pozíciu opäť vylepšil, v porovnaní s predchádzajúcim testom. To znamená, že GWO dosahuje vyššie priečky ako pri individuálnych testoch a jeho úspešnosť iba mierne zaostáva za WOA. Je teda zrejmé, že tento algoritmus si dobre vedie aj pri problémoch s viac premennými. Na rovnakom mieste v priečke sa umiestnil aj PSO, ktorý bol nasledovaný vyššie spomínanými ALO a ABC. Priemer rebríčka bol opäť obsadený metaheuristikami DAO, EHO a MFO. CS síce obsadzuje nižšie priečky, avšak v porovnaní s BA a FA je jeho úspešnosť výrazne vyššia. Tieto algoritmy majú problémy pri riešení zložitejších problémov a opäť skončili na poslednom mieste.

Pri zvyšujúcej sa dimenzií narastá aj časová náročnosť riešenia testovacích funkcií algoritmami. Predovšetkým ALO a DAO potrebujú v porovnaní s ostatnými súťažiacimi viac času na vyriešenie problému, čo predlžovalo aj priebehy jednotlivých testov. V prípade ALO je táto skutočnosť spôsobená veľkými maticami, ktoré modelujú náhodné pohyby mravcov. Ich zostavenie je pomerne zložité a využíva rady parametrov. Časová náročnosť u DAO je spôsobená tým, že každá vážka so sebou nesie informácie o susedoch, ku ktorým mení polohu. Pri konvergencií k riešeniu sa aj vážky približujú a teda počet susedov narastá. Každý agent potom so sebou nesie informácie o niekoľkých objektoch, čo spomaľuje riešenie.

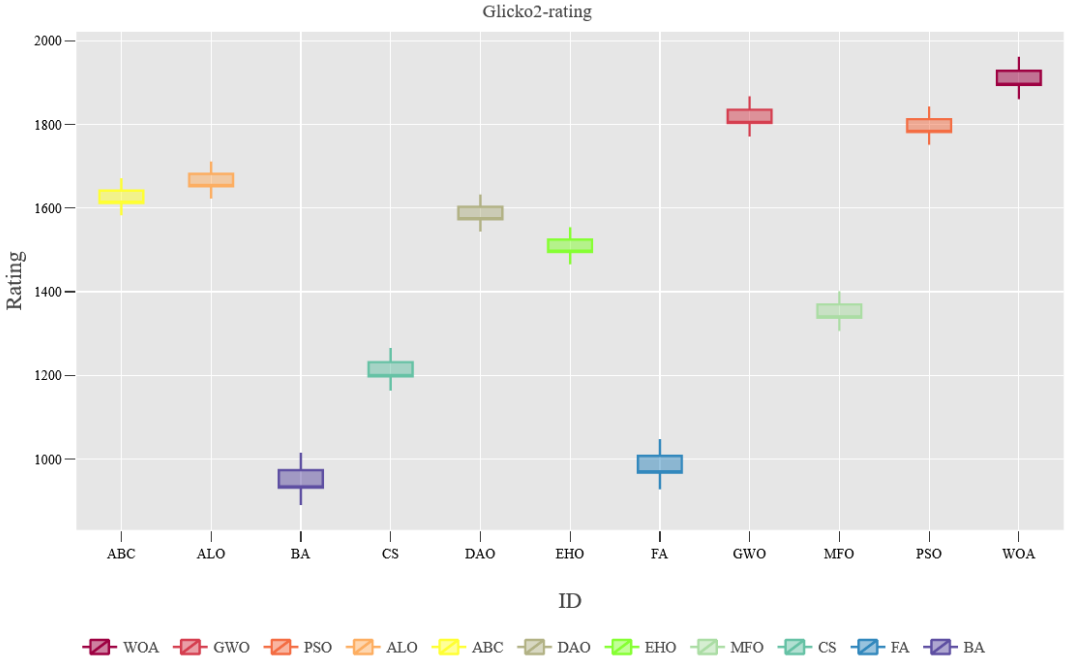
7.6 Výsledky benchmarkingu zlúčených skupín pre DIM15

Rovnako, ako v predchádzajúcich testoch popisovaných v kapitolách 7.4 a 7.5, aj dátaset tohto testu vznikol spojením všetkých skupín. Vybrané testovacie problémy mali dimenziu 15, čo znamená, že fixDIM testovacie funkcie sa v tomto spojenom dátasete už nenachádzajú. Preto algoritmy, ktoré dosahovali lepšiu úspešnosť pri nižších dimenziách môžu získať horšie hodnotenie, ako v prípade osobitných testov na jednotlivých skupinách.

Z *Glicko-2* ohodnotenia na obrázku 93 je možné usúdiť, že výsledky sú podobné, ako v predchádzajúcom teste popísanom v kapitole 7.5. Prvé miesto opäť obsadil WOA algoritmus, nasledovaný algoritmom GWO, s PSO umiestneným iba mierne nižšie v rebríčku. ALO a ABC sa umiestnili s pomerne slušným hodnotením pod spomínanými najlepšími metaheuristikami. To značí, že pri vyšších dimenziách tieto algoritmy dosahujú mierne horšiu úspešnosť. Priemer hodnotenia je opäť tvorený metaheuristikami DAO, EHO a MFO, ktorý v tomto teste oproti predchádzajúcim klesol, čo znamená, že úspešnosť jeho riešenia rovnako ako pri ALO a ABC s narastajúcim počtom premenných klesá. CS nasledoval konvenciu ním nastavenú a umiestnil sa na nižších priečkach, pričom FA a BA obsadili najnižšie umiestnenie.

Bat Algorithm dosahoval najhoršie výsledky v doposiaľ každom teste. Tento fakt môže byť spôsobený jeho vysokou náhodnosťou, ktorú je možné zredukovať

modifikáciami algoritmu. V tejto implementácií bola síce použitá základná varianta, avšak špecifické parametre si počas iterácií upravovala automaticky. Napriek tomu nedokáže úspešne riešiť väčšinu problémov a v porovnaní s ostatnými súťažiacimi výrazne zaostáva.



Obr. 42 GLicko-2 hodnotenie zlúčeného dátasetu s DIM15

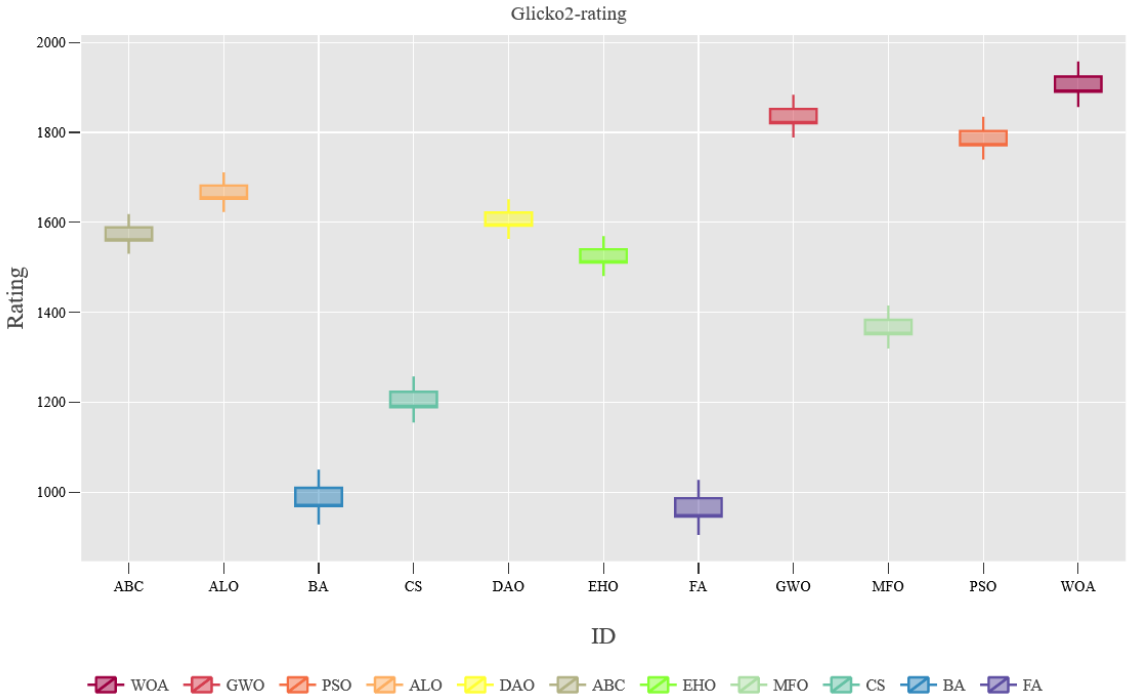
7.7 Výsledky benchmarkingu zlúčených skupín pre DIM20

Testy boli prevedené na problémoch s dimenziou 20 z dátasetu získaného zlúčením všetkých skupín. Výsledky sú takmer zhodné s predchádzajúcim testom popísaný v kapitole 7.6, avšak s miernymi zmenami. Na prvej priečke sa opäť umiestnil algoritmus WOA, nasledovaný algoritmami GWO a PSO. Mierne horšie, no stále skutočne uspokojivé výsledky dosiahli algoritmy ALO, DAO a ABC v tomto poradí. Tu je vidieť zmenu oproti predchádzajúcemu testu, kde Artificial Bee Colony dosahoval lepšie výsledky ako Dragonfly Algorithm, ktorý ho na najvyššej dimenzií tesne predbehol v hodnotení. Nasledujú EHO a MFO algoritmy, ktoré dosiahli takmer rovnaké ohodnotenia a umiestnili sa v prostriedku rebríčka. Najspodnejšie priečky sú opäť tvorené algoritmami CS, BA a FA. Prvý spomínaný je stále lepší ako zvyšné dva, pričom opäť došlo k zmene oproti predchádzajúcemu testu. Pri najvyššej dimenzii sa pri zlúčených skupinách BA umiestnil vyššie ako FA. Napriek tomu dosiahol iba skutočne mierne lepšie výsledky a tak jeho výkonnosť môžeme stále považovať za najslabšiu.

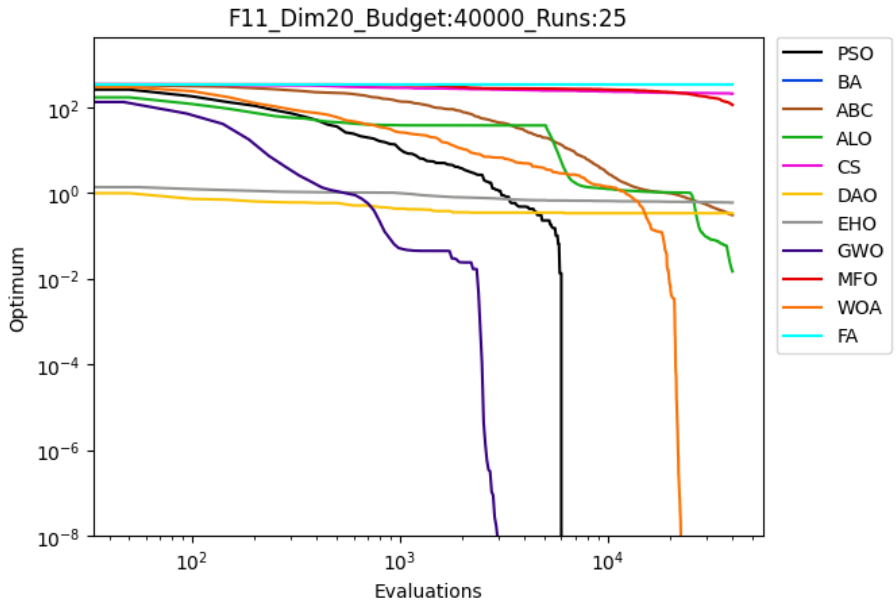
Konvergenčný graf nDIM skupiny na obrázku 95 potvrdzuje výsledky *Glicko-2* ratingu. Dá sa konštatovať, že s výnimkou zigzag problémov dokážu niektoré algoritmy nájsť výsledok s pomerne slušnou presnosťou aj pri vysokých dimenziách. Nižší rozpočet, ktorý sa pri zigzag ukázal nedostatočný, bol nastavený v súlade s výpočetným výkonom osobného počítača. Rovnako, ako v kapitole 7.3, môžeme tvrdiť, že algoritmy

sú schopné správnej konvergencie, čo potvrdzuje aj konvergenčný graf zigzag skupiny na obrázku 96.

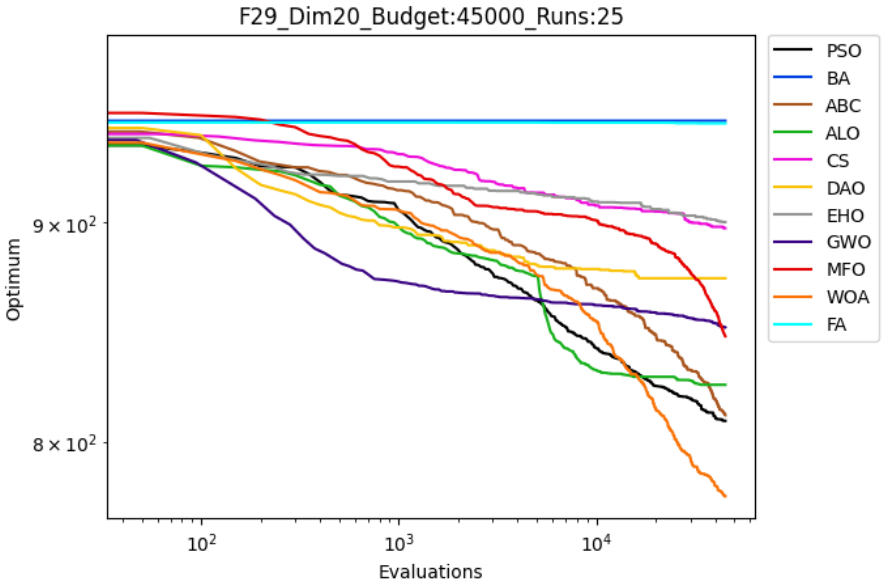
Všetky Glicko-2 ratingy sú okrem obrázkov hodnotení charakterizované aj prostredníctvom tabuliek, kde sa nachádzajú jednotlivé premenné popísané v kapitole 6.5. Tieto výsledky je možné nájsť v prílohe 3.



Obr. 43 Glicko-2 hodnotenie zlúčeného dátasetu s DIM20



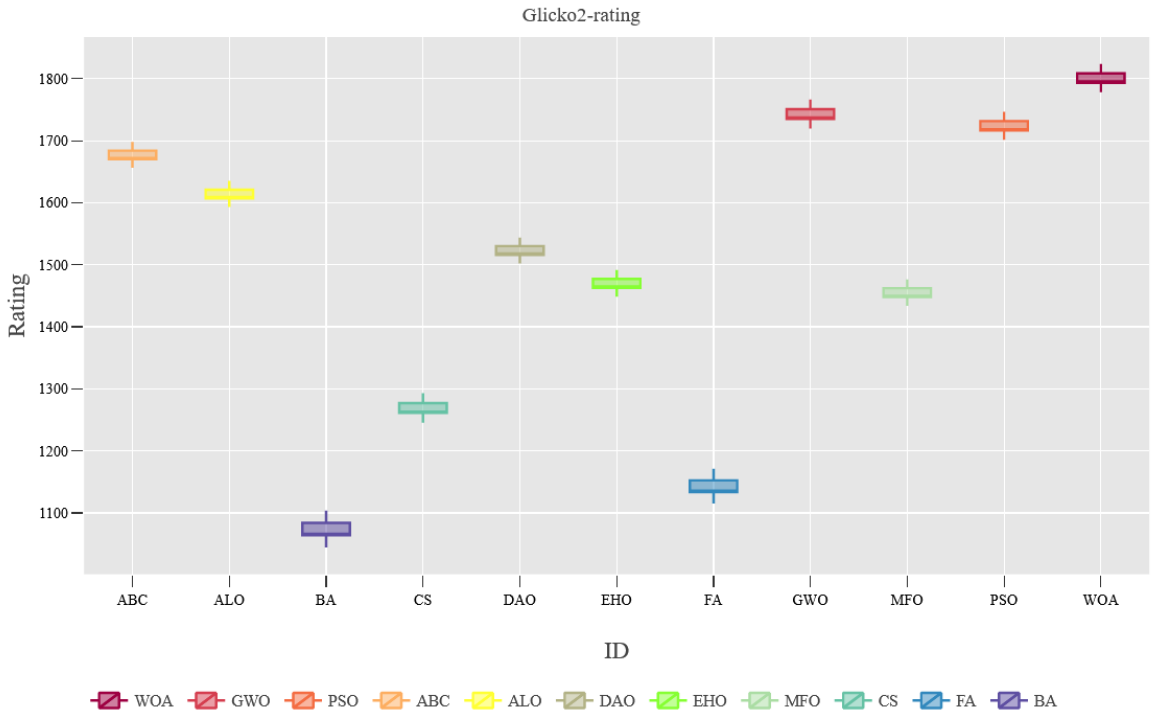
Obr. 44 Konvergenčný graf multimodálnej funkcie F11 zo skupiny nDIM



Obr.45 Konvergenčný graf multimodálnej funkcie F29 zo skupiny zigzag

7.8 Výsledky benchmarkingu zlúčených skupín pre všetky dimenzie

Posledný test bol prevedený na zlúčenom dátasete, ktorý obsahoval všetky funkcie na všetkých dimenziách. Jeho výsledok je možne vidieť na obrázku 97.



Obr.46 Glicko-2 hodnotenie zlúčeného dátasetu so všetkými dimenziami

Výsledok tohto testu zhrnul a potvrdil správnosť predchádzajúcich hodnotení. Najvyššie v rebríčku sa umiestnil algoritmus WOA, nasledovaný algoritmami GWO a PSO. Riešenie dosiahli pomerne rýchlo aj na vysokých dimenziách (napríklad pri F13 D20 za

4.23s, 5.92s a 2.92s) a tiež s pomerne dobrou presnosťou. Môžeme ich teda súhrnne označiť za najlepšie. Stále úspešné metahueristiky poskytujúce dobré výsledky predstavovali ABC, ALO a DAO, ktorých pozícia sa iba mierne menila počas testov na zlúčenom dátasete oproti testom prevádzaným na jednotlivých skupinách osobitne. ABC algoritmus pracoval presnejšie na problémoch s menším počtom premenných, čo znamená, že oproti testom na individuálnych skupinách sa umiestnil mierne nižšie. Rovnaký princíp platí aj pre algoritmus ALO, ktorý napriek jeho dobrej výkonnosti môžeme označiť za časovo najnáročnejší zo všetkých súťažiacich. DAO taktiež predstavoval metaheuristiku, ktorá potrebovala na vyriešenie problému viac času. Oproti ALO išlo síce zväčša o tretinu ním využitého času, avšak pri porovnaní s ostatnými algoritmami to bol stále veľký rozdiel. Metaheuristiky EHO a MFO opakovane obsadzovali priemer hodnotenia a umiestnili sa v prostriedku grafu. Napriek tomu, že nedosahovali takú úspešnosť ako ALO, ABC a DAO, môžeme tvrdiť, že ich časová náročnosť je značne nižšia. Predovšetkým algoritmus EHO predstavoval jeden z najrýchlejších. Presnosť riešenia EHO a MFO rovnako kolísala s narastajúcim počtom premenných, avšak EHO celkovo dosahoval lepšie výsledky ako MFO. Algoritmus CS získaval vo všetkých testoch nižšie hodnotenie v porovnaní s predchádzajúcimi popísanými súťažiacimi. Napriek tomu si však vždy viedol lepšie ako BA a FA metaheuristiky. Firefly Algorithm sa opakovane umiestňoval na posledných priečkach algoritmu, ale až na jednu výnimku tvorenú testom na zlúčenom dátasete s 15D problémami, vždy vyššie ako Bat Algorithm. Ten zo všetkých súťažiacich dosiahol najhoršie výsledky v testoch takmer všetkých kategórií.

Správnosť jednotlivých testov je tiež podporená štatistickými výsledkami exportovanými implementovaným programom a tabuľkovými výsledkami Glicko-2 hodnotení, ktoré sú k dispozícií v prílohe 3.

8. ZÁVER

Táto práca sa zaoberala benchmarkingom pre rojové optimalizačné algoritmy. Jedná sa o populačne založené metaheuristiky, ktoré patria do skupiny prírodou inšpirovaných algoritmov.

Ako prvý bol popísaný benchmarking ako metodológia sledovania a hodnotenia algoritmov využívajúca výkonnostné kritéria k ich analýze. Okrem toho boli stručne popísané jednotlivé software platformy, ktoré umožňujú automatizáciu vyhodnocovania prostredníctvom integrovaných kriteriálnych metód, či vizualizáciu.

Následne boli popísané rojové algoritmy, ktoré boli implementované pre účely práce. Táto programová realizácia bola uskutočnená v programovacom jazyku *Python* a vývojovom prostredí *PyCharm*. Každý algoritmus bol písaný s využitím princípov objektovo orientovaného programovania a bol obsiahnutý v samostatnom pyskripte. Okrem týchto metaheuristík boli vybrané a implementované aj testovacie problémy, v ktorých riešení algoritmy súťažili. Programová reprezentácia týchto optimalizačných problémov bola vytvorená v osobitnom pyskripte, pričom rovnako ako v prípade algoritmov bolo využité techník objektovo orientovaného programovania, s každým problémom reprezentovaným triedou.

Implementovaný program zbieral a ukladal informácie v priebehu riešenia problémov jednotlivými algoritmami, ktoré následne exportoval. Ako hlavné kritérium výkonnosti bol zvolený *fixed-budget*. Časť z týchto informácií bola spracovaná a predaná ako vstupné údaje benchmarkingovej platforme *IOHanalyzer*, prostredníctvom ktorého bolo prevedených niekoľko sérií testov. S kombináciou analýzy poskytnutej exportovanými dátami a výsledkov týchto testov bola vyhodnotená výkonnosť algoritmov pri riešení daných problémov.

Posledná kapitola sa zaoberala práve popisom spomínaných testov a zhodnotením všetkých výsledkov. Jednotlivé testy boli podporené grafmi *Glicko-2* ratingov poskytnutých *IOHanalyzerom* a konvergenčnými grafmi. K zhodnoteniu tiež poslúžili štatistické údaje, ktoré rovnako ako konvergenčné grafy boli výstupmi programovej implementácie.

Ako výsledok benchmarkingu môžeme konštatovať, že ako najúspešnejšia z vybraných metaheuristík skončila *Whale Optimization Algorithm*. Tesne za ním sa v rebríčku umiestnili *Grey Wolf Optimization* a *Particle Swarm Optimization* s takmer rovnakým hodnotením. Ako stále úspešné môžeme označiť algoritmy *Artificial Bee Colony, Antlion Optimization* a *Dragonfly Optimization*, ktorých presnosť síce mierne kolísala pri zvyšujúcich sa dimenziách problémov, no stále boli schopné poskytnúť skutočne uspokojivé výsledky. Priemer rebríčka tvorili algoritmy *Elephant Herding Optimization* a *Moth Flame Optimization*. Výrazne slabšie výsledky poskytovala metaheuristika *Cuckoo Search*, avšak s najhorším skončili algoritmy *Firefly Optimization* a *Bat Algorithm* v tomto poradí. Práve *Bat Algorithm* môžeme suverénne označiť ako algoritmus, ktorý si pri daných testovacích funkciách viedol najhoršie.

ZOZNAM POUŽITEJ LITERATÚRY

[1] NOCEDAL, Jorge a Stephen WRIGHT. Numerical Optimization. Second Edition. Springer Science+Business Media, 2006. ISBN 978-0387-30303-1.

[2] BONNANS, J. Frédéric, J. Charles GILBERT, Claude LEMARÉCHAL a Claudia A. SAGASTIZÁBAL. Numerical Optimization: Theoretical and Practical Aspects. 2nd Edition. Berlin, Heidelberg: Springer, 2006. ISBN 978-3-540-35447-5.

[3] Lin, Ming-Hua & Tsai, Jung-Fa & Yu, Chian-Son. (2012). A Review of Deterministic Optimization Methods in Engineering and Management. Mathematical Problems in Engineering. 2012. 10.1155/2012/756023.

[4] JOHNSON, Steven G. A Brief Overview of Optimization Problems [online]. MIT, 2008 [cit. 2022-05-03]. URL: https://math.mit.edu/~stevenj/18.335/optimization.pdf

[5] Test functions for optimization needs [online]. 2005 [cit. 2022-05-03]. URL: https://robertmarks.org/Classes/ENGR5358/Papers/functions.pdf

[6] THEVENOT, Axel. Optimization & Eye Pleasure: 78 Benchmark Test Functions for Single Objective Optimization [online]. Towards Data Science, Dec 31, 2020 [cit. 2022-05-03]. URL: https://towardsdatascience.com/optimization-eye-pleasure-78-benchmark-test-functions-for-single-objective-optimization-92e7ed1d1f12

[7] Surjanovic, S. & Bingham, D. (2013). Virtual Library of Simulation Experiments: Test Functions and Datasets. Retrieved May 3, 2022, from http://www.sfu.ca/~ssurjano.

[8] Bartz-Beielstein, Thomas & Doerr, Carola & Bossek, Jakob & Chandrasekaran, Sowmya & Eftimov, Tome & Fischbach, Andreas & Kerschke, Pascal & López-Ibáñez, Manuel & Malan, Katherine & Moore, Jason & Naujoks, Boris & Orzechowski, Patryk & Volz, Vanessa & Wagner, Markus & Weise, Thomas. (2020). Benchmarking in Optimization: Best Practice and Open Issues.

[9] Schott, Francois & Chamoret, Dominique & Baron, Thomas & Salmon, Sébastien & Meyer, Yann. (2021). Performance measure and tool for benchmarking metaheuristic optimization algorithms. Journal of Applied and Computational Mechanics. 7. 10.22055/JACM.2021.37664.3060.

[10] Doerr, Carola & Wang, Hao & Ye, Furong & Van Rijn, Sander & Bäck, Thomas. (2018). IOHprofiler: A Benchmarking and Profiling Tool for Iterative Optimization Heuristics.

[11] Hansen, Nikolaus & Auger, Anne & Brockhoff, Dimo & Tušar, Dejan & Tusar, Tea. (2016). COCO: Performance Assessment.

[12] Kreutz, C. Guidelines for benchmarking of optimization-based approaches for fitting mathematical models. Genome Biol 20, 281 (2019). https://doi.org/10.1186/s13059-019-1887-9

[13] GOYAL, Sanchit. A survey on travelling salesman problem. In: Midwest instruction and computing symposium. 2010. p. 1-9.

[14] BONYADI, Mohammad Reza; MICHALEWICZ, Zbigniew; BARONE, Luigi. The travelling thief problem: The first step in the transition from theoretical problems to realistic problems. In: 2013 IEEE Congress on Evolutionary Computation. IEEE, 2013. p. 1037-1044.

[15] IOHprofiler: Iterative Optimization Heuristics Profiler [online]. [cit. 2022-05-03]. URL: https://iohprofiler.github.io

[16] Gu, J. (1994). Optimization Algorithms for the Satisfiability (SAT) Problem. In: Du, DZ., Sun, J. (eds) Advances in Optimization and Approximation. Nonconvex Optimization and

Its Applications, vol 1. Springer, Boston, MA. https://doi.org/10.1007/978-1-4613-3629-7_6

[17] Weise, Thomas & Chiong, Raymond & Laessig, Joerg & Tang, Ke & Tsutsui, Shigeyoshi & Chen, Wenxiang & Michalewicz, Zbigniew & Yao, Xin. (2014). Benchmarking Optimization Algorithms: An Open Source Framework for the Traveling Salesman Problem. Computational Intelligence Magazine, IEEE. 9. 40-52. 10.1109/MCI.2014.2326101.

[18] HANSEN, Nikolaus, et al. COCO: A platform for comparing continuous optimizers in a black-box setting. Optimization Methods and Software, 2021, 36.1: 114-144.

[19] BROCKHOFF, Dimo, et al. Biobjective performance assessment with the COCO platform. arXiv preprint arXiv:1605.01746, 2016.

[20] Black Box Optimization Competition: BBComp [online]. [cit. 2022-05-09]. URL: https://www.ini.rub.de/PEOPLE/glasmtbl/projects/bbcomp/

[21] Bossek, J. (2017). Ecr 2.0: A Modular Framework for Evolutionary Computation > in R. In Proceedings of the Genetic and Evolutionary Computation Conference > (GECCO) Companion (pp. 1187–1193). Berlin, Germany: > ACM. http://doi.org/10.1145/3067695.3082470

[22] HeuristicLab: A Paradigm-Independent and Extensible Environment for Heuristic Optimization [online]. [cit. 2022-05-03]. URL: https://dev.heuristiclab.com/trac.fcgi/

[23] Sean Luke. ECJ Evolutionary Computation Library (1998). Available for free at http://cs.gmu.edu/~eclab/projects/ecj/

[24] Fortin, F.-A., De Rainville, F.-M., Gardner, M.-A., Parizeau, M., & Christian Gagné. (2012). DEAP: Evolutionary Algorithms Made Easy . Journal of Machine Learning Research , 13, 2171–2175.

[25] Rapin, J., & Teytaud, O. (2018). Nevergrad - A gradient-free optimization platform. In GitHub repository. GitHub. https://GitHub.com/FacebookResearch/Nevergrad

[26] S. Wessing. optproblems: Infrastructure to define optimization problems and some test problems for black-box optimization, 2016. Python package version 0.6. https://pypi.python.org/pypi/optproblems.

[27] Platypus: Multiobjective Optimization in Python [online]. [cit. 2022-05-03]. URL: https://platypus.readthedocs.io/en/latest/index.html

[28] LÓPEZ-IBÁÑEZ, Manuel, et al. The irace package: Iterated racing for automatic algorithm configuration. Operations Research Perspectives, 2016, 3: 43-58.

[29] Ventura, S., Romero, C., Zafra, A. *et al.* JCLEC: a Java framework for evolutionary computation. *Soft Comput* **12,** 381–392 (2008). https://doi.org/10.1007/s00500-007-0172-0

[30] DURILLO, Juan J.; NEBRO, Antonio J. jMetal: A Java framework for multi-objective optimization. *Advances in Engineering Software*, 2011, 42.10: 760-771.

[31] Kerschke, P., Trautmann, H. (2019). Comprehensive Feature-Based Landscape Analysis of Continuous and Constrained Optimization Problems Using the R-Package Flacco. In: Bauer, N., Ickstadt, K., Lübke, K., Szepannek, G., Trautmann, H., Vichi, M. (eds) Applications in Statistical Computing. Studies in Classification, Data Analysis, and Knowledge Organization. Springer, Cham. https://doi.org/10.1007/978-3-030-25147-5_7

[32] MOEA Framework: A Free and Open Source Java Framework for Multiobjective Optimization [online]. [cit. 2022-05-03]. URL: http://moeaframework.org/

[33] Paradiseo: a Heuristic Optimization Framework [online]. [cit. 2022-05-03]. URL: https://nojhan.github.io/paradiseo/

[34] Ye Tian, Ran Cheng, Xingyi Zhang, and Yaochu Jin, PlatEMO: A MATLAB Platform for Evolutionary Multi-Objective Optimization [Educational Forum], IEEE Computational Intelligence Magazine, 2017, 12(4): 73-87.

[35] Bossek, J. (2017). smoof: Single- and Multi-Objective Optimization Test Functions. The R Journal, 9(1), 103–113. https://doi.org/10.32614/RJ-2017-004

[36] Hutter, Frank & Hoos, Holger & Leyton-Brown, Kevin & Stützle, Thomas. (2009). ParamILS: An Automatic Algorithm Configuration Framework. J. Artif. Intell. Res. (JAIR). 36. 267-306. 10.1613/jair.2861.

[37] Lindauer, M., Eggensperger, K., Feurer, M., Biedenkapp, A., Deng, D., Benjamins, C., Ruhkopf, T., Sass, R., & Hutter, F. (2021). SMAC3: A Versatile Bayesian Optimization Package for Hyperparameter Optimization. ArXiv: 2109.09831. https://arxiv.org/abs/2109.09831

[38] mlr: Machine learning in R [online]. [cit. 2022-05-07]. URL: https://mlr.mlr-org.com/

[39] Bischl, B., Richter, J., Bossek, J., Horn, D., Thomas, J., & Lang, M. (2017). mlrMBO: A Modular Framework for Model-Based Optimization of Expensive Black-Box Functions. https://arxiv.org/abs/1703.03373

[40] Gijsbers, P., LeDell, E., Thomas, J., Poirier, S., Bischl, B., & Vanschoren, J. (2019). An Open Source AutoML Benchmark. arXiv. https://doi.org/10.48550/ARXIV.1907.00909

[41] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, E. (2011). Scikit-learn: Machine Learning in Python. Journal of Machine Learning Research, 12, 2825–2830.

[42] Igel, C., Heidrich-Meisner, V., & Glasmachers, T. (2008). Shark. Journal of Machine Learning Research, 9, 993–996.

[43] Bartz-Beielstein, T., Zaefferer, M., and Rehbach, F. In a Nutshell – The Sequential Parameter Optimization Toolbox. arXiv e-prints (Dec. 2021), https://arxiv.org/abs/1712.04076.

[44] HASSANIEN, Aboul Ella; EMARY, Eid. Swarm intelligence: principles, advances, and applications. CRC Press, 2018.

[45] Bansal, J.C., Pal, N.R. (2019). Swarm and Evolutionary Computation. In: Bansal, J., Singh, P., Pal, N. (eds) Evolutionary and Swarm Intelligence Algorithms. Studies in Computational Intelligence, vol 779. Springer, Cham. https://doi.org/10.1007/978-3-319-91341-4_1

[46] Du, KL., Swamy, M.N.S. (2016). Particle Swarm Optimization. In: Search and Optimization by Metaheuristics. Birkhäuser, Cham. https://doi.org/10.1007/978-3-319-41192-7_9

[47] Poli, R., Kennedy, J. & Blackwell, T. Particle swarm optimization. Swarm Intell 1, 33–57 (2007). https://doi.org/10.1007/s11721-007-0002-0

[48] J. Kennedy and R. Eberhart, "Particle swarm optimization," Proceedings of ICNN'95 - International Conference on Neural Networks, 1995, pp. 1942-1948 vol.4, doi: 10.1109/ICNN.1995.488968.

[49] Karaboga, D., Basturk, B. A powerful and efficient algorithm for numerical function optimization: artificial bee colony (ABC) algorithm. J Glob Optim 39, 459–471 (2007). https://doi.org/10.1007/s10898-007-9149-x

[50] KARABOGA, Dervis; AKAY, Bahriye. A comparative study of artificial bee colony algorithm. Applied mathematics and computation, 2009, 214.1: 108-132.

[51] KARABOGA, Dervis; OZTURK, Celal. A novel clustering approach: Artificial Bee Colony (ABC) algorithm. Applied soft computing, 2011, 11.1: 652-657.

[52] MIRJALILI, Seyedali. The ant lion optimizer. Advances in engineering software, 2015, 83: 80-98.

[53] Mani M., Bozorg-Haddad O., Chu X. (2018) Ant Lion Optimizer (ALO) Algorithm. In: Bozorg-Haddad O. (eds) Advanced Optimization by Nature-Inspired Algorithms. Studies in Computational Intelligence, vol 720. Springer, Singapore. https://doi.org/10.1007/978-981-10-5221-7_11

[54] Kilic, Haydar & Yüzgeç, Uğur. (2021). Improved Antlion Optimization Algorithm for Quadratic Assignment Problem. Malaysian Journal of Computer Science. 34. 2021. 10.22452/mjcs.vol34no1.3.

[55] Yang XS. (2014) Cuckoo Search and Firefly Algorithm: Overview and Analysis. In: Yang XS. (eds) Cuckoo Search and Firefly Algorithm. Studies in Computational Intelligence, vol 516. Springer, Cham. https://doi.org/10.1007/978-3-319-02141-6_1

[56] Yang, XS., Deb, S. Cuckoo search: recent advances and applications. Neural Comput & Applic 24, 169–174 (2014). https://doi.org/10.1007/s00521-013-1367-1

[57] KAMARUZAMAN, Anis Farhan, et al. Levy flight algorithm for optimization problems-a literature review. *Applied mechanics and materials*, 2013, 421: 496-501.

[58] WEISSTEIN, Eric W. Heaviside step function. https://mathworld. wolfram. com/, 2002.

[59] PAVLYUKEVICH, Ilya. Lévy flights, non-local search and simulated annealing. journal of computational physics, 2007, 226.2: 1830-1844.

[60] Gandomi, A.H., Yang, XS. & Alavi, A.H. Cuckoo search algorithm: a metaheuristic approach to solve structural optimization problems. Engineering with Computers 29, 17–35 (2013). https://doi.org/10.1007/s00366-011-0241-y

[61] Mirjalili, S. Dragonfly algorithm: a new meta-heuristic optimization technique for solving single-objective, discrete, and multi-objective problems. Neural Comput & Applic 27, 1053–1073 (2016). https://doi.org/10.1007/s00521-015-1920-1

[62] Mafarja, M., Heidari, A.A., Faris, H., Mirjalili, S., Aljarah, I. (2020). Dragonfly Algorithm: Theory, Literature Review, and Application in Feature Selection. In: Mirjalili, S., Song Dong, J., Lewis, A. (eds) Nature-Inspired Optimizers. Studies in Computational Intelligence, vol 811. Springer, Cham. https://doi.org/10.1007/978-3-030-12127-3_4

[63] Yang, Xin-She & Gandomi, Amir. (2012). Bat Algorithm: A Novel Approach for Global Engineering Optimization. Engineering Computations. 29. 10.1108/02644401211235834.

[64] Yang, Xin-She. (2013). Bat Algorithm: Literature Review and Applications. International Journal of Bio-Inspired Computation. 5. 10.1504/IJBIC.2013.055093.

[65] Gandomi, A.H., Yang, XS., Alavi, A.H. et al. Bat algorithm for constrained optimization tasks. Neural Comput & Applic 22, 1239–1255 (2013). https://doi.org/10.1007/s00521-012-1028-9

[66] LI, Wei; WANG, Gai-Ge; ALAVI, Amir H. Learning-based elephant herding optimization algorithm for solving numerical optimization problems. Knowledge-Based Systems, 2020, 195: 105675

[67] G. -G. Wang, S. Deb and L. d. S. Coelho, "Elephant Herding Optimization," 2015 3rd International Symposium on Computational and Business Intelligence (ISCBI), 2015, pp. 1-5, doi: 10.1109/ISCBI.2015.8.

[68] ELHOSSEINI, Mostafa A., et al. On the performance improvement of elephant herding optimization algorithm. Knowledge-Based Systems, 2019, 166: 58-70.

[69] MIRJALILI, Seyedali; MIRJALILI, Seyed Mohammad; LEWIS, Andrew. Grey wolf optimizer. Advances in engineering software, 2014, 69: 46-61.

[70] S. Sharma, R. Salgotra and U. Singh, "An enhanced grey wolf optimizer for numerical optimization," 2017 International Conference on Innovations in Information, Embedded and Communication Systems (ICIIECS), 2017, pp. 1-6, doi: 10.1109/ICIIECS. 2017.8275908.

[71] MIRJALILI, Seyedali. Moth-flame optimization algorithm: A novel nature-inspired heuristic paradigm. Knowledge-based systems, 2015, 89: 228-249.

[72] Bahrami, M., Bozorg-Haddad, O., Chu, X. (2018). Moth-Flame Optimization (MFO) Algorithm. In: Bozorg-Haddad, O. (eds) Advanced Optimization by Nature-Inspired Algorithms. Studies in Computational Intelligence, vol 720. Springer, Singapore. https://doi.org/10.1007/978-981-10-5221-7_13

[73] Shehab, M., Abualigah, L., Al Hamad, H. et al. Moth–flame optimization algorithm: variants and applications. Neural Comput & Applic 32, 9859–9884 (2020). https://doi.org/10.1007/s00521-019-04570-6

[74] MIRJALILI, Seyedali; LEWIS, Andrew. The whale optimization algorithm. Advances in engineering software, 2016, 95: 51-67.

[75] Salgotra, R., Singh, U. & Saha, S. On Some Improved Versions of Whale Optimization Algorithm. Arab J Sci Eng 44, 9653–9691 (2019). https://doi.org/10.1007/s13369-019-04016-0

[76] K. Kaur, R. Salgotra and U. Singh, "An improved firefly algorithm for numerical optimization," 2017 International Conference on Innovations in Information, Embedded and Communication Systems (ICIIECS), 2017, pp. 1-5, doi: 10.1109/ICIIECS. 2017.8275914.

[77] WATANABE, Osamu; ZEUGMANN, Thomas (ed.). Stochastic Algorithms: Foundations and Applications: 5th International Symposium, SAGA 2009 Sapporo, Japan, October 26-28, 2009 Proceedings. Springer Science & Business Media, 2009.

[78] Official Python Documentation [online]. [cit. 2022-05-07]. URL: https://www.python.org/doc/

[79] PyCharm Features [online]. [cit. 2022-05-07]. URL: https://www.jetbrains.com/pycharm/features/

[80] J. Kudela and R. Matousek, "New Benchmark Functions for Single-Objective Optimization Based on a Zigzag Pattern," in IEEE Access, vol. 10, pp. 8262-8278, 2022, doi: 10.1109/ACCESS.2022.3144067.

[81] IOHanalyzer: a free online service [online]. [cit. 2022-05-07]. URL:https://iohanalyzer.liacs.nl/

[82] Veček, Niki & Črepinšek, Matej & Mernik, Marjan & Hrnčič, Dejan. (2014). A Comparison between Different Chess Rating Systems for Ranking Evolutionary Algorithms. 2014 Federated Conference on Computer Science and Information Systems, FedCSIS 2014. 511-518. 10.15439/2014F33.

[83] Glicko rating system [online]. [cit. 2022-05-07]. URL: http://www.glicko.net/glicko/glicko.pdf

[84] Glicko2 rating system [online]. [cit. 2022-05-07]. URL:http://www.glicko.net/glicko/glicko2.pdf

[85] Xin-She Yang (2022). Firefly Algorithm (https://www.mathworks.com/matlabcentral/fileexchange/29693-firefly-algorithm), MATLAB Central File Exchange. Retrieved May 8, 2022.

[86] Yang, X.-S., and Deb, S. (2010), "Engineering Optimisation by Cuckoo Search",Int. J. Mathematical Modelling and Numerical Optimisation, Vol. 1, No. 4, 330–343 (2010).

[87] XS Yang (2022). The Standard Bat Algorithm (BA) (https://www.mathworks.com/matlabcentral/fileexchange/74768-the-standard-bat-algorithm-ba), MATLAB Central File Exchange. Retrieved May 8, 2022. Xin-She Yang, Nature-Inspired Optimization Algorithms, Elsevier Insights, (2014)

ZOZNAM SKRATIEK

ABC	Artificial Bee Colony
ALO	Antlion Optimization
BA	Bat Algorithm
CS	Cuckoo Search
DAO	Dragongly Algorithm
EHO	Elephant Herding Optimization
FA	Firefly Algorithm
GWO	Grey Wolf Optimization
MFO	Moth Flame Optimization
PSO	Particle Swarm Optimization
LDWPSO	Lineary decreasing weigth PSO
WOA	Whale Optimization Algorithm
TSP	Travelling salesman problém
TTP	Travelling thief problém
ERT	Expected running time
SAT	Satisfiability problém
PAR	Penalized runtime
PQR	Penalized quantile runtime
ECDF	Empirical cumulative distribution function
PBO	Pseudo boolean optimization
GUI	Graphical user interface

ZOZNAM PRÍLOH

Príloha 1 Implementovaný program a dáta ním exportované v *.zip* prílohe
Príloha 2 Testovacie funkcie – ich vykreslenia, optimá, limity
Príloha 3 Výsledky Glicko-2 testov z kapitoly 7

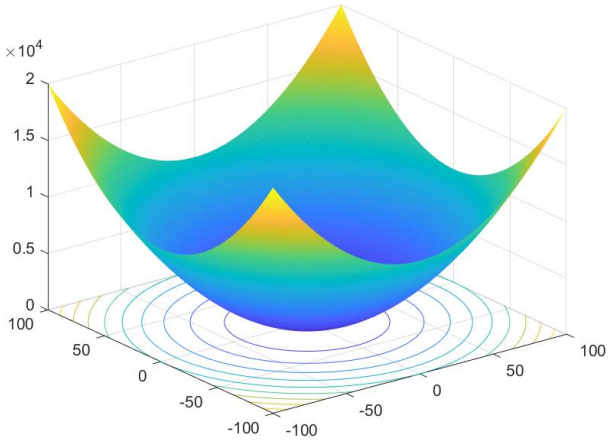
PRÍLOHY

Príloha 1

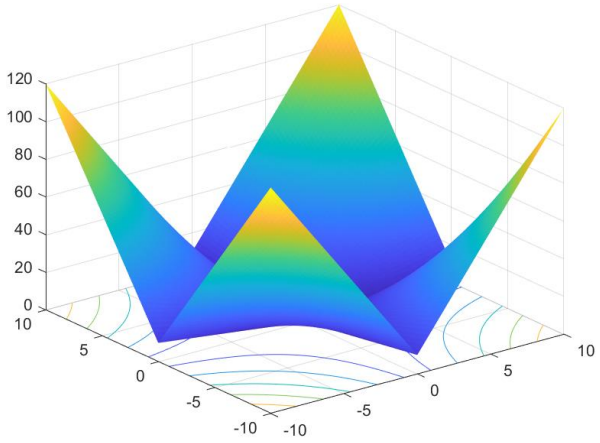
```
DP_Mittas_Eduard_191796_prilohy
├─fix_DIM_statistics-výsledky testov skupiny fixDIM
│   │ F16_benchmarking_statistics
│   │   ├── plots-vykreslenia konvergencie
│   │   │   ├──   F16_D2_convergence_LOG.png
│   │   │   └──   F16_D2_convergence.png
│   │   ├── F16_D2_optimum_data.csv-hodnota optima počas jednotlivých behov algoritmu
│   │   ├── F16_D2_statistics.csv-štatistické údaje zo všetkých behov algoritmu
│   └── ...
├─n_DIM_statistics-výsledky testov skupiny n_DIM
│   │ F1_benchmarking_statistics
│   │   ├── plots-vykreslenia konvergencie
│   │   │   ├──   F1_D2_convergence_LOG.png
│   │   │   └──   F1_D2_convergence.png
│   │   │       ...
│   │   ├── F1_D2_optimum_data.csv-hodnota optima počas jednotlivých behov algoritmu
│   │   ├── F1_D2_statistics.csv-štatistické údaje zo všetkých behov algoritmu
│   │   └── ...
│   ...
├─zig_zag_statistics-výsledky testov skupiny zigzag
│   │  F26_benchmarking_statistics
│   │    ├── plots-vykreslenia konvergencie
│   │    │   ├──  F26_D2_convergence_LOG.png
│   │    │   └──  F26_D2_convergence.png
│   │    │       ...
│   │    ├─ F26_D2_optimum_data.csv-hodnota optima počas jednotlivých behov algoritmu
│   │    ├─ F26_D2_statistics.csv-štatistické údaje zo všetkých behov algoritmu
│   │    └─ ...
│   ...
├─IOHanalyzer_data-upravené dáta použité pri benchmarkingu s využitím IOHanalyzer
│ platformy
│         IOHdata0_fix_DIM.zip
│          ...
│
├─ABC.py-pyskript implemenácie Artificial Bee Colony
├─ALO.py-pyskript implementácie Antlion Optimization
├─BA.py-pyskript implementácie Bat Algorithm
├─CS.py-pyskript implementácie Cuckoo Search
├─DAO.py-pyskript implementácie Dragonfly Aglorithm
├─EHO.py-pyskript implementácie Elephant Herding Optimization
├─FA.py-pyskript implementácie Firefly Algorithm
├─GWO.py-pyskript implementácie Grey Wolf Optimization
├─MFO.py-pyskript implementácie Moth Flame Optimzation
├─PSO.py-pyskript implementácie Particle Swarm Optimization
├─WOA.py-pyskript implementácie Whale Optimization Algorithm
│
├─benchmark_functions.py-pyskript implementácie testovacích funkcií
│
├─optimizer_fixDim.py-pyskript obsahujúci časť hlavného implementujúci testy fixDim
│ skupiny
├─optimizer_Ndim.py-pyskript obsahujúci časť hlavného programu implementujúci testy
│ nDim skupiny
└─optimizer_zigzag.py-pyskript obsahujúci časť hlavného programu implementujúci testy
  zigzag skupiny
```

Príloha 2

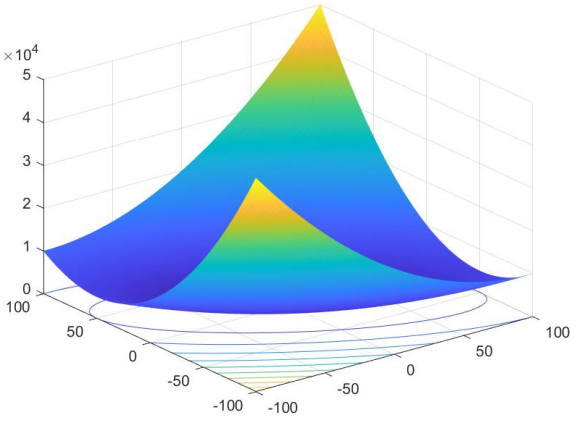
Funkcia F1
Názov: *Sphere Model*
Limity: [-100, 100]
Optimum: 0.0



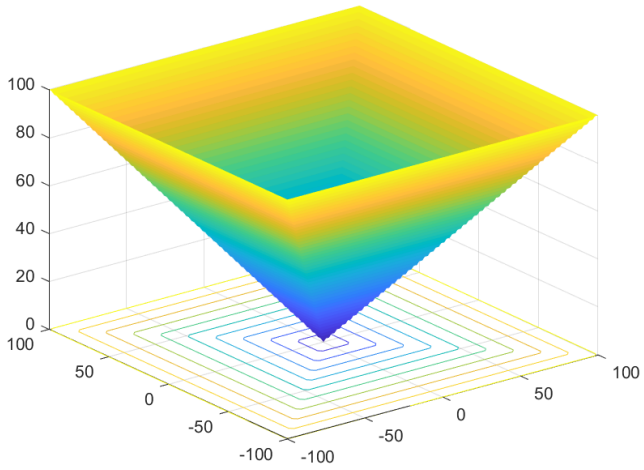
Funkcia F2
Názov: *Schwefel Problem 2.22*
Limity: [-10, 10]
Optimum: 0.0



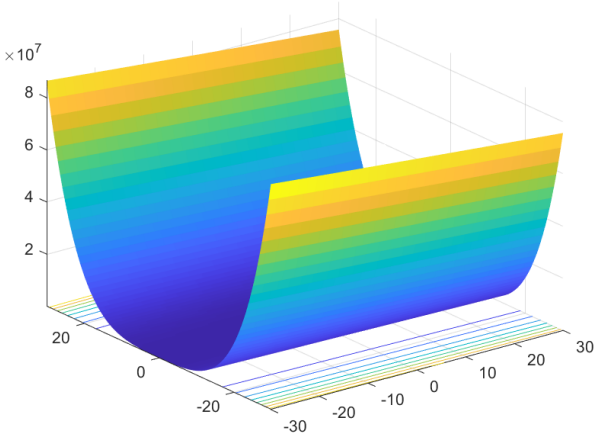
Funkcia F3
Názov: *Schwefel Problem 1.2*
Limity: [-100, 100]
Optimum: 0.0



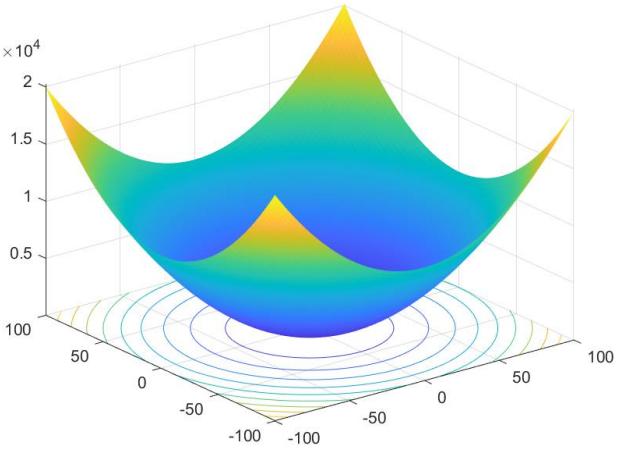
Funkcia F4
Názov: *Schwefel Problem 2.21*
Limity: [-100, 100]
Optimum: 0.0



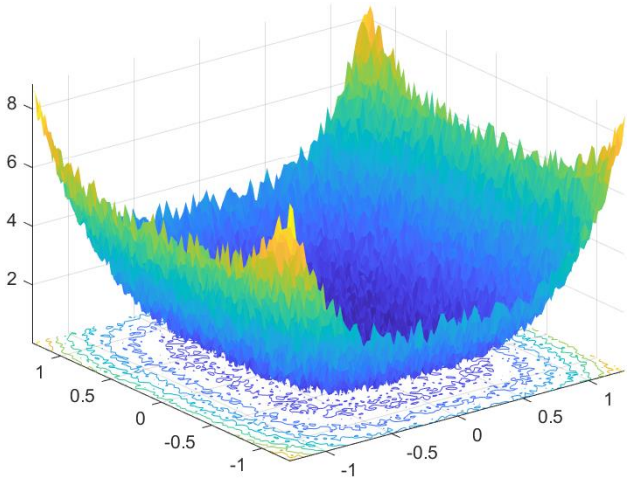
Funkcia F5
Názov: *Generalized Rosenbrock*
Limity: [-30, 30]
Optimum: 0.0



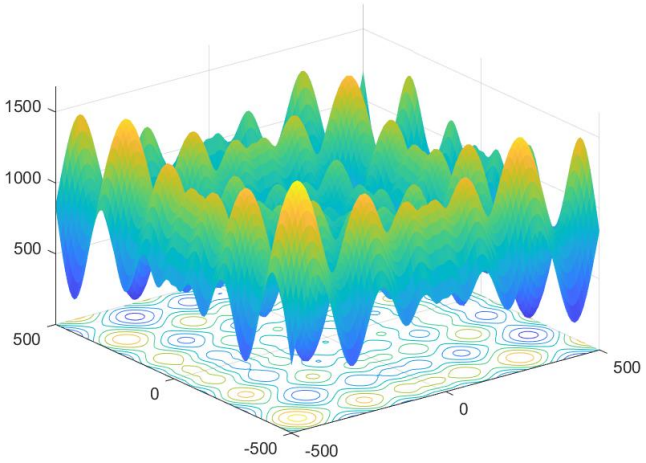
Funkcia F6:
Názov: *Step Function*
Limity: [-100, 100]
Optimum:0.0



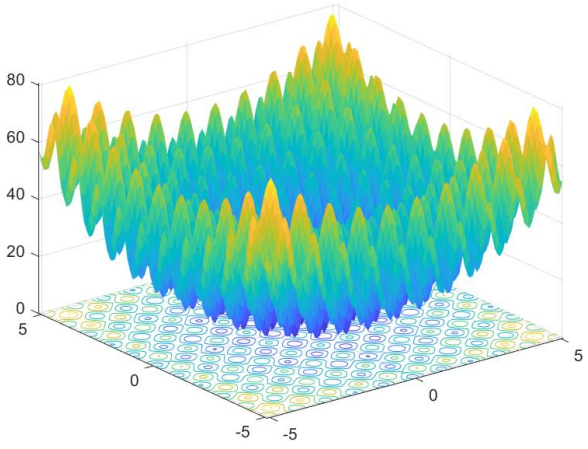
Funkcia F7:
Názov: *Quadratic Function*
Limity: [-1.28, 1.28]
Optimum: 0.0



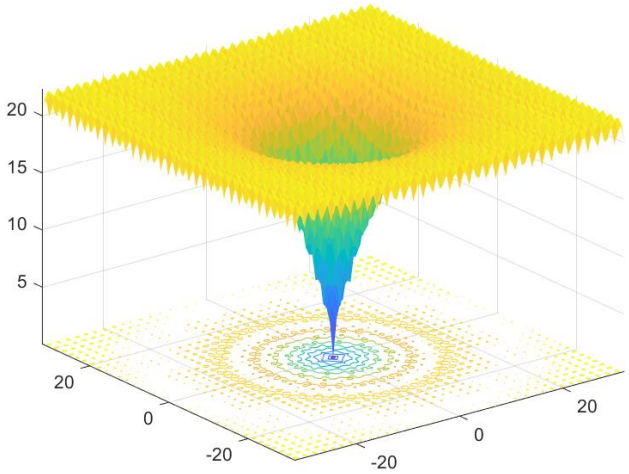
Funkcia F8:
Názov: *Generalized Schwefel*
Limity: [-500, 500]
Optimum: 0.0



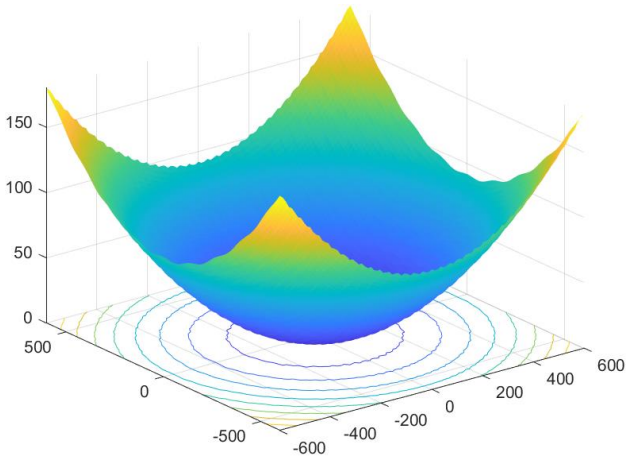
Funkcia F9:
Názov: Generalized Rastrigin
Limity: [-5.12, 5.12]
Optimum: 0.0



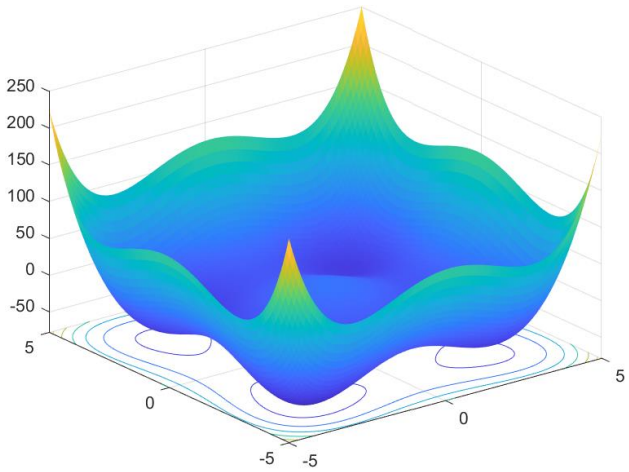
Funkcia F10:
Názov: *Ackkley Function*
Limity: [-32, 32]
Optimum: 0.0



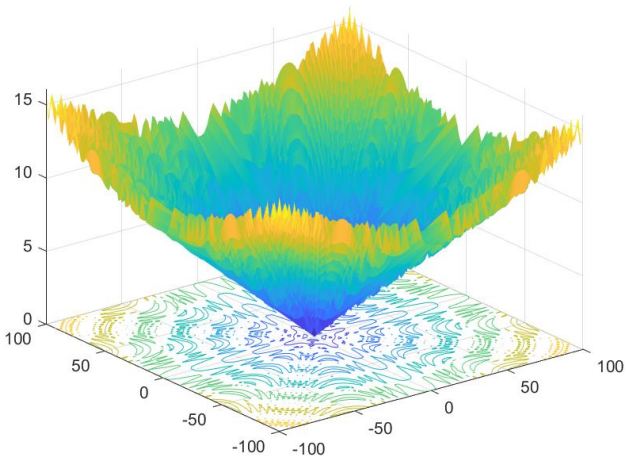
Funkcia F11:
Názov: *Generalized Griewank Function*
Limity: [-600, 600]
Optimum: 0.0



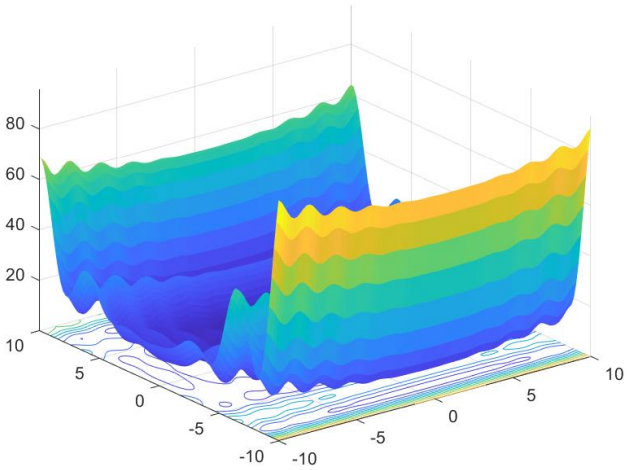
Funkcia F12:
Názov: *Styblinski Tank*
Limity: [-5, 5]
Optimum: 0.0



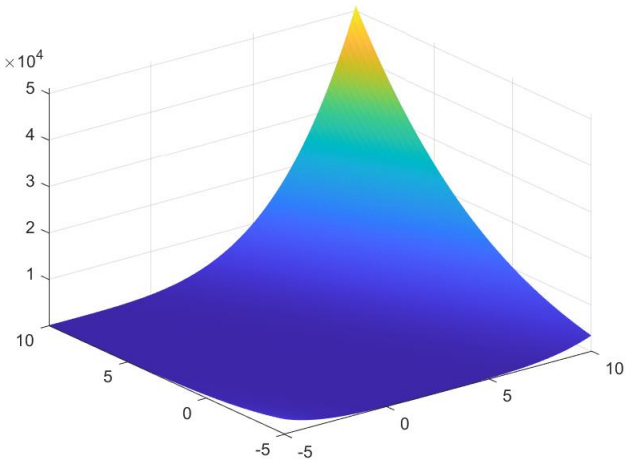
Funkcia F13:
Názov: *Salomon Function*
Limity: [-100, 100]
Optimum: 0.0



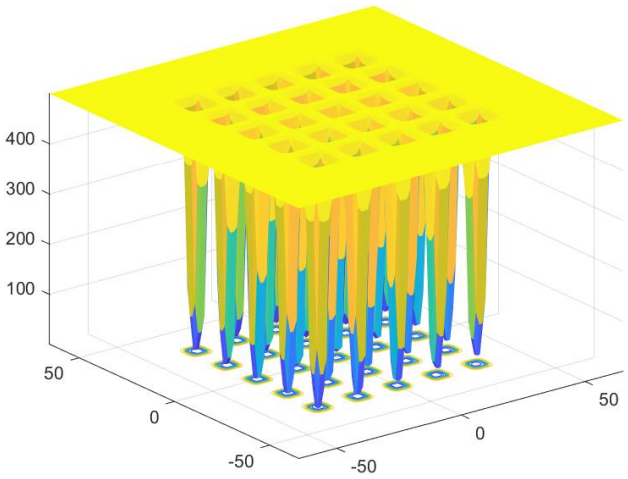
Funkcia F14:
Názov: *Levy Function*
Limity: [-10, 10]
Optimum: 0.0



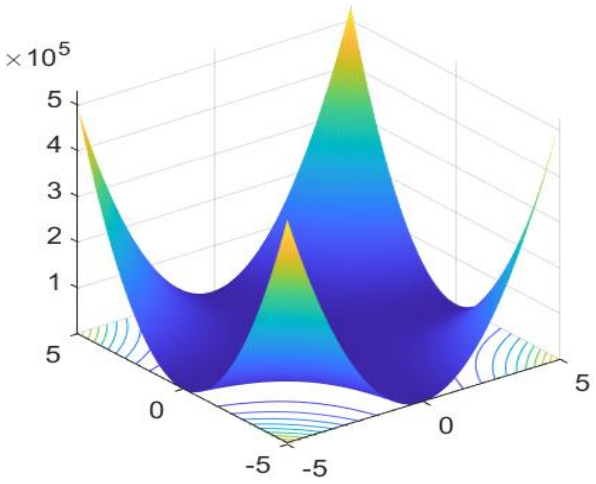
Funkcia F15:
Názov: *Zakharov Function*
Limity: [-5, 10]
Optimum: 0.0



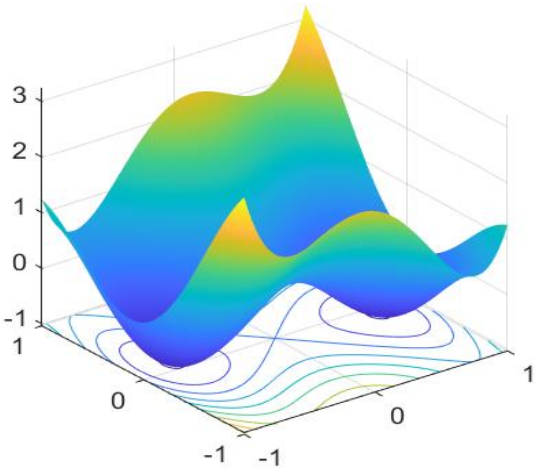
Funkcia F16:
Názov: *M. Shekel's Foxholes Function*
Limity: [-65.536, 65.536]
Optimum: 1.0
Dimenzia: 2



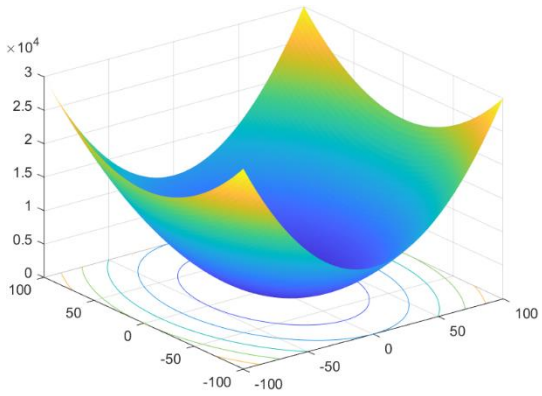
Funkcia F17:
Názov: *N. Kowalik's Function*
Limity: [-5, 5]
Optimum: 0.0003075
Dimenzia: 4



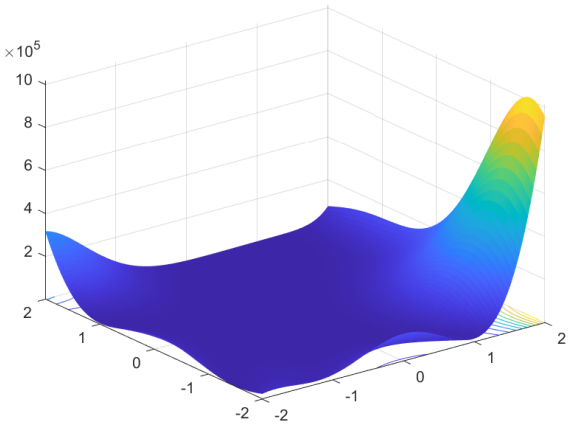
Funkcia F18:
Názov: *Six-Hump Camel*
Limity: [-5, 5]
Optimum: -1.0316285
Dimenzia: 2



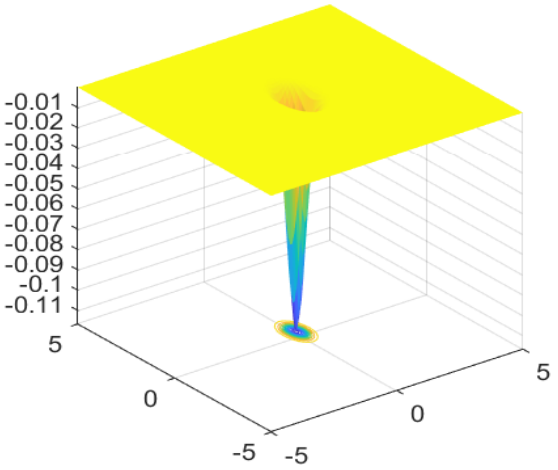
Funkcia F19:
Názov: *Bohachevsky Function f1*
Limity: [-100, 100]
Optimum: 0.0
Dimenzia: 2



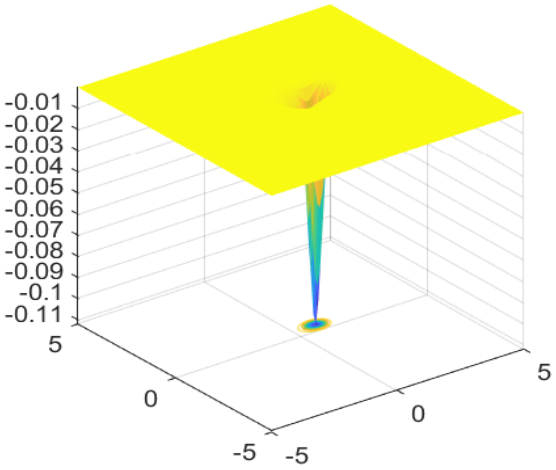
Funkcia F20:
Názov: *Goldstein-Price Function*
Limity: [-2, 2]
Optimum: 3.0
Dimenzia: 2



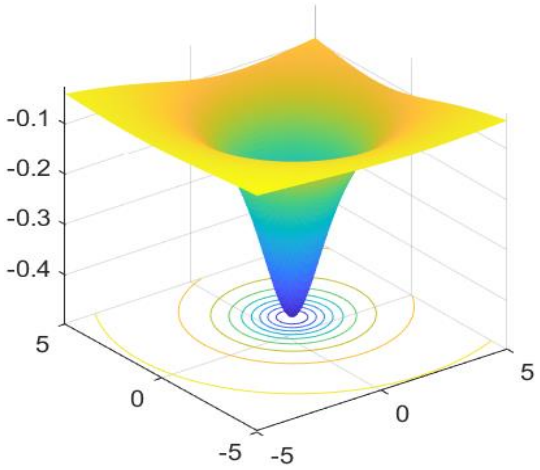
Funkcia F21:
Názov: *Hartmann Family 1 (3D)*
Limity: [-0, 1]
Optimum: -3.86278
Dimenzia: 3



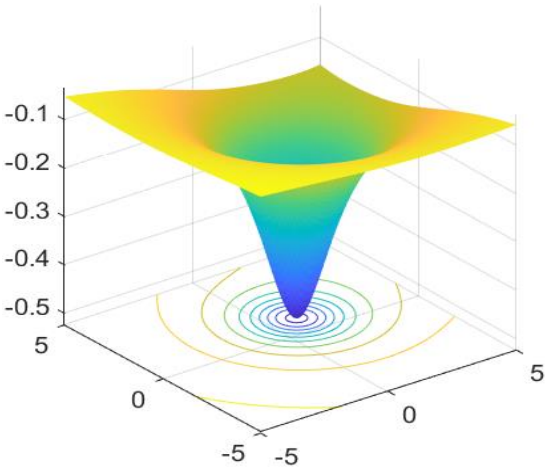
Funkcia F22:
Názov: *Hartman Family 2 (6D)*
Limity: [0, 1]
Optimum: -3.32237
Dimenzia: 6



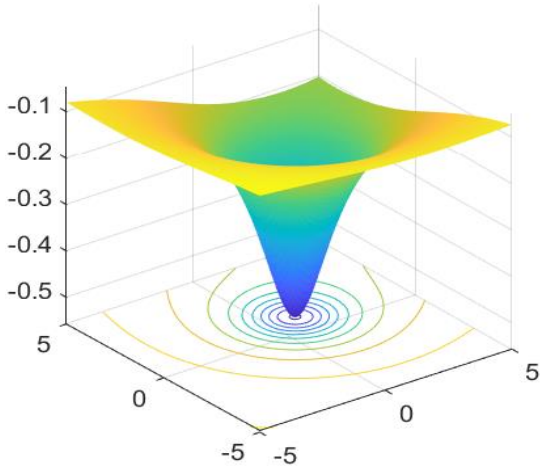
Funkcia F23:
Názov: *Shekel Function 1(m=5)*
Limity: [0, 10]
Optimum: -10.1532
Dimenzia: 4



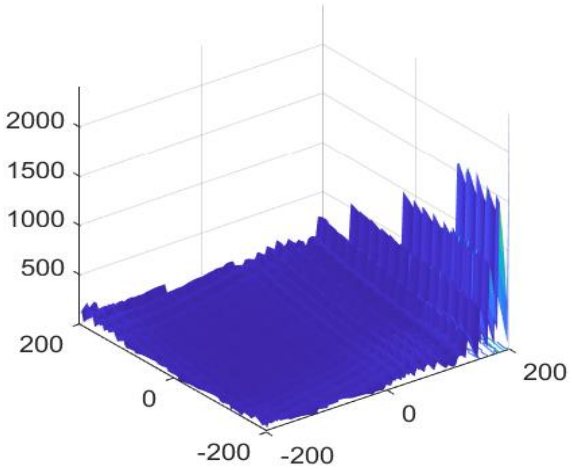
Funkcia F24:
Názov: *Shekel Function 2(m=7)*
Limity: [-0, 10]
Optimum: -10.5364
Dimenzia: 4



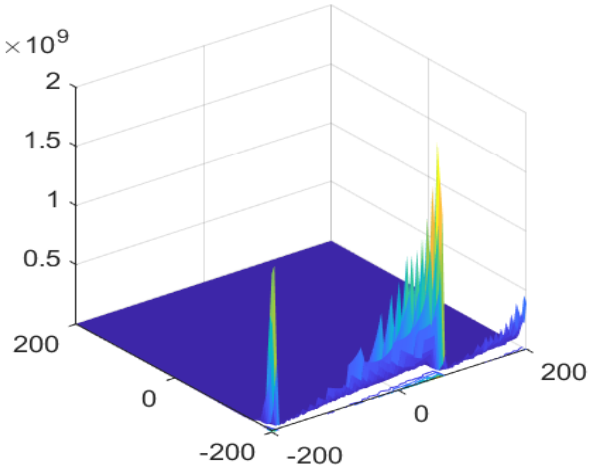
Funkcia F25:
Názov: *Shekel Function 3(m=10)*
Limity: [0, 10]
Optimum: -10.5364
Dimenzia: 4



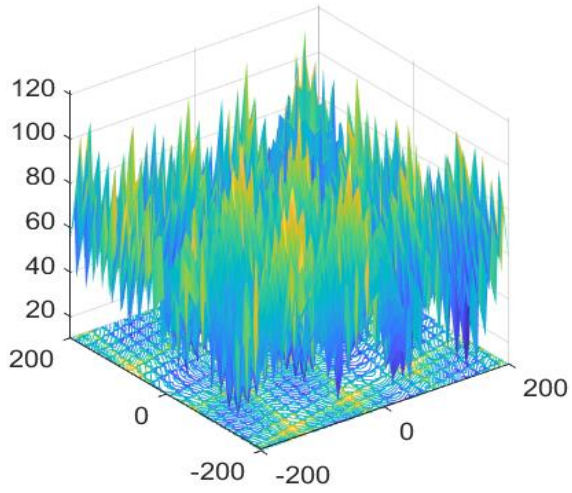
Funkcia F26:
Názov: *Zigzag F1*
Limity: [-200, 200]
Optimum: 0.0



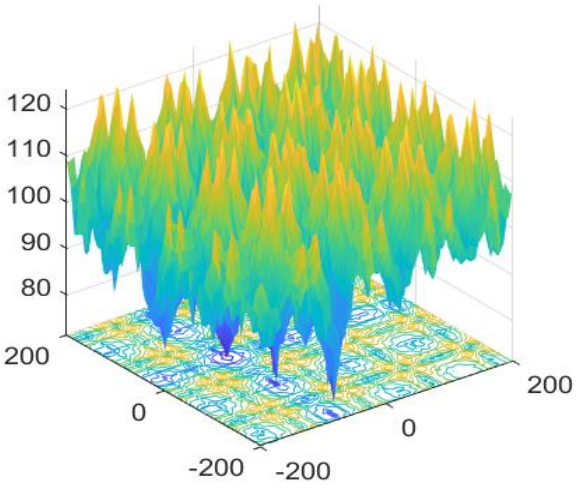
Funkcia F27:
Názov: *Zigzag F2*
Limity: [-200, 200]
Optimum: 0.0



Funkcia F28:
Názov: *Zigzag F3*
Limity: [-200, 200]
Optimum: 0.0



Funkcia F29:
Názov: *Zigzag F4*
Limity: [-200, 200]
Optimum: 0.0



Príloha 3

Výsledky benchmarkingu skupiny nDIM (kapitola 7.1)

Tab. 8 Glicko-2 hodnotenie skupiny nDIM

	ID	Rating	Deviation	Volatility	Games	Win	Draw	Loss	Lag
1	WOA	1831	9.08	0.0296	18750	14213	2133	2404	0
2	GWO	1817	9.12	0.0303	18750	14079	2121	2550	0
3	PSO	1803	8.99	0.03	18750	13810	2198	2742	0
4	ABC	1651	8.04	0.0261	18750	11247	1656	5847	0
5	DAO	1571	8.31	0.0283	18750	9764	1441	7545	0
6	ALO	1554	8.23	0.0277	18750	9375	1442	7933	0
7	EHO	1540	8.33	0.0285	18750	9206	1350	8194	0
8	MFO	1374	8.82	0.0296	18750	6000	1391	11359	0
9	CS	1211	9.54	0.0306	18750	3561	1253	13936	0
10	FA	1087	10.94	0.0358	18750	1960	1278	15512	0
11	BA	1024	12.98	0.0482	18750	1152	1253	16345	0

Výsledky benchmarkingu skupiny fixDIM (kapitola 7.2)

Tab. 9 Glicko-2 hodnotenie skupiny fixDIM

	ID	Rating	Deviation	Volatility	Games	Win	Draw	Loss	Lag
1	ABC	1800	20	0.049	2500	1830	588	82	0
2	ALO	1663	18.2	0.0501	2500	1550	446	504	0
3	WOA	1648	17.7	0.0475	2500	1477	510	513	0
4	MFO	1576	17.5	0.0508	2500	1254	508	738	0
5	GWO	1565	17.2	0.0479	2500	1307	308	885	0
6	FA	1465	16.9	0.0475	2500	1171	40	1289	0
7	PSO	1400	16.8	0.0457	2500	681	495	1324	0
8	DAO	1390	17.9	0.0524	2500	778	208	1514	0
9	CS	1344	17.7	0.0498	2500	692	122	1686	0
10	EHO	1331	18.1	0.0514	2500	656	137	1707	0
11	BA	1310	18.2	0.0499	2500	669	8	1823	0

Výsledky benchmarkingu skupiny zigzag (kapitola 7.3)

Tab. 9 Glicko-2 hodnotenie skupiny zigzag

	ID	Rating	Deviation	Volatility	Games	Win	Draw	Loss	Lag
1	WOA	1946	16.7	0.0473	5000	4213	0	787	0
2	ALO	1943	18	0.0587	5000	4235	1	764	0
3	ABC	1815	16.3	0.0494	5000	3726	0	1274	0
4	PSO	1812	16.2	0.0491	5000	3670	0	1330	0
5	GWO	1709	17.1	0.0557	5000	3204	0	1796	0
6	MFO	1668	16	0.047	5000	2997	1	2002	0
7	DAO	1391	17.4	0.048	5000	1983	0	3017	0
8	CS	1240	18.6	0.0501	5000	1455	0	3545	0
9	EHO	1236	18.7	0.0505	5000	1470	0	3530	0
10	FA	696	30.4	0.0916	5000	293	0	4707	0
11	BA	656	31.7	0.099	5000	253	0	4747	0

Výsledky benchmarkingu zlučených skupín pre DIM<6 (kapitola 7.4)

Tab.10 Glicko-2 hodnotenie zlúčených skupín DIM<6

	ID	Rating	Deviation	Volatility	Games	Win	Draw	Loss	Lag
1	ABC	1743	10.11	0.0324	12000	8532	1522	1946	0
2	WOA	1730	10.09	0.0327	12000	8374	1491	2135	0
3	GWO	1684	10.19	0.0355	12000	7840	1305	2855	0
4	PSO	1682	9.78	0.0323	12000	7680	1533	2787	0
5	ALO	1569	9.34	0.0312	12000	6401	1121	4478	0
6	MFO	1516	9.17	0.0297	12000	5538	1177	5285	0
7	DAO	1463	9.65	0.0333	12000	4930	857	6213	0
8	EHO	1445	9.49	0.0316	12000	4704	676	6620	0
9	CS	1289	10.53	0.0348	12000	2702	627	8671	0
10	FA	1256	10.75	0.0348	12000	2369	557	9074	0
11	BA	1132	12.25	0.0384	12000	1242	510	10248	0

Výsledky benchmarkingu zlúčených skupín pre DIM10 (kapitola 7.5)

Tab.11 Glicko-2 hodnotenie zlúčených skupín pre DIM10

	ID	Rating	Deviation	Volatility	Games	Win	Draw	Loss	Lag
1	WOA	1876	16.3	0.0449	4750	3798	391	561	0
2	GWO	1804	15.5	0.0438	4750	3520	403	827	0
3	PSO	1799	15.5	0.0442	4750	3497	394	859	0
4	ALO	1673	14.8	0.0426	4750	2970	273	1507	0
5	ABC	1655	14.5	0.0408	4750	2905	269	1576	0
6	DAO	1554	14.7	0.0422	4750	2394	269	2087	0
7	EHO	1490	14.9	0.0423	4750	2097	277	2376	0
8	MFO	1379	15.6	0.0436	4750	1643	250	2857	0
9	CS	1232	16.8	0.0445	4750	1018	250	3482	0
10	FA	1006	19.9	0.051	4750	382	250	4118	0
11	BA	967	21.3	0.0549	4750	263	250	4237	0

Výsledky benchmarkingu zlúčených skupín pre DIM15 (kapitola7.6)

Tab. 12 Glicko-2 hodnotenie zlúčených skupín pre DIM15

	ID	Rating	Deviation	Volatility	Games	Win	Draw	Loss	Lag
1	WOA	1911	16.9	0.0449	4750	3889	388	473	0
2	GWO	1819	16	0.0471	4750	3580	383	787	0
3	PSO	1797	15.3	0.0427	4750	3441	383	926	0
4	ALO	1667	14.7	0.0427	4750	2924	252	1574	0
5	ABC	1627	14.7	0.0421	4750	2751	250	1749	0
6	DAO	1588	14.8	0.0429	4750	2564	266	1920	0
7	EHO	1510	14.8	0.0419	4750	2209	270	2271	0
8	MFO	1354	15.8	0.0437	4750	1545	251	2954	0
9	CS	1215	17	0.0445	4750	1006	250	3494	0
10	FA	988	20	0.0504	4750	352	250	4148	0
11	BA	953	20.8	0.0518	4750	267	251	4232	0

Výsledky benchmarkingu zlúčených skupín pre DIM20 (kapitola 7.7)

Tab.12 Glicko-2 hodnotenie zlúčených skupín pre DIM20

	ID	Rating	Deviation	Volatility	Games	Win	Draw	Loss	Lag
1	WOA	1907	16.9	0.046	4750	3903	387	460	0
2	GWO	1836	15.9	0.0443	4750	3640	377	733	0
3	PSO	1787	15.9	0.0462	4750	3471	373	906	0
4	ALO	1667	14.7	0.042	4750	2924	250	1576	0
5	DAO	1607	14.7	0.0421	4750	2654	251	1845	0
6	ABC	1574	14.7	0.042	4750	2516	250	1984	0
7	EHO	1525	14.8	0.0421	4750	2238	266	2246	0
8	MFO	1367	15.9	0.045	4750	1573	253	2924	0
9	CS	1206	17.1	0.0447	4750	989	250	3511	0
10	BA	989	20.3	0.0516	4750	317	253	4180	0
11	FA	966	20.4	0.052	4750	320	250	4180	0

Výsledky benchmarkingu zlúčených skupín pre všetky dimenzie (kapitola 7.8)

Tab. 13 Glicko-2 hodnotenie zlúčených skupín pre všetky dimenzie

	ID	Rating	Deviation	Volatility	Games	Win	Draw	Loss	Lag
1	WOA	1801	7.6	0.026	26250	19950	2671	3629	0
2	GWO	1743	7.83	0.0308	26250	18611	2480	5159	0
3	PSO	1724	7.52	0.0281	26250	18050	2716	5484	0
4	ABC	1677	6.95	0.0242	26250	16778	2282	7190	0
5	ALO	1614	7.01	0.0254	26250	15095	1899	9256	0
6	DAO	1523	6.96	0.0252	26250	12509	1697	12044	0
7	EHO	1470	7.16	0.0265	26250	11291	1484	13475	0
8	MFO	1455	7.11	0.0256	26250	10288	1942	14020	0
9	CS	1269	7.95	0.0279	26250	5772	1372	19106	0
10	FA	1143	9.33	0.0339	26250	3372	1320	21558	0
11	BA	1074	9.9	0.034	26250	2096	1263	22891	0