2022

Computational Intelligence Optimisation

Task 2

ibrahim ahmethan P2699379

Contents

Introduction

In this given task 3 types of single solution optimization algorithms will be implemented and tested on 4 different benchmark models taking into consideration the toro correction function to ensure that the search algorithm is within the boundaries of search space, these benchmark models are different in terms of complexity and pattern, which will result in different performance for each type of single solution optimization algorithms due to their nature and diverse working principles of these search algorithms, the 4 benchmarks are as the following, De jong, Rastrigin, Schwefel and Michalewicz and the 3 types of single solution algorithms are Intelligent Single Particle(ISPO), Convariance Matrix Adaptation Evolutionary Strategy (CMAES) and Hooke Jeeves (S), note that the S algorithm, the 4 types of benchmarks and the toro correction function are not completed in SOS file, the aim of this task is to use the provided SOS file and complete the S algorithm, toro correction function and benchmarks in Java programming language then testing and comparing the performance of each search algorithms over the provided benchmarks.

Single solution optimizer

In this section of the task S optimization algorithm working principle explained and the missing parts of coding in SOS file are completed.

S (Hooke - Jeeves) Algorithm

S algorithm is single solution metaheuristic deterministic search algorithm (black box), used generally for short local search or optimization, S algorithm carries out perturbation for each value and chooses the most appropriate solution (fitness function) at the end of the exploration (with limited amount of computational budget), further more the algorithm performs orthogonal move along each axis and produces only one certain solution (predictable candidate solution) for every input.

S algorithm does not guaranty to converge into the global optimum since it is a metaheuristic algorithm, however the solution obtained might be close to the global optimum, moreover S algorithm is free from randomisation, which means that we will have the same output for every time we use the same input.

- The following code is the Java implementation of S optimization algorithm of task 2.

S Algorithm implementation in Java

```
public class S extends Algorithm
{
    public RunAndStore.FTrend execute(Problem problem, int maxEvaluations) throws
Exception {

       int dimension = problem.getDimension();
       double[] x_value_best = new double[dimension];
       double[][] boundaries = problem.getBounds();

        double learning_rate = getParameter("p0");

        RunAndStore.FTrend FT = new RunAndStore.FTrend();


        int n = 0;
        double f_value_best = Double.NaN;

        if (initialSolution != null) {
             x_value_best = initialSolution;
             f_value_best = initialFitness;

        }
        else {

             x_value_best = generateRandomSolution(boundaries, dimension);
             f_value_best = problem.f(x_value_best);
            FT.add(n, f_value_best);
            n++;
        }

        double f_algorithm = f_value_best;
        double[] x_algorithm = x_value_best;
        double[] exp_radius = new double[dimension];

        for (int i = 0; i < dimension; i++) {
             exp_radius[i] = learning_rate * (boundaries[i][1] - boundaries[i][0]);


        }
        while (n < maxEvaluations) {


            boolean improving = false;
            for (int i = 0; i < dimension && n < maxEvaluations; i++) {

             x_algorithm[i] = x_value_best[i] - exp_radius[i];
             x_algorithm = toro(x_algorithm, boundaries);
                f_algorithm = problem.f(x_algorithm);
                n++;
```

```
                if (n % dimension == 0) {
                    FT.add(n, f_value_best);
                }
                if (f_algorithm <= f_value_best) {

                    x_value_best[i] = x_algorithm[i];
                    f_value_best = f_algorithm;
                    improving = true;
                }
                else if (n < maxEvaluations) {

                    x_algorithm[i] = x_value_best[i] + (exp_radius[i] / 2);
                    x_algorithm = toro(x_algorithm, boundaries);
                    f_algorithm = problem.f(x_algorithm);
                    n++;

                    if (n % dimension == 0) {
                        FT.add(n, f_value_best);
                    }
                    if (f_algorithm <= f_value_best) {

                          x_value_best[i] = x_algorithm[i];
                          f_value_best = f_algorithm;
                        improving = true;
                    }
                    else {
                          x_algorithm[i] = x_value_best[i];
                    }
                }
            }
            if (!improving) {

                for (int i = 0; i < exp_radius.length; i++) {

                    exp_radius[i] = exp_radius[i] / 2;
                }
            }
        }
        finalBest = x_value_best;
        FT.add(n, f_value_best);

        return FT;
    }
}
```

Correction and Solution Generation functions

To ensure that our local search algorithms are working within the specified boundaries (shared search space) and specific domain of the benchmarks, we implement toro correction function, on the other hand we need to generate random solutions for each cycle of perturbation thus we will implement a function that generates arbitrary solutions.

Toro correction algorithm

When it comes to toro correction function, two approaches of correction should be considered as the following:

1- Common search space boundaries (one dimension)
to ensure that every benchmark variable tested on the same search space

2- Specific search space domains (two dimensions)
to ensure that every benchmark variable working in its specified domain range

Common search space boundaries

Toro common search space correction

```
public static double[] toro(double[] x, double[] bounds)
	{
		int c = x.length;
		double[] x_tor_cor = new double[c];
		for (int i = 0; i < c; i++){
				x_tor_cor[i] = (x[i] - bounds[0])/(bounds[1] - bounds[0]);
			if (x_tor_cor[i] > 1)
			{
				x_tor_cor[i] = x_tor_cor[i]-fix(x_tor_cor[i]);}
			else if (x_tor_cor[i] < 0)
			{x_tor_cor[i] = 1 - Math.abs(x_tor_cor[i] - fix(x_tor_cor[i]));
			}
		x_tor_cor[i] = x_tor_cor[i] * (bounds[1] - bounds[0]) + bounds[0];
		}
	return x_tor_cor;}
```

Specific search space domains

Toro specific search space correction

```
public static double[] toro(double[] x, double[][] bounds)
	{int o = x.length;
		double[] x_tor_cor = new double[o];

		for (int i = 0; i < o; i++)
		{x_tor_cor[i] = (x[i] - bounds[i][0]) / (bounds[i][1] bounds[i][0]);
			if (x_tor_cor[i] > 1)
			{x_tor_cor[i] = x_tor_cor[i] - fix(x_tor_cor[i]);
			}else if (x_tor_cor[i] < 0)
			{x_tor_cor[i] = 1 - Math.abs(x_tor_cor[i] - fix(x_tor_cor[i]));}
	x_tor_cor[i] = x_tor_cor[i] * (bounds[i][1] - bounds[i][0]) + bounds[i][0];}
		return x_tor_cor;}
```

arbitrary solution generation

This function used to create random initial guess for the given problem within its search space, the generated solution will be selected and optimized, there are 2 types of solution generation (common and specific), similar to the toro correction algorithm, the two types are as the following:

1- Random solution generation for common search space boundaries (one dimension)
To generate random solutions in the same search space same as the other optimizations search space

2- Random solution generation for specific search space of problem domains (two dimensions)
To generate random solution within the domain of the benchmark model

Random solution common search space boundaries

Generate random solution in common search space

```
public static double[] generateRandomSolution(double[] bounds, int n)
	{
		double[] z = new double[n];

		for (int i = 0; i < n; i++)

			z[i] = bounds[0] + (bounds[1] - bounds[0]) * RandUtils.random();
		return z;
	}
```

Random solution for specific search space domains

Generate random solution in specific domains

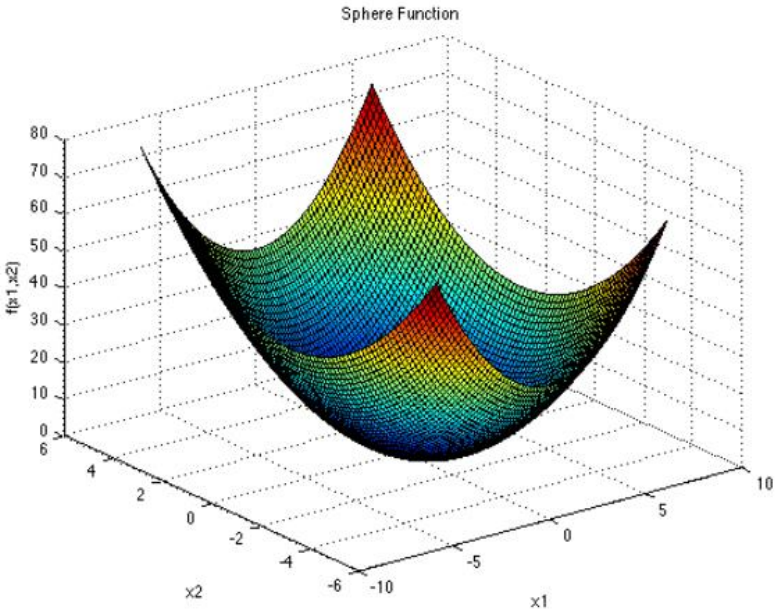
```
public static double[] generateRandomSolution(double[][] bounds, int k){
			double[] f = new double[k];

			for (int i =0; i < k; i++)

	f[i] = bounds[i][0]+(bounds[i][1]-bounds[i][0])*RandUtils.random();

		return f;
}
```

Benchmarks and Evaluation

De Jong benchmark

De Jong (sphere) is a uni-model and simplest model in terms of complexity among the benchmarks, the local minimum of the De Jong is its global minimum at the same time, thus it is the considered the easiest benchmark due to its low complexity.

SPHERE FUNCTION



$$f(\mathbf{x}) = \sum_{i=1}^{d} x_i^2$$

Description:

Dimensions: d

The Sphere function has d local minima except for the global one. It is continuous, convex and unimodal. The plot shows its two-dimensional form.

Input Domain:

The function is usually evaluated on the hypercube $x_i \in [-5.12, 5.12]$, for all i = 1, …, d.

Figure 1.De Jong benchmark. Source: https://www.sfu.ca/~ssurjano/spheref.html

Model evaluation

ISPO CMAES S

```
This experiment contains 3 optimisers and 1 problems with dimension value 5
(Runs,5 for each optimisation process, are spread over 4 available processors)

       ISPO                    CMAES                   W      S                       W
f1     1.175e-35 ± 7.750e-36   3.726e-311 ± 0.000e+00  -      0.000e+00 ± 0.000e+00   -


This experiment contains 3 optimisers and 1 problems with dimension value 10
(Runs,5 for each optimisation process, are spread over 4 available processors)

       ISPO                    CMAES                   W      S                       W
f1     7.884e-35 ± 3.675e-35   0.000e+00 ± 0.000e+00   -      0.000e+00 ± 0.000e+00   -


This experiment contains 3 optimisers and 1 problems with dimension value 15
(Runs,5 for each optimisation process, are spread over 4 available processors)

       ISPO                    CMAES                   W      S                       W
f1     4.326e-35 ± 3.321e-35   0.000e+00 ± 0.000e+00   -      0.000e+00 ± 0.000e+00   -


This experiment contains 3 optimisers and 1 problems with dimension value 20
(Runs,5 for each optimisation process, are spread over 4 available processors)

       ISPO                    CMAES                   W      S                       W
f1     8.105e-35 ± 4.926e-35   0.000e+00 ± 0.000e+00   -      0.000e+00 ± 0.000e+00   -


This experiment contains 3 optimisers and 1 problems with dimension value 25
(Runs,5 for each optimisation process, are spread over 4 available processors)

       ISPO                    CMAES                   W      S                       W
f1     1.268e-34 ± 5.988e-35   0.000e+00 ± 0.000e+00   -      0.000e+00 ± 0.000e+00   -
```

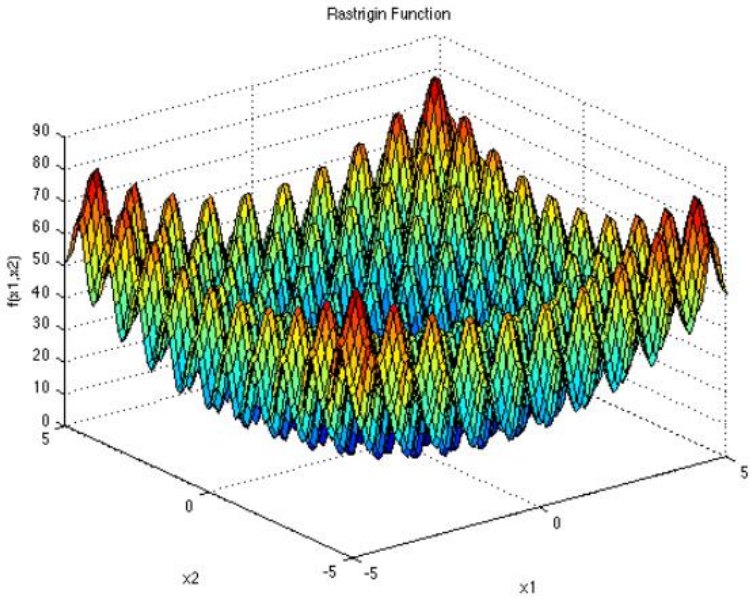
I executed the program for 5, 10, 15, 20 and 25 dimensions, it is obvious that the ISPO did not perform well even thought of the benchmark simplicity, meanwhile the S algorithm had the average mean fitness value of zero all the time and the CMAES also performance was satisfying even though sometimes failed to converge to the global minimum, we could clearly conclude that S algorithm is the best choice for the De Jong benchmark model comparing to ISPO and CMAES.

Since De joing is considered as a convex function, its local minimum is its global minimum at the same time.

Rastrigin benchmark

Rastrigin is high multi modal benchmark, as shown in the following figure, Rastrigin has very complex pattern with many local minimums, thus it has high probability that some simple local search optimization algorithms such as S to get stuck in one the local minimums of the Rastrigin.

RASTRIGIN FUNCTION



$$f(\mathbf{x}) = 10d + \sum_{i=1}^{d} \left[x_i^2 - 10\cos(2\pi x_i) \right]$$

Description:

Dimensions: d

The Rastrigin function has several local minima. It is highly multimodal, but locations of the minima are regularly distributed. It is shown in the plot above in its two-dimensional form.

Input Domain:

The function is usually evaluated on the hypercube $x_i \in [-5.12, 5.12]$, for all i = 1, …, d.

Figure 2. Rastrigin benchmark. Source: https://www.sfu.ca/~ssurjano/rastr.html

Model evaluation

ISPO CMAES S

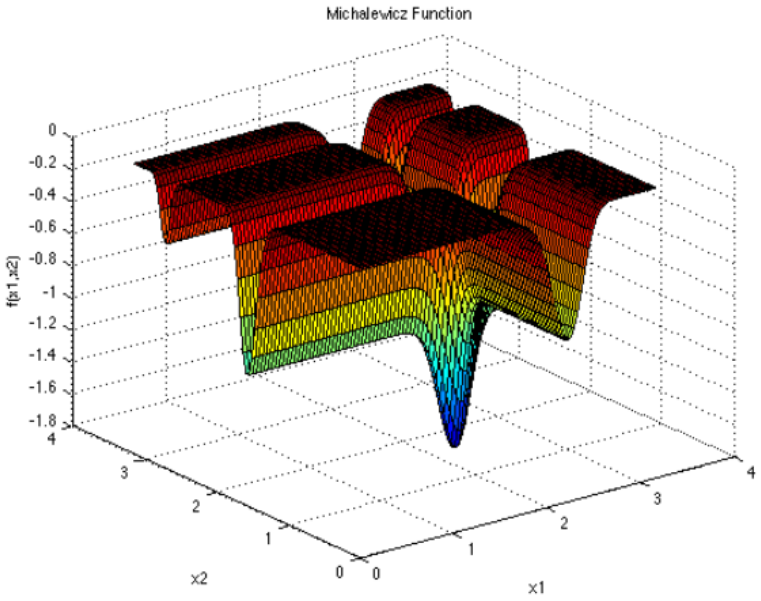
```
This experiment contains 3 optimisers and 1 problems with dimension value 5
(Runs,5 for each optimisation process, are spread over 4 available processors)

        ISPO                    CMAES                   W       S                       W
f1      4.457e+01 ± 1.634e+01   3.602e+01 ± 1.899e+01   =       2.388e+00 ± 1.350e+00   -


This experiment contains 3 optimisers and 1 problems with dimension value 10
(Runs,5 for each optimisation process, are spread over 4 available processors)

        ISPO                    CMAES                   W       S                       W
f1      6.109e+01 ± 1.795e+01   8.935e+01 ± 1.839e+01   =       4.179e+00 ± 7.446e-01   -


This experiment contains 3 optimisers and 1 problems with dimension value 15
(Runs,5 for each optimisation process, are spread over 4 available processors)

        ISPO                    CMAES                   W       S                       W
f1      1.192e+02 ± 3.387e+01   1.166e+02 ± 2.567e+01   =       5.771e+00 ± 1.160e+00   -


This experiment contains 3 optimisers and 1 problems with dimension value 20
(Runs,5 for each optimisation process, are spread over 4 available processors)

        ISPO                    CMAES                   W       S                       W
f1      1.662e+02 ± 1.976e+01   1.751e+02 ± 3.152e+01   =       1.015e+01 ± 7.446e-01   -


This experiment contains 3 optimisers and 1 problems with dimension value 25
(Runs,5 for each optimisation process, are spread over 4 available processors)

        ISPO                    CMAES                   W       S                       W
f1      1.817e+02 ± 6.411e+01   2.012e+02 ± 1.615e+01   =       1.075e+01 ± 1.929e+00   -
```

The program executed for 5, 10, 15, 20, 25 dimensions, non of the 3 types of the optimizers reached the global minimum, this is due the high multimodality of the benchmark, as shown in the results above non of the optimizers have mean average fitness value of zero, further more the fitness value could be worse than ever as the dimension increases and start getting far away from the global minimum which will result in unsatisfactory results and trapped in one of the local minimums, however comparing the 3 algorithms ISPO, CMAES and S, the S algorithm succeeded to achieve better results in terms of pointing to the minimal basin attraction of the Rastrigin benchmark model.

Michalewicz benchmark

Michalewicz is low multi modal benchmark, it has a a unique pattern that includes many valleys mixed with smooth surface.

MICHALEWICZ FUNCTION



$$f(\mathbf{x}) = -\sum_{i=1}^{d} \sin(x_i)\sin^{2m}\left(\frac{ix_i^2}{\pi}\right)$$

Description:

Dimensions: d

The Michalewicz function has d! local minima, and it is multimodal. The parameter m defines the steepness of they valleys and ridges; a larger m leads to a more difficult search. The recommended value of m is m = 10. The function's two-dimensional form is shown in the plot above.

Input Domain:

The function is usually evaluated on the hypercube $x_i \in [0, \pi]$, for all i = 1, …, d.

Figure 3. Michalewicz benchmark Source: https://www.sfu.ca/~ssurjano/michal.html

Model evaluation

ISPO CMAES S

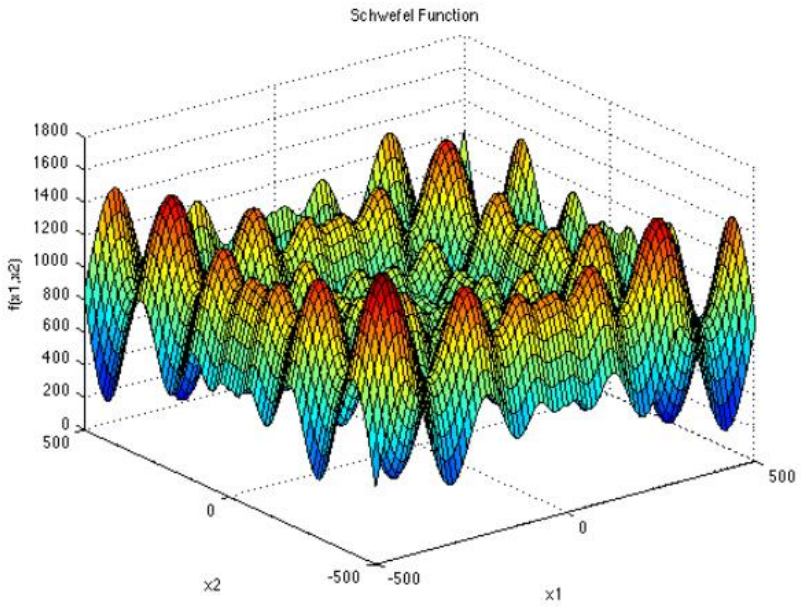
```
This experiment contains 3 optimisers and 1 problems with dimension value 5
(Runs,5 for each optimisation process, are spread over 4 available processors)

        ISPO                    CMAES                   W       S                       W
f1      -4.191e+00 ± 5.730e-01  -3.064e+00 ± 6.586e-01  +       -4.277e+00 ± 2.845e-01  =


This experiment contains 3 optimisers and 1 problems with dimension value 10
(Runs,5 for each optimisation process, are spread over 4 available processors)

        ISPO                    CMAES                   W       S                       W
f1      -8.920e+00 ± 2.358e-01  -7.106e+00 ± 6.514e-01  +       -7.881e+00 ± 6.742e-01  +


This experiment contains 3 optimisers and 1 problems with dimension value 15
(Runs,5 for each optimisation process, are spread over 4 available processors)

        ISPO                    CMAES                   W       S                       W
f1      -1.390e+01 ± 4.569e-01  -9.713e+00 ± 8.984e-01  +       -1.199e+01 ± 4.822e-01  +


This experiment contains 3 optimisers and 1 problems with dimension value 20
(Runs,5 for each optimisation process, are spread over 4 available processors)

        ISPO                    CMAES                   W       S                       W
f1      -1.859e+01 ± 3.927e-01  -1.312e+01 ± 2.555e+00  +       -1.516e+01 ± 9.812e-01  +


This experiment contains 3 optimisers and 1 problems with dimension value 25
(Runs,5 for each optimisation process, are spread over 4 available processors)

        ISPO                    CMAES                   W       S                       W
f1      -2.394e+01 ± 3.296e-01  -1.503e+01 ± 2.708e+00  +       -2.010e+01 ± 6.286e-01  +
```

In this benchmark, it is clear that ISPO and CMAES has better performance than S, since ISPO and CAMES have fitness values close to the global optimum, thus in this benchmark S algorithm is the least efficient among the other optimizers in the given benchmark.

Schwefel

Schwefel is high multi-modal benchmark with peaks in its landscape, these peaks can be encountered during the search phase and cause the model to perform poorly.

SCHWEFEL FUNCTION



$$f(\mathbf{x}) = 418.9829d - \sum_{i=1}^{d} x_i \sin(\sqrt{|x_i|})$$

Description:

Dimensions: d

The Schwefel function is complex, with many local minima. The plot shows the two-dimensional form of the function.

Input Domain:

The function is usually evaluated on the hypercube $x_i \in [-500, 500]$, for all i = 1, …, d.

Figure 4. Scwefel benchmark Source: http://www.sfu.ca/~ssurjano/schwef.html

Model evaluation

ISPO CMAES S

```
This experiment contains 3 optimisers and 1 problems with dimension value 5
(Runs,5 for each optimisation process, are spread over 4 available processors)

        ISPO                    CMAES                   W       S                       W
f1      7.508e+02 ± 2.117e+02   9.870e+02 ± 1.514e+02   +       2.132e+02 ± 4.738e+01   -


This experiment contains 3 optimisers and 1 problems with dimension value 10
(Runs,5 for each optimisation process, are spread over 4 available processors)

        ISPO                    CMAES                   W       S                       W
f1      1.939e+03 ± 3.810e+02   2.109e+03 ± 1.431e+02   =       5.685e+02 ± 2.171e+02   -


This experiment contains 3 optimisers and 1 problems with dimension value 15
(Runs,5 for each optimisation process, are spread over 4 available processors)

        ISPO                    CMAES                   W       S                       W
f1      3.140e+03 ± 4.475e+02   3.382e+03 ± 7.114e+02   =       7.343e+02 ± 2.638e+02   -


This experiment contains 3 optimisers and 1 problems with dimension value 20
(Runs,5 for each optimisation process, are spread over 4 available processors)

        ISPO                    CMAES                   W       S                       W
f1      4.139e+03 ± 8.258e+02   3.966e+03 ± 4.890e+02   =       1.208e+03 ± 2.742e+02   -


This experiment contains 3 optimisers and 1 problems with dimension value 25
(Runs,5 for each optimisation process, are spread over 4 available processors)

        ISPO                    CMAES                   W       S                       W
f1      5.487e+03 ± 5.032e+02   5.165e+03 ± 3.310e+02   =       1.161e+03 ± 2.529e+02   -
```

In Schwefel benchmark, as the previous benchmarks we ran the problem with 5, 10, 15, 20 and 25 dimensions to test the optimizers, the performance of S algorithm is better than ISPO and CMAES in this benchmark, since S algorithm is showing sensitivity toward the optimal value in the landscape, furthermore, as shown above in the test performance as the dimension increases, the fitness value becoming worse than the previous dimension (getting far away from the optimum point), this happens due to the high multi-modality of the landscape and thus the optimizers gets stuck in one of the local minimum points or (basin of attraction)

Conclusion

In conclusion, single solution short distance optimizers achieve the optimal results in the case of uni-modal benchmark, meanwhile the optimizers showed satisfying results in the benchmarks of low/high modals which was close to the optimal solution in some cases, specifically some optimizers performed better at some benchmarks then the others and the vice versa, as in the case of the De jong benchmark, the S algorithm performed more effective (the average mean of the fitness function was zero!) than the ISPO, while in the case Michalewicz (high modal with complex landscape) the ISPO algorithm performed better than the S algorithm, thus we can conclude that an optimization algorithms can achieve different fitness values on different benchmarks, thus we can conclude that S algorithm more efficient with benchmarks with uni/low modal patterns, however its efficiency decreases in the case of medium/high modals

References

- Auger, A. and Teytaud, O. (2010). Continuous lunches are free plus the design of optimal optimization algorithms. Algorithmica, 57(1):21–146.

- Burke, E. K., Hyde, M., Kendall, G., Ochoa, G., Ozcan, E., and Woodward, J. (2010). Classification of hyper-heuristic approaches. In Handbook of Meta-Heuristics, pages 449–468. Springer

- Caraffini, F. (2014). Novel Memetic Computing Structures for Continuous Optimisation. PhD thesis, De Montfort University, Leicetser, United Kingdom. https://www.dora.dmu.ac.uk/xmlui/handle/2086/10629.

- Caraffini, F., Iacca, G., Neri, F., and Mininno, E. (2012a). The importance of being structured: A comparative study on multi stage memetic approaches. In Computational Intelligence (UKCI), 2012 12th UK Workshop on, pages 1–8.

- Caraffini, F., Neri, F., Iacca, G., and Mol, A. (2012b). Parallel memetic structures. Information Sciences, 227(0):60–82

- Eiben, A. E. and Smith, J. E. (2003). Introduction to Evolutionary Computation. Springer-verlag, Berlin

- Hooke, R. and Jeeves, T. A. (1961). Direct search solution of numerical and statistical problems. Journal of the ACM, 8:212–229.

- Özcan, E., Bilgin, B., and Korkmaz, E. E. (2008). A comprehensive analysis of hyper-heuristics. Intelligent Data Analysis, 12(1):3–23.

- Hansen, N. and Ostermeier, A. (1996). Adapting arbitrary normal mutation distributions in evolution strategies: The covariance matrix adaptation. In Proceedings of the IEEE International Conference on Evolutionary Computation, pages 312–317.

- Wilcoxon, F. (1945). Individual comparisons by ranking methods. Biometrics Bulletin, 1(6):80–83.

- Zhen, J., Jiarui, Z., Huilian, L., and Qing-Hua, W. (2010). A novel intelligent single particle optimizer. Chinese Journal of computers, 33(3):556–561.

2022

Computational Intelligence Optimization

IBRAHIM AHMETHAN
P2699379
TASK 3

DE MONTFORT UNIVERSITY | The Gateway

Contents

List of figures

1 Introduction

In this study, a Memetic algorithm (Meme) algorithm is implemented, Meme algorithm is one of the most rapidly expanding fields of research in evolutionary computation, furthermore Meme algorithm is a mixture of local search and evolutionary algorithm, the main purpose of Meme algorithm in this task is to abolish the possibility of undesired optimization drawbacks such as stagnation and premature during the search process in the given problem benchmark, thus the integration of differential evolution DE which is a population based solution approach with the short distance exploration S which is a single based solution algorithm in order to compare these algorithms by utilizing the provided benchmark (CEC 2014) in the SOS file where the algorithms are tested with different modality and dimensionality levels to examine mentioned algorithms under different circumstances[1].

2 Problem description and motivation

Simply said, the problem may be defined as searching for the vector values within the given multi modal domain $x \in D$ to obtain the best possible fitness function $f(x)$ that corresponds to solution near or equal to the optimal solution (global minimum), mathematically the problem can be interpreted as mapping off the objective function as the following f: D → M ∈ |R [1 lect slid 1].

To solve the problem, I implemented the Memetic algorithm which is mixture of a population based algorithm which is jitter – differential evolution(jDE) with a single based solution algorithm S, each one the search algorithms has its own characteristic, mainly the core task of population based algorithm is to search for the optimal solution in the entire given search domain, more explicitly exploration of the entire search space, meanwhile the single solution algorithm is considered as a local search, its task to search in the neighbors of the given point and converge to the optimal solution around that neighborhood, mainly the local search is the process of exploitation of the search space[2].

A novel design is introduced alongside the S and DE algorithms, it is self adaptive jDES, which is the evolved variant of the normal DE beside the S local search algorithm, self adaptive DE is talented in adapting the control parameters during the exploration of search space.

The proposed novel optimizer is compared statistically to the selected S and DE optimizers on the CEC 2014 test bed with 10-D and 50-D dimensions and series of unimodal, multimodal and hybrid composition test function to compare the proposed novel optimizer (self adaptive DE) against the population based (DE) and single solution based (S) optimizers.

3 Algorithms design

3.1 Memetic algorithm

The main point of designing the Meme algorithm is focuses on the balance between the exploration of the search domain and exploitation of possible optima, thus the design of the Meme algorithm by combining the evolutionary algorithm (EA) and local search operator (LS) aims to balance between exploration and exploitation to avoid some undesired situations such as premature convergence and stagnation during the search process that might lead to undesired results of the fitness function *f(x)*, more explicitly the evolutionary algorithm EA (population based solution) offers diversified population distributed in the entire search space in order to avoid early convergence and to quickly escape local minima, thus a degree of exploration is maintained, meanwhile the local research (single based solution) is utilized to exploit the best possible basin of attraction for the purpose of converging and avoiding stagnation, thus EA and LS working together to iteratively to eliminate the stagnation and premature convergence[3].

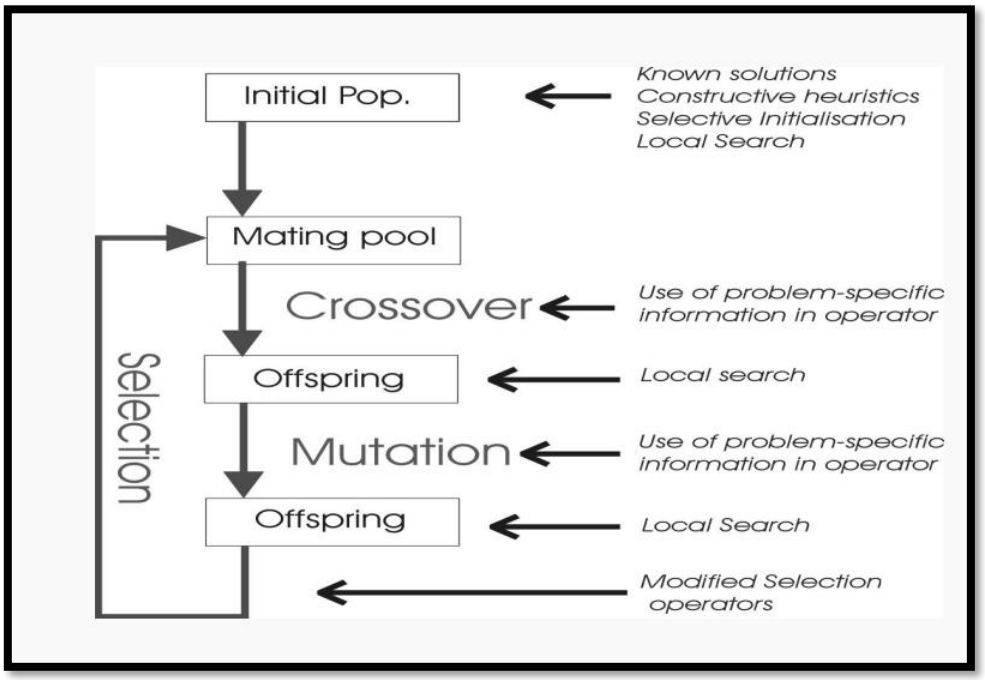


Figure 1. general scheme of memetic algorithm

Many design parameters should be cautiously tuned and constructed such as crossover, mutation and parent selection mechanism to build the Memetic algorithms. On some issues, MA proved to be more robust that EA in terms of time and accuracy

3.2 Global search - differential evolution DE algorithm

DE is a stochastic population-based (derivative free) metaheuristic search technique that iteratively improves a potential solution based on an evolutionary process to optimize a problem, DE utilizes a population of prospective approach to expand the search space from local into global search.

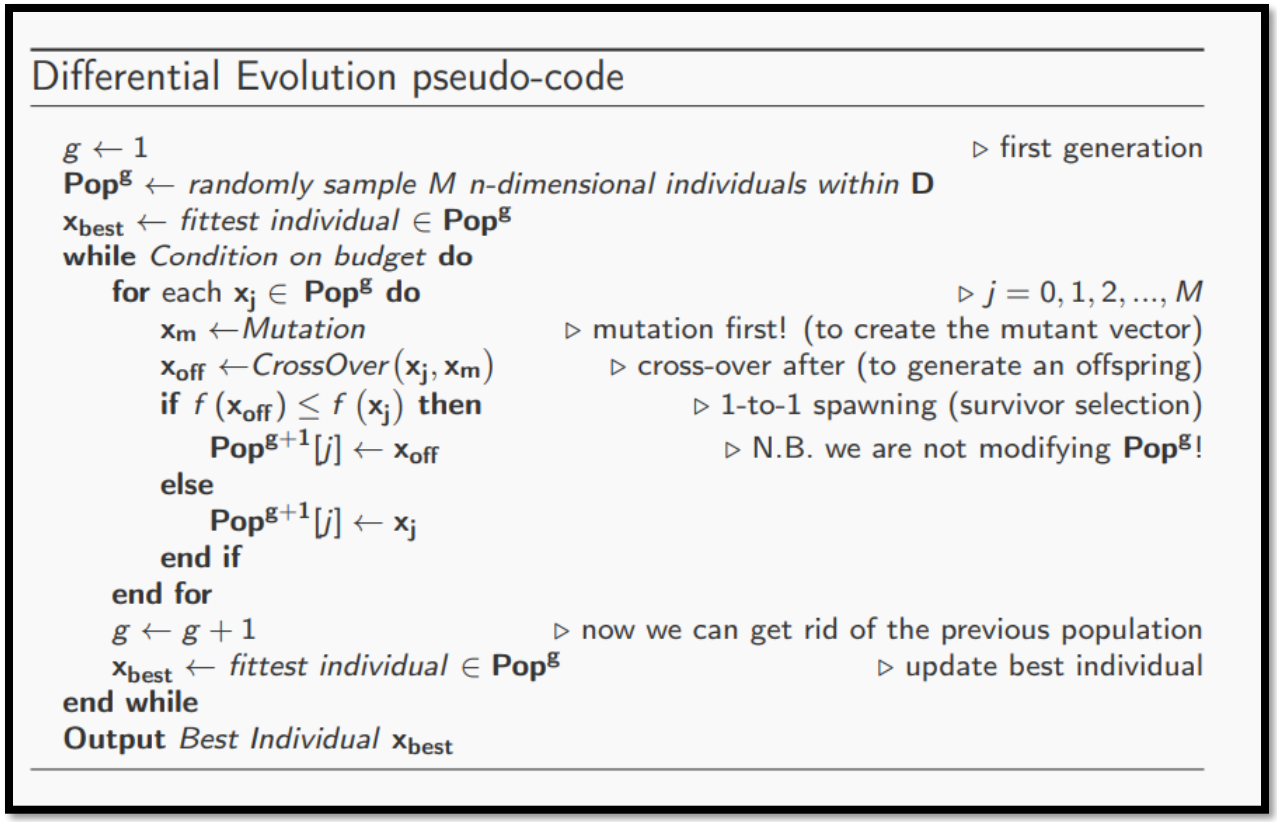


Figure 2. Differential evolution general scheme

At the start of the optimization procedure, the initial population scattered in the decision space, moreover on average, the difference vector is considerable due to the exploration nature of the algorithm, as shown in figure 3 exploratory phase at the beginning of the search process, later on the solutions should be targeted in a specific area of the choice space towards the end of the optimization procedure,

hence the exploratory radius shrinks gradually toward that area, which is considered as exploitation of the search space as shown in figure 4.

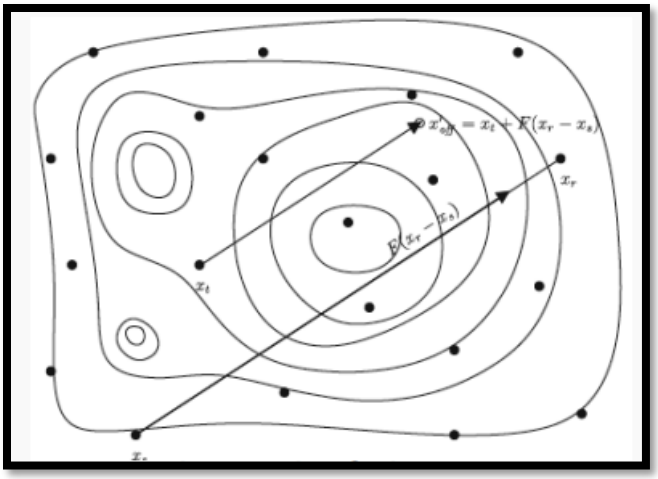


Figure 4. exploration of search space

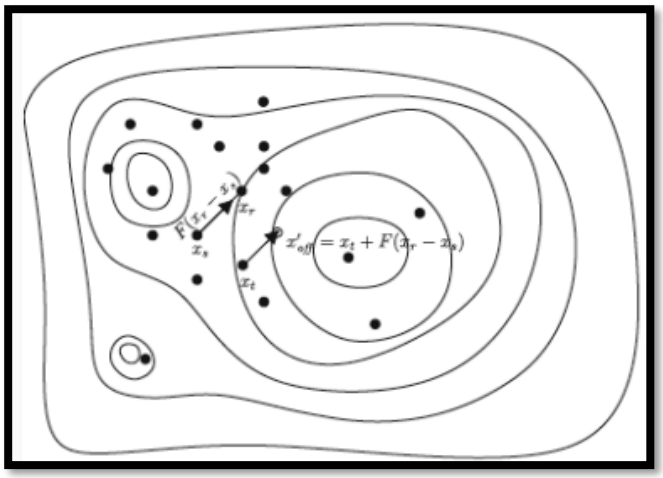


Figure 3. exploitation of search space

Furthermore, some very important evolutionary mechanisms are implemented such is mutation, where in the mutation crossed over with the parent solutions and produces a child (new individuals). After the cross over the final stage is survivor selection where the generation of the entire offspring is done for each individual, some of the individual will be replaced with other for the next generation due to their more efficient fitness function. Generally having the cross over and mutation mechanism is recommended rather than working with one of them![4].

The performance of the DE depends on the tuning parameters of scale factor (F), cross over rate (CR), and population size (n), here emerges the novel design of self adaptive differential evolution, which is talented in adapting the control parameters

Differential Evolution pseudo-code

$g \leftarrow 1$ ▷ first generation
$\mathbf{Pop}^{g} \leftarrow$ *randomly sample M n-dimensional individuals within* **D**
$\mathbf{x}_{best} \leftarrow$ *fittest individual* $\in \mathbf{Pop}^{g}$
while *Condition on budget* **do**
  for each $\mathbf{x}_j \in \mathbf{Pop}^{g}$ **do** ▷ $j = 0, 1, 2, ..., M$
    $\mathbf{x}_m \leftarrow$ *Mutation* ▷ mutation first! (to create the mutant vector)
    $\mathbf{x}_{off} \leftarrow CrossOver(\mathbf{x}_j, \mathbf{x}_m)$ ▷ cross-over after (to generate an offspring)
    if $f(\mathbf{x}_{off}) \leq f(\mathbf{x}_j)$ **then** ▷ 1-to-1 spawning (survivor selection)
      $\mathbf{Pop}^{g+1}[j] \leftarrow \mathbf{x}_{off}$ ▷ N.B. we are not modifying $\mathbf{Pop}^{g}$!
    else
      $\mathbf{Pop}^{g+1}[j] \leftarrow \mathbf{x}_j$
    end if
  end for
  $g \leftarrow g + 1$ ▷ now we can get rid of the previous population
  $\mathbf{x}_{best} \leftarrow$ *fittest individual* $\in \mathbf{Pop}^{g}$ ▷ update best individual
end while
Output *Best Individual* $\mathbf{x}_{best}$

Figure 6. Differential evolution pseudo design

procedure Binomial-XO($\mathbf{x}_1, \mathbf{x}_2$)
  Index $\leftarrow$ rand int $[1, n]$
  for $i = 1 : n$ **do**
    if $\mathcal{U}(0, 1) \leq CR \;||\; i == Index$ **then**
      $\mathbf{x}_{off}[i] \leftarrow \mathbf{x}_1[i]$
    else
      $\mathbf{x}_{off}[i] \leftarrow \mathbf{x}_2[i]$
    end if
  end for
  Output $\mathbf{x}_{off}$
end procedure

Figure 5. Cross over pseudo design

of the differential evolution on the fly.

3.3 Novel design of jDES

With multimodal robustness, DE is one of the most adaptive and flexible population-based search algorithms applicable, thus the adaptive jitter - DE selected to be the evolutionary part of the Memetic algorithm (MA) alongside the local search S algorithm, hence the novel design is the mixture of self adaptive differential evolution jDE with the single solution optimizer S, where we have control parameters such as F and CR will be tuned during the search process to reach the best basin of attraction, the new novel design named (jDES), it is a , offspring are obtained comparing to the default DE, after obtaining better offspring (individuals) here emerges the importance of the local short distance optimizer S to search for the further mutation by utilizing the perturbation logic of S algorithm, consequently the search space will be narrowed down toward more potential optimum basin of attraction, moreover this enhancement in the performance will lead to reduce the stagnation risk[5].

4 Software implementation

In this section the implementation of the DE and its novel hybrid design jDES is

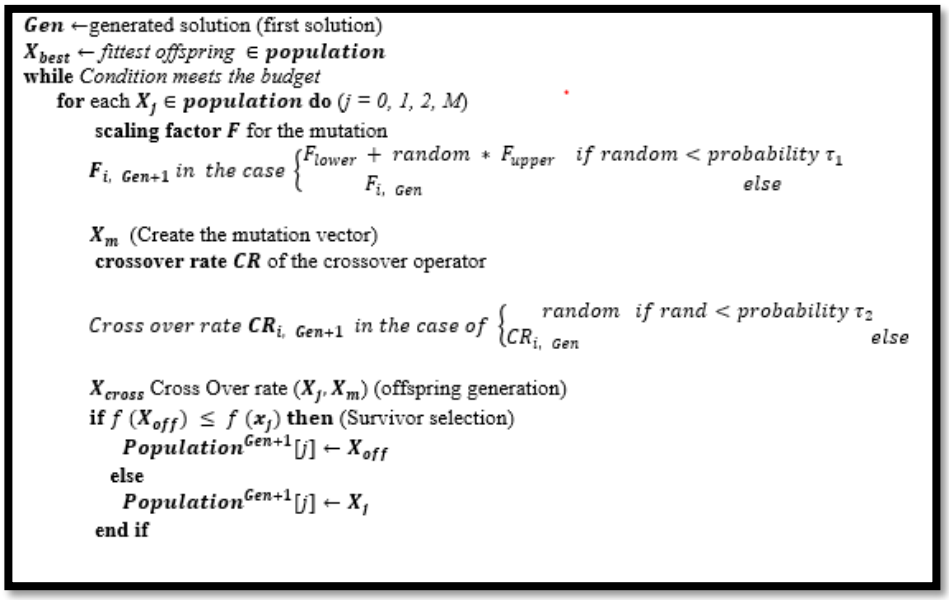


Figure 7. Pseudocode of self adaptive DE algorithm

attached. The S algorithm as already provided in the Task 2.

4.1 DE

```
5
    package algorithms;
    import utils.RunAndStore;
    import utils.algorithms.Misc;
    import interfaces.Algorithm;
    import interfaces.Problem;
    import utils.random.RandUtils;

    public class DE extends Algorithm {
        public DE() {
        }
        public RunAndStore.FTrend execute(Problem prob, int maximum_test)
    throws Exception {
            RunAndStore.FTrend FT = new RunAndStore.FTrend();
            double[] best = new double[dimension];
            double[][] population = new double[population_size][dimension];
            double[] fitness = new double[population_size];
            int dimension = prob.getDimension();
            double[][] boundaries = prob.getBounds();
            double final_Best = Double.NaN;

            int z = 0;
            int population_size = this.getParameter("p0").intValue();
            double scale_factor = this.getParameter("p1");
            double crossover_rate = this.getParameter("p2");

            int fitness_population;
            while (int y = 0; y < population_size; ++y)
            {
                double[] initial_solution =
    Misc.generateRandomSolution(boundaries, dimension);
                for(fitness_population = 0; fitness_population < dimension;
    ++fitness_population)
                {
                    population[y][fitness_population] =
    initial_solution[fitness_population];
                }

                fitness[y] = prob.f(population[y]);
                ++z;
                if (y == 0 || fitness[y] < final_Best) {
                    final_Best = fitness[y];
                    for(fitness_population = 0; fitness_population <
    dimension; ++fitness_population) {
                        best[fitness_population] =
    population[y][fitness_population];
                    }
                }

                if (y == 0 || z % 100 == 0) {
                    FT.add(z, final_Best);
```

```
            }
        }

        while(z < maximum_test) {
            double[][] new_generation = new
double[population_size][dimension];
            for(int j = 0; j < population_size && z < maximum_test;
++j) {
                fitness_population = 0;
                double recent_fitnss = Double.NaN;
                double fitness_cross_over = Double.NaN;
                double[] individual_population = new double[dimension];
                double[] individual_mutant = new double[dimension];
                double[] individual_cross_over = new double[dimension];


                int s;
                for(s = 0; s < dimension; ++s) {
                    individual_population[s] = population[j][s];
                }

                recent_fitnss = fitness[j];
                individual_mutant = original_mutat(population,
scale_factor, dimension);
                individual_cross_over =
Misc.toro(individual_cross_over, boundaries);
                fitness_cross_over = prob.f(individual_cross_over);
                individual_cross_over =
binomial_crossover(individual_population, crossover_rate,
individual_mutant, dimension);

                ++z;
                if (fitness_cross_over < recent_fitnss) {
                    for(s = 0; s < dimension; ++s) {
                        new_generation[j][s] =
individual_cross_over[s];
                    }
                    fitness[j] = fitness_cross_over;
                    if (fitness_cross_over < final_Best) {
                        final_Best = fitness_cross_over;
                        if (z % 100 == 0 && fitness_population != z) {
                            FT.add(z, fitness_cross_over);
                            fitness_population = z;
                        }
                        for(s = 0; s < dimension; ++s) {
                            best[s] = individual_cross_over[s];
                        }
                    }
                } else {
                    for(s = 0; s < dimension; ++s)
                    {
                        new_generation[j][s] =
individual_population[s];
                    }
                    fitness[j] = recent_fitnss;
                }
```

```
                if (z % 100 == 0 && fitness_population != z) {
                    FT.add(z, final_Best);
                }
            }
            population = new_generation;
        }

        this.finalBest = best;
        FT.add(z, final_Best);
        return FT;
    }

    public static double[] original_mutat(double[][] pop, double
scale_factor, int dimensionality) {
        int[] random_permutation = new int[population_size];
        int population_size = pop.length;
        int dimesnsion = dimensionality;
        int randomPointOne;
        for(randomPointOne = 0; randomPointOne < population_size;
random_permutation[randomPointOne] = randomPointOne++) {
        }
        double[] mutanted_idndividual = new double[dimensionality];
        random_at_one = random_permutation[0];
        int random_at_two = random_permutation[1];
        int random_at_three = random_permutation[2];
        random_permutation =
RandUtils.randomPermutation(random_permutation);
        for (int i = 0; i < dimesnsion; ++i)
        {
            mutanted_idndividual[i] = pop[randomPointOne][i] +
scale_factor * (pop[random_at_two][i] - pop[random_at_three][i]);
        }
        return mutanted_idndividual;
    }
    public static double[] binomial_crossover(double[]
population_individual, double crossover_rate, double[]
mutant_individual  , int dimensionality) {
        int dimension = dimensionality;
        int random_index = RandUtils.randomInteger(dimensionality - 1);
        double[] offspring = new double[dimensionality];
        for
        (int i = 0; i < dimension; ++i) {
            if(!(RandUtils.random() < crossover_rate) && i !=
random_index) {
                offspring[i] = population_individual[i];
            } else {
                offspring[i] = mutant_individual[i];
            }
        }
        return offspring;
    }
```

5.1 jDES

```
6  package algorithms;
```

```
import utils.RunAndStore;
import utils.random.RandUtils;
import interfaces.Algorithm;
import utils.algorithms.Misc;
import interfaces.Problem;


public class jDES extends Algorithm {
    public jDES() {
    }

    public RunAndStore.FTrend execute(Problem problem, int
maximum_evaluation) throws Exception {
        RunAndStore.FTrend FT = new RunAndStore.FTrend();
        int dimension = problem.getDimension();
        double[][] boundaries = problem.getBounds();
        double[] best_solution = new double[dimension];
        double final_Best = Double.NaN;
        int z = 0;
        int size_of_population = this.getParameter("p0").intValue();
        double lower_bound_scaling = this.getParameter("p3");
        double upper_bound_scaling = this.getParameter("p4");
        double first_t = this.getParameter("p5");
        double second_t = this.getParameter("p6");
        double[] scale_factor = new double[size_of_population];
        double[] croes_over_rate = new double[size_of_population];
        double[][] population = new
double[size_of_population][dimension];
        double[] fitness_solution = new double[size_of_population];
        double alpha_value = this.getParameter("p7");
        double p_iteration =
(double)this.getParameter("p8").intValue();
        double[] x_search = best_solution;
        double[] exp_raduis = new double[dimension];

        int populated_fitness;
        for(int i = 0; i < size_of_population; ++i) {
            double[] initialSolution =
Misc.generateRandomSolution(boundaries, dimension);

            for(populated_fitness = 0; populated_fitness < dimension;
++populated_fitness) {
                population[i][populated_fitness] =
initialSolution[populated_fitness];
            }

            fitness_solution[i] = problem.f(population[i]);
            ++z;
            if (i == 0 || fitness_solution[i] < final_Best) {
                final_Best = fitness_solution[i];

                for(populated_fitness = 0; populated_fitness <
dimension; ++populated_fitness) {
                    best_solution[populated_fitness] =
population[i][populated_fitness];
                }
            }
```

```
            if (i == 0 || z % 100 == 0) {
                FT.add(z, final_Best);
            }
        }

        while(z < maximum_evaluation) {
            double[][] new_generation = new
double[size_of_population][dimension];

            for(int j = 0; j < size_of_population && z <
maximum_evaluation; ++j) {
                populated_fitness = 0;
                double[] mutated_individual = new double[dimension];
                double[] mutated_cross_over = new double[dimension];
                double[] populated_individual = new double[dimension];
                double recent_fitness = Double.NaN;
                double fitness_cros_over = Double.NaN;

                int w;
                for(w = 0; w < dimension; ++w) {
                    populated_individual[w] = population[j][w];
                }

                recent_fitness = fitness_solution[j];
                if (RandUtils.random() < tauOne) {
                    scale_factor[j] = scalingFactorLowerBound +
RandUtils.random() * upper_bound_scaling;
                }

                mutated_individual = originalMutation(population,
scale_factor[j], dimension);
                if (RandUtils.random() < second_t) {
                    croes_over_rate[j] = RandUtils.random();
                }

                mutated_cross_over =
binomialCrossover(populated_individual, mutated_individual,
croes_over_rate[j], dimension);
                mutated_cross_over = Misc.toro(mutated_cross_over,
boundaries);
                fitness_cros_over = problem.f(mutated_cross_over);
                ++z;
                if (fitness_cros_over < recent_fitness) {
                    for(w = 0; w < dimension; ++w) {
                        new_generation[j][w] = mutated_cross_over[w];
                    }

                    fitness_solution[j] = fitness_cros_over;
                    if (fitness_cros_over < final_Best) {
                        final_Best = fitness_cros_over;
                        if (z % 100 == 0 && populated_fitness != z) {
                            FT.add(z, fitness_cros_over);
                            populated_fitness = z;
                        }

                        for(w = 0; w < dimension; ++w) {
```

```
                            best_solution[w] = mutated_cross_over[w];
                        }

                        for(w = 0; w < dimension; ++w) {
                            exp_raduis[w] = alpha_value *
(boundaries[w][1] - boundaries[w][0]);
                        }

                        for(w = 0; (double) w < p_iteration && z <
maximum_evaluation; ++w) {
                            boolean improved = false;

                            int i;
                            for(i = 0; i < dimension && z <
maximum_evaluation; ++i) {
                                x_search[i] = best_solution[i] -
exp_raduis[i];
                                x_search = Misc.toro(x_search,
boundaries);
                                double short_f = problem.f(x_search);
                                ++z;
                                if (short_f <= final_Best) {
                                    best_solution[i] = x_search[i];
                                    final_Best = short_f;
                                    if (z % 100 == 0 &&
populated_fitness != z) {
                                        FT.add(z, short_f);
                                        populated_fitness = z;
                                    }

                                    improved = true;
                                } else if (z < maximum_evaluation) {
                                    if (z % 100 == 0 &&
populated_fitness != z) {
                                        FT.add(z, final_Best);
                                        populated_fitness = z;
                                    }

                                    x_search[i] = best_solution[i] +
exp_raduis[i] / 2.0;
                                    x_search = Misc.toro(x_search,
boundaries);
                                    short_f = problem.f(x_search);
                                    ++z;
                                    if (short_f <= final_Best) {
                                        best_solution[i] = x_search[i];
                                        final_Best = short_f;
                                        improved = true;
                                    } else {
                                        x_search[i] = best_solution[i];
                                    }

                                    if (z % 100 == 0 &&
populated_fitness != z) {
                                        FT.add(z, final_Best);
                                        populated_fitness = z;
                                    }
```

```
                                    }

                                    if (z % 100 == 0 && populated_fitness
!= z) {
                                        FT.add(z, final_Best);
                                        populated_fitness = z;
                                    }
                                }

                                if (!improved) {
                                    for(i = 0; i < exp_raduis.length; ++i)
{
                                        exp_raduis[i] /= 2.0;
                                    }
                                } else {
                                    for(i = 0; i < dimension; ++i) {
                                        new_generation[j][i] =
best_solution[i];
                                    }

                                    fitness_solution[j] = final_Best;
                                }
                            }
                        }
                    } else {
                        for(w = 0; w < dimension; ++w) {
                            new_generation[j][w] = populated_individual[w];
                        }

                        fitness_solution[j] = recent_fitness;
                    }

                    if (z % 100 == 0 && populated_fitness != z) {
                        FT.add(z, final_Best);
                    }
                }

                population = new_generation;
            }

            this.finalBest = best_solution;
            FT.add(z, final_Best);
            return FT;
        }

        public static double[] originalMutation(double[][] population,
double scale_factor, int dimension) {
            int Dimension = dimension;
            int pop_size = population.length;
            int[] permutation_random = new int[pop_size];

            int first_random;
            for(first_random = 0; first_random < pop_size;
permutation_random[first_random] = first_random++) {
            }

            permutation_random =
```

```
RandUtils.randomPermutation(permutation_random);
        first_random = permutation_random[0];
        int second_random = permutation_random[1];
        int thired_random = permutation_random[2];
        double[] mutanted_individual = new double[dimension];

        for(int i = 0; i < Dimension; ++i) {
            mutanted_individual[i] = population[first_random][i] +
scale_factor * (population[second_random][i] -
population[thired_random][i]);
        }
        return mutanted_individual;
    }
    public static double[] binomialCrossover(double[]
individual_population, double[] mutanted_individual, double rate_of_CR,
int dimensionality) {
        int Dimension = dimensionality;
        int rand_index = RandUtils.randomInteger(dimensionality - 1);
        double[] offspring = new double[dimensionality];
        for(int i = 0; i < Dimension; ++i) {
            if (!(RandUtils.random() < rate_of_CR) && i != rand_index)
{
                offspring[i] = individual_population[i];
            } else {
                offspring[i] = mutanted_individual[i];
            }
        }
        return offspring;
    }
}
```

5 CEC 2014 benchmark

In this task 3, We had the opportunity to run, test and compare the proposed optimization algorithms on the CEC 2014 benchmark, in this benchmark the Memetic, DE and S algorithms will run on different test modals included the CEC 2014 benchmark.
The modality and dimension play important role to measure the fitness of a specific optimization algorithm, thus with the range of 30 run problems that contains different modal function as the following[6]

1- Unimodal benchmark. Range [1-3]
2- Simple multimodal benchmark. Range [4-16]
3- Hybrid benchmark. Range [17-22]
4- Compensation benchmark. Range [23-30]

All three algorithms will be tested along the range of the CEC 2014 benchmark problems with different modalities as mentioned above.

6 Results and conclusions

In this section the test is done with 10-dimension and 50-dimension with 4 different modalities ranging from 1-30 problems.

- Comparison results of 10-D

	S	jDES	W	DE	W
f1	1.463e+08 ± 6.772e+08	2.725e+03 ± 2.627e+03	-	3.580e+02 ± 1.004e+02	-
f2	2.328e+09 ± 7.050e+09	2.006e+02 ± 1.181e+00	-	2.000e+02 ± 2.581e-04	-
f3	1.124e+04 ± 7.385e+03	3.000e+02 ± 5.052e-02	-	3.000e+02 ± 6.161e-08	-
f4	4.236e+02 ± 1.582e+01	4.047e+02 ± 1.021e+01	-	4.116e+02 ± 1.639e+01	-
f5	5.200e+02 ± 4.039e-03	5.181e+02 ± 5.641e+00	=	5.204e+02 ± 6.795e-02	+
f6	6.061e+02 ± 2.516e+00	6.001e+02 ± 1.632e-01	-	6.000e+02 ± 1.559e-02	-
f7	7.001e+02 ± 1.085e-01	7.001e+02 ± 4.041e-02	=	7.004e+02 ± 7.760e-02	+
f8	8.044e+02 ± 2.970e+00	8.000e+02 ± 9.047e-14	-	8.183e+02 ± 3.127e+00	+
f9	9.357e+02 ± 1.579e+01	9.111e+02 ± 4.703e+00	-	9.289e+02 ± 3.636e+00	=
f10	1.263e+03 ± 1.355e+02	1.015e+03 ± 1.112e+01	-	1.848e+03 ± 1.328e+02	+
f11	2.255e+03 ± 5.991e+02	1.583e+03 ± 1.991e+02	-	2.418e+03 ± 1.239e+02	+
f12	1.200e+03 ± 2.713e-01	1.200e+03 ± 1.596e-01	=	1.201e+03 ± 1.930e-01	+
f13	1.300e+03 ± 1.930e-01	1.300e+03 ± 4.620e-02	-	1.300e+03 ± 3.581e-02	-

f14	1.405e+03 ± 1.730e+01	1.400e+03 ± 5.653e-02	-	1.400e+03 ± 4.051e-02	-
f15	1.397e+05 ± 4.857e+05	1.501e+03 ± 4.495e-01	-	1.503e+03 ± 3.994e-01	=
f16	1.604e+03 ± 5.001e-01	1.602e+03 ± 4.185e-01	-	1.603e+03 ± 1.948e-01	-
f17	2.100e+06 ± 7.857e+06	1.800e+03 ± 6.575e+01	-	1.805e+03 ± 2.579e+01	-
f18	1.085e+04 ± 9.465e+03	1.804e+03 ± 1.486e+00	-	1.806e+03 ± 1.258e+00	-
f19	1.902e+03 ± 1.293e+00	1.901e+03 ± 3.139e-01	-	1.901e+03 ± 2.058e-01	-
f20	2.323e+08 ± 4.766e+08	2.001e+03 ± 6.362e-01	-	2.001e+03 ± 5.640e-01	-
f21	2.666e+04 ± 1.508e+04	2.112e+03 ± 1.047e+01	-	2.107e+03 ± 4.574e+00	-
f22	3.273e+03 ± 1.292e+03	2.212e+03 ± 7.080e+00	-	2.207e+03 ± 5.074e+00	-
f23	2.628e+03 ± 5.249e+00	2.629e+03 ± 4.655e-12	=	2.629e+03 ± 8.007e-13	+
f24	2.573e+03 ± 3.438e+01	2.517e+03 ± 4.409e+00	-	2.536e+03 ± 3.958e+00	-
f25	2.699e+03 ± 1.116e+01	2.646e+03 ± 2.138e+01	-	2.659e+03 ± 1.921e+01	-
f26	2.727e+03 ± 4.403e+01	2.700e+03 ± 5.046e-02	-	2.700e+03 ± 3.715e-02	-
f27	3.097e+03 ± 8.157e+01	2.759e+03 ± 1.271e+02	-	2.716e+03 ± 7.126e+01	-
f28	3.406e+03 ± 1.715e+02	3.163e+03 ± 2.163e+01	-	3.164e+03 ± 2.359e+01	-
f29	1.415e+07 ± 7.296e+07	3.130e+03 ± 6.869e+00	-	3.080e+03 ± 2.547e+01	-
f30	4.487e+03 ± 3.685e+02	3.583e+03 ± 3.742e+01	-	3.594e+03 ± 3.013e+01	-

According to the statistical results above, the overall performance of jDES along all the problem functions, is better than DE, meanwhile S performance is satisfactory, thus in this case of 10-D the (low dimensionality) the jDES population-based solution showed more robust performance due to its

comprehensive search mechanism comparing to the single based solution and ist counterpart of DE algorithm in low dimensions.

- Comparison results of 50-D

	S	jDES	W	DE	W
f1	3.891e+05 ± 1.806e+05	1.304e+07 ± 5.029e+06	+	8.246e+08 ± 1.268e+08	+
f2	1.074e+10 ± 5.781e+10	7.458e+03 ± 8.376e+03	=	1.270e+10 ± 1.667e+09	+
f3	3.740e+04 ± 1.527e+04	2.268e+04 ± 1.082e+04	-	1.054e+05 ± 1.172e+04	+
f4	4.225e+02 ± 3.876e+01	5.088e+02 ± 3.650e+01	+	3.921e+03 ± 4.184e+02	+
f5	5.200e+02 ± 5.061e-07	5.200e+02 ± 6.116e-03	=	5.212e+02 ± 4.814e-02	+
f6	6.340e+02 ± 4.813e+00	6.364e+02 ± 4.995e+00	=	6.694e+02 ± 1.789e+00	+
f7	7.000e+02 ± 1.208e-02	7.000e+02 ± 1.071e-02	+	8.203e+02 ± 1.427e+01	+
f8	8.215e+02 ± 3.619e+00	8.191e+02 ± 8.195e+00	=	1.272e+03 ± 1.756e+01	+
f9	1.131e+03 ± 4.940e+01	1.185e+03 ± 7.854e+01	+	1.417e+03 ± 2.066e+01	+
f10	2.219e+03 ± 2.032e+02	2.361e+03 ± 4.076e+02	=	1.338e+04 ± 3.433e+02	+
f11	7.247e+03 ± 7.564e+02	7.177e+03 ± 8.043e+02	=	1.468e+04 ± 3.967e+02	+
f12	1.200e+03 ± 1.168e-01	1.200e+03 ± 1.071e-01	=	1.204e+03 ± 2.732e-01	+
f13	1.301e+03 ± 1.415e-01	1.300e+03 ± 6.243e-02	-	1.301e+03 ± 2.740e-01	+
f14	1.401e+03 ± 3.405e-01	1.400e+03 ± 1.239e-01	-	1.428e+03 ± 4.119e+00	+
f15	1.521e+03 ± 6.635e+00	1.519e+03 ± 3.732e+00	=	7.700e+04 ± 2.686e+04	+
f16	1.621e+03 ± 7.189e-01	1.621e+03 ± 6.412e-01	=	1.623e+03 ± 2.184e-01	+
f17	1.149e+05 ± 5.716e+04	3.583e+06 ± 2.008e+06	+	2.075e+07 ± 4.139e+06	+

f18	4.595e+03 ± 1.097e+03	3.870e+03 ± 1.381e+03	=	3.726e+05 ± 1.275e+05	+
f19	1.916e+03 ± 2.542e+00	1.921e+03 ± 3.194e+00	+	1.942e+03 ± 3.169e+00	+
f20	8.743e+07 ± 2.671e+08	1.712e+04 ± 4.721e+03	-	3.821e+04 ± 9.079e+03	-
f21	2.010e+05 ± 1.215e+05	9.499e+05 ± 6.335e+05	+	2.918e+06 ± 6.070e+05	+
f22	3.781e+03 ± 3.773e+02	3.158e+03 ± 3.219e+02	-	3.914e+03 ± 1.138e+02	=
f23	2.641e+03 ± 1.124e-06	2.641e+03 ± 1.021e-01	+	2.664e+03 ± 2.100e+00	+
f24	2.677e+03 ± 3.782e+00	2.666e+03 ± 3.139e+00	-	2.735e+03 ± 3.962e+00	+
f25	2.714e+03 ± 5.831e+00	2.719e+03 ± 5.630e+00	+	2.807e+03 ± 9.611e+00	+
f26	2.781e+03 ± 1.093e+02	2.700e+03 ± 8.628e-02	-	2.707e+03 ± 4.680e-01	=
f27	3.908e+03 ± 1.073e+02	4.015e+03 ± 1.565e+02	+	4.741e+03 ± 3.494e+01	+
f28	5.742e+03 ± 1.150e+03	4.998e+03 ± 4.050e+02	-	1.054e+04 ± 4.944e+02	+
f29	1.451e+08 ± 7.751e+08	2.310e+04 ± 1.414e+04	-	3.018e+07 ± 8.396e+06	-
f30	1.718e+04 ± 3.996e+03	1.678e+04 ± 2.859e+03	=	3.283e+05 ± 6.723e+04	+

The test results above again shows us again that as the dimensions increases the jDES in general performing better than DE, however the with increasing in the dimension the single solution algorithm performed better than both population based algorithms jDES and DE, this happens due to the exhausting of computational budget in global search (since we have increasing in the number of population “exploration of search space”), further more in the case of S single solution the perturbation and exploitation mechanism and it has no population that exhaust its computational budget, thus the S algorithm can focuses to converge into more optimal basin of attraction.

7 References

[1] Iacca, G., Caraffini, F., Neri, F., and Mininno, E. (2013b).

Single particle algorithms for continuous optimization.

In Evolutionary Computation (CEC), 2013 IEEE Congress on, pages

1610–1617.

[2] 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.

[3] Zhang, J. and Sanderson, A. C. (2009).

Jade: Adaptive differential evolution with optional external archive.

IEEE Transactions on Evolutionary Computation, 13(5):945–958.

[4] 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.

[5] Islam, S., Das, S., Ghosh, S., Roy, S., and Suganthan, P. (2012). An adaptive differential evolution algorithm with novel mutation and crossover strategies for global numerical optimizationcriteri. Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on, 42(2):482–500.

[6] Problem definitions and evaluation criteria for the CEC 2014 special session and competition on single objective real-parameter numerical optimization December 2013 Affiliation: Zhengzhou Univ., China, and Nanyang Technol. Univ., Singapore