A convergence comparison of three optimization strategies of varying population sizes

Shane Ward – P2510580
De Montfort University,
Leicester, England
P2510580@my365.dmu.ac.uk

ABSTRACT

By definition, an optimization algorithm is a method for iteratively comparing solutions until an optimum solution is obtained. This is achieved by comparing the result of a cost function on each iteration until a computational budget is met. In this paper, three optimization algorithms were deployed to the entirety of the CEC2014 Benchmark suite problems for various population sizes. The first implemented algorithm was the S algorithm, which is a single solution. The second algorithm implemented was the Differential Evolution (DE) algorithm, which is a population based solution. The final algorithm implemented was a memetic algorithm, which was a combination of the DE and S algorithms. Each algorithm was tested for a population of 10 dimensions (10D), 50 dimensions (50D) and finally 100 dimensions (100D). Once the results for each test were obtained, the convergence of each was compared and conclusions were drawn on the performance of each optimization strategy.

KEYWORDS: *Mathematical Optimization, Single Solution, Population Based Algorithm, Genetic Algorithms (GA), Evolution Algorithm (EA), Memetic Algorithms (MA)*

1 INTRODUCTION

As previously mentioned, mathematical optimization is a branch of mathematics which aims to solve the problem of locating the elements that maximise or minimise a given real-valued function. There are various useful applications of optimization algorithms. This realisation has contributed to the recent interest and further developments of optimization algorithms. There are two key variations of optimization algorithms, single solution algorithms and population based algorithms. Single solution algorithms are deterministic metaheuristics and are gradient free. Population based algorithms not only use a single solution to reach an optimum value but use a list of solutions in order to search for the optimum. In this paper, the single solution algorithm applied to the chosen benchmark was the S algorithm. It was first implemented in 1961 by two engineers, Hooke and Jeeves. The population based algorithm applied to the chosen benchmark was the Differential Evolution (DE) algorithm. The first implementation of Differential Evolution was in 1995, by Storn and Price.

Also implemented and tested in this paper was the design of a Memetic algorithm for optimization of the same benchmark. This is an extension of the S and DE algorithms. Memetic algorithms are composed mainly of an Evolutionary algorithm and use a local search technique to reduce the likelihood of the premature convergence. The implemented memetic algorithm consisted of a combination of both the single solution and population based algorithms. A comparison was made between the results of both for the CEC2014 benchmark. The performance of each algorithm on each problem of the CEC2014 benchmark was evaluated using the optimum value produced. The Wilcoxon test was also used for the statistical testing of the results for each algorithm on each problem of the CEC2014 benchmark, for which there are 30. Finally, discussed in this report, is the conclusion on which optimization algorithm is most suitable for the corresponding problem based on the results.

2 BENCHMARK

The CEC2014 Benchmark was created in July 2014 in Beijing for a competition on real-parameter single objective optimization. The principle function of the Benchmark is to provide a common reference point for the performance of all optimization algorithms which are developed by the participants of the competition. The Benchmark consists of 30 different problems, which are shown below in Figure 1. It can also be seen that the 30 problems are arranged into four main categories:

Unimodel functions: Functions which monotonically increase for some value x and monotonically decrease for some value x.

Simple Multimodel functions: Functions which contain several modes and increase and decrease for some value x.

Hybrid functions: Functions which are defined by multiple sub functions, where each sub function applies to an interval of the main domain of the function.

Composition functions: Functions, as the name suggests, which takes two functions f and g and produces a function h such that h(x) = g(f(x**))**.

Category	**No.**	**Functions**	***Fi*=Fi(x*)***
Unimodal Functions	1	Rotated High Conditioned Elliptic Function	100
	2	Rotated Bent Cigar Function	200
	3	Rotated Discus Function	300
Simple Multimodal Functions	4	Shifted and Rotated Rosenbrock’s Function	400
	5	Shifted and Rotated Ackley’s Function	500
	6	Shifted and Rotated Weierstrass Function	600
	7	Shifted and Rotated Griewank’s Function	700
	8	Shifted Rastrigin’s Function	800
	9	Shifted and Rotated Rastrigin’s Function	900
	10	Shifted Schwefel’s Function	1000
	11	Shifted and Rotated Schwefel’s Function	1100
	12	Shifted and Rotated Katsuura Function	1200
	13	Shifted and Rotated HappyCat Function	1300
	14	Shifted and Rotated HGBat Function	1400
	15	Shifted and Rotated Expanded Griewank’s plus Rosenbrock’s Function	1500
	16	Shifted and Rotated Expanded Scaffer’s F6 Function	1600
Hybrid Functions	17	Hybrid Function 1 (*N*=3)	1700
	18	Hybrid Function 2 (*N*=3)	1800
	19	Hybrid Function 3 (*N*=4)	1900
	20	Hybrid Function 4 (*N*=4)	2000
	21	Hybrid Function 5 (*N*=5)	2100
	22	Hybrid Function 6 (*N*=5)	2200
Composition Functions	23	Composition Function 1 (*N*=5)	2300
	24	Composition Function 2 (*N*=3)	2400
	25	Composition Function 3 (*N*=3)	2500
	26	Composition Function 4 (*N*=5)	2600
	27	Composition Function 5 (*N*=5)	2700
	28	Composition Function 6 (*N*=5)	2800
	29	Composition Function 7 (*N*=3)	2900
	30	Composition Function 8 (*N*=3)	3000
Search Range: [-100,100]*D*			

Figure 1 – Summary of CEC2014 Test Functions

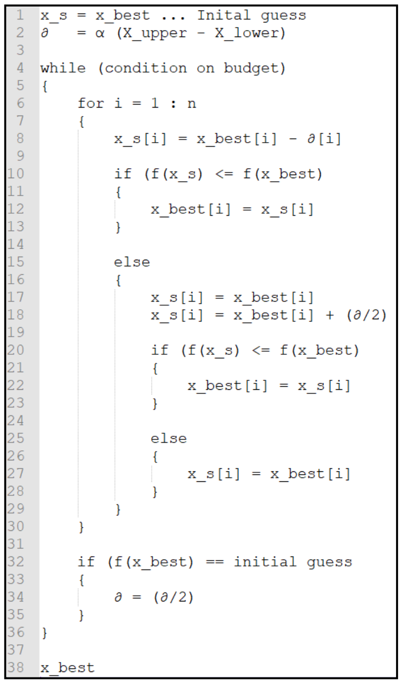
3 ALGORITHM DESIGN

3.1 S Algorithm

The S optimization algorithm is a single solution, deterministic metaheuristic. Its first implementation was by Hooke-Jeeves in 1961, and was subsequently revisited by Tseng and Chen in 2008 where the name S was given. The main features of the algorithm are that it performs the perturbation of each variable and selects the most convenient solution at the end of the exploration. The working principle of the S algorithm is as follows:

- Initial point is sampled within the search space.
- An exploratory radius is initialised as 40% of the length of the search space.
- The design variables of the solution are perturbed one by one.
- If no improvement occurs by the end of the iteration, the exploratory radius is halved.
- The algorithm continues until the set limit of the computational budget is met.

The pseudocode of the implemented S algorithm is shown in Figure 2 below.



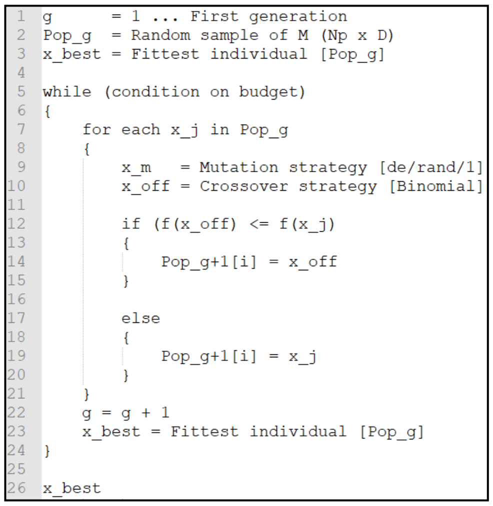
```
x_s = x_best ... Inital guess
∂   = α (X_upper - X_lower)

while (condition on budget)
{
    for i = 1 : n
    {
        x_s[i] = x_best[i] - ∂[i]

        if (f(x_s) <= f(x_best)
        {
            x_best[i] = x_s[i]
        }

        else
        {
            x_s[i] = x_best[i]
            x_s[i] = x_best[i] + (∂/2)

            if (f(x_s) <= f(x_best)
            {
                x_best[i] = x_s[i]
            }

            else
            {
                x_s[i] = x_best[i]
            }
        }
    }

    if (f(x_best) == initial guess
    {
        ∂ = (∂/2)
    }
}

x_best
```

Figure 2 – S Algorithm Pseudo code

3.2 Differential Evolution Algorithm

The Differential Evolution (DE) algorithm is a population based solution which shares some common features of Evolutionary Algorithms and Swarm Intelligence Algorithms. It was defined in 1995 by Storn and Price to address the Chebyshev polynomial fitting problem but it was soon discovered that it was useful for other applications. The algorithm is very efficient and performs well for real-values optimization. The working principle of the DE algorithm is as follows:

- Initial M n-dimensional points sampled within the search space.
- Each point goes through mutation to produce a mutant vector. The method used in this implementation was Original Mutation i.e. DE/rand/1. Each point also goes through a crossover method. In this implementation, Binomial crossover was applied
- Individuals are perturbed one by one.
- Once all offspring of the population are generated, 1-to-1 spawning is applied for survivor selection.



```
g      = 1 ... First generation
Pop_g  = Random sample of M (Np x D)
x_best = Fittest individual [Pop_g]

while (condition on budget)
{
    for each x_j in Pop_g
    {
        x_m   = Mutation strategy [de/rand/1]
        x_off = Crossover strategy [Binomial]

        if (f(x_off) <= f(x_j)
        {
            Pop_g+1[i] = x_off
        }

        else
        {
            Pop_g+1[i] = x_j
        }
    }
    g = g + 1
    x_best = Fittest individual [Pop_g]
}

x_best
```

Figure 3 – Differential Evolution Pseudo code

3.3 Memetic Algorithm

The name "Memetic algorithm" is commonly given to an optimization algorithm which consists of both an Evolutionary Algorithm (EA) and a local search operator which functions within the generation loop. Currently, Memetic algorithms are considered the state-of-the-art in the field of optimization due to their ability to be much faster and more accurate on certain problems. The general scheme of a Memetic algorithm is detailed in Figure 4 below.

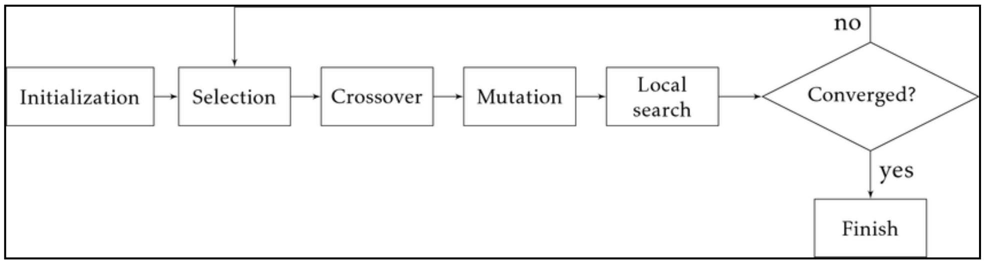
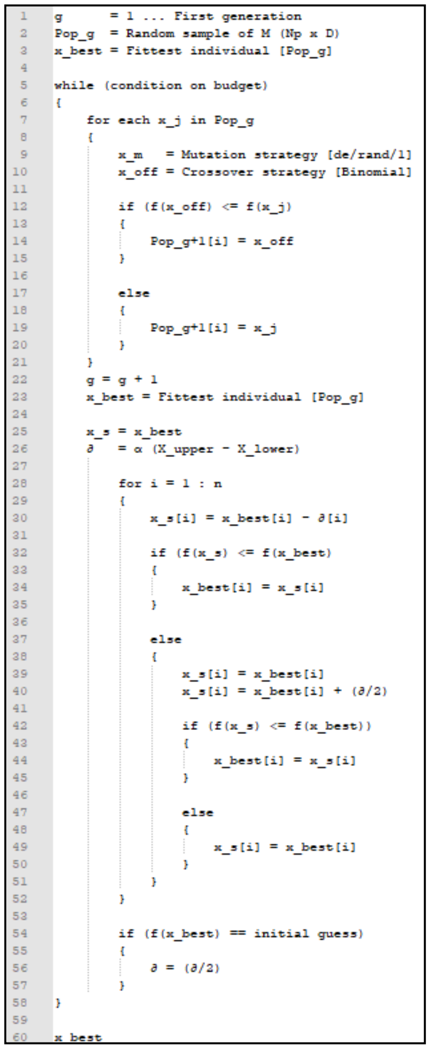


Figure 4 – General scheme of a Memetic Algorithm (MA)

The memetic algorithm implemented in this paper used both the S algorithm and the DE algorithm, therefore combining a single solution deterministic algorithm with a population based algorithm. The algorithm starts with DE algorithm before combining the results with the local search S algorithm. The working principle of the implemented algorithmic steps are defined as follows:

- Parent Selection: Aims to identify solutions that will survive in future generations and be utilised in the creation of new solutions.
- Parent Combination – Offspring Generation: Aims to create new solutions by combining existing solutions, i.e. the selected parents, to guide the process towards the selection of new search areas with potential for better solutions.
- Local Offspring Improvement: Aims to improve the quality of an offspring as much as possible.
- Population Update: Here, the decision on whether a solution should become part of the population is made as well as the decision on which existing solution is to be replaced in the population.



```
g      = 1 ... First generation
Pop_g  = Random sample of M (Np x D)
x_best = Fittest individual [Pop_g]

while (condition on budget)
{
    for each x_j in Pop_g
    {
        x_m   = Mutation strategy [de/rand/1]
        x_off = Crossover strategy [Binomial]

        if (f(x_off) <= f(x_j)
        {
            Pop_g+1[i] = x_off
        }

        else
        {
            Pop_g+1[i] = x_j
        }
    }
    g = g + 1
    x_best = Fittest individual [Pop_g]

    x_s = x_best
    ∂   = α (X_upper - X_lower)

        for i = 1 : n
        {
            x_s[i] = x_best[i] - ∂[i]

            if (f(x_s) <= f(x_best)
            {
                x_best[i] = x_s[i]
            }

            else
            {
                x_s[i] = x_best[i]
                x_s[i] = x_best[i] + (∂/2)

                if (f(x_s) <= f(x_best))
                {
                    x_best[i] = x_s[i]
                }

                else
                {
                    x_s[i] = x_best[i]
                }
            }
        }

        if (f(x_best) == initial guess)
        {
            ∂ = (∂/2)
        }
}

x_best
```

Figure 5 – Memetic Algorithm Pseudo code

4 RESULTS

4.1 Experimental Setup

The SOS platform was used to implement the three optimization algorithms mentioned in Section 3. In order to test the performance of each, an experiment was created. The experiment consisted of running each of the three optimization algorithms (S, DE & DES) on the 30 problems of the CEC2014 Benchmark. The experiment was run three times, using the following parameters:

Run 1:
- No. Runs = 30
- D = 5000
- N` = 10 * D

Run 2:
- No. Runs = 30
- D = 5000
- N` = 50 * D

Run 3:
- No. Runs = 30
- D = 5000
- N` = 100 * D

4.2 Performance comparison

	S Algorithm	DE Algorithm	W	DES Algorithm	W
1	3.762e+09 ± 1.431e-06	5.494e+09 ± 9.537e-07	+	4.108e+09± 1.907e-06	+
2	7.891e+10 ± 1.526e-05	5.443e+10 ± 3.052e-05	-	2.350e+10 ± 3.815e-06	-
3	6.418e+09 ± 0.000e+00	7.915e+09 ± 1.907e-06	+	5.642e+08 ± 0.000e+00	-
4	2.991e+04 ± 1.455e-11	1.758e+04 ± 7.276e-12	-	1.339e+04 ± 1.819e-12	-
5	5.217e+02 ± 2.274e-13	5.219e+02 ± 2.274e-13	+	5.218e+02 ± 3.411e-13	+
6	6.239e+02 ± 4.547e-13	6.198e+02 ± 1.137e-13	-	6.165e+02 ± 1.137e-13	-
7	1.081e+03 ± 4.547e-13	2.317e+03 ± 1.364e-12	+	1.060e+03 ± 0.000e+00	-
8	1.109e+03 ± 4.547e-13	1.049e+03 ± 4.547e-13	-	1.015e+03 ± 5.684e-13	-
9	1.143e+03 ± 6.821e-13	1.214e+03 ± 6.821e-13	+	1.111e+03 ± 0.000e+00	-
10	6.159e+03 ± 2.728e-12	5.458e+03 ± 9.095e-13	-	4.409e+03 ± 0.000e+00	-
11	4.575e+03 ± 1.819e-12	5.040e+03 ± 2.728e-12	+	5.954e+03 ± 2.728e-12	+
12	1.212e+03 ± 4.547e-13	1.222e+03 ± 6.821e-13	+	1.213e+03 ± 4.547e-13	+
13	1.314e+03 ± 2.274e-13	1.313e+03 ± 0.000e+00	-	1.309e+03 ± 6.821e-13	-
14	1.527e+03 ± 6.821e-13	1.716e+03 ± 6.821e-13	+	1.448e+03 ± 0.000e+00	-
15	1.653e+07 ± 9.313e-09	9.332e+06 ± 3.725e-09	-	4.554e+06 ± 1.863e-09	-
16	1.605e+03 ± 2.274e-13	1.605e+03 ± 2.274e-13	+	1.605e+03 ± 2.274e-13	-
17	1.492e+10 ± 9.537e-06	4.209e+09 ± 0.000e+00	-	3.577e+09 ± 1.431e-06	-
18	8.043e+09 ± 2.861e-06	1.212e+10 ± 1.907e-06	+	6.183e+08 ± 4.768e-07	-
19	4.673e+03 ± 2.728e-12	1.544e+04 ± 5.457e-12	+	2.271e+03 ± 4.547e-13	-
20	1.837e+10 ± 7.629e-06	1.954e+08 ± 0.000e+00	-	9.025e+09 ± 3.815e-06	-
21	1.600e+09 ± 4.768e-07	2.054e+10 ± 3.815e-06	+	4.512e+08 ± 1.788e-07	-
22	6.751e+06 ± 0.000e+00	1.449e+06 ± 9.313e-10	-	3.678e+06 ± 1.397e-09	-
23	4.914e+03 ± 2.728e-12	1.033e+04 ± 5.457e-12	+	3.638e+03 ± 9.095e-13	-
24	6.587e+03 ± 4.547e-12	2.722e+03 ± 2.274e-12	-	2.894e+03 ± 0.000e+00	-
25	2.809e+03 ± 0.000e+00	2.724e+03 ± 1.364e-12	-	2.758e+03 ± 1.364e-12	-
26	3.564e+03 ± 1.364e-12	3.388e+03 ± 0.000e+00	-	3.040e+03 ± 0.000e+00	-
27	9.816e+03 ± 7.276e-12	4.025e+03 ± 1.364e-12	-	6.740e+03 ± 1.819e-12	-
28	1.512e+04 ± 1.091e-11	7.767e+03 ± 2.728e-12	-	1.036e+04 ± 9.095e-12	-
29	2.868e+08 ± 1.788e-07	3.678e+09 ± 2.384e-06	+	8.425e+08 ± 2.384e-07	+
30	4.043e+06 ± 1.863e-09	2.469e+08 ± 1.192e-07	+	5.270e+07 ± 2.980e-08	+

Figure 6 – Results for n = 10*D

	S Algorithm	DE Algorithm	W	DES Algorithm	W
1	4.876e+10 ± 1.526e-05	2.856e+10 ± 7.629e-06	-	1.420e+10 ± 3.815e-06	-
2	9.826e+11 ± 3.662e-04	4.504e+11 ± 1.831e-04	-	2.335e+11 ± 6.104e-05	-
3	1.479e+09 ± 7.153e-07	1.513e+09 ± 2.384e-07	+	1.582e+09 ± 4.768e-07	+
4	1.459e+06 ± 4.657e-10	4.390e+05 ± 1.746e-10	-	5.538e+04 ± 3.638e-11	-
5	5.217e+02 ± 1.137e-13	5.218e+02 ± 2.274e-13	+	5.216e+02 ± 1.137e-13	-
6	7.059e+02 ± 2.274e-13	6.908e+02 ± 0.000e+00	-	6.948e+02 ± 3.411e-13	-
7	1.202e+04 ± 5.457e-12	6.516e+03 ± 1.819e-12	-	2.915e+03 ± 1.364e-12	-
8	2.752e+03 ± 1.364e-12	2.038e+03 ± 6.821e-13	-	1.816e+03 ± 2.274e-13	-
9	3.932e+03 ± 0.000e+00	2.601e+03 ± 9.095e-13	-	1.927e+03 ± 4.547e-13	-
10	2.349e+04 ± 0.000e+00	2.312e+04 ± 1.091e-11	-	2.098e+04 ± 1.091e-11	-
11	1.932e+04 ± 1.455e-11	1.892e+04 ± 0.000e+00	-	2.231e+04 ± 0.000e+00	+
12	1.213e+03 ± 6.821e-13	1.212e+03 ± 2.274e-13	-	1.215e+03 ± 4.547e-13	+
13	1.334e+03 ± 2.274e-13	1.319e+03 ± 4.547e-13	-	1.311e+03 ± 4.547e-13	-
14	4.908e+03 ± 2.728e-12	3.199e+03 ± 1.364e-12	-	1.799e+03 ± 1.137e-12	-
15	1.256e+11 ± 6.104e-05	1.037e+10 ± 0.000e+00	-	4.233e+07 ± 1.490e-08	-
16	1.625e+03 ± 2.274e-13	1.625e+03 ± 6.821e-13	+	1.625e+03 ± 2.274e-13	+
17	5.963e+10 ± 2.289e-05	2.298e+10 ± 1.907e-05	-	7.072e+09 ± 0.000e+00	-
18	1.955e+11 ± 3.052e-05	1.050e+11 ± 4.578e-05	-	3.434e+10 ± 0.000e+00	-
19	8.508e+04 ± 2.910e-11	2.884e+04 ± 1.455e-11	-	1.386e+04 ± 3.638e-12	-
20	2.087e+10 ± 0.000e+00	7.758e+09 ± 2.861e-06	-	3.652e+07 ± 2.235e-08	-
21	3.180e+10 ± 1.144e-05	1.473e+10 ± 0.000e+00	-	9.829e+08 ± 2.384e-07	-
22	5.481e+08 ± 1.192e-07	8.015e+07 ± 1.490e-08	-	2.824e+07 ± 1.118e-08	-
23	3.994e+04 ± 0.000e+00	1.530e+04 ± 0.000e+00	-	6.037e+03 ± 9.095e-13	-
24	4.853e+03 ± 1.819e-12	3.675e+03 ± 1.364e-12	-	3.388e+03 ± 0.000e+00	-
25	8.384e+03 ± 3.638e-12	4.734e+03 ± 9.095e-13	-	2.981e+03 ± 2.274e-12	-
26	5.890e+03 ± 9.095e-13	5.445e+03 ± 1.819e-12	-	5.303e+03 ± 9.095e-13	-
27	1.963e+04 ± 1.455e-11	1.464e+04 ± 5.457e-12	-	4.175e+04 ± 2.183e-11	+
28	3.341e+04 ± 0.000e+00	2.752e+04 ± 1.455e-11	-	3.874e+04 ± 1.455e-11	+
29	1.964e+09 ± 9.537e-07	5.365e+09 ± 2.861e-06	+	6.988e+09 ± 3.815e-06	+
30	3.584e+09 ± 2.384e-06	2.082e+08 ± 1.490e-07	-	2.828e+08 ± 1.788e-07	-

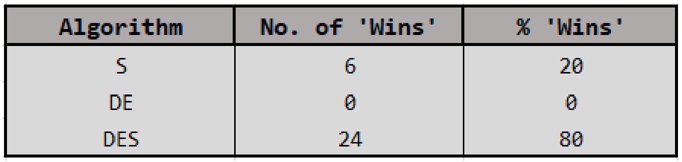
Figure 7 – Results for n = 50*D

	S Algorithm	DE Algorithm	W	DES Algorithm	W
1	9.023e+10 ± 1.526e-05	5.549e+10 ± 1.526e-05	-	2.793e+10 ± 1.907e-05	-
2	1.060e+12 ± 7.324e-04	8.698e+11 ± 3.662e-04	-	4.438e+11 ± 3.052e-04	-
3	6.058e+09 ± 2.861e-06	2.238e+09 ± 1.431e-06	-	1.305e+09 ± 4.768e-07	-
4	9.226e+05 ± 3.492e-10	5.167e+05 ± 1.746e-10	-	2.145e+05 ± 2.910e-11	-
5	5.216e+02 ± 1.137e-13	5.217e+02 ± 2.274e-13	+	5.218e+02 ± 2.274e-13	+
6	8.030e+02 ± 3.411e-13	7.867e+02 ± 2.274e-13	-	7.890e+02 ± 4.547e-13	+
7	7.489e+03 ± 2.728e-12	9.258e+03 ± 1.819e-12	+	1.007e+04 ± 7.276e-12	+
8	3.889e+03 ± 1.364e-12	3.298e+03 ± 4.547e-13	-	3.556e+03 ± 9.095e-13	+
9	3.961e+03 ± 1.819e-12	3.533e+03 ± 2.274e-12	-	3.968e+03 ± 2.728e-12	+
10	4.077e+04 ± 7.276e-12	4.489e+04 ± 7.276e-12	+	4.354e+04 ± 7.276e-12	-
11	4.048e+04 ± 1.455e-11	4.059e+04 ± 1.455e-11	+	4.171e+04 ± 7.276e-12	+
12	1.213e+03 ± 0.000e+00	1.211e+03 ± 4.547e-13	-	1.210e+03 ± 4.547e-13	-
13	1.322e+03 ± 9.095e-13	1.321e+03 ± 4.547e-13	-	1.319e+03 ± 9.095e-13	-
14	4.247e+03 ± 1.819e-12	4.071e+03 ± 1.364e-12	-	3.895e+03 ± 1.364e-12	-
15	1.501e+09 ± 2.384e-07	1.888e+09 ± 1.192e-06	+	1.982e+09 ± 9.537e-07	+
16	1.648e+03 ± 6.821e-13	1.650e+03 ± 0.000e+00	+	1.650e+03 ± 2.274e-13	+
17	1.335e+10 ± 7.629e-06	6.598e+09 ± 1.907e-06	-	1.172e+10 ± 1.907e-06	-
18	1.418e+11 ± 0.000e+00	1.988e+11 ± 0.000e+00	+	2.040e+11 ± 1.221e-04	+
19	1.058e+05 ± 1.455e-11	9.006e+04 ± 1.455e-11	-	9.467e+04 ± 2.910e-11	-
20	7.043e+08 ± 1.192e-07	7.794e+09 ± 3.815e-06	+	1.257e+10 ± 5.722e-06	+
21	5.089e+09 ± 1.907e-06	-1.330e+10 ± 0.000e+00	+	1.221e+10 ± 1.907e-06	+
22	8.948e+08 ± 1.192e-07	2.019e+08 ± 8.941e-08	-	3.059e+08 ± 1.788e-07	-
23	1.706e+04 ± 7.276e-12	2.690e+04 ± 1.819e-11	+	1.983e+04 ± 3.638e-12	+
24	4.803e+03 ± 0.000e+00	5.268e+03 ± 3.638e-12	+	4.965e+03 ± 2.728e-12	+
25	5.459e+03 ± 9.095e-13	6.090e+03 ± 0.000e+00	+	7.066e+03 ± 4.547e-12	+
26	4.718e+03 ± 0.000e+00	5.440e+03 ± 0.000e+00	+	5.393e+03 ± 9.095e-13	+
27	3.642e+04 ± 1.455e-11	1.713e+04 ± 0.000e+00	-	1.360e+04 ± 1.819e-12	-
28	7.398e+04 ± 1.455e-11	8.527e+04 ± 7.276e-11	+	6.599e+04 ± 0.000e+00	-
29	2.773e+10 ± 0.000e+00	1.625e+10 ± 1.144e-05	-	1.451e+10 ± 7.629e-06	-
30	1.438e+09 ± 4.768e-07	2.030e+09 ± 0.000e+00	+	2.388e+09 ± 4.768e-07	+

Figure 8 – Results for n = 100*D

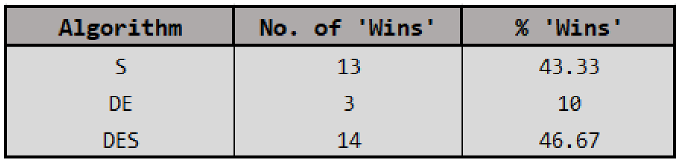
4.3 Statistical Analysis

Following the three experiments conducted and the resulting obtained data, there are some clear assumptions that can be made on the S, DE and DES optimization algorithms. The results for the 30 different experiments are shown in the below Figures (9, 10 & 11).



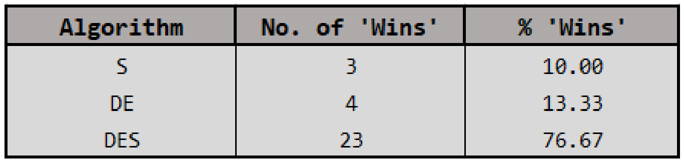
Algorithm	No. of 'Wins'	% 'Wins'
S	6	20
DE	0	0
DES	24	80

Figure 9 – Number of 'Wins' for each algorithm at N = 10 * D



Algorithm	No. of 'Wins'	% 'Wins'
S	13	43.33
DE	3	10
DES	14	46.67

Figure 10 – Number of 'Wins' for each algorithm at N = 50 * D



Algorithm	No. of 'Wins'	% 'Wins'
S	3	10.00
DE	4	13.33
DES	23	76.67

Figure 11 – Number of 'Wins' for each algorithm at N = 100 * D

Comparing the performance for each of the 4 different categories of functions in the CEC2014 Benchmark (Unimodal, Multimodal Hybrid & Composition functions) also provided us with a clear description of how each of the optimization algorithms performed. These results are tabulated in the below Figures (12, 13 & 14).

Algorithm	Unimodal	Multimodal	Hybrid	Composite
S	1	3	0	2
DE	0	0	0	0
DES	2	10	6	6

Figure 12 – Number of 'Wins' per Benchmark category at N = 10 * D

Algorithm	Unimodal	Multimodal	Hybrid	Composite
S	1	1	0	1
DE	0	2	0	2
DES	2	10	6	5

Figure 13 – Number of 'Wins' per Benchmark category at N = 50 * D

Algorithm	Unimodal	Multimodal	Hybrid	Composite
S	0	5	3	5
DE	0	3	0	0
DES	3	5	3	3

Figure 14 – Number of 'Wins' per Benchmark category at N = 100 * D

5 CONCLUSION

In this paper, three optimization algorithms were developed and tested on the CEC2014 Benchmark problems. The first optimization algorithm was the 'S' Algorithm, the second optimization algorithm was the Differential Evolution 'DE' Algorithm and the third optimization algorithm was a Memetic Algorithm 'DES' consisting of a combination of both the 'S' and 'DE' algorithms. As previously discussed, three experiments were run using the SOS software platform to compare the performance of each optimization algorithm, the first for N = 10 * D, the second for N = 50 * D and finally the third for N = 100 * D. The results of each experiment were documented in Section 4 and the following conclusions were drawn.

- Overall, the 'DES' memetic algorithm outperformed the 'S' and 'DE' algorithms quite considerably across all 30 problems in the CEC2014 Benchmark, with a 'Win' percentage of 80% (N = 10*D), 46.67% (N = 50*D) and 76.67% (N = 100*D) in each case.
- Upon further inspection of results for each of the Benchmark categories, a number of conclusions were drawn:
    - Unimodal: 'DES' performed best for 10*D, 50*D and 100*D on Unimodal problems with 'S' performing second best and 'DE' performing the worst.
    - Multimodal: 'DES' performed best for 10*D and 50*D for Multimodal problems, however as the dimensionality was increased the 'DES' performance decreased while the performance of both 'S' and 'DE' increased.
    - Hybrid: Similar to the Multimodal results, 'DES' performed best for 10*D and 50*D, however for 100*D both 'DES' and 'S' had the same number of wins.
    - Composite: 'DES' again performed best for 10*D and 50*D, however for 100*D the 'S' algorithm had 5 wins compared to 3 wins for 'DES'.

  It is clear that the 'DES' algorithm had the best performance, but an interesting finding was that the performance of the 'S' algorithm outperformed 'DES' in some Benchmark categories as the dimensionality was increased to 100*D.

In conclusion, the results present a strong argument that the design of the Memetic Algorithm (MA) 'DES' is capable of finding the local minimum very well in comparison with a single solution or population based algorithm alone. This was completed by combining the population-based algorithm (DE) with the local search operator (S). As the dimensionality was increased to 100*D, it is also clear that the performance of the 'DES' algorithm began to decline. Future works in relation to this topic are to be completed in varying the local search operator and the trialling of different tuning parameters such as mutation and crossover strategies with the objective of improving the results.

6 APPENDICES

APPENDIX I

Implementation of S Algorithm in java

```java
double[] x_s = x_best;
double radius = 0.4 * (bounds[i][1] - bounds[i][0]);

while (i < maxEvaluations) // Condition on budget = No of Runs
{
      for (int j = 1; j < problemDimension; j++)
      {
            oldfFParticle = fParticle;
            posOld = x_best[j];

            x_s[j] = x_best[j] - radius;

            x_s = toro(x_s, bounds);//Used to saturate solution if outside bounds
            fParticle = problem.f(x_s);
            i++; // Increment counter

            if (fParticle <= problem.f(x_best))
            {
                  x_best[j] = x_s[j];
            }
            else
            {
                  x_s[j] = x_best[j];
                  x_s[j] = x_best[j] + (radius/2);

                  if (fParticle <= problem.f(x_best))
                  {
                        x_best[j] = x_s[j];
                  }
                  else
                  {
                        x_s[j] = x_best[j];
                  }
            }
      }
      if (problem.f(x_best) == oldfFParticle)
      {
            radius = (radius/2);
      }
      FT.add(i, problem.f(x_best)); // Add to fitness trend
}
```

APPENDIX II

Implementation of DE Algorithm in java

```
// Initial Random Sample of Population
for (int j = 0; j < Pop_g; j++)
{
      double[] temp = generateRandomSolution(bounds, probDim);
      for (int n = 0; n < probDim; n++)
            Pop_g[j][n] = temp[n];
            fitnesses[j] = problem.f(Pop_g[j]);

      if (j == 0 || fitnesses[j] < fBest)
      {
            fBest = fitnesses[j];
            for (int n = 0; n < probDim; n++)
                  best[n] = Pop_g[j][n];
      }
      i++;
}

// Temporary variable
double[] CurrentParticle = new double[probDim];
double[] NewParticle = new double[probDim];
double[] CrossoverParticle = new double[probDim];
double CurrentFitness = Double.NaN;
double CrossoverFitness = Double.NaN;

// While I < maximum number of evaluations
while (i < maxEvaluations)
{
      double[][] NewPopulation = new double[Pop_g][probDim];

      for (int j = 0; j < Pop_g && i < maxEvaluations; j++)
      {
            for (int n = 0; n < probDim; n++)
                  CurrentParticle[n] = Pop_g[j][n];
                  CurrentFitness = fitnesses[j];

            // Mutation Strategy //
            double x[] = {1};
            double y[] = {1};
            int n = x.length;
            double[] x_off = new double[n];
            int index = RandUtils.randomInteger(n-1);
            for (int o = 0; o < n; o++)
            {
                  if (RandUtils.random() < CR || i == index)
                        x_off[i] = y[i];
                  else
                        x_off[i] = x[i];
            }

            // Crossover strategy //
            for (int k = 0; k < n; k++)
            {
                  if (RandUtils.random() < CR || i == index)
                        x_off[k] = y[k];
                  else
                        x_off[k] = x[k];
            }
```

```
                CrossoverParticle = toro(CrossoverParticle, bounds);
                CrossoverFitness = problem.f(CrossoverParticle);
                i++;

                // Replacement of value
                if (CrossoverFitness < CurrentFitness)
                {
                        for (int l = 0; l < probDim; n++)
                                NewPopulation[l][n] = CrossoverParticle[n];
                                fitnesses[j] = CrossoverFitness;

                        // Update the best value
                        if (CrossoverFitness < fBest)
                        {
                                fBest = CrossoverFitness;
                                for (int k = 0; k < probDim; k++)
                                        best[k] = CrossoverParticle[k];
                        }

                }
                else
                {
                        for (int m = 0; m < probDim; m++)
                                NewPopulation[m][n] = CurrentParticle[n];
                                fitnesses[j] = CurrentFitness;
                }

                CrossoverParticle = null;
                NewParticle = null;

        }

        Pop_g = NewPopulation;
        NewPopulation = null;

}

finalBest = best;
FT.add(1, fBest);

return FT;
```

APPENDIX III

Implementation of DES Algorithm in java

```
// Initial Random Sample of Population
for (int j = 0; j < Pop_g; j++)
{
      double[] temp = generateRandomSolution(bounds, probDim);
      for (int n = 0; n < probDim; n++)
            Pop_g[j][n] = temp[n];
            fitnesses[j] = problem.f(Pop_g[j]);

      if (j == 0 || fitnesses[j] < fBest)
      {
            fBest = fitnesses[j];
            for (int n = 0; n < probDim; n++)
                  best[n] = Pop_g[j][n];
      }
      i++;
}

// Temporary variable
double[] CurrentParticle = new double[probDim];
double[] NewParticle = new double[probDim];
double[] CrossoverParticle = new double[probDim];
double CurrentFitness = Double.NaN;
double CrossoverFitness = Double.NaN;

// While I < maximum number of evaluations
while (i < maxEvaluations)
{
      double[][] NewPopulation = new double[Pop_g][probDim];

      for (int j = 0; j < Pop_g && i < maxEvaluations; j++)
      {
            for (int n = 0; n < probDim; n++)
                  CurrentParticle[n] = Pop_g[j][n];
                  CurrentFitness = fitnesses[j];

            // Mutation Strategy //
            double x[] = {1};
            double y[] = {1};
            int n = x.length;
            double[] x_off = new double[n];
            int index = RandUtils.randomInteger(n-1);
            for (int o = 0; o < n; o++)
            {
                  if (RandUtils.random() < CR || i == index)
                        x_off[i] = y[i];
                  else
                        x_off[i] = x[i];
            }

            // Crossover strategy //
            for (int k = 0; k < n; k++)
            {
                  if (RandUtils.random() < CR || i == index)
                        x_off[k] = y[k];
                  else
                        x_off[k] = x[k];
            }
```

```
		CrossoverParticle = toro(CrossoverParticle, bounds);
		CrossoverFitness = problem.f(CrossoverParticle);
		i++;

		// Replacement of value
		if (CrossoverFitness < CurrentFitness)
		{
			for (int l = 0; l < probDim; n++)
				NewPopulation[l][n] = CrossoverParticle[n];
				fitnesses[j] = CrossoverFitness;

			// Update the best value
			if (CrossoverFitness < fBest)
			{
				fBest = CrossoverFitness;
				for (int k = 0; k < probDim; k++)
					best[k] = CrossoverParticle[k];
			}

		}
		else
		{
			for (int m = 0; m < probDim; m++)
				NewPopulation[m][n] = CurrentParticle[n];
				fitnesses[j] = CurrentFitness;
		}

		CrossoverParticle = null;
		NewParticle = null;

	}

	Pop_g = NewPopulation;
	NewPopulation = null;


	double oldfFParticle = fBest;
	double[] x_s = best;
	double posOld;
	double radius = 0.4 * (bounds[j][1] - bounds[j][0]);
	double fPart = problem.f(best);

	for (int k = 1; k < problemDimension; k++)
	{
		oldfFParticle = fBest;
		posOld = best[k];

		x_s[k] = best[k] - radius;

		x_s = toro(x_s, bounds);
		fBest = problem.f(x_s);
		i++;

		if (fBest < fPart)
		{
			 best[j] = x_s[j];
		}
		else
		{
			 x_s[j] = best[j];
			 x_s[j] = best[j] + (radius/2);
```

```
                    if (fBest < fPart)
                    {
                          best[j] = x_s[j];
                    }

                    else
                    {
                          x_s[j] = best[j];
                    }
              }

              if (fPart == oldfFParticle)
              {
                    radius = (radius/2);
              }
        }

        Pop_g = NewPopulation;
        NewPopulation = null;

  }

  finalBest = best;
  FT.add(1, fBest);

  return FT;
```

REFERENCES

[1] Caraffini, F., Neri, F., and Picinali, L. (2014), "An analysis on separability for memetic computing automatic design." Information Sciences, 265(0):1 – 22.

[2] Zaharie, D. (2002)., "Critical values for control parameters of differential evolution algorithm.", In Matuˆsek, R. and Oˆsmera, P., editors, Proceedings of 8th International Mendel Conference on Soft Computing, pages 62–67.

[3] Tirronen, V., Neri, F., and Rossi, T. (2009). "Enhancing differential evolution frameworks by scale factor local search - part I.", In Proceedings of the IEEE Congress on Evolutionary Computation.

[4] J. J. Liang, B. Y. Qu and P. N. Suganthan, "Problem Definitions and Evaluation Criteria for the CEC 2014 Special Session and Competition on Single Objective Real-Parameter Numerical Optimization", Technical Report 201311, Computational Intelligence Laboratory, December 2013.

[5] F. Caraffini, "Algorithmic Issues in Computational Intelligence Optimization From Design to Implementation, from Implementation to Design", September 2016.