7

Discrete Set Handling

Ivan Zelinka

Tomas Bata Univerzity in Zlin, Faculty of Applied Informatics,
Nad Stranemi 4511, Zlin 76001, Czech Republic
zelinka@fai.utb.cz

Abstract. Discrete Set Handling and its application to permutative problems is presented in this chapter. Discrete Set is applied to Differential Evolution Algorithm, in order to enable it to solve strict-sence combinatorial problems. In addition to the theoretical framework and description, benchmark Flow Shop Scheduling and Traveling Salesman Problems are solved. The results are compared with published literature to illustrate the effectiveness of the developed approach. Also, general applications of Discrete Set Handling to Chaotic, non-linear and symbolic regression systems are given.

7.1 Introduction

In recent years, a broad class of algorithms has been developed for stochastic optimization, i.e. for optimizing systems where the functional relationship between the independent input variables x and output (objective function) y of a system S is not known. Using stochastic optimization algorithms such as Genetic Algorithms (GA), Simulated Annealing (SA) and Differential Evolution (DE), a system is confronted with a random input vector and its response is measured. This response is then used by the algorithm to tune the input vector in such a way that the system produces the desired output or target value in an iterative process.

Most engineering problems can be defined as optimization problems, e.g. the finding of an optimal trajectory for a robot arm, the optimal thickness of steel in pressure vessels, the optimal set of parameters for controllers, optimal relations or fuzzy sets in fuzzy models, etc. Solutions to such problems are usually difficult to find, since their parameters usually include variables of different types, such as floating point or integer variables. Evolutionary algorithms (EAs), such as the Genetic Algorithms and Differential Evolutionary Algorithms, have been successfully used in the past for these engineering problems, because they can offer solutions to almost any problem in a simplified manner: they are able to handle optimizing tasks with mixed variables, including the appropriate constraints, and they do not rely on the existence of derivatives or auxiliary information about the system, e.g. its transfer function.

Evolutionary algorithms work on populations of candidate solutions that are evolved in generations in which only the best−suited − or fittest − individuals are likely to survive. This article introduces Differential Evolution, a well known stochastic optimization algorithm. It explains the principles of permutation optimization behind DE and demonstrates how this algorithm can assist in solving of various permutation optimization problems.

Differential Evolution, which can also works on a population of individuals, is based on a few simple arithmetic operations. Individuals are generated by means of a few randomly selected individuals.

In the following text the principle of the DE algorithm and permutative optimization will be explained. The description is divided into short sections to increase the understandability of principles of DE and permutative optimization.

7.2 Permutative Optimization

A permutative optimization problem is one, where the solution representation is ordered and discrete; implying that all the values in the solutions are firstly *unique*, and secondly *concrete*.

In the general sense, if a problem representation is given as n, then the solution representation is always given as some combination of range $\{1,....,n\}$. For example, given a problem of size 4, the solution representation is $\{1,2,3,4\}$ and all its possible permutative combinations.

Two of the problems solved in this chapter, which are of this nature, are the Traveling Salesman Problem (TSP) and Flow Shop Scheduling (FSS) Problems as discussed in the following sections. The third subsection describes the 2 Opt Local search, which is a routine embedded in this heuristic to find better solutions within the neighbourhood of a solution.

7.2.1 Travelling Salesman Problem

A TSP is a classical combinatorial optimization problem. Simply stated, the objective of a travelling salesman is to move from city to city, visiting each city only once and returning back to the starting city. This is called a *tour* of the salesman. In mathematical formulation, there is a group of distinct cities $\{C_1, C_2, C_3, ..., C_N\}$, and there is given for each pair of city $\{C_i, C_j\}$ a distance $d\{C_i, C_j\}$. The objective then is to find an ordering π of cities such that the total time for the salesman is minimised. The lowest possible time is termed the optimal time. The objective function is given as:

$$\sum_{i=1}^{N-1} d\left(C_{\pi(i)}, C_{\pi(i+1)}\right) + d\left(C_{\pi(N)}, C_{\pi(1)}\right) \tag{7.1}$$

This quality is known as the *tour length*. Two branches of this problem exist, symmetric and asymmetric. A symmetric problem is one where the distance between two cities is identical, given as: $d\{C_i, C_j\} = d\{C_j, C_i\}$ for $1 \le i, j \le N$ and the asymmetric is where the distances are not equal. An asymmetric problem is generally more difficult to solve.

The TSP has many real world applications; VSLA fabrication [7] to X-ray crystallography [1]. Another consideration is that TSP is *NP-Hard* as shown by [12], and so any algorithm for finding optimal tours must have a worst-case running time that grows faster than any polynomial (assuming the widely believed conjecture that $P \neq NP$).

TSP has been solved to such an extent that traditional heuristics are able to find good solutions to merely a small percantage error. It is normal for the simple 3-Opt heuristic typically getting with 3-4% to the optimal and the *variable-opt* algorithm of [20] typically getting around 1-2%.

The objective for new emerging evolutionary systems is to find a guided approach to TSP and leave simple local search heuristics to find better local regions, as is the case for this chapter.

7.2.2 Flow Shop Scheduling Problem

A flow shop is a scheduling problem, typical for a manufacturing floor. The terminology for this problem is typical of a manufacturing sector. Consider n number of jobs $j\,(i=1,...n)$, and a number of machines M: $M\,(j=1,....,m)$.

A job consists of m operation and the j^{th} of each job must be processed on machine j. So, one job can start on machine *j* if it is completed on machine *j-1* and if machine *j* is free. Each job has a known processing time $p_{i,j}$. The operating sequence of the jobs is the same on all the machines. If one job is at the i^{th} position on machine 1, then it will be on the i^{th} position on all machines.

The objective function is then to find the minimal time for the completion of all the jobs on all the machines. A job J_i is a sequence of operations, having one operation for each of the M machines.

1. $J_i = \{O_{i1}, O_{i2}, O_{i3}, .., O_{iM}\}$, where O_{ij} represents the j^{th} operation on J_i.
2. O_{ij} operation must be processed on M_j machine.
3. for each operation O_{ij}, there is a processing time $p_{i,j}$.

Now let a permutation be represented as $\{\prod_1, \prod_2, ..., \prod_N\}$. The formulation of the completion time for $C(\prod_i, j)$, for the i^{th} job on the j^{th} machine can be given as:

$$\begin{array}{ll} C(\prod_1,1) = p_{\prod},1 & \\ C(\prod_1,1) = C(\prod_{i-1},1) + p_{\prod},1, & i=2,...,N \\ C(\prod_1,j) = C(\prod_1,j-1) + p_{\prod 1},j, & i=2,...,M \\ C(\prod_1,j) = \max\{C(\prod_{i-1},1), C(\prod_1,j-1)\} + p_{\prod 1},j, & i=2,...,N; j=2,..,M \end{array} \tag{7.2}$$

The makespan or the completion time is given as the $C(\prod_N, M)$, as the completion time of the *last* job in the schedule on the *last* machine.

7.2.3 2 Opt Local Search

A local search heuristic is usually based on simple tour modifications (exchange heuristics). Usually these are specified in terms of the class of operators (exchanges/moves), which is used to modify one tour into another. This usually works on a feasible tour, where a *neighborhood* is all moves, which can be reached, in a single move. The tour iterates till a better tour is reached.

Among simple local search algorithms, the most famous are 2−Opt and 3−Opt. The 2-Opt algorithm was initially proposed by [2] although it was already suggested by [5]. This move deletes two edges, thus breaking the tour into two paths, and then reconnects those paths in the other possible way as given in Fig 7.1.

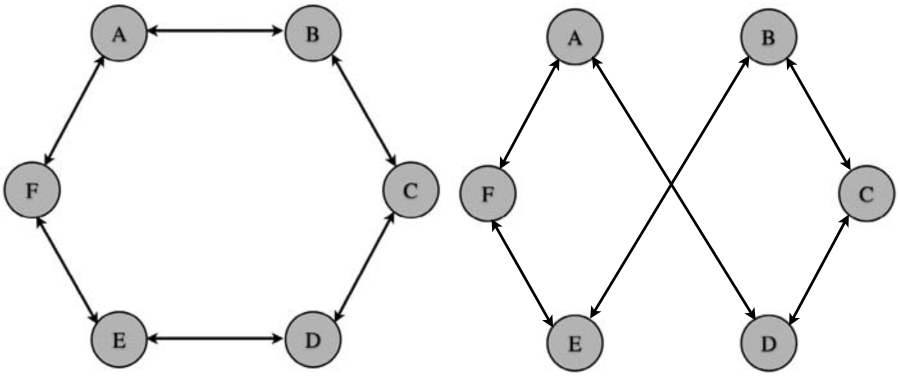

Fig. 7.1. 2-Opt exchange tour

7.3 Discrete Set Handling and Its Application

7.3.1 Introduction and Principle

In its canonical form, DE is only capable of handling continuous variables. However, extending it for optimization of integer variables is rather easy. Only a couple of simple modifications are required. First, for evaluation of the cost-function, integer values should be used. Despite this, the DE algorithm itself may still work internally with continuous floating-point values. Thus,

$$\begin{aligned}
&f_{\text{cost}}(y_i) \;\; i = 1, .., n_{param} \\
&where: \\
&\quad y_i = \begin{cases} x_i & \text{for continuous variables} \\ INT(x_i) & \text{for integer variables} \end{cases} \\
&x_i \in X
\end{aligned} \tag{7.3}$$

INT() is a function for converting a real value to an integer value by truncation. Truncation is performed here only for purposes of cost function value evaluation. Truncated values are not assigned elsewhere. Thus, EA works with a population of continuous variables regardless of the corresponding object variable type. This is essential for maintaining the diversity of the population and the robustness of the algorithm.

Secondly, in case of integer variables, the population should be initialized as follows:

$$\begin{aligned}
&P^{(0)} = x_{i,j}^{(0)} = r_{i,j}\left(x_j^{(High)} - x_j^{(Low)} + 1\right) + x_j^{(Low)} \\
&i = 1, ..., n_{pop}, \quad j = 1, ..., n_{param}
\end{aligned} \tag{7.4}$$

Additionally, the boundary constraint handling for integer variables should be performed as follows:

$$x_{i,j}^{(ML+1)} = \begin{cases} r_{i,j}\left(x_j^{(High)} - x_j^{(Low)} + 1\right) + x_j^{(Low)} \\ \quad if \quad INT\left(x_{i,j}^{(ML+1)}\right) < x_j^{(Low)} \vee INT\left(x_{i,j}^{(ML+1)}\right) > x_j^{(High)} \\ x_{i,j}^{(ML+1)} \quad \text{otherwise} \end{cases} \tag{7.5}$$

where,

$i = 1, ..., n_{pop}, \quad j = 1, ..., n_{param}$

Discrete values can also be handled in a straight forward manner. Suppose that the subset of discrete variables, *X(d)*, contains *i* elements that can be assigned to variable *x*:

$$X^{(d)} = x_i^{(d)} \qquad i = 1, ..., l \quad \text{where} \quad x_i^{(d)} < x_{i+1}^{(d)} \tag{7.6}$$

Instead of the discrete value x_i itself, its index, *i*, can be assigned to *x*. Now the discrete variable can be handled as an integer variable that is boundary constrained to range $\{1, 2, 3, .., N\}$. In order to evaluate the objective function, the discrete value, x_i, is used instead of its index *i*. In other words, instead of optimizing the value of the discrete variable directly, the value of its index *i* is optimized. Only during evaluation is the indicated discrete value used. Once the discrete problem has been converted into an integer one, the previously described methods for handling integer variables can be applied. The principle of discrete parameter handling is depicted in Fig 7.2.

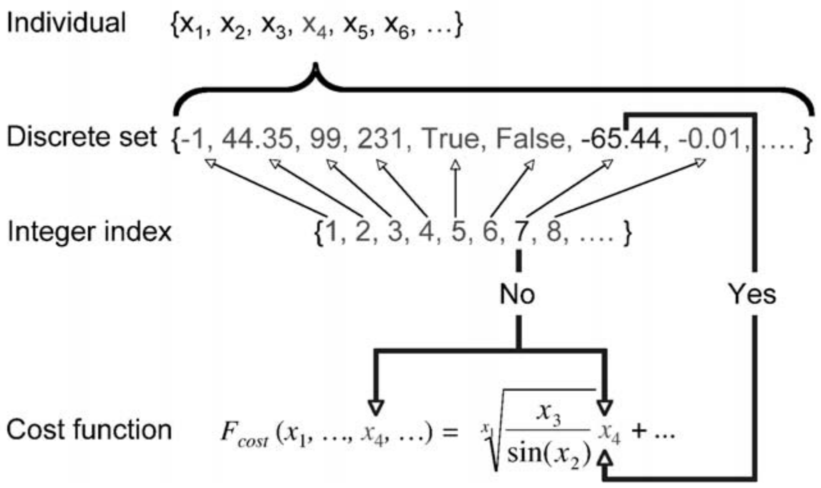
Individual
{x1, x2, x3, x4, x5, x6, ...}
Discrete set
{-1, 44.35, 99, 231, True, False, -65.44, -0.01, ... }
Integer index
{1, 2, 3, 4, 5, 6, 7, 8, ... }
No
Yes
Cost function
Fcost (x1, ..., x4, ...) = x1√(x3/sin(x2)) x4 + ...

Fig. 7.2. Discrete parameter handling

7.3.2 DSH Applications on Standard Evolutionary Algorithms

DSH has been used in many previous experiments in standard EAs as well as in genetic programming like techniques. An example of the usage of DSH in mechanical engineering problem in C++ language in given in Fig 7.3.

Here, only the set of discrete values is described in order to show that DSH is basically a field of values (real values) and individuals in integer form serve like pointers to

```
 // Mixed problem (Integer - Continuous - Discrete) - Case B !!!!!!!!!!!!!!!!!!
 // New Ideas in Optimization - Table 9 : Allowable spring steel wire diameters
 // for the coil spring design problem
discrete[0] = 0.009; discrete[1] = 0.0095; discrete[2] = 0.0104; discrete[3] = 0.0118;
discrete[4] = 0.0128; discrete[5] = 0.0132; discrete[6] = 0.014; discrete[7] = 0.015;
discrete[8] = 0.0162; discrete[9] = 0.0173; discrete[10] = 0.018; discrete[11] = 0.020;
discrete[12] = 0.023; discrete[13] = 0.025; discrete[14] = 0.028; discrete[15] = 0.032;
discrete[16] = 0.035; discrete[17] = 0.041; discrete[18] = 0.047; discrete[19] = 0.054;
discrete[20] = 0.063; discrete[21] = 0.072; discrete[22] = 0.080; discrete[23] = 0.092;
discrete[24] = 0.105; discrete[25] = 0.120; discrete[26] = 0.135; discrete[27] = 0.148;
discrete[28] = 0.162; discrete[29] = 0.177; discrete[30] = 0.192; discrete[31] = 0.207;
discrete[32] = 0.225; discrete[33] = 0.244; discrete[34] = 0.263; discrete[35] = 0.283;
discrete[36] = 0.307; discrete[37] = 0.331; discrete[38] = 0.362; discrete[39] = 0.394;
discrete[40] = 0.4375; discrete[41] = 0.500;
```

Fig. 7.3. C++ DSH code

```
#include <stdlib.h>

int     tempval,MachineJob[25],Cmatrix[5][5];
int     loop1,loop2;

//in C language is [0][0] the first item of defined field i.e. [1][1] of
normaly defined matrix
MachineJob[0]=5;MachineJob[1]=7;MachineJob[2]=4;MachineJob[3]=3;MachineJob[4]=6;
MachineJob[5]=6;MachineJob[6]=5;MachineJob[7]=7;MachineJob[8]=6;MachineJob[9]=7;
MachineJob[10]=7;MachineJob[11]=8;MachineJob[12]=3;MachineJob[13]=8;MachineJob[14]=5;
MachineJob[15]=8;MachineJob[16]=6;MachineJob[17]=5;MachineJob[18]=5;MachineJob[19]=8;
MachineJob[20]=4;MachineJob[21]=4;MachineJob[22]=8;MachineJob[23]=7;MachineJob[24]=3;

for(loop1=0;loop1<25;loop1++)
    {
   tempval=MachineJob[loop1];
    MachineJob[loop1]=MachineJob[getIntPopulation(0,Individual)];
   MachineJob[getIntPopulation(0,Individual)]=tempval;
   };

//Competition time for all jobs on machine 1
Cmatrix[0][0]=MachineJob[0];
Cmatrix[0][1]=MachineJob[0]+MachineJob[1];
Cmatrix[0][2]=MachineJob[0]+MachineJob[1]+MachineJob[2];
Cmatrix[0][3]=MachineJob[0]+MachineJob[1]+MachineJob[2]+MachineJob[3];
Cmatrix[0][4]=MachineJob[0]+MachineJob[1]+MachineJob[2]+MachineJob[3]+MachineJob[4];

//Competition time jobs 1 on all machines
Cmatrix[1][0]=MachineJob[0]+MachineJob[5];
Cmatrix[2][0]=MachineJob[0]+MachineJob[5]+MachineJob[10];
Cmatrix[3][0]=MachineJob[0]+MachineJob[5]+MachineJob[10]+MachineJob[15];
Cmatrix[4][0]=MachineJob[0]+MachineJob[5]+MachineJob[10]+MachineJob[15]+
                MachineJob[20];

for(loop1=1;loop1<5;loop1++)
    for(loop2=1;loop2<5;loop2++)
        Cmatrix[loop1][loop2]=max(Cmatrix[loop1-1][loop2],Cmatrix[loop1][loop2-1])+
        MachineJob[5*loop1+loop2];

CostValue=Cmatrix[4][4];
```

Fig. 7.4. DSH FSS example

that field. A more complex example from FSS is now described in Fig 7.4. Discrete set has the name **MachineJob** and contains different values. Individuals again serve like an index.

More interesting applications of DSH can be found in genetic programming like techniques.

7.3.3 DSH Applications on Class of Genetic Programming Techniques

The term *symbolic regression* represents a process during which measured data sets are fitted such thereby a corresponding mathematical formula is obtained in an analytical way. An output of the symbolic expression could be, for example, $x^2 + y^3/K$, and the like. For a long time, symbolic regression was a domain of human calculations but in the last few decades it involves computers for symbolic computation as well.

The initial idea of symbolic regression by means of a computer program was proposed in Genetic Programming (GP) [8, 9]. The other approaches are Grammatical Evolution (GE) developed in [19, 13] and Analytic Programming (AP) in [27]. Oher interesting investigations using symbolic regression were carried out in [6] on Artificial Immune Systems and Probabilistic Incremental Program Evolution (PIPE), which generates functional programs from an adaptive probability distribution over all possible programs. As an extension of GE to the another algorithms is also [14], where DE was used with the GE. Symbolic regression, generally speaking, is a process which combines, evaluates and creates more complex structures based on some elementary and noncomplex objects, in an evolutionary way. Such elementary objects are usually simple mathematical operators $(+, -, *, ...)$, simple functions (sin, cos, And, Not,.), user-defined functions (simple commands for robots – MoveLeft, TurnRight,.), etc.

An output of symbolic regression is a more complex *object* (formula, function, command,.), solving a given problem like data fitting of the so-called Sextic and Quintic problem described by Equation 7.7) [10, 26], randomly synthesized function by Equation 7.8 [26], Boolean problems of parity and symmetry solution (basically logical circuits synthesis) by Equation 7.9) [11, 27], synthesis of Chaos by utilizing DSH and Evolutionary Algorithms [28] given in Table 7.1 and in Figs 7.7 – 7.10.

Synthesis of quite complex robot control command by Equation 7.10 [10, 15] is also accomplished with DSH. Equation 7.7 – 7.10 mentioned are just a few samples from numerous repeated experiments done by AP, which are used to demonstrate how complex structures can be produced by symbolic regression in general for different problems.

$$x\left(K_1 + \frac{(x^2 K_3)}{K_4\,(K_5 + K_6)}\right) \bullet (-1 + K_2 + 2x(-x - K_7)) \tag{7.7}$$

$$\sqrt{t}\left(\frac{1}{\log(t)}\right)^{\sec^{-1}(1.28)} \log^{\sec^{-1}(1.28)}(\sinh(\sec(\cos(1)))) \tag{7.8}$$

```
Nor[(Nand[Nand[B || B, B && A], B]) && C && A && B,
    Nor[(! C && B && A || ! A && C && B || ! C && ! B && ! A) &&
      (! C && B && A || ! A && C && B || ! C && ! B && ! A) ||
     A && (! C && B && A || ! A && C && B || ! C && ! B && ! A),
    (C || ! C && B && A || ! A && C && B || ! C && ! B && ! A) && A]]
```

(7.9)

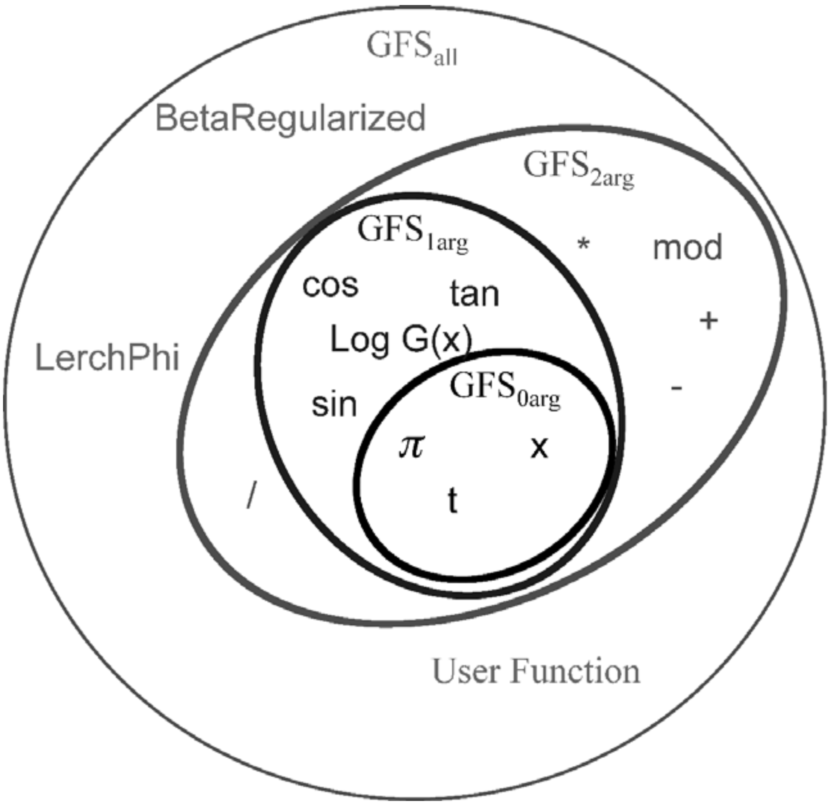

Fig. 7.5. Principle of the general functional set

```
TreeForm[IfFoodAhead[Move, Prog3[IfFoodAhead[Move, Right],
   Prog2[Right, Prog2[Left, Right]],
   Prog2[IfFoodAhead[Move, Left], Move]]]]
```

(7.10)

The final method described here and used for experiments is called Analytic Programming (AP), which has been compared to GP with very good results (see, for example, [26, 15, 27]) or visit the online univeristy website [http://www.fai.utb.cz/people/zelinka/ap].

The basic principles of AP were developed in 2001 and first published in [24, 25]. AP is also based on the set of functions, operators and terminals, which are usually constants or independent variables alike, for example:

1. functions: sin, tan, tanh, And, Or
2. operators: +, -, *, /, dt,
3. terminals: 2.73, 3.14, t,

All these *mathematical* objects create a set, from which AP tries to synthesize an appropriate solution. Because of the variability of the content of this set, it is called a

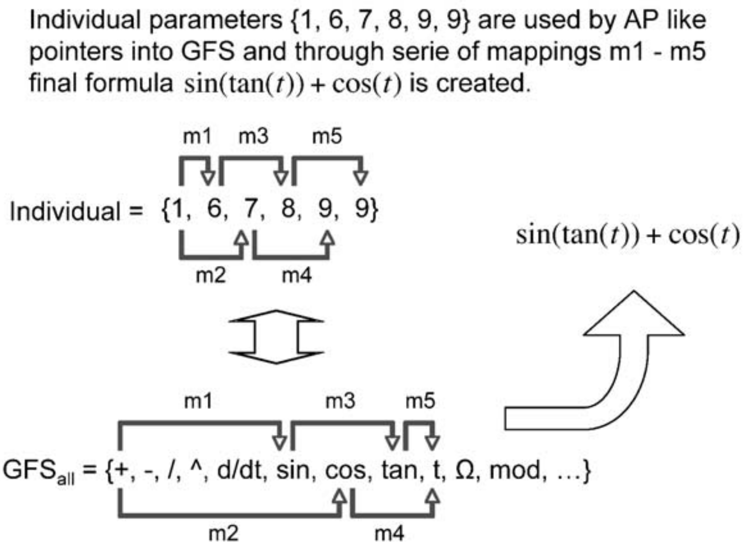
Individual parameters {1, 6, 7, 8, 9, 9} are used by AP like pointers into GFS and through serie of mappings m1 - m5 final formula $\sin(\tan(t)) + \cos(t)$ is created.
m1
m3
m5
Individual = {1, 6, 7, 8, 9, 9}
m2
m4
$\sin(\tan(t)) + \cos(t)$
m1
m3
m5
GFS$_{all}$ = {+, -, /, ^, d/dt, sin, cos, tan, t, Ω, mod, …}
m2
m4

Fig. 7.6. Main principles of AP based on DSH

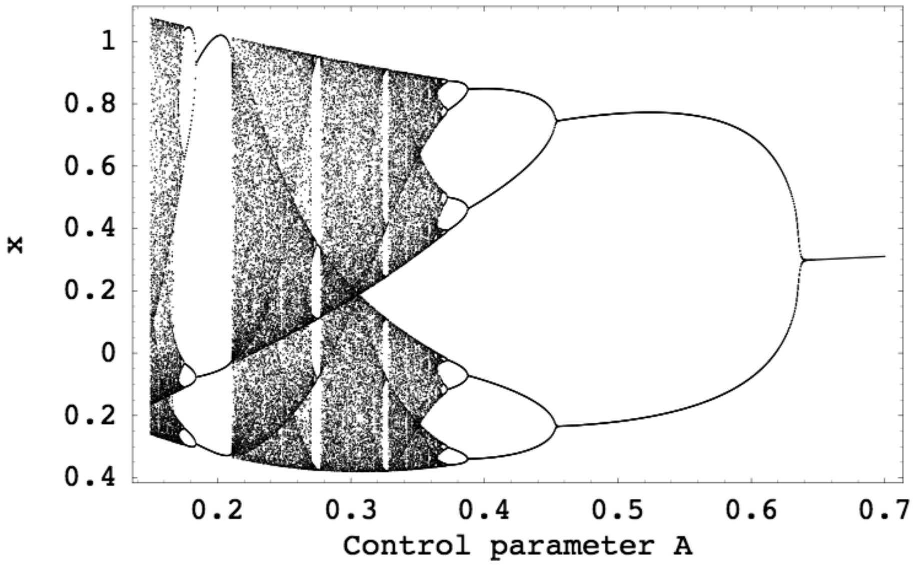

Fig. 7.7. Bifurcation diagram, exhibiting chaos and generated by artificially synthesied equations

general functional set (GFS). The structure of GFS is nested, i.e., it is created by subsets of functions according to the number of their arguments (The content of GFS is dependent only on the user. Various functions and terminals can be mixed together. For example, GFS_{all} is a set of all functions, operators and terminals, GFS_{3arg} is a subset containing functions with maximally three arguments, GFS_{0arg} represents only terminals, etc. (see Fig 7.5).

Table 7.1. Selected solutions synthesized by EA and DSH

Equation	Bifurcations and chaos
$A-x\left(-A+\frac{x\left(-\frac{A}{x}+x+Ax\right)}{A+x^2}\right)$	$\{0.4\}$
$\frac{A\left(2A-2x^2-3x\left(A-x-Ax\right)\right)}{-A+x-x^2}$	$\{0.1,0.13\}$ $\{0.8,1.2\}$
$-x-\frac{1-2A+2x+2A^2x}{1-A+\frac{A^2-x}{x}+x}$	$\{0.3,0.5\}$
$\frac{x-A\left(A-x-2x^2\right)}{-A-x+Ax^2-A\left(-A+\frac{A^3}{x}+2x\right)}$	$\{0.4,0.5\}$
$\frac{2A\left(-2A+2x\right)}{x+\frac{1+A^2+x}{x}}$	$\{0.4\}$
$\frac{x}{\left(3A+2x\right)\left(-1-A-x+\frac{x(A+2x)}{A^2+x}\right)}$	$\{0.12,0.23\}$ $\{0.3,0.36\}$

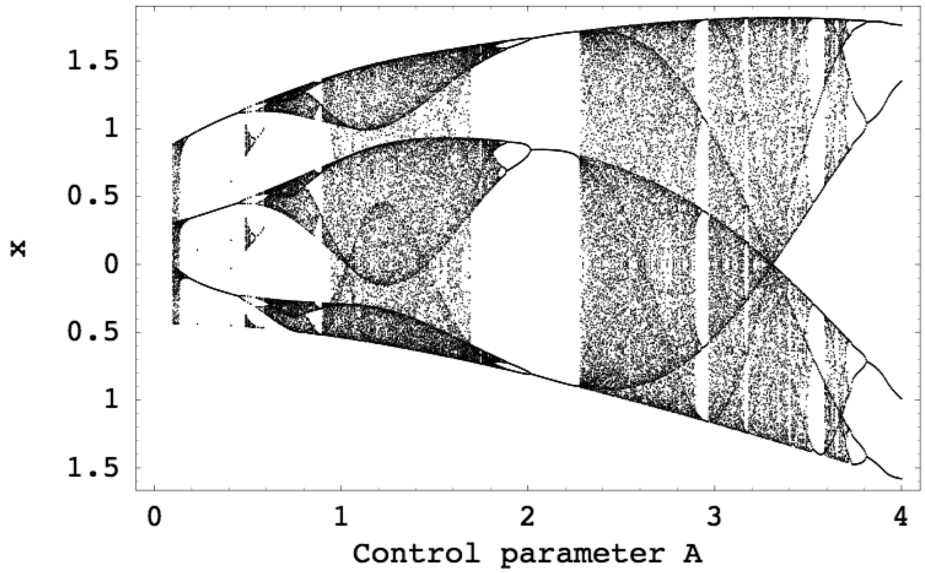

Fig. 7.8. Another bifurcation diagram

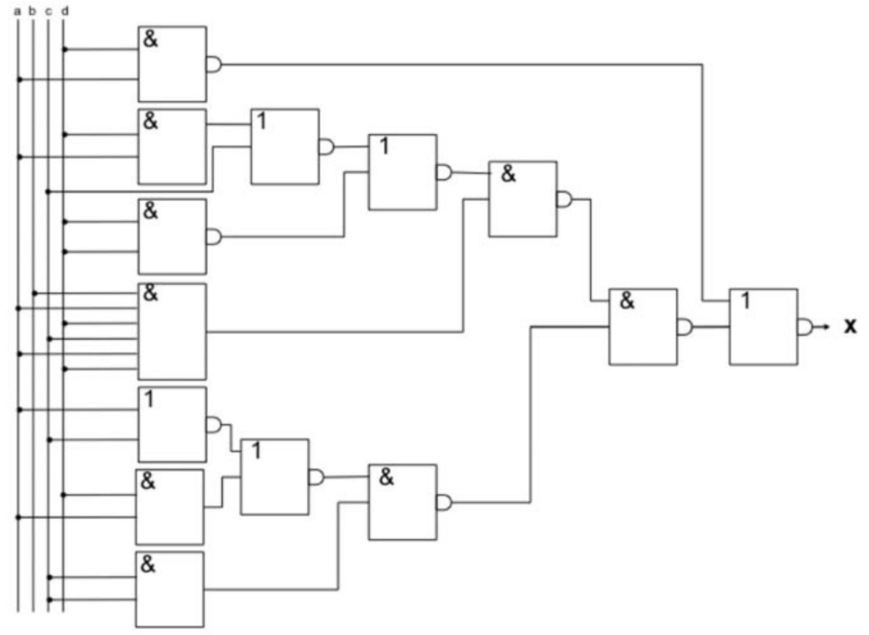

Fig. 7.9. Synthesized logical circuit by means of EA and DSH

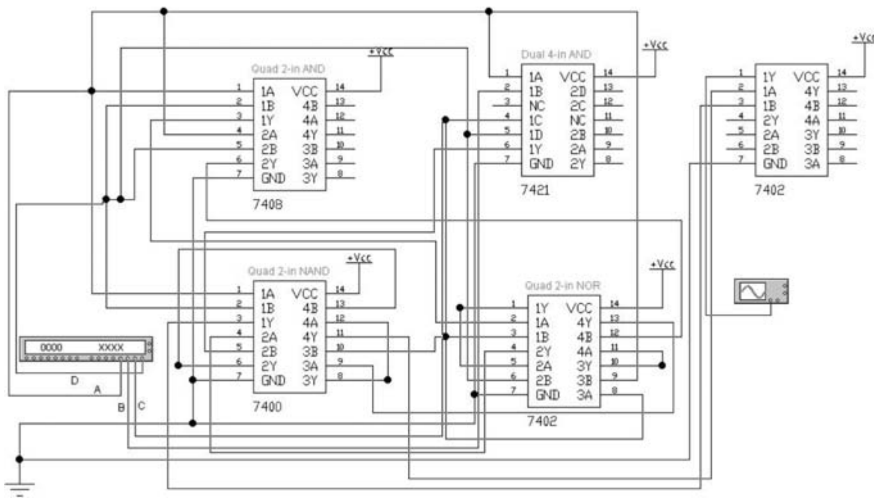

Fig. 7.10. Realization of logical circuit from Equation 7.7

AP, is a mapping from set of individuals into set of posssible programs. Individuals in population and used by AP consist of non-numerical expressions (operators, func tions, .), as described above, which are in the evolutionary process represented by their integer position indexes (Fig 7.6). This index then serves as a pointer into the set of expressions and AP uses it to synthesize the resulting function-program for cost function evaluation.

AP was evaluated in three versions. These three versions utilize for program synthesis the same set of functions, terminals, etc., as in GP [9, 10]. The second version labelled as AP_{meta} (the first version, AP_{basic}) is modified in the sense of constant estimation. For example, the so-called sextic problem was used in [9] to randomly generate constants, whereas AP here uses only one, called K, which is inserted into Equation 7.11 below at various places by the evolutionary process. When a program is synthesized, all Ks are indexed as K_1, K_2,. , K_n to obtain Equation 7.12 in the formula, and then all K_n are estimated by using a second evolutionary algorithm, the result of which can be, for example, Equation 7.13. Because EA (slave) *works under* EA (master), i.e., $EA_{master} \rightarrow$ program $\rightarrow$ K indexing $\rightarrow EA_{slave} \rightarrow$ estimation of K_n, this version is called AP with metaevolution, denoted as AP_{meta}.

$$\frac{x^2 + K}{\pi^K} \tag{7.11}$$

$$\frac{x^2 + K_1}{\pi^{K_2}} \tag{7.12}$$

$$\frac{x^2 + 3.56}{\pi^{-229}} \tag{7.13}$$

Because this version is quite time-consuming, AP_{meta} was further modified to the third version, which differs from the second one in the estimation of K. This is accomplished by using a suitable method for nonlinear fitting (denoted AP_{nf}). This method has shown the most promising performance when unknown constants are present. Results of some comparative simulations can be found in [25, 26, 27].

7.4 Differential Evolution in Mathematica Code

Differential Evolution used in all experiments reported in this chapter has been based on the Mathematica Programming environment. The aim of this part is to describe the structure of the DE code and final code development. Source codes reported here are only a part of fully developed notebook in environment Mathematica. Only the main ideas and some parts of the final code are described here.

For the beginning of DE code development, it is important to prepare the population and set all DE algorithm parameters like F, CR, NP and Generation. Population is initialized as shown in Fig 7.11.

```
In[25]:= Population = DoPopulation[NP, Specimen]

Out[25]= {{534.695, {-442.422, -188.47}}, {191.21, {194.845, -437.931}},
          {-70.135, {-127.976, 99.3825}}, {-208.07, {214.324, 244.138}},
          {-41.6243, {-236.027, -128.204}}, {161.461, {335.943, 355.91}},
          {106.047, {-317.752, -402.607}}, {-157.266, {-119.503, 163.852}},
          {464.407, {507.525, 502.251}}, {-62.8734, {160.401, -149.99}}}
```

Fig. 7.11. Population initialization

```
In[26]:= Table[Random[Integer, {1, NP}], {i, 3}]

Out[26]= {3, 6, 8}
```

Fig. 7.12. Random values

```
SelectOther[active_] := Module[{},
  rand = {0, 0, 0};
  While[rand[[1]] == rand[[2]] || rand[[1]] == rand[[3]] || rand[[2]]
     == rand[[3]] || active == rand[[1]] || active == rand[[2]] ||
    active == rand[[3]], rand = Table[Random[Integer, {1, NP}], {i, 3}]];
  Return[rand]
 ]
```

Fig. 7.13. SelectOther function

```
SelectOther[active_] := Module[{rand = {0, 0, 0, 0, 0}, allvals},
  While[allvals = Append[rand, active];
   Length[allvals] != Length[Union[allvals]],
   rand = Table[Random[Integer, {1, NP}], {i, 5}]];
  Return[rand]]
```

Fig. 7.14. SelectOther compressed function

```
In[29]:= SelectOther[1]

Out[29]= {2, 8, 3}
```

Fig. 7.15. Three random indexes

This command returns the initial population of individuals with the structure Cost-Value, parameter1, parameter2, . . , parameterNP. *NP* is a size of the population. Canonical version of the DE is based on the selection of the three (or more based on DE version) randomly chosen individuals from the population. Random selection, or more precisely, random selection of three pointers, can be done by command *Table* in Fig 7.12.

The random values selects pointers to three individuals of *NP*. To avoid the possibility that two or more will be the same, *SelectOther* function is used. Its argument *active* is a pointer to the actively selected individual – parent. *SelectOther* function is shown in Fig 7.13.

SelectOther function can also be given in a compressed form as in Fig 7.14.

Counters $\{1, 2, .., NP\}$ and $\{i, 3\}$ are used for selection of three different individuals from NP individuals. Fig 7.15 shows three individuals selected from the first solution (parent). Note that all these individuals differ from the first one (position 3).

Till this point, the initial population has been initialized and three individuals have been randomly selected from the population. In the following step, the function *SelectOther* is applied to the entire population at once. Mathematica language allows *parallel–like* programming, which is visible throughout of the code. This is also the case of the following command of function *MapIndexed* which is used to apply *SelectOther* on all individuals, so that the *virtual* population of pointers (randomly selected

```
In[30]:= TRVIndex = MapIndexed[SelectOther[#2[[1]]] &, Population]

Out[30]= {{7, 9, 8}, {5, 1, 7}, {6, 2, 10}, {10, 7, 6},
          {10, 6, 4}, {8, 7, 5}, {1, 2, 4}, {1, 7, 9}, {5, 3, 2}, {2, 9, 8}}
```

Fig. 7.16. TVR Index of pointers

```
In[31]:= TRV = Population[[#1]] & /@ TRVIndex

Out[31]= {{{106.047, {-317.752, -402.607}}, {464.407, {507.525, 502.251}},
            {-157.266, {-119.503, 163.852}}}, {{-41.6243, {-236.027, -128.204}},
            {534.695, {-442.422, -188.47}}, {106.047, {-317.752, -402.607}}},
          {{161.461, {335.943, 355.91}}, {191.21, {194.845, -437.931}},
            {-62.8734, {160.401, -149.99}}}, {{-62.8734, {160.401, -149.99}},
            {106.047, {-317.752, -402.607}}, {161.461, {335.943, 355.91}}},
          {{-62.8734, {160.401, -149.99}}, {161.461, {335.943, 355.91}},
            {-208.07, {214.324, 244.138}}}, {{-157.266, {-119.503, 163.852}},
            {106.047, {-317.752, -402.607}}, {-41.6243, {-236.027, -128.204}}},
          {{534.695, {-442.422, -188.47}}, {191.21, {194.845, -437.931}},
            {-208.07, {214.324, 244.138}}}, {{534.695, {-442.422, -188.47}},
            {106.047, {-317.752, -402.607}}, {464.407, {507.525, 502.251}}},
          {{-41.6243, {-236.027, -128.204}}, {-70.135, {-127.976, 99.3825}},
            {191.21, {194.845, -437.931}}}, {{191.21, {194.845, -437.931}},
            {464.407, {507.525, 502.251}}, {-157.266, {-119.503, 163.852}}}}
```

Fig. 7.17. TVR Index of pointers for entire population

triplets) is created. Fig 7.16 shows the varible *TRVIndex* (trial vector index) created from the *MapIndexed* function.

To unfold the code, the operator /@ (function *Map*) is used, which takes all the arguments of pointers shown in Fig 7.16 and creates an entire array of pointers for the population, given in Fig 7.17.

The result of Fig 7.17 is a list of physically selected individuals (three for each parent). Application of mutation principle and all DE arithmetic operations on TRV list is straight forward. It is accomplished by the means of the operator /@ which in this case applies the arithmetic operation from the left to the elements of the TRV list. Entity #1[[X,2]] in the arithmetic formula F∗(#1[[1,2]] − #1[[2,2]]) +#1[[3,2]] represents X_{th} individual from the selected triplets in TRV. The *Noisy* vector is thus calculated like in Fig 7.18.

```
Noisy = F * (#1[[1, 2]] - #1[[2, 2]]) + #1[[3, 2]] & /@ TRV

{{-779.725, -560.034}, {-152.636, -354.393}, {273.28, 485.082},
 {718.466, 558.003}, {73.8901, -160.582}, {-77.4278, 324.963},
 {-295.49, 443.706}, {407.789, 673.56}, {108.404, -620.}, {-369.647, -588.294}}
```

Fig. 7.18. Noisy Vector

The output of Fig 7.18 is a set of *Noisy* vectors (cardinality of NP), which is consequently used to generate trial vectors − individuals. Parameter selection from the parent or noisy vector is done by the condition If[Cr < Random[]...]. *Flatten* is only a cosmetic command which removes redundant brackets, generated by the command *Table*. In the standard programming approach the command *For* would be used. To

```
Trial = Flatten[Table[If[Cr < Random[],
          Population[[i, 2, j]], #1[[i, j]]],
             {i, NP}, {j, Dim}] & /@ {Noisy}, 1]

{{-442.422, -560.034}, {194.845, -354.393}, {-127.976, 99.3825},
 {718.466, 244.138}, {-236.027, -128.204}, {335.943, 355.91},
 {-317.752, -402.607}, {-119.503, 163.852}, {507.525, 502.251}, {160.401, -149.99}}
```

Fig. 7.19. Trial Vector

```
BoundaryChecking = Flatten[MapIndexed
         [CheckInterval[#1, #2] &, #1]
            , 1] & /@ (Trial)

{{-442.422, -2.47524}, {194.845, -354.393}, {-127.976, 99.3825},
 {300.954, 244.138}, {-236.027, -128.204}, {335.943, 355.91},
 {-317.752, -402.607}, {-119.503, 163.852}, {507.525, 502.251}, {160.401, -149.99}}
```

Fig. 7.20. Boundary Checking

```
In[44]:= IndividualsCostValue = {CostFunction[#1], #1} & /@ (BoundaryChecking)

Out[44]= {{364.224, {-442.422, -2.47524}}, {-200.331, {194.845, -354.393}},
          {-70.135, {-127.976, 99.3825}}, {279.978, {300.954, 244.138}},
          {-41.6243, {-236.027, -128.204}}, {161.461, {335.943, 355.91}},
          {106.047, {-317.752, -402.607}}, {-157.266, {-119.503, 163.852}},
          {464.407, {507.525, 502.251}}, {-62.8734, {160.401, -149.99}}}
```

Fig. 7.21. Individual Cost Value

```
NewPopulation = MapThread[If[#1[[1]] < #2[[1]], #1, #2] &,
            {Population, (IndividualsCostValue)}]

{{364.224, {-442.422, -2.47524}}, {-200.331, {194.845, -354.393}},
 {-70.135, {-127.976, 99.3825}}, {-208.07, {214.324, 244.138}},
 {-41.6243, {-236.027, -128.204}}, {161.461, {335.943, 355.91}},
 {106.047, {-317.752, -402.607}}, {-157.266, {-119.503, 163.852}},
 {464.407, {507.525, 502.251}}, {-62.8734, {160.401, -149.99}}}
```

Fig. 7.22. Next Population Selection

avoid setting of local or global variables for the trial vector list, the *Table* command is used instead of *For*. *Trial* vectors are returned in the list given in Fig 7.19, which is created automatically.

All the *Trial* vectors are created at once. Before the fitness is calculated, the population of the trial individuals are checked for boundary conditions. If some parameter is out of the allowed boundary, then it is *randomly* returned back. The function is given in Fig 7.20.

Now, there exists a repaired set of trial vectors, which is evaluated by the cost function. It is done by the function *CostFunction* applied by /@ on the *BoundaryChecking* set. Note that the body of each individual in Fig 7.21 is enlarged by the function *IndividualsCostValue*.

The better individual of both *parent* and *child* is selected into the new population by means of the *MapThread* function.

```
NewPop[Pop_] := Module[{},
  TRVIndex = MapIndexed[SelectOther[#2[[1]]] &, Population];
  TRV = Population[[#1]] & /@ TRVIndex;
  Noisy = F * (#1[[1, 2]] - #1[[2, 2]]) + #1[[3, 2]] & /@ TRV;
  Trial = Flatten[Table[If[Cr < Random[], Pop[[i, 2, j]], #1[[i, j]]],
          {i, NP}, {j, Dim}] & /@ {Noisy}, 1];
  BoundaryChecking = Flatten[MapIndexed[CheckInterval[#1, #2] &, #1], 1]
         & /@ (Trial);
  IndividualsCostValue = {CostFunction[#1], #1} & /@ (BoundaryChecking);
  NewPopulation = MapThread[If[#1[[1]] < #2[[1]], #1, #2] &,
        {Population, (IndividualsCostValue)}]
 ]
```

Fig. 7.23. Compiled DE crossover code

```
In[47]:= np = NewPop[Population];
         MatrixForm[np]

Out[48]//MatrixForm=
 -1.65327        {5, 11.2}
  86.0232   {-297.175, -437.931}
  -70.135    {-127.976, 99.3825}
  -208.07     {214.324, 244.138}
 -41.6243   {-236.027, -128.204}
  161.461      {335.943, 355.91}
 -341.433   {-297.175, -150.458}
 -216.653     {-499.82, 163.852}
  464.407     {507.525, 502.251}
 -62.8734     {160.401, -149.99}
```

Fig. 7.24. Function call of New population

If all the preceding steps are joined together, then final DE code in Mathematica is given in Fig 7.23.

When the function *NewPop* in Fig 7.23 is called with the variable *Population* like an argument the new population is created as shown in Fig 7.24.

When the output of *NewPop* in Fig 7.24 is repeatedly used as an input in some loop procedure (one loop – one generation), the *DE* algorithm is iterated.

Some additive procedures can also be used, like selection of the best individual from the population. An example is given in Fig 7.25.

A more compressed (but less readable) and similar version of DE is shown in Fig 7.26.

Canonical version of the DE described is a priori suitable for the real valued variables. However, due to the problems being solved here are based on integer–valued variables and permutative problems, some additional subroutines have been added to the DE code. The first one is a *Repair* subroutine. An input of this subroutine is an infesible solution and the output is a repaired solution so that each variable only appears once in the solution.The routine is shown in Fig 7.27.

```
ExpForm[nmbr_] := PaddedForm[nmbr, {6, 5}, ExponentFunction → (#1 &),
      NumberFormat → (#1 <> "<E>" <> #3 &), NumberSigns → {"-", "+"}];

BestInd[pop_] := Module[{best, ind, str},
  best = Position[{##}, Min[##]][[1]][[1]] & @@ Transpose[pop][[1]];
  ind = pop[[best]];
  str = "Best individual is on position " <> ToString[best] <>
    " with cost value " <> ToString[ExpForm[ind[[1]]]] <> " and
      parameters " <> ToString[ind[[2]]];
  Print[str];
  Return[Flatten[{best, ind}, 1]]
 ]

BestInd[np]
```

```
Best individual is on position 7 with cost
    value -3.41433<E>2 and parameters {-297.175, -150.458}

{7, -341.433, {-297.175, -150.458}}
```

Fig. 7.25. Best individual from population

```
NewPop[Pop_] := MapThread[If[#1[[1]] < #2[[1]], #1, #2] &,
  {Pop, ({CostFunction[#1], #1} & /@ (Flatten
       [MapIndexed[CheckInterval[#1, #2] &, #1], 1] & /@
     (Flatten[Table[If[Cr < Random[], Pop[[i, 2, j]],
          #1[[i, j]]], {i, NP}, {j, Dim}] & /@ {F * (#1[[1, 2]]
              - #1[[2, 2]]) + #1[[3, 2]] & /@ (Pop[[#1]] & /@ MapIndexed
            [SelectOther[#2[[1]]] &, Pop])}, 1])))}]
```

Fig. 7.26. Compressed DE form

```
Repair[Sol_] := Module[{Temp, MissingValue, Solution, Pos, Size},
  Solution = Sol; Size = Length[Solution];
  MissingValue = RandomRelist[Complement[Range[Size], Solution]];
  Pos = Position[Solution, #] & /@ Range[Size];
  (Solution = Drop[Solution, {#}]) & /@ (Pos = Sort[Flatten[MapIndexed
        [Drop[#1, 1] &, #1] &[MapIndexed[RandomRelist[#1] &, #] &[
       Join[Temp[[#]]] & /@ (#1[[# & /@ Range[Length[#1]]]]) &[Flatten[
         Position[Flatten[Dimensions /@ (Temp = Flatten[Pos[[#]]] & /@
              Range[Size])], _? (1 < # &)]]]]]], Greater]);
  Pos = Sort[Pos, Less];
  MapThread[(Solution = Insert[Solution, #1, #2]) &, {MissingValue, Pos}]
   Return[Solution]
 ]
```

Fig. 7.27. Repair routine

The *Repair* function is broken down and explained in-depth. The initial process is to find all the *missing values* in the solution. Since this is a *permutative* solution, each value is exist only *once* in the solution. Therefore it stands to reason that if there are more than one single value in the solution, then some values will be missing.

The function:
MissingValue = RandomRelist[Complement[Range[Size], Solution]]; finds the missing values in the solution.

The second phase is to map all the values in the solution. The routine:
Pos = Position[Solution, #]&/@Range[Size]; maps the *occurrence* of each value in the solution.

The *repetitive* values are identified in the function:
Flatten[Position[Flatten[Dimensions/@
  (Temp = Flatten[Pos[[#]]]&/@
    Range[Size])], _?(1 < #&)]]

The routine: Join[Temp[[#]]]&/@(#1[[#&/@Range[Length[#1]]]])& calculates the positions of the *replicated* values in the solution.

These replicated positions are *randomly shuffled*, since the objective is not to create any bias to replacement. This routine is given in the function:
MapIndexed[RandomRelist[#1]&, #]&

The variable *Pos* isolates the positions of replicated values which will be replaced as given in:
Pos = Sort[Flatten[MapIndexed[Drop[#1, 1]&, #1]&

The routine: Drop[Solution, {#}]&/@ removes the replicated values from the solution.

The final routine:
MapThread[(Solution = Insert[Solution, #1, #2])&, {MissingValue, Pos}]; inserts the missing values from the array *Missing Value* into randomly allocated indexs identified by variable *Pos*.

DE is consequently modified so that before the function *CostFunction* a *Repair/@DSH* function is used as in Fig 7.28.

```
DERand1Bin[Pop_] := MapThread[If[#1[[1]] < #2[[1]], #1, #2] &,
  {Pop, ({CostFunction[#1, Prob, Mach], #1} & /@
     (Repair /@ DSH[Flatten[MapIndexed[CheckInterval[#1, #2] &,
           #1], 1] & /@ (Flatten[Table[If[Cr < Random[], Pop[[i, 2, j]],
              #1[[i, j]]], {i, NP}, {j, Dim}] & /@
           {F * (#1[[1, 2]] - #1[[2, 2]]) + #1[[3, 2]] & /@ (Pop[[#1]] & /@ MapIndexed[
                SelectOtherRand1Bin[#2[[1]]] &, Pop])}, 1])
       ]
     ))}]
```

Fig. 7.28. Repair DSH routine

```
In[86]:=  DS = {M1, M2, M3, M4, M5, M6, M7, M8, M9, M10}

Out[86]=  {M1, M2, M3, M4, M5, M6, M7, M8, M9, M10}
```

Fig. 7.29. Discrete Set

```
In[87]:=  DSH[Pop_] := Module[{},
            RoundPop = Round[Pop];
            DS[[#1]] & /@ #1 & /@ RoundPop
          ]
```

Fig. 7.30. Discrete Set

```
In[88]:= DSH[BoundaryChecking] // MatrixForm

Out[88]//MatrixForm=
         ⎛M4  M6  M3  M8  M3  M10  M7  M1   M4   M5⎞
         ⎜M2  M4  M8  M5  M5  M2   M7  M10  M10  M4⎟
         ⎜M8  M1  M2  M1  M6  M7   M9  M3   M3   M7⎟
         ⎜M7  M3  M9  M3  M6  M9   M2  M7   M8   M9⎟
         ⎜M4  M2  M8  M5  M7  M1   M5  M1   M2   M4⎟
         ⎜M6  M2  M9  M4  M1  M9   M2  M6   M6   M5⎟
         ⎜M9  M6  M7  M4  M6  M5   M9  M9   M10  M5⎟
         ⎜M3  M5  M9  M2  M7  M4   M9  M4   M10  M5⎟
         ⎜M9  M8  M6  M4  M9  M3   M5  M1   M4   M8⎟
         ⎝M2  M2  M4  M4  M9  M7   M6  M4   M5   M4⎠
```

Fig. 7.31. Discrete Set Output

A discrete set can be created as shown in Fig 7.29.

The DSH function is given in Fig 7.30.

The result of applying the DSH set on the population is given in Fig 7.31.

Such or similar set can be used in other different methods (if needed) like fuzzy logic etc. Due to the nature of permutative problems, (sequence has to be complete and unique), the discrete set been set to the same sequence of numbers.

Due to the complex nature of permutative problems, a *Local Search* routine has been added to the heuristic. Local search is used to search in the *neighbourhood* of the current solutions. Keeping in mind the computational nature of the code, a 2 OPT local search outine was selected as in Fig 7.32.

```
LocalSearch[Sol_] :=
 Module[{Solution, NewSolution, CostVal, NewCostVal, Temp},
  CostVal = Sol[[1]]; Solution = Sol[[2]]; NewCostVal = CostVal;
      NewSolution = Solution;
  Label[start]; CostVal = NewCostVal; NewSolution = Solution;
  Do[
   Temp = Solution[[i]]; Solution[[i]] = Solution[[j]];
       Solution[[j]] = Temp;
   NewCostVal = CostFunction[Solution, Prob, Mach];
   If[NewCostVal < CostVal, Goto[start]],
       {i, Job - 1}, {j, i + 1, Job}];
  Solution = {CostVal, NewSolution}; Return[Solution]
 ]
```

Fig. 7.32. Local Search routine

The current fitness of the solution is kept in the variable *CostVal*, and the current active solution is kept in *Solution*. The start flag is *Label*[*start*].

Two *iterators* are activated, *i*, which is the index to the current variable in the solution and *j*, which is the iterator from the current position indexed by *i* till the end of the solution given as $\{j, i+1, \text{Job}\}$.

Each two values in the solution are taken pairwise and exchanged as $\mathrm{Temp} = \mathrm{Solution}[[i]]; \mathrm{Solution}[[i]] = \mathrm{Solution}[[j]]; \mathrm{Solution}[[j]] = \mathrm{Temp}$, where *Temp* is the intermediary placeholder. Another syntax for this process can be given as $\{\mathrm{Solution}[[i]], \mathrm{Solution}[[j]]\} = \{\mathrm{Solution}[[j]], \mathrm{Solution}[[i]]\}$. Each value indexed by i and j are exchanged.

The new fitness of the solution is calculated. If the new fitness is better than the old value, then the new solution is admitted into the population and the starting position is again set to $Label[start]$ given as $\mathrm{If}[\mathrm{NewCostVal} < \mathrm{CostVal}, \mathrm{Goto}[\mathrm{start}]]$. This process iterates till the index i iterates to the end of the solution $\{\mathrm{i}, \mathrm{Job}-1\}$ taking into account all the resets done by the finding of new solutions.

The outline of the entire code is given in Fig 7.33 and the data flow diagram is given in Fig 7.34.

```
1.Input : $D, G_{\max}, NP \geq 4, F \in (0,1+), CR \in [0,1]$, initial bounds : $\mathbf{x}^{(lo)}, \mathbf{x}^{(hi)}$.
2.Initialize : $DoPopulation[NP, Specimen]$
3.While  $G < G_{\max}$
    $\forall i \leq NP$
        **Create TRVIndex by command** :
            $TRVIndex = MapIndexed[SelectOther[\#2[[1]]]\&, Population]$
        **Selection of three vectors by TRVIndex**
            $TRV = Population[[\#1]]\&/@TRVIndex$
        **Create noisy vectors**
            $Noisy = F*(\#1[[1,2]] - \#1[[2,2]]) + \#1[[3,2]]\&/@TRV$
        **Create trial vectors** :
            $Trial = Flatten[Table[If[Cr < Random[], Pop[[i,2,j]], \#1[[i,j]]],$
                $\{i, NP\}, \{j, Dim\}]\&/@\{Noisy\}, 1]$
        **Check for boundary** :
            $BoundaryChecking = Flatten[MapIndexed[CheckInterval[\#1, \#2]$
                $\&, \#1], 1]\&/@(Trial)$
        **Cost value**
            $IndividualsCostValue = \{CostFunction[\#1], \#1\}\&/@(BoundaryChecking)$
        **DSH conversion** :
            $Repair/@DSH$
        **New population** :
            $NewPopulation = MapThread[If[\#1[[1]] < \#2[[1]], \#1, \#2]\&,$
                $\{Population, (IndividualsCostValue)\}]$
    $G = G + 1$
```

Fig. 7.33. DE outline

7.4.1 DE Flow Shop Scheduling

This section describes the application of Flow Shop scheduling as given in Fig 7.35. In this function, the obtained solution is simply passed into the *CostFunction* function.

The first variable, *JTime* accumulates the processing time of all the jobs in the first machine given as: $\mathrm{JTime} = \mathrm{Accumulate}[\#]\&[\mathrm{Prob}[[1,\#]]\&/@\mathrm{Solution}];$.

The second variable, *LMach*, computes the job times on all the subsequent machines iteratively. Since the maximum of the processing times is taken between the jobs:

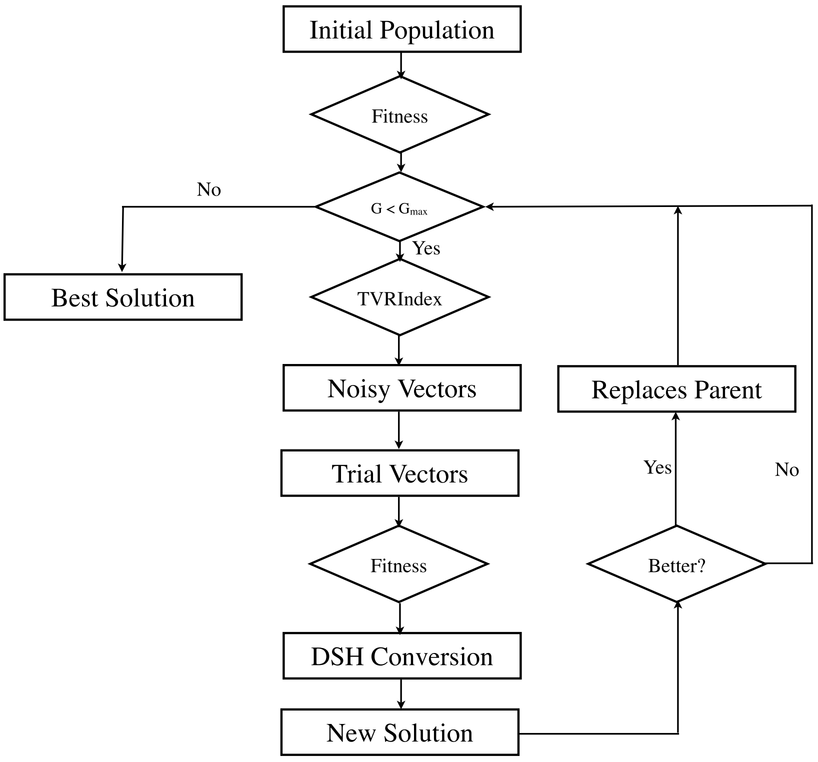

Fig. 7.34. Data flow diagram of DE

```
CostFunction = Compile[{{Solution, _Integer, 1}, {Prob, _Integer, 2}, {Mach, _Integer}},
  Module[{JTime, LMach},
   JTime = Accumulate[#] &[Prob[[1, #]] & /@ Solution];
   LMach = Accumulate[#] &[Prob[[#, Solution[[1]]]] & /@ Range[Mach]];
       Table[JTime[[1]] = LMach[[i + 1]];
    MapIndexed[(JTime[[First[#2] + 1]] = Max[JTime[[First[#2]]], JTime[[First[#2] + 1]]] +
         Prob[[i + 1, #1]]) &, Rest[Solution]], {i, Mach - 1}]; Return[JTime[[-1]]]
  ]]
```

Fig. 7.35. Flow Shop Schedluing routine

Max[JTime[[First[#2]]], JTime[[First[#2] + 1]]] + Prob[[i + 1, #1]])&, Rest[Solution]], the processing time value is simply accumulated in *LMach*.

For more information about Flow Shop, please see [17].

7.4.2 DE Traveling Salesman Problem

The Traveling salesman function is simply the accumulation of the distances from one city to the next. The function is given in Fig 7.36.

The first routine simply picks up the times between the cities in the *Solution*. The distance times are stored in the matrix *Distance*.
Time+ = (Distance[[Solution[[# + 1]], Solution[[#]]]])&/@Range[Size − 1];.

```
CostFunction = Compile[{{Solution, _Integer, 1}, {Distance, _Real, 2},
   {Size, _Integer}},
  Module[{Time = 0.0},
   Time += (Distance[[Solution[[# + 1]], Solution[[#]]]]) & /@ Range[Size - 1];
   Time += (Distance[[Solution[[1]], Solution[[Size]]]]); Return[Time]]]
```

Fig. 7.36. Traveling Salesman routine

Once all the related city distances have been added, the distance from the *last* city to the *first* city is added to complete the *tour*, given as:
$\text{Time}+=(\text{Distance}[[\text{Solution}[[1]],\text{Solution}[[\text{Size}]]]])$

7.5 DE Example

The simplest approach of explaining the application of discrete set handling is to implement a worked example. In that respect, a TSP problem is proposed with only five cities, in order to make it more viable.

Assume a symmetric TSP problem given as in Table 7.2. Symmetric implies that the distances between the two cities are equal both ways of travelling.

Table 7.2. Symmetric TSP problem

Cities	A	B	C	D	E
A	0	5	10	14	24
B	5	0	5	9	19
C	10	5	0	10	14
D	14	9	10	0	10
E	24	19	14	10	0

Table 7.3. Decomposed symmetric TSP problem

Cities	A	B	C	D	E
A	**0**				
B	5	**0**			
C	10	5	**0**		
D	14	9	10	**0**	
E	24	19	14	10	**0**

Since this is a symmetric TSP problem, the *Distance Matrix* can be decomposed to the leading triangle as given in Table 7.3.

In order to use DE, some operational parameters are required, in this case the tuning parameters of *CR* and *F*, and well as the size of the population *NP* and the number of generations *Gen*. For the purpose of this example, the population is specified as 10 individuals.

7.5.1 Initialization

The first phase is the initialization of the population. Since NP has been arbitrarily set as 10, ten random permutative solutions are generated to fill the initial population as given in Table 7.4.

Table 7.4. Initial population

Solution	City 1	City 2	City 3	City 4	City 5
1	A	D	B	E	C
2	D	B	A	C	E
3	C	A	E	B	D
4	E	C	D	A	B
5	E	B	C	D	A
6	B	D	A	E	C
7	A	D	C	E	B
8	E	C	A	D	B
9	B	E	C	A	D
10	A	C	E	B	D

7.5.2 DSH Conversion

The second part is to create the discrete set for the solution. DSH assigns a raw number for each position index in the solution. In this case the most logical phase is to assign consecutive numbers for the consecutive alphabets as shown in Table 7.5.

The problem assignment now switches to the discrete set. This is given in Table 7.6.

Table 7.5. Discrete set for the cities

Cities	A	B	C	D	E
Discrete Set	1	2	3	4	5

Table 7.6. Initial Population

Solution	City 1	City 2	City 3	City 4	City 5
1	1	4	2	5	3
2	4	2	1	3	5
3	3	1	5	2	4
4	5	3	4	1	2
5	5	2	3	4	1
6	2	4	1	5	3
7	1	4	3	5	2
8	5	3	1	4	2
9	2	5	3	1	4
10	1	3	5	2	4

Table 7.7. Distance matrix for Tour 1

Cities	A	D	B	E	C
A					
D	**14**				
B	5	**9**			
E	24	9	**19**		
C	**10**	4	5	**14**	

Table 7.8. Fitness for the population

Solution	City 1	City 2	City 3	City 4	City 5	Fitness
1	1	4	2	5	3	66
2	4	2	1	3	5	48
3	3	1	5	2	4	48
4	5	3	4	1	2	62
5	5	2	3	4	1	72
6	2	4	1	5	3	66
7	1	4	3	5	2	62
8	5	3	1	4	2	66
9	2	5	3	1	4	66
10	1	3	5	2	4	66

7.5.3 Fitness Evaluation

The objective function for TSP is the cumulative distance between the cities, ending and starting from the same city. Taking the example of the first solution in Table 7.6; now termed Tour 1 $= \{A,D,B,E,C\}$, the equivalent representation is Tour 1 = $= \{1,4,2,5,3\}$. The distance matrix can now be represented as in Table 7.7.

Since the tour is cyclic, the tour can further be completely represented as Tour 1 = $\{A \to D \to B \to E \to C \to A\}$. From distance matrix it is now the accumulation of the tour distances Tour 1 = 14 + 9 + 19 + 14 + 10, which gives a total of 66.

Likewise, the total tour for all the solutions is calculated and is presented in Table 7.8.

7.5.4 DE Application

The next step is the application of DE to each solution in the population. For this example the DE **Rand1Bin** strategy is selected. At this point it is important to set the DE scaling factor F. It can be given a value of 0.4.

DE application is simple. Starting from the first solution, each solution is evolved sequentially. Evolution in DE consists of a number of steps. The first step is to *randomly* select two other solutions from the population, which are unique from the solution currently under evolution. If we take the assumption that Solution 1 is currently under evolution, then we can randomly select Solution 4 and Solution 7 for example. These make the batch of *parent* solutions as given in Table 7.9.

Table 7.9. Parent solutions

Solution	City 1	City 2	City 3	City 4	City 5	Fitness
1	1	4	2	5	3	66
4	5	3	4	1	2	62
7	1	4	3	5	2	62

Table 7.10. Parent solutions crossover

Solution	City 1	City 2	City 3	City 4	City 5
1	1	4	2	5	3
4	5	3	4	1	2
7	1	4	3	5	2
Index	4	5	1	2	3

The second DE operating parameter crossover *CR* can now be set as 0.4. The starting point of evolution in the solution is randomly selected. In this example, solution index 3 is selected as the first variable for crossover as given in Table 7.10.

The mathematical representation of DE Rand1Bin is given as: $x_{current} + F \bullet \left(x_{random_1} - x_{random_2}\right)$. x_{random_1} in this instance refers to the first randomly selected solution 4, and x_{random_2} is the second random solution 7. Since the starting index has been randomly selected as 3, the linked values for the two solutions are subtracted as 4 − 3 = 1. This value is multiplied by *F*, which is 0.4 The result is (1 x 0.4 = 0.4). This value is added to the current indexed solution 1: (0.4 + 2 = 2.4).

Likewise, applying the equation to the selected parent solutions yields the following values given in Table 7.11:

Table 7.11. Parent solutions final values

Solution	City 1	City 2	City 3	City 4	City 5
1	1	4	2	5	3
4	5	3	4	1	2
7	1	4	3	5	2
Index	4	5	1	2	3
Final	**2.6**	**3.6**	**2.4**	**3.4**	**3**

The second part shown in Table 7.12 is to select which of the new variables in the solutions will actually be accepted in the final child solution. The procedure of this is to randomly generate random numbers between 0 and 1and if these random numbers are greater than the user specified constant *CR*, then these values are accepted in the child solution. Otherwise the current index values are retained.

Table 7.12. CR Application

Solution	City 1	City 2	City 3	City 4	City 5
Parent	1	4	2	5	3
Final	2.6	3.6	2.4	3.4	3
Random value	0.6	0.2	0.5	0.9	0.3

Table 7.13. Child solution

Solution	City 1	City 2	City 3	City 4	City 5
Parent	1	4	2	5	3
Child	2.6	4	2.4	3.4	3

Table 7.14. Closest Integer Approach

Solution	City 1	City 2	City 3	City 4	City 5
Child	2.6	4	2.4	3.4	3
Closest integer	3	4	2	3	3

Table 7.15. Hierarchical Approach

Solution	City 1	City 2	City 3	City 4	City 5
Child	2.6	4	2.4	3.4	3
Hierarchical Approach	2	5	1	4	3

Since *CR* has been set as 0.4, all indexes with random values greater than 0.4 are selected into the child population. The rest of the indexes are filled by the variables from the parent solution as given in Table 7.13.

Two different approaches now can be used in order to realize the *child* solution. The first is to *closest integer approach*. In this approach the integer value closest to the obtained real value is used. This is given as in Table 7.14.

The second approach is the *hierarchical approach*. In this approach, the solutions are listed according to their placement in the solution itself. This is given in Table 7.15.

The advantage of the *hierarchical approach* is that no repairment is needed to the final solution. However, it does not reflect the placements of DE values, and can be misleading. Due to this factor, the first approach of *closest integer* approach is now described.

The next step is to check if any solution exists outside of the bounds. According to [18], all out of bound variables are randomly repaired. If the case of this example all the values are within the bounds specified by the problem.

Table 7.16. Feasible solutions

Solution	City 1	City 2	City 3	City 4	City 5
Child	3	4	2	3	3
Feasible	-	**4**	**2**	-	-

Table 7.17. Final solution

Solution	City 1	City 2	City 3	City 4	City 5
Child	3	4	2	3	3
Final Solution	**3**	**4**	**2**	**1**	**5**

Table 7.18. Final solution fitness

Solution	City 1	City 2	City 3	City 4	City 5	Fitness
Final Solution	3	4	2	1	5	62

Table 7.19. DSH application

Solution	City 1	City 2	City 3	City 4	City 5
Final Solution	3	4	2	1	5
City	**C**	**D**	**B**	**A**	**E**

The final routine is to repair the solution if repetitive solutions exist. It must be stressed that not all the solutions obtained are infeasible.

The approach is to first isolate all the unique solutions as given in Table 7.16.

The missing values in this case are 1, 3, 5. Using random selection, each missing value is replaced in the final solution in Table 7.17.

Random placement is selected since it has proven highly effective [3].

The new solution is vetted for its fitness.

The new fitness of 62 improves the old fitness of the parent solution of 66 and hense the child solution is accepted in the population for the next generation. The correct arrangement is obtained by converting back using DSH into City representation as given in Table 7.19.

Using the above process, all the solutions are evolved from one generation to another. At the termination of the algorithm, the best-placed solution is retrieved.

7.6 Experimentation

All experiments have been done on the *grid cluster* of the XServers (Apple technology). Such a kind of computer technology is now commonly used for hard computing tasks.

Fig. 7.37. 1000 PC cluster 1

An example is the 1000 PCs used in genetic programming (Fig 7.37 – 7.38). In Czech Republic, there also exists such grid computers. An example of a grid configuration is the supercomputer named Amalka with 360 processors used in space research and related problems shown in Fig 7.39.

The grid cluster used for the FSS and TSP experiments, consisted of 16 XServers 2 x 2 GHz Intel Xeon, 1 GB RAM, 80 GB HD (Fig 7.40 – 7.41). Each Xserve contain 4 computational cores, so there are in total 64 computational cores. Part of the computational force has been used for FSS and TSP calculations.

7.6.1 Flow Shop Scheduling Tuning

The main issue for almost all meta-heuristics, which does optimization without knowledge of the system, is that there are parameters to tune in the algorithm. In DE, there are two control parameters, *F* and *CR*. These parameters are required in order to induce the stochastic process in the heuristic, which will enable it to find the optimal solution for that specific problem.

Fig. 7.38. 1000 PC cluster 2

[18] gave a brief outline for the different operating parameters as given in Table 7.20.

These general outlines were formulated after experimentation [18], however they were not intended for permutative problems. Since this is realized as a novel approach for DE, it becomes than imperative to create a experiment procedure for the formulation of these control values. Alongside these control values, these are altogether 7 general operating DE strategies.

1. Rand 1 Bin
2. Rand 2 Bin
3. Best 2 Bin
4. Local to Best
5. Best 1 JIter
6. Rand 1 DIter
7. Rand 1 GenDIter

Fig. 7.39. Amalka Grid

Table 7.20. Operating parameters for original DE

Control Variables	Lo	Hi	Best?	Comments
F : Scaling Factor	0	1.0+	0.3 0.9	$F \geq 0.5$
CR: Crossover probability	0	1	0.8 1.0	CR = 0, seperable
				CR = 1, epistatic

Table 7.21. Tuning Parameters

Strategy	CR	F
Rand1Bin	0.1	0.1
Rand2Bin	0.2	0.2
Best2Bin	0.3	0.3
LocaltoBest	0.4	0.4
Best1JIter	0.5	0.5
Rand1DIter	0.6	0.6
Rand1Gen DIter	0.7	0.7
	0.8	0.8
	0.9	0.9
	1	1

Fig. 7.40. Emanuel Cluster at UTB

Table 7.22. FSS operating parameters

Parameters	Values
Strategy	Rand 1 DIter
F	0.5
CR	0.1

So the task then is to also find the optimal operating strategy alongside the two control variables. This in itself becomes a three phase permutative problem. The sampling rate for the two control variables was kept as small as possible to 0.1.

The permutative outline for the tuning parameter is now given in Table 7.21.

Each value is permutated through the other values, so the total number of tuning experimentation conducted is 7 x 10 x 10 = 700.

Fig. 7.41. Emanuel Cluster at UTB

The second aspect is to select an appropriate test instance. For our purpose, a moderately difficult instance of 50 jobs and 20 machines from the Taillard benchmark problem set was selected.

Experimentation was conducted with Population set to 200 individuals and 100 generations allowed. The solution mesh is given in Fig 7.42.

A histogram projection in Fig 7.43. gives a better representation with the frequency of makespan.

The optimal value obtained through this experimentation is given in Table 7.22.

7.6.2 Traveling Salesman Problem Tuning

The identical tuning procedure used for Flow shop was used for parameter tuning on the Traveling Salesman Problem. Once again, 700 experimentations were conducted, and for this problem set, the moderately difficult Eil51 city problem set was selected.

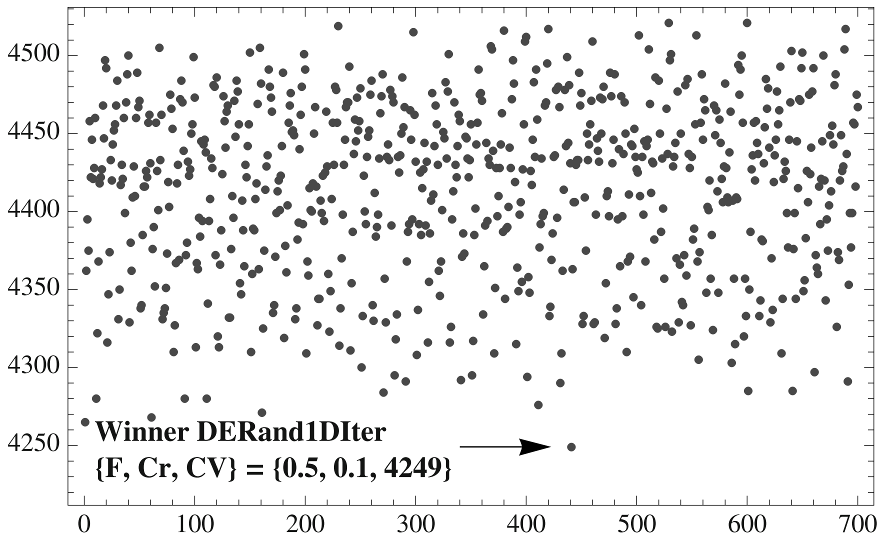

Fig. 7.42. FSS tuning graphical display

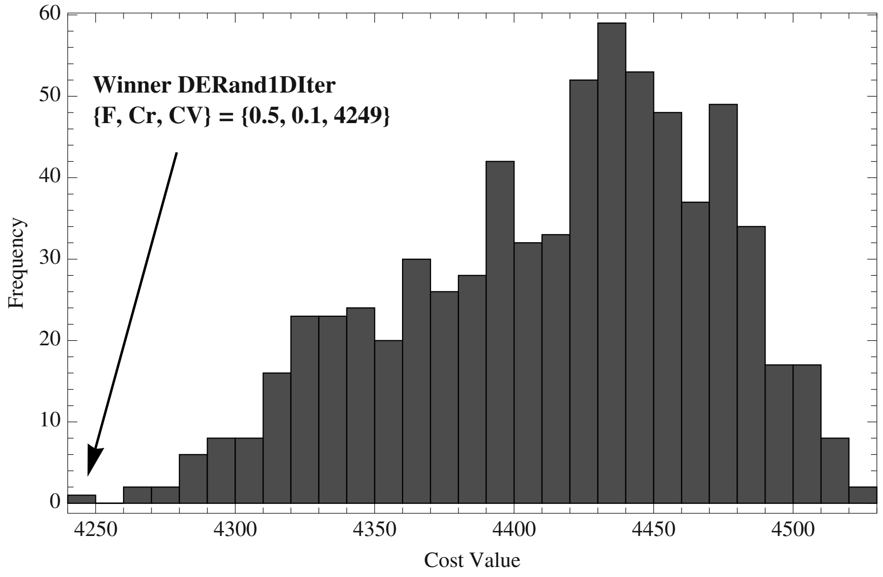

Fig. 7.43. Frequency display for FSS tuning

The solution mesh for TSP is given in Fig 7.44.
The histogram display for all the values is given in Fig 7.45.
The optimal value obtained through this experimentation is given in Table 7.23.

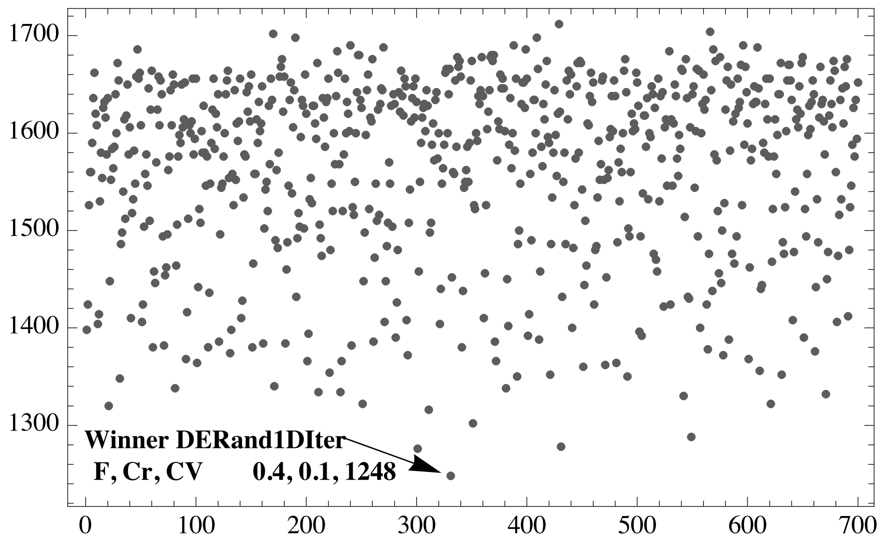

Fig. 7.44. TSP tuning graphical display

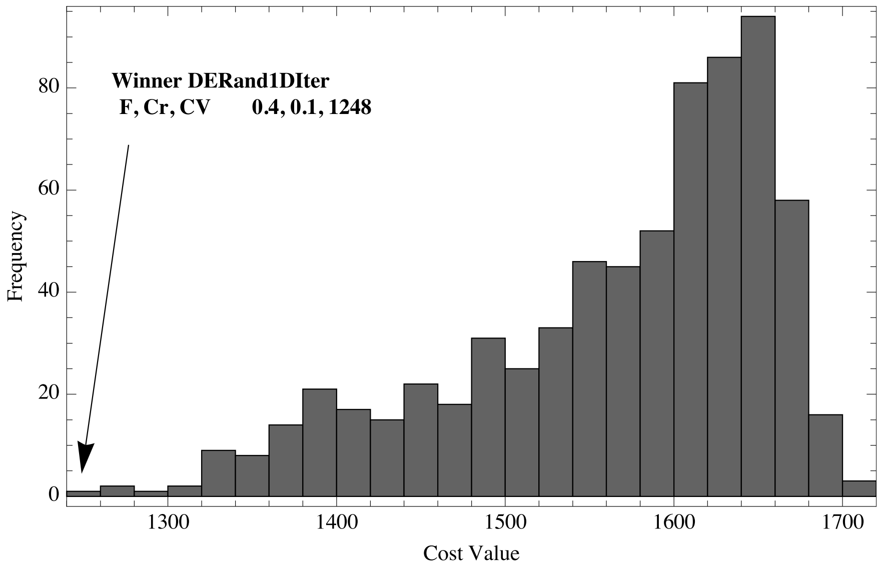

Fig. 7.45. Frequency display for TSP tuning

Using these obtained values, extensive experimentation was conducted on both the FSS and TSP problems. The core issue here is that it is shown that small changes in the control variables leads to different results. The hypothesis that parameter tuning is highly important for tuning of DE for permutative optimization is proven.

Table 7.23. FSS operating parameters

Parameters	Values
Strategy	Rand 1 Bin
F	0.4
CR	0.1

7.6.3 Flow Shop Scheduling Results

The primary experimentation was conducted on Taillard benchmark Flowshop Scheduling [21]. These sets are considered primary with a core mix of complexity and scale. Altogether 120 problem sets are involved, 10 problem instances of n job and m machine problems of 20x5, 20x10, 20x20, 50x5, 50x10, 50x20 100x5, 100x10, 100x20, 200x10, 200x20, and 500x10 are involved. For each problem instance, two bounds are given, the upper bound and the lower bound. Most reference is taken from the upper bound, which is the hypothetical optimal of a particular instance.

So the objective then is not to find the optimal solution (one can if one wants), but to gauge how effective a heuristic is over the entire range of these problems. In others words, to observe the consistency of the heuristic. To this effect, the results are presented in the following format by applying Equation 7.14.

$$\Delta_{avg} = \frac{(H-U) \bullet 100}{U} \tag{7.14}$$

Equation 7.14 is where H represents the obtained value and U is the bound specified by [21]. The Δ_{avg} , gives the average value for all the instances in that particular class, and gives the standard deviation across all the instances. This is important in order to gauge the consistency of the heuristic.

The operating parameters of DE using Discrete Set Handling (DE_{DSH}) is given in Table 7.24. The values of CR and F were obtained through extensive parameter tuning and NP (population size) and Gen (number of generations) was kept at 700.

Table 7.24. DE_{DSH} operating parameters

Parameters	CR	F	NP	Gen
Value	0.5	0.1	500	700

The collated results are presented in Table 7.25. These results are presented with the results compiled by [22].

Generally, two classes of heuristics are observed: those, which are canonical, and those, which have embedded local search. To the first class of heuristics belong GA (Genetic Algorithm), PSO_{spv} (Particle Swamp Optimization with smallest position value) and DE_{spv} (Differential Evolution with smallest position value). The second class has $DE_{spv+exchange}$, which is DE_{spv} with local search.

Table 7.25. Flowshop scheduling results

	GA		PSO_{spv}		DE_{spv}		$DE_{spv+exchange}$		DE_{DSH}		$DE_{DSH+EXH}$	
	Δ_{avg}	Δ_{std}	Δ_{avg}	Δ_{std}	Δ_{avg}	Δ_{std}	Δ_{avg}	Δ_{std}	Δ_{avg}	Δ_{std}	Δ_{avg}	Δ_{std}
20x5	3.13	1.86	1.71	1.25	2.25	1.37	0.69	0.64	1.2	0.42	1.07	0.55
20x10	5.42	1.72	3.28	1.19	3.71	1.24	2.01	0.93	2.5	0.41	2.35	0.6
20x20	4.22	1.31	2.84	1.15	3.03	0.98	1.85	0.87	2.52	0.32	1.92	0.53
50x5	1.69	0.79	1.15	0.7	0.88	0.52	0.41	0.37	0.84	0.56	0.5	0.56
50x10	5.61	1.41	4.83	1.16	4.12	1.1	2.41	0.9	5.09	1.02	3.21	1.11
50x20	6.95	1.09	6.68	1.35	5.56	1.22	3.59	0.78	7.05	1.08	4.21	0.85
100x5	0.81	0.39	0.59	0.34	0.44	0.29	0.21	0.21	0.73	0.32	0.32	0.24
100x10	3.12	0.95	3.26	1.04	2.28	0.75	1.41	0.57	3.11	1.2	1.5	1.08
100x20	6.32	0.89	7.19	0.99	6.78	1.12	3.11	0.55	5.98	0.57	4.19	0.82
200x10	2.08	0.45	2.47	0.71	1.88	0.69	1.06	0.35	3.77	1.31	1.781	1.1
200x20									9.82	0.7	4.32	0.68
500x10									6.28	0.39	4.13	0.41

Table 7.26. Comparison results of heuristics without local search

	GA		PSO_{spv}		DE_{spv}		DE_{DSH}	
	Δ_{avg}	Δ_{std}	Δ_{avg}	Δ_{std}	Δ_{avg}	Δ_{std}	Δ_{avg}	Δ_{std}
20x5	3.13	1.86	1.71	1.25	2.25	1.37	**1.2**	0.42
20x10	5.42	1.72	3.28	1.19	3.71	1.24	**2.5**	0.41
20x20	4.22	1.31	2.84	1.15	3.03	0.98	**2.52**	0.32
50x5	1.69	0.79	1.15	0.7	0.88	0.52	**0.84**	0.56
50x10	5.61	1.41	4.83	1.16	**4.12**	1.1	5.09	1.02
50x20	6.95	1.09	6.68	1.35	**5.56**	1.22	7.05	1.08
100x5	0.81	0.39	0.59	0.34	**0.44**	0.29	0.73	0.32
100x10	3.12	0.95	3.26	1.04	**2.28**	0.75	3.11	1.2
100x20	6.32	0.89	7.19	0.99	6.78	1.12	**5.98**	0.57
200x10	2.08	0.45	2.47	0.71	1.88	0.69	3.77	1.31
200x20							9.82	0.7
500x10							6.28	0.39

The experimentation of $DE_{DSH+EXH}$ was done on two parts, one with local search and one without. The comparison result of DE_{DSH} is given in Table 7.26.

DE_{DSH} was able to find the better average values for the problem sets of 20x5, 20x10, 20x20, 50x5 and 100x20. The others sets was dominated by DE_{spv}. A graphical output for the different sets is given in Fig 7.46. The deviation output is given in Fig 7.47.

The second set is the comparison of the heuristics with local search, namely $DE_{spv+exchange}$ and $DE_{DSH+EXC}$ as presented in Table 7.27.

As observed $DE_{spv+exchange}$ is the better performing heuristic. The last two columns gives the analysis comparisons and on average $DE_{DSH+EXH}$ is only 0.42% away from $DE_{spv+exchange}$. The graphical displays are given in Figs 7.48 and 7.49.

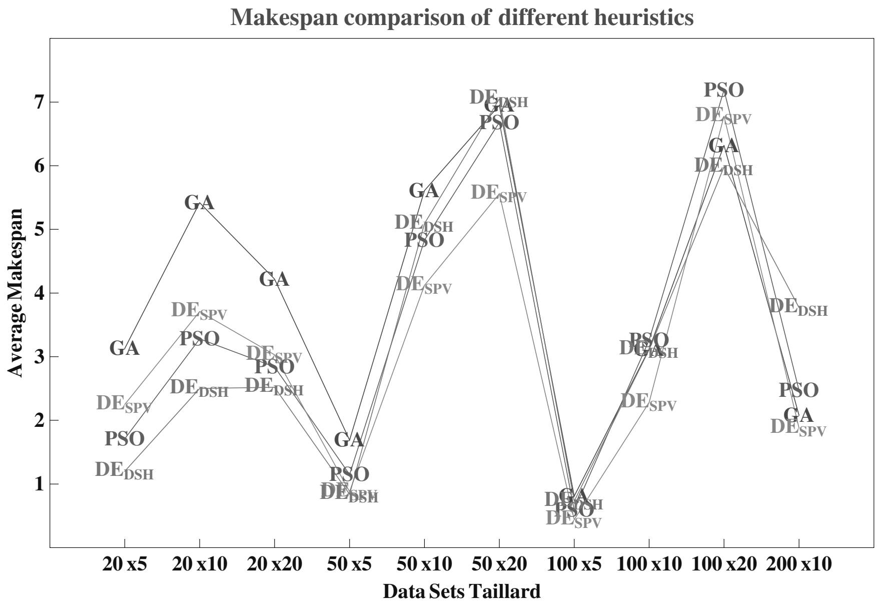

Fig. 7.46. Makespan display of different heuristics without local search

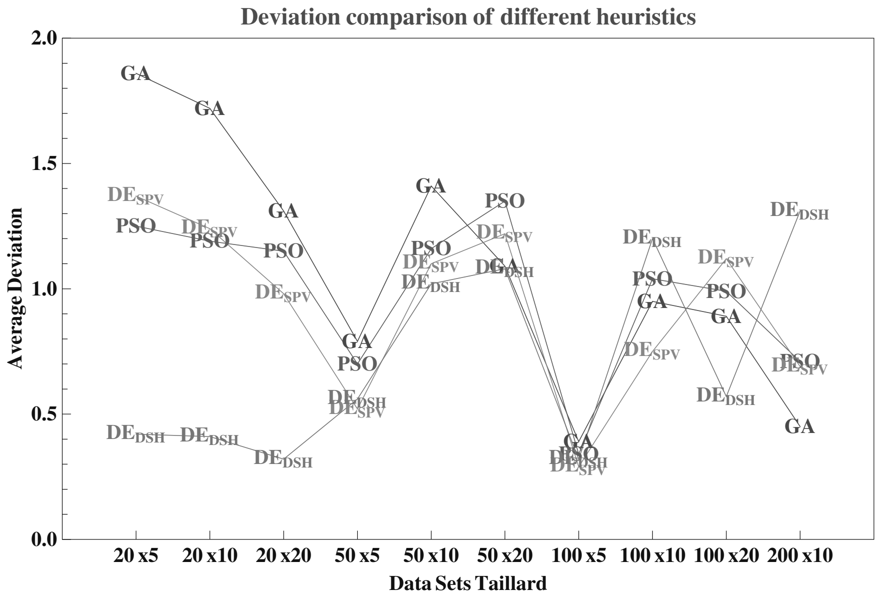

Fig. 7.47. Deviation display of different heuristics without local search

Table 7.27. Comparison results of heuristics with local search

	$DE_{spv+exchange}$		$DE_{DSH+EXH}$		Analysis	
	Δ_{avg}	Δ_{std}	Δ_{avg}	Δ_{std}	Δ_{avg}	Δ_{std}
20x5	0.69	0.64	1.07	0.42	0.38	0.22
20x10	2.01	0.93	2.35	0.41	0.34	0.52
20x20	1.85	0.87	1.92	0.32	0.06	0.55
50x5	0.41	0.37	0.5	0.56	0.09	0.19
50x10	2.41	0.9	3.21	1.02	0.8	0.12
50x20	3.59	0.78	4.21	1.08	0.62	0.3
100x5	0.21	0.21	0.32	0.32	0.11	0.11
100x10	1.41	0.57	1.5	1.2	0.09	0.63
100x20	3.11	0.55	4.19	0.57	1.08	0.02
200x10	1.06	0.35	1.78	1.31	0.72	0.96
200x20			4.32	0.7		
500x10			4.13	0.39		
Average					**0.42**	**0.361**

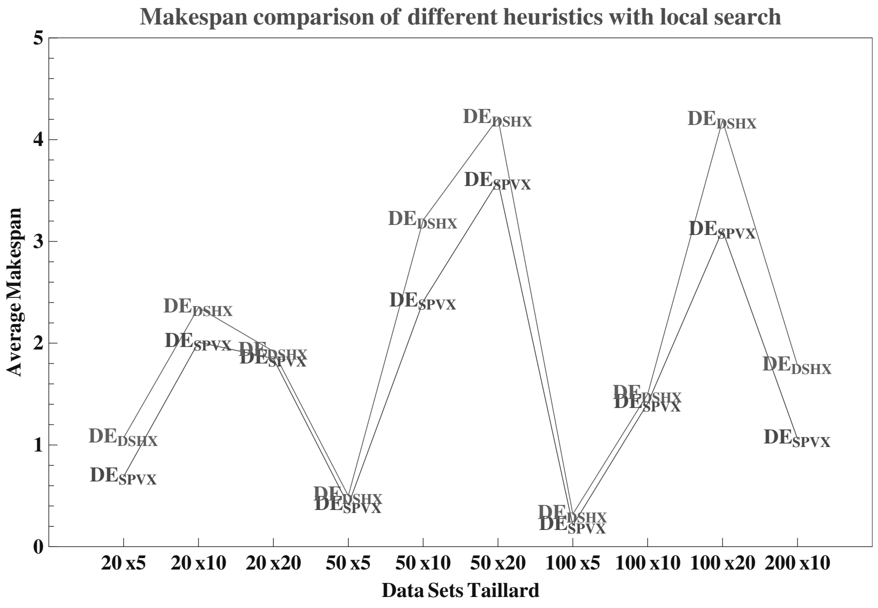

Fig. 7.48. Makespan display of different heuristics with local search

In terms of average deviation, $DE_{DSH+EXH}$ generally has better values than $DE_{spv+exchange}$. This implies that $DE_{DSH+EXC}$ obtains solutions with greater regularity and consistency than $DE_{spv+exchange}$.

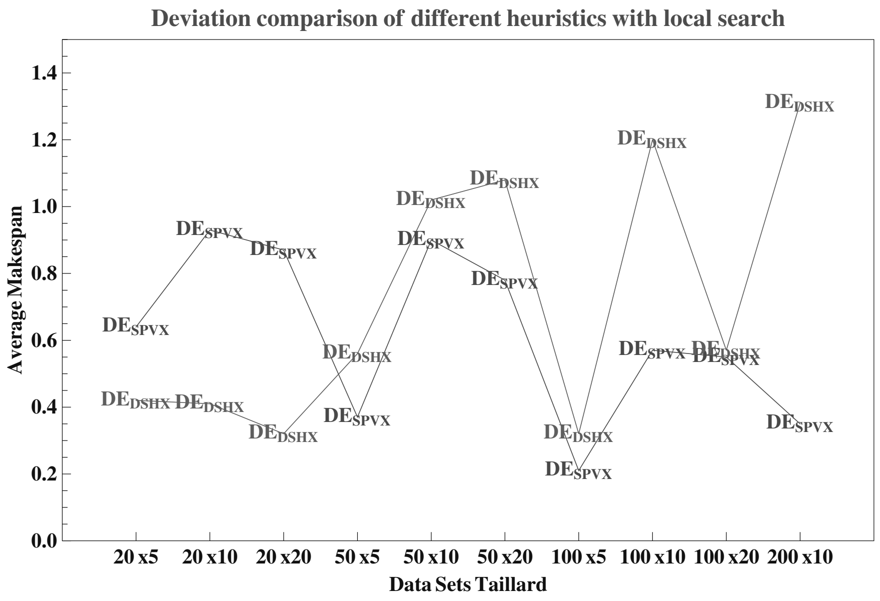

Fig. 7.49. Deviation display of different heuristics with local search

7.6.4 Traveling Salesman Problem Results

7.6.4.1 Symmetric Traveling Salesman

The second set of problem set to be considered is the Traveling Salesman Problem (TSP). TSP is a widely realized problem with many applications in real life problems. However, to compensate for its myriad usage, a number of targeted heuristics have evolved to solve it; often to optimal as is the case for all for all known problem instance in the TSPLIB. For evolutionary heuristics to operate in TSP, it has become a norm for them to employ local search, usually 3 opt [4]. Utilizing local search heuristics always improve the quality of the results of the solutions, since triangle inequality rule and Lin−Kernigham are very robust *deterministic* search heuristics.

The operating parameters for the TSP is given in Table 7.28.

A sample of TSP problem is given in Table 7.28. Comparison is done with the Ant Colony (AC), Simulated Annealing (SA), Self Organising Map (SOM) and Furthest Insertion (FI) of [4].

$$\Delta_{avg} = \frac{H-U}{U} \tag{7.15}$$

Table 7.28. DE_{DSH} TSP operating parameters

Parameters	CR	F	NP	Gen
Value	0.4	0.1	500	700

Table 7.29. STSP comparison results

Instance	Optimal	ACS_{3opt}	SA_{3opt}	SOM	FI_{3opt}	DE_{DSH}
City 1	5.84	0	0	0	0.002	0.002
City 2	5.99	0.002	0	0.002	0	0.03
City 3	5.57	0	0	0.002	0	0.059
City 4	5.06	0.12	0.12	0	0.13	0.16
City 5	6.17	0	0	0.003	0.03	0.01

Table 7.30. General STSP comparison results

Instance	Optimal	ACS_{3opt}	DE_{DSH}
att532	27,686	0	0.17
d198	15,780	0.006	0.54
eil51	426	-	0.08
eil76	538	-	0.1
fl1577	8,806	0.03	1.23
kroA100	21,282	0	0.56
pcb442	50,779	0.01	0.32
rat783	8,806	-	0.92
Average			**0.49**

The results are presented in Table 7.29 as percentage increase upon the reported optimal as given in Equation 7.15.

In this instance, DE_{DSH}, was competitive to the other performing heuristics. As shown, no one heuristic was able to find all optimal values, and some heuristic performed better than other for specific instances.

The second set of experiment was conducted on some selective TSP instances [23]. The results are presented in Table 7.30.

The comparison is done with ACS_{3opt} of [4]. ACS performs very well, almost achieving the optimal solution. DE_{DSH} performs well, obtaining on average 0.49% to the optimal results for the entire set. The set contains instance$'$s ranging from sizes of 51 to 1577 cities.

7.6.4.2 Asymmetric Traveling Salesman

The second set of problems is that, which involves the asymmetric TSP. Asymmetric TSP is one where the distances between two cities are not equal, to and from. This implies that going from one city to another has a different distance than coming back from that city to the original one. The results are presented in Table 7.31.

The results for ATSP are on average 1.112% over the optimal value. However, it should be noted that the experimentation values was kept stagnant to fixed values, even as the problem size was increased, hence the trend of worsening solutions as problem size increases.

Table 7.31. General ATSP comparison results

Instance	Optimal	ACS_{3opt}	DE_{DSH}
ft70	38673	0.001	0.96
ftv170	2755	0.002	2.32
kro124p	36230	0	1.57
p43	5620	0	0.24
ry48p	14422	0	0.47
Average			**1.112**

7.7 Conclusion

Differential Evolution is an effective heuristic for optimization. This approach was an attempt to show it effectiveness in permutative problems. The key approach was to keep the conversion of the operational domain as simple as possible, as shown in this variant of discrete set handling. Simplicity removes excess computation overhead to this heuristic while at the same time delivering comparative results.

Two different problem scopes of Flow Shop scheduling and Traveling Salesman problems were attempted. This was done in order to show that this generic version of DE is able to work in different classes of problems, and not simply tailor made for a special class. The core research focused on Flow Shop with Traveling Salesman providing a secondary comparison.

A principle direction as seen in this research has been the tuning of the heuristic. Researchers, who generally take the default values, often overlook this process, however it is imperative to check for the best values. As shown from the obtained results, the operating values obtained for the two different problems were unique in all aspects.

The results obtained can be visualized as competitive for their own classes. The most promising is the results obtained for Flow Shop, and the worst performing is the Asymmetric Traveling Salesman. It is believed that a better local search heuristic, like Lin−Kernighan or a 3 Opt heuristic will further improve the quality of the solutions.

Further directions for this approach will involve further testing with other problem classes like Vehicle Routing and Quadratic Assignment, which are also realised in real systems.

Acknowledgement

This work was supported by grant No. MSM 7088352101 of the Ministry of Education of the Czech Republic and by grants of the Grant Agency of the Czech Republic GACR 102/06/1132.

References

1. Bland, G., Shallcross, D.: Large traveling salesman problems arising fromexperiments in X-ray crystallography: A preliminary report on computation. OpersRes. Lett. 8, 125–128 (1989)
2. Croes, G.: A method for solving traveling salesman problems. Oper. Res. 6, 791–812 (1958)

3. Davendra, D., Onwubolu, G.: Enhanced Differential Evolution hybrid Scatter Search for Discrete Optimisation. In: Proceeding of the IEEE Congress on Evolutionary Computation, Singapore, September 25-28, pp. 1156–1162 (2007)
4. Dorigo, M., Gambardella, M.: Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem. IEEE Trans. Evol. Comput. 1(1), 53–66 (1997)
5. Flood, M.: The traveling-salesman problem. Oper. Res. 4, 61–75 (1956)
6. Johnson, G.: Artificial immune systems programming for symbolic regression. In: Ryan, C., Soule, T., Keijzer, M., Tsang, E.P.K., Poli, R., Costa, E. (eds.) EuroGP 2003. LNCS, vol. 2610, pp. 345–353. Springer, Heidelberg (2003)
7. Korte, B.: Applications of combinatorial optimization. In: The 13th International Mathematical Programming Symposium, Tokyo (1988)
8. Koza, J.: Genetic Programming: A paradigm for genetically breeding populations of computer programs to solve problems. Stanford University, Computer Science Department, Technical Report STAN–CS–90–1314 (1990)
9. Koza, J.: Genetic Programming. MIT Press, Boston (1998)
10. Koza, J., Bennet, F., Andre, D., Keane, M.: Genetic Programming III. Morgan Kaufnamm, New York (1999)
11. Koza, J., Keane, M., Streeter, M.: Evolving inventions. Sci. Am., 40–47 (February 2003)
12. Garey, M., Johnson, D.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman, San Francisco (1979)
13. O′Neill, M., Ryan, C.: Grammatical Evolution. In: Evolutionary Automatic Programming in an Arbitrary Language. Springer, New York (2003)
14. O′Neill, M., Brabazon, A.: Grammatical Differential Evolution. In: Proc. International Conference on Artificial Intelligence (ICAI 2006), pp. 231–236. CSEA Press (2006)
15. Oplatkova, Z., Zelinka, I.: Investigation on artificial ant using analytic programming. In: Proc. Genetic and Evolutionary Computation Conference 2006, Seattle, WA, pp. 949–950 (2006)
16. Operations Reserach Library (Cited September 1, 2008), `http://people.brunel.ac.uk/~mastjjb/jeb/info.htm`
17. Pinedo, M.: Scheduling: theory, algorithms and systems. Prentice Hall, Inc., New Jersey (1995)
18. Price, K.: An introduction to differential evolution. In: Corne, D., Dorigo, M., Glover, F. (eds.) New Ideas in Optimisation, pp. 79–108. McGraw Hill, International (1999)
19. Ryan, C., Collins, J., O'Neill, M.: Grammatical evolution: Evolving programs for an arbitrary language. In: Banzhaf, W., Poli, R., Schoenauer, M., Fogarty, T.C. (eds.) EuroGP 1998. LNCS, vol. 1391, p. 83. Springer, Heidelberg (1998)
20. Lin, S., Kernighan, B.: An Effective Heuristic Algorithm for the Traveling-Salesman Problem. Oper. Res. 21, 498–516 (1973)
21. Taillard, E.: Benchmarks for basic scheduling problems. Eur. J. Oper. Res. 64, 278–285 (1993)
22. Tasgetiren, M., Liang, Y.-C., Sevkli, M., Gencyilmaz, G.: Differential Evolution Algorithm for Permutative Flowshops Sequencing Problem with Makespan Criterion. In: 4th International Symposium on Intelligent Manufacturing Systems. IMS 2004, Sakaraya, Turkey, September 5–8, pp. 442–452 (2004)
23. TSPLIB (Cited September 1, 2008), `http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsplib.html`
24. Zelinka, I.: Analytic programming by Means of new evolutionary algorithms. In: Proc. 1st International Conference on New Trends in Physics 2001, Brno, Czech Republic, pp. 210–214 (2001)

25. Zelinka, I.: Analytic programming by means of soma algorithm. In: ICICIS 2002, First International Conference on Intelligent Computing and Information Systems, Cairo, Egypt, pp. 148–154 (2002)
26. Zelinka, I., Oplatkova, Z.: Analytic programming – Comparative study. In: Proc. the Second International Conference on Computational Intelligence, Robotics, and Autonomous Systems, Singapore, paper No. PS04-2-04 (2003)
27. Zelinka, I., Oplatkova, Z., Nolle, L.: Analytic programming – Symbolic regression by means of arbitrary evolutionary algorithms. Int. J. Simulat. Syst. Sci. Tech. 6(9), 44–56 (2005)
28. Zelinka, I., Chen, G., Celikovsky, S.: Chaos sythesis by menas of evolutionary algorithms. Int. J. Bifurcat Chaos 4, 911–942 (2008)