Arithmetic Coding Differential Evolution with Tabu Search

Orawan Watchanupaporn
Department of Computer Science
Kasetsart University Sriracha Campus, Thailand
Email: orawan.liu@gmail.com

Worasait Suwannik
Department of Computer Science
Kasetsart University, Thailand
Email: worasait.suwannik@gmail.com

Prabhas Chongstitvatana
Department of Computer Engineering
Chulalongkorn University, Thailand
Email: prabhas@chula.ac.th

***Abstract*—This paper combines Arithmetic Coding Differential Evolution (ACDE) with Tabu Search. ACDE is a population based binary optimization algorithm. Tabu Search is a local search algorithm. The proposed algorithm can solve very difficult problems reliably and quickly. From the experiment, it can always find an optimum solution for 120-bit Trap, 256-bit HIFF, and 243-bit hTrap in 23, 367, and 60 milliseconds respectively on Intel Core i7. The number of evaluation, assumed being in a polynomial class, is about $O(n^{2.10})$, $O(n^{2.45})$, and $O(n^{1.92})$, where n is a problem size.**

Keywords*—*local search, population based search, binary optimization.

I. Introduction

Search algorithms can be categoried into two classes: local and population-based search. For each iteration in a local search algorithm, neighbors of a point in a search space are explored. For binary optimization problem, a point in a search space is a binary string. Its neighbor is another binary string with one bit difference. Examples of local search algorithms are Hill Climbing, Simulated Annealing, and Tabu Search. In contrast, for each iteration in a population based search algorithm, several (unrelated or seemingly unrelated) points in the search space are explored. Populations based search algorithms includes Genetic Algorithm, Genetic Programming, and Differential Evolution. The first one is a binary optimizer.

This paper proposes a hybrid of local and population-based search algorithms. The goal is to solve difficult binary optimization problems quickly. We combine Tabu Search with an extension of Differential Evolution to solve 3 difficult binary optimization problems: Trap, HIFF, and hTrap. From a preliminary experiment, Tabu Search or Differential Evolution alone cannot solve Trap problems reliably. However, a combination can find the optimal solutions.

II. Literature Review

A. Estimation of Distribution Algorithms

Estimation of Distribution Algorithms (EDAs) is a new class of Evolutionary Algorithm (EA). EDA evolves a population in more principle manner than Genetic Algorithm (GA) [1] [2], which is its predecessor. EDA builds a probabilistic model of highly-fit chromosomes and generates a new population from the model. EDAs can be categorized by the relationship between variables: independent variable, bivariate dependencies, and multiple dependencies [3]. For example, Compact Genetic Algorithm (cGA) is a univariate EDA because it does not model a relationship between variables [4]. MIMIC model dependencies between two variables [5]. BOA [6], iBOA [7], and hBOA [8] are multivariate EDA. The main advantage of iBOA over BOA is that iBOA does not need to maintain a population of candidate solutions thus memory requirement is lower. hBOA is suitable for a hierarchical problem.

In this paper, we compare the performance of the proposed algorithm with BOA, iBOA, hBOA. BOA and its successors are considered as representative of state of the art binary optimizers.

B. Arithmetic Coding Differential Evolution

Differential Evolution (DE) is an evolutionary algorithm designed for solving real value optimization problems [9]. DE is a population-based algorithm. The first generation of population is created randomly. A new generation is created by the following methods. Each vector competes with its trial vector. The one with less cost (i.e., the better one) is selected to the next generation. A trial vector is created by combining the original vector with a mutant vector. The combination is similar to crossover in Genetic Algorithm. The mutant vector is created by adding a random vector with a weight difference of other two random vectors (hence the name Differential Evolution). A pseudo of DE is shown in Figure 1. This code is adapted from C code from [15].

DE performs very well in continuous optimization. DE is adapted to solve binary optimization problem by using Arithmetic Coding decompression algorithm [10]. The resulting algorithm is called ACDE.

Arithmetic coding compression algorithm represents a binary string by two real numbers ranged between [0, 1) [11]. The first number is the probability that zero will occurs in the binary string. The second number is the compressed message. The first number is denoted by p and the second number is denoted by c. The coding is best explained by an illustration. The following example demonstrates a decompression of (p, c) = (0.4, 0.6) to a 4-bit binary string. As shown in Figure 2, p divides the interval [0, 1) into 2 sub-intervals: [0, 0.4) and [0.4, 1). Since the compressed message c is in the second sub-interval, the algorithm outputs 1. Next, the algorithm partitions the second interval [0.4, 1) into two sub-intervals proportional to p. The resulting subintervals are [0.4, 0.64) and [0.64, 1). Since the compressed message c is in the first sub-interval, the algorithm outputs 0. Then, the algorithm

```
while (!terminate()) {
   for (i = 0; i < NP; i++) {
      (a, b, c) = randomDifferentInts(NP)

      j = randomInt(D)
      for (k = 1; k <= D; k++) {
         if (random() < CR || k == D)
            trial[j] = x1[c][j] +
                       F*(x1[a][j]-x1[b][j])
         else
            trial[j] = x1[i][j]
         j = (j+1) % D
      }
      score = evaluate(trial)
      if (score <= cost[i]) {
         cost[i]  = score;
         x2[i][j] = copyFrom(trial[j])
      } else {
         x2[i][j] = copyFrom(x1[i][j])
      }
   }
   swapArray(x1, x2)
}
```

where

- `randomDifferentInts(n)` returns random integers in [0, n). The returned numbers are distinct.
- `randomInt(n)` returns a random integer in [0, n).
- `random()` returns a real number in [0, n).

Fig. 1. Differential Evolution pseudo code

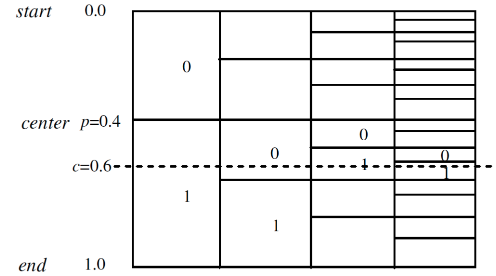

Fig. 2. Decompressing 4 bits from (p, c)=(0.4, 0.6). The output is 1011

```
acdeEvaluation(realVector) {
    binaryString = decompress(realVector)
    return evaluate(binaryString)
}
```

Fig. 3. The pseudo code of ACDE evaluation

```
TabuSearch {
   sol = initial solution
   while (not terminate())
      neighbors = generate_neighbors(sol)
      sol = select(neighbors,
                   tabu_list,
                   aspiration_criteria)
      update(tabu_list)
      update(aspiration_criteria)
   return sol
}
```

Fig. 4. Tabu Search pseudo code

partitions the first interval [0.4, 0.64) into two sub-intervals proportional to p. The resulting sub-intervals are [0.4, 0.496) and [0.496, 0.64). Since the compressed message c is in the second sub-interval, the algorithm outputs 1. Finally, the algorithm partitions the second interval [0.496, 0.64) into two sub-intervals proportional to p. The resulting sub-intervals are [0.496, 0.5536) and [0.5536, 0.64). Since the message c is in the second sub-interval, the algorithm outputs 1.

The difference between ACDE and DE is in the evaluation of a real-value vector. In ACDE, a real-value vector is an array of c's in Arithmetic Coding. p is not used in order to reduce the dimension of the problem by half. A real value is decompressed to a fixed-length binary string using Arithmetic Coding. After that, all binary strings are concatenated. Then, the resulting string is evaluated its fitness is returned to DE. As a result, ACDE is an algorithm that evolves a population of binary strings. A pseudo code of ACDE evaluation is shown in Figure 3.

In this paper, we propose a combination of ACDE and Tabu Search.

C. Tabu Search

Tabu Search [12] is a local search method. A local search method starts from an initial candidate solution. After that, it moves from the candidate solution to one of its neighbors. Generating the list of neighbors is problem dependent. For example, in Traveling Salesperson Problem (TSP), a neighbor can be generated by swapping two cities in the path. Our interested is to solve a generic binary optimization problem. Thus, we generated a neighbor by flipping one bit in the binary string. For example, suppose the current solution is 0100. Its neighbors are 1100, 0000, 0110, and 0101.

Local search methods are varied by a neighbor selection method. Hill Climbing selects the better neighbor with the best objective value. If no such neighbor can be found, the search is terminated. Simulated Annealing may select inferior neighbor based on the objective value and period of the search. This strategy can help avoiding a local optima. Tabu Search select the best neighbor that is not restricted (hence the name tabu search). The restriction avoid repetition and induces the search to explore new region of search space. However, Tabu Search may select a neighbor that is restricted if it meets an aspiration criteria. An example of the criteria is the best solution ever found during the search period. The pseudo code of Tabu Search is shown in Figure 4.

In this paper, we use OpenTS framework of as an implementation Tabu Search algorithm [13]. The framework is written in Java, which is platform-independent and has large amount of libraries. The framework provides both single and multithreaded search. We use the single threaded search in this paper.

III. Proposed Algorithm

The proposed algorithm, ACDETS, is almost the same as ACDE except at the end of each iteration. At that point, ACDETS sends the best vector to Tabu Search as an initial

```
bestRealVector = getBestRealVector()
binaryString = decompress(bestRealVector)
binaryString = tabuSearch(binaryString,
                          objectiveFn)
bestRealVector = compress(binaryString)
```

Fig. 5. The pseudo code of ACDETS at the end of each iteration

solution. After that, the result from Tabu Search is compressed and replaces the best vector. The extra code is shown in Figure 5.

IV. EXPERIMENT

A. Test Problems

1) Trap: We compare the performance using Trap problem. Trap problem is a binary optimization problem. The problem is designed to resist Hill Climbing search. To calculate an objective value, a binary string is decomposed to groups of k bits. Each group may contain contiguous or noncontiguous bits. Trap problems with contiguous and noncontiguous bit groups are referred to as $Trap_c$ and $Trap_{nc}$ respectively. The equation of $Trap_c$ (group size 5) is defined as:

$$trap_c(X) = \sum_{i=1}^{|X|/5} f(u(block(X,i)))$$

where

- $block(X,i)$ returns the ith block of X. Each block has 5 bits.
- $u(X) = \sum_{b \in X} b$
- $f(n)$ returns 4, 3, 2, 1, 0, 5 when n is 0, 1, 2, 3, 4, 5 respectively. This function fools a hill climbing algorithm to return all 0's but a better solution is all 1's.

The optimal solution of $Trap_c$ is all 1's. In [7], $Trap_c$ is the only benchmark problem for comparing the performance of iBOA and BOA. However, in this paper, we do not use $Trap_c$ because Tabu Search can solve $Trap_c$ much more efficiently than iBOA and BOA (see Table IV).

The optimal solution of $Trap_{nc}$ is also all 1's. ACDE uses compression algorithm which has bias towards regular solution. Therefore, we use $Trap_r$, a random version of $Trap_{nc}$ problem. The optimal solution is randomly generated for each run. The function u is replaced by a function that counts the number of bits in a candidate solution that match the corresponding bit in the optimal solution.

2) HIFF: HIFF (hierarchical if and only if) is a binary optimization problem with a hierarchical structure [14]. The problem is designed to have strong dependencies and have a recursive property. Hill Climbing algorithm cannot find optimal solution of this problem. The problem size is 2^k, where $k \geq 2$. Each bit in a binary string is a value of a leaf of a perfect binary tree. The tree is evaluated bottom up. A leaf node contribute to the objective value by 1. Another node contributes by 2^h where h is its height. It will contribute if its children are both 0's or 1's. The value of an interior node is 1 if both children are 1 (and same for 0). Otherwise, its value will be neither 0 nor 1 (represented by -). The objective value of the whole string is summation of contribution from all nodes. The HIFF is defined as:

$$hiff(X) = \begin{cases} 1, & \text{if } c_1 \\ hiff(X_L) + hiff(X_R) + |X|, & \text{if } c_2 \\ hiff(X_L) + hiff(X_R), & \text{otherwise} \end{cases}$$

where

- c_1 is $|X| = 1$
- c_2 is $|X| > 1$ and $(\forall_{b \in X}(b = 0)$ or $(\forall_{b \in X}(b = 1))$
- $X = X_L || X_R$ and $|X_L| = |X_R|$

In this paper, we use a random version of HIFF, which is called $HIFF_r$. Calculating an objective value is the same as HIFF except for a parent of a leaf node. The parent will contribute if both or neither children match the corresponding part of optimal solution, which is randomly generated for each run. If both leaves match the corresponding part, its parent will be 1. If neither match, its parent will be 0. Otherwise, its parent is -.

3) hTrap: hTrap (hierarchical Trap) is similar to HIFF problem [8]. However, its structure is a perfect k-ary tree. In this paper, k=3 is used. An interior node contribute by 3^h multiply by a value return from a $trap_other$ function. For k=3, the returned value is 1.0, 0.5, 0.0, or 1.0 if there are 0, 1, 2, or 3 children that are 1's. The root will contribute using multiplication of 3^h and a value returned from a $trap_root$ function. For k=3, the returned value is 0.90, 0.45, 0.00, or 1.00 if there are 0, 1, 2, or 3 children that are 1's. A node will not contribute if it has a child with -.

The $hTrap$ is defined as:

$$ht(X) = \begin{cases} 0, & \text{if } c_1 \\ ht(X_L) + ht(X_C) + ht(X_R) + f(X), & \text{if } c_2 \\ ht(X_L) + ht(X_C) + ht(X_R), & \text{otherwise} \end{cases}$$

where

- c_1 is $|X| = 1$
- c_2 is $|X| > 1$ and $(\forall_{p \in \{L,C,R\}} \forall_{b \in X_p}(b = 0)$ or $(\forall_{p \in \{L,C,R\}} \forall_{b \in X_p}(b = 1))$
- $X = X_L || X_C || X_R$ and $|X_L| = |X_C| = |X_R|$
- $f(X) = trap(g(X), |X|)$
- $g(X) = \frac{u(X)}{3^{log_3|X|-1}}$
- $trap(C,S) = \begin{cases} trap_{root}(C), & \text{if } S = \text{problem size} \\ trap_{other}(C), & \text{otherwise} \end{cases}$
- $trap_{root}(C)$ = 0.90, 0.45, 0.00, or 1.00 if C = 0, 1, 2, or 3 respectively.
- $trap_{other}(C)$ = 1.00, 0.50, 0.00, or 1.00 if C = 0, 1, 2, or 3 respectively.

In this paper, $hTrap_r$, a random version of hTrap, is used.

TABLE I. ACDETS PARAMETERS FOR $Trap_r$

Parameter	Problem size (bit)			
	15	30	60	120
NP	300	600	1200	2400
Tenure size	12	25	50	100
Iteration	15	30	60	120

TABLE II. ACDETS PARAMETERS FOR $HIFF_r$

Parameter	Problem size (bit)		
	64	128	256
NP	80	160	320
Tenure size	40	70	130
Iteration	40	70	130

TABLE III. ACDETS PARAMETERS FOR $hTrap_r$

Parameter	Problem size (bit)		
	27	81	243
NP	30	90	270
Tenure size	8	15	30
Iteration	27	50	100

TABLE IV. TABU SEARCH ON VARIOUS 120-BIT TRAP PROBLEMS

Problem	No. of evaluations	Avg. best fitness (%found)	Time (ms)
$Trap_c$	16909.00	-120.00 (100)	6.72
$Trap_{nc}$	-	-100.30 (0)	-
$Trap_r$	-	-100.36 (0)	-

TABLE V. PERFORMANCE OF VARIOUS ALGORITHMS TESTED ON 120-BIT $Trap_r$ PROBLEM

Algorithm	Avg. best fitness (%found)	Time (ms)
TS	-114.47 (0)	-
ACDE	-74.55 (0)	-
ACDETS	-120.00 (100)	29.21

TABLE VI. PERFORMANCE OF BOA IN SOLVING $Trap_c$ PROBLEM

Problem size	Evaluations	Time (ms)
15	4550	15.16
30	13300	88.68
60	36400	748.67
120	95200	8579.34

B. Description of Experiment

We run the experiment 100 times. The number of the run is used as a seed to random the optimal solution of each run. The experiment focus on the number of successful runs, number of evaluations, and running time. ACDETS is implemented in Java. The program runs on 2.3GHz Intel Core $i7$ with 4GB 1333MHz DDR3. The operating system is Mac OS X 10.9.2.

1) $Arithmetic\ Coding\ Parameter$: The number of decompressed bit per DE real value is 5, 9, and 8 for $Trap_r$, $HIFF_r$, and $hTrap_r$ respectively. Those numbers divides problem sizes in the experiment. This parameter affects the size of DE real vector and DE population size. For example, a 120-bit $Trap_r$ problem requires 24 real numbers in a DE real value vector. Note that, even though this parameter for $Trap_r$ is equal to the size of a group in $Trap_r$ problem, this would not help finding the solution. This is because bits in a group is not usually located contiguously.

2) $Differential\ Evolution\ Parameters$: For all problems, NP (i.e., DE population size) is set to 10 times the size of real value vector (see Table I to III). F (i.e., a DE weight) and CR (i.e., probability that a real value in DE is from a noisy random vector) are set to 0.5. Those parameters are suggested in [15]. The maximum number of generation is 100, which is enough to find the optimal solution.

3) $Tabu\ Search\ Parameters$: The number of TS iterations has significant effect on the performance of the overall algorithm. A single ACDE iteration evaluates NP real vectors, which is not more than twice the problem size (in bits). However, each TS iteration evaluates all neighbors of a binary string, which number is equal to the problem size (in bits). A number of iterations and tenure size are obtained by trial and error (see Table I to III).

V. RESULT

We compare the performance of each optimization algorithm which are BOA, iBOA, and ACDETS based on the number of evaluations. Optimization will stop when the optimum is found. The numbers of BOA and iBOA are estimation obtained from [7].

For DE, the lower objective value means the better solution. Thus, the result from each objective function is multiplied by -1.

A. $Trap_r$

Table V shows the average best fitness and running time of TS, ACDE, and ACDETS on 120-bit $Trap_r$ problem. The number in the parenthesis is the percentage of success. TS and ACDE cannot find the optimal solution of Trap. However, their combination (ACDETS) can solve the problem reliably and efficiently. This shows that local search can improve the performance of ACDE.

We cannot find the report on the running time of BOA and hBOA. Therefore, we conduct a small experiment to measure a BOA running time. We used the C++ BOA code written by Martin Pelikan [16]. The population sizes are 700, 1400, 2800, and 5600 for 15-, 30-, 60-, and 120-bit problems. The seed is set to 123. The result is shown in Table VI. The average running time for BOA is in $O(n^{3.05})$ seconds while the average running time for ACDETS is in $O(n^{1.61})$ seconds. To solve a 120-bit problem, BOA takes 0.09012 ms per evaluation on average while ACDTS takes 0.00067 ms per evaluation (135 times faster). hBOA code is not available for download.

B. HIFF

Table VIII shows the average number of evaluations required by hBOA and ACDETS to solve HIFF and $HIFF_r$ problems respectively. We do not have the data of number of generation and running time for hBOA. Therefore, we only compare the average number of evaluations of hBOA and ACDETS. The result show that hBOA outperforms ACDETS in terms of the number of evaluation. However, in terms of running time per one objective evaluation, ACDETS should be faster due to its simplicity.

TABLE VII. PERFORMANCE OF ACDETS (ON $Trap_r$) AND BOA (ON $Trap_c$). NOTE THAT THE NUMBERS OF BOA AND IBOA ARE ROUGH ESTIMATION OBTAINED FROM [7]

Problem size	Algorithm	No. of evaluations	Gen.	Time (ms)
15	ACDETS	429.21	1.71	0.81
	BOA	3000.00	-	-
	iBOA	8000.00	-	-
30	ACDETS	1837.22	2.12	1.43
	BOA	10000.00	-	-
	iBOA	20000.00	-	-
60	ACDETS	7782.01	2.41	4.33
	BOA	50000.00	-	-
	iBOA	60000.00	-	-
120	ACDETS	34086.28	2.68	22.97
	BOA	-	-	-
	iBOA	-	-	-

TABLE VIII. PERFORMANCE OF ACDETS (ON $HIFF_r$) AND HBOA (ON HIFF). NOTE THAT THE NUMBERS OF HBOA ARE ROUGH ESTIMATION OBTAINED FROM [8]

Problem size	Algorithm	No. of evaluations	Gen.	Time (ms)
64	ACDETS	8592.32	3.52	5.94
	hBOA	7000.00	-	-
128	ACDETS	46665.86	5.38	42.47
	hBOA	25000.00	-	-
256	ACDETS	257006.28	7.88	368.86
	hBOA	85000.00	-	-

TABLE IX. PERFORMANCE OF ACDETS (ON $hTrap_r$) AND HBOA (ON HTRAP). NOTE THAT THE NUMBERS OF HBOA ARE ROUGH ESTIMATION OBTAINED FROM [8]

Problem size	Algorithm	Fitness	Evaluation	Gen.	Time (ms)
27	ACDETS	-27.00	685.34	1.16	0.92
	hBOA		3000.00	-	-
81	ACDETS	-108.00	5525.37	1.89	4.02
	hBOA		30000.00	-	-
243	ACDETS	-405.00	46151.31	2.10	59.81
	hBOA		200000.00	-	-

TABLE X. THE PERFORMANCE OF TS ON $hTrap_r$ PROBLEM

Problem size	Tenure	Iteration	Evaluation
27	10	100	690.85
81	15	200	4859.38
243	20	300	44370.37

C. hTrap

ACDETS outperforms hBOA in hTrap in terms of the number of evaluations (see Table IX). In fact, TS alone can outperform hBOA (see Table X). The result is surprising because hTrap is supposed to be the most difficult problem in this experiment.

VI. DISCUSSION

We conduct another experiment to see the interaction between Tabu Search and ACDE. At the end of each iteration, ACDETS selects the best solution and uses it as initial solution for Tabu Search. Table XI to XIII shows how much Tabu Search can improve the best solution of each ACDE iteration in solving 120-bit $Trap_r$ 256-bit $HIFF_r$, and 243-bit $hTrap_r$ problem. The column Run shows the number of runs that

TABLE XI. IMPROVEMENT OF THE BEST SOLUTION OF EACH ITERATION OBTAINED FROM ACDE IN 120-BIT $Trap_r$ PROBLEM BY TABU SEARCH

Generation	Improvement	Run
1	56.13	100
2	3.74	100
3	1.11	79
4	1.00	1

TABLE XII. IMPROVEMENT OF THE THE BEST SOLUTION OF EACH ITERATION OBTAINED FROM ACDE IN 120-BIT $HIFF_r$ PROBLEM BY TABU SEARCH

Generation	Improvement	Run
1	524.06	100
2	299.90	100
3	196.72	100
4	141.60	100
5	124.08	98
6	113.43	87
7	163.32	77
8	202.43	61
9	212.31	39
10	231.83	23
11	248.00	11
12	278.40	5
13	384.00	1

TABLE XIII. IMPROVEMENT OF THE BEST SOLUTION OF EACH ITERATION OBTAINED FROM ACDE IN 120-BIT $hTrap_r$ PROBLEM BY TABU SEARCH

Generation	Improvement	Run
1	122.38	100
2	216.63	100
3	124.7	14

reach the corresponding generation. Note that for Trap problem changing objective value from 119 to 120 (i.e., the optimal solution), Tabu Search has to change at least 5 bits.

In this paper, we use simplest scheme of Tabu Search. The scheme requires 2 parameters; number of iteration and tenure size. Lowering the number of iterations will significantly reduce number of overall evaluations. This is because, for 120-bit $Trap_r$ experiment, Tabu Search evaluates 40 times more than ACDE. However, when the number of iteration is changed from 160 to 80, the solution can be found only 15%. Similarly, when tenure size is changed from 100 to 50, the solution can be found only 11%.

VII. CONCLUSION

This paper combines ACDE with Tabu Search. Each algorithm alone cannot solve $Trap_r$, and $HIFF_r$ problems. However, the combination, ACDETS, can always find the optimal solution for all test problems in the experiment. Its speed is vary fast. However, to be successfully solving various sizes of problems, adjusting Tabu Search parameter is required.

ACKNOWLEDGMENT

The authors would like to thank Thotsaporn Thanatipanonda for suggesting how to calculate scalability.

REFERENCES

[1] D.E. Goldberg, Genetic Algorithms in Search, Optimization, and Machine Learning, Addison-Wesley, 1989.

[2] M. Mitchell, An Introduction to Genetic Algorithms, MIT Press, 1998.

[3] T.K. Paul and H. Iba, "Linear and Combinatorial Optimizations by Estimation of Distribution Algorithms," In Proceedings of 9th MPS Symposium on Evolutionary Computation (IPSJ), 2002.

[4] G. Harik, F.G. Lobo, D.E. Golberg, "The compact genetic algorithm." In Proceedings of the IEEE Conference on Evolutionary Computation, pp. 523-528, 1998.

[5] J.S. De Bonet, C.L. Isbell, and P. Viola, "MIMIC: Finding optima by estimating probability densities," Advances in Neural Information Processing Systems, 9, 1997.

[6] M. Pelikan, D.E. Goldberg, and E. Cantú-Paz, "BOA: The Bayesian optimization algorithm," In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO), pp. 525-532, 1999.

[7] M. Pelikan, K. Sastry, and D.E. Goldberg, "iBOA: The Incremental Bayesian Optimization Algorithm," In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO), ACM Press, pp. 455-462, 2008.

[8] M. Pelikan and D.E. Goldberg, "Escaping hierarchical traps with competent genetic algorithms," In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO), pp. 511-518, 2001, Also IlliGAL Report No. 2000020.

[9] R. Storn and K. Price, "Differential Evolution-A simple and efficient adaptive scheme for global optimization over continuous spaces,"Technical Report TR-95-012, ICSI, March, 1995.

[10] O. Watchanupaporn and W. Suwannik, "Arithmetic Coding Differential Evolution for Binary Encoding," Advances in Information Technology and Applied Computing (AITAC) ISSN 2251-3418, vol.1, pp. 155-158, 2012.

[11] J. Rissanen, "Generalized Kraft Inequality and Arithmetic Coding," IBM J. Res. Develop., vol. 20, pp. 198-203, 1976.

[12] F. Glover, Tabu Search: A Tutorial, Interface, pp. 74-94, 1990.

[13] R. Harder, OpenTS, http://www.coin-or.org/Ots, accessed April. 8. 2014.

[14] R. Watson, G.S. Hornby, and J.B. Pollack, "Modeling Building-Block Interdependency," In Parallel Problem Solving from Nature - PPSN V, Lecture Notes in Computer Science, Springer, vol. 1498, pp.97-106, 1998.

[15] K. Price and R. Storn, "Differential Evolution," Dr.Dobb's Journal, http://www.drdobbs.com/database/differential-evolution/184410166, 1997, accessed April. 8. 2014.

[16] M. Pelikan, "A Simple Implementation of Bayesian Optimization Algorithm in C++ (Version 1.0)," Illigal Report 99011, February 1999.