

TDK dolgozat

Marcsák Gábor Zoltán



MISKOLCI EGYETEM
GÉPÉSZMÉRNÖKI ÉS INFORMATIKAI KAR
Anyagmozgatási és Logisztikai Tanszék



TUDOMÁNYOS DIÁKKÖRI DOLGOZAT

**Metaheurisztikus algoritmusok
hatékonyságvizsgálata saját keretrendszer által
létrehozott komplex tesztfüggvények segítségével, az
eredmények felhasználása futódaru főtartó
optimálásához**

Marcsák Gábor Zoltán
II. éves logisztikai mérnök
MSc szakos hallgató

Konzulens:
Prof. Dr. Jármay Károly
egyetemi tanár, rektorhelyettes
Anyagmozgatási és Logisztikai Tanszék

Miskolc, 2013

Köszönetnyilvánítás

A kutatás az Európai Unió és Magyarország támogatásával, az Európai Szociális Alap társfinanszírozásával a TÁMOP 4.2.4.A/2-11-1-2012-0001 azonosító számú „Nemzeti Kiválóság Program – Hazai hallgatói, illetve kutatói személyi támogatást biztosító rendszer kidolgozása és működtetése konvergencia program” című kiemelt projekt keretei között valósult meg”.

Tartalomjegyzék

1	BEVEZETÉS	1
2	A HEURISZTIKUS ALGORITMUSOK ELMÉLETI HÁTTERE.....	3
2.1	MIRE JÓK A HEURISZTIKUS ALGORITMUSOK?.....	3
2.2	MENNYIRE HATÉKONYAK A HEURISZTIKUS ALGORITMUSOK?	5
2.3	MI A KÜLÖNBség A HEURISZTIKA ÉS A METAHEURISZTIKA KÖZÖTT?	5
3	VIZSGÁLT HEURISZTIKUS ALGORITMUSOK.....	6
3.1	ARTIFICIAL IMMUNE NETWORK (AiNet)	6
3.1.1	<i>Pszéudokód</i>	<i>7</i>
3.1.2	<i>Bemenő paraméterek.....</i>	<i>7</i>
3.1.3	<i>C# forráskód</i>	<i>7</i>
3.2	BACTERIAL FORAGING (BFOA).....	8
3.2.1	<i>Pszéudokód</i>	<i>9</i>
3.2.2	<i>Bemenő paraméterek.....</i>	<i>9</i>
3.2.3	<i>C# forráskód</i>	<i>10</i>
3.3	BEES ALGORITHM (BA)	10
3.3.1	<i>Pszéudokód</i>	<i>11</i>
3.3.2	<i>Bemenő paraméterek.....</i>	<i>12</i>
3.3.3	<i>C# forráskód</i>	<i>12</i>
3.4	CULTURAL ALGORITHM (CA).....	12
3.4.1	<i>Pszéudokód</i>	<i>13</i>
3.4.2	<i>Bemenő paraméterek.....</i>	<i>14</i>
3.4.3	<i>C# forráskód</i>	<i>14</i>
3.5	DIFFERENTIAL EVOLUTION (DE).....	14
3.5.1	<i>Pszéudokód</i>	<i>15</i>
3.5.2	<i>Bemenő paraméterek.....</i>	<i>16</i>
3.5.3	<i>C# forráskód</i>	<i>16</i>
3.6	HARMONY SEARCH (HS).....	16
3.6.1	<i>Pszéudokód</i>	<i>17</i>
3.6.2	<i>Bemenő paraméterek.....</i>	<i>18</i>
3.6.3	<i>C# forráskód</i>	<i>18</i>
3.7	KRILL HERD (KH)	18
3.7.1	<i>Pszéudokód</i>	<i>19</i>
3.7.2	<i>Bemenő paraméterek.....</i>	<i>19</i>
3.7.3	<i>C# forráskód</i>	<i>20</i>

3.8	MEMETIC ALGORITHM (MA)	20
3.8.1	<i>Pszudokód</i>	21
3.8.2	<i>Bemenő paraméterek</i>	21
3.8.3	<i>C# forráskód</i>	21
3.9	NELDER-MEAD ALGORITHM (NM)	22
3.9.1	<i>Pszudokód</i>	23
3.9.2	<i>Bemenő paraméterek</i>	23
3.9.3	<i>C# forráskód</i>	23
3.10	PARTICLE SWARM OPTIMIZATION (PSO)	23
3.10.1	<i>Pszudokód</i>	25
3.10.2	<i>Bemenő paraméterek</i>	25
3.10.3	<i>C# forráskód</i>	25
3.11	RANDOM SEARCH (RS)	26
3.11.1	<i>Pszudokód</i>	26
3.11.2	<i>Bemenő paraméterek</i>	26
3.11.3	<i>C# forráskód</i>	26
3.12	SIMULATED ANNEALING (SA)	27
3.12.1	<i>Pszudokód</i>	28
3.12.2	<i>Bemenő paraméterek</i>	28
3.12.3	<i>C# forráskód</i>	28
4	GLÓBÁLIS OPTIMUM KERESÉSE ISMERT TESZTFÜGGVÉNYEK ESETÉBEN	29
4.1	ACKLEY'S FUNCTION (F1)	34
4.1.1	<i>C# forráskód</i>	34
4.1.2	<i>Mérési eredmények</i>	35
4.1.2.1	Konvergencia grafikonok	35
4.1.2.2	Statisztikai adatok	36
4.2	DE JONG'S FUNCTION (F2)	37
4.2.1	<i>C# forráskód</i>	37
4.2.2	<i>Mérési eredmények</i>	38
4.2.2.1	Konvergencia grafikonok	38
4.2.2.2	Statisztikai adatok	39
4.3	DROP-WAVE FUNCTION (F3)	39
4.3.1	<i>C# forráskód</i>	40
4.3.2	<i>Mérési eredmények</i>	41
4.3.2.1	Konvergencia grafikonok	41
4.3.2.2	Statisztikai adatok	42
4.4	EASOM'S FUNCTION (F4)	42

4.4.1	C# forráskód	42
4.4.2	Mérési eredmények.....	43
4.4.2.1	Konvergencia grafikonok.....	43
4.4.2.2	Statisztikai adatok.....	44
4.5	GRIEWANGK'S FUNCTION (F5).....	45
4.5.1	C# forráskód	45
4.5.2	Mérési eredmények.....	46
4.5.2.1	Konvergencia grafikonok.....	46
4.5.2.2	Statisztikai adatok.....	47
4.6	MATYAS'S FUNCTION (F6).....	48
4.6.1	C# forráskód	48
4.6.2	Mérési eredmények.....	49
4.6.2.1	Konvergencia grafikonok.....	49
4.6.2.2	Statisztikai adatok.....	50
4.7	RASTRIGIN'S FUNCTION (F7)	50
4.7.1	C# forráskód	50
4.7.2	Mérési eredmények.....	51
4.7.2.1	Konvergencia grafikonok.....	51
4.7.2.2	Statisztikai adatok.....	52
4.8	ROSENBROCK'S VALLEY (F8).....	53
4.8.1	C# forráskód	53
4.8.2	Mérési eredmények.....	54
4.8.2.1	Konvergencia grafikonok.....	54
4.8.2.2	Statisztikai adatok.....	55
4.9	SCHAFFER'S N. 2. FUNCTION (F9)	55
4.9.1	C# forráskód	55
4.9.2	Mérési eredmények.....	56
4.9.2.1	Konvergencia grafikonok.....	56
4.9.2.2	Statisztikai adatok.....	57
4.10	THREE-HUMP CAMELBACK (F10)	58
4.10.1	C# forráskód.....	58
4.10.2	Mérési eredmények	59
4.10.2.1	Konvergencia grafikonok.....	59
4.10.2.2	Statisztikai adatok.....	60
4.11	AZ EREDMÉNYEK ÉRTÉKELÉSE	61
5	KOMPLEX TESZTFÜGGVÉNYEK LÉTREHOZÁSA SAJÁT KERETRENDSZER SEGÍTSÉGÉVEL	63
5.1	KOMPLEX TESZTFÜGGVÉNYEK LÉTREHOZÁSÁNAK MÓDSZERE	63
5.1.1	Súlyozó függvények.....	64

5.1.1.1	Euklideszi távolság.....	64
5.1.1.2	Gábor-függvény.....	65
5.1.1.3	Gauss-függvény.....	65
5.1.2	Gyakorlati példa komplex tesztfüggvény létrehozására.....	65
5.2	SZOFTVER KOMPLEX TESZTFÜGGVÉNYEK LÉTREHOZÁSÁHOZ	67
5.2.1	Komplex függvény C# kódjának automatikus létrehozása	68
5.2.2	Továbbfejlesztési lehetőségek	69
6	GLOBÁLIS OPTIMUM KERESÉSE SAJÁT KOMPLEX TESZTFÜGGVÉNY ESETÉBEN.....	70
6.1	KOMPLEX N. 1. FESZTFÜGGVÉNY.....	71
6.1.1	C# forráskódok.....	71
6.1.2	Komplex N. 1. euklideszi távolságon alapuló súlyozás esetén (F11)	72
6.1.2.1	Konvergencia grafikonok.....	72
6.1.2.2	Statisztikai adatok.....	73
6.1.3	Komplex N. 1. Gábor-féle súlyozás ($\tau_1 = 3, \tau_2 = 8$) esetén (F12).....	74
6.1.3.1	Konvergencia grafikonok.....	74
6.1.3.2	Statisztikai adatok.....	75
6.1.4	Komplex N. 1. Gauss-féle súlyozás esetén (F13)	76
6.1.4.1	Konvergencia grafikonok.....	76
6.1.4.2	Statisztikai adatok.....	77
6.2	AZ EREDMÉNYEK ÉRTÉKELÉSE.....	78
7	FUTÓDARU FŐTARTÓ OPTIMÁLÁSA.....	79
7.1	AZ OPTIMÁLÁSI FELADAT MATEMATIKAI LEÍRÁSA []	81
7.1.1	Döntési változók meghatározása.....	81
7.1.2	Feltételek meghatározása	81
7.1.2.1	Méretkorlátok	81
7.1.2.2	Fáradás	82
7.1.2.3	Statikus terhelés.....	82
7.1.2.4	Gerinc horpadás.....	83
7.1.2.5	Öv horpadás.....	83
7.1.2.6	Lehajlás.....	83
7.1.3	Célfüggvény.....	83
7.2	AZ OPTIMÁLÁSI FELADAT MEGOLDÁSA	84
7.2.1	C# forráskód	84
8	ÖSSZEFOGLALÁS	87
9	1. SZÁMÚ MELLÉKLET, AZ ALGORITMUSOK C# FORRÁSKÓDJA.....	88
9.1	ARTIFICIAL IMMUNE NETWORK (AiNet) C# FORRÁSKÓDJA	88

9.2	BACTERIAL FORAGING (BFOA) C# FORRÁSKÓDJA.....	93
9.3	BEES ALGORITHM (BA) C# FORRÁSKÓDJA	97
9.4	CULTURAL ALGORITHM (CA) C# FORRÁSKÓDJA	100
9.5	DIFFERENTIAL EVOLUTION (DE) C# FORRÁSKÓDJA.....	104
9.6	HARMONY SEARCH (HS) C# FORRÁSKÓDJA.....	107
9.7	KRILL HERD (KH) C# FORRÁSKÓDJA	110
9.8	MEMETIC ALGORITHM (MA) C# FORRÁSKÓDJA.....	116
9.9	NELDER-MEAD ALGORITHM (NM) C# FORRÁSKÓDJA	121
9.10	PARTICLE SWARM OPTIMIZATION (PSO) C# FORRÁSKÓDJA	125
9.11	RANDOM SEARCH (RS) C# FORRÁSKÓDJA	128
9.12	SIMULATED ANNEALING (SA) C# FORRÁSKÓDJA	130
10	2. SZÁMÚ MELLÉKLET, FUTÓDARU FŐTARTÓ OPTIMÁLÁS C# FORRÁSKÓDJA.....	133
10.1	BÜNTETŐFÜGGVÉNY	133
10.2	CÉLFÜGGVÉNY	135
11	IRODALOMJEGYZÉK.....	136

1 Bevezetés

Az optimalizálás mindennapi életünk egyik meghatározó tevékenysége. A legtöbb ember folyamatosan azon dolgozik, hogy az adott feltételek mellett a maga, vagy mások számára valamilyen szempontból legkedvezőbb kimenetelt válassza ki egy probléma megoldása során. E mindenki által végzett tudatos vagy tudat alatti tevékenység az optimalizálás. Számos kiváló algoritmust dolgoztak ki a különböző optimalizálási problémák megoldására. A számítógépek rohamos fejlődése ellenére még mindig sok olyan feladat ismert, mely nem oldható meg pusztán a számítási teljesítményre alapozva. A megoldást az informált kereső, úgynevezett heurisztikus algoritmusok jelentik. A különböző heurisztikus algoritmusok korábban megoldhatatlannak vélt problémák megoldásával bizonyították létjogosultságukat [1]. Az élet számos területén találkozhatunk optimalizálási problémákkal, legyen szó mérnöki, informatikai, orvosi, vagy bármilyen egyéb tudományterületről. A [2, 3, 4] forrásművekben a mérnöki tudományterületen felmerülő optimalizálási problémákra találhatunk példákat, melyek a nagyszámú változó, feltétel, és bonyolult célfüggvények miatt csak heurisztikus módszerekkel oldhatók meg.

A problémák megoldásához szükséges azok matematikai megfogalmazása. Az optimalizálás során a különböző értékek két csoportba sorolhatók: előre megadott (bemenő) paraméterek, illetve döntési változók. Az alapvető különbséget az jelenti közöttük, hogy a bemenő paraméterek értéke rögzített (konstans), ezzel szemben a döntési változók értéke az optimalizálás során változik. Annak függvényében, hogy a döntési változók milyen értékeket vehetnek fel, léteznek diszkrét és folytonos változók. A döntési változók értékét különböző feltételek definiálásával befolyásolhatjuk. Ha definiálunk feltételt, akkor feltételes, egyébként feltétel nélküli optimalizálást végzünk. A feltételek matematikailag lehetnek egyenlőségek vagy egyenlőtlenségek. Az optimalizálás célját, a megoldási alternatívák vizsgálatát célfüggvény segítségével határozzuk meg. Abban az esetben, ha csupán egyetlen célfüggvény van, egycélfüggvényes, ellenkező esetben többcélfüggvényes optimalizálásról beszélünk. Egycélfüggvényes optimalizálás lehet például egy rácsos tartó súlyminimumának meghatározása, mely egyetlen végeredményt, legtöbbször egy skálár értéket ad [5]. A többcélfüggvényes eset bonyolultabb, mivel az egyes, általában egymással konfliktusban lévő célfüggvények minimumának és maximumának egyidejű meghatározása szükséges. Vizsgáljunk például egy egyszerű kéttámaszú tartót, ahol az egyik minimálandó célfüggvény a tartó súlya, a másik célfüggvény pedig a maximális

merevség. Nyilvánvalóan a két célfüggvény egymással konfliktusban van. A többcélfüggvényes optimalási feladat megoldását Pareto fogalmazta meg [6], ezért szokás Pareto optimumnak nevezni. A definíció szerint akkor beszélünk optimumról, ha egyik célfüggvény értéke sem javítható úgy, hogy legalább egy másik célfüggvény értéke ne romlana. Az optimum tehát nem olyan egyértelmű, mint egycélfüggvényes optimalás esetén, mert alternatív megoldások egész halmazát (Pareto halmaz) jelenti. A végső megoldás csak további kritériumok, feltételek segítségével határozható meg.

Dolgozatomban metaheurisztikus algoritmusokkal foglalkozom, melyek segítségével akár nagyon bonyolult, sok bemenő paraméterrel, döntési változóval és feltétellel leírható, többcélfüggvényes optimalási feladatok is eredményesen megoldhatók. A metaheurisztikus algoritmusok működése általában valamilyen természeti jelenségen alapul. Vizsgálom többek között evolúciós (Differential Evolution, Cultural Algorithm, Memetic Algorithm), fizikai (Simulated Annealing, Harmony Search), biológiai (Artificial Immune Network) és rajintelligencia (Bacterial Foraging, Bees Algorithm, Krill Herd, Particle Swarm) ihletésű eljárásokat.

Az algoritmusok hatékonyságát elsőként a szakirodalomban tárgyalt tesztfüggvényekkel vizsgálom. Ismertetek egy új módszert, aminek segítségével komplex tesztfüggvények hozhatók létre a már ismert alapfüggvényekből. A módszer gyakorlatilag határtalan lehetőségeket nyújt újfajta, tetszőleges bonyolultságú egyéni tesztfüggvények konstruálására. A hatékonyságvizsgálat során meghatározom a különböző algoritmusok ideális bemenő paramétereit, majd elvégzem egy futódaru főtartó szerkezetoptimalását.

2 A heurisztikus algoritmusok elméleti háttere

A hagyományos algoritmusok előre definiált instrukciók alapján, lépésről lépésre hajtanak végre egy adott feladatot, eredményük pedig egzakt, determinisztikus. Az egyik legegyszerűbb algoritmus például a Pitagorasz-tétel, mely két bemenő paraméter alapján egzakt módon meghatároz egy harmadikat. A hagyományos kereső algoritmusokkal szemben a heurisztikus algoritmusok próbálgatással, a korábban megszerzett tapasztalatok felhasználásával jutnak eredményre. Szokás ezért informált kereső eljárásoknak is nevezni őket. A heurisztika kifejezés a görög heureszisz szóból származik, melynek jelentése rátalálás. Az egyik legegyszerűbb példa a sakk játék. Elméletileg konstruálható lenne olyan egzakt megoldást biztosító eljárás, mely a sakkbábuk aktuális helyzete alapján, az összes további lehetséges lépés elemzésével kiszámítaná, hogyan nyerhetünk. A problémát az jelenti, hogy adott lépésszám fölött a probléma túl bonyolult lesz, még a legmodernebb számítástechnikai eszközökkel sem oldható meg belátható időn belül. A heurisztikus algoritmus azonban nem vizsgálja az összes lehetséges lépést, csupán a problémátér egy adott részlete alapján, valamilyen logika szerint hozza meg döntését.

2.1 Mire jók a heurisztikus algoritmusok?

A heurisztikus algoritmusok hatalmas előnye, hogy nagy bonyolultságú problémák esetében is képesek viszonylag rövid idő alatt, kevés számítás árán eredményt szolgáltatni. Hátrányuk azonban, hogy nem garantálható teljes bizonyossággal az optimális megoldás megtalálása. Használatukkal tehát sebességet kapunk, azonban cserébe pontossággal fizetünk. Lokális kereső eljárások, mivel nem vizsgálják a teljes problémateret, minden lehetséges kimenetelt (sok esetben ez egyébként is fizikai képtelenség), és a korábban bejárt utat sem tárolják. Akkor érdemes heurisztikus algoritmust használni, ha az adott probléma megoldása hagyományos, egzakt megoldást adó eljárással belátható időn belül nem hajtható végre.

Jó példa az Utazó ügynök probléma (TSP- Traveling Salesman Problem), ami egy gráfelméleti feladat, elnevezését azonban onnan kapta, hogy egy mindennapi életből vett példával szokták szemléltetni. Adott bizonyos számú város, továbbá ismerjük a távolságot minden város között. Egy ügynök körutat szeretne tenni a városokban úgy, hogy mindegyiket pontosan egyszer érinti, és az útiköltség a lehető legkisebb legyen. Az útiköltség egyenesen arányos a városok közötti távolsággal. A feladat tehát olyan útvonal

meghatározása, melynek költsége minimális. A probléma gráfelméleti megfogalmazása szerint adott egy teljes súlyozott gráf, és e gráfon keressük a legkisebb összsúlyú Hamilton kört. A gráf csúcspontjai a városok, élei pedig a városokat összekötő utak. Az élek súlyozása az útiköltségnek felel meg. A gráf teljes, mivel feltételezzük, hogy tetszőleges városból közvetlenül eljuthatunk bármelyik másikba.

A Utazó ügynök pontos eredetéről megoszlanak a vélemények, a témában már a XIX. században is készültek publikációk [7]. Az általános változat matematikai megfogalmazása 1930-ban készült el. Azért bír különösen nagy jelentőséggel, mert számos gyakorlati alkalmazás vezethető vissza rá. A logisztikában a gyűjtő és elosztó járatok útvonalának megtervezése (futárszolgálat, kommunális hulladékszállítás) tipikusan ilyen alkalmazás. Az elektronikai gyártásban is gyakran előfordul, hogy egy nyákon bizonyos pontokat szeretnénk összekötni úgy, hogy a kötés hossza a lehető legrövidebb legyen. Az élet számos területén felmerül a TSP, ezért az egyik legtöbbet kutatott kombinatorikus optimalizálási probléma. Az Utazó ügynök problémát úgy lehetne teljes bizonyossággal megoldani, ha az összes lehetséges körútnak kiszámolnánk a költségét. Az összes csúcspont lehetséges permutációinak száma $n!$. Minden permutáció meghatároz egy Hamilton kört a teljes gráfban. Az 1. táblázatban a TSP számítás mérföldkövei szerepelnek, könnyedén kiszámítható, hogy adott számú csúcspont esetében mennyi lehetséges megoldás van:

Csúcspontok száma	Lehetséges megoldások száma	Megoldás éve
5	120	-
10	3628800	-
25	$1,55112101 \cdot 10^{25}$	-
49 város (USA 48 tagállam fővárosa és Washington)	$6,08281864 \cdot 10^{62}$	1954
120 város (Nyugat-Németország)	$6,68950291 \cdot 10^{198}$	1977
532 telefonközpont (USA AT&T)	?	1987
666 nevezetesség a világon	?	1987
4461 város	?	1988
13509 város (USA)	?	1998
24978 város (Svédország)	?	2004
85900 pont (Integrált áramkör)	?	2006

1. táblázat: Az Utazó ügynök probléma mérföldkövei

A táblázatból leolvasható, hogy viszonylag kis számú csúcspont esetén is rengeteg lehetséges megoldás létezik. Az Utazó ügynök probléma n méretű bemenet esetén $O(n!)$ bonyolultságú, ezért nem oldható meg polinomiális időben. A TSP az NP-nehez problémák osztályába tartozik, hatékony megoldás nem ismert nagyszámú csúcspont esetében. Érdekes, hogy a híres fizikus, Stephen Hawking szerint a megfigyelhető univerzum nagyságrendileg 10^{78} - 10^{82} atomból áll [8]. Nagyságrendileg ugyanennyi lehetséges megoldása van az Utazó ügynök problémának 60 város esetén. Nyilvánvaló, hogy bizonyos számú város felett az összes lehetséges megoldás vizsgálata fizikai képtelenség. A megoldást a heurisztikus algoritmusok jelentik, mivel nagy problémaméret esetén is képesek optimális, vagy ahhoz közelítő megoldást adni.

2.2 Mennyire hatékonyak a heurisztikus algoritmusok?

A heurisztikus algoritmusok hatékonyságáról megoszlanak a vélemények. Korábban megoldhatatlannak vélt problémákra képesek optimális, vagy ahhoz közeli megoldást adni. Egyesek szerint azonban attól, hogy egy bizonyos problémára elfogadható megoldást találnak, semmi sem garantálja, hogy egy hasonló feladatra is jobb megoldást adnak, mint akár egy véletlen kereső algoritmus. Ezen aggályokat írja le a No-Free-Lunch (Nincs ingyen ebéd) teória, melyről bővebben a [9] forrásműben olvashat az érdeklődő.

2.3 Mi a különbség a heurisztika és a metaheurisztika között?

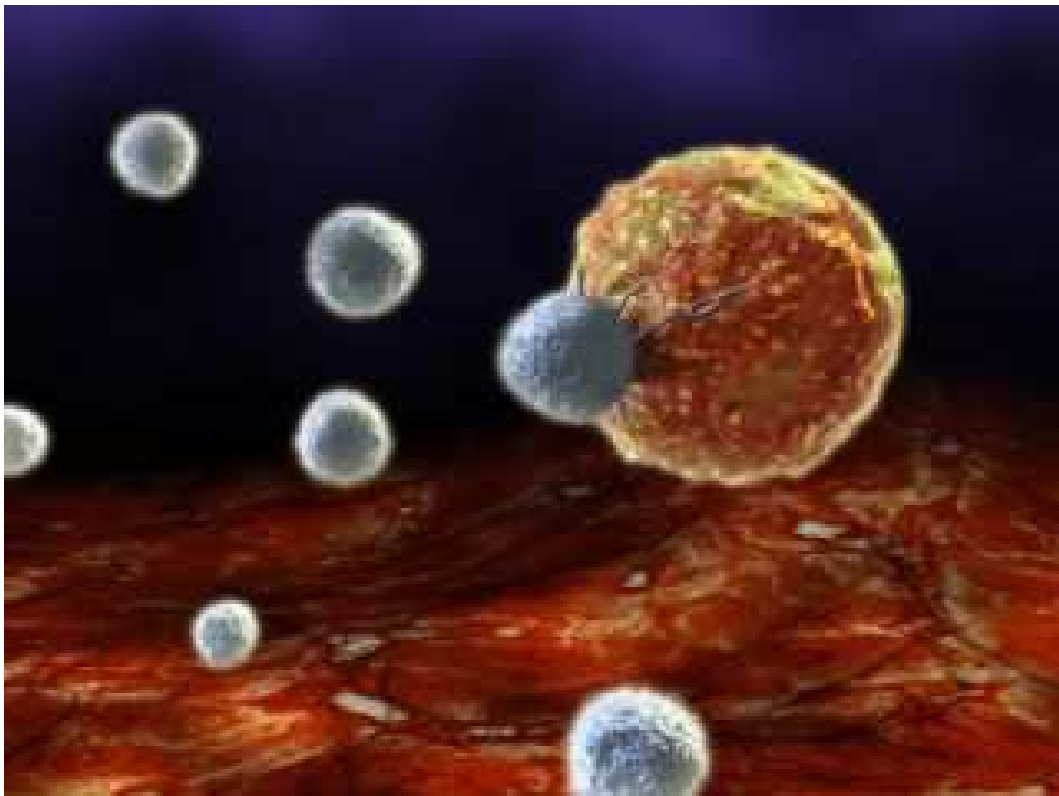
A metaheurisztikus eljárások olyan univerzális, robosztus heurisztikus kereső algoritmusok, melyek nem csak egy adott típusú feladathoz használhatóak eredményesen. Bizonyos korábban szerzett ismereteket eltárolnak, rendelkeznek tehát probléma specifikus ismeretekkel a megoldás során, ezáltal az esetek nagy részében képesek kikerülni a lokális optimum pontokból. A metaheurisztikus algoritmusok egyik jellemző tulajdonsága, hogy működésüket gyakran sztochasztikus jellemzők is befolyásolják. Hatékonyságvizsgálatuk ezért bonyolult feladat, mivel kis túlzással nincs két egyforma futás.

3 Vizsgált heurisztikus algoritmusok

Dolgozatomban összesen egy tucat modern algoritmust vizsgáltam. Az algoritmusok elméleti háttere, működési elve mellett ismertetem azok pszeudokódját, valamint implementációjukat C# programnyelven.

3.1 Artificial Immune Network (AiNet)

Az Artificial Immune Network algoritmust de Castro és Von Zuben dolgozta ki egy klaszterezési feladat megoldására 2000-ben [10]. Működési elve szerint az Artificial Immune System (AIS), azaz Mesterséges immunrendszer algoritmusok osztályába tartozik, mely az élőlények immunrendszerének struktúráját, működését másolja. Az immunrendszer sejtjei a testben folyamatosan a potenciálisan veszélyes, testidegen sejtek után kutatnak. Ennek megfelelően egy biológiai-ihletésű, evolúciós eljárásról van szó. Metaheurisztikus algoritmus, melyet különböző optimalási problémákhoz, klaszterezéshez, és gépi látáshoz használtak eredményesen.



1. ábra: Az immunrendszer fehérvérsejtjei

3.1.1 Pszeudokód [11]

```
Input: PopulationSize, ProblemSize, Nclones, Nrandom, AffinityThreshold
Output: Sbest
1. Population ← InitializePopulation(PopulationSize, 1 ProblemSize);
2. while ¬StopCondition() do
3.     EvaluatePopulation(Population);
4.     Sbest ← GetBestSolution(Population);
5.     Progeny ← ∅;
6.     Costavg ← CalculateAveragePopulationCost(Population);
7.     while CalculateAveragePopulationCost(Population) > Costavg do
8.         foreach Celli ∈ Population do
9.             Clones ← CreateClones(Celli, Nclones);
10.            foreach Clonei ∈ Clones do
11.                Clonei ←
12.                    MutateRelativeToFitnessOfParent(Clonei, Celli);
13.            end
14.            EvaluatePopulation(Clones);
15.            Progeny ← GetBestSolution(Clones);
16.        end
17.    end
18.    SuppressLowAffinityCells(Progeny, AffinityThreshold);
19.    Progeny ← CreateRandomCells(Nrandom);
20.    Population ← Progeny;
21. end
22. return Sbest;
```

3.1.2 Bemenő paraméterek

- **PopulationSize:** Az immunsejtek kezdeti populációjának nagysága. A futás során a populációszám dinamikusan változik. Döntési változónként 10-50 ajánlott.
- **ProblemSize:** A döntési változók száma.
- **Nclones:** Minden egyedből adott számú klónt hoz létre az algoritmus, amiket egy mutációs operátor is befolyásol. A következő generációnak a legrátermettebb klón lesz tagja. Az adott egyedhez létrehozott klónok számát érdemes 10 alatt tartani!
- **Nrandom:** Minden iteráció során adott számú egyedet hoz létre az algoritmus, véletlenszerű pozícióban a problématerén belül, ezzel elkerülve az esetleges lokális optimum pontokban ragadást. Ajánlott 2-10 között tartani az értékét.
- **AffinityThreshold:** Közelségi küszöbérték, a szakirodalom 0,1-es értéket ajánl.

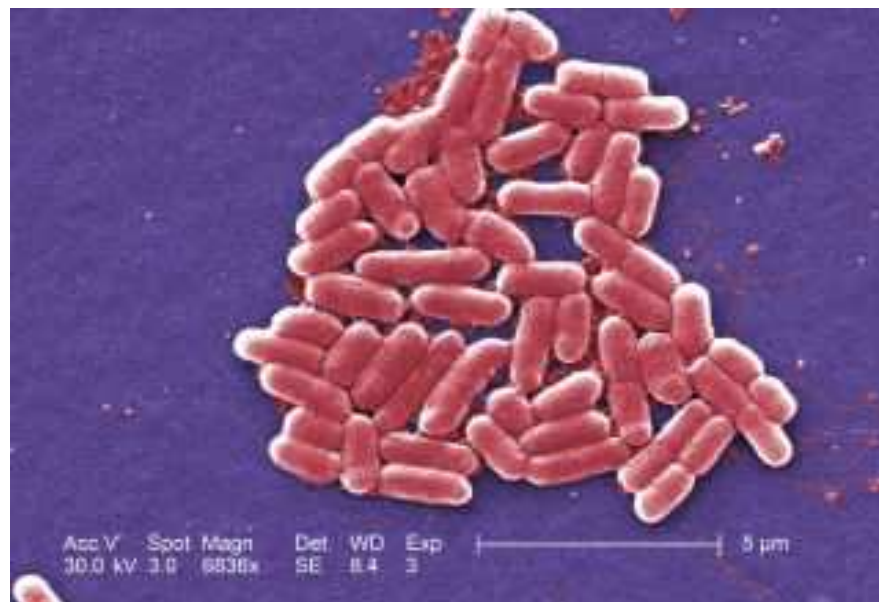
3.1.3 C# forráskód

A C# forráskódban az algoritmus a Rastrigin's tesztfüggvény globális minimum pontját keresi. A kód terjedelme miatt az 1. számú mellékletben található, illetve letölthető az alábbi linkről:

<https://drive.google.com/file/d/0BxE6yHbGFZBAazFBLWc4cFZGd00/edit?usp=sharing>

3.2 Bacterial Foraging (BFOA)

A Bacterial Foraging (BFOA) algoritmust először Liu és Passino írta le 2002-ben [12]. Egy Swarm Intelligence (Rajintelligencia) elven működő eljárás. A rajintelligencia (kollektív intelligencia) módszerek közös tulajdonsága, hogy nagyszámú homogén egyed viselkedésmintáin alapulnak. Az alapelv szerint lehetséges hogy egy individuális egyed nem képes megoldani adott feladatot, azonban ha nagyszámú egyed csoportot alkot, akkor a csoport kollektív intelligenciája már elég lehet a feladat sikeres megoldásához [13]. Viszonylag újnak számít a természeti jelenségeken alapuló rajintelligencia stratégiák családjában. A 2. ábra látható *E. coli* baktériumkolóniák táplálékkereső és reprodukciós viselkedésmintáin alapul a működése. Az *Escherichia Coli* az ostoros baktériumok családjába tartozik. Ostorszerű végtagjai, az úgynevezett flagellumok segítségével képes az önálló úszásra. Ez a mozgás a kemotaxis, mely során a fő cél a tápanyagban gazdag helyek elérése, továbbá a valamilyen okból veszélyesnek ítélt helyek elkerülése. A Bacterial Foraging (Baktérium táplálkozás) lényegében az *E. Coli* baktériumkolónia egyedei által végzett kemotaxist másolja a keresés során. Az egyedek hol tudatosan, hol véletlenszerűen mozognak az egyre jobb megoldást adó pontok felé.



2. ábra: *E. Coli* baktérium kolónia

3.2.1 Pszeudokód [14]

```
Input: Problemsize, Cellsnum, Ned, Nre, Nc, Ns, Stepsize, Ped
Output: Cellbest
1. Population ← InitializePopulation(Cellsnum, 1 Problemsize);
2. for l = 0 to Ned do
3.     for k = 0 to Nre do
4.         for j = 0 to Nc do
5.             ChemotaxisAndSwim(Population, Problemsize, Cellsnum, Ns, Stepsize);
6.             foreach Cell ∈ Population do
7.                 if Cost(Cell) ≤ Cost(Cellbest) then
8.                     Cellbest ← Cell;
9.                 end
10.            end
11.        end
12.        SortByCellHealth(Population);
13.        Selected ← SelectByCellHealth(Population, Cellsnum/2);
14.        Population ← Selected;
15.        Population ← Selected;
16.    end
17.    foreach Cell ∈ Population do
18.        if Rand() ≤ Ped then
19.            Cell ← CreateCellAtRandomLocation();
20.        end
21.    end
22. end
23. return Cellbest;

25. ChemotaxisAndSwim(Population, Problemsize, Cellsnum, Ns, Stepsize)
26.     foreach Cell ∈ Population do
27.         Cellfitness ← Cost(Cell) + Interaction(Cell, Population);
28.         Cellhealth ← Cellfitness;
29.         Cell' ← ∅;
30.         for i = 0 to Ns do
31.             RandomStepDirection ← CreateStep(Problemsize);
32.             Cell' ← TakeStep(RandomStepDirection, Stepsize);
33.             Cell'fitness ← Cost(Cell') + Interaction(Cell', Population);
34.             if Cell'fitness > Cellfitness then
35.                 i ← Ns;
36.             else
37.                 Cell ← Cell';
38.                 Cellhealth ← Cellhealth + Cell'fitness;
39.             end
40.         end
41.     end
42. end
```

3.2.2 Bemenő paraméterek

- **Problemsize:** A döntési változók száma.
- **Cellsnum:** A baktériumok kezdeti populációjának nagysága. Döntési változónként 10-50 ajánlott. Fontos, hogy 2-vel oszthatónak kell lennie!

- **Ned, Nre, Nc, Ns:** Dispersal, Reproduction, Chemotactic, Maximum úszás lépések száma. A keresést három egymásba ágyazott ciklus valósítja meg. A külső ciklus maga a kemotaxis, a baktériumok mozgását megvalósító szakasz. Evolúciós algoritmusról van szó, az egyedek meghatározott élettartammal rendelkeznek. Minél jobb pozícióban vannak az adott probléma szempontjából, annál kevésbé csökken az élettartamuk. A középső ciklus a reprodukció, minden iteráció során az algoritmus rendezzi az egyedeket élettartam szempontjából, majd a halmazt általában középen kettébontva az idősebb baktériumok helyére átmásolja a fiatalabb példányokat, ezzel szimulálva az osztódást. A legbelső ciklus az elimináció, amikor véletlenszerű egyedek meghatározott valószínűséggel likvidálásra kerülnek. A paraméterek értéke nagymértékben befolyásolja a keresés közbeni viselkedést. Általános szabály, hogy a kemotaxis lépések száma nagy legyen, a többi viszonylag kicsi, pl.: 4, 8, 20, 5 [15].
- **Stepsize:** Az alaplépés nagysága a problématerén belül.
- **Ped:** Az elimináció valószínűségi változója, érdemes viszonylag nagy értéket választani a dinamikus kereséshez, például 0,25-0,5.

3.2.3 C# forráskód

Az algoritmus a Rastrigin's tesztfüggvény globális minimum pontját keresi, a kód terjedelme miatt az 1. számú mellékletben található, vagy letölthető az alábbi linkről:

<https://drive.google.com/file/d/0BxE6yHbGFZBAQ1p4dV85d21RUVE/edit?usp=sharing>

3.3 Bees Algorithm (BA)

A Bees Algorithm (BA) eljárást Pham publikálta 2005-ben [16], elsősorban folytonos matematikai függvények szélsőérték keresésére dolgozta ki. Az algoritmus a rajintelligencia eljárások osztályába tartozik, működését tekintve nagyon hasonló a Bacterial Foraging és Particle Swarm algoritmusokhoz. A Bees Algorithm kifejlesztését, mint az neve is mutatja, a méhek táplálékkereső viselkedése inspirálta. A méhkaptárakból először felderítő méhek indulnak nektár után kutatni. A kaptárba visszatérve tudatják a többiekkel a nektár helyét és mennyiségét, amik függvényében adott számú munkás méh tér vissza velük a nektárhoz. Az algoritmus működése során a felderítők folyamatosan keresik az ígéretes pontokat. A pontok fitness értéke alapján további egyedek csatlakoznak hozzájuk, és lokális keresést hajtanak végre. A lokális optimumok elkerülése érdekében az algoritmus folyamatosan hoz létre felderítőket véletlenszerű pozíciókban.



3. ábra: Méhek táplálékkeresés közben

3.3.1 Pszeudokód [17]

```
Input: Problemsize, Beesnum, Sitesnum, EliteSitesnum, PatchSizeinit, EliteBeesnum,
OtherBeesnum
Output: Beebest
1. Population ← InitializePopulation(Beesnum, 1 Problemsize);
2. while ¬StopCondition() do
3.   EvaluatePopulation(Population);
4.   Beebest ← GetBestSolution(Population);
5.   NextGeneration ← ∅;
6.   Patchsize ← ( PatchSizeinit × PatchDecreasefactor);
7.   Sitesbest ← SelectBestSites(Population, Sitesnum);
8.   foreach Sitei ∈ Sitesbest do
9.     RecruitedBeesnum ← ∅;
10.    if i < EliteSitesnum then
11.      RecruitedBeesnum ← EliteBeesnum;
12.    else
13.      RecruitedBeesnum ← OtherBeesnum;
14.    end
15.    Neighborhood ← ∅;
16.    for j to RecruitedBeesnum do
17.      Neighborhood ← CreateNeighborhoodBee(Sitei, Patchsize);
18.    end
19.    NextGeneration ← GetBestSolution(Neighborhood);
20.  end
21.  RemainingBeesnum ← (Beesnum- Sitesnum);
22.  for j to RemainingBeesnum do
23.    NextGeneration ← CreateRandomBee();
24.  end
25.  Population ← NextGeneration;
26. end
27. return Beebest;
```

3.3.2 Bemelő paraméterek

- **Beesnum:** A méhek kezdeti populációjának nagysága. Döntési változónként 10-50 ajánlott.
- **Problemsize:** A döntési változók száma.
- **Sitesnum, EliteSitesnum:** Az ígéretes pontok száma, ahol az algoritmus lokális keresést hajt végre. A második paraméter a kiemelten kezelt ígéretes pontok száma, ahol a lokális keresés alaposabb. $EliteSitesnum < Sitesnum$, pl. 3 és 1.
- **OtherBeesnum, EliteBeesnum:** A munkás méhek száma, akik a felderítők által talált ígéretes pontokban a lokális keresést végrehajtják. A második paraméterben megadott számú munkás méhek csak a kiemelten kezelt ígéretes pontokban keresnek. $EliteBeesnum > Beesnum$, pl. 7 és 3.
- **PatchSizeinit:** Az ígéretes pontokban, lokális keresésnél a keresési tér nagyságának kiinduló értéke, mely a továbbiakban az alábbi képlet szerint alakul:

$$x_i = x_i \pm rand(1) \times PatchSize \quad (1)$$

Az optimumhoz való konvergencia biztosítása érdekében a PatchSize mérete iterációnként egy pozitív konstans értékkel szorzódik, ami kisebb mint 1, pl. 0,95.

3.3.3 C# forráskód

A C# forráskódban az algoritmus a Rastrigin's tesztfüggvény globális minimum pontját keresi. A kód terjedelme miatt az 1. számú mellékletben található, illetve letölthető az alábbi linkről:

<https://drive.google.com/file/d/0BxE6yHbGFZBANUZHTzdfRnJ4aVE/edit?usp=sharing>

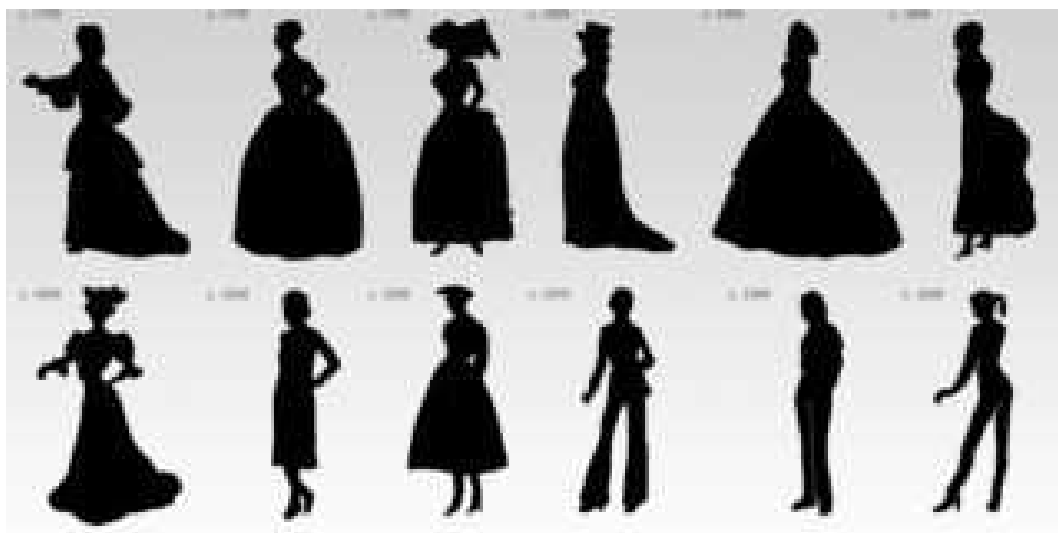
3.4 Cultural Algorithm (CA)

A Cultural Algorithm (CA) heurisztikát először Reynolds publikálta 1994-ben [18]. Az algoritmus az evolúciós eljárások osztályába tartozik, működését tekintve hasonló a Memetic Algorithm eljáráshoz. Működése a társadalom kulturális evolúcióján alapul, ami egy generációkon átívelő jelenség. Egy társadalom kultúrájához tartoznak például különböző népszokások, hitek, viselkedési normák, tudományos ismeretek, stb. Hogy mi marad tartósan a kultúra része, az egyének pozitív és negatív visszajelzései alapján dől el. A kulturális környezet és az ember közötti kapcsolat tanulmányozásával a kulturális ökológia foglalkozik [19].

„A kulturális ökológia azoknak a jelenségeknek és folyamatoknak a megismerésére törekszik, amelyek a környezethez való alkalmazkodás a kultúra egészében vagy részeiben idéz elő. Kulturális teljesítményeket, változásokat állapotokat, magatartási formákat tanulmányoz, amelyek az ember és környezet közötti kölcsönös kapcsolatok következményei.” (Gunda B. 1981, pp. 291. [19])

Az egyik legegyszerűbb példa egy klasszikus zenei mű, például Beethoven IX. szimfóniája. Mivel az adott művészeti alkotás tetszett az embereknek, a kultúra része lett, és generációkon át fennmaradt.

A Cultural Algorithm pontosan ezen az elven működik. Alapvetően egy evolúciós eljárás, ahol a generációk egyedei egyre jobb megoldásokkal állnak elő. Az újdonságot az jelenti, hogy az egyedek megosztják egymás között a keresési információkat, és a jobb megoldásokat eltárolják egy generációkon átívelő kulturális szinten. A kulturális tudásbázis (Knowledge base) a keresés során az egyedek pozitív és negatív visszajelzései alapján változik, egyre jobb megoldásokat adva.



4. ábra: Kulturális evolúció

3.4.1 Pszeudokód [20]

```
Input: Problemsize, Populationnum
Output: KnowledgeBase
1. Population ← InitializePopulation(Problemsize, Populationnum);
2. KnowledgeBase ← InitializeKnowledgebase(Problemsize, Populationnum);
3. while ¬StopCondition() do
4.   Evaluate(Population);
5.   SituationalKnowledgencandidate ← AcceptSituationalKnowledge(Population);
6.   UpdateSituationalKnowledge(KnowledgeBase, SituationalKnowledgencandidate);
7.   Children ← ReproduceWithInfluence(Population, KnowledgeBase);
8.   Population ← Select(Children, Population);
9.   NormativeKnowledgencandidate ← AcceptNormativeKnowledge(Population);
10.  UpdateNormativeKnowledge(KnowledgeBase, NormativeKnowledgencandidate);
11. end
12. return KnowledgeBase;
```

3.4.2 Bemenő paraméterek

- **Populationnum:** A populáció nagysága. Döntési változónként 50-100 ajánlott.
- **Problemsize:** A döntési változók száma.

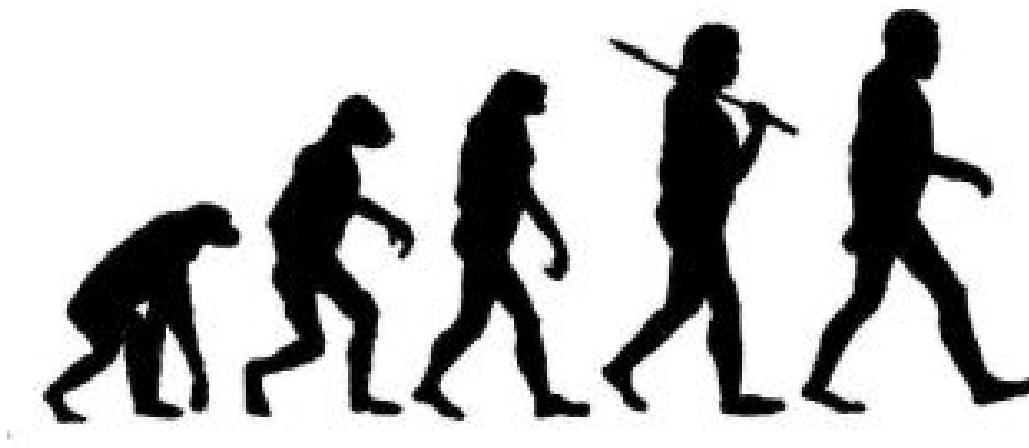
3.4.3 C# forráskód

Az algoritmus a Rastrigin's tesztfüggvény globális minimum pontját keresi, a kód terjedelme miatt az 1. számú mellékletben található, vagy letölthető az alábbi linkről:

<https://drive.google.com/file/d/0BxE6yHbGFZBAQnlOekgzMHZzMzQ/edit?usp=sharing>

3.5 Differential Evolution (DE)

A Differential Evolution (DE) eljárást Storn és Price dolgozta ki 1995-ben [21]. Az algoritmus az evolúciós algoritmusok osztályába tartozik, mely eljárások közös tulajdonsága, hogy Darwin evolúciós elméletén alapul működésük. Ennek megfelelően központi eleme a természetes kiválasztódás, tehát a problémára jobb megoldást adó egyedek hozhatnak létre új generációt (Survival of the fittest theory) [22]. Az evolúció során számos faj esetében megfigyelhető, hogy a generációváltásokkal az adott környezet kihívásainak egyre inkább megfelelő egyedek jöttek létre. A leszármazott egyed új tulajdonságait a szülők tulajdonságainak keresztezéséből kapta.



5. ábra: A darwini evolúció

Először adott számú (N) kiinduló egyedet hoz létre véletlenszerű pozíciókban a problématerületen belül. Az egyedszám az algoritmus futása során nem változik. Az iterációk során adott generáció (G index) minden tagjának egy új leszármazott egyedet ($G+1$ index) hoz létre keresztezéssel az alábbi differenciáló képlet (innen ered az algoritmus neve) alapján:

$$\underline{x}_{i,G+1} = \underline{x}_{r_1,G} + F(\underline{x}_{r_2,G} - \underline{x}_{r_3,G}) \quad (2)$$

$\underline{x}_{r_1}, \underline{x}_{r_2}, \underline{x}_{r_3}$ Véletlenszerűen kiválasztott egyedek
 F Súlyozási tényező

Ezután összehasonlítja az eredeti és a leszármazott egyed rátermettségét az adott probléma szempontjából. A rátermettség fitness függvény alapján kerül meghatározásra. A kedvezőbb megoldást adó egyed lesz tagja a következő generációnak [23].

3.5.1 Pszeudokód [24]

```

Input: Populationsize, Problemsize, Weightingfactor, Crossoverrate
Output: Sbest
1. Population ← InitializePopulation(Populationsize, Problemsize);
2. EvaluatePopulation(Population);
3. Sbest ← GetBestSolution(Population);
4. while ¬ StopCondition() do
5.     NewPopulation ← ∅;
6.     foreach Pi ∈ Population do
7.         Si ← NewSample(Pi, Population, Problemsize, Wf, Cr);
8.         if Cost(Si) ≤ Cost(Pi) then
9.             NewPopulation ← Si;
10.        else
11.            NewPopulation ← Pi;
12.        end
13.    end
14.    Population ← NewPopulation;
15.    EvaluatePopulation(Population);
16.    Sbest ← GetBestSolution(Population);
17. end
18. return Sbest;

19. NewSample(Pi, Population, Problemsize, Weightingfactor, Crossoverrate)
20.     repeat
21.         P1 ← RandomMember(Population);
22.         until P1 ≠ P0 ;
23.         repeat
24.             P2 ← RandomMember(Population);
25.             until P2 ≠ P0 ∨ P2 ≠ P1;
26.             repeat
27.                 P3 ← RandomMember(Population);
28.                 until P3 ≠ P0 ∨ P3 ≠ P1 ∨ P3 ≠ P2 ;
29.                 CutPoint ← RandomPosition(NP);
30.                 S ← ∅;
31.                 for i to NP do
32.                     if i ≡ CutPoint ∧ Rand() < CR then
33.                         Si ← P3i + F × (P1i - P2i );
34.                     else
35.                         Si ← P0i;
36.                     end
37.                 end
38.                 return S;
39. end

```

3.5.2 Bemenő paraméterek

- **Populationsize:** A populáció nagysága. Döntési változónként 10-50 ajánlott.
- **Problemsize:** A döntési változók száma.
- **Weightingfactor:** A differenciáló képletben szereplő súlyozási tényező, melynek értéke 0 és 2 között változhat. A szakirodalom 0,8-as értéket ajánl.
- **Crossoverrate:** Valószínűségi változó, mely azt adja meg, hogy az aktuális egyed legyen-e tagja a következő generációnak, vagy a differenciáló képlet alapján keresztezéssel jöjjön létre egy leszármazott egyed. Az értéke valószínűségi változó lévén 0 és 1 között változhat. A szakirodalom szerint 0,9 az ajánlott érték.

3.5.3 C# forráskód

A C# forráskódban Az algoritmus a Rastrigin's tesztfüggvény globális minimum pontját keresi. A kód terjedelme miatt az 1. számú mellékletben található, illetve letölthető az alábbi linkről:

<https://drive.google.com/file/d/0BxE6yHbGFZBAY3RuQU1BRldMeGs/edit?usp=sharing>

3.6 Harmony Search (HS)

A Harmony Search (HS) algoritmus leírását Geem, Kim és Loganathan publikálta 2001-ben [25]. Működését a Jazz zenészek azon viselkedésmintája inspirálta, amikor közösen kezdenek el játszani valamilyen darabot, és saját játékokat fokozatosan a zenekarhoz igazítják, zenei harmóniát létrehozva. Fals hang esetén kisebb módosításokkal, improvizációval javítanak az előadáson. Az algoritmus a legjobb megoldásokat a Harmónia memóriában (Harmony Memory) tárolja. Az új megoldásokat vagy ebből a harmónia memóriából közvetlenül, vagy a harmónia memóriából kisebb módosításokkal, vagy véletlenszerűen a problématerén belül hozza létre. A harmónia memória mindig úgy aktualizálódik, hogy a legjobb megoldást adó értékek szerepeljenek benne. A Harmony Search algoritmust több különböző jellegű optimalási probléma esetében használták sikerrel, például vízvezeték hálózat tervezésénél [26], valamint szerkezetoptimalási probléma megoldásakor [27].



6. ábra: A Harmony Search algoritmus inspirációja

3.6.1 Pszeudokód [28]

```
Input: Pitchnum, Pitchbounds, Memorysize, Consolidationrate, PitchAdjustrate,
Improvisationmax
Output: Harmonybest
1. Harmonies  $\leftarrow$  InitializeHarmonyMemory(Pitchnum, 1 Pitchbounds, Memorysize);
2. EvaluateHarmonies(Harmonies);
3. for i to Improvisationmax do
4.   Harmony  $\leftarrow$   $\emptyset$ ;
5.   foreach Pitchi  $\in$  Pitchnum do
6.     if Rand()  $\leq$  Consolidationrate then
7.       RandomHarmonyipitch  $\leftarrow$ 
8.       SelectRandomHarmonyPitch(Harmonies, Pitchi);
9.       if Rand()  $\leq$  PitchAdjustrate then
10.        Harmonyipitch  $\leftarrow$  AdjustPitch(RandomHarmonyipitch);
11.       else
12.        Harmonyipitch  $\leftarrow$  RandomHarmonyipitch ;
13.       end
14.     else
15.       Harmonyipitch  $\leftarrow$  RandomPitch(Pitchbounds);
16.     end
17.   end
18.   EvaluateHarmonies(Harmony);
19.   if Cost(Harmony)  $\leq$  Cost(Worst(Harmonies)) then
20.     Worst(Harmonies)  $\leftarrow$  Harmony;
21.   end
22. end
23. return Harmonybest;
```

3.6.2 Bemenő paraméterek

- **Pitchnum:** A döntési változók száma.
- **Pitchbounds:** A döntési változó lehetséges minimum és maximum értéke.
- **Memorysize:** A harmónia memória mérete. Döntési változónként 10-50 ajánlott.
- **Consolidationrate:** Valószínűségi változó, mely azt adja meg, hogy az új megoldás a harmónia memória alapján, vagy véletlenszerűen jöjjön létre. Minél nagyobb az értéke, az algoritmus annál gyorsabban konvergál a harmónia memóriában található megoldások felé. A szakirodalom szerint 0,7 és 0,95 közötti értéket ajánl.
- **PitchAdjustrate:** Ha az új megoldást a harmónia memóriában lévő érték kisebb módosításával hozzuk létre, a módosítás mértékét adja, érdemes 0,5 alatt tartani.
- **ImprovisationMax:** Az algoritmusban az iteráció számnak felel meg.

3.6.3 C# forráskód

Az algoritmus a Rastrigin's tesztfüggvény globális minimum pontját keresi, a kód terjedelme miatt az 1. számú mellékletben található, vagy letölthető az alábbi linkről:

<https://drive.google.com/file/d/0BxE6yHbGFZBANnU3MjgzNVBLUkk/edit?usp=sharing>

3.7 Krill Herd (KH)

A Krill Herd algoritmust Gandomi és Alavi dolgozta ki 2012-ben [29]. A Bacterial Foraging, Bees Algorithm és Particle Swarm eljárásokhoz hasonlóan a rajintelligencia módszerek osztályába tartozik. Működését a déli-sarki világítórák (Antarctic krill), latin nevén *Euphausia superba* [30] állatfaj táplálékkereső viselkedése inspirálta. Ez a rákfélé sűrű rajokba verődik, a köbméterenkénti egyedszám akár 10-30 ezer is lehet. A raj egyrészt védelmet jelent a ragadozókkal szemben, másrészt könnyebben találnak élelmet az egyedek. A két legfontosabb cél tehát a raj sűrűségének növelése, és a minél bőségesebb táplálék lelőhelyek felkutatása. Az algoritmus adott populációszámú egyed problématérén belüli véletlenszerű szétszórásával kezdődik. Az egyedek ezután arra törekednek, hogy minél közelebb kerüljenek a táplálékhoz (a jobb fitness értékű pontok), és a raj azon pontjához, ahol a legmagasabb a rákok sűrűsége. Az egyedek új pozícióját befolyásolja: a raj által gerjesztett mozgás (N_i), a táplálékkereső mozgás (F_i), és egy sztochasztikus szétszóródás (D_i), ami segít elkerülni a lokális optimumokat. Az egyedek mozgását az alábbi képlet írja le:

$$\frac{dx_i}{dt} = N_i + F_i + D_i \quad (3)$$



7. ábra: Euphausia superba raj

3.7.1 Pszeudokód [31]

Input: Populationsize, Problemsize, WeightMovement, WeightForaging, Timechange, Foragingspeed, Maximumdiffusionspeed, Maximuminducedspeed

Output: Sbest

```
1. Initialization: Set the generation counter G=1
2. Initialize the population P according to Populationsize
3.   Add krill individuals randomly, each krill corresponds to a solution
4.   Set the foraging speed according to Foragingspeed
5.   Set the maximum diffusion speed according to Maximumdiffusionspeed
6.   Set the maximum induced speed according to Maximuminducedspeed
7.   Evaluate each krill individual according to its position
8. While G < MaxGeneration do
9.   Sort the krill from best to worst.
10.  for i=1:Populationsize (all krill) do
11.    Perform the following motion calculation.
12.      Motion induced by the presence of other individuals
13.      Foraging motion
14.      Physical diffusion
15.      Update the krill individual position in the search space.
16.      Evaluate each krill individual according to its position.
17.  end for i
18.  Sort the krill from best to worst and find the current best.
19.  G=G+1.
20. end
```

3.7.2 Bemenő paraméterek

- **Populationsize:** A populáció nagysága. Döntési változónként 10-50 ajánlott.
- **Problemsize:** A döntési változók száma.
- **WeightMovement:** A raj által gerjesztett mozgás súlyozása. Az értékét 0 és 1 között érdemes változtatni. A szakirodalom szerint 0,9 az ajánlott érték.

- **WeightForaging:** A táplálkozás által gerjesztett mozgás súlyozása. Az értékét 0 és 1 között érdemes változtatni. A szakirodalom szerint 0,9 az ajánlott érték.
- **Timechange:** Az időváltozás, amíg az egyed egyik pozícióból átmehet egy másikba. Minél kisebb értéket adunk meg, annál alaposabb a lokális keresés. Fennáll viszont a veszély, hogy túl kicsi értéknél az egyedek lokális optimumban ragadnak.
- **Foragingspeed:** A sebesség, amivel az egyed a táplálék felé mozog. A szakirodalom szerint 0,02 az ajánlott érték.
- **Maximumdiffusionspeed:** A sebesség, amivel az egyedek véletlenszerűen szétszóródhatnak. Az értékét 0,002 és 0,01 között érdemes változtatni.
- **Maximuminducedspeed:** A sebesség, amivel az egyed a raj legnagyobb sűrűségű pontja felé mozog. A szakirodalom szerint 0,01 az ajánlott érték.

3.7.3 C# forráskód

Az algoritmus a Rastrigin's tesztfüggvény globális minimum pontját keresi, a kód terjedelme miatt az 1. számú mellékletben található, vagy letölthető az alábbi linkről:

<https://drive.google.com/file/d/0BxE6yHbGFZBATVFWU05wbXlGMzQ/edit?usp=sharing>

3.8 Memetic Algorithm (MA)

A Memetic Algorithm (MA) leírását Moscato [32] dolgozta ki 1989-ben. A Memetics a kulturális információk cserélődését, átadását leíró teória, mely Richard Dawkins 1976-ban megjelent „The Selfish Gene” című művében jelent meg. Lényege, hogy a kulturális információ áramlását az univerzális darwinizmus jegyében írja le. Az univerzális darwinizmus elmélete szerint minden komplex rendszer leírható a biológiai darwini evolúció analógiájára, ahol diszkrét információ egységek terjednek és öröklődnek az individuumok között. A meme (mém) a kulturális információ alapegysége (pl. egy ötlet, felfedezés, észrevétel, stb.), aminek az elnevezése a biológiában jól ismert génből ered. A kulturális információt az egyének elméje tárolja, az egyének közötti kommunikáció révén pedig terjed, sokszorozódik, reprodukálja önmagát. Az információ terjedése során a befogadó individuum elméjének függvényében torzulhat, mutálódhat, ezzel gyengítve, vagy éppen erősítve eredeti jelentését. Az algoritmus alapvetően egy evolúciós eljárás, ahol az egyedek kommunikálnak egymással. A keresési információk mémként terjednek a populációban, az egyedek saját fitness értéküktől függően gyengítik, vagy erősítik a mémek jelentőségét. A mémeket az algoritmus egy generációkon átívelő kulturális szinten tárolja. A Memetic Algorithm működési elvét tekintve közeli rokona a korábban ismertetett Cultural Algorithm eljárásnak.



8. ábra: A Memetic Algorithm inspirációja

3.8.1 Pszeudokód [33]

```
Input: ProblemSize, Popsze, MemePopsze
Output: Sbest
1. Population ← InitializePopulation(1 ProblemSize, Popsze);
2. while ¬StopCondition() do
3.   foreach Si ∈ Population do
4.     Sicost ← Cost(Si);
5.   end
6.   Sbest ← GetBestSolution(Population);
7.   Population ← StochasticGlobalSearch(Population);
8.   MemeticPopulation ← SelectMemeticPopulation(Population, MemePopsze);
9.   foreach Si ∈ MemeticPopulation do
10.    Si ← LocalSearch(Si);
11.  end
12. end
13. return Sbest;
```

3.8.2 Bemenő paraméterek

- **ProblemSize:** A döntési változók száma.
- **Populationsize:** A populáció nagysága. Döntési változónként 10-50 ajánlott.
- **MemePopsze:** A generációkon átívelő kulturális szinten eltárolt mémek száma. Döntési változónként 10-50 ajánlott.

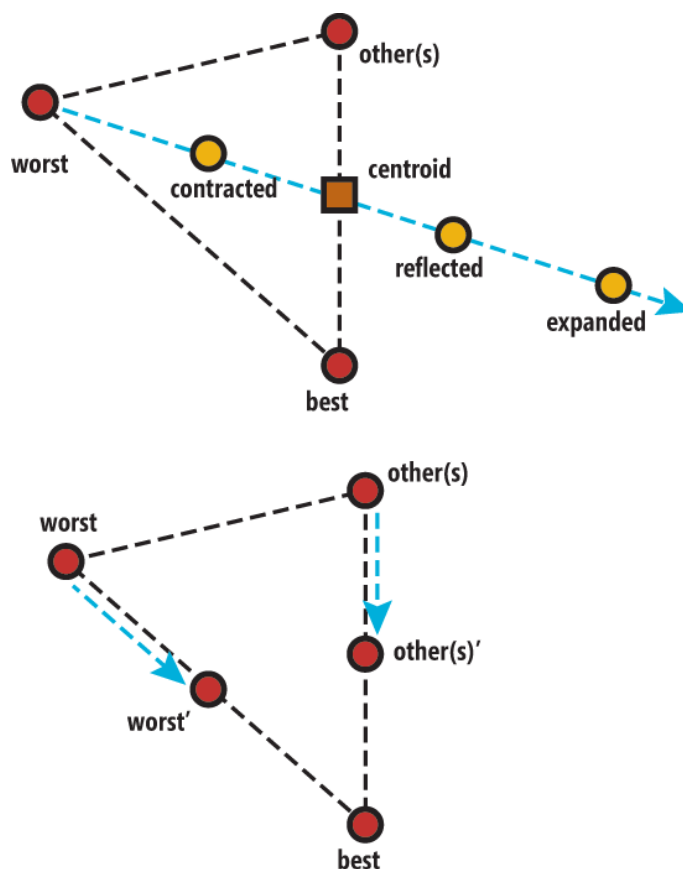
3.8.3 C# forráskód

Az algoritmus a Rastrigin's tesztfüggvény globális minimum pontját keresi, a kód terjedelme miatt az 1. számú mellékletben található, vagy letölthető az alábbi linkről:

<https://drive.google.com/file/d/0BxE6yHbGFZBAVUJhU0ZFMWxRelE/edit?usp=sharing>

3.9 Nelder-Mead Algorithm (NM)

A Nelder-Mead algoritmus nevét kitalálói után kapta, a módszert Nelder és Mead dolgozta ki 1965-ben [34]. A legrégebbi általam vizsgált heurisztikus kereső eljárás, mely még napjainkban is megállja a helyét. A szakirodalomban Amoeba Method (Amőba módszer) néven is szokták emlegetni. Az algoritmus egy szimplex kereső eljárás, melynek lényege, hogy a keresés során mindig több lehetséges megoldás van. A lehetséges megoldások halmazát fitness érték szerint háromfelé bontja, a legjobb megoldás (best), a legrosszabb megoldás (worst), és az összes többi (others). Az iterációk során az algoritmus mindig a legrosszabb megoldást próbálja jobbra cserélni. Elsőként egy egyenest határoz meg, aminek kiinduló pontja a legrosszabb pont, irányát pedig a legjobb pont és az összes többi pont súlypontja (centroid) adja. A legrosszabb megoldást az egyenesen lévő három lehetséges megoldással hasonlítja össze, az összezsugorított (contracted), a tükrözött (reflected) és a kiterjesztett (expanded) pontokkal. Ha bármelyik jobb megoldást ad, akkor megtörténik a csere, ellenkező esetben az esetben az amőba összezsugorítja önmagát, minden pont távolságát megfelelve a legjobb ponthoz képest (9. ábra).



9. ábra: A Nelder-Mead algoritmus működési elve [35]

3.9.1 Pszeudokód [36]

```
Input: ProblemSize, Amoebasize
Output: Sbest
1. generate amoebaSize random solutions
2. while not done loop
3.     compute centroid, reflected
4.     if reflected is better than best solution then
5.         compute expanded
6.         replace worst solution with better of reflected, expanded
7.     else if reflected is worse than all but worst then
8.         if reflected is better than worst solution then
9.             replace worst solution with reflected
10.        end if
11.        compute contracted
12.        if contracted is worse than worst
13.            shrink the amoeba
14.        else
15.            replace worst solution with contracted
16.        end if
17.    else
18.        replace worst solution with reflected
19.    end if
20. end loop
21. return best solution found
```

3.9.2 Bemenő paraméterek

- **ProblemSize:** A döntési változók száma.
- **AmoebaSize:** A populáció nagysága. Döntési változónként 10-25 ajánlott.

3.9.3 C# forráskód

Az algoritmus a Rastrigin's tesztfüggvény globális minimum pontját keresi, a kód terjedelme miatt az 1. számú mellékletben található, vagy letölthető az alábbi linkről:

<https://drive.google.com/file/d/0BxE6yHbGFZBAYklpUzNRbzZ2aXM/edit?usp=sharing>

3.10 Particle Swarm Optimization (PSO)

A Particle Swarm (részecske-csapat) algoritmust 1995-ben Eberhart és Kennedy dolgozta ki [37]. A Bacterial Foraging, Bees Algorithm és Krill Herd eljárásokhoz hasonlóan rajintelligencia módszer. Napjaink egyik legígéretesebb metaheurisztikus optimaló algoritmus. Működését a madár és halrajok mozgása inspirálta. A nagyszámú egyed mozgásában egyfajta rendezettség figyelhető meg. Érzékelik egymás helyzetét, bizonyos mértékben pedig emlékeznek a korábbi pozíciókra.

A keresés adott számú részecske létrehozásával kezdődik, amik véletlenszerű kiindulási pontokban helyezkednek el. A részecskék a problématerben az egyre jobb megoldást adó helyek felé mozognak, a csapatot a legjobb egyedek vezetik. Tisztában vannak aktuális és addigi legjobb pozíciójukkal, valamint a részecske csapat addigi legjobb pozíciójával, az új pozíció kiszámításához ezeket is figyelembe veszi az eljárás. Hogy az adott pozíció mennyire jó az adott keresés szempontjából, mindig egy fitness függvénnyel határozható meg. Az iterációk során az alábbi képletek alapján számítható ki a részecskék új pozíciója (4), (5):

$$V_{id} = w \times V_{id} + c_1 \times rand() \times (p_{id} - x_{id}) + c_2 \times rand() \times (p_{gd} - x_{id}) \quad (4)$$

$$x_{id} = x_{id} + V_{id}t \quad (5)$$

V_{id}	Részecske sebessége
w	Gyorsulás konstans
c_1	Egyéni súlyozás konstans
c_2	Szociális súlyozás konstans
p_{id}	Részecske eddigi legjobb pozíciója
p_{gd}	Csapat eddigi legjobb pozíciója
x_{id}	Részecske aktuális pozíciója
t	Időegység

Az iteráció addig folytatódik, míg nem teljesül valamilyen leállási feltétel. Evolúciós algoritmusok esetében sokszor nem egyszerű egyértelműen jó leállási feltételt találni. A Particle Swarm módszernél a feltétel többek között lehet megszabott számú iteráció végrehajtása, vagy ha a csapat legjobb pozíciója bizonyos számú iteráció után sem változott meg [38].



10. ábra: Madárraj rendezett mozgása

3.10.1 Pszeudokód [39]

```
Input: ProblemSize, Populationsize, w, c1, c2
Output: Pg best
1. Population  $\leftarrow \emptyset$ , Pg best  $\leftarrow \emptyset$ ;
2. for i = 1 to Populationsize do
3.     Pvelocity  $\leftarrow$  RandomVelocity();
4.     Pposition  $\leftarrow$  RandomPosition(ProblemSize);
5.     Pcost  $\leftarrow$  Cost(Pposition);
6.     Pp best  $\leftarrow$  Pposition;
7.     if Pcost  $\leq$  Pg best then
8.         Pg best  $\leftarrow$  Pp best;
9.     end
10. end
11. while  $\neg$ StopCondition() do
12.     foreach P  $\in$  Population do
13.         Pvelocity  $\leftarrow$  UpdateVelocity(Pvelocity, Pg best, Pp best, w, c1, c2);
14.         Pposition  $\leftarrow$  UpdatePosition(Pposition, Pvelocity);
15.         Pcost  $\leftarrow$  Cost(Pposition);
16.         if Pcost  $\leq$  Pp best then
17.             Pp best  $\leftarrow$  Pposition;
18.             if Pcost  $\leq$  Pg best then
19.                 Pg best  $\leftarrow$  Pp best;
20.             end
21.         end
22.     end
23. end
24. return Pg best;
```

3.10.2 Bemelő paraméterek

- **ProblemSize:** A döntési változók száma.
- **Populationsize:** A populáció nagysága. Döntési változónként 10-50 ajánlott.
- **w:** Gyorsulás konstans, minél nagyobb az értéke, annál dinamikusabban mozognak a részecskék a problématerben. A szakirodalomban ajánlott érték 0,729.
- **c1:** Egyéni (kognitív) súlyozás konstans, mely a részecske egyéni legjobb pozíciójának fontosságát befolyásolja az új pozíció meghatározása során. A szakirodalomban ajánlott érték 1,49445.
- **c2:** Szociális súlyozás konstans, mely a raj legjobb pozíciójának fontosságát befolyásolja az új pozíció meghatározása során. A szakirodalomban ajánlott érték 1,49445.

3.10.3 C# forráskód

Az algoritmus a Rastrigin's tesztfüggvény globális minimum pontját keresi, a kód terjedelme miatt az 1. számú mellékletben található, vagy letölthető az alábbi linkről:

<https://drive.google.com/file/d/0BxE6yHbGFZBAd1JmM0sxMU5lWVU/edit?usp=sharing>

3.11 Random Search (RS)

A Random Search algoritmus egyszerű véletlen kereső eljárás, a problématerén belül azonos valószínűséggel veheti fel bármelyik pozíciót [40]. Az iterációk során szintén véletlenszerűen új megoldást hoz létre, melyek függetlenek a korábbi megoldásoktól.



11. ábra: A Random Search algoritmus működését kizárólag a véletlen befolyásolja

3.11.1 Pszeudokód

```
Input: NumIterations, ProblemSize, Populationsize, SearchSpace
Output: Best
1. Best  $\leftarrow \emptyset$ ;
2. foreach iteri  $\in$  NumIterations do
3.     candidatei  $\leftarrow$  RandomSolution(ProblemSize, SearchSpace);
4.     if Cost(candidatei) < Cost(Best) then
5.         Best  $\leftarrow$  candidatei;
6.     end
7. end
8. return Best;
```

3.11.2 Bemenő paraméterek

- **NumIterations:** Iterációk száma.
- **ProblemSize:** A döntési változók száma.
- **SearchSpace:** A dimenziónkénti keresési tartomány.

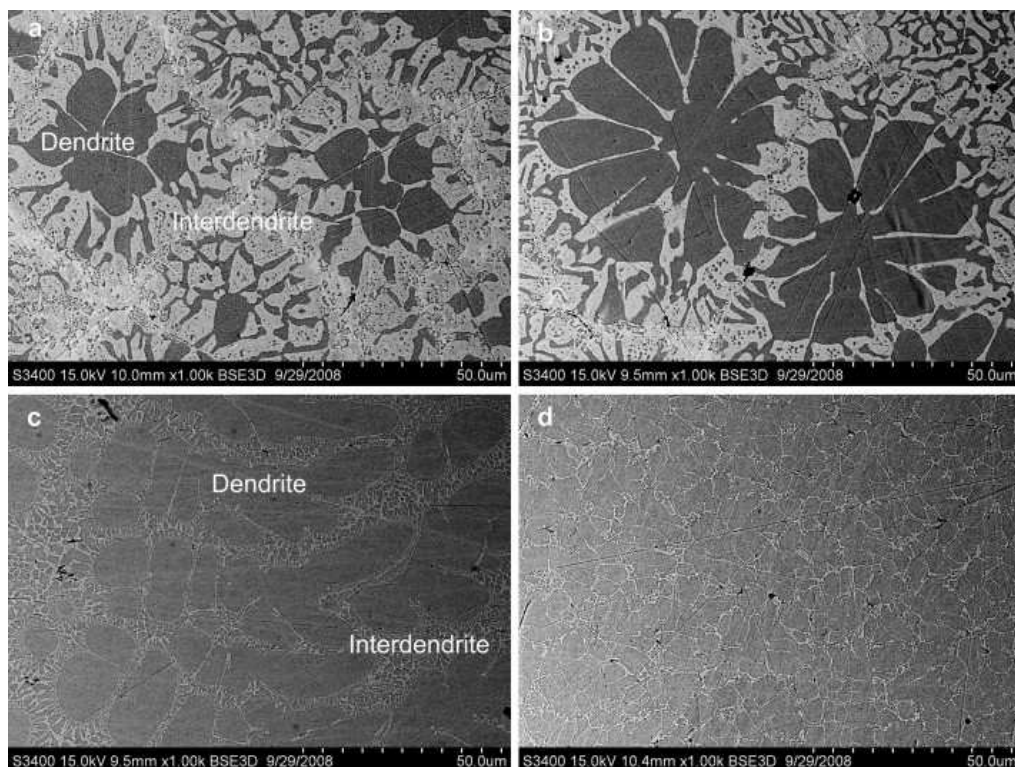
3.11.3 C# forráskód

Az algoritmus a Rastrigin's tesztfüggvény globális minimum pontját keresi, a kód terjedelme miatt az 1. számú mellékletben található, vagy letölthető az alábbi linkről:

<https://drive.google.com/file/d/0BxE6yHbGFZBANDIDWmRmMDJxVVE/edit?usp=sharing>

3.12 Simulated Annealing (SA)

A módszert Kirkpatrick, Gelatt és Vecchi dolgozta ki 1983-ban [41]. A Simulated Annealing (Szimulált Hűtés) algoritmus egy fizikai jelenségen alapuló heurisztika. A rajtintelligencia és evolúciós eljárásokhoz hasonlóan a természet inspirálta kidolgozását. A metallurgiában bizonyos anyagok kedvező tulajdonságokra tesznek szert, ha felhevítik, majd szabályozott körülmények között lehűtik őket. A folyamat során átalakul kristályszerkezetük, mivel a felhevített anyagban az atomok képesek elmozdulni, a hűtési folyamat során pedig új, számukra kedvezőbb pozíciót vesznek fel (energiaminimumra törekednek). A kiinduló permutáció véletlenszerűen jön létre. Az algoritmus az iteráció során új permutációt hoz létre a sztochasztikus 2-opt lokális kereső algoritmus segítségével. Ezután egy döntési függvény (Metropolisz kritérium) a permutációk költsége alapján meghatározza, hogy az új permutáció vagy a régi a jobb. Az iterációk során folyamatosan csökken a hőmérséklet, melynek következtében a döntési függvény egyre kevésbé fogadja el az új megoldásokat. Az algoritmust eredetileg kombinatorikai problémák megoldására fejlesztették. A dolgozatomban használt, folytonos változókkal való munkára alkalmas változatát Corana, Marchesi, Martini és Ridella alkotta meg 1987-ben [42].



12. ábra: Temperálás során kialakuló kristályszerkezet [43]

3.12.1 Pszeudokód [44]

```
Input: ProblemSize, iterationsmax, tempmax
Output: Sbest
1. Scurrent 1 ← CreateInitialSolution(ProblemSize);
2. Sbest ← Scurrent;
3. for i = 1 to iterationsmax do
4.     Si ← CreateNeighborSolution(Scurrent);
5.     tempcurr ← CalculateTemperature(i, tempmax);
6.     if Cost(Si) ≤ Cost(Scurrent) then
7.         Scurrent ← Si;
8.         if Cost(Si) ≤ Cost(Sbest) then
9.             Sbest ← Si;
10.        end
11.    else if Exp( (Cost(Scurrent) - Cost(Si)) / tempcurr ) > Rand() then
12.        Scurrent ← Si;
13.    end
14. end
15. return Sbest;
```

3.12.2 Bemenő paraméterek

- **ProblemSize:** A döntési változók száma.
- **iterationsmax:** Iterációk száma.
- **tempmax:** A kiinduló hőmérséklet, aminek a nagyságát érdemes úgy megadni, hogy az algoritmus a keresés elején a teljes problémateret bejárhasa. Az alapvető szabály tehát, hogy a kiinduló hőmérséklet legyen nagyobb, mint a legnagyobb értelmezési tartományú dimenzió.

3.12.3 C# forráskód

A C# forráskódban Az algoritmus a Rastrigin's tesztfüggvény globális minimum pontját keresi. A kód terjedelme miatt az 1. számú mellékletben található, illetve letölthető az alábbi linkről:

<https://drive.google.com/file/d/0BxE6yHbGFZBAY2M1UkVmOFRLYWM/edit?usp=sharing>

4 Globális optimum keresése ismert tesztfüggvények esetében

Az optimalálási problémák sok esetben visszavezethetők folytonos matematikai függvények globális szélsőértékeinek keresésére, ahol a döntési változók a matematikai függvény dimenzióit adják. Vegyük például az egyik legegyszerűbb egyváltozós függvényt, melynek képlete:

$$f(x) = x^2 \quad (6)$$

Az egyetlen döntési változó az x . Tegyük fel, hogy az értéke a $[-5; 5]$ tartományba eső értékeket vehet fel, ezzel egy, a változóra vonatkozó feltételt definiáltunk. Keressük e matematikai függvény globális minimum értékét.

A legtöbben valószínűleg azonnal meg tudják mondani, hogy a függvény globális minimuma az $x = 0$ pontban van. A globális maximum értékének meghatározása sem jelent nagyobb kihívást, igaz abból mindjárt két darab is van, az $x = -5$ és az $x = 5$ értékeknél. Ha az ember korábban már találkozott ezzel a függvénnyel, akkor hatalmas előnyt élvez akár a leggyorsabb számítógéppel szemben is, nevezetesen, hogy probléma specifikus ismeretekkel rendelkezik a vizsgált függvényt illetően. A számítógépnek a feladat megoldásához kereső algoritmusra van szüksége. Általában egzakt megoldást adó analitikus módszerrel, valamilyen logika szerint megvizsgálja az összes lehetséges megoldást, majd a számítás végén megadja a megoldást. Az egyváltozós függvény után tekintsünk egy kétváltozós folytonos függvényt, melynek képlete:

$$f(x, y) = -20 \exp\left(-0.2\sqrt{0.5(x^2 + y^2)}\right) - \exp(0.5(\cos(2\pi x) + \cos(2\pi y))) + 20 + e \quad (7)$$

Ez alapján valószínűleg már kevesebben tudnák azonnal megmondani, hogy a függvénynek hol vannak a globális szélsőértékei. A képlet egy gyakran használt optimalálási tesztfüggvényt, az Ackley's function (F1) függvényt definiálja [45].

Tegyük fel, hogy az x és y döntési változók $[-32,768; 32,768]$ tartományba eső értékeket vehetnek fel. A függvény globális minimuma az $x = 0, y = 0$ pontban található, értéke 0.

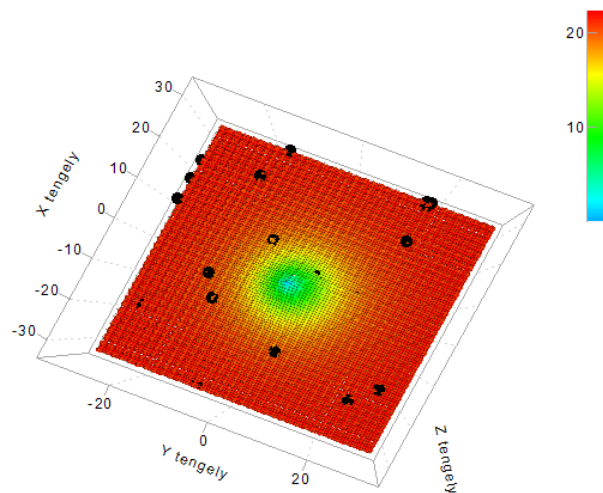
Gyakran előfordul, hogy a fentieknél sokkal bonyolultabb matematikai függvények globális szélsőértékeit keressük. A nagyméretű, sokváltozós problémák esetében a hagyományos kereső eljárások csődöt mondanak. A lehetséges megoldások száma túl sok ahhoz, hogy mindegyiket külön megvizsgáljuk. Ebben az esetben nyújtanak segítséget a heurisztikus algoritmusok.

Mint az a bevezetésben említésre került, a heurisztikus algoritmusok előnye, hogy bonyolult problémák esetén is képesek viszonylag rövid idő alatt eredményt szolgáltatni. Hátrányuk azonban, hogy nem garantálható teljes bizonyossággal az optimális megoldás megtalálása. Az összes általam vizsgált heurisztikus algoritmusra igaz, hogy működésüket sztochasztikus események befolyásolják. Kis túlzással ez azt jelenti, hogy gyakorlatilag nincs két egyforma futás. A heurisztikus algoritmusokkal végzett optimalálás egy igen nehéz és összetett folyamat, a bemenő paraméterek és nagyszámú változó gyakorlatilag végtelen variációs lehetőséget jelent. A káosz elkerülése érdekében mindenképp szükséges a teszt körülményeinek pontos dokumentálása. Az algoritmusok bemenő paramétereit az ismert tesztfüggvények globális optimum értékének keresésekor a szakirodalom által javasolt értékekre állítottam (2. táblázat). A javasolt értékektől való bármi nemű esetleges eltérést a későbbiekben természetesen külön jelzem, annak pontos indoklásával egyetemben.

Az egyik legfontosabb tényező a futás során az algoritmusok leállási feltételének meghatározása. A keresés addig tart, amíg a leállási feltétel nem teljesül. Heurisztikus algoritmusok esetében az egyik legegyszerűbb leállási feltétel a fix iterációszám. Lehetséges a futás konkrét idejének maximalizálása, vagy fitness limit meghatározása is. A fitness limit vonatkozhat a populáció által talált legjobb fitness értékre, ami ha elég közel van az elvárt megoldáshoz, az algoritmus leáll. Vonatkozhat továbbá a populáció tetszőleges egyedeire, ha azok fitness értékének eltérése egy adott limiten belül van, az algoritmus szintén leáll. Természetesen a felsoroltakon kívül számos egyéb leállási feltétel meghatározható.

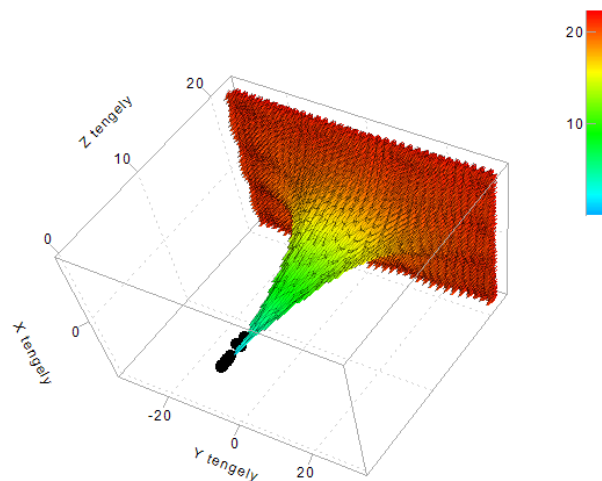
A dolgozatban leállási feltételként a fix iterációszám mellett döntöttem, mivel használatával véleményem szerint könnyebben hasonlítható össze a különböző algoritmusok karakterisztikája. Nyilvánvaló hátránya, hogy egy viszonylag durva beavatkozás az algoritmusok futásába. Egy gyorsan konvergáló algoritmus esetében előfordulhat, hogy a limitnél sokkal kevesebb iteráció alatt eredményre jut, de kénytelen folytatni futását, ami az erőforrások pazarlását jelenti. Lassan konvergáló algoritmus esetében pedig megtörténhet, hogy még nem találta meg a legjobb megoldást, futását azonban meg kell szakítania az iterációs korlát miatt. A fenti esetek minimalizálása érdekében az iterációs határt nagy körütekintéssel, a feladat sajátosságait szem előtt tartva próbáltam meghatározni. A kétváltozós függvények hatékonyságvizsgálata egy viszonylag kis dimenziószámú probléma, ezért az algoritmusok számára az iterációs határ 500 volt.

A kétváltozós tesztfüggvények 3 dimenziós megjelenítésére az ILNumerics matematikai függvénykönyvtár ILPanel vezérlőjét használtam. A vezérlőt úgy állítottam be, hogy alapesetben OpenGL módban próbáljon meg futni, ami kihasználja a különböző videokártyák által biztosított hardveres gyorsítást. Szem előtt kellett tartanom azonban, hogy nem minden számítógépben található OpenGL 3.1 verzióval kompatibilis GPU. Az utóbbi eshetőséget a program érzékeli, és automatikusan visszavált szoftveres (GDI) renderelésre, mely hardverfüggetlen módon működik. Hátránya azonban, hogy a szoftveres mód jelentősen lassabb, gyengébb hardverkörnyezetben szembetűnő lassulást eredményezhet.



13. ábra: Particle Swarm algoritmus futásának 1. iterációja

A futás során az algoritmusok populációinak egyedeit 3 dimenziós ábra is szemlélteti, fekete gömbök segítségével. Az 13. ábra a Particle Swarm algoritmus első iterációja látható, a részecskék véletlenszerű pozíciókban helyezkednek el.



14. ábra Particle Swarm algoritmus futásának 100. iterációja

Ha 100 iteráción át futtatom az algoritmust, akkor az optimálishoz sokkal közelebbi eredményt szolgáltat. A 14. ábra látható, hogy a részecskék aktuális pozícióját szimbolizáló fekete gömbök szinte teljes mértékben besorakoztak a globális minimum pontra.

Az eredmények vizsgálatakor a sztochasztikus tényező miatt érdemes statisztikai módszereket használni. Ha csupán egyetlen futás eredményeit venném alapul, megtörténhetne, hogy a legegyszerűbb véletlen kereső algoritmus sokkal jobb eredményt ad vissza, mint a legfejlettebb heurisztikus rajintelligencia módszer, mivel a véletlen generált pozíció pontosan a globális optimumban van. Az algoritmusok hatékonyságvizsgálata során ezért minden tesztfüggvény esetében 100 darab Monte Carlo módszerrel végzett futtatás adatait dokumentáltam.

Az adatok szemléltetésére egyrészt konvergencia grafikonokat használtam. Az első konvergencia grafikon mindig a 100 darab Monte Carlo futás során az adott iterációig megtalált legjobb fitness értékek átlagát jeleníti meg, ezzel viszonylag jó képet adva az algoritmusok általános konvergencia sebességéről, azaz hogy milyen gyorsan találják meg az optimumot. A második konvergencia grafikon a 100 darab Monte Carlo futás során talált legjobb végeredményhez vezető konvergenciát szemlélteti, azaz kizárólag a legjobb eredményt adó futás adatait jeleníti meg. A második grafikon az algoritmusok karakterisztikáját bizonyos szempontból jobban szemlélteti, mivel nem lép fel az átlagszámítás okozta torzítás. Eredményeit viszont némi fenntartással kell kezelni, mivel sokkal inkább befolyásolják a sztochasztikus események. A konvergencia grafikonok az első 100 iteráció adatait jelenítik meg.

A futásokról statisztikai táblázatot is készítek, ami algoritmusonként lebontva, 500 iteráció után tartalmazza az eredmények (a populáció legjobb fitness értéke) átlagát, a legjobb eredményt, a legrosszabb eredményt, valamint a fitness értékek szórását (standard deviation). A táblázat átfogó képet ad az algoritmusok hatékonyságáról és megbízhatóságáról, hogy mekkora eséllyel találják meg az optimumot, illetve esetlegesen milyen mértékben tévednek.

A 2. táblázatban a szakirodalom által az algoritmusokhoz javasolt bemenő paramétereket foglaltam össze, a teszt során ezeket az értékeket használtam.

Metaheurisztikus algoritmusok hatékonyságvizsgálata saját keretrendszer által létrehozott komplex tesztfüggvények segítségével, az eredmények felhasználása futódaru főtartó optimalásához

Algoritmus	Paraméter neve	Paraméter értéke
Artificial Immune Network (AiNet)	PopulationSize	50
	Nclones	10
	Nrandom	2
	AffinityTreshold	0,1
Bacterial Foraging Algorithm (BFOA)	Cellsnum	50
	Ned	4
	Nre	5
	Nc	25
	Ns	10
	Stepsize	0,05
	Ped	0,25
Bees Algorithm (BA)	Beesnum	50
	Sitesnum	3
	EliteSitesnum	1
	PatchSizeinit	3,0
	OtherBeesnum	2
	EliteBeesnum	7
Cultural Algorithm (CA)	Populationnum	50
Differential Evolution (DE)	Populationsize	50
	Weightingfactor	0,8
	Crossoverrate	0,9
Harmony Search (HS)	Memorysize	50
	Consolidationrate	0,95
	PitchAdjustrate	0,7
Krill Herd (KH)	Populationsize	50
	WeightMovement	0,9
	WeightForaging	0,9
	Timechange	0,5
	Foragingspeed	0,02
	Maximumdiffusionspeed	0,002
	Maximuminducedspeed	0,01
Memetic Algorithm (MA)	Populationsize	50
	MemePopsiz	16
Nelder-Mead Algorithm (NM)	Amoebasize	20
Random Search (RS)	Populationsize	50
Particle Swarm Optimization (PSO)	Populationsize	50
	W	0,729
	C1	1,49445
	C2	1,49445
Simulated Annealing (SA)	Tempmax	Searchspace (max)- Searchspace (min)

2. táblázat: A heurisztikus algoritmusok szakirodalom által javasolt bemenő paraméterei

4.1 Ackley's function (F1)

Az Ackley's tesztfüggvény globális optimuma az $x_i = 0, i = 1, \dots, n$ pontban található, fitness értéke $f(x) = 0$. Az n változó a dimenziószámot jelöli, a dimenziók minimum és maximum korlátjára az ajánlott érték $-32,768 \leq x_i \leq 32,768, i = 1, \dots, n$. A függvényt az alábbi képlet írja le:

$$f(x) = -20 \times \exp \left(-0,2 \times \sqrt{\frac{1}{n} \sum_{i=1}^n (x_i^2)} \right) - \exp \left(\frac{1}{n} \sum_{i=1}^n \cos(2\pi x_i) \right) + 20 + \exp(1) \quad (8)$$

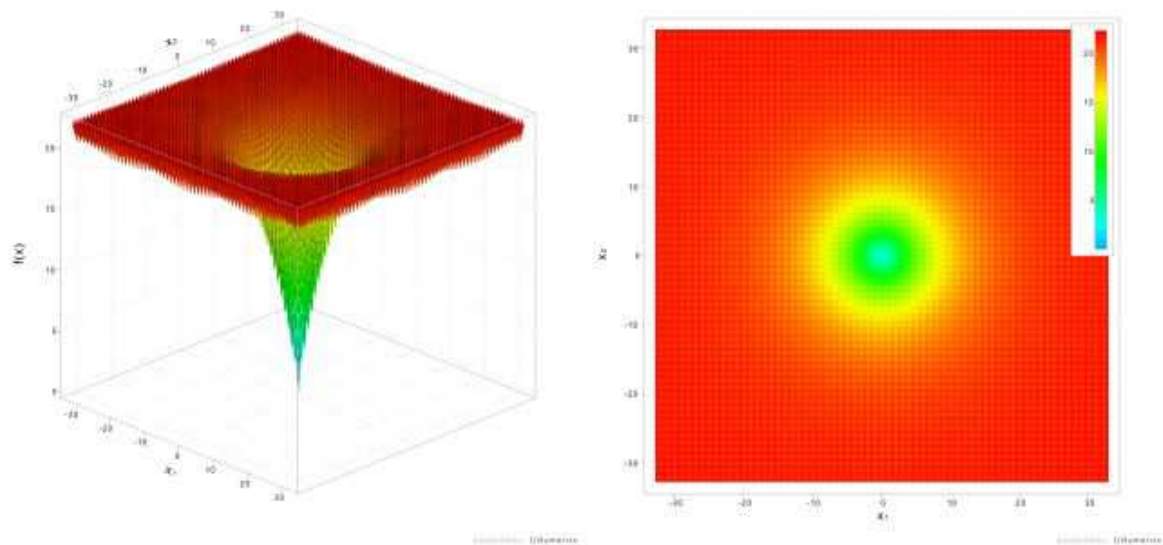
A kétváltozós Ackley's tesztfüggvény háromdimenziós ábrázolása és szintérképe a 15. ábraán látható.

4.1.1 C# forráskód

```
public double Evaluate(double[] coords)
{
    int n = 2;
    double a = 20;
    double b = 0.2;
    double c = 2 * Math.PI;
    double s1 = 0;
    double s2 = 0;

    for (int i = 0; i < n; i++)
    {
        s1 = s1 + Math.Pow(coords[i], 2);
        s2 = s2 + Math.Cos(c * coords[i]);
    }

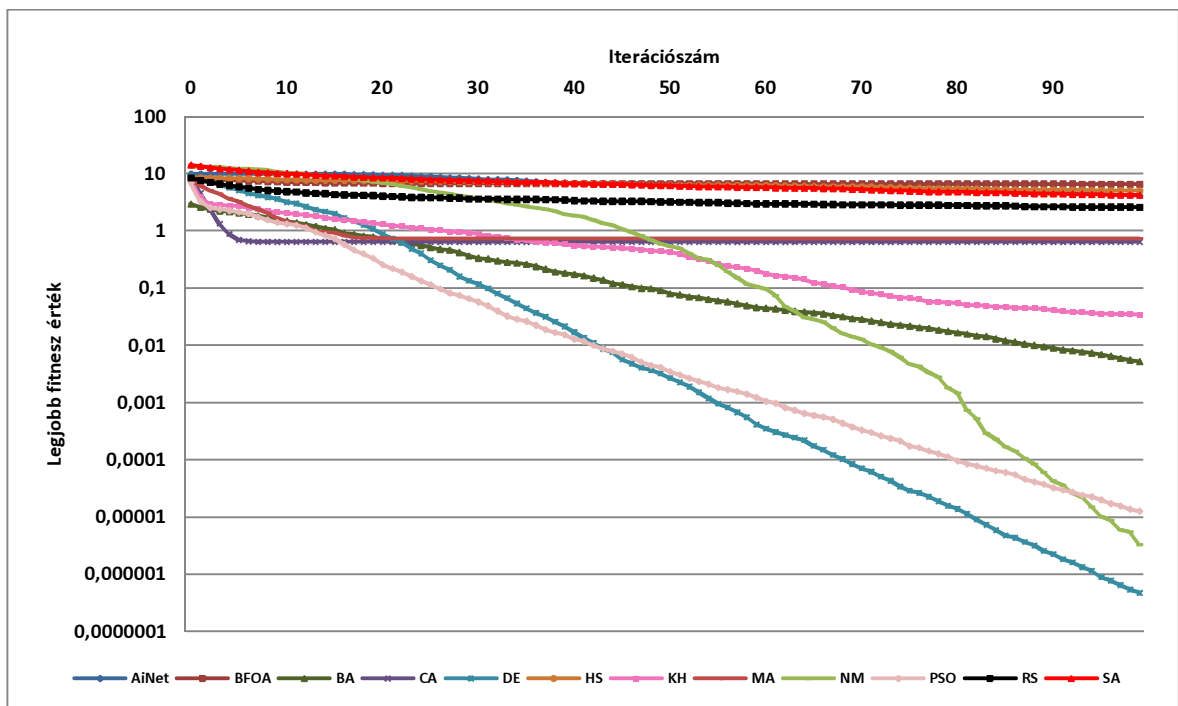
    double fitness = -a * Math.Exp(-b * Math.Sqrt(1.0 / n * s1)) - Math.Exp(1.0 / n *
s2) + a + Math.Exp(1.0);
    return -fitness;
}
```



15. ábra: A kétváltozós Ackley's tesztfüggvény háromdimenziós ábrázolása és szintérképe

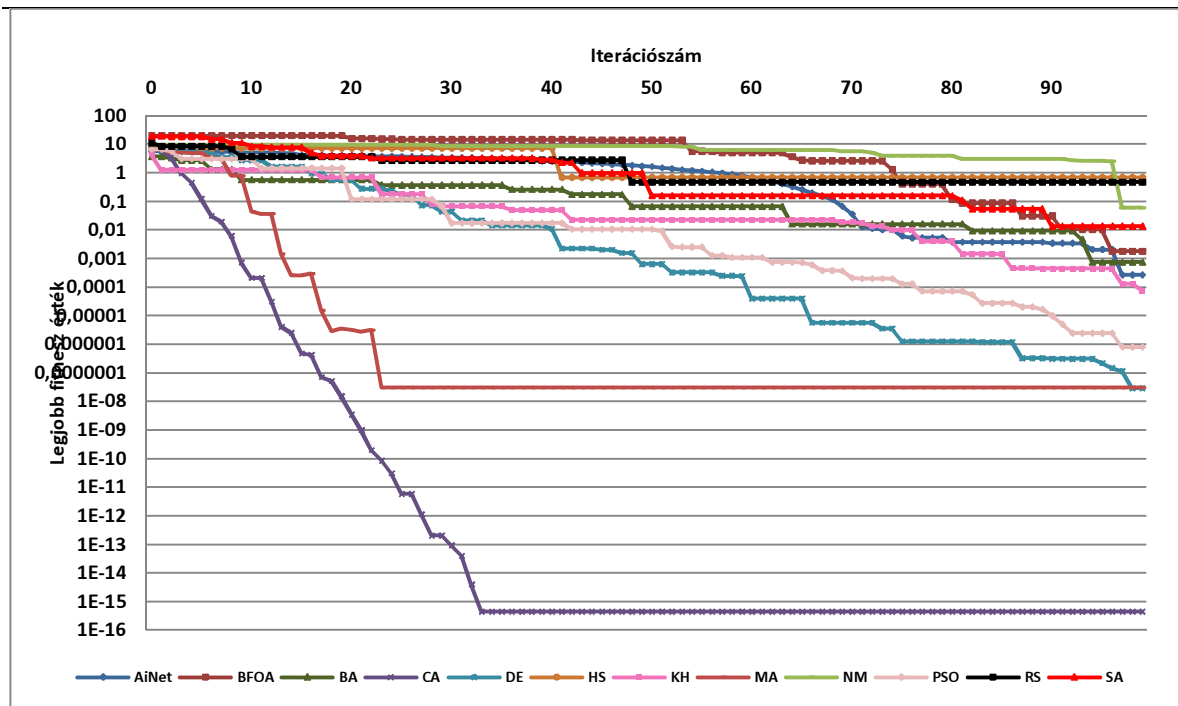
4.1.2 Mérési eredmények

4.1.2.1 Konvergencia grafikonok



16. ábra: Legjobb fitness értékek átlaga 100 Monte Carlo futás adatai alapján (F1)

Metaheurisztikus algoritmusok hatékonyságvizsgálata saját keretrendszer által létrehozott komplex tesztfüggvények segítségével, az eredmények felhasználása futódaru főtartó optimalásához



17. ábra: Tényleges legjobb fitness érték 100 Monte Carlo futás adatai alapján (F1)

4.1.2.2 Statisztikai adatok

	Átlag fitnessz (Mean)	Legjobb fitnessz	Legrosszabb fitnessz	Fitnessz szórás (Standard Deviation)
AiNet	1,8E-08	1,61E-10	3,87E-07	4,76E-08
BFOA	1,891596	0,000102	9,902966	2,776526
BA	7E-12	2,17E-13	1,69E-11	3,55E-12
CA	0,641443	4,44E-16	10,22344	1,594372
DE	4,44E-16	4,44E-16	4,44E-16	0
HS	2,623005	0,01139	7,681816	1,807531
KH	3,56E-05	5,91E-07	8,87E-05	1,98E-05
MA	0,737611	3,05E-08	3,633828	1,174338
NM	6,89E-14	1,47E-14	4,41E-13	6,42E-14
PSO	4,44E-16	4,44E-16	4,44E-16	0
RS	1,405826	0,037104	2,898997	0,746866
SA	3,816731	8,32E-07	17,68951	4,201285

3. táblázat: Fitnessz értékek alakulása 500 iteráció után, 100 Monte Carlo futás adatai alapján (F1)

4.2 De Jong's function (F2)

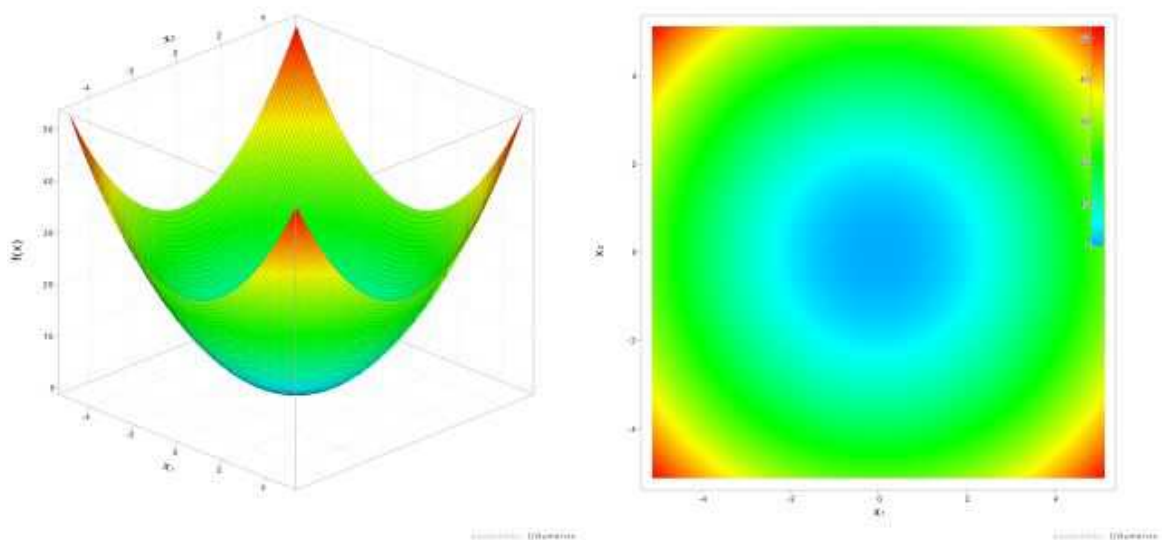
A De Jong's tesztfüggvény globális optimuma az $x_i = 0, i = 1, \dots, n$ pontban található, fitness értéke $f(x) = 0$. Az n változó a dimenziószámot jelöli, a dimenziók minimum és maximum korlátjára az ajánlott érték $-5,12 \leq x_i \leq 5,12, i = 1, \dots, n$. A függvényt az alábbi képlet írja le:

$$f(x) = \sum_{i=1}^n x_i^2 \quad (9)$$

A kétváltozós De Jong's tesztfüggvény háromdimenziós ábrázolása és szintértéke a 18. ábra látható.

4.2.1 C# forráskód

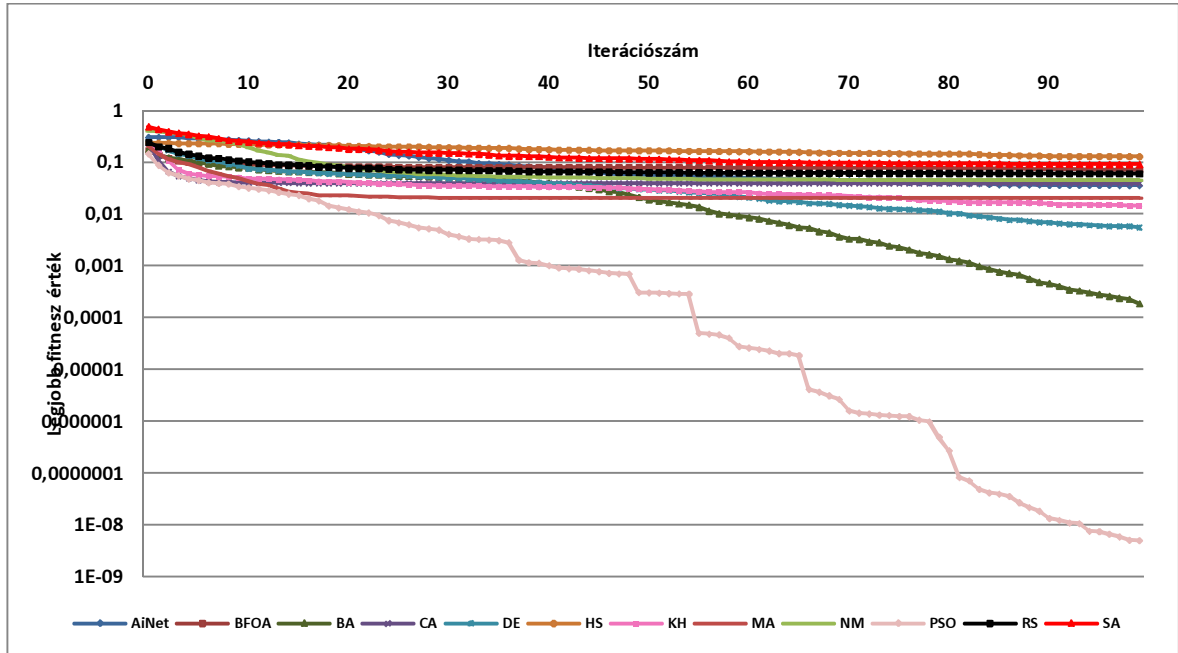
```
public double Evaluate(double[] coords)
{
    double fitness = Math.Pow(coords[0],2) + Math.Pow(coords[1],2);
    return fitness;
}
```



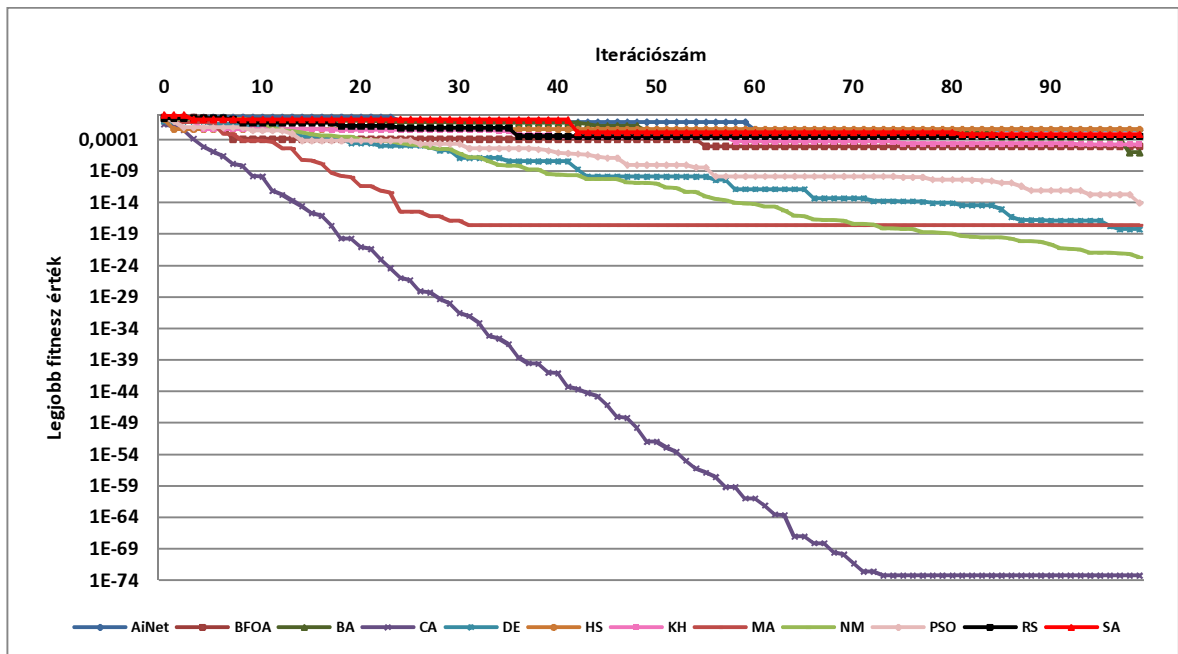
18. ábra: A kétváltozós De Jong's tesztfüggvény háromdimenziós ábrázolása és szintértéke

4.2.2 Mérési eredmények

4.2.2.1 Konvergencia grafikonok



19. ábra: Legjobb fitness értékek átlaga 100 Monte Carlo futás adatai alapján (F2)



20. ábra: Tényleges legjobb fitness érték 100 Monte Carlo futás adatai alapján (F2)

4.2.2.2 Statisztikai adatok

	Átlag fitnessz (Mean)	Legjobb fitnessz	Legrosszabb fitnessz	Fitnessz szórás (Standard Deviation)
AiNet	9,27E-08	3,9758E-10	5,02316E-07	1,04064E-07
BFOA	2,02E-08	1,1022E-10	9,41085E-08	1,90854E-08
BA	6,64E-24	1,9073E-26	7,87257E-23	9,56461E-24
CA	0,023136	5,9749E-74	0,7862181	0,10913929
DE	6,98E-78	2,9553E-82	1,90048E-76	2,66536E-77
HS	0,00215	3,7527E-07	0,034257127	0,005467524
KH	4,07E-07	1,3746E-09	2,56186E-06	4,24778E-07
MA	0,00107	2,8422E-18	0,039472039	0,004888765
NM	1,9E-110	1,292E-146	1,8223E-108	1,8222E-109
PSO	2,65E-48	6,3382E-51	9,43298E-47	1,08559E-47
RS	0,001426	7,8003E-08	0,006458675	0,001254512
SA	2,57E-05	2,3737E-14	0,002564939	0,000256493

4. táblázat: Fitnessz értékek alakulása 500 iteráció után, 100 Monte Carlo futás adatai alapján (F2)

4.3 Drop-Wave function (F3)

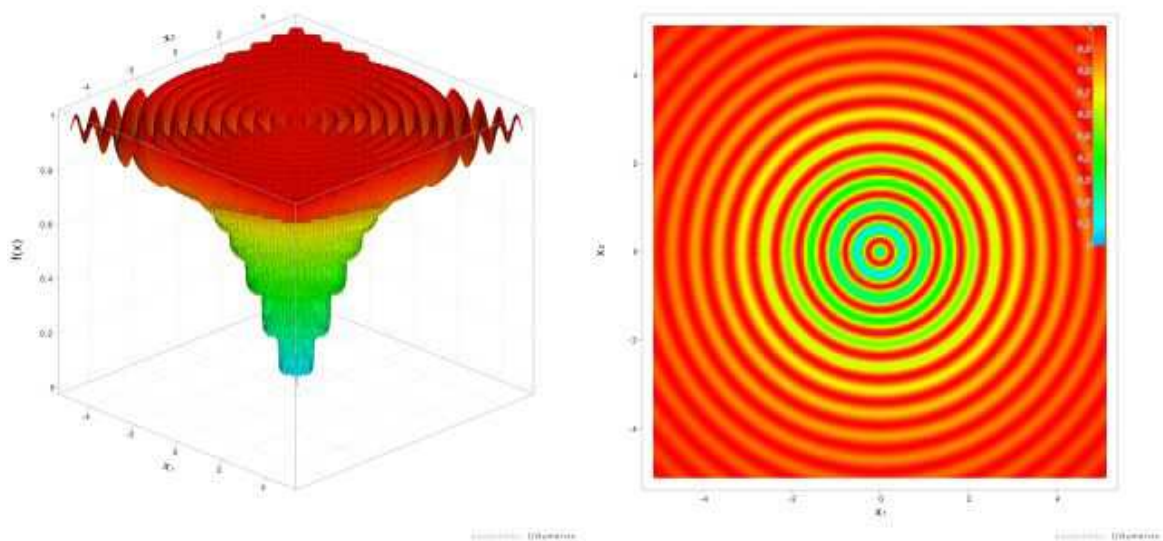
A Drop-Wave tesztfüggvény globális optima az $x_i = 0$, $i = 1, \dots, n$ pontban található, fitnessz értéke $f(x) = -1$. Az n változó a dimenziószámot jelöli, a dimenziók minimum és maximum korlátjára az ajánlott érték $-5,12 \leq x_i \leq 5,12$, $i = 1, \dots, n$. A függvény kétváltozós változatát az alábbi képlet írja le:

$$f(x_1, x_2) = -\frac{1 + \cos(12 \times \sqrt{x_1^2 + x_2^2})}{\frac{1}{2}(x_1^2 + x_2^2) + 2} \quad (10)$$

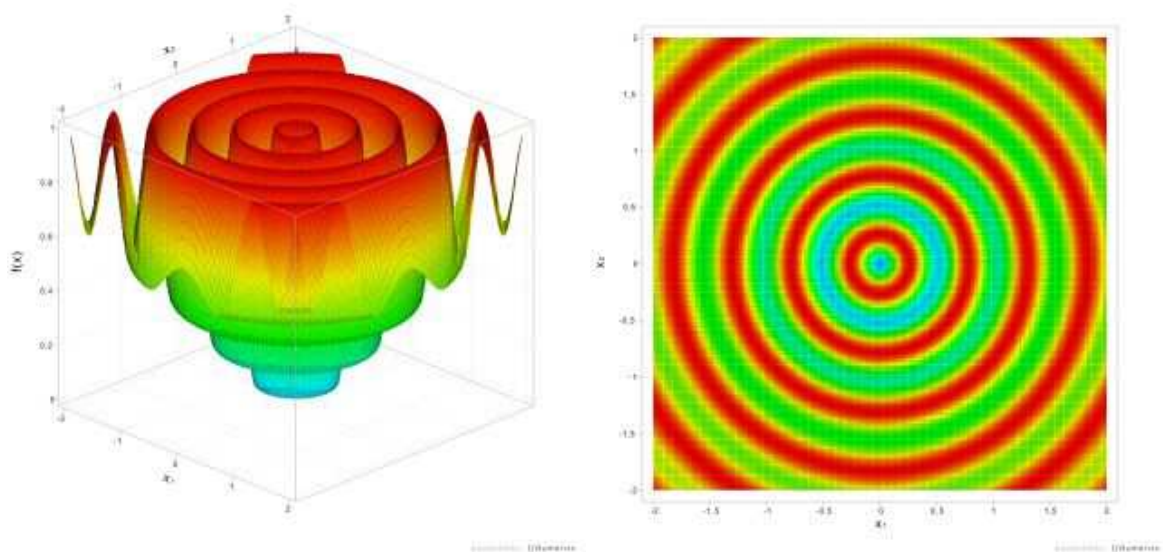
A függvény globális minimumának értéke eredetileg -1, a dolgozatomban azonban a Drop-Wave tesztfüggvényt úgy módosítottam, hogy az optimum pont fitnessz értéke 0 legyen, amit úgy értem el, hogy az eredeti értékekhez hozzáadtam 1-et. A kétváltozós Drop-Wave tesztfüggvény háromdimenziós ábrázolását és szintértképét a 21. ábra és a 22. ábra szemlélteti

4.3.1 C# forráskód

```
public double Evaluate(double[] coords)
{
    double s1 = 0, s2 = 0;
    s1 = 1 + Math.Cos(12 * Math.Sqrt(Math.Pow(coords[0], 2) + Math.Pow(coords[1], 2)));
    s2 = 0.5 * (Math.Pow(coords[0], 2) + Math.Pow(coords[1], 2)) + 2;
    double fitness = -s1 / s2;
    return fitness+1;
}
```



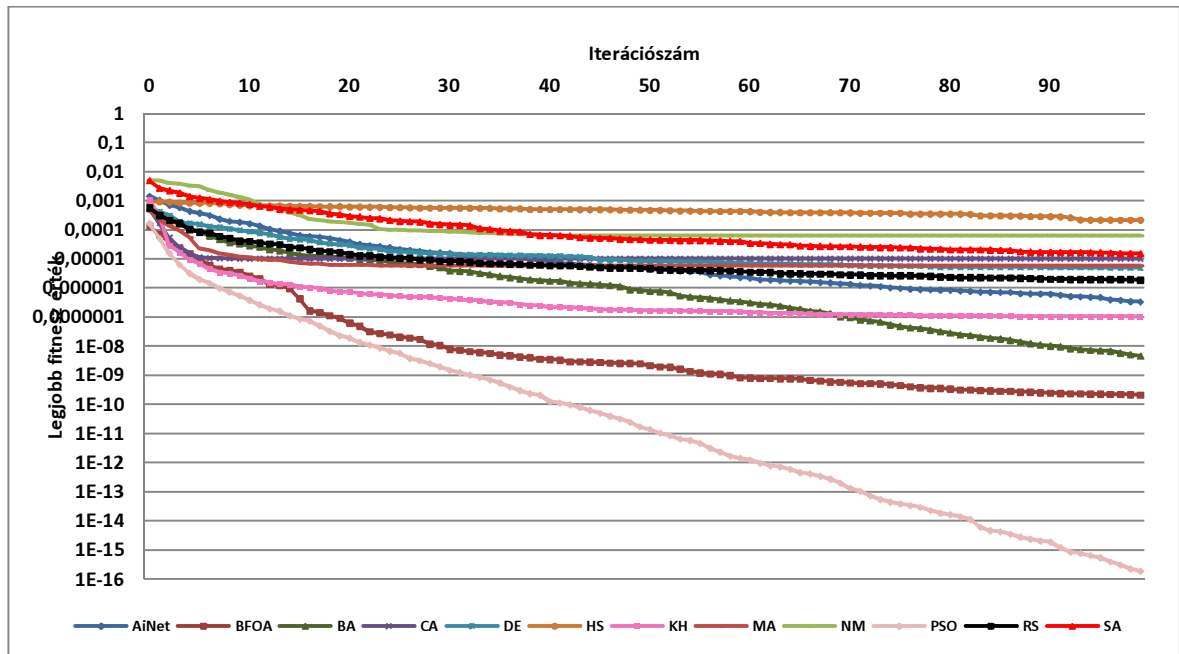
21. ábra: A kétváltozós Drop-Wave tesztfüggvény háromdimenziós ábrázolása és szintértéke, $-5, 12 \leq x_i \leq 5, 12$, $i = 1, \dots, n$



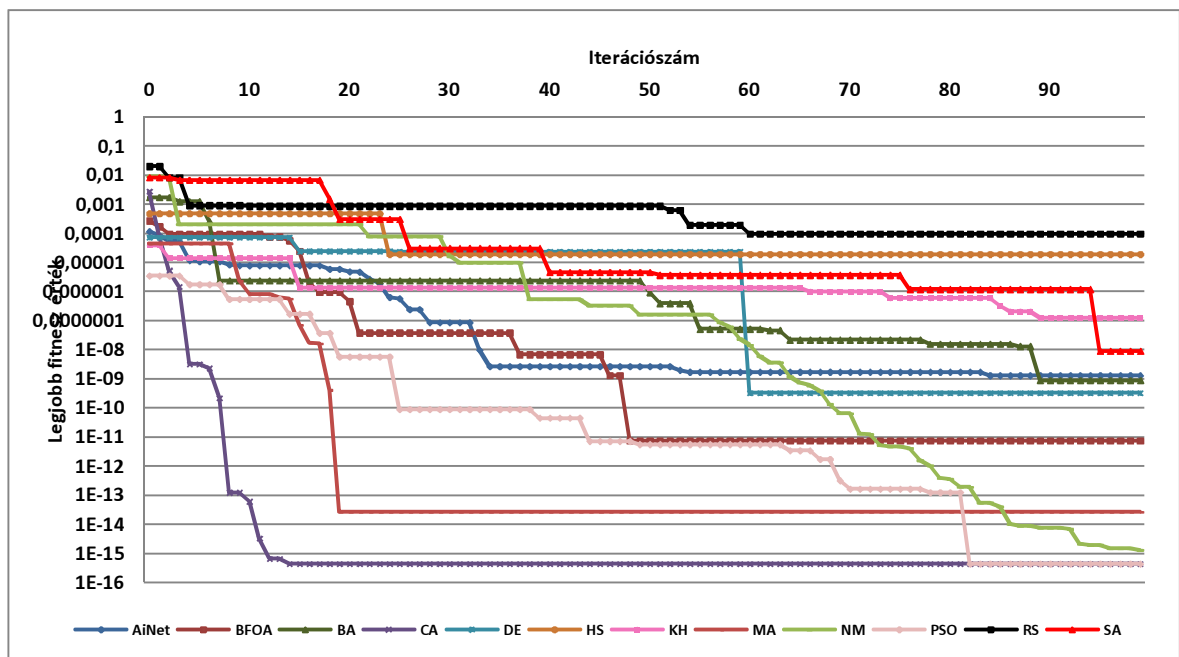
22. ábra: A kétváltozós Drop-Wave tesztfüggvény háromdimenziós ábrázolása és szintértéke, $-2 \leq x_i \leq 2$, $i = 1, \dots, n$

4.3.2 Mérési eredmények

4.3.2.1 Konvergencia grafikonok



23. ábra: Legjobb fitness értékek átlaga 100 Monte Carlo futás adatai alapján (F3)



24. ábra: Tényleges legjobb fitness érték 100 Monte Carlo futás adatai alapján (F3)

4.3.2.2 Statisztikai adatok

	Átlag fitnessz (Mean)	Legjobb fitnessz	Legrosszabb fitnessz	Fitnessz szórás (Standard Deviation)
AiNet	0,028052	4,44089E-16	0,063754672	0,031806419
BFOA	0,006904	1,54057E-07	0,063754672	0,017956139
BA	7,00E-12	2,1716E-13	1,69043E-11	3,54758E-12
CA	0,03924	0,00048295	0,095372838	0,027287962
DE	0,000133	6,26789E-07	0,001267426	0,000194716
HS	0,062025	1,14584E-05	0,21425571	0,034522777
KH	7,53E-06	4,27841E-08	0,00038778	3,96694E-05
MA	0,02028	1,11022E-16	0,254657721	0,039995144
NM	0,044858	1,11022E-16	0,214248665	0,034378982
PSO	4,44E-16	4,44089E-16	4,44089E-16	0
RS	0,034242	9,7296E-05	0,063975186	0,023124933
SA	0,087906	1,63314E-13	0,231241885	0,055169549

5. táblázat: Fitnessz értékek alakulása 500 iteráció után, 100 Monte Carlo futás adatai alapján (F3)

4.4 Easom's function (F4)

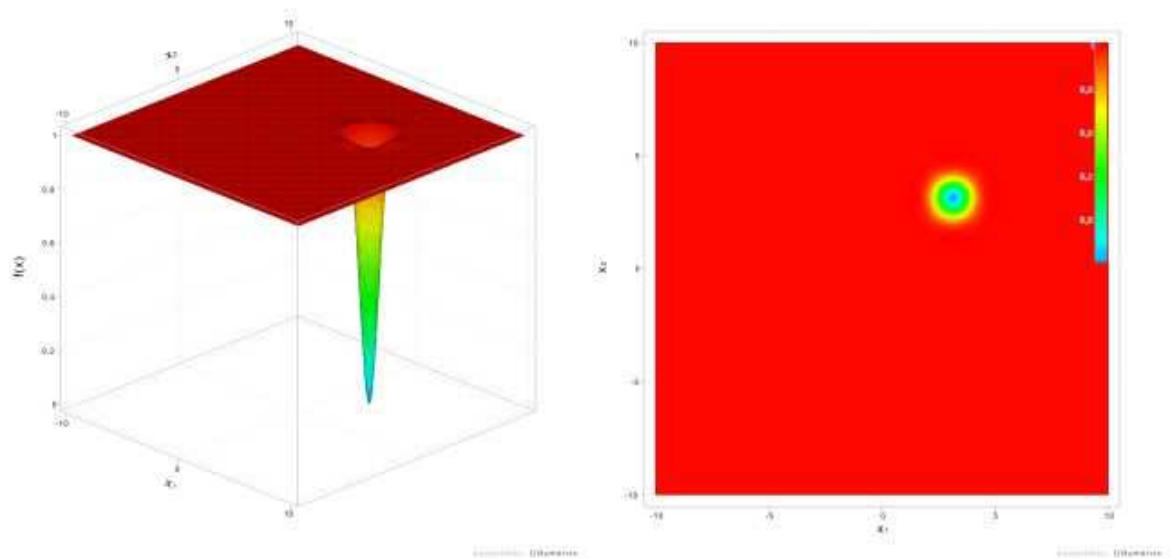
Az Easom's tesztfüggvény globális optimuma az $x_i = \pi$, $i = 1, \dots, n$ pontban található, fitnessz értéke $f(x) = -1$. Az n változó a dimenziószámot jelöli, a dimenziók minimum és maximum korlátjára az ajánlott érték $-10 \leq x_i \leq 10$, $i = 1, \dots, n$. A függvény kétváltozós változatát az alábbi képlet írja le:

$$f(x_1, x_2) = -\cos(x_1) \times \cos(x_2) \times \exp(-(x_1 - \pi)^2 - (x_2 - \pi)^2) \quad (11)$$

A függvény globális minimumának értéke eredetileg -1, a dolgozatomban azonban az Easom tesztfüggvényt úgy módosítottam, hogy az optimum pont fitnessz értéke 0 legyen, amit úgy értem el, hogy az eredeti értékekhez hozzáadtam 1-et. A kétváltozós Easom's tesztfüggvény háromdimenziós ábrázolása és szintértképe a 24. ábra látható.

4.4.1 C# forráskód

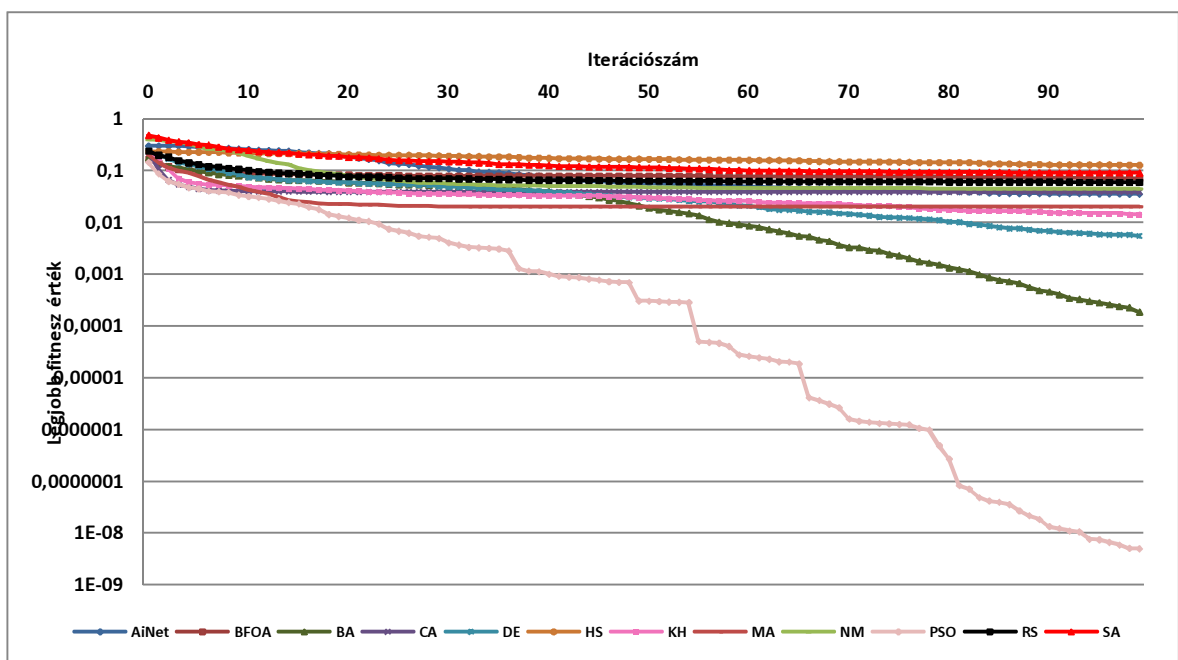
```
public double Evaluate(double[] coords)
{
    double fitness = -Math.Cos(coords[0]) * Math.Cos(coords[1]) * Math.Exp(-
Math.Pow((coords[0] - Math.PI), 2) - Math.Pow((coords[1] - Math.PI), 2));
    return -fitness-1;
}
```



25. ábra: A kétváltozós Easom's tesztfüggvény háromdimenziós ábrázolása és szintérképe

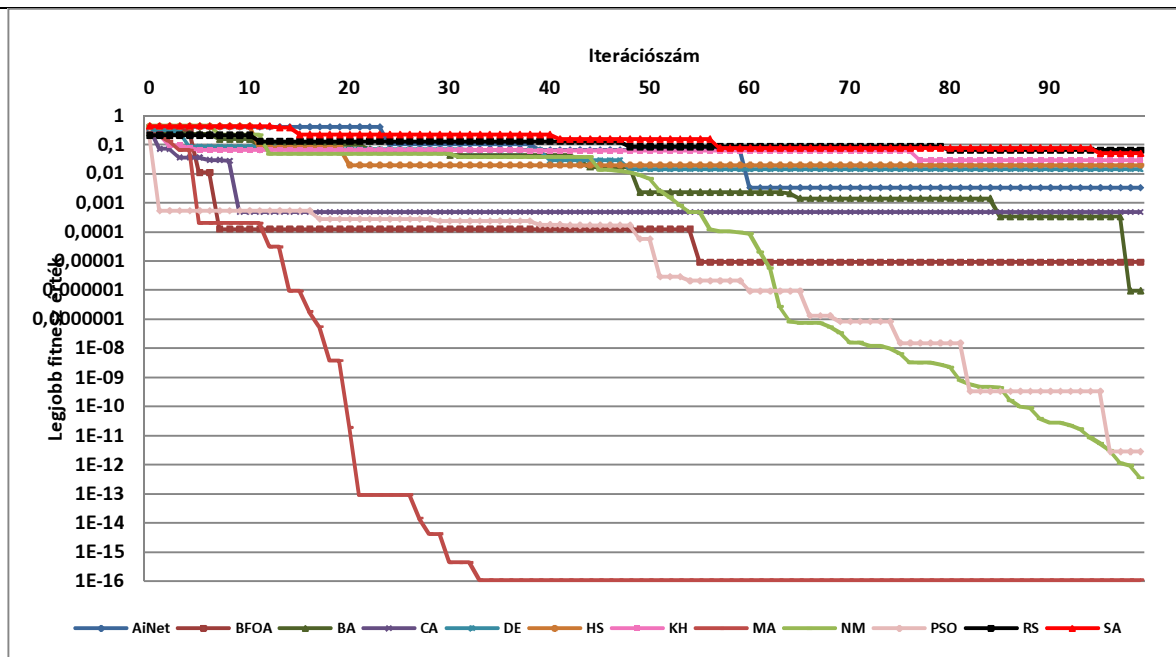
4.4.2 Mérési eredmények

4.4.2.1 Konvergencia grafikonok



26. ábra: Legjobb fitness értékek átlaga 100 Monte Carlo futás adatai alapján (F4)

Metaheurisztikus algoritmusok hatékonyságvizsgálata saját keretrendszer által létrehozott komplex tesztfüggvények segítségével, az eredmények felhasználása futódaru főtartó optimalásához



27. ábra: Tényleges legjobb fitness érték 100 Monte Carlo futás adatai alapján (F4)

4.4.2.2 Statisztikai adatok

	Átlag fitness (Mean)	Legjobb fitness	Legrosszabb fitness	Fitness szórás (Standard Deviation)
AiNet	0,338572	4,32987E-15	0,946645091	0,273240013
BFOA	3,53E-08	4,73221E-10	1,70976E-07	3,56354E-08
BA	0,008362	0,000153783	0,037693114	0,007807324
CA	0,047606	4,44E-16	0,973596888	0,160472552
DE	3,1E-09	4,44E-16	2,57043E-07	2,61292E-08
HS	0,351185	2,07161E-05	0,999921906	0,338724548
KH	0,079998	3,48994E-11	0,999999962	0,272653273
MA	0,23503	0,000412915	0,999920838	0,284305599
NM	0,790139	1,11022E-16	1	0,339287667
PSO	4,44E-16	4,44E-16	4,44E-16	0
RS	0,00732	5,40134E-05	0,038722312	0,006834043
SA	0,195038	1,08913E-13	0,997571531	0,284518899

6. táblázat: Fitness értékek alakulása 500 iteráció után, 100 Monte Carlo futás adatai alapján (F4)

4.5 Griewangk's function (F5)

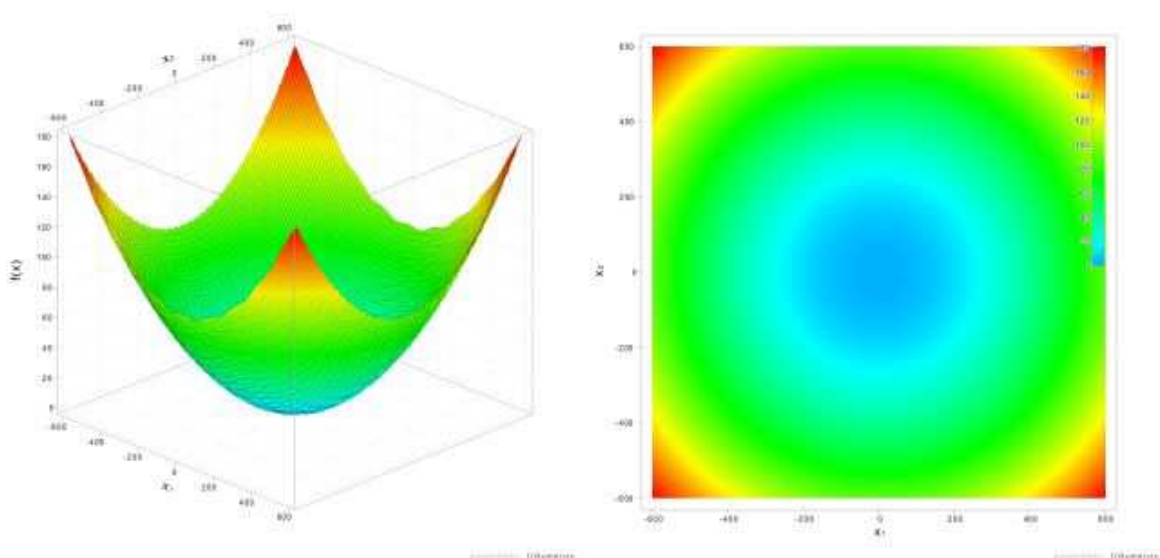
A Griewangk's tesztfüggvény globális optimuma az $x_i = 0, i = 1, \dots, n$ pontban található, fitness értéke $f(x) = 0$. Az n változó a dimenziószámot jelöli, a dimenziók minimum és maximum korlátjára az ajánlott érték $-600 \leq x_i \leq 600, i = 1, \dots, n$. A függvényt az alábbi képlet írja le:

$$f(x) = \frac{1}{4000} \times \sum_{i=1}^n x_i^2 - \prod_{i=1}^n \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1 \quad (12)$$

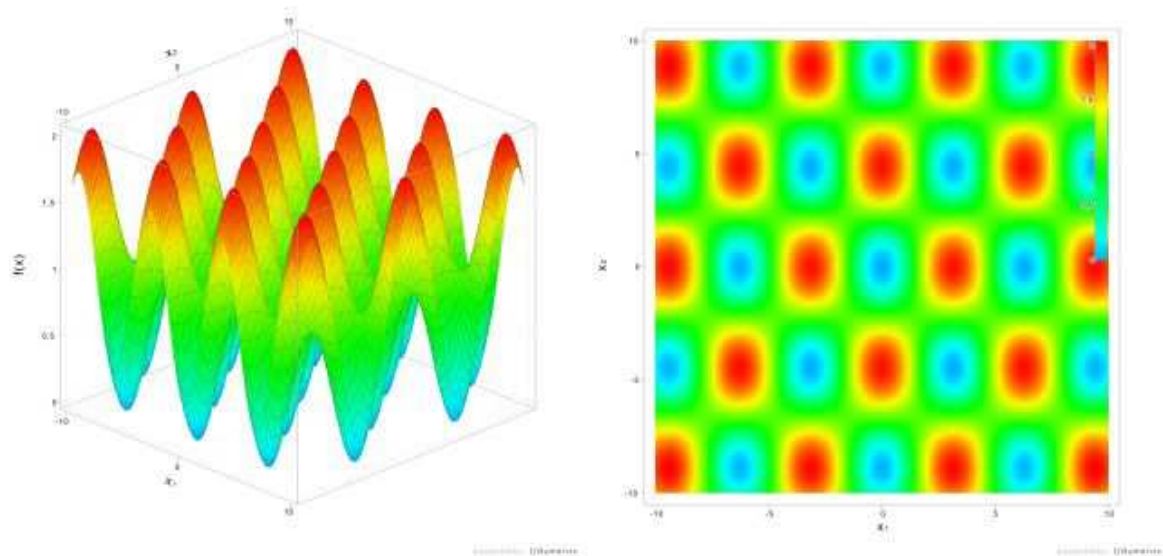
A kétváltozós Griewangk's tesztfüggvény háromdimenziós ábrázolása és szintérképe a 28. ábraán és a 29. ábra látható.

4.5.1 C# forráskód

```
public double Evaluate(double[] coords)
{
    int n = 2;
    double fr = 4000;
    double s = 0;
    double p = 1.0;
    for (int i = 1; i <= n; i++) {
        s = s + Math.Pow(coords[i-1], 2);
        p = p * Math.Cos( coords[i-1]/Math.Sqrt(i));
    }
    double fitness = s/fr-p+1;
    return fitness;
}
```



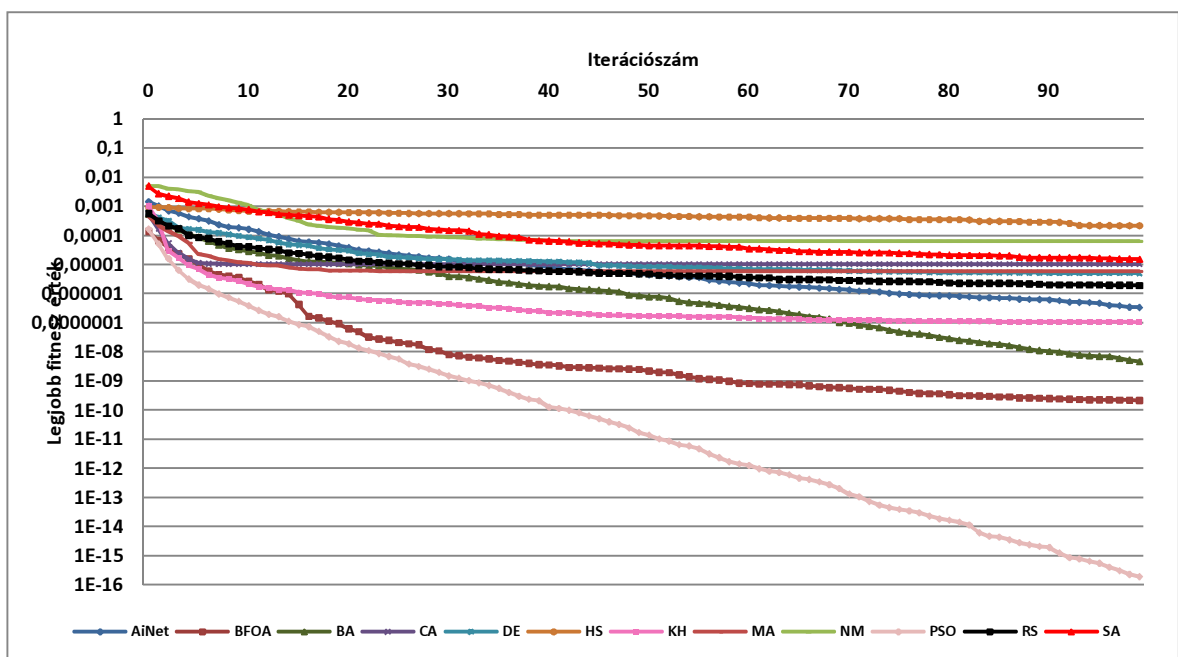
28. ábra: A kétváltozós Griewangk's tesztfüggvény háromdimenziós ábrázolása és szintérképe, $-600 \leq x_i \leq 600, i = 1, \dots, n$



29. ábra: A kétváltozós Griewangk's tesztfüggvény háromdimenziós ábrázolása és szintértéke, $-10 \leq x_i \leq 10$, $i = 1, \dots, n$

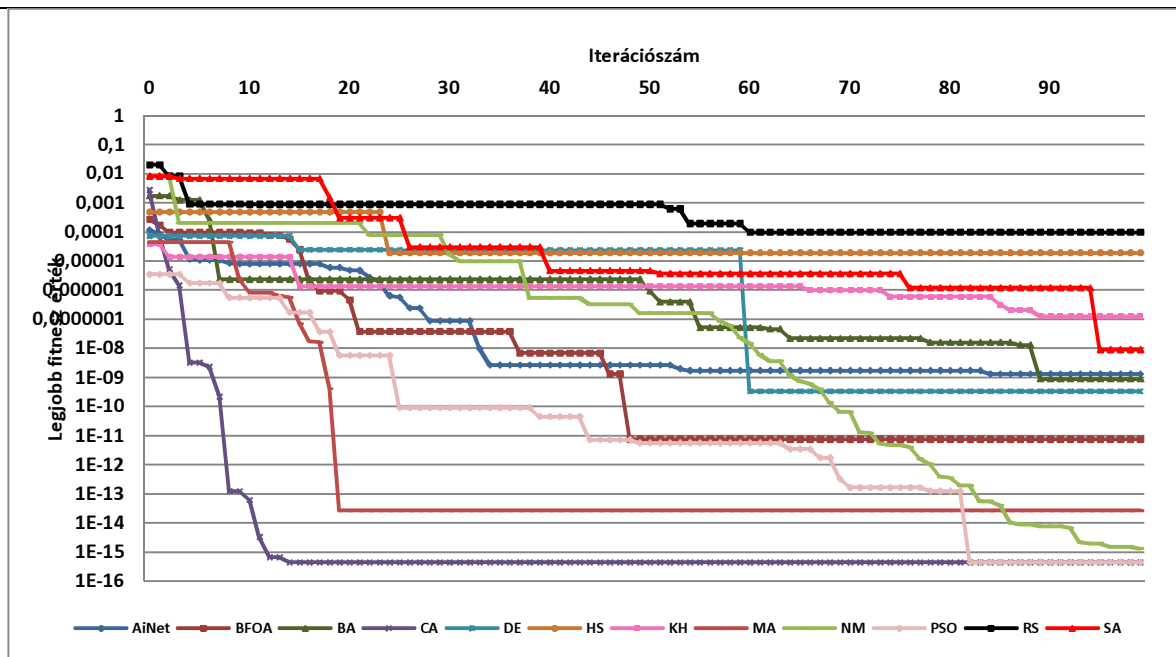
4.5.2 Mérési eredmények

4.5.2.1 Konvergencia grafikonok



30. ábra: Legjobb fitness értékek átlaga 100 Monte Carlo futás adatai alapján (F5)

Metaheurisztikus algoritmusok hatékonyságvizsgálata saját keretrendszer által létrehozott komplex tesztfüggvények segítségével, az eredmények felhasználása futódaru főtartó optimalálásához



31. ábra: Tényleges legjobb fitnessz érték 100 Monte Carlo futás adatai alapján (F5)

4.5.2.2 Statisztikai adatok

	Átlag fitnessz (Mean)	Legjobb fitnessz	Legrosszabb fitnessz	Fitnessz szórás (Standard Deviation)
AiNet	0,185132	0,001688447	0,663255318	0,133070054
BFOA	0,947568	0,007396041	5,857943399	0,972015632
BA	4,44E-16	4,44E-16	4,44E-16	0
CA	0,086298	0,000698935	0,875463838	0,16001206
DE	0,000148	4,44089E-16	0,00739604	0,001040662
HS	0,484628	0,019770559	1,381682555	0,313480954
KH	0,003428	7,08549E-08	0,010152165	0,003691122
MA	0,044128	1,46549E-14	1,452393845	0,166268017
NM	0,035997	1,11022E-16	0,184901002	0,041310122
PSO	0,000195	4,44E-16	0,00739604	0,001135971
RS	0,136252	0,024694543	0,315601608	0,068650969
SA	6,755522	0,031042636	33,46874207	7,787806018

7. táblázat: Fitnessz értékek alakulása 500 iteráció után, 100 Monte Carlo futás adatai alapján (F5)

4.6 Matyas's function (F6)

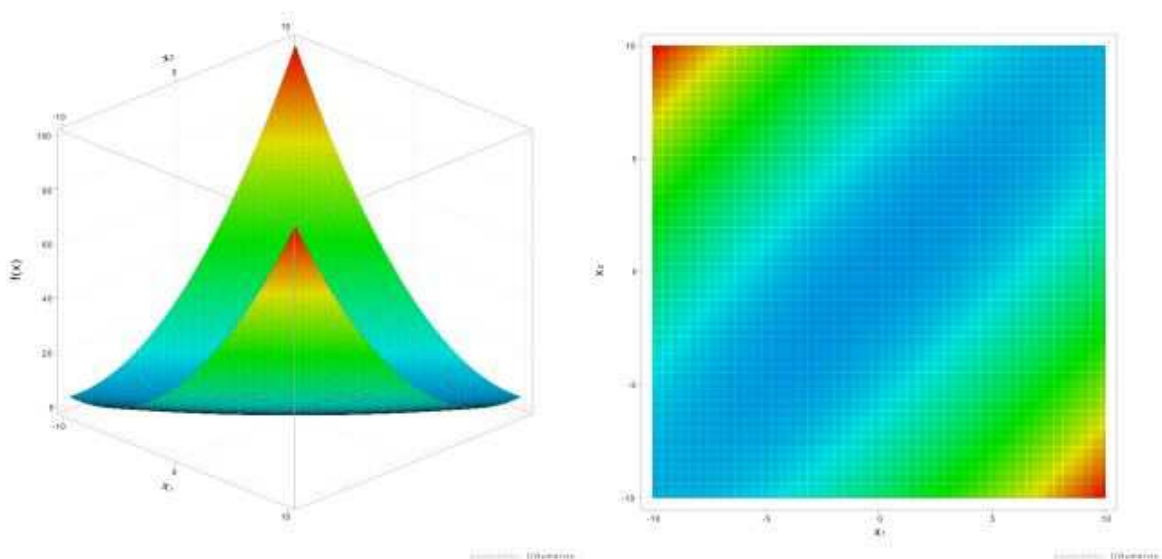
A Matyas's tesztfüggvény globális optimuma az $x_i = 0, i = 1, \dots, n$ pontban található, fitness értéke $f(x) = 0$. Az n változó a dimenziószámot jelöli, a dimenziók minimum és maximum korlátjára az ajánlott érték $-10 \leq x_i \leq 10, i = 1, \dots, n$. A függvény kétváltozós változatát az alábbi képlet írja le:

$$f(x_1, x_2) = 0,26 \times (x_1^2 + x_2^2) - 0,48 \times x_1 \times x_2 \quad (13)$$

A kétváltozós Matyas's tesztfüggvény háromdimenziós ábrázolása és szintértéke a 32. ábra látható.

4.6.1 C# forráskód

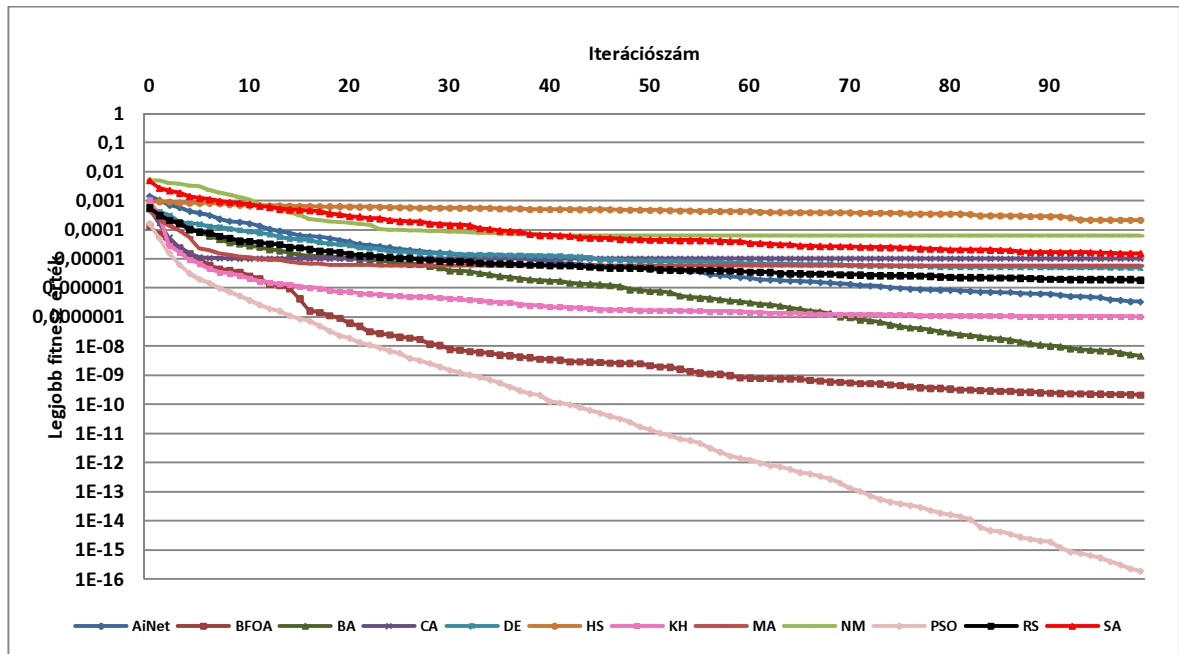
```
public double Evaluate(double[] coords)
{
    int n = 2;
    double fr = 4000;
    double s = 0;
    double p = 1.0;
    for (int i = 1; i <= n; i++)
    {
        s = s + Math.Pow(coords[i-1], 2);
        p = p * Math.Cos( coords[i-1]/Math.Sqrt(i));
    }
    double fitness = s/fr-p+1;
    return fitness;
}
```



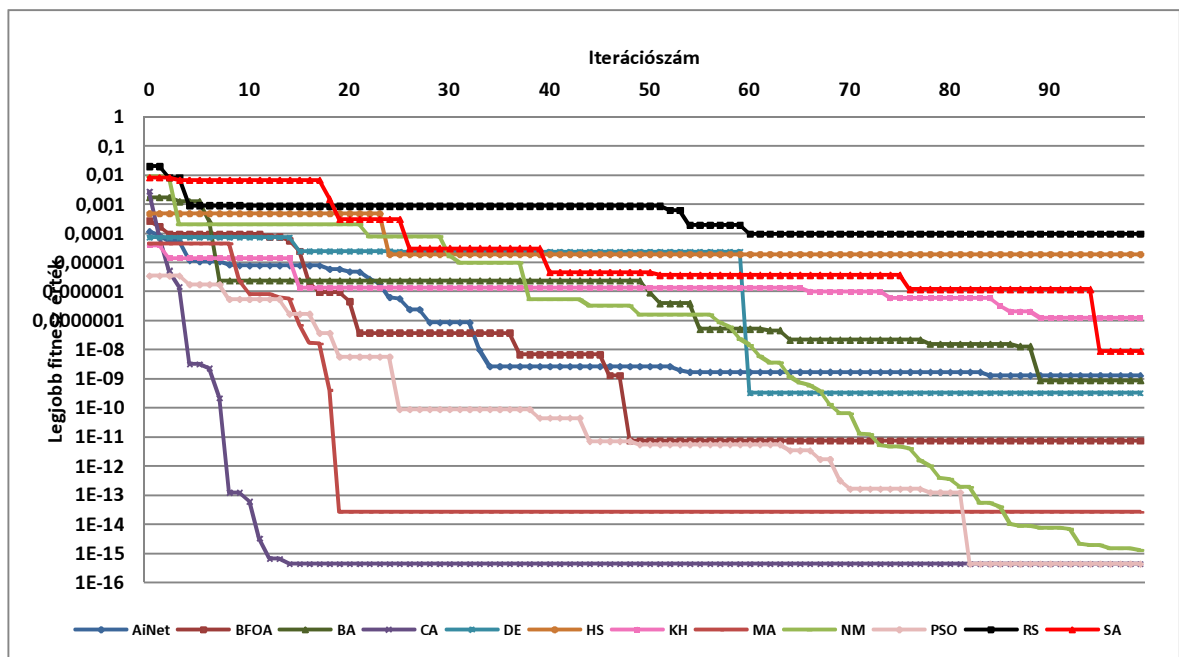
32. ábra: A kétváltozós Matyas's tesztfüggvény háromdimenziós ábrázolása és szintértéke

4.6.2 Mérési eredmények

4.6.2.1 Konvergencia grafikonok



33. ábra: Legjobb fitness értékek átlaga 100 Monte Carlo futás adatai alapján (F6)



34. ábra: Tényleges legjobb fitness érték 100 Monte Carlo futás adatai alapján (F6)

4.6.2.2 Statisztikai adatok

	Átlag fitnessz (Mean)	Legjobb fitnessz	Legrosszabb fitnessz	Fitnessz szórás (Standard Deviation)
AiNet	2,02E-06	6,63695E-10	0,000121831	1,33893E-05
BFOA	2,9E-09	1,2489E-11	1,62785E-08	3,00468E-09
BA	7,32E-25	5,43892E-27	3,64498E-24	7,49404E-25
CA	0,003758	9,34628E-38	0,161422695	0,017754712
DE	0,000396	1,26367E-07	0,007386929	0,000877501
HS	0,022756	1,74517E-06	0,259872516	0,036619625
KH	7,37E-05	3,43738E-08	0,002557653	0,000298693
MA	0,006888	2,1684E-19	0,14039499	0,018993333
NM	5,4E-111	5,3196E-149	4,7925E-109	4,8044E-110
PSO	8,08E-45	2,49604E-48	2,25383E-43	2,50697E-44
RS	0,000421	6,13172E-06	0,002367232	0,000378703
SA	0,003422	7,91346E-15	0,082585165	0,011858499

8. táblázat: Fitnessz értékek alakulása 500 iteráció után, 100 Monte Carlo futás adatai alapján (F6)

4.7 Rastrigin's function (F7)

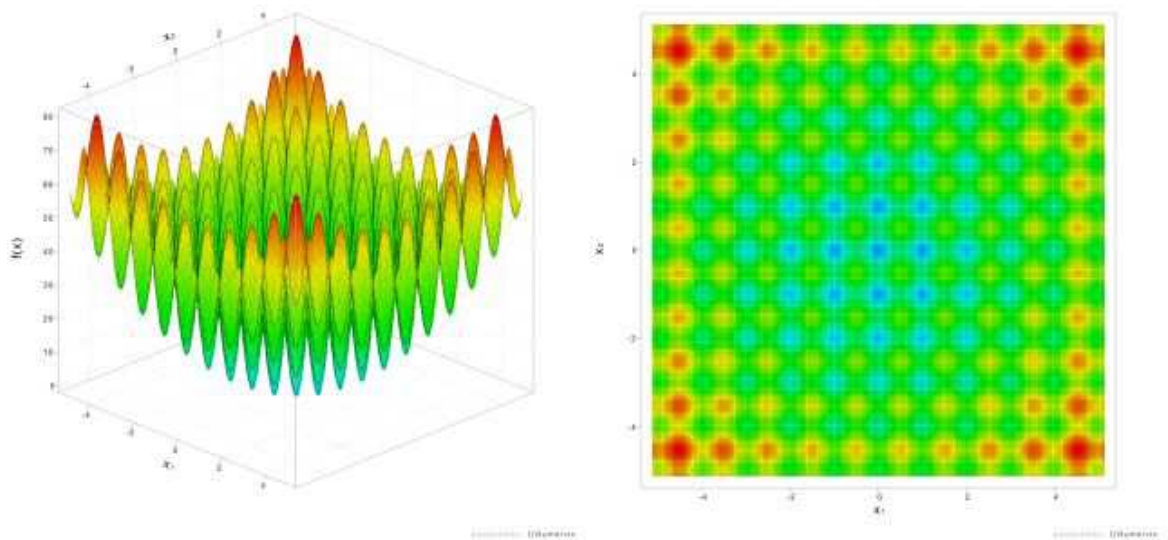
A Rastrigin's tesztfüggvény globális optimuma az $x_i = 0$, $i = 1, \dots, n$ pontban található, fitnessz értéke $f(x) = 0$. Az n változó a dimenziószámot jelöli, a dimenziók minimum és maximum korlátjára az ajánlott érték $-5,12 \leq x_i \leq 5,12$, $i = 1, \dots, n$. A függvényt az alábbi képlet írja le:

$$f(x) = 10 \times n + \sum_{i=1}^n [x_i^2 - 10 \times \cos(2\pi x_i)] \quad (14)$$

A kétváltozós Rastrigin's tesztfüggvény háromdimenziós ábrázolása és szintértékepe a 35. ábra látható.

4.7.1 C# forráskód

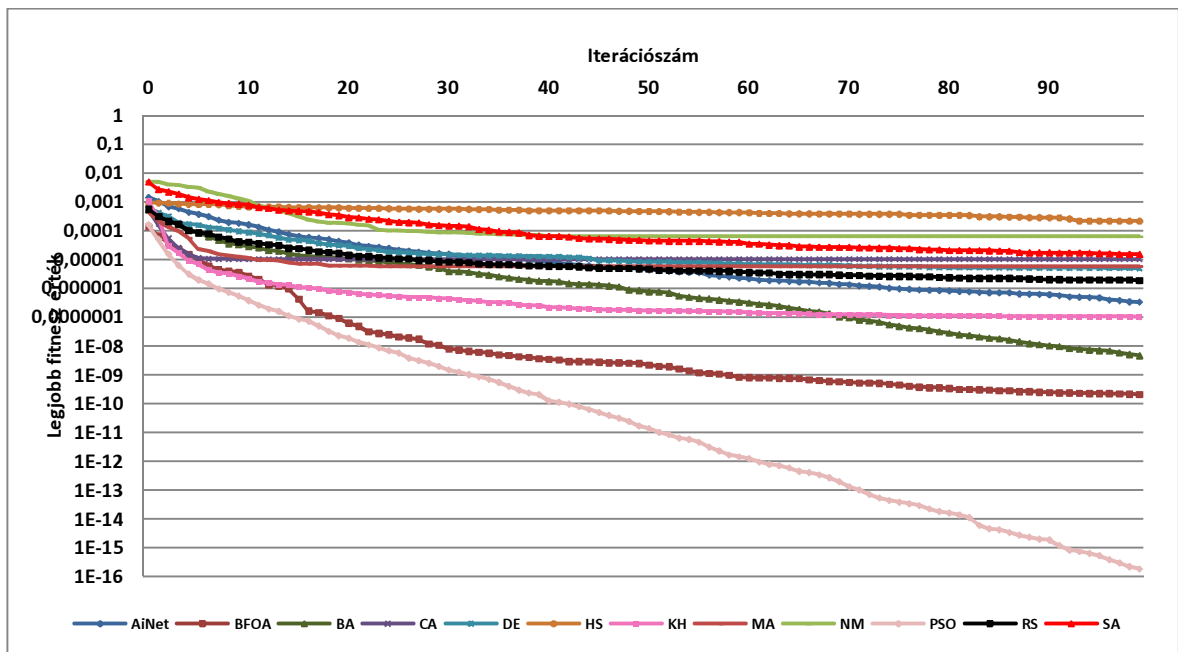
```
public double Evaluate(double[] coords)
{
    double fitness = 10 * 2 + (Math.Pow(coords[0], 2) - 10 * Math.Cos(2 * Math.PI *
coords[0])) + (Math.Pow(coords[1], 2) - 10 * Math.Cos(2 * Math.PI * coords[1]));
    return fitness;
}
```



35. ábra: A kétváltozós Rastrigin's tesztfüggvény háromdimenziós ábrázolása és szintértéke

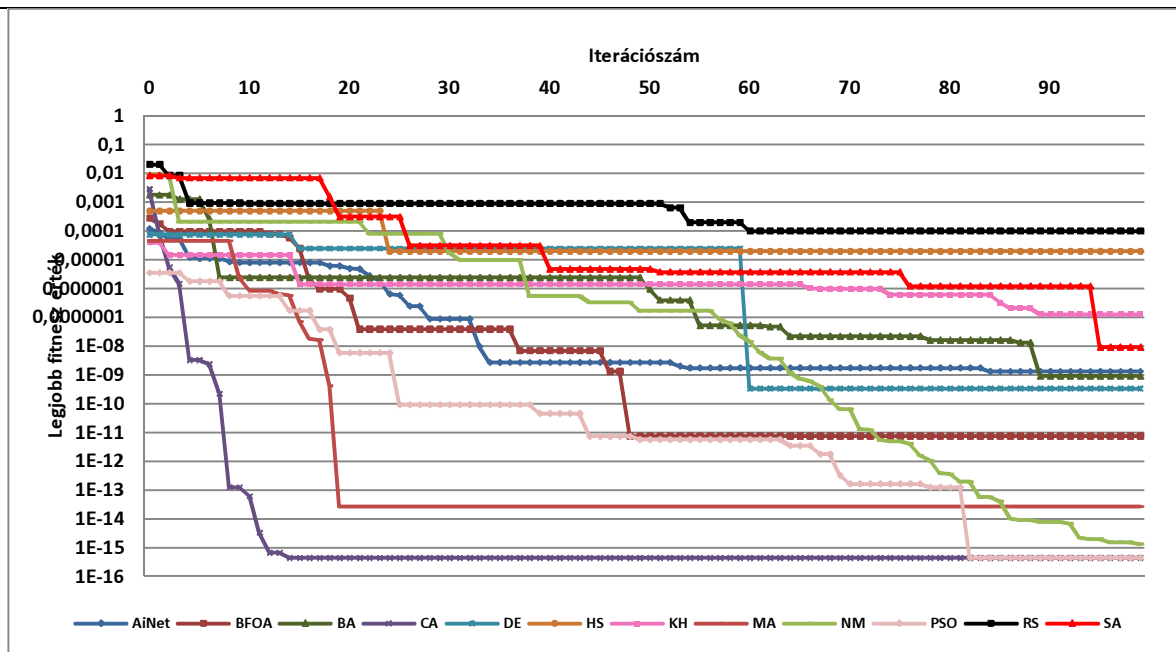
4.7.2 Mérési eredmények

4.7.2.1 Konvergencia grafikonok



36. ábra: Legjobb fitness értékek átlaga 100 Monte Carlo futás adatai alapján (F7)

Metaheurisztikus algoritmusok hatékonyságvizsgálata saját keretrendszer által létrehozott komplex tesztfüggvények segítségével, az eredmények felhasználása futódaru főtartó optimalálásához



37. ábra: Tényleges legjobb fitness érték 100 Monte Carlo futás adatai alapján (F7)

4.7.2.2 Statisztikai adatok

	Átlag fitness (Mean)	Legjobb fitness	Legrosszabb fitness	Fitness szórás (Standard Deviation)
AiNet	1,52E-05	5,29173E-08	0,000115135	1,7123E-05
BFOA	0,12446	2,56332E-07	0,994985571	0,326347226
BA	4,44E-16	4,44E-16	4,44E-16	0
CA	0,926519	2,4869E-14	4,839347915	1,190445873
DE	4,44E-16	4,44E-16	4,44E-16	0
HS	0,305383	0,000116687	1,990157355	0,486489021
KH	0,000194	4,96276E-09	0,009812024	0,001361129
MA	0,254456	4,44E-16	2,33520454	0,477743198
NM	0,945211	1,77636E-15	4,974790248	0,931369628
PSO	4,44E-16	4,44E-16	4,44E-16	0
RS	0,304903	0,001513642	1,080883251	0,286530854
SA	0,451692	5,97389E-12	1,989918114	0,515544514

9. táblázat: Fitness értékek alakulása 500 iteráció után, 100 Monte Carlo futás adatai alapján (F7)

4.8 Rosenbrock's valley (F8)

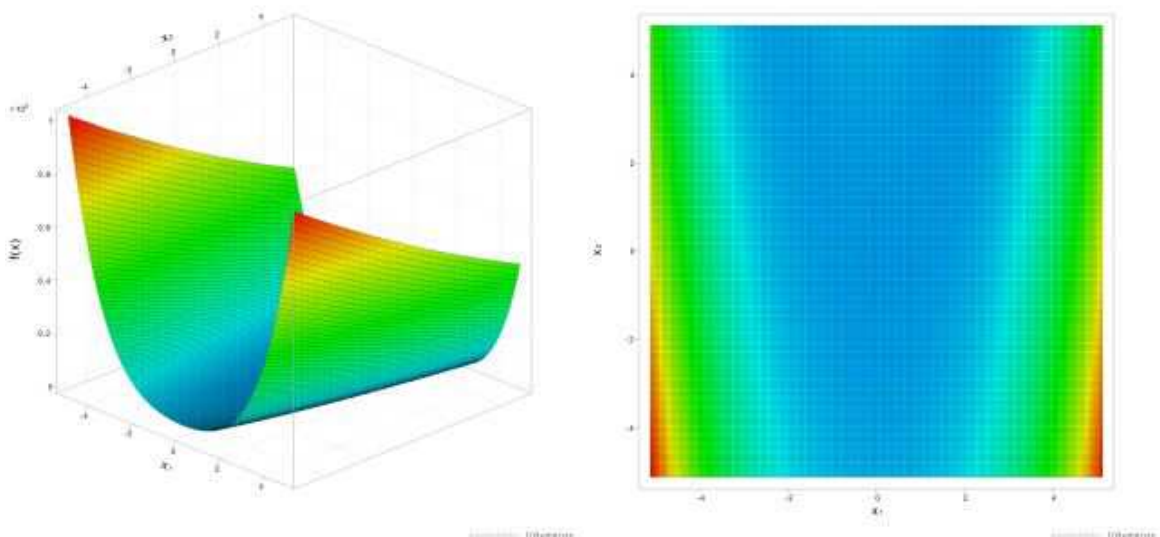
A Rosenbrock's valley tesztfüggvény globális optima az $x_i = 1, i = 1, \dots, n$ pontban található, fitness értéke $f(x) = 0$. Az n változó a dimenziószámot jelöli, a dimenziók minimum és maximum korlátjára az ajánlott érték $-2,048 \leq x_i \leq 2,048, i = 1, \dots, n$. A függvényt az alábbi képlet írja le:

$$f(x) = \sum_{i=1}^{n-1} [100 \times (x_{i+1} - x_i^2)^2 + (1 - x_i)^2] \quad (15)$$

A kétváltozós Rosenbrock's valley tesztfüggvény háromdimenziós ábrázolása és szintérképe a 38. ábra látható.

4.8.1 C# forráskód

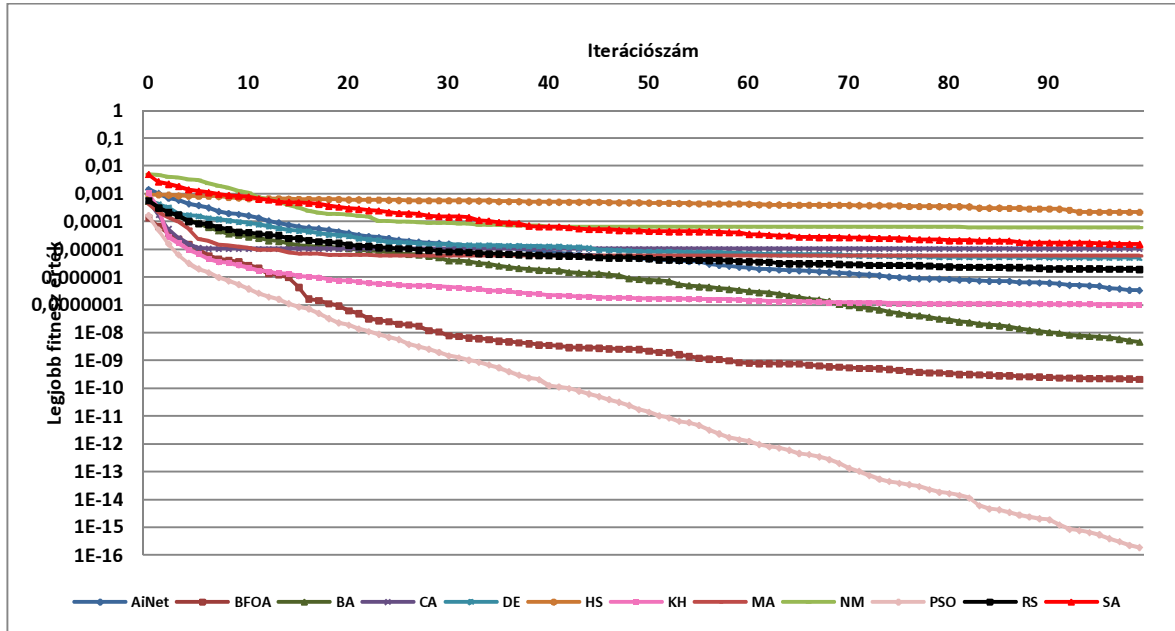
```
public double Evaluate(double[] coords)
{
    double fitness = 100 * (Math.Pow((coords[1] - Math.Pow(coords[0], 2)), 2) +
    Math.Pow(1 - coords[0], 2));
    return fitness;
}
```



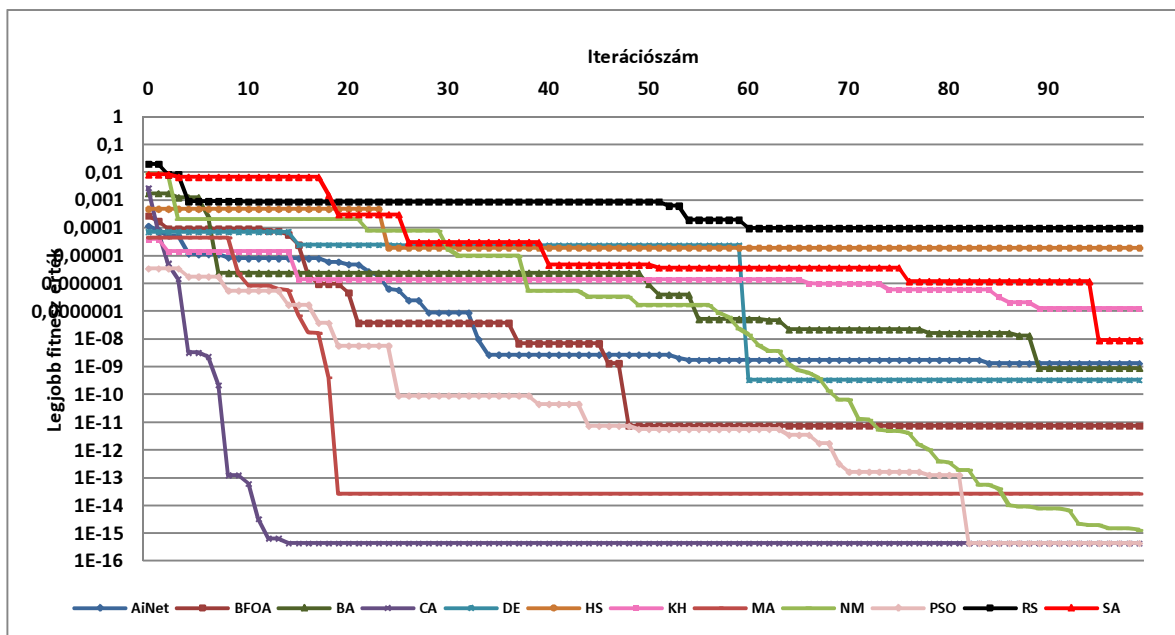
38. ábra: A kétváltozós Rosenbrock's valley tesztfüggvény háromdimenziós ábrázolása és szintérképe

4.8.2 Mérési eredmények

4.8.2.1 Konvergencia grafikonok



39. ábra: Legjobb fitness értékek átlaga 100 Monte Carlo futás adatai alapján (F8)



40. ábra: Tényleges legjobb fitness érték 100 Monte Carlo futás adatai alapján (F8)

4.8.2.2 Statisztikai adatok

	Átlag fitnessz (Mean)	Legjobb fitnessz	Legrosszabb fitnessz	Fitnessz szórás (Standard Deviation)
AiNet	2,22542E-05	3,84712E-08	0,000185621	2,89046E-05
BFOA	3,15616E-06	3,10962E-08	1,60929E-05	3,53035E-06
BA	0,123897477	0,000328027	0,581609638	0,115787504
CA	0,685049334	1,2326E-30	21,12260643	2,765763702
DE	0,009326976	2,43832E-06	0,149061247	0,019412511
HS	1,259331516	0,00020019	9,030061181	2,031430484
KH	0,03099418	8,61695E-07	0,332197409	0,057486674
MA	8,372952836	0,001709591	100,0000002	14,63558992
NM	1,09085E-28	4,93038E-30	2,46519E-27	2,80614E-28
PSO	2,43491E-11	2,09541E-27	2,19399E-09	2,20152E-10
RS	0,12307684	0,000502781	0,781231895	0,126339764
SA	4,85649E-11	5,14593E-14	2,25836E-10	4,71631E-11

10. táblázat: Fitnessz értékek alakulása 500 iteráció után, 100 Monte Carlo futás adatai alapján (F8)

4.9 Schaffer's N. 2. function (F9)

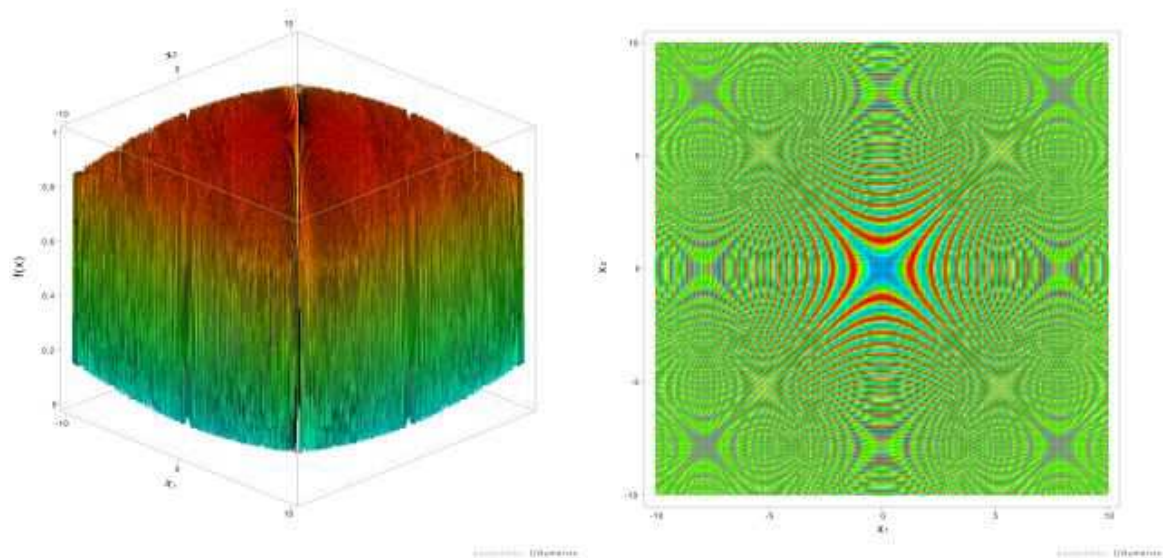
A Schaffer's N. 2. tesztfüggvény globális optimuma az $x_i = 0$, $i = 1, \dots, n$ pontban található, fitnessz értéke $f(x) = 0$. Az n változó a dimenziószámot jelöli, a dimenziók minimum és maximum korlátjára az ajánlott érték $-10 \leq x_i \leq 10$, $i = 1, \dots, n$. A függvény kétváltozós változatát az alábbi képlet írja le:

$$f(x_1, x_2) = 0,5 + \frac{\sin^2(x_1^2 - x_2^2) - 0,5}{[1 + 0,001 \times (x_1^2 + x_2^2)]^2} \quad (16)$$

A kétváltozós Schaffer's N. 2. tesztfüggvény háromdimenziós ábrázolása és szintértékepe a 41. ábra látható.

4.9.1 C# forráskód

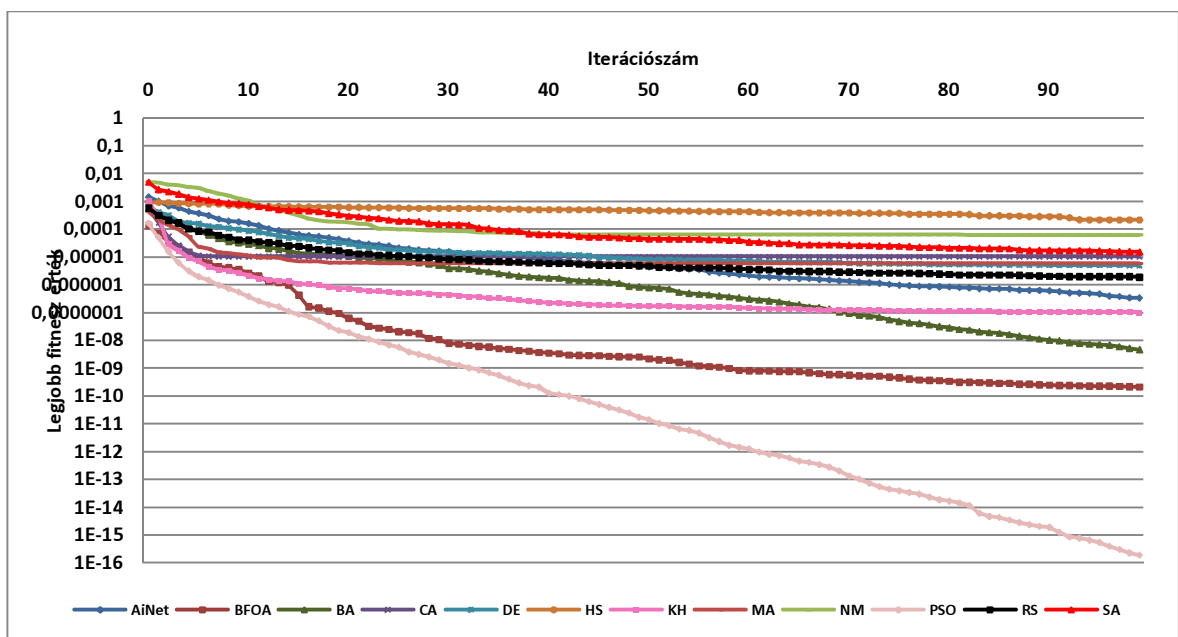
```
public double Evaluate(double[] coords)
{
    double s1 = 0;
    double s2 = 0;
    s1 = Math.Pow(Math.Sin(Math.Pow(coords[0], 2) - Math.Pow(coords[1], 2)), 2) - 0.5;
    s2 = Math.Pow((1 + 0.001 * (Math.Pow(coords[0], 2) + Math.Pow(coords[1], 2))), 2);
    double fitness = 0.5 + s1 / s2;
    return fitness;
}
```



41. ábra: A kétváltozós Schaffer's N. 2. tesztfüggvény háromdimenziós ábrázolása és szintértéke

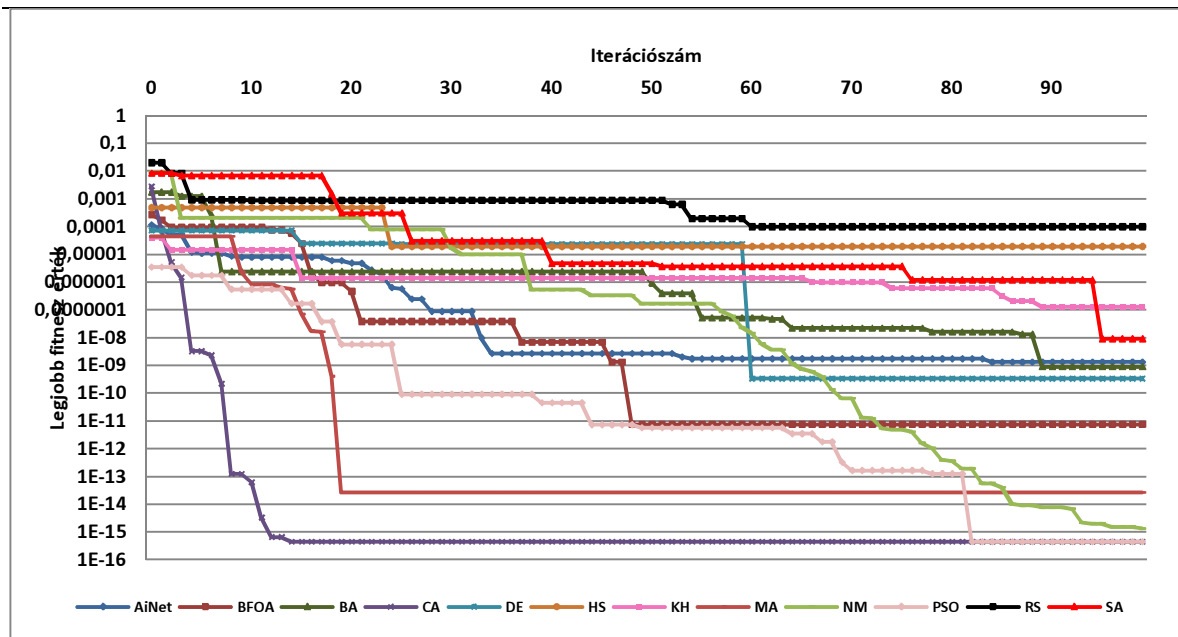
4.9.2 Mérési eredmények

4.9.2.1 Konvergencia grafikonok



42. ábra: Legjobb fitness értékek átlaga 100 Monte Carlo futás adatai alapján (F9)

Metaheurisztikus algoritmusok hatékonyságvizsgálata saját keretrendszer által létrehozott komplex tesztfüggvények segítségével, az eredmények felhasználása futódaru főtartó optimalásához



43. ábra: Tényleges legjobb fitnessz érték 100 Monte Carlo futás adatai alapján (F9)

4.9.2.2 Statisztikai adatok

	Átlag fitnessz (Mean)	Legjobb fitnessz	Legrosszabb fitnessz	Fitnessz szórás (Standard Deviation)
AiNet	4,70068E-15	8,88178E-16	3,75255E-14	5,06438E-15
BFOA	3,12663E-05	1,06004E-12	0,003126626	0,000312663
BA	4,44E-16	4,44E-16	4,44E-16	0
CA	0,001081325	4,44E-16	0,013597463	0,002106409
DE	8,28634E-06	3,07333E-09	0,000128563	1,88993E-05
HS	0,002849583	9,74143E-09	0,021138157	0,00414325
KH	4,98327E-05	1,31776E-09	0,001866252	0,000212142
MA	0,000662942	4,44E-16	0,008564271	0,001720352
NM	0,002853483	4,44E-16	0,030293441	0,005195602
PSO	4,44E-16	4,44E-16	4,44E-16	0
RS	1,51438E-05	8,7595E-08	0,000125568	2,03204E-05
SA	0,003484289	3,08961E-08	0,019564999	0,004380017

11. táblázat: Fitnessz értékek alakulása 500 iteráció után, 100 Monte Carlo futás adatai alapján (F9)

4.10 Three-hump camelback (F10)

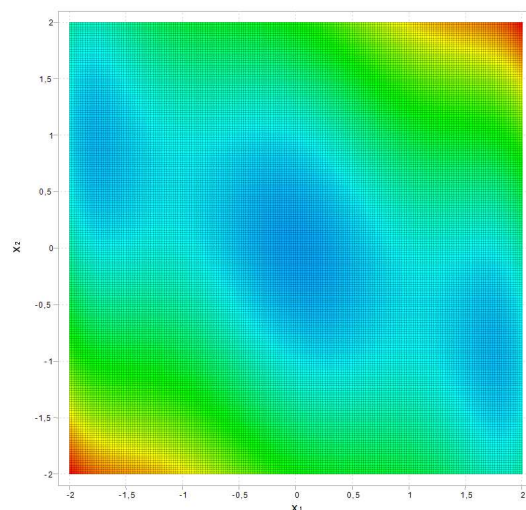
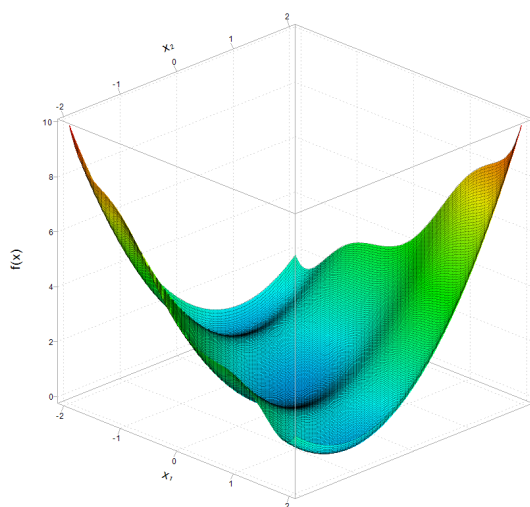
A Three-hump camelback tesztfüggvény globális optimuma az $x_i = 0, i = 1, \dots, n$ pontban található, fitness értéke $f(x) = 0$. Az n változó a dimenziószámot jelöli, a dimenziók minimum és maximum korlátjára az ajánlott érték $-2,048 \leq x_i \leq 2,048, i = 1, \dots, n$. A függvény kétváltozós változatát az alábbi képlet írja le:

$$f(x_1, x_2) = 2x_1^2 - 1,05x_1^4 + \frac{x_1^6}{6} + x_1x_2 + x_2^2 \quad (17)$$

A kétváltozós Three-hump camelback tesztfüggvény háromdimenziós ábrázolása és szintértéke a 44. ábra látható.

4.10.1 C# forráskód

```
public double Evaluate(double[] coords)
{
    double s1 = 0, s2 = 0, s3 = 0, s4 = 0, s5 = 0;
    s1 = 2 * Math.Pow(coords[0], 2);
    s2 = -1.05 * Math.Pow(coords[0], 4);
    s3 = Math.Pow(coords[0], 6) / 6;
    s4 = coords[0] * coords[1];
    s5 = Math.Pow(coords[1], 2);
    double fitness = s1 + s2 + s3 + s4 + s5;
    return fitness;
}
```



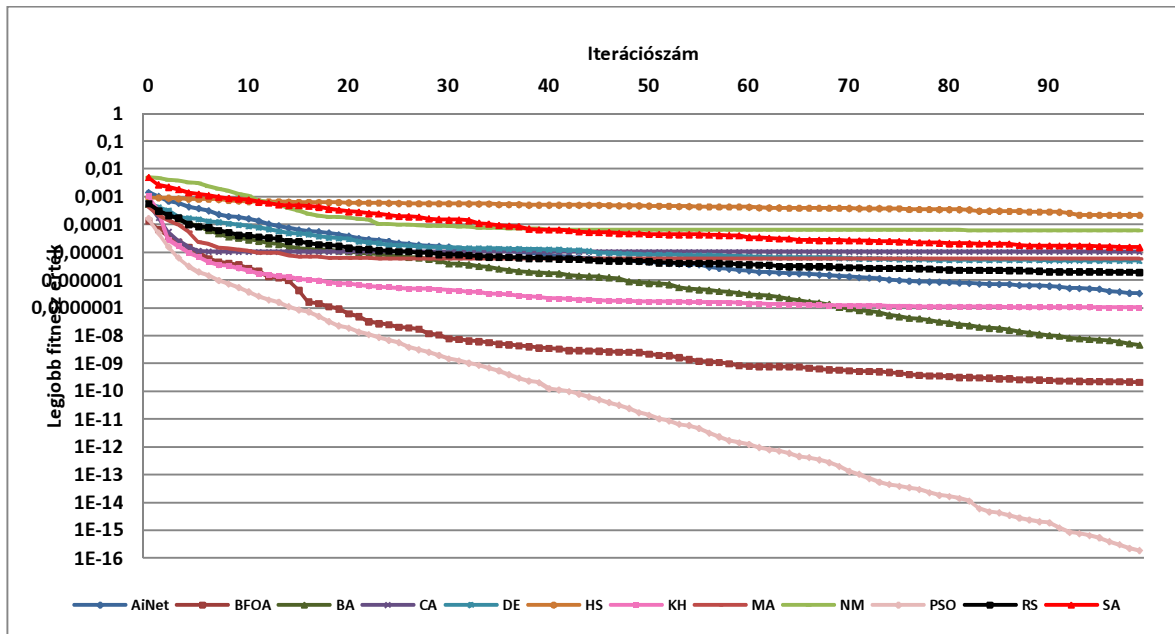
powered by: ILNumerics

powered by: ILNumerics

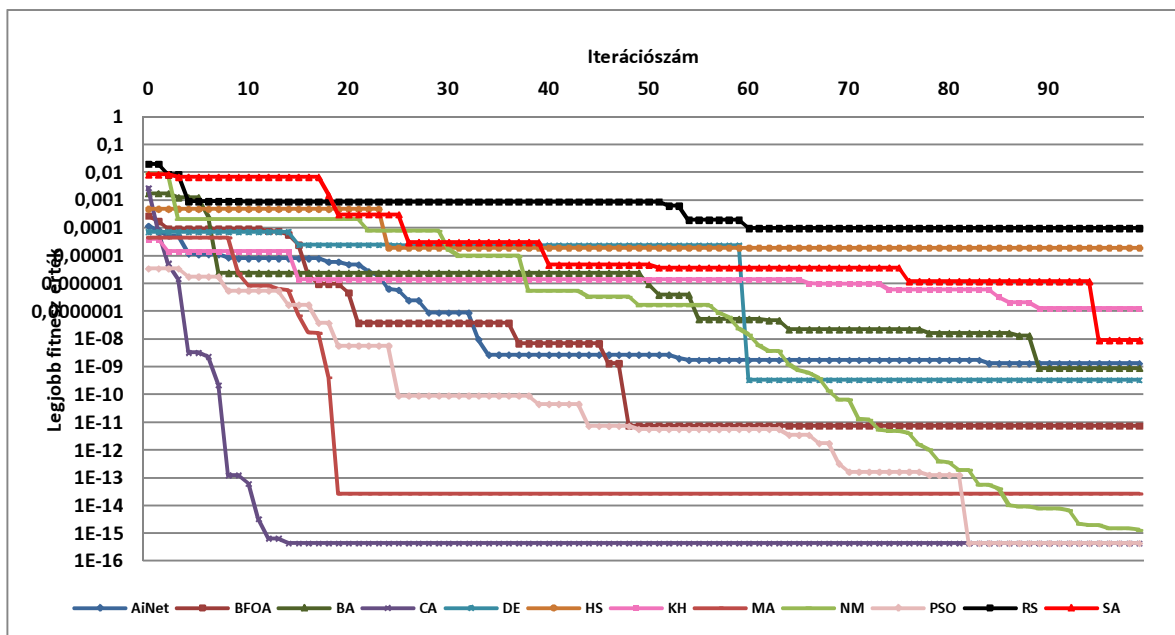
44. ábra: A kétváltozós Three-hump camelback tesztfüggvény háromdimenziós ábrázolása és szintértéke

4.10.2 Mérési eredmények

4.10.2.1 Konvergencia grafikonok



45. ábra: Legjobb fitnessz értékek átlaga 100 Monte Carlo futás adatai alapján (F10)



46. ábra: Tényleges legjobb fitnessz érték 100 Monte Carlo futás adatai alapján (F10)

4.10.2.2 Statisztikai adatok

	Átlag fitnessz (Mean)	Legjobb fitnessz	Legrosszabb fitnessz	Fitnessz szórás (Standard Deviation)
AiNet	1,1779E-09	1,16512E-11	9,52146E-09	1,42234E-09
BFOA	2,33231E-11	2,68452E-13	1,75567E-10	2,81774E-11
BA	4,44E-16	4,44E-16	4,44E-16	0
CA	1,03952E-05	4,44E-16	0,000424196	4,56164E-05
DE	1,26241E-06	3,33682E-10	3,45727E-05	4,3069E-06
HS	6,51929E-05	1,94617E-09	0,00051099	0,000101762
KH	8,09406E-08	1,92565E-10	3,67388E-07	9,95737E-08
MA	5,87049E-06	4,44E-16	8,082E-05	1,6093E-05
NM	6,25321E-05	2,22045E-16	0,003126603	0,00043993
PSO	4,44E-16	4,44E-16	4,44E-16	0
RS	2,14692E-07	2,31709E-10	1,20607E-06	2,24445E-07
SA	5,93131E-06	7,5481E-11	0,000107399	1,42284E-05

12. táblázat: Fitnessz értékek alakulása 500 iteráció után, 100 Monte Carlo futás adatai alapján (F10)

4.11 Az eredmények értékelése

Ackley's function (F1)	$f(x) = -20 \times \exp \left(-0,2 \times \sqrt{\frac{1}{n} \sum_{i=1}^n (x_i^2)} \right) - \exp \left(\frac{1}{n} \sum_{i=1}^n \cos(2\pi x_i) \right) + 20 + \exp(1)$	$-32,768 \leq x_i \leq 32,768$ $x_i = 0, i = 1, \dots, n$ $f(x)=0$
De Jong's function (F2)	$f(x) = \sum_{i=1}^n x_i^2$	$-5,12 \leq x_i \leq 5,12$ $x_i = 0, i = 1, \dots, n$ $f(x)=0$
Drop-Wave function (F3)	$f(x_1, x_2) = -\frac{1 + \cos(12 \times \sqrt{x_1^2 + x_2^2})}{\frac{1}{2}(x_1^2 + x_2^2) + 2} + 1$	$-5,12 \leq x_i \leq 5,12$ $x_i = 0, i = 1, \dots, n$ $f(x)=0$
Easom's function (F4)	$f(x_1, x_2) = -\cos(x_1) \times \cos(x_2) \times \exp(-(x_1 - \pi)^2 - (x_2 - \pi)^2) + 1$	$-10 \leq x_i \leq 10$ $x_i = \pi, i = 1, \dots, n$ $f(x)=0$
Griewangk's function (F5)	$f(x) = \frac{1}{4000} \times \sum_{i=1}^n x_i^2 - \prod_{i=1}^n \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$	$-600 \leq x_i \leq 600$ $x_i = 0, i = 1, \dots, n$ $f(x)=0$
Matyas's function (F6)	$f(x_1, x_2) = 0,26 \times (x_1^2 + x_2^2) - 0,48 \times x_1 \times x_2$	$-10 \leq x_i \leq 10$ $x_i = 0, i = 1, \dots, n$ $f(x)=0$
Rastrigin's function (F7)	$f(x) = 10 \times n + \sum_{i=1}^n [x_i^2 - 10 \times \cos(2\pi x_i)]$	$-5,12 \leq x_i \leq 5,12$ $x_i = 0, i = 1, \dots, n$ $f(x)=0$
Rosenbrock's valley (F8)	$f(x) = \sum_{i=1}^{n-1} [100 \times (x_{i+1} - x_i^2)^2 + (1 - x_i)^2]$	$-2,048 \leq x_i \leq 2,048$ $x_i = 1, i = 1, \dots, n$ $f(x)=0$
Schaffer's N. 2. function (F9)	$f(x_1, x_2) = 0,5 + \frac{\sin^2(x_1^2 - x_2^2) - 0,5}{[1 + 0,001 \times (x_1^2 + x_2^2)]^2}$	$-10 \leq x_i \leq 10$ $x_i = 0, i = 1, \dots, n$ $f(x)=0$
Three-hump camelback (F10)	$f(x_1, x_2) = 2x_1^2 - 1,05x_1^4 + \frac{x_1^6}{6} + x_1x_2 + x_2^2$	$-2,048 \leq x_i \leq 2,048$ $x_i = 0, i = 1, \dots, n$ $f(x)=0$

13. táblázat: Vizsgált alapfüggvények összefoglaló táblázata

A fejezetben összesen tíz, szakirodalomból vett tesztfüggvény segítségével végeztem el a heurisztikus algoritmusok hatékonyságvizsgálatát. A jelentős mennyiségű statisztikai adatot vizsgálva számos érdekességet fedeztem fel. Jól megfigyelhető például, hogy a különböző karakterisztikájú tesztfüggvények élesen befolyásolják az algoritmusok hatékonyságát.

A zajos, sok lokális optimummal rendelkező feladatoknál (Drop-wave, Rastrigin's, Schaffer N. 2.) inkább a rajintelligencia eljárások (BFOA, BA, KH, PSO), az Artificial Immune Network (AiNet) és a Differential Evolution (DE) működtek hatékonyan, alacsony szórással.

A Cultural Algorithm (CA) és Memetic Algorithm (MA) evolúciós algoritmusok működésén észrevehető volt, hogy azonos elven alapulnak. Az említett feladattípusok esetében gyakran ragadtak lokális optimumban. Elvitathatatlan előnyük azonban a grafikonokról jól leolvasható, nevezetesen hogy nagyon gyors és hatékony a konvergenciájuk. Ez a tulajdonság főképp a „völgy” jellegű függvényeknél (De Jong's, Matyas's, Rosenbrock's, Three-hump camelback) volt hasznos.

A lokális keresés bajnoka azonban egyértelműen a Nelder-Mead (NM) algoritmus, ami 500 iteráció alatt akár 10^{-146} nagyságrendig is eljutott. A legtöbb tesztfüggvény esetében legfeljebb 10^{-16} pontosság volt elérhető, mivel a .NET keretrendszer néhány általam használt matematikai függvénye (pl. Math.Sin(), Math.Cos(), Math.Exp(), stb.) ennyit tesz lehetővé. Másfajta matematikai függvénykönyvtárak segítségével természetesen nagyobb pontosság is elérhető, és bizonyos feladatokhoz (pl. csillagászat) bizonyára szükséges is, azonban én ragaszkodtam a .NET keretrendszer beépített megoldásához.

A Harmony Search (HS) és Simulated Annealing (SA) eljárás is gyakran volt az optimum közelében, működésük azonban eléggé hektikus volt. A statisztikai adatokban jól látható, hogy nagy szórással dolgoztak. A Harmony Search eljárás esetében még meg kell említeni, hogy konvergenciája nagyon lassú.

Egyértelműen legjobb eljárást nehéz lenne meghatározni, de a Particle Swarm Optimization (PSO) kiváló ötvözte a hatékony globális keresésnek, gyors konvergenciának és precíz lokális keresésnek. A Differenciális Evolúció (DE) és Bees Algorithm (BA) eljárások is megbízhatóan működtek a legtöbb esetben. A legprecízebb eredményt pedig több feladat esetében is a Nelder-Mead (NM) adta.

5 Komplex tesztfüggvények létrehozása saját keretrendszer segítségével

Az előző fejezetben a szakirodalomban található tesztfüggvények segítségével végeztem a heurisztikus algoritmusok hatékonyságvizsgálatát. Bebizonyosodott, hogy minél több, egymáshoz viszonylag közel lévő lokális optimum van, a kereső eljárásoknak annál nehezebb a dolga. Az irodalomkutatás során sajnos viszonylag kevés olyan tesztfüggvényt találtam (pl. Drop-Wave, Schaffer N. 2.), ami valódi nehézséget jelentett a legjobb algoritmusoknak. Megfogalmazódott bennem a kérdés, hogy nem lehetne-e az ismert tesztfüggvények (alapfüggvények) alapján még bonyolultabb tesztfeladatokat létrehozni. Liang, Suganthan és Deb „Novel composition test functions for numerical global optimization” című művében egy újfajta, komplex tesztfüggvények létrehozására szolgáló módszer alapjait ismerteti [46]. A módszer gyakorlati megvalósítására példát a [47] műben találtam, amiben a szerzők egzakt leírást adnak komplex tesztfüggvények dinamikus létrehozásához. Az elméleti és gyakorlati információk alapján sikerült létrehoznom egy szoftvert, ami ismert alapfüggvények kombinálásával, egyszerűen képes tetszőleges bonyolultságú komplex tesztfüggvényeket komponálni. Hatalmas előnye, hogy az alapfüggvények globális optimum pontjainak helyét és értékét a felhasználó határozhatja meg a létrehozott komplex tesztfüggvényen belül.

5.1 Komplex tesztfüggvények létrehozásának módszere

Az alábbiakban ismertetett módszer gyakorlatilag végtelen variációs lehetőséget nyújt tetszőleges bonyolultságú tesztfüggvények létrehozására úgy, hogy azok globális optima egyértelműen meghatározható. A komplex tesztfüggvény létrehozásához az alábbi bemenő paraméterekre van szükség:

- $[X_{min}, X_{max}]^D$ A komplex függvény változóinak minimum és maximum korlátja (értelmezési tartománya)
- $f_i(\theta)$ A kiválasztott alapfüggvények
- $[x_{min}, x_{max}]^D$ Az alapfüggvények változóinak minimum és maximum korlátja (értelmezési tartománya)
- D A függvények változóinak száma (dimenziószám)
- o_i Az i -edik alapfüggvény globális optimumának helye
- $bias(\psi)$ Vektor, aminek segítségével a felhasználó tetszőlegesen eltolhatja az alapfüggvények globális optimum értékét. Az alapfüggvény globális optimum értéke és az eltolás vektor megfelelő eleme alapján

egyértelműen meghatározható a komplex függvény globális optimum értéke és annak helye.

Az alapfüggvények globális optimum helyének eltolásához függvényértéküket a komplex függvény változóinak értelmezési tartományán kívül is meg kell határozni. A módszer működésének fontos feltétele, hogy az alapfüggvények globális optima független legyen az értelmezési tartománytól. A Schwefel's function például nem használható a módszerrel, mivel az értelmezési tartomány kiterjesztésével megváltozik a globális optimum helye. A komplex tesztfüggvény az alábbi képletek alapján határozható meg:

$$F(\theta) = \sum \left[w_i \left(\frac{f_i(\vartheta_i)}{f_i(o_i)} + bias_i \right) \right] \quad (18)$$

$$\vartheta_i = \frac{\theta - (X_{min} - x_{min_i} \kappa_i + \psi_i)}{\kappa_i} \quad (19)$$

$$\kappa_i = \frac{(X_{max} - X_{min})}{(x_{max_i} - x_{min_i})} \quad (20)$$

- ψ_i Az eltolási vektor.
- w_i A súlyozó függvény, ami biztosítja, hogy az előre definiált optimum pontok nem tolódnak el. A komplex függvényérték számításakor minél közelebb kerülünk egy alapfüggvény optimum pontjához, az annál nagyobb súllyal szerepel. A szoftver jelenleg háromféle súlyozó függvényt ismer.

5.1.1 Súlyozó függvények

5.1.1.1 Euklideszi távolság

Az euklideszi távolságon alapuló súlyozáskor a komplex függvény adott pontja, és az alapfüggvények globális optimum pontjai közti euklideszi távolság a döntő.

$$w_i = \text{euklideszi_távolság}(\vartheta_i, o_i) \quad (21)$$

A távolságok normalizálása:

$$w_i = \frac{w_i}{\sum w} \quad (22)$$

$$w_i = 1 - w_i \quad (23)$$

Ha $w_i \neq \text{maximum}(w)$, és n az alapfüggvények száma:

$$w_i = \frac{1 - \text{maximum}(w)}{(n-1)} \quad (24)$$

5.1.1.2 Gábor-függvény

Bonyolultabb komplex függvényeket hozhatunk létre, ha a súlyozó függvény zajos jellegű. Az eredeti optimum értékek megtartásához a súlyozó függvény 0 és 1 közötti értékeket kell visszaadjon úgy, hogy az alapfüggvények optimum pontjában az érték 1 legyen. A Gábor-féle függvény biztosítja ezt a feltételt:

$$w_i = \left| \prod_k \cos(\vartheta_i(k)\tau_1) e^{-\frac{\sum_k \vartheta_i(k)^2}{\tau_2}} \right| \quad (25)$$

- $\vartheta_i(k)$ A ϑ_i vektor k -adik eleme
- τ_1 A zajossági tényező. Minél nagyobb az érték, annál zajosabb a függvény.
- τ_2 A konvergencia távolság.

Ha $w_i \neq \text{maximum}(w)$, és n az alapfüggvények száma:

$$w_i = \frac{1 - \text{maximum}(w)}{(n-1)} \quad (26)$$

5.1.1.3 Gauss-függvény

A Gauss-függvény használatával simább éleket érhetünk el a komplex tesztfüggvény esetében:

$$w_i = e^{-\frac{(\vartheta_i - o_i)^2}{6}} \quad (27)$$

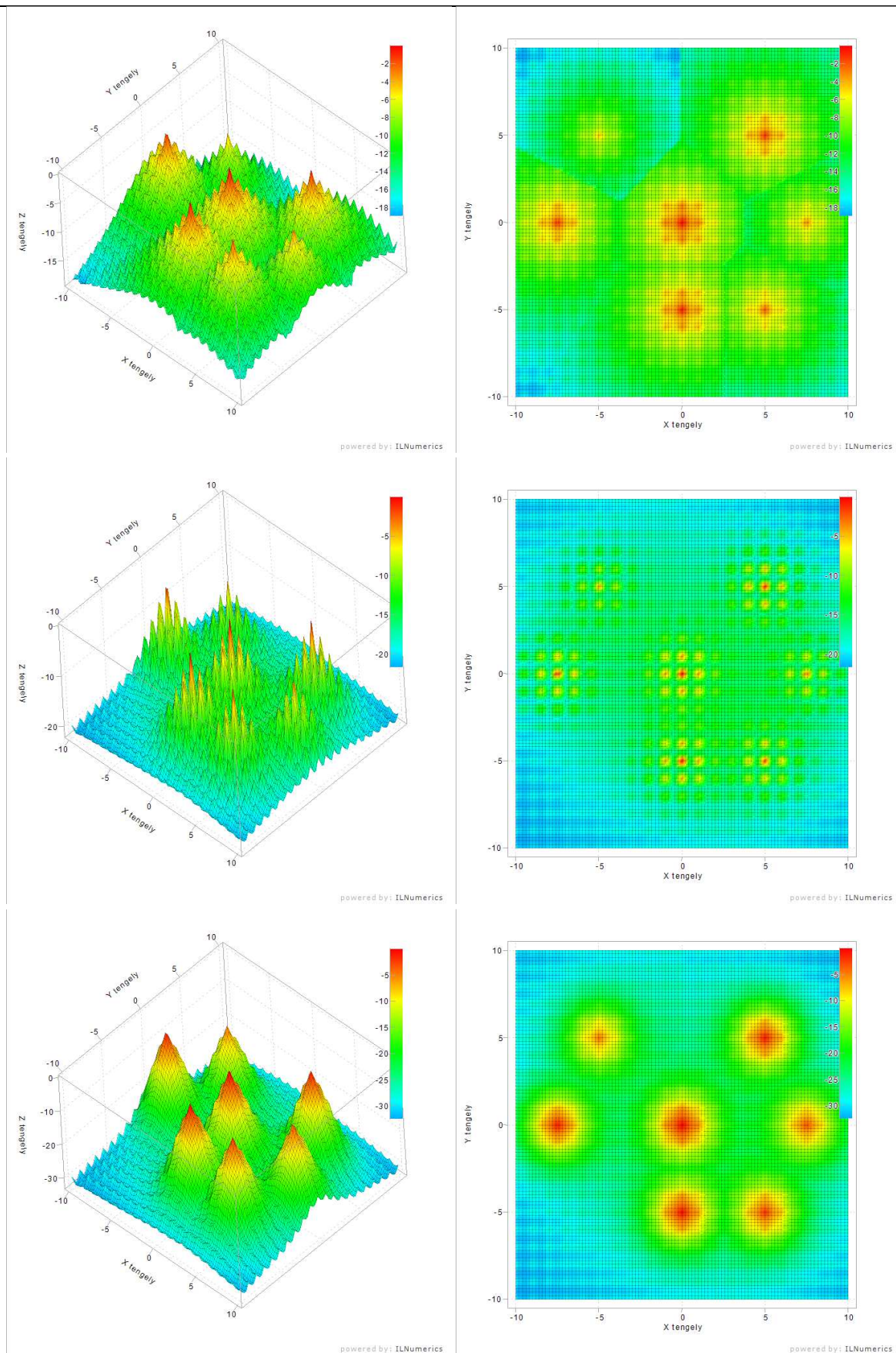
Ha $w_i \neq \text{maximum}(w)$, és n az alapfüggvények száma:

$$w_i = \frac{1 - \text{maximum}(w)}{(n-1)} \quad (28)$$

5.1.2 Gyakorlati példa komplex tesztfüggvény létrehozására

Az alábbiakban egy kétváltozós komplex tesztfüggvényt (Komplex N. 1.) hozok létre az ismertetett módszerrel, a következő bemenő paraméterekkel:

- $[-10, 10]^D$ A komplex függvény változóinak értelmezési tartománya
- $f_i(\theta)$ A kiválasztott alapfüggvény 7 darab Ackely's function.
- $[x_{min}, x_{max}]^D$ Az alapfüggvények globális optimum helyének eltolásához függvényértékeket a komplex függvény változóinak értelmezési tartományán kívül is meg kell határozni, ezért nem fontos megadni.
- $D = 2$ A függvények változóinak száma (dimenziószám)
- ox Vektor, az optimumok x koordinátái: [7,5; 5; 5; 0; -7,5; -5; 0]
- oy Vektor, az optimumok y koordinátái: [0; 5; -5; -5; 0; 5; 0]
- $bias$ Vektor, az optimumok értékek eltolása: [3; 1; 2; 0,5; 1; 4,5; 0]

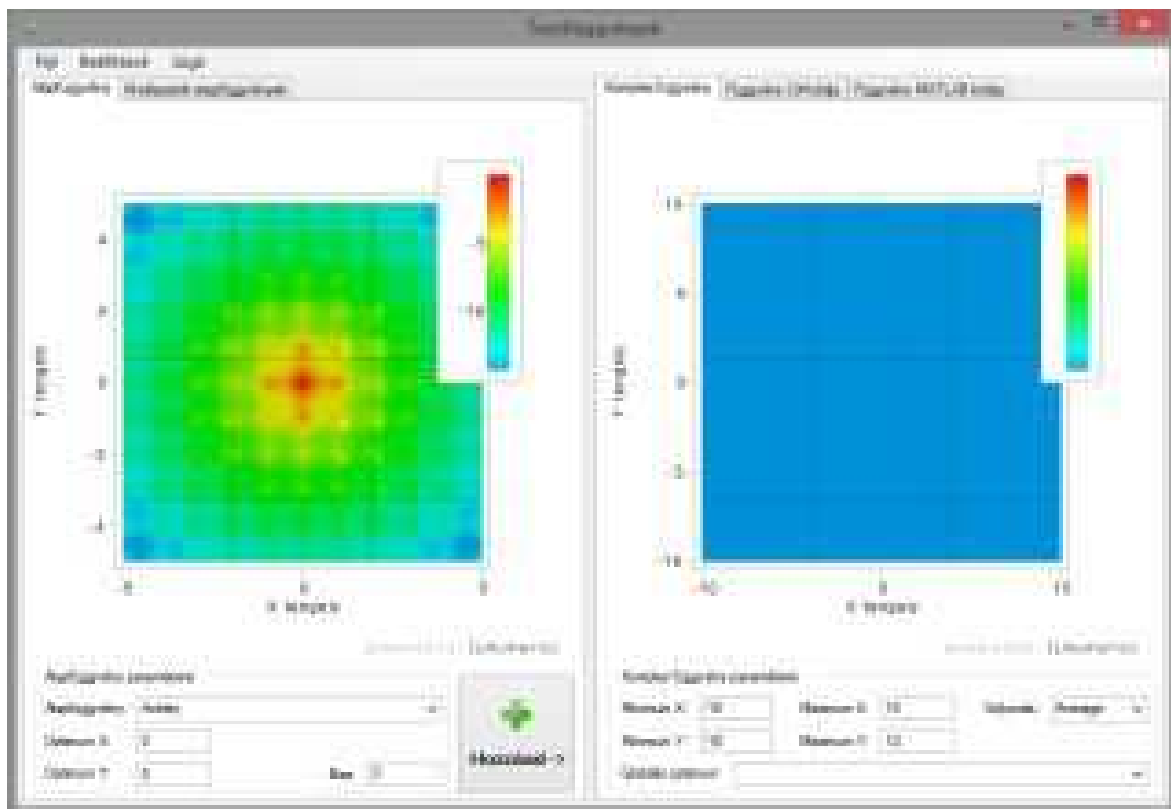


47. ábra: Komplex N. 1. tesztfüggvény euklideszi, Gábor és Gauss ($\tau_1 = 3, \tau_2 = 8$) súlyozással

5.2 Szoftver komplex tesztfüggvények létrehozásához

Olyan szoftvert készítettem, ami a felhasználó számára lehetővé teszi, hogy a lehető legegyszerűbben konstruálhasson saját tesztfüggvényt. Az alkalmazás elkészítéséhez a napjainkban rendkívül elterjedt .NET keretrendszert választottam, a kódolást C# nyelven végeztem. Az ILNumerics matematikai függvénykönyvtárat használtam a 3D felületek és kontúr térképek megjelenítésére, valamint a felhasználói interakciók lekezelésére a 3 dimenziós térben. Az alkalmazás főképernyője a 48. ábra látható. A felhasználói felület alapvetően két részre van osztva, bal oldalon az alapfüggvényekhez, jobb oldalon a komplex függvényhez tartozó vezérlők találhatók. A bal oldalon a következő bemenő paraméterek adhatók meg:

- $f_i(\theta)$ Alapfüggvény kiválasztása és komplex tesztfüggvényhez adása. Egy legördülő menüből választhat a felhasználó összesen 21 darab alapfüggvény közül.
- ox, oy Az alapfüggvény globális optimum pontjának kívánt x és y koordinátája a komplex függvényben.
- $bias$ Az alapfüggvény globális optimum értékének eltolása a komplex függvényben.

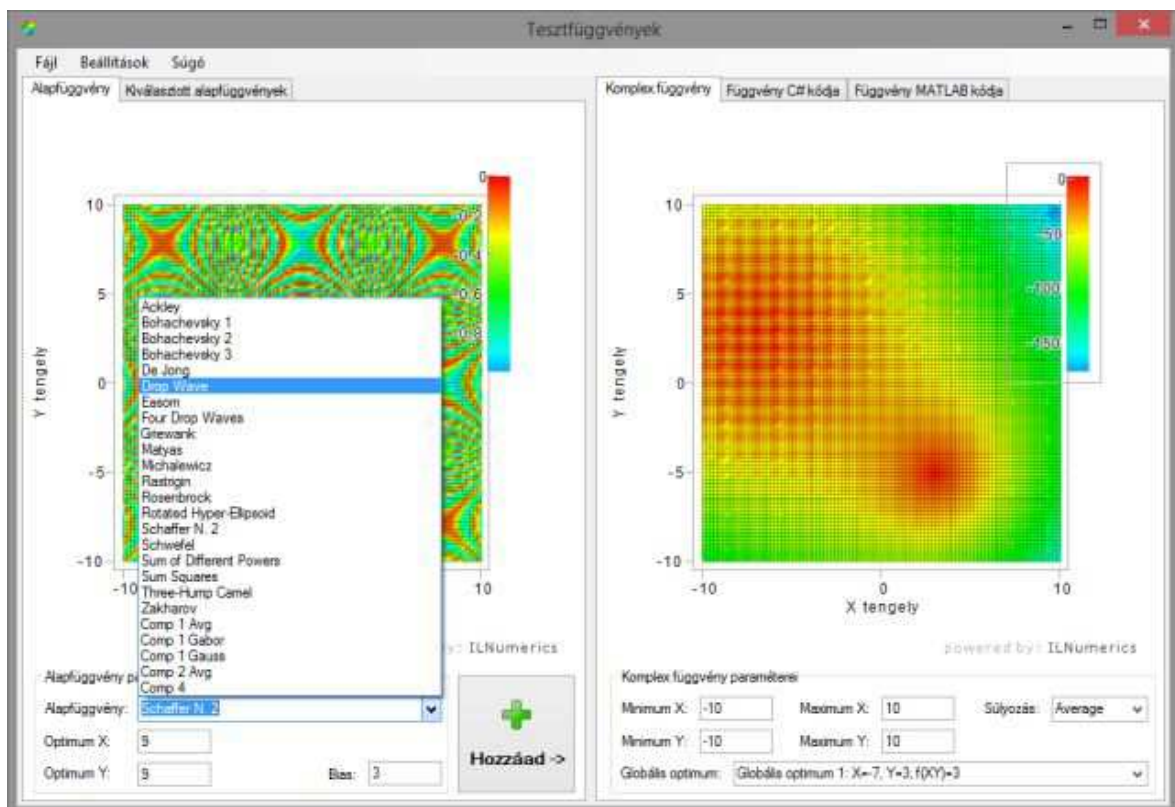


48. ábra: Komplex tesztfüggvények létrehozására szolgáló szoftver főképernyője

Az ox , oy , bias paramétereket szövegdoboz vezérlőbe kell beírni. Természetesen a szoftverbe épített fejlett hibaellenőrzés kiszűri a hibásan megadott értékeket, és jelez a felhasználónak.

A felhasználói felület jobb oldalán a komplex függvény ábrázolása, illetve a hozzá tartozó paraméterek vannak. Bármelyik vezérlő értéke változzon is meg, a komplex függvény ábrázolása dinamikusan követi azt. A komplex függvény globális optimum pontját (vagy pontjait) is egyértelműen jelzi a program (49. ábra).

- $[X_{min}, X_{max}]^D$ A komplex függvény változóinak minimum és maximum korlátja (értelmezési tartománya)
- **Súlyozás** A felhasználó egy legördülő menüből választhatja ki a három beépített súlyozó függvény valamelyikét.

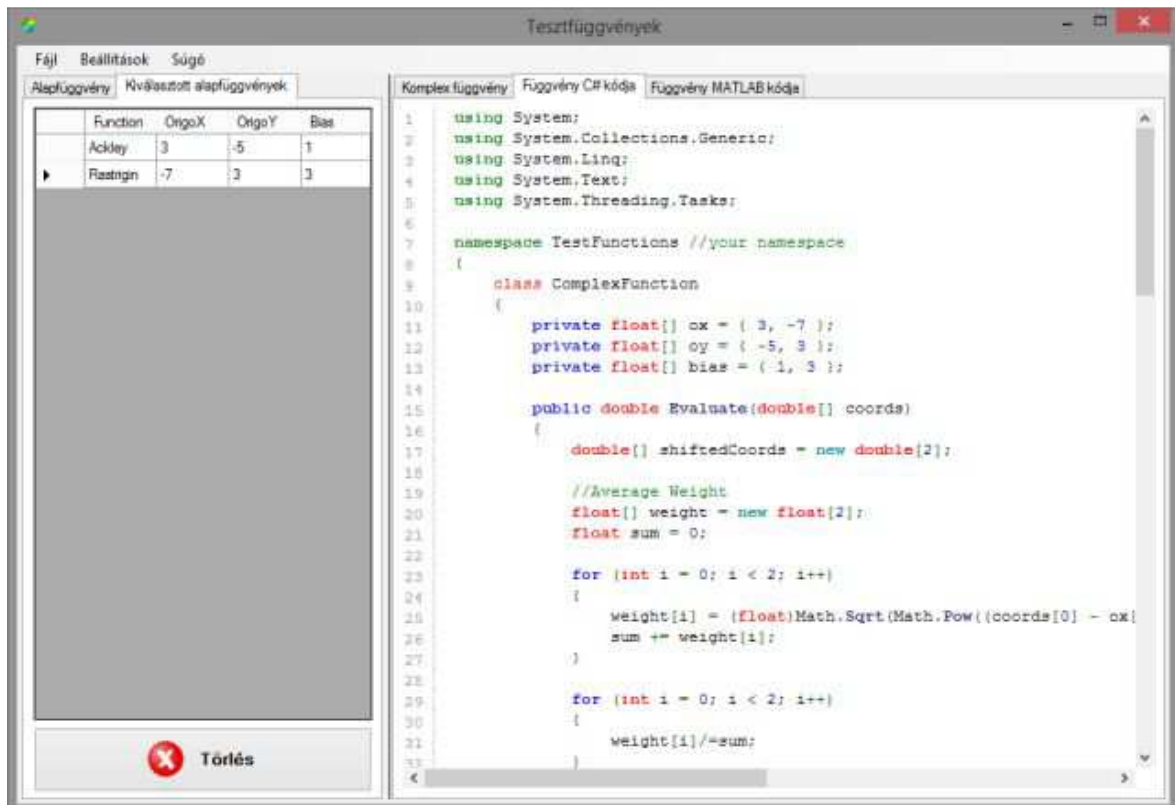


49. ábra: Alapfüggvény kiválasztása

5.2.1 Komplex függvény C# kódjának automatikus létrehozása

A szoftver legfontosabb és egyben legjobb funkciója, hogy dinamikusan hozza létre a komplex függvényt leíró C# forráskódot. A felhasználó ezt egy speciális kódjelölő (syntax highlighting) szövegszerkesztő-vezérlőben követheti nyomon. A kód dinamikus létrehozásához a .NET StringBuilder osztályát használtam. Magát a funkciót az teszi

lehetővé, hogy a komplex tesztfüggvények ismert építőkövekből (azaz alapfüggvényekből) épülnek fel. Az alapfüggvények forráskódjának struktúrája alapvetően nem változik, bárhogyan dolgozik a felhasználó. Kiemelt figyelmet a különböző bemenő paraméterekre, az alapfüggvények megadási sorrendjére, és a kiválasztott súlyozásra kellett fordítanom. Az általam készített osztály a felsorolt változók alapján, sorról sorra építi fel a kódot, rengeteg időt és munkát megtakarítva a felhasználók számára.



50. ábra: C# forráskód automatikus létrehozása

5.2.2 Továbbfejlesztési lehetőségek

Céлом a későbbiekben egy webalkalmazás létrehozása, amit böngészőből, Interneten keresztül lehet elérni. A tesztfüggvény generátor ezáltal könnyen hozzáférhetővé válna bárki számára, aki érdeklődik a témában. A problémát az jelenti, hogy a szoftvernek gyors 3 dimenziós megjelenítésre van szüksége. Bár a jelenlegi webes technológiákkal (pl. Javascript) elvileg ez megoldható, véleményem szerint még mindig komoly kompromisszumokra van szükség, ha az asztali alkalmazások sebességéhez viszonyítunk. A rohamosan terjedő WebGL technológia azonban nagyon ígéretesnek tűnik a hardveres gyorsítási lehetőségei miatt.

6 Globális optimum keresése saját komplex tesztfüggvény esetében

Az 5. fejezetben ismertettem egy eljárást komplex tesztfüggvények létrehozására, valamint egy általam készített szoftvert, ami az elméleti módszert átülteti a gyakorlatba. A szoftver létrehozta a 47. ábra látható Komplex N. 1. függvény C# kódját, így kézenfekvő, ha a heurisztikus algoritmusok hatékonyságvizsgálatát ezzel a feladattal folytatom. Külön vizsgálom az euklideszi távolságon, valamint a Gábor és Gauss függvényeken alapuló súlyozási eseteket. Az algoritmusok bemenő paramétereit a 4. fejezetben szerzett tapasztalatok alapján módosítottam. A legtöbb algoritmus esetében a populációszámot növeltem meg, a Bacterial Foraging Algorithm (BFOA) és Bees Algorithm (BA) eljárásoknál pedig a lokális keresés hatékonyságán próbáltam javítani.

Algoritmus	Paraméter neve	Paraméter értéke
Artificial Immune Network (AiNet)	PopulationSize	50→100
	Nrandom	2→5
Bacterial Foraging Algorithm (BFOA)	Cellsnum	50→100
	Ns	10→20
Bees Algorithm (BA)	Beesnum	50→100
	Sitesnum	3→5
	EliteSitesnum	1→2
	OtherBeesnum	2→5
	EliteBeesnum	7→15
Cultural Algorithm (CA)	Populationnum	50→100
Differential Evolution (DE)	Populationsize	50→100
Harmony Search (HS)	Memorysize	50→100
Krill Herd (KH)	Populationsize	50→100
Memetic Algorithm (MA)	Populationsize	50→100
Random Search (RS)	Populationsize	50→100
Particle Swarm Optimization (PSO)	Populationsize	50→100

14. táblázat: Az algoritmusok bemenő paramétereinek módosítása a 4. fejezetben szerzett tapasztalatok alapján

6.1 Komplex N. 1. tesztfüggvény

Ismétlésképpen, a kétváltozós Komplex N. 1. tesztfüggvényt az alábbi bemenő paraméterekkel hoztam létre:

- $[-10, 10]^D$ A komplex függvény változóinak értelmezési tartománya
- $f_i(\theta)$ A kiválasztott alapfüggvények halmaza 7 darab Ackely's function.
- $D = 2$ A függvények változóinak száma (dimenziószám)
- ox Vektor, az optimumok x koordinátái: [7,5; 5; 5; 0; -7,5; -5; 0]
- oy Vektor, az optimumok y koordinátái: [0; 5; -5; -5; 0; 5; 0]
- $bias$ Vektor, az optimum értékek eltolása: [3; 1; 2; 0,5; 1; 4,5; 0]

A Komplex N. 1. tesztfüggvény globális optimumának helye és értéke a bemenő paraméterek alapján egyértelműen meghatározható. Kizárólag Ackley's alapfüggvényeket használtam, melyek globális optimum (minimum) értéke alapesetben 0. A komplex tesztfüggvény globális optimum (minimum) értéke tehát súlyozástól függetlenül a bias vektor legkisebb eleme. A vektor legkisebb elemének értéke 0, indexe pedig 7. A globális optimum helyét ez alapján az ox és oy vektorok 7. eleme adja (0 és 0).

A Komplex N. 1. tesztfüggvény globális optima az $x_i = 0$, $i = 1, 2$ pontban található, fitness értéke $f(x) = 0$. A dimenziók minimum és maximum korlátjára az ajánlott érték $-10 \leq x_i \leq 10$, $i = 1, 2$.

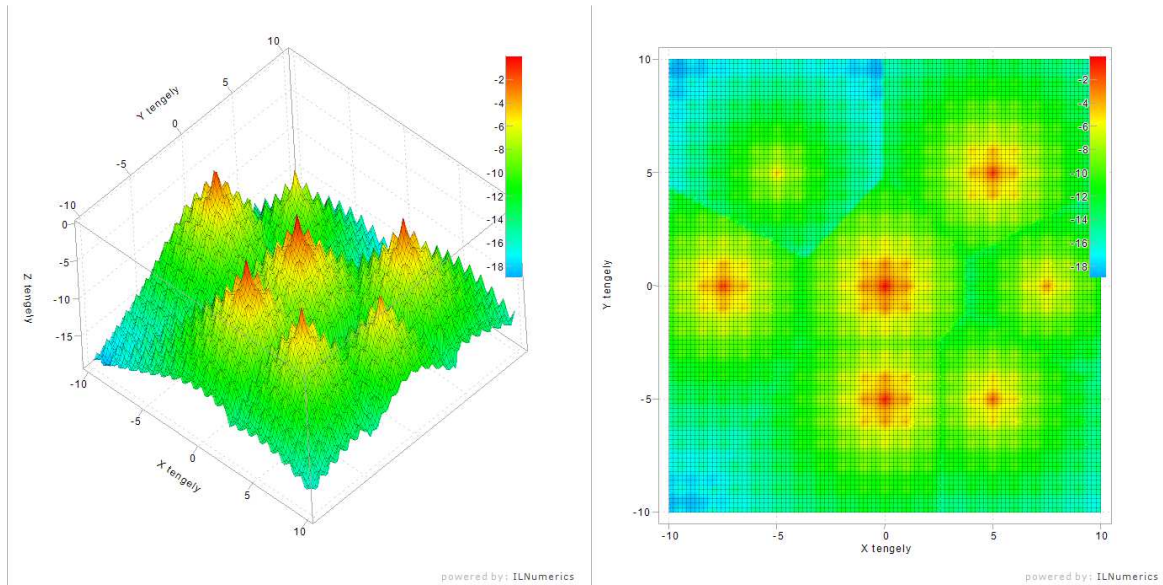
A szoftver által létrehozott függvény eredetileg maximum keresési feladathoz való (globális maximuma ismert). A dolgozatban azonban minimum keresési feladatokat oldok meg. A függvényt minimum keresési feladattá alakítom azáltal, hogy az eredeti fitness értékeket beszorzom -1-el.

6.1.1 C# forráskódok

A komplex függvények C# forráskódját terjedelmük miatt nem mellékelem a dolgozathoz. Az általam fejlesztett, komplex függvények létrehozására szolgáló szoftvert viszont elérhetővé tettem az Interneten (letölthető a bekezdés alján lévő linkről). A leírt bemenő paraméterek alapján az érdeklődő egyszerűen rekonstruálhatja a dolgozatban ismertetett komplex tesztfüggvényeket, a program pedig automatikusan létrehozza a hozzájuk tartozó C# forráskódot.

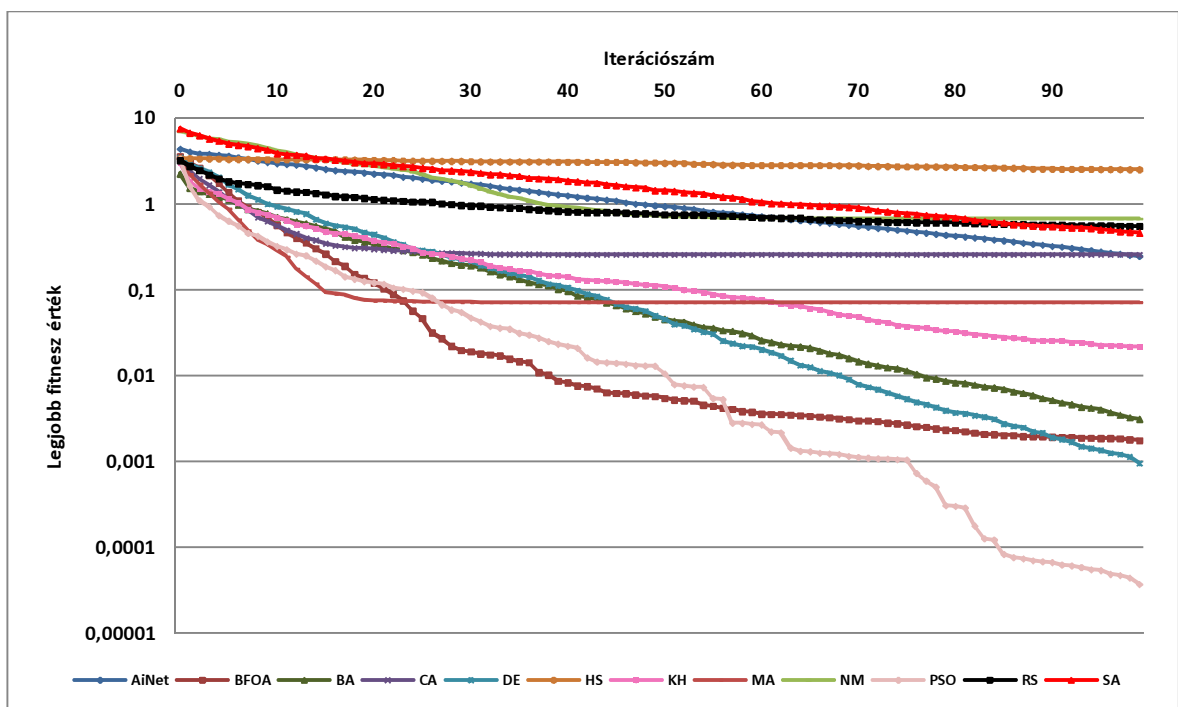
<https://drive.google.com/file/d/0BxE6yHbGFZBARlpkaXVLR2xDaWc/edit?usp=sharing>

6.1.2 Komplex N. 1. euklideszi távolságon alapuló súlyozás esetén (F11)



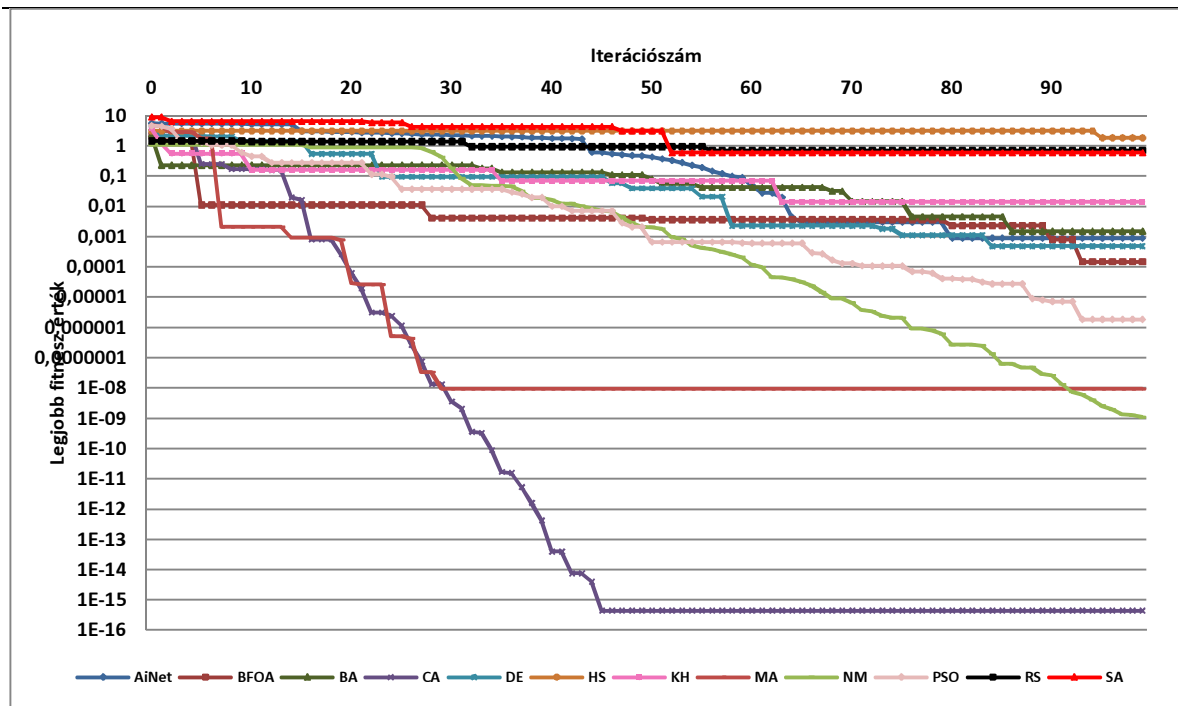
51. ábra: A kétváltozós Komplex N. 1. tesztfüggvény háromdimenziós ábrázolása és szintértéke euklideszi távolságon alapuló súlyozás esetén

6.1.2.1 Konvergencia grafikonok



52. ábra: Legjobb fitness értékek átlaga 100 Monte Carlo futás adatai alapján (F11)

Metaheurisztikus algoritmusok hatékonyságvizsgálata saját keretrendszer által létrehozott komplex tesztfüggvények segítségével, az eredmények felhasználása futódaru főtartó optimalásához



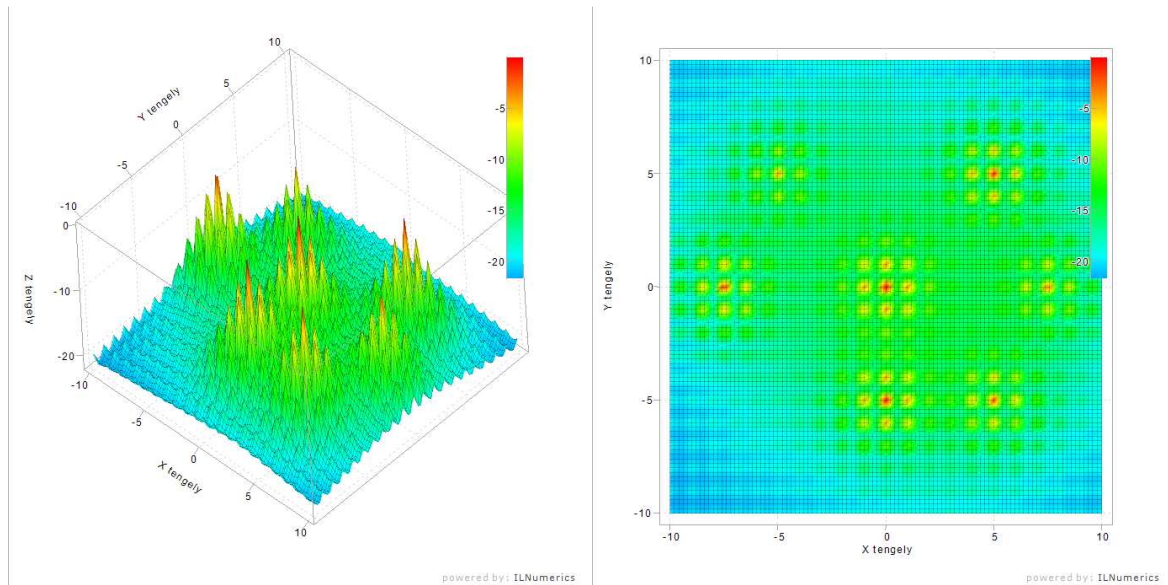
53. ábra: Tényleges legjobb fitness érték 100 Monte Carlo futás adatai alapján (F11)

6.1.2.2 Statisztikai adatok

	Átlag fitness (Mean)	Legjobb fitness	Legrosszabb fitness	Fitness szórás (Standard Deviation)
AiNet	0,000956	0,000109	0,003107	0,00055
BFOA	0,210503	6,3E-05	1,00069	0,334683
BA	6,59E-12	4,84E-13	1,83E-11	3,64E-12
CA	0,689681	4,44E-16	2,858973	0,778441
DE	5,9E-12	4,44E-16	3,19E-10	3,38E-11
HS	0,479263	0,002236	3,426226	0,689813
KH	0,062728	5,34E-06	0,500044	0,160567
MA	0,398274	9,31E-09	2,948793	0,587266
NM	0,495001	1,82E-14	3,000003	0,613383
PSO	4,44E-16	4,44E-16	4,44E-16	0
RS	0,357327	0,034228	0,876233	0,207257
SA	0,173802	9,05E-07	2,093875	0,328309

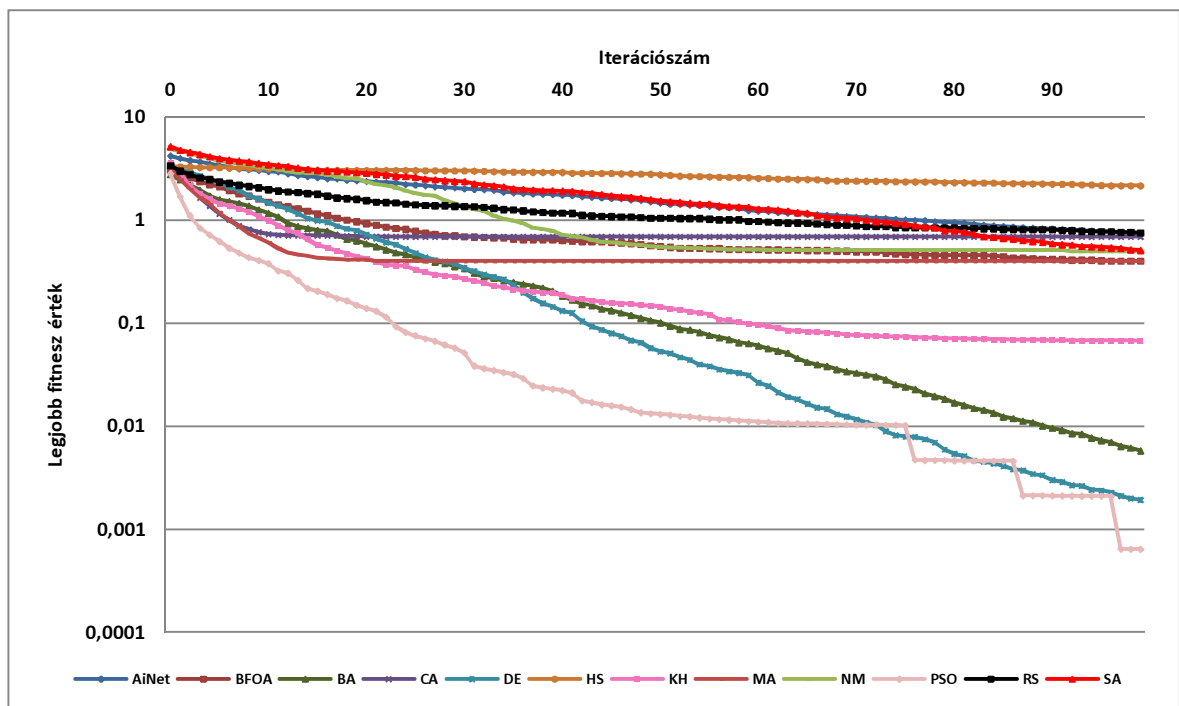
15. táblázat: Fitness értékek alakulása 500 iteráció után, 100 Monte Carlo futás adatai alapján (F11)

6.1.3 Komplex N. 1. Gábor-féle súlyozás ($\tau_1 = 3, \tau_2 = 8$) esetén (F12)



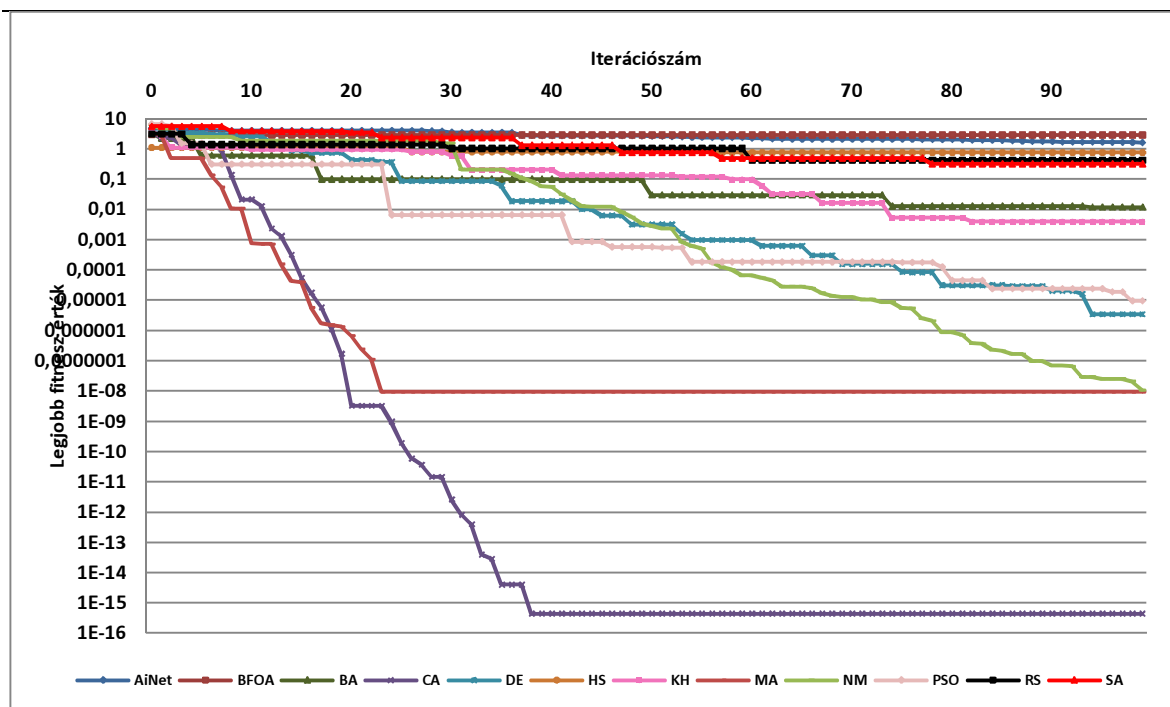
54. ábra: A kétváltozós Komplex N. 1. tesztfüggvény háromdimenziós ábrázolása és szintértéke
Gábor-féle súlyozás ($\tau_1 = 3, \tau_2 = 8$) esetén

6.1.3.1 Konvergencia grafikonok



55. ábra: Legjobb fitnessz értékek átlaga 100 Monte Carlo futás adatai alapján (F12)

Metaheurisztikus algoritmusok hatékonyságvizsgálata saját keretrendszer által létrehozott komplex tesztfüggvények segítségével, az eredmények felhasználása futódaru főtartó optimalásához



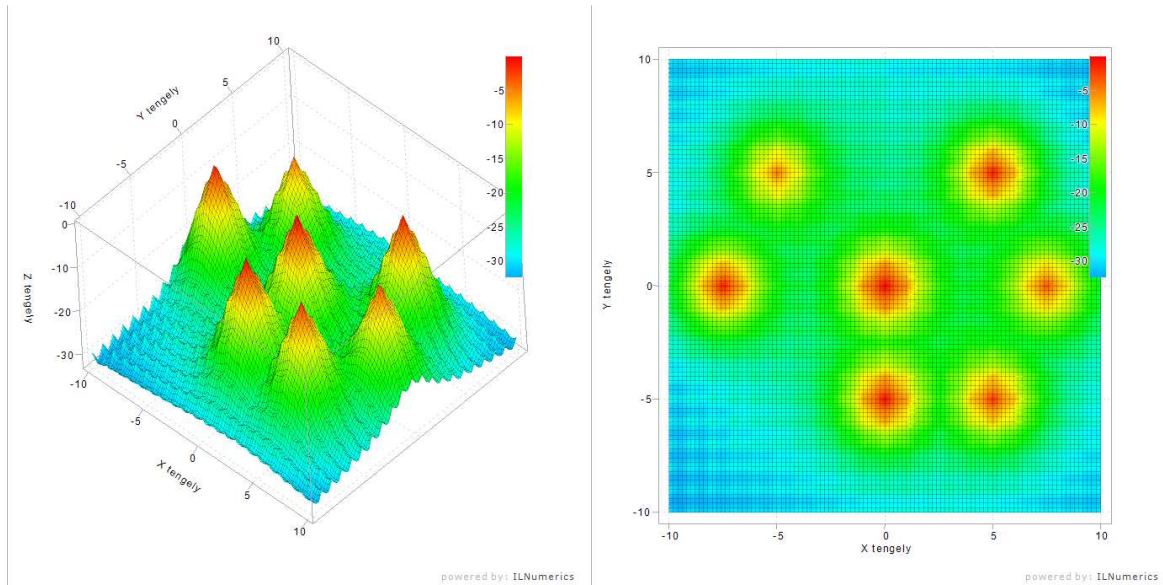
56. ábra: Tényleges legjobb fitnessz érték 100 Monte Carlo futás adatai alapján (F12)

6.1.3.2 Statisztikai adatok

	Átlag fitnessz (Mean)	Legjobb fitnessz	Legrosszabb fitnessz	Fitnessz szórás (Standard Deviation)
AiNet	0,000956	0,000109	0,003107	0,00055
BFOA	0,210503	6,3E-05	1,00069	0,334683
BA	6,59E-12	4,84E-13	1,83E-11	3,64E-12
CA	0,689681	4,44E-16	2,858973	0,778441
DE	5,9E-12	4,44E-16	3,19E-10	3,38E-11
HS	0,479263	0,002236	3,426226	0,689813
KH	0,062728	5,34E-06	0,500044	0,160567
MA	0,398274	9,31E-09	2,948793	0,587266
NM	0,495001	1,82E-14	3,000003	0,613383
PSO	4,44E-16	4,44E-16	4,44E-16	0
RS	0,357327	0,034228	0,876233	0,207257
SA	0,173802	9,05E-07	2,093875	0,328309

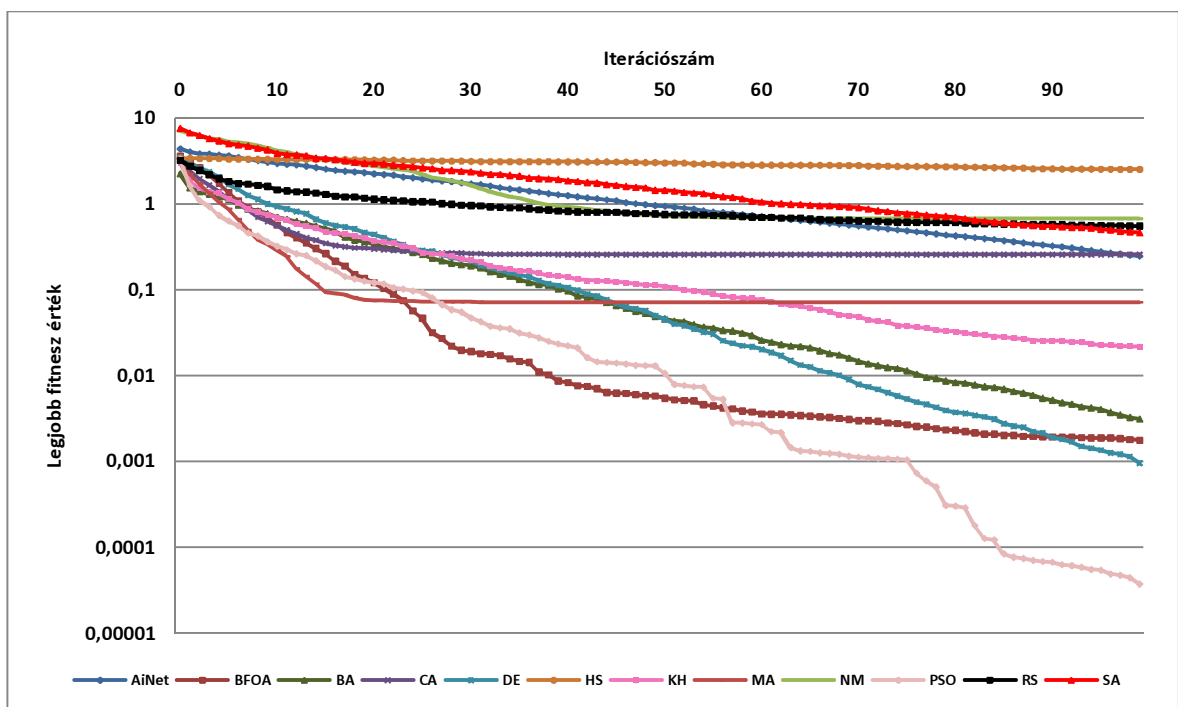
16. táblázat: Fitnessz értékek alakulása 500 iteráció után, 100 Monte Carlo futás adatai alapján (F12)

6.1.4 Komplex N. 1. Gauss-féle súlyozás esetén (F13)



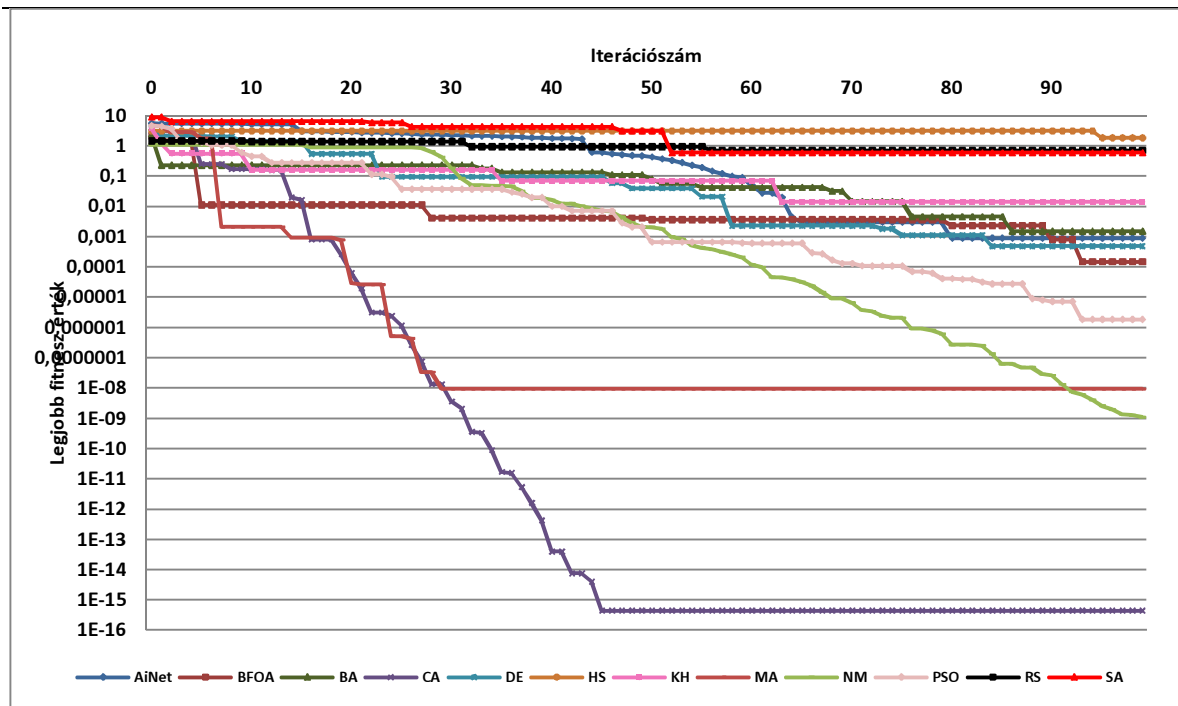
57. ábra: A kétváltozós Komplex N. 1. tesztfüggvény háromdimenziós ábrázolása és színtérképe Gauss-féle súlyozás esetén

6.1.4.1 Konvergencia grafikonok



58. ábra: Legjobb fitnessz értékek átlaga 100 Monte Carlo futás adatai alapján (F13)

Metaheurisztikus algoritmusok hatékonyságvizsgálata saját keretrendszer által létrehozott komplex tesztfüggvények segítségével, az eredmények felhasználása futódaru főtartó optimalásához



59. ábra: Tényleges legjobb fitness érték 100 Monte Carlo futás adatai alapján (F13)

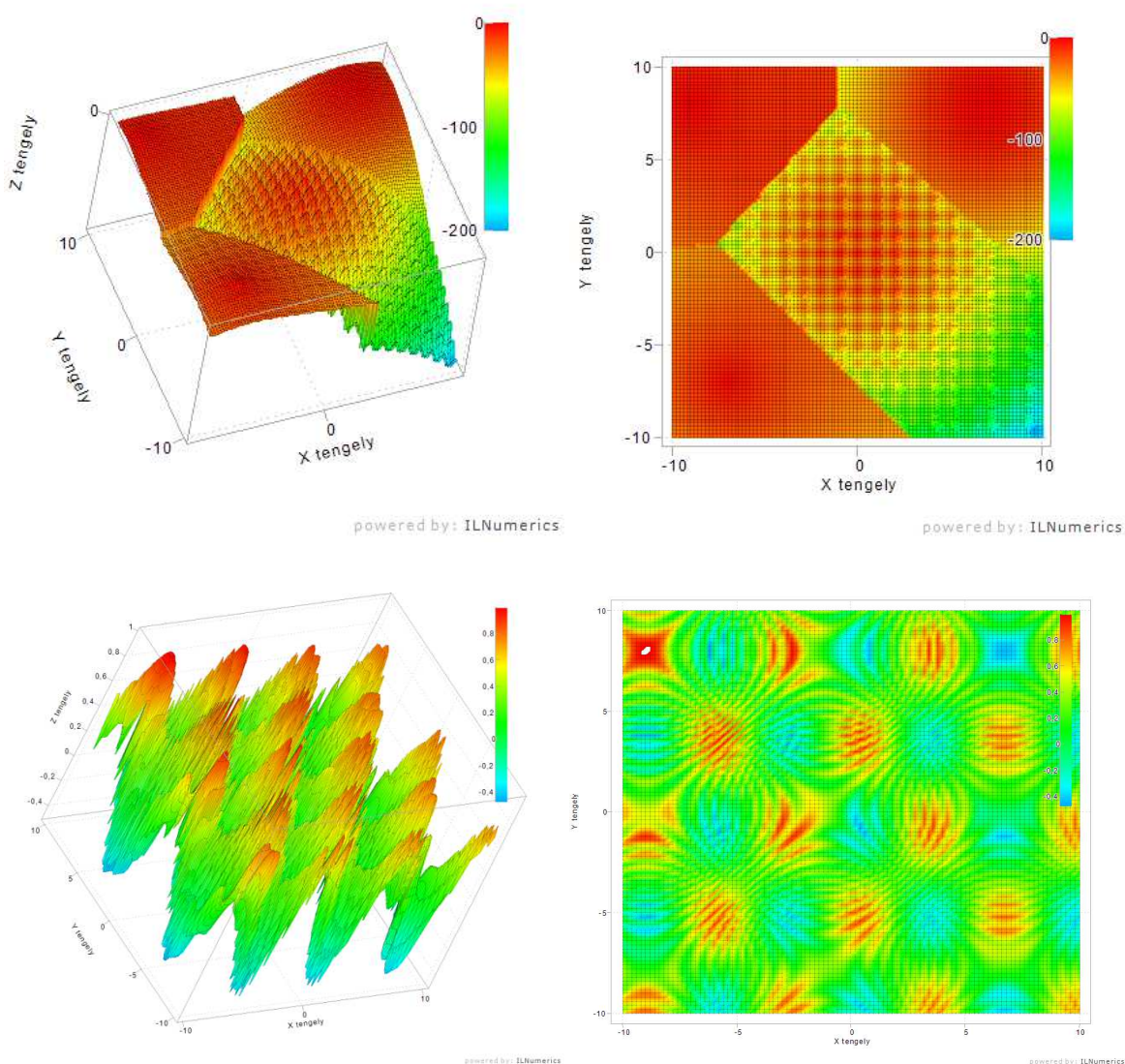
6.1.4.2 Statisztikai adatok

	Átlag fitnessz (Mean)	Legjobb fitnessz	Legrosszabb fitnessz	Fitnessz szórás (Standard Deviation)
AiNet	1,8E-08	1,61E-10	3,87E-07	4,76E-08
BFOA	1,891596	0,000102	9,902966	2,776526
BA	7E-12	2,17E-13	1,69E-11	3,55E-12
CA	0,641443	4,44E-16	10,22344	1,594372
DE	4,44E-16	4,44E-16	4,44E-16	0
HS	2,623005	0,01139	7,681816	1,807531
KH	3,56E-05	5,91E-07	8,87E-05	1,98E-05
MA	0,737611	3,05E-08	3,633828	1,174338
NM	6,89E-14	1,47E-14	4,41E-13	6,42E-14
PSO	4,44E-16	4,44E-16	4,44E-16	0
RS	1,405826	0,037104	2,898997	0,746866
SA	3,816731	8,32E-07	17,68951	4,201285

17. táblázat: Fitnessz értékek alakulása 500 iteráció után, 100 Monte Carlo futás adatai alapján (F2)

6.2 Az eredmények értékelése

A 6. fejezetben egy általam létrehozott komplex tesztfüggvény segítségével végeztem el a heurisztikus algoritmusok hatékonyságvizsgálatát. Jól látható az eredményeken, hogy sokkal nehezebb dolga van a kereső eljárásoknak. Az 5. fejezetben ismertetett szoftver gyakorlatilag végtelen lehetőséget kínál tetszőlegesen bonyolult tesztfeladatok létrehozására. Az is megfigyelhető, hogy a különböző súlyozási módszerek eltérő karakterisztikát kölcsönöznek a függvénynek. A Gábor-féle súlyozás erősen zajosít, ezáltal rengeteg, egymáshoz közeli lokális optimumot létrehozva. A Gauss-féle súlyozás ezzel szemben lefaragja, elsimítja a kiugró éleket. A 60. ábra további komplex tesztfüggvény példák láthatók, amik több különböző alapfüggvényből épülnek fel.



60. ábra: Példák a szoftverrel létrehozott komplex tesztfüggvényekre

7 Futódaru főtartó optimalizálása

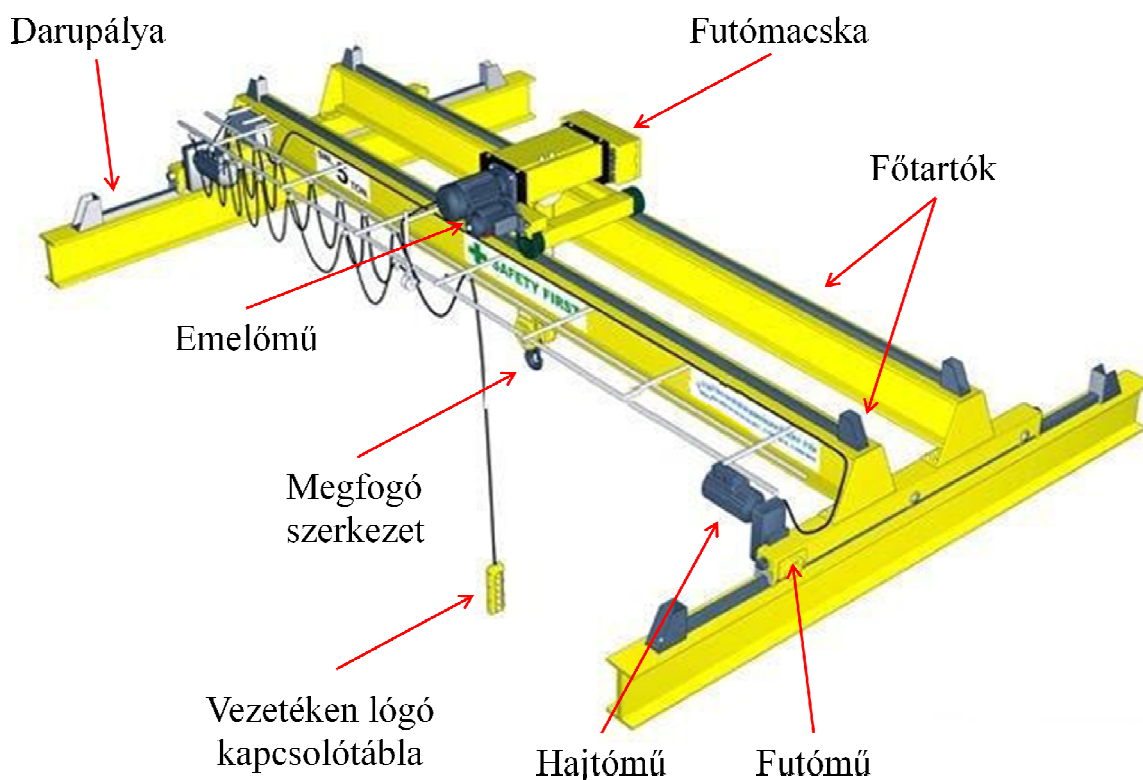
A futódaru a modern ipar egyik leggyakrabban használt emelő szerkezete. Nemcsak emelésre szolgál, a teher kis távolságú horizontális szállítására is alkalmas. Üzemcsarnokon belül általában nagy tömegű alapanyagok, félkész és késztermékek munkahelyek közötti szállítására használják. Kültéri változatát elsősorban logisztikai központokban (például kikötőkben, vasútállomásokon), és nagyobb telephelyeken alkalmazzák. A daru hídszerkezete vasbeton oszlopokra, vagy az üzemcsarnok megerősített falaira épített, magasan elhelyezett darupályán, síneken mozog. A futódaru egyik legnagyobb előnye, hogy szerkezetének nagy része a magasban van. Viszonylag keveset foglal el a hasznos térből, lehetővé teszi a maximális térkihasználtságot.



61. ábra: Futódaru működés közben egy üzemcsarnokban

A daru hídszerkezetének legfontosabb elemei a főtartók, a futómacska, valamint a darupályán való mozgást lehetővé tevő futómű és hajtómű. A főtartók számát tekintve az egy főtartós és két főtartós konstrukciók a leggyakoribbak. A főtartók felépítése lehet zárt szekrényszelvényes, vagy rácsos szerkezetű. A rácsos szerkezetnek többek között kültéri használatnál vannak előnyei, ahol jelentős mértékű szélterhelés léphet fel. A főtartók anyaga elsősorban acél, de speciális esetben akár alumíniumból is készíthetik. A főtartón

(vagy főtartókon) elhelyezett sínen, a darupályára merőlegesen mozog a futómacska. A futómacska lényegében egy kerekekkel ellátott, sínen mozgó kocsi, aminek saját mozgatóműve és fékberendezése van. A teher emelését a futómacskára szerelt emelőmű végzi. Az emelőmű lényeges elemei a megfogó szerkezet, meghajtás, fék és kötéldob. A futódaru áramellátását legtöbbször görgős vagy csúszós áramszedő, vagy flexibilis gumitömlő kábel biztosítja. A daru irányítása lehet automatizált, vagy emberi erőforrást igénylő. Utóbbi esetben a kezelés történhet vezetéken lógó, vagy vezeték nélküli kapcsolótáblával a talajon kísérve, vagy a hídszerkezetre szerelt kezelőfülkéből (62. ábra).



62. ábra: Futódaru szerkezeti felépítése

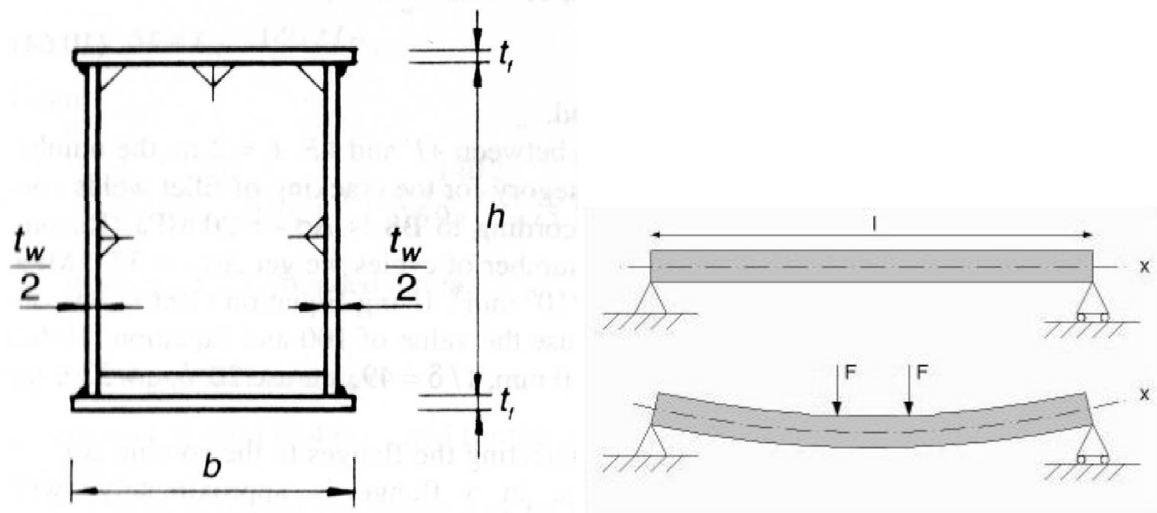
Dolgozatomban acélból készült, hegesztett négyszög keresztmetszetű, szekrényszelvényes, két főtartós futódarut vizsgáltam. A futódaru tervezése során az egyik legfontosabb kérdés a főtartók méretezése. Az üzemeltetési és gyártási költségek annál alacsonyabbak lesznek, minél inkább minimalizáljuk a főtartók önsúlyát. Ugyanakkor elég erősnek kell maradnia a szerkezetnek ahhoz, hogy a daru a különböző terhelések mellett is egész élettartama során biztosítsa névleges emelési teljesítményét. A futódaru főtartó optimalizálása egy nemlineáris, feltételes optimalizációs feladat, aminek megoldására a dolgozatban ismertetett heurisztikus algoritmusokat fogom használni.

7.1 Az optimalálási feladat matematikai leírása [48]

Az optimalálási feladat megoldásához mindenképpen szükséges annak matematikai leírása. Meg kell határozni a változókat, a különböző feltételeket, valamint a célfüggvényt.

7.1.1 Döntési változók meghatározása

A futódaru főtartói két oldalt alátámasztott, hegesztett, négyszög keresztmetszetű zárt szekrényszelvények. A főtartó méreteit négy változó írja le, melyek sorrendben $h, \frac{t_w}{2}, b, t_f$ (63. ábra).



63. ábra: Az optimalálási feladat döntési változói

7.1.2 Feltételek meghatározása

7.1.2.1 Méretkorlátok

A négy döntési változónak célszerű minimum és maximum méretkorlátot (F1, F2, F3, F4 feltételek) definiálni:

$$4 \text{ mm} \leq \frac{t_w}{2} \leq 50 \text{ mm}$$

$$4 \text{ mm} \leq t_f \leq 50 \text{ mm}$$

$$300 \text{ mm} \leq h \leq 2000 \text{ mm}$$

$$300 \text{ mm} \leq b \leq 1200 \text{ mm}$$

7.1.2.2 Fáradás

A fáradási (fatigue) feltétel (F5):

$$\Delta\sigma = \frac{M_q}{4W_x} \leq \frac{\Delta\sigma_n}{\gamma_{Mf}} \quad (29)$$

$$M_q = \psi_d \frac{F}{2L} \left(L - \frac{k}{2} \right)^2 \quad (30)$$

$$W_x = \frac{2I_x}{(h+t_f)} \quad (31)$$

$$I_x = \frac{h^3 t_w}{12} + 2b \frac{t_f^3}{12} + 2bt_f \left(\frac{h}{2} + \frac{t_f}{2} \right)^2 \quad (32)$$

Ahol:

- M_q : Változó terhelés
- ψ_d : Dinamikus tényező
- F : A teher súlya
- L : A futódaru főtartójának fesztávolsága
- k : A futómacska hossza
- W_x : Elastic section modulus
- I_x : Moment of inertia
- $\Delta\sigma_n$: Fáradási stressz N számú ciklus után.
- γ_{Mf} : Biztonsági tényező

7.1.2.3 Statikus terhelés

A statikus terhelés (static stress) feltétel (F6):

$$\sigma_{max} = \frac{M_s}{W_x} \leq \frac{f_y}{\gamma_{M1}} \quad (33)$$

$$M_s = \frac{\gamma_G L^2}{8} (A\rho g + g_j + g_s) + \gamma_G \frac{G_1 L}{4} \quad (34)$$

Ahol:

- M_s : Statikus terhelés
- f_y : Folyáshatár
- γ_{M1} : Biztonsági tényező
- γ_G : Biztonsági tényező
- A : Keresztmetszet, $A = ht_w + 2bt_f$
- ρ : Az anyag sűrűsége
- g : gravitációs gyorsulás
- g_j : A hídstruktúrára lévő járda terhelése
- g_s : A hídstruktúrára lévő sín terhelése
- G_1 : A hídstruktúrát mozgató gépészeti (futómű, hajtómű) súlya

7.1.2.4 Gerinc horpadás

A gerinc horpadás (flange buckling) feltétel (F7):

$$\frac{(b-40)}{t_f} \leq \frac{1}{\delta} \quad (35)$$

$$\delta = \frac{1}{42\varepsilon} \quad (36)$$

$$\varepsilon = \sqrt{\frac{235}{f_y}} \quad (37)$$

7.1.2.5 Öv horpadás

Az öv horpadás (web buckling) feltétel (F8):

$$\frac{2h}{t_w} \leq \frac{1}{\beta} \quad (38)$$

$$\beta = \frac{1}{124\varepsilon} \quad (39)$$

$$\varepsilon = \sqrt{\frac{235}{f_y}} \quad (40)$$

7.1.2.6 Lehajlás

A lehajlás (deflection) feltétel (F9):

$$\frac{F(L-k)}{48E_S I_x} (3L^2 - (L-k)^2) \leq \frac{L}{600} \quad (41)$$

Ahol:

- E_S : Az acél rugalmassági tényezője

7.1.3 Célfüggvény

Az optimalizálás során a cél a fűtő tartó tömegminimumának meghatározása. Egycélfüggvényes optimalizálásról van szó, ahol a célfüggvény:

$$m = AL\rho \quad (42)$$

7.2 Az optimalálási feladat megoldása

Az optimalálási feladat megoldásához a dolgozatban ismertetett heurisztikus algoritmusokat fogom használni. Az algoritmusok a 2. táblázat és a 14. táblázat bemenő paramétereivel futnak. A kétváltozós algoritmusokkal szemben ebben a szerkezetoptimalálási feladatban 4 döntési változó van, ezért az algoritmusok 500 helyett 1000 iteráción át keresnek. A keresések eredményéről táblázatot készítek, ami algoritmusonként 100 Monte Carlo futás legjobb fitness értékű megoldásának adatait tartalmazza. Statisztikai táblázatot és konvergencia táblázatot ebben a fejezetben nem készítek, mivel most nem elsősorban az algoritmusok karakterisztikája érdekel, hanem az optimalálási feladat eredménye. A feltételek betartásáról egy általam készített büntetőfüggvény gondoskodik. Az alábbi paraméterek adottak:

- ψ_d : 2
- F : 240000 N
- L : 20000 mm
- k : 1900 mm
- $\Delta\sigma_n$: 69,8 MPa
- γ_{Mf} : 1,35
- f_y : Fe 360 acél, 235 $\frac{\text{N}}{\text{mm}^2}$
- γ_{M1} : 1,1
- γ_G : 1,35
- ρ : 7,85 $\frac{\text{kg}}{\text{dm}^3}$
- g : 9,81 $\frac{\text{m}}{\text{s}^2}$
- g_j : 1 $\frac{\text{N}}{\text{mm}}$
- g_s : 0,2 $\frac{\text{N}}{\text{mm}}$
- G_1 : 30 kN
- E_s : $2,1 \times 10^5 \frac{\text{N}}{\text{mm}^2}$

7.2.1 C# forráskód

A futódaru főtartó optimalálási feladat célfüggvényét és büntetőfüggvényét C# nyelven készítettem el. A kód terjedelme miatt a 2. számú mellékletben található, vagy letölthető az alábbi linkről:

<https://drive.google.com/file/d/0BxE6yHbGFZBANDIDWmRmMDJxVVE/edit?usp=sharing>

	h	$\frac{t_w}{2}$	b	t_f	Keresztmetszet (A)	Tömeg (m)
AiNet	1194,13	19,26	663,87	18,80	47958,91	7529,55
BFOA	1216,08	19,62	709,15	16,97	47930,24	7525,05
BA	1241,23	20,16	568,29	20,33	48140,00	7557,98
CA	1227,34	19,81	509,50	23,28	48033,16	7541,21
DE	1197,68	19,32	680,18	18,27	48001,78	7536,28
HS	1268,93	20,47	376,68	29,48	48182,79	7564,70
KH	1211,62	19,58	468,66	26,41	48480,25	7611,40
MA	1232,37	19,88	622,08	18,86	47960,73	7529,84
NM	1214,97	19,60	674,83	17,88	47936,23	7525,99
PSO	1209,95	19,52	700,03	17,37	47930,25	7525,05
RS	1260,02	20,44	516,44	21,73	48202,47	7567,79
SA	1202,95	19,40	708,92	17,35	47933,08	7525,49

18. táblázat: Az optimalizációs feladat megoldása

Az optimalizációs feladatra a legjobb megoldást 100 Monte Carlo futás adatai alapján a A Bacterial Foraging (BFOA) algoritmus és a Particle Swarm Optimization (PSO) adta. Megállapítható, hogy valamennyi kereső eljárás optimum-közeli megoldást talált, ami a statikus terhelés, gerinc horpadás és fáradás feltételek értékeiből is látszik (19. táblázat, 20. táblázat). Nincs nagy eltérés az eredmények között, ami azt jelzi, hogy minden algoritmus megfelelően működött. A szekrényszelvény ideális méretezése A BFOA szerint:

$$h = 1216,08 \text{ mm}$$

$$\frac{t_w}{2} = 19,62 \text{ mm}$$

$$b = 709,15 \text{ mm}$$

$$t_f = 16,97 \text{ mm}$$

A méretek alapján a futódaru főtartó tömegminimuma:

$$m = 7525,05 \text{ kg}$$

A szekrényszelvény ideális méretezése A PSO szerint:

$$h = 1209,95 \text{ mm}$$

$$\frac{t_w}{2} = 19,52 \text{ mm}$$

$$b = 700,03 \text{ mm}$$

$$t_f = 17,37 \text{ mm}$$

A méretek alapján a futódaru főtartó tömegminimuma:

$$m = 7525,05 \text{ kg}$$

	Fáradás	Fáradás (max)	Statikus terhelés	Statikus terhelés (max)	Gerinc horpadás	Gerinc horpadás (max)
AiNet	56,28	66,48	213,64	213,64	35,31	42,00
BFOA	56,28	66,48	213,63	213,64	41,78	42,00
BA	56,23	66,48	213,50	213,64	27,95	42,00
CA	56,24	66,48	213,53	213,64	21,88	42,00
DE	56,20	66,48	213,34	213,64	37,22	42,00
HS	56,26	66,48	213,63	213,64	12,78	42,00
KH	55,56	66,48	211,04	213,64	17,75	42,00
MA	56,28	66,48	213,64	213,64	32,98	42,00
NM	56,28	66,48	213,64	213,64	37,75	42,00
PSO	56,28	66,48	213,64	213,64	40,30	42,00
RS	56,19	66,48	213,36	213,64	23,76	42,00
SA	56,28	66,48	213,64	213,64	40,87	42,00

19. táblázat: A megoldás táblázatban lévő méretekhez tartozó feltételek 1. része

	Öv horpadás	Öv horpadás (max)	Lehajlás	Lehajlás (max)
AiNet	124,00	124,00	31,56	66,67
BFOA	123,97	124,00	31,09	66,67
BA	123,11	124,00	30,30	66,67
CA	123,94	124,00	30,49	66,67
DE	123,96	124,00	31,45	66,67
HS	124,00	124,00	29,26	66,67
KH	123,74	124,00	30,35	66,67
MA	124,00	124,00	30,60	66,67
NM	124,00	124,00	31,08	66,67
PSO	124,00	124,00	31,23	66,67
RS	123,29	124,00	29,77	66,67
SA	124,00	124,00	31,41	66,67

20. táblázat: A megoldás táblázatban lévő méretekhez tartozó feltételek 2. része

A heurisztikus algoritmusok által szolgáltatott eredmények ellenőrzésére a szerkezetoptimalizációs feladatot Nyers erő (Brute force) módszerrel is megoldottam, ami az összes lehetséges megoldás vizsgálatát jelenti. Hogy belátható időn belül megoldást kapjak, a folytonos változók helyett csak diszkrét értékeket használtam. A 3.237.480.000 lehetséges méretezés vizsgálata majdnem 70 percig (4149129.594 msec) tartott, az optimális megoldás pedig

$$h = 1240 \text{ mm}, \frac{t_w}{2} = 20 \text{ mm}, b = 681 \text{ mm}, t_f = 17 \text{ mm}, A = 47954 \text{ mm}^2, m = 7528.778 \text{ kg}$$

8 Összefoglalás

Dolgozatomban a heurisztikus optimáló algoritmusok működését, és a használatukban rejlő sokszínű lehetőségeket vizsgáltam. Véleményem szerint a tudomány és technológia fejlődésével számos új területen merül majd fel az igény a nagy bonyolultságú, hagyományos eszközökkel megoldhatatlannak tűnő számítási feladatok elvégzésére. Ez az a problémakör, amiben a metaheurisztikus optimáló algoritmusok a leginkább erősek. Összefoglalásképpen dolgozatom főbb eredményei:

Összesen 12 algoritmus elméleti hátterét vizsgáltam, megvalósítottam őket C# programnyelven, forráskódjukat online elérhetővé tettem. Hatékonyságvizsgálatukat először 10 darab, szakirodalomból vett kétváltozós tesztfüggvény segítségével végeztem. Minden függvény esetében készítettem konvergencia grafikonokat és statisztikai táblázatot az eredményekről. Az algoritmusok teljesítménye alapján ajánlást tettem a bemenő paraméterek javítására.

Kidolgoztam egy szoftvert, amivel új, komplex tesztfüggvényeket lehet építeni. A szoftver gyakorlatilag végtelen variációs lehetőséget kínál tetszőleges bonyolultságú tesztfüggvények létrehozására. Kiemelkedő funkciója, hogy a fejlett felhasználói felület segítségével programozói tudás nélkül, másodpercek alatt elkészíthetők a komplex függvények, ezzel egy időben a program automatikusan létrehozza a függvény C# forráskódját. Elvégeztem az algoritmusok hatékonyságvizsgálatát egy általam létrehozott komplex tesztfüggvény segítségével. A teszt során a különböző súlyozási módszerek hatásait is dokumentáltam.

Matematikailag megfogalmaztam, és megoldottam egy szerkezetoptimalizási problémát, ahol a cél egy futódaru főtartójának tömeg-minimalizálása volt. Az általam javasolt bemenő paraméterekkel az algoritmusok nagyon jó megoldásokat adtak, a legtöbb feltételt maximálisan kihasználták.

Céлом a későbbiekben további heurisztikus algoritmusok közzététele és teljesítményvizsgálata, lehetőleg minél több, saját készítésű tesztfüggvény segítségével. A tesztadatok alapján tovább fogom vizsgálni a paraméterek kérdését, illetve a hibrid algoritmusok létrehozásának lehetőségeit. Szeretném a tesztfüggvény generátort webes alkalmazás formájában bárki számára hozzáférhetővé tenni, aki érdeklődik a témában. Végezetül pedig az algoritmus gyűjteménnyel további optimalizációs feladatok megoldását tűztam ki célul.

9 1. számú melléklet, az algoritmusok C# forráskódja

9.1 Artificial Immune Network (AiNet) C# forráskódja

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ArtificialImmuneNetwork
{
    class Program
    {
        static Random random = new Random();
        static void Main(string[] args)
        {
            const int problem_size = 2;
            double[,] search_space = new double[problem_size, 2] { { -5, 5 }, { -5, 5 } };

            int max_gens = 200;
            int pop_size = 50;
            int num_clones = 20;
            int beta = 100;
            int num_rand = 5;
            double aff_tresh = (search_space[0, 1] - search_space[0, 0]) * 0.05;
            double best = search("", problem_size, search_space, max_gens, pop_size, num_clones, beta, num_rand, aff_tresh);
            Console.ReadLine();
        }

        public static double search(string test_function, int problem_size, double[,] search_space, int max_gens, int pop_size, int num_clones, int beta, int num_rand, double aff_tresh)
        {
            ImmuneCell[] population = new ImmuneCell[pop_size];
            for (int i = 0; i < pop_size; i++)
            {
                double[] randomPosition = random_vector(problem_size, search_space);
                double fitness = objective_function(test_function, randomPosition);
                population[i] = new ImmuneCell(randomPosition, fitness, 0.0);
            }
            ImmuneCell best = null;
            for (int i = 0; i < max_gens; i++)
            {
                for (int j = 0; j < population.Length; j++)
                {
                    population[j].fitness = objective_function(test_function, population[j].position);
                }
                population = calculate_normalized_cost(population);
                Array.Sort(population);
                if (best == null || population[0].fitness < best.fitness)
                {
                    best = population[0];
                }
                double avgCost = average_cost(population);
            }
        }
    }
}
```

```
ImmuneCell[] progeny = new ImmuneCell[population.Length];
double progenyAvgCost;
do
{
    for (int j = 0; j < progeny.Length; j++)
    {
        progeny[j] = clone_cell(test_function, beta, num_clones,
population[j]);
    }
    progenyAvgCost = average_cost(progeny);
} while (progenyAvgCost > avgCost);
population = affinity_supress(progeny, aff_tresh);
List<ImmuneCell> listPopulation = new List<ImmuneCell>();
listPopulation = population.ToList();
for (int j = 0; j < num_rand; j++)
{
    double[] randomPosition = random_vector(problem_size,
search_space);
    double fitness = objective_function(test_function,
randomPosition);
    listPopulation.Add(new ImmuneCell(randomPosition, fitness, 0.0));
}
population = listPopulation.ToArray();
Console.WriteLine("" + i + " .generáció, Legjobb fitness:" +
best.fitness);
}
return 0.0;
}

public static ImmuneCell[] calculate_normalized_cost(ImmuneCell[] population)
{
    Array.Sort(population);
    double range = population[population.Length - 1].fitness -
population[0].fitness;
    if (range == 0.0)
    {
        foreach (ImmuneCell i in population)
        {
            i.normalizedFitness = 1.0;
        }
    }
    else
    {
        foreach (ImmuneCell i in population)
        {
            i.normalizedFitness = 1.0 - (i.normalizedFitness / range);
        }
    }
    return population;
}

public static double average_cost(ImmuneCell[] population)
{
    double sum = 0.0;
    foreach (ImmuneCell i in population)
    {
        sum += i.fitness;
    }
    return sum / (float)population.Length;
}
```

```
public static ImmuneCell clone_cell(string test_function, int beta, int
num_clones, ImmuneCell parent)
{
    ImmuneCell[] clones = new ImmuneCell[num_clones];
    for (int i = 0; i < clones.Length; i++)
    {
        clones[i] = new ImmuneCell(parent.position, parent.fitness,
parent.normalizedFitness);
    }
    ImmuneCell[] mutatedClones = new ImmuneCell[num_clones];
    for (int i = 0; i < clones.Length; i++)
    {
        mutatedClones[i] = mutate(test_function, beta, clones[i],
parent.normalizedFitness);
    }
    Array.Sort(mutatedClones);
    return mutatedClones[0];
}

public static ImmuneCell mutate(string test_function, int beta, ImmuneCell child,
double normalized_cost)
{
    double[] mutatedPosition = new double[child.position.Length];
    for (int i = 0; i < child.position.Length; i++)
    {
        double alpha = mutation_rate(beta, normalized_cost);
        mutatedPosition[i] = child.position[i] + alpha * random_gaussian(0.0,
1.0);
    }
    double mutatedFitness = objective_function(test_function, mutatedPosition);
    ImmuneCell mutatedCell = new ImmuneCell(mutatedPosition, mutatedFitness,
child.normalizedFitness);
    return mutatedCell;
}

public static double mutation_rate(int beta, double normalized_cost)
{
    return (1.0 / beta) * Math.Exp(-normalized_cost);
}

public static double random_gaussian(double mean, double stdev)
{
    double u1 = 0.0;
    double u2 = 0.0;
    double w = 0.0;
    do
    {
        u1 = 2 * random.NextDouble() - 1;
        u2 = 2 * random.NextDouble() - 1;
        w = u1 * u1 + u2 * u2;
    }
    while (w >= 1);
    w = Math.Sqrt((-2.0 * Math.Log(w)) / w);
    return mean + (u2 * w) * stdev;
}

public static ImmuneCell[] affinity_suppress(ImmuneCell[] progeny, double
aff_tresh)
```

```
{
    List<ImmuneCell> pop = new List<ImmuneCell>();
    for (int i = 0; i < progeny.Length; i++)
    {
        ImmuneCell[] neighbors = get_neighborhood(progeny[i], progeny, aff_tresh);
        Array.Sort(neighbors);
        if (neighbors.Length == 0 || progeny[i] == neighbors[0])
        {
            pop.Add(progeny[i]);
        }
    }
    return pop.ToArray<ImmuneCell>();
}

public static ImmuneCell[] get_neighborhood(ImmuneCell cell, ImmuneCell[] pop,
double aff_tresh)
{
    List<ImmuneCell> listNeighbors = new List<ImmuneCell>();
    for (int i = 0; i < pop.Length; i++)
    {
        if (distance(pop[i].position, cell.position) < aff_tresh)
        {
            listNeighbors.Add(pop[i]);
        }
    }
    return listNeighbors.ToArray<ImmuneCell>();
}

public static double distance(double[] c1, double[] c2)
{
    double sum = 0.0;
    for (int i = 0; i < c1.Length; i++)
    {
        sum += Math.Pow((c1[i] - c2[i]), 2);
    }
    return Math.Sqrt(sum);
}

public static double objective_function(string test_function, double[] xy)
{
    double cost = 0;
    cost = 10 * 2 + (Math.Pow(xy[0], 2) - 10 * Math.Cos(2 * Math.PI * xy[0])) +
(Math.Pow(xy[1], 2) - 10 * Math.Cos(2 * Math.PI * xy[1]));
    return cost;
}

public static double random_in_bounds(double min, double max)
{
    return min + ((max - min) * random.NextDouble());
}

public static double[] random_vector(int problem_size, double[,] search_space)
{
    double[] randomPosition = new double[problem_size];
    for (int i = 0; i < randomPosition.Length; i++)
    {
        double lo = search_space[i, 0];
```

```
        double hi = search_space[i, 1];
        randomPosition[i] = (hi - lo) * random.NextDouble() + lo;
    }
    return randomPosition;
}

public class ImmuneCell : IComparable<ImmuneCell>
{
    public double[] position;
    public double fitness;
    public double normalizedFitness;
    public ImmuneCell(double[] position, double fitness, double
normalizedFitness)
    {
        this.position = new double[position.Length];
        position.CopyTo(this.position, 0);
        this.fitness = fitness;
        this.normalizedFitness = normalizedFitness;
    }
    public int CompareTo(ImmuneCell other)
    {
        if (this.fitness < other.fitness)
            return -1;
        else if (this.fitness > other.fitness)
            return 1;
        else
            return 0;
    }
}
}
```

A forráskód letölthető az alábbi linkről:

<https://drive.google.com/file/d/0BxE6yHbGFZBAazFBLWc4cFZGd00/edit?usp=sharing>

9.2 Bacterial Foraging (BFOA) C# forráskódja

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace BacterialForaging
{
    class Program
    {
        static Random random = null;
        static void Main(string[] args)
        {
            const int problem_size = 2;
            double[,] search_space = new double[problem_size, 2] { { -5.12, 5.12 }, {
-5.12, 5.12 } };
            int S = 100; // colony size
            int Nc = 20; // chemotactic steps, index = j
            int Ns = 5; // maximum swim steps, index = m
            int Nre = 8; // reproduction steps, index = k
            int Ned = 4; // dispersal steps, index = l
            double Ped = 0.25; // probability of dispersal
            double Ci = 0.05; // basic bacteria movement increment size (for all
bacteria)
            double[] best = search(problem_size, search_space, S, Nc, Ns, Nre, Ned,
Ped, Ci);
            Console.ReadLine();
        }

        private static double[] search(int problem_size, double[,] search_space, int
S, int Nc, int Ns, int Nre, int Ned, double Ped, double Ci)
        {
            random = new Random();
            try
            {
                Colony colony = new Colony(S, problem_size, search_space);
                for (int i = 0; i < S; ++i)
                {
                    double cost = Cost(colony.bacteria[i].position);
                    colony.bacteria[i].cost = cost;
                    colony.bacteria[i].prevCost = cost;
                }
                double bestCost = colony.bacteria[0].cost;
                int indexOfBest = 0;
                for (int i = 0; i < S; ++i) { if (colony.bacteria[i].cost < bestCost)
{ bestCost = colony.bacteria[i].cost; indexOfBest = i; } }
                double[] bestPosition = new double[problem_size];
                colony.bacteria[indexOfBest].position.CopyTo(bestPosition, 0);
                for (int l = 0; l < Ned; ++l) // eliminate-disperse loop
                {
                    for (int k = 0; k < Nre; ++k) // reproduce-eliminate loop
                    {
                        for (int j = 0; j < Nc; ++j) // chemotactic loop; the lifespan
of each bacterium
                        {
```

```

    0.0; }
        for (int i = 0; i < S; ++i) { colony.bacteria[i].health =
        for (int i = 0; i < S; ++i)
        {
            double[] tumble = new double[problem_size];
            for (int p = 0; p < problem_size; ++p) { tumble[p] =
2.0 * random.NextDouble() - 1.0; }
            double rootProduct = 0.0;
            for (int p = 0; p < problem_size; ++p) { rootProduct
+= (tumble[p] * tumble[p]); }
            for (int p = 0; p < problem_size; ++p) {
colony.bacteria[i].position[p] += (Ci * tumble[p]) / rootProduct; }
            colony.bacteria[i].prevCost = colony.bacteria[i].cost;
            colony.bacteria[i].cost =
Cost(colony.bacteria[i].position);
            colony.bacteria[i].health += colony.bacteria[i].cost;
            if (colony.bacteria[i].cost < bestCost)
            {
                bestCost = colony.bacteria[i].cost;
                colony.bacteria[i].position.CopyTo(bestPosition,
0);
            }
            int m = 0;
            while (m < Ns && colony.bacteria[i].cost <
colony.bacteria[i].prevCost)
            {
                ++m;
                for (int p = 0; p < problem_size; ++p) {
colony.bacteria[i].position[p] += (Ci * tumble[p]) / rootProduct; }
                colony.bacteria[i].prevCost =
colony.bacteria[i].cost;
                colony.bacteria[i].cost =
Cost(colony.bacteria[i].position);
                if (colony.bacteria[i].cost < bestCost)
                {
                    bestCost = colony.bacteria[i].cost;
colony.bacteria[i].position.CopyTo(bestPosition, 0);
                }
            }
        }
        Console.WriteLine(((l * Nre * Nc) + k * Nc + j) + ".
iteráció. Legjobb fitness= " + bestCost);
    }
    Array.Sort(colony.bacteria);
    for (int left = 0; left < S / 2; ++left)
    {
        int right = left + S / 2;

        colony.bacteria[left].position.CopyTo(colony.bacteria[right].position, 0);
        colony.bacteria[right].cost = colony.bacteria[left].cost;
        colony.bacteria[right].prevCost =
colony.bacteria[left].prevCost;
        colony.bacteria[right].health =
colony.bacteria[left].health;
    }
    for (int i = 0; i < S; ++i)
    {
        double probab = random.NextDouble();
        if (probab < Ped)

```

```

        {
            for (int p = 0; p < problem_size; ++p)
            {
                double x = (search_space[p, 1] - search_space[p, 0]) *
random.NextDouble() + search_space[p, 0];
                colony.bacteria[i].position[p] = x;
            }
            double cost = Cost(colony.bacteria[i].position);
            colony.bacteria[i].cost = cost;
            colony.bacteria[i].prevCost = cost;
            colony.bacteria[i].health = 0.0;
            if (colony.bacteria[i].cost < bestCost)
            {
                bestCost = colony.bacteria[i].cost;
                colony.bacteria[i].position.CopyTo(bestPosition, 0);
            }
        }
    }
    double[] solution = new double[] { bestPosition[0], bestPosition[1],
bestCost };
    return solution;
}
catch (Exception ex)
{
    Console.WriteLine("Fatal: " + ex.Message);
    return null;
}
}

static double Cost(double[] position)
{
    // Rastrigin function.  $f(x,y) = (x^2 - 10 \cdot \cos(2 \cdot \pi \cdot x)) + (y^2 - 10 \cdot \cos(2 \cdot \pi \cdot y)) + 20$ 
    double result = 0.0;
    for (int i = 0; i < position.Length; ++i)
    {
        double xi = position[i];
        result += (xi * xi) - (10 * Math.Cos(2 * Math.PI * xi)) + 10;
    }
    return result;
}

}

public class Colony
{
    public Bacterium[] bacteria;

    public Colony(int S, int problem_size, double[,] search_space)
    {
        this.bacteria = new Bacterium[S];
        for (int i = 0; i < S; ++i)
            bacteria[i] = new Bacterium(problem_size, search_space);
    }

    public override string ToString()
    {
        string s = "";
        for (int i = 0; i < this.bacteria.Length; ++i)
            s += "[" + i + "] : " + bacteria[i].ToString() + Environment.NewLine;
    }
}

```

```
        return s;
    }

    public class Bacterium : IComparable<Bacterium>
    {
        public double[] position;
        public double cost;
        public double prevCost;
        public double health;
        static Random random = new Random(0);
        public Bacterium(int problem_size, double[,] search_space)
        {
            this.position = new double[problem_size];
            for (int p = 0; p < problem_size; ++p)
            {
                double x = (search_space[p, 1] - search_space[p, 0]) *
random.NextDouble() + search_space[p, 0];
                this.position[p] = x;
            }
            this.health = 0.0;
        }

        public override string ToString()
        {
            string s = "[ ";
            for (int p = 0; p < this.position.Length; ++p)
                s += position[p].ToString("F2") + " ";
            s += "] ";
            s += "cost = " + this.cost.ToString("F2") + " prevCost = " +
this.prevCost.ToString("F2");
            s += " health = " + this.health.ToString("F2");
            return s;
        }

        public int CompareTo(Bacterium other)
        {
            if (this.health < other.health)
                return -1;
            else if (this.health > other.health)
                return 1;
            else
                return 0;
        }
    }
}
```

A forráskód letölthető az alábbi linkről:

<https://drive.google.com/file/d/0BxE6yHbGFZBAQ1p4dV85d21RUVE/edit?usp=sharing>

9.3 Bees Algorithm (BA) C# forráskódja

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace BeesAlgorithm
{
    class Program
    {
        static Random random = new Random();
        static void Main(string[] args)
        {
            const int problem_size = 2;
            double[,] search_space = new double[problem_size, 2] { { -5, 5 }, { -5, 5 } };

            int max_gens = 200;
            int num_bees = 50;
            int num_sites = 3;
            int elite_sites = 1;
            double patch_size = 3.0;
            int e_bees = 7;
            int o_bees = 2;
            double best = search("", problem_size, search_space, max_gens, num_bees, num_sites, elite_sites, patch_size, e_bees, o_bees);
            Console.ReadLine();
        }

        public static double search(string test_function, int problem_size, double[,] search_space, int max_gens, int num_bees, int num_sites, int elite_sites, double patch_size, int e_bees, int o_bees)
        {
            Bee best = null;
            Bee[] population = new Bee[num_bees];
            for (int i = 0; i < population.Length; i++)
            {
                double[] position = random_vector(problem_size, search_space);
                double fitness = objective_function(test_function, position);
                population[i] = new Bee(position, fitness);
            }
            Array.Sort(population);
            best = population[0];
            for (int i = 0; i < max_gens; i++)
            {
                Bee[] next_gen = new Bee[num_sites];
                for (int j = 0; j < num_sites; j++)
                {
                    int neigh_size = (i < elite_sites) ? e_bees : o_bees;
                    next_gen[j] = search_neigh(test_function, population[j], neigh_size, patch_size, search_space);
                    population[j] = next_gen[j];
                }
                for (int j = num_sites; j < num_bees; j++)
                {
                    population[j] = create_scout_bee(test_function, problem_size, search_space);
                }
            }
        }
    }
}
```

```

    }
    Array.Sort(population);
    if (population[0].fitness < best.fitness)
    {
        best = population[0];
    }
    patch_size = patch_size * 0.95;
    Console.WriteLine("'" + i + " .generáció, Legjobb fitness:" +
best.fitness);
    }
    return 0.0;
}

public static Bee search_neigh(string test_function, Bee parent, int
neigh_size, double patch_size, double[,] search_space)
{
    Bee[] neigh = new Bee[neigh_size];
    for (int i = 0; i < neigh_size; i++)
    {
        neigh[i] = create_neigh_bee(test_function, parent.position,
patch_size, search_space);
    }
    Array.Sort(neigh);
    return neigh[0];
}

public static Bee create_neigh_bee(string test_function, double[] position,
double patch_size, double[,] search_space)
{
    double[] vector = new double[position.Length];
    for (int i = 0; i < position.Length; i++)
    {
        double v = 0.0;
        v = (random.NextDouble() < 0.5) ? v + random.NextDouble() * patch_size
: v - random.NextDouble() * patch_size;
        if (v < search_space[i, 0])
        {
            v = search_space[i, 0];
        }
        else if (v > search_space[i, 1])
        {
            v = search_space[i, 1];
        }
        vector[i] = v;
    }
    double fitness = objective_function(test_function, vector);
    Bee neighBee = new Bee(vector, fitness);
    return neighBee;
}

public static Bee create_scout_bee(string test_function, int problem_size,
double[,] search_space)
{
    double[] position = random_vector(problem_size, search_space);
    double fitness = objective_function(test_function, position);
    Bee scoutBee = new Bee(position, fitness);
    return scoutBee;
}

public static double objective_function(string test_function, double[] xy)
{

```

```
double cost = 0;
cost = 10 * 2 + (Math.Pow(xy[0], 2) - 10 * Math.Cos(2 * Math.PI * xy[0]))
+ (Math.Pow(xy[1], 2) - 10 * Math.Cos(2 * Math.PI * xy[1]));
return cost;
}

public static double random_in_bounds(double min, double max)
{
    return min + ((max - min) * random.NextDouble());
}

public static double[] random_vector(int problem_size, double[,] search_space)
{
    double[] randomPosition = new double[problem_size];
    for (int i = 0; i < randomPosition.Length; i++)
    {
        double lo = search_space[i, 0];
        double hi = search_space[i, 1];
        randomPosition[i] = (hi - lo) * random.NextDouble() + lo;
    }
    return randomPosition;
}

public class Bee : IComparable<Bee>
{
    public double[] position;
    public double fitness;
    public Bee(double[] position, double fitness)
    {
        this.position = new double[position.Length];
        position.CopyTo(this.position, 0);
        this.fitness = fitness;
    }

    public int CompareTo(Bee other) // based on vector/solution value
    {
        if (this.fitness < other.fitness)
            return -1;
        else if (this.fitness > other.fitness)
            return 1;
        else
            return 0;
    }
}
}
```

A forráskód letölthető az alábbi linkről:

<https://drive.google.com/file/d/0BxE6yHbGFZBANUZHTzdfRnJ4aVE/edit?usp=sharing>

9.4 Cultural Algorithm (CA) C# forráskódja

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CulturalAlgorithm
{
    class Program
    {
        static Random random = new Random();
        static void Main(string[] args)
        {
            const int problem_size = 2;
            double[,] search_space = new double[problem_size, 2] { { -5, 5 }, { -5, 5 } };

            int max_gens = 200;
            int pop_size = 200;
            int num_accepted = (int)(pop_size * 0.20);
            double best = search("", problem_size, search_space, max_gens, pop_size, num_accepted);
            Console.ReadLine();
        }

        public static double search(string test_function, int problem_size, double[,] search_space, int max_gens, int pop_size, int num_accepted)
        {
            BeliefSpace beliefSpace = initialize_beliefspace(search_space);
            Individual[] population = new Individual[pop_size];
            for (int i = 0; i < pop_size; i++)
            {
                double[] randomPosition = random_vector(problem_size, search_space);
                double fitness = objective_function(test_function, randomPosition);
                population[i] = new Individual(randomPosition, fitness);
            }
            Array.Sort(population);
            Individual best = population[0];
            beliefSpace = update_beliefspace_situational(beliefSpace, best);
            for (int i = 0; i < max_gens; i++)
            {
                Individual[] children = new Individual[pop_size];
                for (int j = 0; j < pop_size; j++)
                {
                    double[] childrenPosition = mutate_with_inf(problem_size, beliefSpace, search_space);
                    double fitness = objective_function(test_function, childrenPosition);
                    children[j] = new Individual(childrenPosition, fitness);
                }
                Array.Sort(children);
                best = children[0];
                beliefSpace = update_beliefspace_situational(beliefSpace, best);
                Individual[] populationBuffer = new Individual[pop_size];
                for (int j = 0; j < populationBuffer.Length; j++)
                {
                    populationBuffer[j] = population[j];
                }
            }
        }
    }
}
```

```

    }
    for (int j = 0; j < population.Length; j++)
    {
        population[j] = binary_tournament(populationBuffer, children);
    }
    Array.Sort(population);
    Individual[] accepted = new Individual[num_accepted];
    for (int j = 0; j < num_accepted; j++)
    {
        accepted[j] = population[j];
    }
    beliefSpace = update_beliefSpace_normative(problem_size, beliefSpace,
accepted);
    Console.WriteLine("'" + i + " .generáció, Legjobb fitness:" +
best.fitness);
    }
    return 0.0;
}

public static BeliefSpace initialize_beliefSpace(double[,] search_space)
{
    BeliefSpace beliefSpace = new BeliefSpace(null, search_space);
    return beliefSpace;
}

public static BeliefSpace update_beliefSpace_situational(BeliefSpace
beliefSpace, Individual best)
{
    Individual curr_best = beliefSpace.situational;
    if (curr_best == null || best.fitness < curr_best.fitness)
    {
        beliefSpace.situational = best;
    }
    return beliefSpace;
}

public static BeliefSpace update_beliefSpace_normative(int problem_size,
BeliefSpace beliefSpace, Individual[] accepted)
{
    double[] accMin=new double[problem_size];
    double[] accMax=new double[problem_size];
    for (int i = 0; i < problem_size; i++)
    {
        accMin[i]=accepted[0].position[i];
        accMax[i]=accepted[0].position[i];
        for(int j=1;j<accepted.Length;j++){
            if(accepted[j].position[i]<accMin[i]){
                accMin[i]=accepted[j].position[i];
            }
            if(accepted[j].position[i]>accMax[i]){
                accMax[i]=accepted[j].position[i];
            }
        }
    }
    for (int i = 0; i < problem_size; i++)
    {
        beliefSpace.normative[i, 0] = accMin[i];
        beliefSpace.normative[i, 1] = accMax[i];
    }
    return beliefSpace;
}

```

```
public static double[] mutate_with_inf(int problem_size, BeliefSpace
beliefSpace, double[,] search_space)
{
    double[] newPosition = new double[problem_size];
    for (int i = 0; i < newPosition.Length; i++)
    {
        newPosition[i] = random_in_bounds(beliefSpace.normative[i, 0],
beliefSpace.normative[i, 1]);
        if (newPosition[i] < search_space[i, 0])
        {
            newPosition[i] = search_space[i, 0];
        }
        else if (newPosition[i] > search_space[i, 1])
        {
            newPosition[i] = search_space[i, 1];
        }
    }
    return newPosition;
}

public static Individual binary_tournament(Individual[] population,
Individual[] children)
{
    Individual[] everyone = new Individual[population.Length +
children.Length];
    for (int i = 0; i < children.Length; i++)
    {
        everyone[i] = children[i];
    }
    for (int i = children.Length; i < children.Length + population.Length;
i++)
    {
        everyone[i] = population[i-children.Length];
    }
    int randOne = random.Next(population.Length + children.Length);
    int randTwo = random.Next(population.Length + children.Length);

    while (randOne == randTwo)
    {
        randTwo = random.Next(population.Length + children.Length);
    }

    if (everyone[randOne].fitness < everyone[randTwo].fitness)
    {
        return everyone[randOne];
    }
    else
    {
        return everyone[randTwo];
    }
}

public static double objective_function(string test_function, double[] xy)
{
    double cost = 0;
    cost = 10 * 2 + (Math.Pow(xy[0], 2) - 10 * Math.Cos(2 * Math.PI * xy[0]))
+ (Math.Pow(xy[1], 2) - 10 * Math.Cos(2 * Math.PI * xy[1]));
    return cost;
}
```

```
public static double random_in_bounds(double min, double max)
{
    return min + ((max - min) * random.NextDouble());
}

public static double[] random_vector(int problem_size, double[,] search_space)
{
    double[] randomPosition = new double[problem_size];
    for (int i = 0; i < randomPosition.Length; i++)
    {
        double lo = search_space[i, 0];
        double hi = search_space[i, 1];
        randomPosition[i] = (hi - lo) * random.NextDouble() + lo;
    }
    return randomPosition;
}

public class BeliefSpace
{
    public Individual situational = null;
    public double[,] normative;
    public BeliefSpace(double[,] situational, double[,] normative)
    {
        this.normative = new double[normative.Length / 2, 2];
        Array.Copy(normative, this.normative, normative.Length);
    }
}

public class Individual : IComparable<Individual>
{
    public double[] position;
    public double fitness;
    public Individual(double[] position, double fitness)
    {
        this.position = new double[position.Length];
        position.CopyTo(this.position, 0);
        this.fitness = fitness;
    }
    public int CompareTo(Individual other) // based on vector/solution value
    {
        if (this.fitness < other.fitness)
            return -1;
        else if (this.fitness > other.fitness)
            return 1;
        else
            return 0;
    }
}
}
```

A forráskód letölthető az alábbi linkről:

<https://drive.google.com/file/d/0BxE6yHbGFZBAQnlOekgzMHZzMzQ/edit?usp=sharing>

9.5 Differential Evolution (DE) C# forráskódja

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DifferentialEvolution
{
    class Program
    {
        static void Main(string[] args)
        {
            const int problem_size = 2;
            double[,] search_space = new double[problem_size, 2] { { -5.12, 5.12 }, {
-5.12, 5.12 } };
            int max_gens = 100;
            int pop_size = 10 * problem_size;
            double weightf = 0.8;
            double crossf = 0.9;
            double[] best = search(problem_size, search_space, pop_size, max_gens,
weightf, crossf);
            Console.WriteLine("Keresés vége, Legjobb megoldás: x=" + best[0] + " y=" +
best[1] + " f(xy)=" + best[2]);
            Console.ReadLine();
        }

        private static double[] search(int problem_size, double[,] search_space, int
pop_size, int max_gens, double weightf, double crossf)
        {
            Population[] population;
            Random random = new Random();
            double bestFitness = Double.MaxValue;
            int bestPopulation = 0;
            Population[] children;
            population = new Population[pop_size];
            for (int i = 0; i < pop_size; i++)
            {
                double[] randomPosition = new double[problem_size];
                for (int j = 0; j < randomPosition.Length; j++)
                {
                    double lo = search_space[j, 0];
                    double hi = search_space[j, 1];
                    randomPosition[j] = (hi - lo) * random.NextDouble() + lo;
                }
                double cost = objective_function(randomPosition);
                population[i] = new Population(randomPosition, cost);
                if (population[i].fitness < bestFitness)
                {
                    bestFitness = population[i].fitness;
                    bestPopulation = i;
                }
            }
            children = new Population[pop_size];
            for (int i = 0; i < max_gens; i++)
            {
```

```

        for (int j = 0; j < pop_size; j++)
        {
            int p0 = j;
            int p1 = j;
            int p2 = j;
            int p3 = j;
            while (p1 == p0)
            {
                p1 = random.Next(population.Length - 1);
            }
            while ((p2 == p0) || (p2 == p1))
            {
                p2 = random.Next(population.Length - 1);
            }
            while ((p3 == p0) || (p3 == p1) || (p3 == p2))
            {
                p3 = random.Next(population.Length - 1);
            }
            int cut = random.Next(problem_size);
            Population sample = new Population(population[p0].position,
population[p0].fitness);
            for (int k = 0; k < problem_size; k++)
            {
                if (k == cut && random.NextDouble() < crossf)
                {
                    sample.position[k] = (population[p3].position[k] + weightf
* (population[p1].position[k] - population[p2].position[k]));
                }

                sample.fitness = objective_function(sample.position);
                children[j] = sample;
            }
            for (int j = 0; j < pop_size; j++)
            {
                if (children[j].fitness <= population[j].fitness)
                {
                    population[j] = children[j];
                }
            }
            for (int j = 0; j < pop_size; j++)
            {
                if (population[j].fitness < bestFitness)
                {
                    bestFitness = population[j].fitness;
                    bestPopulation = j;
                }
            }
            Console.WriteLine(i + ". iteráció, Legjobb fitness= " + bestFitness);
        }
        double[] solution = new double[] { population[bestPopulation].position[0],
population[bestPopulation].position[1], bestFitness };
        return solution;
    }

    static double objective_function(double[] x)
    {
        double cost = 0;
        cost = 10 * 2 + (Math.Pow(x[0], 2) - 10 * Math.Cos(2 * Math.PI * x[0])) +
(Math.Pow(x[1], 2) - 10 * Math.Cos(2 * Math.PI * x[1]));
        return cost;
    }

```

```
}  
  
class Population  
{  
    public double[] position;  
    public double fitness;  
    public Population(double[] position, double fitness)  
    {  
        this.position = new double[position.Length];  
        position.CopyTo(this.position, 0);  
        this.fitness = fitness;  
    }  
}  
}
```

A forráskód letölthető az alábbi linkről:

<https://drive.google.com/file/d/0BxE6yHbGFZBAY3RuQU1BRldMeGs/edit?usp=sharing>

9.6 Harmony Search (HS) C# forráskódja

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace HarmonySearch
{
    class Program
    {
        static Random random = new Random();
        static void Main(string[] args)
        {
            const int problem_size = 2;
            double[,] search_space = new double[problem_size, 2] { { -5.12, 5.12 }, {
-5.12, 5.12 } };
            int max_iter = 500;
            int mem_size = 20;
            double consid_rate = 0.95;
            double adjust_rate = 0.7;
            double range = 0.05;
            double[] best = search("Valami", problem_size, search_space, mem_size,
max_iter, consid_rate, adjust_rate, range);
            Console.WriteLine("Keresés vége. Legjobb megoldás: x=" + best[0] + " y=" +
best[1] + " f(xy)=" + best[2]);
            Console.ReadLine();
        }

        public static double[] search(string test_function, int problem_size,
double[,] search_space, int mem_size, int max_iter, double consid_rate, double
adjust_rate, double range)
        {
            HarmonyMemory[] harmonyMemory;
            Random random = new Random();
            harmonyMemory = new HarmonyMemory[mem_size];
            harmonyMemory = InitializeHarmonyMemory(test_function, problem_size,
search_space, mem_size);
            HarmonyMemory best = new HarmonyMemory(harmonyMemory[0].position,
harmonyMemory[0].fitness);
            for (int i = 0; i < max_iter; i++)
            {
                double[] candidatePosition = createHarmony(problem_size, search_space,
harmonyMemory, consid_rate, adjust_rate, range);
                double candidateFitness = objective_function(test_function,
candidatePosition);
                HarmonyMemory candidate = new
HarmonyMemory(candidatePosition, candidateFitness);
                if (candidate.fitness < best.fitness)
                {
                    candidate.position.CopyTo(best.position, 0);
                    best.fitness = candidate.fitness;
                }
                List<HarmonyMemory> listHarmony = harmonyMemory.ToList();
                listHarmony.Add(candidate);
                harmonyMemory = listHarmony.ToArray();
                Array.Sort(harmonyMemory);
            }
        }
    }
}
```

```

        listHarmony = harmonyMemory.ToList();
        listHarmony.RemoveAt(listHarmony.Count - 1);
        harmonyMemory = listHarmony.ToArray();
        Console.WriteLine(i + ". iteráció. Legjobb fitness= " + best.fitness);
    }

    double[] result=new
double[] {best.position[0],best.position[1],best.fitness};
    return result;
}

public static double[] createHarmony(int problem_size, double[,] search_space,
HarmonyMemory[] memory, double consid_rate, double adjust_rate, double range)
{
    double[] vector = new double[problem_size];
    for (int i = 0; i < problem_size; i++)
    {
        if (random.NextDouble() < consid_rate)
        {
            double value = memory[random.Next(0, memory.Length)].position[i];
            if (random.NextDouble() < adjust_rate)
            {
                value = value + range * RandomInBounds(-1.0, 1.0);
                if (value < search_space[i, 0])
                {
                    value = search_space[i, 0];
                }
                if (value > search_space[i, 1])
                {
                    value = search_space[i, 1];
                }
            }
            vector[i] = value;
        }
        else
        {
            vector[i] = RandomInBounds(search_space[i, 0], search_space[i,
1]);
        }
    }
    return vector;
}

public static double objective_function(string test_function, double[] xy)
{
    double cost = 0;
    cost = 10 * 2 + (Math.Pow(xy[0], 2) - 10 * Math.Cos(2 * Math.PI * xy[0]))
+ (Math.Pow(xy[1], 2) - 10 * Math.Cos(2 * Math.PI * xy[1]));
    return cost;
}

public static double RandomInBounds(double min, double max)
{
    return min + ((max - min) * random.NextDouble());
}

public static HarmonyMemory[] InitializeHarmonyMemory(string test_function,
int problem_size, double[,] search_space, int mem_size)
{

```

```
        HarmonyMemory[] initializeMemory = new HarmonyMemory[problem_size * mem_size];
        for (int i = 0; i < initializeMemory.Length; i++)
        {
            initializeMemory[i] = CreateRandomHarmony(test_function, problem_size,
search_space);
        }
        Array.Sort(initializeMemory);
        HarmonyMemory[] returnInitializedMemory = new HarmonyMemory[mem_size];
        for (int i = 0; i < returnInitializedMemory.Length; i++)
        {
            returnInitializedMemory[i] = new
HarmonyMemory(initializeMemory[i].position, initializeMemory[i].fitness);
        }
        return returnInitializedMemory;
    }

    public static HarmonyMemory CreateRandomHarmony(string test_function, int
problem_size, double[,] search_space)
    {
        double[] randomPosition = new double[problem_size];
        for (int i = 0; i < randomPosition.Length; i++)
        {
            double lo = search_space[i, 0];
            double hi = search_space[i, 1];
            randomPosition[i] = (hi - lo) * random.NextDouble() + lo;
        }
        double cost = objective_function(test_function, randomPosition);
        HarmonyMemory randomMemory = new HarmonyMemory(randomPosition, cost);
        return randomMemory;
    }

    public class HarmonyMemory : IComparable<HarmonyMemory>
    {
        public double[] position;
        public double fitness;
        public HarmonyMemory(double[] position, double fitness)
        {
            this.position = new double[position.Length];
            position.CopyTo(this.position, 0);
            this.fitness = fitness;
        }

        public int CompareTo(HarmonyMemory other) // based on vector/solution value
        {
            if (this.fitness < other.fitness)
                return -1;
            else if (this.fitness > other.fitness)
                return 1;
            else
                return 0;
        }
    }
}
```

A forráskód letölthető az alábbi linkről:

<https://drive.google.com/file/d/0BxE6yHbGFZBANnU3MjgzNVBLUkk/edit?usp=sharing>

9.7 Krill Herd (KH) C# forráskódja

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace KrillHerdAlgorithm
{
    class Program
    {
        static Random random = null;
        const int Dimension = 2;
        static int iteration = 0;

        static void Main(string[] args)
        {
            const int problem_size = 2;
            double[,] search_space = new double[problem_size, 2] { { -5.12, 5.12 }, {
-5.12, 5.12 } };
            int max_gens = 100;
            int pop_size = 50;
            double weightMovement = 0.9;
            double weightForaging = 0.9;
            double timeChange = 0.5;
            double best = search(problem_size, search_space, pop_size, max_gens,
weightMovement, weightForaging, timeChange);
            Console.WriteLine("Keresés vége, Legjobb megoldás: f(xy)="+ best);
            Console.ReadLine();
        }

        private static double search(int problem_size, double[,] search_space, int
numberOfKrills, int numberIterations, double weightMovement, double weightForaging,
double timeChange)
        {
            random = new Random();
            Krill[] swarm = new Krill[numberOfKrills];
            double[] bestGlobalPosition = new double[Dimension];
            double bestGlobalFitness = double.MaxValue;
            double[] worstGlobalPosition = new double[Dimension];
            double worstGlobalFitness = double.MinValue;

            for (int i = 0; i < swarm.Length; i++)
            {
                double[] randomPosition = new double[Dimension];
                for (int j = 0; j < randomPosition.Length; j++)
                {
                    double lo = search_space[j, 0];
                    double hi = search_space[j, 1];
                    randomPosition[j] = (hi - lo) * random.NextDouble() + lo;
                }
                double fitness = ObjectiveFunction( randomPosition);
                double[] randomVelocity = new double[Dimension];
```

```
        swarm[i] = new Krill(randomPosition, fitness, randomPosition, fitness,
0, new double[Dimension] { 0, 0 }, new double[Dimension] { 0, 0 }, new
double[Dimension] { 0, 0 });
        if (swarm[i].fitness < bestGlobalFitness)
        {
            bestGlobalFitness = swarm[i].fitness;
            swarm[i].position.CopyTo(bestGlobalPosition, 0);
        }
        if (swarm[i].fitness > worstGlobalFitness)
        {
            worstGlobalFitness = swarm[i].fitness;
            swarm[i].position.CopyTo(worstGlobalPosition, 0);
        }
    }

    while (iteration < numberIterations)
    {
        ++iteration;
        double[] newPosition = new double[Dimension];
        double newFitness;
        int bestIndividual = 0;
        double[] xFood = new double[Dimension];
        double xFoodFitness;
        double[] sumSzamlalo = new double[Dimension];
        double[] sumNevezo = new double[Dimension];
        for (int i = 0; i < xFood.Length; i++)
        {
            for (int j = 0; j < swarm.Length; j++)
            {
                sumSzamlalo[i] += (1.0 / swarm[j].fitness) *
swarm[j].position[i];
                sumNevezo[i] += (1.0 / swarm[j].fitness);
                if (j == swarm.Length - 1)
                {
                    xFood[i] = sumSzamlalo[i] / sumNevezo[i];
                }
            }
        }
        xFoodFitness = ObjectiveFunction(xFood);
        for (int i = 1; i < swarm.Length; i++)
        {
            if (swarm[i].fitness < swarm[bestIndividual].fitness)
                bestIndividual = i;
        }
        for (int i = 0; i < swarm.Length; i++)
        {
            double xIJ = 0;
            double sqrtXIj = 0;
            for (int k = 0; k < swarm.Length; k++)
            {
                for (int l = 0; l < Dimension; l++)
                {
                    xIJ += Math.Pow(swarm[i].position[l] -
swarm[k].position[l], 2);
                }
                sqrtXIj += Math.Sqrt(xIJ);
            }
            swarm[i].sensingDistance = (1.0 / (5.0 * (float)numberOfKrills)) *
sqrtXIj;
        }
    }
}
```



```

        newFitness = ObjectiveFunction(swarm[i].position);
        currentKrill.fitness = newFitness;
        if (newFitness < currentKrill.bestFitness)
        {
            swarm[i].position.CopyTo(swarm[i].bestPosition, 0);
            swarm[i].bestFitness = newFitness;
        }
        if (newFitness < bestGlobalFitness)
        {
            swarm[i].position.CopyTo(bestGlobalPosition, 0);
            bestGlobalFitness = newFitness;
        }
        if (newFitness > worstGlobalFitness)
        {
            swarm[i].position.CopyTo(worstGlobalPosition, 0);
            worstGlobalFitness = newFitness;
        }
    }
    Console.WriteLine(iteration + ". iteráció, Legjobb fitness= " +
bestGlobalFitness);
    if (iteration % (numberIterations / 10) == 0)
    {
        if ((weightForaging > 0.1) && (weightMovement > 0.1))
        {
            weightMovement -= 0.1;
            weightMovement -= 0.1;
        }
        if (timeChange > 0.1)
        {
            timeChange -= 0.05;
        }
    }
}
return bestGlobalFitness;
}

static double ObjectiveFunction(double[] x)
{
    double cost = 0;
    cost = 10 * 2 + (Math.Pow(x[0], 2) - 10 * Math.Cos(2 * Math.PI * x[0])) +
(Math.Pow(x[1], 2) - 10 * Math.Cos(2 * Math.PI * x[1]));
    return cost;
}

public class Krill
{
    public double[] position;
    public double fitness;
    public double[] bestPosition;
    public double bestFitness;
    public double sensingDistance;
    public double[] movement;
    public double[] foraging;
    public double[] diffusion;
}

```

```
public Krill(double[] position, double fitness, double[] bestPosition, double
bestFitness, double sensingDistance, double[] movement, double[] foraging, double[]
diffusion)
{
    this.position = new double[position.Length];
    position.CopyTo(this.position, 0);
    this.fitness = fitness;
    this.bestPosition = new double[bestPosition.Length];
    bestPosition.CopyTo(this.bestPosition, 0);
    this.bestFitness = bestFitness;
    this.sensingDistance = sensingDistance;
    this.movement = new double[movement.Length];
    movement.CopyTo(this.movement, 0);
    this.foraging = new double[foraging.Length];
    foraging.CopyTo(this.foraging, 0);
    this.diffusion = new double[diffusion.Length];
    diffusion.CopyTo(this.diffusion, 0);
}
}
```

A forráskód letölthető az alábbi linkről:

<https://drive.google.com/file/d/0BxE6yHbGFZBATVFWU05wbXlGMzQ/edit?usp=sharing>

9.8 Memetic Algorithm (MA) C# forráskódja

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MemeticAlgorithm
{
    class Program
    {
        static Random random = new Random();
        static void Main(string[] args)
        {
            const int problem_size = 2;
            double[,] search_space = new double[problem_size, 2] { { -5, 5 }, { -5, 5 } };

            int max_gens = 100;
            int pop_size = 100;
            double p_cross = 0.98;
            double p_mut = 1.0 / (double)(problem_size * 16);
            int max_local_gens = 20;
            double p_local = 0.5;
            int bits_per_param = 16;
            double best = search("Valami", problem_size, search_space, max_gens,
pop_size, p_cross, p_mut, max_local_gens, p_local, bits_per_param);
            Console.ReadLine();
        }

        public static double search(string test_function, int problem_size, double[,]
search_space, int max_gens, int pop_size, double p_cross, double p_mut, int
max_local_gens, double p_local, int bits_per_param)
        {
            Individual[] population = new Individual[pop_size];

            for (int i = 0; i < pop_size; i++)
            {
                string randomBitString = random_bitstring(problem_size *
bits_per_param);
                population[i] = fitness(test_function, randomBitString, problem_size,
search_space, bits_per_param);
            }

            Array.Sort(population);
            Individual best = population[0];

            for (int i = 0; i < max_gens; i++)
            {
                Individual[] selected = new Individual[pop_size];

                for (int j = 0; j < population.Length; j++)
                {
                    selected[j] = binary_tournament(population);
                }
            }
        }
    }
}
```

Metaheurisztikus algoritmusok hatékonyságvizsgálata saját keretrendszer által létrehozott komplex tesztfüggvények segítségével, az eredmények felhasználása futódaru főtartó optimalásához

```
Individual[] children = reproduce(test_function, problem_size,
search_space, selected, pop_size, p_cross, p_mut, bits_per_param);

    for (int j = 0; j < children.Length; j++)
    {
        if (random.NextDouble() < p_local)
        {
            population[j] = bitclimber(test_function, problem_size,
search_space, bits_per_param, children[j], p_mut, max_local_gens);
        }
    }

    Array.Sort(population);
    best = population[0];
    Console.WriteLine("" + i + " .generáció " + best.fitness);
}
return best.fitness;
}

public static string random_bitstring(int num_bits)
{
    String randomBitString = "";
    for (int i = 0; i < num_bits; i++)
    {
        if (random.NextDouble() < 0.5)
        {
            randomBitString += "0";
        }
        else
        {
            randomBitString += "1";
        }
    }
    return randomBitString;
}

public static Individual fitness(string test_function, string bitString, int
problem_size, double[,] search_space, int bits_per_param)
{
    double[] position = decode(bitString, problem_size, search_space,
bits_per_param);
    double fitness = objective_function(test_function, position);
    Individual individual = new Individual(bitString, position, fitness);
    return individual;
}

public static Individual bitclimber(string test_function, int problem_size,
double[,] search_space, int bits_per_param, Individual child, double p_mut, int
max_local_gens)
{
    Individual current = child;
    for (int i = 0; i < max_local_gens; i++)
    {
        string bitString = point_mutation(child.bitString, p_mut);
        Individual candidate = fitness(test_function, bitString, problem_size,
search_space, bits_per_param);
        if (candidate.fitness <= current.fitness)
        {
            current = candidate;
        }
    }
}
```

```
    }
    return current;
}

public static double[] decode(string bitString, int problem_size, double[, ]
search_space, int bits_per_param)
{
    double[] vector = new double[problem_size];
    for (int i = 0; i < problem_size; i++)
    {
        int offset = i * bits_per_param;
        double sum = 0.0;
        string param = bitString.Substring(offset, bits_per_param);
        char[] charArray = param.ToCharArray();
        Array.Reverse(charArray);
        param = new string(charArray);
        for (int j = 0; j < param.Length; j++)
        {
            sum += ((param[j] == '1' ? 1.0 : 0.0) * (Math.Pow(2.0, j)));
        }
        double min = search_space[i, 0];
        double max = search_space[i, 1];
        vector[i] = min + ((max - min) / (Math.Pow(2.0, bits_per_param) - 1.0)) *
sum;
    }
    return vector;
}

public static double objective_function(string test_function, double[] xy)
{
    double cost = 0;
    cost = 10 * 2 + (Math.Pow(xy[0], 2) - 10 * Math.Cos(2 * Math.PI * xy[0])) +
(Math.Pow(xy[1], 2) - 10 * Math.Cos(2 * Math.PI * xy[1]));
    return cost;
}

public static Individual[] reproduce(string test_function, int problem_size,
double[, ] search_space, Individual[] selected, int pop_size, double p_cross, double p_mut,
int bits_per_param)
{
    Individual[] children = new Individual[pop_size];

    for (int i = 0; i < selected.Length; i++)
    {
        Individual p1 = selected[i];
        Individual p2 = null;
        if (i != selected.Length - 1)
        {
            p2 = (i % 2 == 0) ? selected[i + 1] : selected[i - 1];
        }
        else
        {
            p2 = selected[0];
        }
        string bitString = crossover(p1.bitString, p2.bitString, p_cross);
        bitString = point_mutation(bitString, p_mut);
        children[i] = fitness(test_function, bitString, problem_size,
search_space, bits_per_param);
    }
}
```

```
    }  
    return children;  
}  
  
public static string crossover(string p1, string p2, double p_cross)  
{  
    if (random.NextDouble() >= p_cross)  
    {  
        return p1;  
    }  
    else  
    {  
        String crossoverBitString = "";  
        for (int i = 0; i < p1.Length; i++)  
        {  
            if (random.NextDouble() < 0.5)  
            {  
                crossoverBitString += p1[i];  
            }  
            else  
            {  
                crossoverBitString += p2[i];  
            }  
        }  
        return crossoverBitString;  
    }  
}  
  
public static string point_mutation(string bitString, double p_mut)  
{  
    String mutationBitString = "";  
    for (int i = 0; i < bitString.Length; i++)  
    {  
        if (random.NextDouble() < p_mut)  
        {  
            if (bitString[i] == 1)  
            {  
                mutationBitString += 0;  
            }  
            else  
            {  
                mutationBitString += 1;  
            }  
        }  
        else  
        {  
            mutationBitString += bitString[i];  
        }  
    }  
    return mutationBitString;  
}  
  
public static Individual binary_tournament(Individual[] population)  
{  
    int randOne = random.Next(population.Length);  
    int randTwo = random.Next(population.Length);
```

```
        while (randOne == randTwo)
        {
            randTwo = random.Next(population.Length);
        }
        if (population[randOne].fitness < population[randTwo].fitness)
        {
            return population[randOne];
        }
        else
        {
            return population[randTwo];
        }
    }

    public class Individual : IComparable<Individual>
    {
        public string bitString;
        public double[] position;
        public double fitness;

        public Individual(string bitString, double[] position, double fitness)
        {
            this.bitString = bitString;
            this.position = new double[position.Length];
            position.CopyTo(this.position, 0);
            this.fitness = fitness;
        }

        public int CompareTo(Individual other) // based on vector/solution value
        {
            if (this.fitness < other.fitness)
                return -1;
            else if (this.fitness > other.fitness)
                return 1;
            else
                return 0;
        }
    }
}
```

A forráskód letölthető az alábbi linkről:

<https://drive.google.com/file/d/0BxE6yHbGFZBAVUJhU0ZFMWxRelE/edit?usp=sharing>

9.9 Nelder-Mead Algorithm (NM) C# forráskódja

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace AmoebaMethod
{
    class Program
    {
        static void Main(string[] args)
        {
            int dim = 2;
            int amoebaSize = 50;
            double minX = -5.12;
            double maxX = 5.12;
            int maxLoop = 1000;
            Amoeba a = new Amoeba(amoebaSize, dim, minX, maxX, maxLoop);
            Solution sln = a.Solve();
            Console.ReadLine();
        }

        public static double ObjectiveFunction(double[] x, object dataSource)
        {
            return 10 * 2 + (Math.Pow(x[0], 2) - 10 * Math.Cos(2 * Math.PI * x[0])) +
                (Math.Pow(x[1], 2) - 10 * Math.Cos(2 * Math.PI * x[1]));
        }
    }

    public class Solution : IComparable<Solution>
    {
        public double[] vector;
        public double value;
        static Random random = new Random();

        public Solution(int dim, double minX, double maxX)
        {
            this.vector = new double[dim];
            for (int i = 0; i < dim; ++i)
                this.vector[i] = (maxX - minX) * random.NextDouble() + minX;
            this.value = Program.ObjectiveFunction(this.vector, null);
        }

        public Solution(double[] vector)
        {
            this.vector = new double[vector.Length];
            Array.Copy(vector, this.vector, vector.Length);
            this.value = Program.ObjectiveFunction(this.vector, null);
        }

        public int CompareTo(Solution other)
        {
            if (this.value < other.value)
                return -1;
            else if (this.value > other.value)
                return 1;
            else
                return 0;
        }
    }
}
```

```
        return 0;
    }
}

public class Amoeba
{
    public int amoebaSize;
    public int dim;
    public Solution[] solutions;
    public double minX;
    public double maxX;
    public double alpha;
    public double beta;
    public double gamma;
    public int maxLoop;

    public Amoeba(int amoebaSize, int dim, double minX, double maxX, int maxLoop)
    {
        this.amoebaSize = amoebaSize;
        this.dim = dim;
        this.minX = minX;
        this.maxX = maxX;
        this.alpha = 1.0;
        this.beta = 0.5;
        this.gamma = 2.0;
        this.maxLoop = maxLoop;
        this.solutions = new Solution[amoebaSize];
        for (int i = 0; i < solutions.Length; ++i)
            solutions[i] = new Solution(dim, minX, maxX);
        Array.Sort(solutions);
    }

    public Solution Centroid()
    {
        double[] c = new double[dim];
        for (int i = 0; i < amoebaSize - 1; ++i)
            for (int j = 0; j < dim; ++j)
                c[j] += solutions[i].vector[j];
        for (int j = 0; j < dim; ++j)
            c[j] = c[j] / (amoebaSize - 1);
        Solution s = new Solution(c);
        return s;
    }

    public Solution Reflected(Solution centroid)
    {
        double[] r = new double[dim];
        double[] worst = this.solutions[amoebaSize - 1].vector;
        for (int j = 0; j < dim; ++j)
            r[j] = ((1 + alpha) * centroid.vector[j]) - (alpha * worst[j]);
        Solution s = new Solution(r);
        return s;
    }

    public Solution Expanded(Solution reflected, Solution centroid)
    {
        double[] e = new double[dim];
        for (int j = 0; j < dim; ++j)
            e[j] = (gamma * reflected.vector[j]) + ((1 - gamma) *
centroid.vector[j]);
        Solution s = new Solution(e);
    }
}
```

```
        return s;
    }

    public Solution Contracted(Solution centroid)
    {
        double[] v = new double[dim];
        double[] worst = this.solutions[amoebaSize - 1].vector;
        for (int j = 0; j < dim; ++j)
            v[j] = (beta * worst[j]) + ((1 - beta) * centroid.vector[j]);
        Solution s = new Solution(v);
        return s;
    }

    public void Shrink()
    {
        for (int i = 1; i < amoebaSize; ++i)
        {
            for (int j = 0; j < dim; ++j)
            {
                solutions[i].vector[j] = (solutions[i].vector[j] +
solutions[0].vector[j]) / 2.0;
                solutions[i].value =
Program.ObjectiveFunction(solutions[i].vector, null);
            }
        }
        Array.Sort(solutions);
    }

    public void ReplaceWorst(Solution newSolution)
    {
        for (int j = 0; j < dim; ++j)
            solutions[amoebaSize - 1].vector[j] = newSolution.vector[j];
        solutions[amoebaSize - 1].value = newSolution.value;
        Array.Sort(solutions);
    }

    public bool IsWorseThanAllButWorst(Solution reflected)
    {
        for (int i = 0; i < amoebaSize - 1; ++i)
        {
            if (reflected.value <= solutions[i].value)
                return false;
        }
        return true;
    }

    public Solution Solve()
    {
        int t = 0;
        while (t < maxLoop)
        {
            ++t;
            Console.WriteLine("" + t + " .generáció " + solutions[0].value);
            Solution centroid = Centroid();
            Solution reflected = Reflected(centroid);
            if (reflected.value < solutions[0].value)
            {
                Solution expanded = Expanded(reflected, centroid);
                if (expanded.value < solutions[0].value)
                    ReplaceWorst(expanded);
                else

```

Metaheurisztikus algoritmusok hatékonyságvizsgálata saját keretrendszer által létrehozott komplex tesztfüggvények segítségével, az eredmények felhasználása futódaru főtartó optimalálásához

```
        ReplaceWorst(reflected);  
        continue;  
    }  
    if (IsWorseThanAllButWorst(reflected) == true)  
    {  
        if (reflected.value <= solutions[amoebaSize - 1].value)  
            ReplaceWorst(reflected);  
        Solution contracted = Contracted(centroid);  
        if (contracted.value > solutions[amoebaSize - 1].value)  
            Shrink();  
        else  
            ReplaceWorst(contracte  
        continue;  
    }  
    ReplaceWorst(reflected);  
}  
return solutions[0];  
}
```

A forráskód letölthető az alábbi linkről:

<https://drive.google.com/file/d/0BxE6yHbGFZBAYklpUzNRbzZ2aXM/edit?usp=sharing>

9.10 Particle Swarm Optimization (PSO) C# forráskódja

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ParticleSwarm
{
    class Program
    {
        static Random ran = null;
        static void Main(string[] args)
        {
            int numberParticles = 10;
            int numberIterations = 200;
            const int problem_size = 2;
            double[,] search_space = new double[problem_size, 2] { { -5.12, 5.12 }, {
-5.12, 5.12 } };
            double w = 0.729; // inertia weight
            double c1 = 1.49445; // cognitive/local weight
            double c2 = 1.49445; // social/global weight
            double[] best = search(problem_size, search_space, numberParticles,
numberIterations, w, c1, c2);
            Console.WriteLine("Keresés vége. Legjobb megoldás: x=" + best[0] + " y=" +
best[1] + " f(xy)=" + best[2]);
            Console.ReadLine();
        }

        private static double[] search(int problem_size, double[,] search_space, int
numberParticles, int numberIterations, double w, double c1, double c2)
        {
            try
            {
                ran = new Random();
                Particle[] swarm = new Particle[numberParticles];
                double[] bestGlobalPosition = new double[problem_size];
                double bestGlobalFitness = double.MaxValue;

                for (int i = 0; i < swarm.Length; ++i)
                {
                    double[] randomPosition = new double[problem_size];
                    for (int j = 0; j < randomPosition.Length; ++j)
                    {
                        double lo = search_space[j, 0];
                        double hi = search_space[j, 1];
                        randomPosition[j] = (hi - lo) * ran.NextDouble() + lo; //
                    }
                    double fitness = ObjectiveFunction(randomPosition);
                    double[] randomVelocity = new double[problem_size];

                    for (int j = 0; j < randomVelocity.Length; ++j)
                    {
                        double lo = -1.0 * Math.Abs(search_space[j, 1] -
search_space[j, 0]);
                        double hi = Math.Abs(search_space[j, 1] - search_space[j, 0]);
                        randomVelocity[j] = (hi - lo) * ran.NextDouble() + lo;
                    }
                }
            }
            catch { }
        }
    }
}
```

```
    }
    swarm[i] = new Particle(randomPosition, fitness, randomVelocity,
randomPosition, fitness);
    if (swarm[i].fitness < bestGlobalFitness)
    {
        bestGlobalFitness = swarm[i].fitness;
        swarm[i].position.CopyTo(bestGlobalPosition, 0);
    }
}

double r1, r2;
int iteration = 0;

while (iteration < numberIterations)
{
    double[] newVelocity = new double[problem_size];
    double[] newPosition = new double[problem_size];
    double newFitness;

    for (int i = 0; i < swarm.Length; ++i)
    {
        Particle currP = swarm[i];

        for (int j = 0; j < currP.velocity.Length; ++j)
        {
            r1 = ran.NextDouble();
            r2 = ran.NextDouble();
            newVelocity[j] = (w * currP.velocity[j]) +
                (c1 * r1 * (currP.bestPosition[j] - currP.position[j]))
                +
                (c2 * r2 * (bestGlobalPosition[j] - currP.position[j]));
            if (newVelocity[j] < search_space[j, 0])
                newVelocity[j] = search_space[j, 0];
            else if (newVelocity[j] > search_space[j, 1])
                newVelocity[j] = search_space[j, 1];
        }
        newVelocity.CopyTo(currP.velocity, 0);
        for (int j = 0; j < currP.position.Length; ++j)
        {
            newPosition[j] = currP.position[j] + newVelocity[j];
            if (newPosition[j] < search_space[j, 0])
                newPosition[j] = search_space[j, 0];
            else if (newPosition[j] > search_space[j, 1])
                newPosition[j] = search_space[j, 1];
        }
        newPosition.CopyTo(currP.position, 0);
        newFitness = ObjectiveFunction(newPosition);
        currP.fitness = newFitness;
        if (newFitness < currP.bestFitness)
        {
            newPosition.CopyTo(currP.bestPosition, 0);
            currP.bestFitness = newFitness;
        }
        if (newFitness < bestGlobalFitness)
        {
            newPosition.CopyTo(bestGlobalPosition, 0);
            bestGlobalFitness = newFitness;
        }
    }
    Console.WriteLine(iteration + ". iteráció. Legjobb fitness= " +
bestGlobalFitness);
```

```
        ++iteration;
    }
    double[] solution = new double[] { bestGlobalPosition[0],
bestGlobalPosition[1], bestGlobalFitness };
    return solution;
}
catch (Exception ex)
{
    return null;
}
}

static double ObjectiveFunction(double[] x)
{
    double cost = 0;
    cost = 10 * 2 + (Math.Pow(x[0], 2) - 10 * Math.Cos(2 * Math.PI * x[0])) +
(Math.Pow(x[1], 2) - 10 * Math.Cos(2 * Math.PI * x[1]));
    return cost;
}

public class Particle
{
    public double[] position;
    public double fitness;
    public double[] velocity;
    public double[] bestPosition;
    public double bestFitness;

    public Particle(double[] position, double fitness, double[] velocity, double[]
bestPosition, double bestFitness)
    {
        this.position = new double[position.Length];
        position.CopyTo(this.position, 0);
        this.fitness = fitness;
        this.velocity = new double[velocity.Length];
        velocity.CopyTo(this.velocity, 0);
        this.bestPosition = new double[bestPosition.Length];
        bestPosition.CopyTo(this.bestPosition, 0);
        this.bestFitness = bestFitness;
    }
}
}
```

A forráskód letölthető az alábbi linkről:

<https://drive.google.com/file/d/0BxE6yHbGFZBAd1JmM0sxMU5lWVU/edit?usp=sharing>

9.11 Random Search (RS) C# forráskódja

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace RandomSearch
{
    class Program
    {
        static void Main(string[] args)
        {
            const int problem_size = 2;
            double[,] search_space = new double[problem_size, 2] { { -5.12, 5.12 }, {
-5.12, 5.12 } };
            int max_gens = 1000;
            int pop_size = 10;
            Bod bod = new Bod();
            double[] best = bod.search(problem_size, search_space, pop_size,
max_gens);
            Console.WriteLine("Keresés vége. Legjobb megoldás: x=" + best[0] + " y=" +
best[1] + " f(xy)=" + best[2]);
            Console.ReadLine();
        }
        public class Bod
        {
            Random random = new Random();
            public double[] search(int problem_size, double[,] search_space, int
num_sols, int max_iterations)
            {
                Solution[] current = new Solution[num_sols];
                Solution[] candidate = new Solution[num_sols];
                double[] bestGlobalPosition = new double[problem_size];
                double bestGlobalFitness = Double.MaxValue;
                for (int i = 0; i < current.Length; i++)
                {
                    double[] randomPosition = new double[problem_size];
                    for (int j = 0; j < randomPosition.Length; j++)
                    {
                        double lo = search_space[j, 0];
                        double hi = search_space[j, 1];
                        randomPosition[j] = (hi - lo) * random.NextDouble() + lo;
                    }
                    double cost = objective_function(randomPosition);
                    current[i] = new Solution(randomPosition, cost);
                    if (current[i].fitness < bestGlobalFitness)
                    {
                        current[i].position.CopyTo(bestGlobalPosition, 0);
                        bestGlobalFitness = current[i].fitness;
                    }
                }
                for (int i = 0; i < max_iterations; i++)
                {
                    for (int j = 0; j < current.Length; j++)
                    {
                        candidate[j] = FindCandidate(problem_size, search_space);
                        if (candidate[j].fitness < bestGlobalFitness)
```

```
        {
            candidate[j].position.CopyTo(bestGlobalPosition, 0);
            bestGlobalFitness = candidate[j].fitness;
        }
    }
    if (i % 100 == 0)
    {
        Console.WriteLine(i + ". iteráció. Legjobb fitness= " +
bestGlobalFitness);
    }
}
double[] solution = new double[] { bestGlobalPosition[0],
bestGlobalPosition[1], bestGlobalFitness };
return solution;
}
public Solution FindCandidate(int problem_size, double[,] search_space)
{
    Random ran = new Random();
    double[] newPosition = new double[problem_size];
    for (int j = 0; j < problem_size; j++)
    {
        double lo = search_space[j, 0];
        double hi = search_space[j, 1];
        newPosition[j] = ((hi - lo) * random.NextDouble() + lo);
    }
    double cost = objective_function(newPosition);
    Solution candidate = new Solution(newPosition, cost);
    return candidate;
}
public double objective_function(double[] x)
{
    double cost = 0;
    cost = 10 * 2 + (Math.Pow(x[0], 2) - 10 * Math.Cos(2 * Math.PI * x[0])) +
(Math.Pow(x[1], 2) - 10 * Math.Cos(2 * Math.PI * x[1]));
    return cost;
}
public class Solution
{
    public double[] position;
    public double fitness;
    public Solution(double[] position, double fitness)
    {
        this.position = new double[position.Length];
        position.CopyTo(this.position, 0);
        this.fitness = fitness;
    }
}
}
}
```

A forráskód letölthető az alábbi linkről:

<https://drive.google.com/file/d/0BxE6yHbGFZBANDIDWmRmMDJxVVE/edit?usp=sharing>

9.12 Simulated Annealing (SA) C# forráskódja

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SimulatedAnnealing
{
    class Program
    {
        static void Main(string[] args)
        {
            const int problem_size = 2;
            double[,] search_space = new double[problem_size, 2] { { -5.12, 5.12 }, {
-5.12, 5.12 } };
            int max_iterations = 100;
            double max_temp = 10.0;
            double temp_change = 0.995;
            int num_sols = 10;
            double[] best = search(problem_size, search_space, num_sols,
max_iterations, max_temp, temp_change);
            Console.WriteLine("Keresés vége. Legjobb megoldás: x=" + best[0] + " y=" +
best[1] + " f(xy)=" + best[2]);
            Console.ReadLine();
        }

        private static double[] search(int problem_size, double[,] search_space, int
num_sols, int max_iterations, double max_temp, double temp_change)
        {
            Solution[] current = new Solution[num_sols];
            double[] bestGlobalPosition = new double[problem_size];
            double bestGlobalFitness = Double.MaxValue;
            Solution[] neighbour = new Solution[num_sols];
            Random random = new Random();

            for (int i = 0; i < current.Length; i++)
            {
                double[] randomPosition = new double[problem_size];
                for (int j = 0; j < randomPosition.Length; j++)
                {
                    double lo = search_space[j, 0];
                    double hi = search_space[j, 1];
                    randomPosition[j] = (hi - lo) * random.NextDouble() + lo;
                }
                double cost = objective_function(randomPosition);
                current[i] = new Solution(randomPosition, cost);
                if (current[i].fitness < bestGlobalFitness)
                {
                    current[i].position.CopyTo(bestGlobalPosition, 0);
                    bestGlobalFitness = current[i].fitness;
                }
            }
            double temp = max_temp;
            for (int i = 0; i < max_iterations; i++)
```

```

        {
            for (int j = 0; j < current.Length; j++)
            {
                neighbour[j] = FindNeighbour(current[j], problem_size,
search_space, current.Length);
                if (should_accept(neighbour[j], current[j], temp))
                {
                    neighbour[j].position.CopyTo(current[j].position, 0);
                    current[j].fitness = neighbour[j].fitness;
                }
                if (neighbour[j].fitness < bestGlobalFitness)
                {
                    neighbour[j].position.CopyTo(bestGlobalPosition, 0);
                    bestGlobalFitness = neighbour[j].fitness;
                }
            }
            temp = temp * temp_change;
            if (i % 10000 == 0)
            {
                Console.WriteLine(i + ". iteráció. Legjobb fitness= " +
bestGlobalFitness);
            }
            double[] solution = new double[] { bestGlobalPosition[0],
bestGlobalPosition[1], bestGlobalFitness };
            return solution;
        }

private static bool should_accept(Solution candidate, Solution current, double
temp)
{
    Random rand = new Random();
    if (candidate.fitness <= current.fitness)
    {
        return true;
    }
    else
    {
        double ertekFix = Math.Exp((current.fitness - candidate.fitness) /
temp);

        double ertekRandom = rand.NextDouble();
        return ertekFix > ertekRandom;
    }
}

public static Solution FindNeighbour(Solution current, int problem_size,
double[,] search_space, int num_sols)
{
    Random ran = new Random();
    double[] randomDistance = new double[problem_size];
    double[] newPosition = new double[problem_size];
    for (int j = 0; j < current.position.Length; j++)
    {
        double lo = search_space[j, 0];
        double hi = search_space[j, 1];
        randomDistance[j] = ((hi - lo) * ran.NextDouble() + lo) /
(double)num_sols; //
        newPosition[j] = current.position[j] + randomDistance[j];
        if (newPosition[j] < search_space[j, 0])
            newPosition[j] = search_space[j, 0];
    }
}

```

Metaheurisztikus algoritmusok hatékonyságvizsgálata saját keretrendszer által létrehozott komplex tesztfüggvények segítségével, az eredmények felhasználása futódaru főtartó optimalásához

```
        else if (newPosition[j] > search_space[j, 1])
            newPosition[j] = search_space[j, 1];
    }
    double cost = objective_function(newPosition);
    Solution neighbour = new Solution(newPosition, cost);
    return neighbour;
}

public static double objective_function(double[] x)
{
    double cost = 0;
    cost = 10 * 2 + (Math.Pow(x[0], 2) - 10 * Math.Cos(2 * Math.PI * x[0])) +
    (Math.Pow(x[1], 2) - 10 * Math.Cos(2 * Math.PI * x[1]));
    return cost;
}

public class Solution
{
    public double[] position;
    public double fitness;
    public Solution(double[] position, double fitness)
    {
        this.position = new double[position.Length];
        position.CopyTo(this.position, 0);
        this.fitness = fitness;
    }
}
}
```

A forráskód letölthető az alábbi linkről:

<https://drive.google.com/file/d/0BxE6yHbGFZBAY2MlUkVmOFRLYWM/edit?usp=sharing>

10 2. számú melléklet, futódaru főtartó optimalás C# forráskódja

10.1 Büntetőfüggvény

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace BridgeCraneMainBeam.Data
{
    public static class Constraints
    {
        public static double force = 240000f;
        public static double longitude = 20000f;
        public static double fatigue;
        public static double fatiguemax;
        public static double staticstress;
        public static double staticstressmax;
        public static double flange;
        public static double flangemax;
        public static double web;
        public static double webmax;
        public static double deflection;
        public static double deflectionmax;

        public static double checkConstraints(double[] variables, double[,]
searchSpace)
        {
            double fatigue = FatigueConstraint(variables);
            double staticstress = StaticStressConstraint(variables);
            double flangebuckling = FlangeBuckling(variables);
            double webbuckling = WebBuckling(variables);
            double deflection = DeflectionConstraint(variables);
            double searchspace = Limits(variables, searchSpace);
            return fatigue + staticstress + flangebuckling + webbuckling + deflection
+ searchspace;
        }

        public static double FatigueConstraint(double[] variables)
        {
            double Ix = (2f * (Math.Pow(variables[0], 3) * variables[1] / 2f) / 12f) +
((2f * variables[2] * Math.Pow(variables[3], 3)) / 12f) + (2f * variables[2] *
variables[3] * Math.Pow(variables[0] / 2f + variables[3] / 2f, 2));
            double Wx = Ix / (variables[0] / 2f + variables[3]);
            double deltaSigma = (2f * (force / (2f * longitude)) * Math.Pow(longitude
- (1900f / 2f), 2)) / (4f*Wx);
            fatigue = deltaSigma;
            fatiguemax = 69.8 / 1.05;
            if ((fatigue - fatiguemax) > 0)
            {
                return 1000000000000.0;
            }
            else
            {
                return 0.0;
            }
        }
    }
}
```

```
    }  
}  
  
public static double StaticStressConstraint(double[] variables)  
{  
    double Ix = (2f * (Math.Pow(variables[0], 3) * variables[1] / 2f) / 12f) +  
    ((2f * variables[2] * Math.Pow(variables[3], 3)) / 12f) + (2f * variables[2] *  
variables[3] * Math.Pow(variables[0] / 2f + variables[3] / 2f, 2));  
    double Wx = Ix / (variables[0] / 2f + variables[3]);  
    double sigmaMax = (1.35 * (((variables[0] * variables[1]) + (2f *  
variables[2] * variables[3])) * 0.00000785 * 9.81 + 0.2 + 1.0+3.0) *  
Math.Pow(longitude, 2)) / (float)8 + (1.5 * (float)2 * force * longitude) / (float)4;  
    staticstress = sigmaMax / Wx;  
    staticstressmax = 235f / 1.1;  
    if ((staticstress - staticstressmax) > 0)  
    {  
        return 1000000000000.0;  
    }  
    else  
    {  
        return 0.0;  
    }  
}  
  
public static double FlangeBuckling(double[] variables)  
{  
    flange = (variables[2]) / variables[3];  
    flangemax = 42f;  
    if ((flange - flangemax) > 0)  
    {  
        return 1000000000000.0;  
    }  
    else  
    {  
        return 0.0;  
    }  
}  
  
public static double WebBuckling(double[] variables)  
{  
    double web = (2f*variables[0]) / (variables[1] );  
    double webmax = 124f;  
    if ((web - webmax) > 0)  
    {  
        return 1000000000000.0;  
    }  
    else  
    {  
        return 0.0;  
    }  
}  
  
public static double DeflectionConstraint(double[] variables)  
{  
    double Ix = (2f * (Math.Pow(variables[0], 3) * variables[1] / 2f) / 12f) +  
    ((2f * variables[2] * Math.Pow(variables[3], 3)) / 12f) + (2f * variables[2] *  
variables[3] * Math.Pow(variables[0] / 2f + variables[3] / 2f, 2));  
    double Wx = Ix / (variables[0] / 2f + variables[3]);  
    deflection = ((force * (longitude - 1900f)) / (48f * 210000 * Ix)) * (3f  
* Math.Pow(longitude, 2) - (Math.Pow(longitude - 1900f, 2)));  
}
```

```
deflectionmax = longitude / (float)300;
if ((deflection - deflectionmax) > 0)
{
    return 1000000000000.0;
}
else
{
    return 0.0;
}
}

public static double Limits(double[] variables, double[,] searchSpace)
{
    if (variables[0] <= searchSpace[0, 0] || variables[0] >= searchSpace[0, 1] ||
variables[1] <= searchSpace[1, 0] || variables[1] >= searchSpace[1, 1] || variables[2] <=
searchSpace[2, 0] || variables[2] >= searchSpace[2, 1] || variables[3] <= searchSpace[3,
0] || variables[3] >= searchSpace[3, 1])
    {
        return Double.MaxValue;
    }
    else return 0.0;
}
}
}
```

10.2 Célfüggvény

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace BridgeCraneMainBeam.Data
{
    public static class ObjectiveFunctions
    {
        public static double CalculateWeight(double[] variables)
        {
            double felulet = (variables[0] * variables[1]) + (2.0 * variables[2] *
variables[3]);
            double V = felulet * Form1.L;
            double tomeg = V * Form1.suruseg;

            return tomeg;
        }
    }
}
```

A forráskód letölthető az alábbi linkről:

<https://drive.google.com/file/d/0BxE6yHbGFZBAejNZRC1iN0VsUjA/edit?usp=sharing>

11 Irodalomjegyzék

- [1] JÁRMAI K., IVÁNYI M.: Gazdaságos fém szerkezetek analízise és tervezése, Műegyetemi kiadó, Budapest, 2001. ISBN 963 4206743
- [2] TÍMÁR I.: Műszaki problémák optimális megoldása, A Miskolci egyetem habilitációs füzetek, Miskolc, 2001.
- [3] VIRÁG Z., JÁRMAI K.: Szabványos kör keresztmetszetű többtámaszú tartók optimális méretezése, GÉP folyóirat, 2013/2. ISSN 0016-8572
- [4] ALI M. M., STOREY C. és TORN A.: Application of stochastic global optimization algorithms to practical problems, Journal of Optimization Theory and Applications, 1997. pp. 545-563
- [5] MICHELL, A. G. M.: The limits of economy of material in frame-structures, Philosophical Magazine, Vol. 8(47), 1904. pp. 589-597
- [6] PARETO V.: Cours d'Économie Politique Professe a l'Université de Lausanne. Vol. I, 1896; Vol. II, 1897.
- [7] BIGGS N.L., LLOYD E.K., és WILSON R.J.: Graph Theory 1736-1936, Clarendon Press, Oxford, 1976.
- [8] S. HAWKING: A Brief History of Time, Bantam, 10th anniversary edition, 1998. ISBN: 978-0553380163
- [9] WOLPERT D. H., MACREADY W. G.: No free lunch theorems for optimization, IEEE Transactions on Evolutionary Computation, 1997. pp. 67–82
- [10] DE CASTRO L. N. AND VON ZUBEN F. J.: An evolutionary immune network for data clustering. In Proceedings Sixth Brazilian Symposium on Neural Networks, IEEE Computer Society, 2000. pp. 84–89
- [11] BROWNLEE J.: Clever Algorithms: Nature-Inspired Programming Recipes, Lulu, 2011. ISBN: 978-1-4467-8506-5, pp. 294
- [12] LIU Y. és PASSINO K. M.: Biomimicry of social foraging bacteria for distributed optimization: Models, principles, and emergent behaviors, Journal of Optimization Theory and Applications, 2002. pp. 603–628
- [13] WOOLLEY A.W., CHABRIS C. F., PENTLAND A., HASHMI N., és T. W. MALONE: Evidence for a Collective Intelligence Factor in the Performance of Human Groups, Journal of Science Vol. 330, 2010. pp. 686-688
- [14] BROWNLEE J. pp. 259-260
- [15] PASSINO K. M.: Bacterial foraging optimization, International Journal of Swarm Intelligence Research, 2010. pp. 1–16

- [16] PHAM D. T., GHANBARZADEH A., KOC E., OTRI S., RAHIM S., AND ZAIDI M.: The bees algorithm. Technical report, Manufacturing Engineering Centre, Cardiff University, 2005.
- [17] BROWNLEE J. pp. 253
- [18] REYNOLDS R. G.: An introduction to cultural algorithms. In Proceedings of the 3rd Annual Conference on Evolutionary Programming, World Scientific Publishing, 1994. pp. 131–139
- [19] GUNDA B.: The Cultural Ecology of Service Nomads. The Eastern Anthropologist, 1981. pp. 287-299.
- [20] BROWNLEE J. pp. 188
- [21] STORN R. és PRICE K.: Differential evolution: A simple and efficient adaptive scheme for global optimization over continuous spaces, Technical Report TR-95-012, International Computer Science Institute, Berkeley, CA, 1995.
- [22] DARWIN C.: On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life (1st ed., 1859), London, John Murray, 2008. ISBN 1-4353-9386-4.
- [23] PRICE K., STORN R. M. ÉS LAMPINEN J. A.: Differential evolution: A practical approach to global optimization, Springer, 2005.
- [24] BROWNLEE J. pp. 115-116
- [25] GEEM Z. W., KIM J. H., AND LOGANATHAN G. V.: A new heuristic optimization algorithm: Harmony search. Simulation, 76:60–68, 2001.
- [26] GEEM Z. W.: Optimal Design of Water Distribution Networks Using Harmony Search. Lap Lambert Academic Publishing, 2009.
- [27] GEEM Z. W., editor. Harmony Search Algorithms for Structural Design Optimization. Springer, 2009.
- [28] BROWNLEE J. pp. 183
- [29] GANDOMI A. H. AND ALAVI A. H.: "Krill herd: a new bio-inspired optimization algorithm," Communications in Nonlinear Science and Numerical Simulation, vol. 17, no. 12, 2012. pp. 4831–4845
- [30] HAMNER W. M., HAMNER P. P., STRAND S. W. & GILMER R. W.: "Behavior of Antarctic krill, *Euphausia superba*: chemoreception, feeding, schooling and molting". Science 220 (4595), 1983. pp. 433–435

- [31] WANG G. G, GUO L., GANDOMI A. H., ALAVI A. H., AND DUAN H.: Simulated Annealing-Based Krill Herd Algorithm for Global Optimization, Hindawi Publishing Corporation, Volume 2013, Article ID 213853,
- [32] MOSCATO P.: On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. Technical report, California Institute of Technology, 1989.
- [33] BROWNLEE J. pp. 194
- [34] NELDER J. A.; MEAD R.: "A simplex method for function minimization". Computer Journal 7, 1965. pp. 308–313.
- [35] MCCAFFREY J.: Amoeba Method Optimization using C#, MSDN Magazine 2013 June. <http://msdn.microsoft.com/en-us/magazine/dn201752.aspx>
- [36] MCCAFFREY J: Amoeba Method Optimization using C#
- [37] KENNEDY J. és EBERHART R. C.: Particle swarm optimization, In Proceedings IEEE int'l conf. on neural networks Vol. IV, 1995. pp 1942–1948
- [38] HU X., EBERHART R.: Solving Constrained Nonlinear Optimization Problems with Particle Swarm Optimization, In Proceedings of the 6th World Multiconference on Systemics, Cybernetics and Informatics (SCI 2002), volume 5. Orlando, USA, IIS, 2002.
- [39] BROWNLEE J. pp. 234
- [40] BROOKS S. H.: A discussion of random methods for seeking maxima. Operations Research, 6(2), 1958. pp. 244–251
- [41] KIRKPATRICK S.: Optimization by simulated annealing: Quantitative studies. Journal of Statistical Physics, 1983. pp 975–986
- [42] CORANA A., MARCHESI M., MARTINI C., and RIDELLA S.: Minimizing Multimodal Functions of Continuous Variables with the ‘Simulated Annealing’ Algorithm, Istituto per i Circuiti Elettronici-C.N.R., ACM Transactions on Mathematical Software, Vol. 13, No. 3, September 1987. pp. 262-280
- [43] ZHANG K., FU Z.: Effects of annealing treatment on phase composition and microstructure of CoCrFeNiTiAl_x high-entropy alloys, Intermetallics Volume 22, March 2012. pp 24-32
- [44] BROWNLEE J. pp. 170

- [45] MOLGA M., SMUTNICKI C.: Test functions for optimization needs, 2005, <http://www.zsd.ict.pwr.wroc.pl/files/docs/functions.pdf>
- [46] LIANG J., SUGANTHAN N., DEB K.: Novel composition test functions for numerical global optimization, Swarm Intelligence Symposium, SIS 2005. Proceedings IEEE, 2005. pp. 68-75
- [47] BARCSÁK CS., JÁRMAI K.: Benchmark for testing evolutionary algorithms, 10th World Congress on Structural and Multidisciplinary Optimization, May 19 -24, 2013, Orlando, Florida, USA
- [48] FARKAS J., JÁRMAI K.: Analysis and Optimum Design of Metal Structures, Taylor & Francis, ISBN: 978-9-0541-0669-2, 1997. pp. 236-239