CENTRAL WASHINGTON UNIVERSITY, DEPARTMENT OF COMPUTER SCIENCE

Optimization using Dragonfly Algorithm

Emily Bodenhamer

December 7, 2019

CWU ID: 41119306

Dr. Donald Davendra

1 INTRODUCTION TO THE PROBLEM

The purpose for this project is to code a new meta-heuristic for solving single-objective problems. The Dragonfly algorithm (DA) was coded and results were extracted and analyzed. This algorithm was used to improve the fitness from eighteen different optimization functions. Thirty experimentations have been conducted with a minimum of 100 iterations. The average, standard deviation, median, range, time, and a calculation of the total number of calls to the fitness function are recorded for the algorithm.

1.1 DRAGONFLY OPTIMIZATION

Dragonfly optimization is modeled after the static and dynamic swarm behavior of dragonflies. It was invented by Seyedali Mirjalili in 2014. The swam behaviors in this algorithm, mirrors the exploration and exploitation phases in optimization. In a static swarm, dragonflies create a sub-swarm and fly over different areas similar to the exploration phase. In a dynamic swarm, dragonflies fly in bigger swarms along one direction similar to the exploitation phase.

1.2 DRAGONFLY ALGORITHM

The dragonfly algorithm starts by initializing the values on every weight. In this experimentation the different weights are s, a, c, f, e, w, and y.

w is calculated using the following formula:

$$w = 0.9 - \texttt{currentIteration} * \frac{0.9 - 0.4}{\texttt{maxIterations}} \tag{1.1}$$

The weight y is calculated using the following formula:

$$y = 0.1 - \texttt{currentIteration} * \frac{0.1}{\frac{\texttt{maxIteration}}{2}} \tag{1.2}$$

The weight y, is used for the initialization of the rest of the weights. s, a, and c are calculated using the following formula:

$$s = 2 * \texttt{rand} * y \tag{1.3}$$

After the weights have been initialized, a *radius* for each individual in the population will be calculated. This *radius* is calculated by the following formula:

$$radius = \frac{(\texttt{upperBounds} - \texttt{lowerBounds})}{4 + ((\texttt{upperBounds} - \texttt{lowerBounds}) * (\frac{\texttt{currentIteration}}{\texttt{maxIteration}})} \tag{1.4}$$

Every individual in the population will have a neighbor population which will have a calculated velocity per individual. In order to fill the neighboring population, a distance vector is initialized. This distance vector will hold the Euclidean distance between the current individual at the current iteration with the next neighboring individual in the population. This will be done until all individuals are calculated for the entire population.

Once the distance vector is calculated for the current iteration and if the radius of that individual is greater than or equal to the distance of that individual, the neighbor population and neighbor vector are set equal to the current individual in the population and the current individual's population velocity.

After the dragonfly neighbors have been calculated, the population will need to be updated. A dragonfly swarm will be attracted towards food sources and distracted from enemies in order to survive. In order to update the population, five factors that contribute to updating an individuals position in the swarm are calculated.

The first factor is separation. Separation is when a dragonfly avoids the other individuals in their neighborhood.

Separation is calculated as the following:

$$S_i = -\sum_{j=1}^{N} X - X_j \tag{1.5}$$

X is the position of the current individual, X_j is the position j-th neighboring individual, and N is the number of neighboring individuals.

The next factor is alignment. Alignment is when a dragonfly matches the velocity of other individual in their neighborhood.

Alignment is calculated as the following:

$$A_i = \frac{\sum_{j=1}^{N} V_j}{N} \tag{1.6}$$

V_j is the velocity of the j-th neighboring individual.

The next factor is cohesion. Cohesion is when a dragonfly moves towards the center of the mass of the other individuals in their neighborhood.

Cohesion is calculated as the following:

$$C_i = \frac{\sum_{j=1}^{N} X_j}{N} - X \tag{1.7}$$

The next factor is attraction. Attraction is when a dragonfly moves towards the food source.

Attraction is calculated as the following:

$$F_i = X^+ - X \tag{1.8}$$

X^+ is the position of the food source.

The last factor is distraction. Distraction is when a dragonfly moves away from the enemy.

Distraction is calculated as the following:

$$E_i = X^- + X \tag{1.9}$$

X^- is the position of the enemy.

These factors are then used to update the velocity vector of the population. The velocity vector is noted as ΔX.

ΔX is calculated as the following:

$$\Delta X_{t+1} = (sS_i + aA_i + cC_i + fF_i + eE_i) + w\Delta X_t \tag{1.10}$$

The weights, factors, and t, the iteration counter, are used for the calculation.

ΔX is then used to update the population, the following equation shows how this is calculated:

$$X_{t+1} = X_t + \Delta X_{t+1} \tag{1.11}$$

Updating the population using this method is only done if the radius of that individual is greater than the distance of that individuals neighbors or if there are more than one neighbor in the population. If neither conditions are met then random walk will be used to improve randomness, stochastic behavior, and exploration of the dragonflies.

Finally, the new position of the dragonflies are checked and corrected based on the bounds. The food source and enemy are updated if better and worst solutions are found in the population.

1.3 Experimentation Setup

A population matrix is created from pseudo random values. These values are bounded based on 18 different basic optimization benchmark functions. The fitness of each individual vectors from the population are calculated and sent to the algorithm in order to further minimize the results. Each of the 30 experimentations have 100 iterations, a population size of 100 and 30 dimensions.

1.4 Results and Analysis

The best and worst results from DA were recorded for each iteration and each experimentation.

DA called each benchmark function 10000 times over the 30 experimentations. DA performed the best when the results were not too far from the optimal value. Benchmark functions such as DeJong 1, Rosenbrocks Saddle, Rastrigin, Quartic, and Alpine, did not get close enough to their optimal values. They were all able to improve over time, however not as drastically as the other benchmark functions. The times for each benchmark function were within the same amount of time except for Rastrigin. Rastrigin took about 2704 ms more to finish the algorithm compared to the second longest function, Stretch V Sine Wave, which took 4377.1 ms.

1.5 Other Algorithms

Comparing DA with the Gray Wolf Optimizer (GWO), the results obtained from GWO were almost all better than the results obtained from DA. For GWO, the vector $\vec{A}$, is constantly being recalculated during each iteration. If the vector finds a better prey than the current prey in the search space, then the predators will move towards that prey instead of the current prey. This could be the reason why GWO out performs DA. DA will only update the current food source when a new population is calculated. $\vec{A}$ is calculated and moves the predator before a new population is created. Another advantage of GWO is that the execution time is much faster than DA. The fastest time for DA was 4210 ms while the slowest time for GWO was 54 ms.

Comparing DA with the Ant Lion Optimization (ALO), almost all the benchmark functions for DA were better than ALO. This may be because the ALO involves a lot more randomization than both GWO and DA. ALO uses roulette selection and random walk. DA also uses random walk, however only when the radius is greater than the distances of the dragonflies neighbors. While comparing the amount of time it takes to execute both algorithms, ALO overall is faster. An argument could be made to use Ant Lion instead of DA purely just for a faster execution time, since although DA does better, some functions only have a slightly better result. Masters Cosine Wave was about -11.12 for DA and for ALO it was -2.11. The time difference between the two functions was about 635 ms. For other functions that did a lot better like schwefel's, it would be better to use DA regardless of the time constraint.

Comparing DA with Differential Evolution Algorithm (DE), the results show that overall, DA out performs PSO. This is because DA could be seen as an enhanced PSO algorithm. DA does the basic functionality that PSO does, however it is expanded more by adding more weights and vectors. Only two algorithms under performed from DA, which were Pathological and Michalewicz. These two functions had a slight improvement for PSO. When looking at the time that PSO takes compared to DA, usually PSO was faster. There were a few functions took more time than DA, which could make PSO seem unreliable for determining if it would be better to use than DA.

Comparing DA with Differential Evolution Algorithm the results show that overall, DA out performs DE. Previously, PSO was compared with DE and it was also outperformed. Since DA

could be seen as an enhanced PSO algorithm, it is no surprise that DA also outperformed DE.

1.6 Conclusion

After reviewing the results obtained from DA and comparing them between four different optimization algorithms, it can be concluded that DA can outperform a variety of algorithms. Although DA is able to provide good results its downfall is the amount of time it takes to get its results. Out of all of the algorithms that were analyzed, the Gray Wolf Optimizer had the best performance while the Dragonfly Algorithm comes in second. A lot of changes could be made to improve the time and results obtained. Some improvements could be using threads or using any high performance computing softwares. The equations used for the weights were not tuned during this experimentation. Tuning the weights could also provide improvements to this algorithm.

Table 1.1: Dragonfly Algorithm Best Solution Analytic Over 30 Experimentations

f(x)	Average	Standard Deviation	Median	Range	Time (ms)	Function Call
Schwefel	-117210.5104	229928.4593	-46395.77318	3335693.265	4210.066667	10000
DeJong 1	17224.88395	23781.01372	3685.608538	69344.99448	4.21E+03	10000
Rosenbrocks Saddle	5164611101	9281541580	22216914.64	33809277825	4268.6	10000
Rastrigin	451901.4719	631267.2083	57672.88822	1909735.49	7081.457627	10000
Griewangk	104.6888714	145.8502238	14.0767555	426.500918	4262.766667	10000
Sine Envelope Sine Wave	-22.11285496	2.736525247	-21.920334	10.388126	4369.7	10000
Stretch V Sine Wave	-154.4883816	109.7962651	-129.062768	553.474658	4377.1	10000
Ackley One	140.0012477	149.7402539	58.458612	458.433819	4.30E+03	10000
Ackley Two	225.782694	177.9501781	190.543691	525.566853	4339.9	10000
Egg Holder	-220615.0498	492288.211	-79718.7173	4672632.688	4251.6	10000
Rana	-177374.0414	311840.0009	-60907.84317	1928129.369	4297.8	10000
Pathological	-4.775482427	3.499182693	-5.652142	17.11091	4351.666667	10000
Michalewicz	-5.094057729	0.780102441	-5.04517	4.146122	4270.6	10000
Masters Cosine Wave	-11.12432776	10.9795787	-4.724859	27.680931	4372.133333	10000
Quartic	777044380.3	1361148515	8213336.517	4989970681	4223.033333	10000
Levy	214.9503487	261.9944889	55.0614835	751.587675	4228.166667	10000
Step	109.3108829	174.6024219	25.81832	656.797013	4239.066667	10000
Alpine	2598.956233	5224.715532	127.086666	23371.08988	4353	10000

Table 1.2: Dragonfly Algorithm Worst Solution Analytic over 30 Experimentations

f(x)	Average	Standard Deviation	Median	Range	Time (ms)	Function Call
Schwefel	124149.4718	160351.4575	62851.17984	1166541.047	4210.066667	10000
DeJong 1	1.04E+11	8.41E+11	329805054.2	1.44E+13	4.21E+03	10000
Rosenbrocks Saddle	3.08E+23	2.23E+24	9.70E+18	4.49E+25	4268.6	10000
Rastrigin	1.35E+12	5.76E+12	11427069278	7.91E+13	7081.457627	10000
Griewangk	1192691005	12858081342	3441358.275	2.33E+11	4262.766667	10000
Sine Envelope Sine Wave	-7.741083406	1.081377941	-7.912	4.9543	4369.7	10000
Stretch V Sine Wave	214.5287284	120.1828417	172.666305	495.685136	4377.1	10000
Ackley One	52023.09781	254506.3201	11178.20719	4142262.036	4.30E+03	10000
Ackley Two	613.591723	47.47513804	615.25907	28.303451	4339.9	10000
Egg Holder	184412.1325	3.25E+05	77065.28073	4.90E+06	4251.6	10000
Rana	178301.427	323752.3816	58852.68211	2144793.041	4297.8	10000
Pathological	15.65014188	1.277669702	15.628011	3.409773	4351.666667	10000
Michalewicz	5.029507154	0.728390521	4.952429	3.833737	4270.6	10000
Masters Cosine Wave	5.900066699	4.656794734	4.616048	15.580359	4372.133333	10000
Quartic	3.24E+22	3.96E+23	9.95E+17	1.29E+25	4223.033333	10000
Levy	179986301.1	906627160.6	3507284.253	13486987171	4228.166667	10000
Step	177922.9258	516934.5116	34578.20732	5730651.918	4239.066667	10000
Alpine	20376133416	1.02E+11	197308074.1	1.75E+12	4353	10000

Table 1.3: Gray Wolf Optimizer Best Solution Analytic

F(x)	Average	Standard Deviation	Median	Range	Execution Time	Best
Schwefel's	-6715515.542	25857615.37	-88903	139255220.3	46.215	7.63904
1st De Jong's	3.874955013	16.53700424	3.7376E-12	86.7283	47.491	0.00E+00
Rosenbrock	2731.236667	14789.42908	28.96835	81007.3424	46.572	28.7576
Rastrigin	-86191.90667	3661.109756	-87000	19867.2	45.868	-67132.8
Griewangk	1.028460333	0.121908631	1	0.64144	46.062	1.00002
Sine Envelope Sine Wave	-26.28648333	0.71362537	-26.211	3.5028	46.157	-24.9337
Stretched V Sine Wave	9.79643	7.2269E-15	9.79643	0	48.199	9.79643
Ackley's One	52.33064333	7.082641895	51.71955	30.5834	46.316	34.4954
Ackley's Two	-1819697568	8432991208	-2949790	46142279887	47.479	-120113
Egg Holder	-3323632.719	10945060.21	-80352.2	56784623.13	45.895	-5576.87
Rana	-567836.4087	912461.4671	-166902.5	3938740.14	47.288	-5329.86
Pathological	-3.699627	1.051061899	-3.547615	3.55815	46.288	-2.10938
Michalewicz	-3.248725333	0.416557474	-3.223535	1.59468	45.914	-2.58779
Master Cosine Wave	-14.18115	1.282123359	-14.03335	4.7719	47.04	-12.5253
Quartic	0.002988066	0.016349399	0	0.0895524	46.97	1.49E-11
Levy	7.776723667	2.427106452	7.25	13.2448	54.945	7.25245
Step	0.152320324	0.652807815	2.722E-152	3.52097	46.05	0.00263862
Alpine	-3.19868E+13	1.25151E+14	-180508950	5.91909E+14	50.787	-1.66E+05

Table 1.4: Ant Lion Algorithm Best Solution Analytic

f(x)	Time (ms)	Best Result	Mean	Std Dev	Median	Range
Schwefel's	3407.0	7963.93	9565.7	-171.75	9465.91	5036.7
De Jong's First	2878.0	45644.65	81755.07	-692.5	85697.47	49960.44
Rosenbrock	2995.0	15382044028.63	34362648424.25	-814181701.34	36386743826.99	35264238422.53
Rastagrin	3266.0	1220476.65	2019456.23	-26865.94	2087180.91	1336298.3
Griewangk	3415.0	272.5	536.97	-6.87	568.23	401.93
Sine Envelope Sine Wave	3481.0	-27.63	-24.41	-0.19	-25.65	6.99
Stretched V Sine Wave	4062.0	227.79	283.65	-3.67	280.31	129.32
Ackley's One	3641.0	243.45	335.4	-3.08	330.04	153.58
Ackley's Two	4209.0	563.15	581.15	-0.66	584.19	31.15
Egg Holder	3806.0	-5969.61	-3675.48	-84.78	-3501.94	3989.82
Rana	4667.0	-3972.34	-2969.38	-152.48	-3078.32	4052.61
Pathological	3575.0	12.45	13.5	-0.06	13.13	2.17
Michalewicz	3748.0	-8.51	-7.07	-0.17	-6.93	4.75
Master's Consine Wave	3737.0	-2.11	-0.64	-0.03	-0.17	2.11
Quartic	3143.0	2263132495.45	4701151269.62	-138960440.97	4411815306.3	5217227593.64
Levy	3469.0	63285.16	107331.85	-1690.43	111049.15	77855.25
Step	2823.0	46774.45	76987.14	-1139.92	77114.76	53011.07
Alpine	3141.0	422.59	582.79	-12.4	556.04	408.16

Table 1.5: Particle Swarm Optimization Analytics

f(x)	Average	Standard Deviation	Median	Range	Time (ms)	Function Call
Schwefel	4634.597194	1776.680852	3442.602522	6213.293621	1091	249500
DeJong 1	51578.96884	957.2013967	51309.07176	3985.79369	968	249500
Rosenbrocks Saddle	20786769657	2109333175	19702283538	5334615619	2419	249500
Rastrigin	1331686.918	1522.913667	1330504.61	3148.016	1451	249500
Griewangk	208.8146687	26.26797293	205.108708	90.475603	1772	249500
Sine Envelope Sine Wave	-22.91717742	0.294695522	-22.960404	3.026892	4799	249500
Stretch V Sine Wave	-22.53543921	0.181477727	-22.558463	3.085485	4826	249500
Ackley One	316.1918392	10.59070695	313.026707	57.386537	5480	249500
Ackley Two	478.907586	5.039585632	477.040548	18.644082	3012	249500
Egg Holder	-5219.357186	620.6480812	-5716.940012	1395.794881	4396	249500
Rana	-9210.539488	3617.378882	-10701.2924	8926.023049	3261	249500
Pathological	-24.18549	6.939245082	-27.997573	27.490333	5136	249500
Michalewicz	-9.006188134	0.265701026	-9.095885	1.767938	2317	249500
Masters Cosine Wave	-3.516495559	1.453906749	-3.218388	3.744591	4404	249500
Quartic	2712809381	62820052.67	2705143081	651825554	882	249500
Levy	694.869074	0.318843792	694.816117	2.706602	940	249500
Step	408.8517643	21.84376109	393.922336	122.058311	864	249500
Alpine	9450.753823	542.6975577	9306.506257	3152.132133	4380	249500

Table 1.6: Differential Evolution Algorithm Best Strategies Analytics

Average	StdDev	Time(msec)	DE strategy (1-10)
7920.003093	68.94682032	77	4
11849.20525	22.94100415	80	4
11845.59715	73.0235972	73	5
12287.11656	1.27E-11	74	5
288.633807	6.25E-13	78	1
-18.694878	1.42E-14	72	5
-8.563981	1.07E-14	78	1
340.52742	6.25E-13	72	2
492.736131	3.41E-13	80	2
-3858.456528	5.46E-12	84	2
-2304.712093	2.73E-12	91	9
3.186096	1.78E-15	77	4
-3.044226	0	96	8
-0.700846	9.99E-16	90	3
11853.03971	83.27710427	107	10
587.990224	5.68E-13	86	6
505.100898	8.53E-13	74	4
11868.15807	80.08025673	88	9

	Name	$f(x^*)$	Dimensions	Range
f_1	Schwefel	0	10,20,30	$[-512, 512]^n$
f_2	De Jong 1	0	10,20,30	$[-100, 100]^n$
f_3	Rosenbrock's Saddle	0	10,20,30	$[-100, 100]^n$
f_4	Rastrigin	0	10,20,30	$[-30, 30]^n$
f_5	Griewangk	0	10,20,30	$[-500, 500]^n$
f_6	Sine Envelope Sine Wave	$-1.4915(n-1)$	10,20,30	$[-30, 30]^n$
f_7	Stretch V Sine Wave	0	10,20,30	$[-30, 30]^n$
f_8	Ackley One	$-7.54276 - 2.91867(n-3)$	10,20,30	$[-32, 32]^n$
f_9	Ackley Two	0	10,20,30	$[-32, 32]^n$
f_{10}	Egg Holder	–	10,20,30	$[-500, 500]^n$
f_{11}	Rana	–	10,20,30	$[-500, 500]^n$
f_{12}	Pathological	–	10,20,30	$[-100, 100]^n$
f_{13}	Michalewicz	$0.966n$	10,20,30	$[0, \pi]^n$
f_{14}	Masters' Cosine Wave	$1-n$	10,20,30	$[-30, 30]^n$
f_{15}	Quartic	0	10,20,30	$[-100, 100]^n$
f_{16}	Levy	0	10,20,30	$[-10, 10]^n$
f_{17}	Step	0	10,20,30	$[-100, 100]^n$
f_{18}	Alpine	0	10,20,30	$[-100, 100]^n$

Table 1.7: Benchmark Functions.