3

Forward Backward Transformation

Donald Davendra[1] and Godfrey Onwubolu[2]

[1] Tomas Bata University in Zlin, Faculty of Applied Informatics, Nad Stranemi 4511, Zlin 76001, Czech Republic
davendra@fai.utb.cz

[2] Knowledge Management & Mining, Inc., Richmond Hill, Ontario, Canada
onwubolu_g@dsgm.ca

Abstract. Forward Backward Transformation and its realization, Enhanced Differential Evolution algorithm is one of the permutative versions of Differential Evolution, which has been developed to solve permutative combinatorial optimization problems. Novel domain conversions routines, alongside special enhancement routines and local search heuristic have been incorporated into the canonical Differential Evolution in order to make it more robust and effective.

Three unique and challenging problems of Flow Shop Scheduling, Quadratic Assignment and Traveling Salesman have been solved, utilizing this new approach. The promising results obtained have been compared and analysed against other benchmark heuristics and published work.

3.1 Introduction

Complexity and advancement of technology have been in synch since the industrial revolution. As technology advances, so does the complexity of formulation of these resources.

Current technological trends require a great deal of sophisticated knowledge, both hardware and software supported. This chapter discusses a specific notion of this knowledge, namely the advent of complex heuristics of problem solving.

The notion of *evolutionary heuristics* is one which has its roots in common surrounding. Its premise is that co-operative behavior between many *agents* leads to better and somewhat faster utilisation of the provided resources in the objective of finding the optimal solution to the proposed problem. The optimal solution here refers to a solution, not necessarily the best, but one which can be accepted given the constraints.

Agent based heuristics are those which incorporate a multitude of solutions (unique or replicated) which are then processed using some defined operators to yield new solutions which are presumably better then the previous solutions. These solutions in turn form the next *generation* of solutions. This process iterates for a distinct and predefined number of *generations*.

One of the most prominent heuristic in the scope of real domain problems in Differential Evolution (DE) Algorithm proposed by [31]. Real domain problems are those whose values are essentially real numbers, and the entire solution string can have replicated values. Some of the prominent problems are “De Jong” and “Shwafel” problems which are multi$-$dimensional.

The aim of the research was to ascertain the feasibility of DE to solve a unique class of problems; Permutative Problems. Permutative problems belong to the Nondeterministic Polynomial−Time hard (NP hard) problems. A problem is assigned to such a class if it is solvable in polynomial time by a nondeterministic Turing Machine.

Two different versions of permutative DE are presented. The first is the Discrete Differential Evolution [23, 26, 6] and its superset Enhanced Differential Evolution Algorithm [8, 9].

3.2 Differential Evolution

In order to describe DE, a schematic is given in Fig 3.1.

There are essentially five sections to the code. Section 1 describes the input to the heuristic. D is the size of the problem, $G_{\max}$ is the maximum number of generations, NP is the total number of solutions, F is the scaling factor of the solution and CR is the factor for crossover. F and CR together make the internal tuning parameters for the heuristic.

Section 2 outlines the initialisation of the heuristic. Each solution $x_{i,j,G=0}$ is created randomly between the two bounds $x^{(lo)}$ and $x^{(hi)}$. The parameter j represents the index to the values within the solution and i indexes the solutions within the population. So, to illustrate, $x_{4,2,0}$ represents the second value of the fourth solution at the initial generation.

After initialisation, the population is subjected to repeated iterations in section 3.

Section 4 describes the conversion routines of DE. Initially, three random numbers r_1, r_2, r_3 are selected, unique to each other and to the current indexed solution i in the population in 4.1. Henceforth, a new index j_{rand} is selected in the solution. j_{rand} points to the value being modified in the solution as given in 4.2. In 4.3, two solutions, $x_{j,r_1,G}$ and $x_{j,r_2,G}$ are selected through the index r_1 and r_2 and their values subtracted. This

```
1.Input :$D, G_{\max}, NP \geq 4, F \in (0,1+), CR \in [0,1]$, and initial bounds :$x^{(lo)}, x^{(hi)}$.
2.Initialize : $\begin{cases} \forall i \leq NP \wedge \forall j \leq D : x_{i,j,G=0} = x_j^{(lo)} + rand_j[0,1] \bullet \left(x_j^{(hi)} - x_j^{(lo)}\right) \\ i = \{1,2,...,NP\}, j = \{1,2,...,D\}, G = 0, rand_j[0,1] \in [0,1] \end{cases}$
3.While $G < G_{\max}$
    $\forall i \leq NP$
        4. Mutate and recombine :
        4.1 $r_1, r_2, r_3 \in \{1,2,....,NP\}$,
            randomly selected, except :$r_1 \neq r_2 \neq r_3 \neq i$
        4.2 $j_{rand} \in \{1,2,...,D\}$, randomly selected once each $i$
        4.3 $\forall j \leq D, u_{j,i,G+1} = \begin{cases} x_{j,r_3,G} + F \cdot (x_{j,r_1,G} - x_{j,r_2,G}) \\ \text{if } (rand_j[0,1] < CR \vee j = j_{rand}) \\ x_{j,i,G} \text{ otherwise} \end{cases}$
        5. Select
            $x_{i,G+1} = \begin{cases} u_{i,G+1} & \text{if } f(u_{i,G+1}) \leq f(x_{i,G}) \\ x_{i,G} & \text{otherwise} \end{cases}$
    $G = G + 1$
```

Fig. 3.1. Canonical Differential Evolution Algorithm

value is then multiplied by F, the predefined scaling factor. This is added to the value indexed by r_3 .

However, this solution is not arbitrarily accepted in the solution. A new random number is generated, and if this random number is less than the value of CR, then the new value replaces the old value in the current solution. Once all the values in the solution are obtained, the new solution is vetted for its fitness or value and if this improves on the value of the previous solution, the new solution replaces the previous solution in the population. Hence the competition is only between the new *child* solution and its *parent* solution.

[32] have suggested ten different working strategies. It mainly depends on the problem on hand for which strategy to choose. The strategies vary on the solutions to be perturbed, number of difference solutions considered for perturbation, and finally the type of crossover used. The following are the different strategies being applied.

Strategy 1: DE/best/1/exp:	$u_{i,G+1} = x_{best,G} + F \bullet (x_{r_1,G} - x_{r_2,G})$
Strategy 2: DE/rand/1/exp:	$u_{i,G+1} = x_{r_1,G} + F \bullet (x_{r_2,G} - x_{r_3,G})$
Strategy 3: DE/rand−best/1/exp:	$u_{i,G+1} = x_{i,G} + \lambda \bullet (x_{best,G} - x_{r_1,G})$
	$+F \bullet (x_{r_1,G} - x_{r_2,G})$
Strategy 4: DE/best/2/exp:	$u_{i,G+1} = x_{best,G} + F \bullet (x_{r_1,G} - x_{r_2,G} - x_{r_3,G} - x_{r_4,G})$
Strategy 5: DE/rand/2/exp:	$u_{i,G+1} = x_{5,G} + F \bullet (x_{r_1,G} - x_{r_2,G} - x_{r_3,G} - x_{r_4,G})$
Strategy 6: DE/best/1/bin:	$u_{i,G+1} = x_{best,G} + F \bullet (x_{r_1,G} - x_{r_2,G})$
Strategy 7: DE/rand/1/bin:	$u_{i,G+1} = x_{r_1,G} + F \bullet (x_{r_2,G} - x_{r_3,G})$
Strategy 8: DE/rand−best/1/bin:	$u_{i,G+1} = x_{i,G} + \lambda \bullet (x_{best,G} - x_{r_1,G})$
	$+F \bullet (x_{r_1,G} - x_{r_2,G})$
Strategy 9: DE/best/2/bin:	$u_{i,G+1} = x_{best,G} + F \bullet (x_{r_1,G} - x_{r_2,G} - x_{r_3,G} - x_{r_4,G})$
Strategy 10: DE/rand/2/bin:	$u_{i,G+1} = x_{5,G} + F \bullet (x_{r_1,G} - x_{r_2,G} - x_{r_3,G} - x_{r_4,G})$

The convention shown is DE/x/y/z. DE stands for Differential Evolution, x represents a string denoting the solution to be perturbed, y is the number of difference solutions considered for perturbation of x, and z is the type of crossover being used (exp: exponential; bin: binomial).

DE has two main phases of crossover: binomial and exponential. Generally a child solution $u_{i,G+1}$ is either taken from the parent solution $x_{i,G}$ or from a mutated donor solution $v_{i,G+1}$ as shown: $u_{j,i,G+1} = v_{j,i,G+1} = x_{j,r_3,G} + F \bullet (x_{j,r_1,G} - x_{j,r_2,G})$.

The frequency with which the donor solution $v_{i,G+1}$ is chosen over the parent solution $x_{i,G}$ as the source of the child solution is controlled by both phases of crossover. This is achieved through a user defined constant, crossover CR which is held constant throughout the execution of the heuristic.

The *binomial* scheme takes parameters from the donor solution every time that the generated random number is less than the CR as given by $rand_j[0,1] < CR$, else all parameters come from the parent solution $x_{i,G}$.

The *exponential* scheme takes the child solutions from $x_{i,G}$ until the first time that the random number is greater than CR, as given by $rand_j[0,1] < CR$, otherwise the parameters comes from the parent solution $x_{i,G}$.

To ensure that each child solution differs from the parent solution, both the exponential and binomial schemes take at least one value from the mutated donor solution $v_{i,G+1}$.

3.2.1 Tuning Parameters

Outlining an absolute value for *CR* is difficult. It is largely problem dependent. However a few guidelines have been laid down [31]. When using binomial scheme, intermediate values of *CR* produce good results. If the objective function is known to be separable, then $CR = 0$ in conjunction with binomial scheme is recommended. The recommended value of *CR* should be close to or equal to 1, since the possibility or crossover occurring is high. The higher the value of *CR*, the greater the possibility of the random number generated being less than the value of *CR*, and thus initiating the crossover.

The general description of *F* is that it should be at least above 0.5, in order to provide sufficient scaling of the produced value.

The tuning parameters and their guidelines are given in Table 3.1

Table 3.1. Guide to choosing best initial control variables

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

3.3 Discrete Differential Evolution

The canonical DE cannot be applied to discrete or permutative problems without modification. The internal crossover and mutation mechanism invariably change any applied value to a real number. This in itself will lead to in-feasible solutions.

The objective then becomes one of transformation, either that of the population or that of the internal crossover/mutation mechanism of DE. For this chapter, it was decided not to modify in any way the operation of DE strategies, but to manipulate the population in such a way as to enable DE to operate unhindered.

Since the solution for the population is permutative, a suitable conversion routine was required in order to change the solution from integer to real and then back to integer after crossover. The population was generated as a permutative string. Two conversions routines were devised, one was Forward transformation and the other Backward transformation for the conversion between integer and real values. This new heuristic was termed Discrete Differential Evolution (DDE) [28].

The basic outline DDE is given below.

1. **Initial Phase**
   a) *Population Generation*: An initial number of discrete trial solutions are generated for the initial population.

2. **Conversion**
   a) *Discrete to Floating Conversion*: This conversion schema transforms the parent solution into the required continuous solution.
   b) *DE Strategy*: The DE strategy transforms the parent solution into the child solution using its inbuilt crossover and mutation schemas.
   c) *Floating to Discrete Conversion*: This conversion schema transforms the continuous child solution into a discrete solution.
3. **Selection**
   a) *Validation*: If the child solution is feasible, then it is evaluated and accepted in the next population, if it improves on the parent solution.

3.3.1 Permutative Population

The first part of the heuristic generates the permutative population. A permutative solution is one, where each value within the solution is unique and systematic. A basic description is given in Equation 3.1.

$$\begin{aligned} P_G = \{x_{1,G}, x_{2,G}, \dots, x_{NP,G}\},\ x_{i,G} = x_{j,i,G} \\ x_{j,i,G=0} = (\text{int})\left(rand_j[0,1] \bullet \left(x_j^{(hi)} + 1 - x_j^{(lo)}\right) + \left(x_j^{(lo)}\right)\right) \\ if\ x_{j,i} \notin \{x_{0,i}, x_{1,i}, \dots, x_{j-1,i}\} \\ i = \{1,2,3,\dots,NP\}, j = \{1,2,3,..,D\} \end{aligned} \tag{3.1}$$

where P_G represents the population, $x_{j,i,G=0}$ represents each solution within the population and $x_j^{(lo)}$ and $x_j^{(hi)}$ represents the bounds. The index *i* references the solution from 1 to *NP*, and *j* which references the values in the solution.

3.3.2 Forward Transformation

The transformation schema represents the most integral part of the code. [23] developed an effective routine for the conversion.

Let a set of integer numbers be represented as in Equation 3.2:

$$x_i \in x_{i,G} \tag{3.2}$$

which belong to solution $x_{j,i,G=0}$. The equivalent continuous value for x_i is given as $1 \bullet 10^2 < 5 \bullet 10^2 \leq 10^2$.

The domain of the variable x_i has length = 5 as shown in $5 \bullet 10^2$. The precision of the value to be generated is set to two decimal places (2 d.p.) as given by the superscript two (2) in 10^2 . The range of the variable x_i is between 1 and 10^3. The lower bound is 1 whereas the upper bound of 10^3 was obtained after extensive experimentation. The upper bound 10^3 provides optimal filtering of values which are generated close together [27].

The formulation of the forward transformation is given as:

$$x_i' = -1 + \frac{x_i \bullet f \bullet 5}{10^3 - 1} \tag{3.3}$$

Equation 3.3 when broken down, shows the value x_i multiplied by the length 5 and a scaling factor f. This is then divided by the upper bound minus one (1). The value computed is then decrement by one (1). The value for the scaling factor f was established after extensive experimentation. It was found that when f was set to 100, there was a tight grouping of the value, with the retention of optimal filtration′s of values. The subsequent formulation is given as:

$$x_i' = -1 + \frac{x_i \bullet f \bullet 5}{10^3 - 1} = -1 + \frac{x_i \bullet f \bullet 5}{10^3 - 1} \tag{3.4}$$

Illustration:

Take a integer value 15 for example. Applying Equation 3.3, we get:

$$x_i' = -1 + \frac{15 \bullet 500}{999} = 6.50751$$

This value is used in the DE internal representation of the population solution parameters so that mutation and crossover can take place.

3.3.3 Backward Transformation

The reverse operation to forward transformation, backward transformation converts the real value back into integer as given in Equation 3.5 assuming x_i to be the real value obtained from Equation 3.4.

$$\text{int}\,[x_i] = \frac{(1 + x_i) \bullet \left(10^3 - 1\right)}{5 \bullet f} = \frac{(1 + x_i) \bullet \left(10^3 - 1\right)}{500} \tag{3.5}$$

The value x_i is rounded to the nearest integer.

Illustration:

Take a continuous value -0.17. Applying equation Equation 3.5:

$$\text{int}\,[x_i] = \frac{(1 + -0.17) \bullet \left(10^3 - 1\right)}{500} = |3.3367| = 3$$

The obtained value is 3, which is the rounded value after transformation.

These two procedures effectively allow DE to optimise permutative solutions.

3.3.4 Recursive Mutation

Once the solution is obtained after transformation, it is checked for feasibility. Feasibility refers to whether the solutions are within the bounds and unique in the solution.

$$x_{i,G+1} = \begin{cases} u_{i,G+1} \text{ if } \begin{cases} u_{j,i,G+1} \neq \left\{u_{1,i,G+1}, \ldots, u_{j-1,i,G+1}\right\} \\ x^{(lo)} \leq u_{j,i,G+1} \leq x^{(lo)} \end{cases} \\ x_{i,G} \end{cases} \tag{3.6}$$

$$
\begin{array}{l}
\text{Input}: D, G_{\max}, NP \geq 4, F \in (0,1+), CR \in [0,1], \text{bounds}: x^{(lo)}, x^{(hi)}.\\
\text{Initialize}: \begin{cases} \forall i \leq NP \wedge \forall j \leq D \begin{cases} x_{i,j,G=0} = x_j^{(lo)} + rand_j[0,1] \bullet \left(x_j^{(hi)} - x_j^{(lo)}\right) \\ if x_{j,i} \notin \{x_{0,i}, x_{1,i}, \ldots, x_{j-1,i}\} \end{cases} \\ i = \{1,2,\ldots,NP\}, j = \{1,2,\ldots,D\}, G = 0, rand_j[0,1] \in [0,1] \end{cases}\\
\text{Cost}: \forall i \leq NP : f(x_{i,G=0})\\
\begin{cases}
\text{While} \quad G < G_{\max} \\
\forall i \leq NP \begin{cases}
\text{Mutate and recombine}: \\
\quad r_1, r_2, r_3 \in \{1,2,\ldots.,NP\}, \text{randomly selected, except}: r_1 \neq r_2 \neq r_3 \neq i \\
\qquad j_{rand} \in \{1,2,\ldots,D\}, \text{randomly selected once each } i \\
\forall j \leq D, u_{j,i,G+1} = \begin{cases}
(\gamma_{j,r_3,G}) \leftarrow (x_{j,r_3,G}) : (\gamma_{j,r_1,G}) \leftarrow (x_{j,r_1,G}) : (\gamma_{j,r_2,G}) \leftarrow (x_{j,r_2,G}) \\
\qquad \text{Forward Transformation} \\
\gamma_{j,r_3,G} + F \cdot (\gamma_{j,r_1,G} - \gamma_{j,r_2,G}) \\
\qquad \text{if } (rand_j[0,1] < CR \vee j = j_{rand}) \\
(\gamma_{j,i,G}) \leftarrow (x_{j,i,G}) \text{ otherwise}
\end{cases} \\
\left(u'_{i,G+1}\right) = (\rho_{j,i,G+1}) \leftarrow (\varphi_{j,i,G+1}) \quad \text{Backward Transformation} \\
\text{Recursive Mutation}: \\
u_{i,G+1} = \begin{cases} u_{i,G+1} \text{if} \begin{cases} u_{j,i,G+1} \neq \{u_{1,i,G+1}, \ldots, u_{j-1,i,G+1}\} \\ x^{(lo)} \leq u_{j,i,G+1} \leq x^{(hi)} \end{cases} \\ x_{i,G} \quad \text{otherwise} \end{cases} \\
\text{Select}: \\
x_{i,G+1} = \begin{cases} u_{i,G+1} \text{ if } f(u_{i,G+1}) \leq f(x_{i,G}) \\ x_{i,G} \quad \text{otherwise} \end{cases}
\end{cases} \\
G = G + 1
\end{cases}
\end{array}
$$

Fig. 3.2. DDE schematic

Recursive mutation refers to the fact that if a solution is deemed in-feasible, it is discarded and the parent solution is retained in the population as given in Equation 3.6.

The general schematic is given in Figure 3.2.

A number of experiments were conducted by DDE on Flowshop Scheduling problems. These are collectively given in the results section of this chapter.

3.4 Enhanced Differential Evolution

Enhanced Differential Evolution (EDE) [7, 8, 9], heuristic is an extension of the DDE variant of DE. One of the major drawbacks of the DDE algorithm was the high frequency of in-feasible solutions, which were created after evaluation. However, since DDE showed much promise, the next logical step was to devise a method, which would repair the in-feasible solutions and hence add viability to the heuristic.

To this effect, three different repairment strategies were developed, each of which used a different index to repair the solution. After repairment, three different enhancement features were added. This was done to add more depth to the code in order to solve permutative problems. The enhancement routines were standard mutation, insertion and local search. The basic outline is given below.

1. **Initial Phase**
   a) *Population Generation*: An initial number of discrete trial solutions are generated for the initial population.

2. **Conversion**
   a) *Discrete to Floating Conversion*: This conversion schema transforms the parent solution into the required continuous solution.
   b) *DE Strategy*: The DE strategy transforms the parent solution into the child solution using its inbuilt crossover and mutation schemas.
   c) *Floating to Discrete Conversion*: This conversion schema transforms the continuous child solution into a discrete solution.
3. **Mutation**
   a) *Relative Mutation Schema*: Formulates the child solution into the discrete solution of unique values.
4. **Improvement Strategy**
   a) *Mutation*: Standard mutation is applied to obtain a better solution.
   b) *Insertion*: Uses a two-point cascade to obtain a better solution.
5. **Local Search**
   a) *Local Search*: 2 Opt local search is used to explore the neighborhood of the solution.

3.4.1 Repairment

In order to repair the solutions, each solution is initially vetted. Vetting requires the resolution of two parameters: firstly to check for any bound offending values, and secondly for repeating values in the solution. If a solution is detected to have violated a bound, it is dragged to the offending boundary.

```
Input : $D$
Array $Solution, ViolateVal, MissingVal$
int $Counter$
for (int $i = 0; i < D; i++$) {
  for (int j $= 0; j < D; j++$) {
    if ($i == Solution[j]$) {
      $Counter++$; }
  }
  if ($Counter > 1$) {
    int $Index = 0$;
      for (int $j = 0; j < D; j++$) {
        if ($i = Solution[j]$) {
          $Index++$
          if ($Index > 1$) {
          ViolateVal $\overset{Append}{\leftarrow} j$; }
  }}}
  if ($Counter == 0$) {
    MissingVal $\overset{Append}{\leftarrow} i$; }
  $Counter = 0$;
  }
```

Fig. 3.3. Pseudocode for replication detection

$$u_{j,i,G+1} = \begin{cases} x^{(lo)} \text{ if } u_{j,i,G+1} < x^{(lo)} \\ x^{(hi)} \text{ if } u_{j,i,G+1} > x^{(hi)} \end{cases} \tag{3.7}$$

Each value, which is replicated, is tagged for its value and index. Only those values, which are deemed replicated, are repaired, and the rest of the values are not manipulated. A second sequence is now calculated for values, which are not present in the solution. It stands to reason that if there are replicated values, then some feasible values are missing. The pseudocode if given in Fig 3.3.

Three unique repairment strategies were developed to repair the replicated values: *front mutation*, *back mutation* and *random mutation*, named after the indexing used for each particular one.

3.4.1.1 Front Mutation

Front mutation indexes the repairment from the front of the replicated array with values randomly selected from the missing value array as shown in Fig 3.4.

```
Array $Solution, ViolateVal, MissingVal$;
for (int $i = 0; i < sizeofViolateVal; i++$)
  Solution $[ViolateVal[i]]$ = Random $[MissingVal]$;
}
```

Fig. 3.4. Pseudocode for front mutation

Illustration:

In order to understand *front mutation*, assume an in−feasible solution of dimension $D = 10$: $x = \{3,4,2,1,3,5,6,7,10,5\}$.

The first step is to isolate all repetitive values in the solution. These are highlighted in the following array: $x = \{\mathbf{3},4,2,1,\mathbf{3},\mathbf{5},6,7,10,\mathbf{5}\}$. As shown, the values 3 and 5 are repeated in the solution.

All *first* occurring values are now set as default: $x = \{3,4,2,1,\mathbf{3},5,6,7,10,\mathbf{5}\}$. So now only two positions are replicated, index 5 and 10 as given: $x = \{3,4,2,1,\underset{5}{3},5,6,7,10,\underset{10}{5}\}$.

An array of missing values is now generated as $MV = \{8,9\}$, since values 8 and 9 are missing from the solution.

An insertion array is now randomly generated, which specifies the position of the insertion of each value: $IA = \{2,1\}$. Since only two values were missing so only two random numbers are generated. In this respect, the first value 2 in IA, outlines that the value pointed by index 1 in MV which is 8 is to be placed in the second indexed in-feasible solution and likewise for the other missing value given as: $x = \{3,4,2,1,\underset{1}{9},5,6,7,10,\underset{2}{8}\}$.

3.4.1.2 Back Mutation

Back mutation is the opposite of front mutation, and indexes the repairment from the rear of the replicated array as given in Fig 3.5.

```
Array $Solution, ViolateVal, MissingVal$;
for (int $i = sizeofViolateVal; i > 0; i++$)
    Solution $[ViolateVal\,[i]]$ = Random $[MissingVal]$;
}
```

Fig. 3.5. Pseudocode for back mutation

Illustration:

In order to understand back mutation assume the same in-feasible solution as in the previous example: $x = \{3,4,2,1,3,5,6,7,10,5\}$.

The first step is to isolate all repetitive values in the solution. These are highlighted in the following array: $x = \{\mathbf{3},4,2,1,\mathbf{3},\mathbf{5},6,7,10,\mathbf{5}\}$. As shown the values 3 and 5 are repeated in the solution.

All last occurring values are now set as default: $x = \{\mathbf{3},4,2,1,3,\mathbf{5},6,7,10,5\}$. So now only two positions are replicated, index 1 and 6 as given: $x = \{\underset{1}{3},4,2,1,3,\underset{6}{5},6,7,10,5\}$.

An array of missing values is now generated as $MV = \{8,9\}$, since values 8 and 9 are missing from the solution.

An insertion array is now randomly generated, which specifies the position of the insertion of each value: $IA = \{2,1\}$. Since only two values were missing so only two random numbers are generated. In this respect, the first value 1 in IA, outlines that the value pointed by index 1 in MV which is 8 is to be placed in the first indexed in-feasible solution and likewise for the other missing value given as: $x = \{\underset{1}{8},4,2,1,3,\underset{2}{9},6,7,10,5\}$.

3.4.1.3 Random Mutation

The most complex repairment schema is the random mutation routine. Each value is selected randomly from the replicated array and replaced randomly from the missing value array as given in Fig 3.6.

```
Array $Solution, ViolateVal, MissingVal$;
for (int $i = sizeofViolateVal; i > 0; i++$)
    Solution $\left[ViolateVal_{Random[i]}\right] = MissingVal_{Random[i]}$;
$ViolateVal \overset{delete}{\leftarrow} ViolateVal_{Random[i]}$;
$MissingVal \overset{delete}{\leftarrow} MissingVal_{Random[i]}$;
}
```

Fig. 3.6. Pseudocode for random mutation

Since each value is randomly selected, the value has to be removed from the array after selection in order to avoid duplication. Through experimentation it was shown that random mutation was the most effective in solution repairment.

Illustration:

Following the previous illustrations, assume the same in-feasible solution: $x = \{3,4,2,1,3,5,6,7,10,5\}$.

The first step is to isolate all repetitive values in the solution. These are highlighted in the following array: $x = \{\mathbf{3},4,2,1,\mathbf{3},\mathbf{5},6,7,10,\mathbf{5}\}$. As shown the values 3 and 5 are repeated in the solution.

A random array is created which sets the default values: $DV = \{2,1\}$, . Here, it shows that the first replicated value which is 3 should be set as default on its second occurrence. The second replicated value 5 should be set as default on its first occurrence: $x = \{\mathbf{3},4,2,1,3,5,6,7,10,\mathbf{5}\}$. The in-feasible values are now in index 1 and 10 given as $x = \{\underset{1}{3},4,2,1,3,5,6,7,10,\underset{10}{5}\}$

An array of missing values is now generated as $MV = \{8,9\}$, since values 8 and 9 are missing from the solution.

An insertion array is now randomly generated, which specifies the position of the insertion of each value: $IA = \{1,2\}$. Since only two values were missing so only two random numbers are generated. In this respect, the first value 1 in IA, outlines that the value pointed by index 1 in MV which is 8 is to be placed in the first indexed in-feasible solution and likewise for the other missing value given as: $x = \{\underset{1}{8},4,2,1,3,5,6,7,10,\underset{10}{9}\}$.

3.4.2 Improvement Strategies

Improvement strategies were included in order to improve the quality of the solutions. Three improvement strategies were embedded into the heuristic. All of these are one time application based. What this entails is that, once a solution is created each strategy is applied only once to that solution. If improvement is shown, then it is accepted as the new solution, else the original solution is accepted in the next population.

3.4.2.1 Standard Mutation

Standard mutation is used as an improvement technique, to explore random regions of space in the hopes of finding a better solution. Standard mutation is simply the exchange of two values in the single solution.

Two unique random values are selected $r_1, r_2 \in rand\,[1,D]$, where as $r_1 \neq r_2$. The values indexed by these values are exchanged: $Solution_{r_1} \overset{exchange}{\leftrightarrow} Solution_{r_1}$ and the solution is evaluated. If the fitness improves, then the new solution is accepted in the population.

Illustration:

In Standard Mutation assume a solution given as: $x = \{8,4,2,1,3,5,6,7,10,9\}$. Two random numbers are generated within the bounds: $Rnd = \{3,8\}$. These are the indexes of the values in the solution: $x = \{8,4,\underset{3}{2},1,3,5,6,\underset{8}{7},10,9\}$. The values are exchanged $x = \{8,4,7,1,3,5,6,2,10,9\}$ and the solution is evaluated for its fitness.

3.4.2.2 Insertion

Insertion is a more complicated form of mutation. However, insertion is seen as providing greater diversity to the solution than standard mutation.

As with standard mutation, two unique random numbers are selected $r_1, r_2 \in rand\ [1,D]$. The value indexed by the lower random number $Solution_{r_1}$ is removed and the solution from that value to the value indexed by the other random number is shifted one index down. The removed value is then inserted in the vacant slot of the higher indexed value $Solution_{r_2}$ as given in Fig 3.7.

```
$temp = Solution_{r_1};$
for (int $i = r_1; i < r_2; i{+}{+}$)
    $Solution_i = Solution_{i++};$
}
$Solution_{r_2} = temp;$
```

Fig. 3.7. Pseudocode for Insertion

Illustration:

In this Insertion example, assume a solution given as: $x = \{8,4,2,1,3,5,6,7,10,9\}$. Two random numbers are generated within the bounds: $Rnd = \{4,7\}$. These are the indexes of the values in the solution: $x = \{8,4,7,\left|\underset{4}{1},3,5,\underset{7}{6}\right|,2,10,9\}$. The lower indexed value is removed from the solution $x = \{8,4,7,\left|\underset{4}{\ },3,5,\underset{7}{6}\right|,2,10,9\}$, and all values from the upper index are moved one position down $x = \{8,4,7,|3,5,6,\ |,2,10,9\}$. The lower indexed value is then slotted in the upper index: $x = \{8,4,7,3,5,6,1,2,10,9\}$.

3.4.3 Local Search

There is always a possibility of stagnation in Evolutionary Algorithms. DE is no exemption to this phenomenon.

Stagnation is the state where there is no improvement in the populations over a period of generations. The solution is unable to find new search space in order to find global optimal solutions. The length of stagnation is not usually defined. Sometimes a period of twenty generation does not constitute stagnation. Also care has to be taken as not be confuse the local optimal solution with stagnation. Sometimes better search space simply does not exist. In EDE, a period of five generations of non-improving optimal solution is classified as stagnation. Five generations is taken in light of the fact that EDE usually operates on an average of hundred generations. This yields to the maximum of twenty stagnations within one run of the heuristic.

```
$\alpha = \emptyset$
$while \quad \alpha| < D$
  $i = rand[1,D], i \notin \alpha$
  $\beta = \{i\}$
  $while \;\; \beta| < D$
    $j = rand[1,D], j \notin \beta$
    If $\Delta(x,i,j) < 0; \begin{cases} x_i = x_j \\ x_j = x_i \end{cases}$
    $\beta = \beta \cup \{j\}$
  $\alpha = \alpha \cup \{j\}$
```

Fig. 3.8. Pseudocode for 2 Opt Local Search

To move away from the point of stagnation, a feasible operation is a neighborhood or local search, which can be applied to a solution to find better feasible solution in the local neighborhood. Local search in an improvement strategy. It is usually independent of the search heuristic, and considered as a plug-in to the main heuristic. The point of note is that local search is very expensive in terms of time and memory. Local search can sometimes be considered as a brute force method of exploring the search space. These constraints make the insertion and the operation of local search very delicate to implement. The route that EDE has adapted is to check the optimal solution in the population for stagnation, instead of the whole population. As mentioned earlier five (5) non-improving generations constitute stagnation. The point of insertion of local search is very critical. The local search is inserted at the termination of the improvement module in the EDE heuristic.

Local Search is an approximation algorithm or heuristic. Local Search works on a *neighborhood*. A complete *neighborhood* of a solution is defined as the set of all solutions that can be arrived at by a move. The word solution should be explicitly defined to reflect the problem being solved. This variant of the local search routine is described in [24] as is generally known as a 2-opt local search.

The basic outline of a Local Search technique is given in Fig 3.8. A number α is chosen equal to zero (0) ($\alpha = \emptyset$). This number iterates through the entire population, by choosing each progressive value from the solution. On each iteration of α, a random number i is chosen which is between the lower (1) and upper (D) bound. A second number β starts at the position i, and iterates till the end of the solution. In this second iteration another random number j is chosen, which is between the lower and upper bound and not equal to value of β. The values in the solution indexed by i and j are swapped. The objective function of the new solution is calculated and only if there is an improvement given as $\Delta(x,i,j) < 0$, then the new solution is accepted.

Illustration:

To understand how this local search operates, consider two solutions x_1 and x_2. The operations parameters of these solutions are:
Upper bound $x^{(hi)} = 5$
Lower bound $x^{(lo)} = 1$
Solution size $D = 5$

```
Input : $D, G_{\max}, NP \geq 4, F \in (0, 1+), CR \in [0,1]$, and bounds : $x^{(lo)}, x^{(hi)}$.
Initialize : $\forall i \leq NP \wedge \forall j \leq D \begin{cases} x_{i,j,G=0} = x_j^{(lo)} + rand_j[0,1] \bullet \left(x_j^{(hi)} - x_j^{(lo)}\right) \\ if \;\; x_{j,i} \notin \{x_{0,i}, x_{1,i}, \ldots, x_{j-1,i}\} \end{cases}$
             $i = \{1,2,\ldots,NP\}, j = \{1,2,\ldots,D\}, G = 0, rand_j[0,1] \in [0,1]$
Cost : $\forall i \leq NP : f\left(x_{i,G=0}\right)$
While $G < G_{\max}$
    $\forall i \leq NP$
        Mutate and recombine :
        $r_1, r_2, r_3 \in \{1,2,\ldots,NP\}$, randomly selected, except : $r_1 \neq r_2 \neq r_3 \neq i$
        $j_{rand} \in \{1,2,\ldots,D\}$, randomly selected once each $i$
        $\forall j \leq D, u_{j,i,G+1} = \begin{cases} \left(\gamma_{j,r_3,G}\right) \leftarrow \left(x_{j,r_3,G}\right) : \left(\gamma_{j,r_1,G}\right) \leftarrow \left(x_{j,r_1,G}\right) : \\ \left(\gamma_{j,r_2,G}\right) \leftarrow \left(x_{j,r_2,G}\right) \quad \text{Forward Transformation} \\ \gamma_{j,r_3,G} + F \cdot \left(\gamma_{j,r_1,G} - \gamma_{j,r_2,G}\right) \\ \qquad \text{if } \left(rand_j[0,1] < CR \vee j = j_{rand}\right) \\ \left(\gamma_{j,i,G}\right) \leftarrow \left(x_{j,i,G}\right) \quad \text{otherwise} \end{cases}$
        $\left(u'_{i,G+1}\right) = \begin{cases} \left(\rho_{j,i,G+1}\right) \leftarrow \left(\varphi_{j,i,G+1}\right) \text{Backward Transformation} \\ \left(u_{j,i,G+1}\right) \overset{mutate}{\leftarrow} \left(\rho_{j,i,G+1}\right) \text{Mutate Schema} \\ \text{if} \left(u'_{j,i,G+1}\right) \notin \left\{\left(u_{0,i,G+1}\right), \left(u_{1,i,G+1}\right), .. \left(u_{j-1,i,G+1}\right)\right\} \end{cases}$
        $\left(u_{j,i,G+1}\right) \leftarrow \left(u'_{i,G+1}\right)$ Standard Mutation
        $\left(u_{j,i,G+1}\right) \leftarrow \left(u'_{i,G+1}\right)$ Insertion
        Select :
        $x_{i,G+1} = \begin{cases} u_{i,G+1} & \text{if } f(u_{i,G+1}) \leq f(x_{i,G}) \\ x_{i,G} & \text{otherwise} \end{cases}$
    $G = G + 1$
    Local Search   $x_{best} = \Delta\left(x_{best}, i, j\right)$   if stagnation
```

Fig. 3.9. EDE Template

$x_1 = \{2,5,4,3,1\}$ and $x_1 = \{2,5,4,3,1\}$

Each value in x_1 and x_2 are paired up and considered.

$$\Delta(i,j) = \left\{ \begin{array}{l} \{2,4\},\{2,2\},\{2,1\},\{2,5\},\{2,3\}, \\ \{5,4\},\{5,2\},\{5,1\},\{5,5\},\{5,3\}, \\ \{3,4\},\{3,2\},\{3,1\},\{3,5\},\{3,3\}, \\ \{1,4\},\{1,2\},\{1,1\},\{1,5\},\{1,3\} \end{array} \right\}$$

The cost of the move $\Delta(x,i,j)$ is evaluated. If this value is negative the objective function value for the problem is decrement by $\Delta(x,i,j)$. Hence the solution is improved to a near optimal solution.

The complete template of Enhanced Differential Evolution is given in Fig 3.9.

3.5 Worked Example

This worked example outlines how EDE is used to solve the flowshop scheduling problem. The problem to be solved is the one represented in Table 3.26 (Section 3.6.1: Flow Shop Scheduling Example).

Table 3.2. Table of solutions

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

Table 3.3. Table of initial population with fitness

Fitness	Population				
32	1	2	3	4	5
31	2	1	4	3	5
33	4	5	3	1	2
35	3	1	4	5	2
34	4	2	5	3	1
31	5	4	3	2	1
33	3	5	4	1	2
32	1	2	3	5	4
32	2	5	1	4	3
31	5	3	1	2	4

As presented, this is a 5 job - 4 machine problem.

This example follows the schematic presented in Fig 3.10.

Initially the operating parameters are outlined:

$NP = 10$
$D = 5$
$G_{\max} = 1$

For the case of illustration, the operating parameter of NP and $G_{\max}$ are kept at a minimum. The other parameters $x^{(lo)}$, $x^{(hi)}$ and D are problem dependent.

Step (1) initialises the population to the required number of solutions.

Since NP is initialised to 10, only 10 permutative solutions are generated. Table 3.2 gives the solution index which represents the positions of each value in the solution in the leading row.

The next procedure is to calculate the objective function of each solution in the population. The time flow matrix for each solution is presented. For detailed explanation on the construction of the time flow matrix, please see Section 3.6.1.

$$Solution_1 = \begin{vmatrix} 6 & 10 & 14 & 19 & 20 \\ 10 & 16 & 18 & 23 & 26 \\ 13 & 19 & 23 & 24 & 29 \\ 17 & 23 & 28 & 31 & 32 \end{vmatrix} \quad Solution_2 = \begin{vmatrix} 4 & 10 & 15 & 19 & 20 \\ 10 & 14 & 19 & 21 & 24 \\ 13 & 17 & 20 & 25 & 28 \\ 17 & 21 & 24 & 30 & 31 \end{vmatrix}$$

$$Solution_3 = \begin{vmatrix} 5 & 6 & 10 & 16 & 20 \\ 9 & 12 & 14 & 20 & 26 \\ 10 & 15 & 19 & 23 & 29 \\ 13 & 26 & 24 & 28 & 33 \end{vmatrix} \quad Solution_4 = \begin{vmatrix} 4 & 10 & 15 & 16 & 20 \\ 6 & 14 & 19 & 22 & 28 \\ 10 & 17 & 20 & 25 & 31 \\ 15 & 21 & 24 & 26 & 35 \end{vmatrix}$$

$$Solution_5 = \begin{vmatrix} 5 & 9 & 10 & 14 & 20 \\ 9 & 15 & 18 & 20 & 24 \\ 10 & 18 & 21 & 25 & 28 \\ 13 & 22 & 23 & 30 & 34 \end{vmatrix} \quad Solution_6 = \begin{vmatrix} 1 & 6 & 10 & 14 & 20 \\ 4 & 10 & 12 & 20 & 24 \\ 7 & 11 & 16 & 23 & 27 \\ 8 & 14 & 21 & 27 & 31 \end{vmatrix}$$

$$Solution_7 = \begin{vmatrix} 4 & 5 & 10 & 16 & 20 \\ 6 & 9 & 14 & 20 & 26 \\ 10 & 13 & 15 & 23 & 29 \\ 15 & 16 & 19 & 27 & 33 \end{vmatrix} \quad Solution_8 = \begin{vmatrix} 6 & 10 & 14 & 15 & 20 \\ 10 & 16 & 18 & 21 & 25 \\ 13 & 19 & 23 & 26 & 27 \\ 17 & 23 & 28 & 29 & 32 \end{vmatrix}$$

$$Solution_9 = \begin{vmatrix} 4 & 5 & 11 & 16 & 20 \\ 10 & 13 & 17 & 21 & 23 \\ 13 & 16 & 20 & 22 & 27 \\ 17 & 18 & 24 & 27 & 32 \end{vmatrix} \quad Solution_{10} = \begin{vmatrix} 1 & 5 & 11 & 15 & 20 \\ 4 & 7 & 15 & 21 & 25 \\ 7 & 11 & 18 & 24 & 26 \\ 8 & 16 & 22 & 28 & 3 \end{vmatrix}$$

The fitness of each solution is given as the last right bottom entry in each solution matrix for that particular solution. The population can now be represented as in Table 3.3.

The optimal value and its corresponding solution, for the current generation is highlighted.

Step (2) is the *forward* transformation of the solution into real numbers. Using Equation 3.3, each value in the solution is transformed. An example of the first $Solution_1 = \{1,2,3,4,5\}$ is given as an illustration:

$$x_1 = -1 + \tfrac{1 \bullet 500}{999} = -0.499 \qquad x_2 = -1 + \tfrac{2 \bullet 500}{999} = 0.001$$

$$x_3 = -1 + \tfrac{3 \bullet 500}{999} = 0.501 \qquad x_4 = -1 + \tfrac{4 \bullet 500}{999} = 1.002$$

$$x_5 = -1 + \tfrac{5 \bullet 500}{999} = 1.502$$

Table 3.4 gives the table with values in real numbers. The results are presented in 3 d.p. format.

In Step (3), DE strategies are applied to the real population in order to find better solutions.

An example of DE operation is shown. Strategy DE/rand/1/exp is used for this example: $u_{i,G+1} = x_{r_1,G} + F \bullet \left(x_{r_2,G} - x_{r_3,G}\right)$.

Table 3.4. Table of initial solutions in real number format

Solution Index	1	2	3	4	5
1	0.001	-0.499	1.002	0.501	1.502
2	-0.499	0.001	0.501	1.002	1.502
3	1.002	1.502	0.501	-0.499	0.001
4	0.501	-0.499	1.002	1.502	0.001
5	1.002	0.001	1.502	0.501	-0.499
6	1.502	1.002	0.501	0.001	-0.499
7	0.501	1.502	1.002	-0.499	0.001
8	-0.499	0.001	0.501	1.502	1.002
9	0.001	1.502	-0.499	1.002	0.501
10	1.502	0.501	-0.499	0.001	1.002

Table 3.5. Table of selected solutions

Operation	1	2	3	4	5
X1	-0.499	0.001	0.501	1.002	1.502
X2	0.501	-0.499	1.002	1.502	0.001
X3	1.502	1.002	0.501	0.001	-0.499
(X1 - X2)	-1	0.5	-0.501	-0.5	1.501
F $(X_1 - X_2)$	-0.2	0.1	-0.1002	-0.1	0.3002
X_3 + F $(X_1 - X_2)$	1.302	1.102	0.4008	-0.099	-0.1988

Three random numbers are required to index the solutions in the population given as r_1, r_2 and r_3. These numbers can be chosen as 2, 4 and 6. F is set as 0.2. The procedure is given in Table 3.5.

Table 3.6. Table of final solutions in real number format

Solution Index	1	2	3	4	5
1	-0.435	0.321	0.432	1.543	0.987
2	1.302	1.102	0.401	-0.099	-0.198
3	0.344	1.231	-2.443	-0.401	0.332
4	0.334	-1.043	1.442	0.621	1.551
5	-1.563	1.887	2.522	0.221	-0.432
6	0.221	-0.344	-0.552	0.886	-0.221
7	0.442	1.223	1.423	2.567	0.221
8	-0.244	1.332	0.371	1.421	1.558
9	0.551	0.384	0.397	0.556	0.213
10	-0.532	1.882	-0.345	-0.523	0.512

Table 3.7. Table of solutions with backward transformation

Solution Index	1	2	3	4	5
1	1.128	2.639	2.86	5.08	3.97
2	4.599	4.199	2.798	1.8	1.6
3	2.685	4.457	-2.883	1.196	2.661
4	2.665	-0.085	4.879	3.238	5.096
5	-1.124	5.768	7.036	2.439	1.134
6	2.439	1.31	0.895	3.768	1.556
7	2.881	4.441	4.841	7.126	2.439
8	1.51	4.659	2.739	4.837	5.11
9	3.098	2.765	2.791	3.108	2.423
10	0.935	5.758	1.308	0.953	3.02

Table 3.8. Rounded solutions

Solution Index	1	2	3	4	5
1	1	3	3	5	4
2	5	4	3	2	2
3	3	4	-3	1	3
4	3	-1	5	3	5
5	-1	6	7	2	1
6	2	1	1	4	2
7	3	4	5	7	2
8	2	5	3	5	5
9	3	3	3	3	2
10	1	6	1	1	3

Using the above procedure the final solution for the entire population can be given as in Table 3.6.

Backward transformation is applied to each solution in Step (4). Taking the first $Solution_1 = \{-0.435, 0.321, 0.432, 1.543, 0.987\}$, a illustrative example is given using Equation 3.5.

$$x_1 = \frac{(1+-0.435)\bullet 999}{500} = 1.128 \qquad x_2 = \frac{(1+0.001)\bullet 999}{500} = 2.639$$

$$x_3 = \frac{(1+0.501)\bullet 999}{500} = 2.86 \qquad x_4 = \frac{(1+1.002)\bullet 999}{500} = 5.08$$

$$x_5 = \frac{(1+1.502)\bullet 999}{500} = 3.97$$

The *raw* results are given in Table 3.7 with tolerance of 3 d.p.

Each value in the population is rounded to the nearest integer as given in Table 3.8.

Table 3.9. Bounded solutions

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

Table 3.10. Replucated values

Solution Index	1	2	3	4	5
1	1	**3**	**3**	5	4
2	5	4	3	**2**	**2**
3	**3**	4	**1**	**1**	**3**
4	**3**	1	**5**	**3**	**5**
5	**1**	**5**	**5**	2	**1**
6	**2**	**1**	**1**	4	**2**
7	3	4	**5**	**5**	2
8	2	**5**	3	**5**	**5**
9	**3**	**3**	**3**	**3**	2
10	**1**	5	**1**	**1**	3

Recursive mutation is applied in Step (5). For this illustration, the random mutation schema is used as this was the most potent and also the most complicated.

The first routine is to drag all bound offending values to the offending boundary. The boundary constraints are given as $x^{(lo)} = 1$ and $x^{(hi)} = 5$ which is lower and upper bound of the problem. Table 3.9 gives the *bounded* solution.

In *random mutation*, initially all the duplicated values are isolated as given in Table 3.10.

The next step is to randomly set default values for each replication. For example, in Solution 1, the value 3 is replicated in 2 indexes; 2 and 3. So a random number is generated to select the default value of 3. Let us assume that index 3 is generated. In this respect, only value 3 indexed by 2 is labelled as replicated. This routine is applied to the entire population, solution piece wise in order to set the default values.

A possible representation can be given as in Table 3.11.

Table 3.11. Ramdomly replaced values

Solution Index	1	2	3	4	5
1	1	**3**	*3*	5	4
2	5	4	3	**2**	*2*
3	**3**	4	*1*	**1**	**3**
4	**3**	1	5	*3*	*5*
5	**1**	*5*	**5**	2	*1*
6	*2*	*1*	**1**	4	**2**
7	3	4	*5*	**5**	2
8	2	**5**	3	*5*	**5**
9	**3**	**3**	*3*	**3**	2
10	*1*	5	**1**	**1**	3

Table 3.12. Missing values

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

The italicised values in Table 3.11 have been selected as default through randomisation. The next phase is to find those values which are not present in the solution. All the missing values in the solutions are given in Table 3.12.

In the case of Solutions 1, 2 and 7, it is very simple to repair the solution, since there is only one missing value. The missing value is simply placed in the replicated index for that solution. In the other cases, *positional indexes* are randomly generated. A positional index tells as to where the value will be inserted in the solution. A representation is given in Table 3.13.

Table 3.13 shows that the *first* missing value will be placed in the *second* replicated value index in the solution, and the *second* missing value will be placed in the *first* replicated index value. The final placement is given in Table 3.14.

The solutions are now permutative. The fitness for each solution is calculated in Table 3.15.

Table 3.13. Positional Index

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

Table 3.14. Final placement of missing values

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

Table 3.15. Fitness of new population

Fitness	Population				
32	1	2	3	5	4
31	5	4	3	2	1
34	5	4	1	2	3
31	2	1	4	3	5
31	3	5	4	2	1
32	2	1	5	4	3
33	3	4	5	1	2
30	2	1	3	5	4
33	1	5	3	4	2
35	1	5	4	2	3

Table 3.16. Random Index

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

Table 3.17. Fitness of new *mutated* population

Solution	Fitness	Solution Index				
		1	2	3	4	5
1	32	1	5	3	2	4
2	31	2	4	3	5	1
3	34	5	1	4	2	3
4	32	2	1	5	3	4
5	35	1	5	4	2	3
6	33	2	4	5	1	3
7	32	4	3	5	1	2
8	32	2	1	5	3	4
9	33	3	5	1	4	2
10	35	1	2	4	5	3

Table 3.18. Population after *mutation*

Fitness	Population				
32	1	2	3	5	4
31	5	4	3	2	1
34	5	4	1	2	3
31	2	1	4	3	5
31	3	5	4	2	1
32	2	1	5	4	3
32	4	3	5	1	2
30	2	1	3	5	4
33	1	5	3	4	2
35	1	5	4	2	3

Table 3.19. Random Index

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

Table 3.20. Insertion process 1

Index	1	2	3	4	5
Solution 1	1		3	5	4

Table 3.21. Insertion process 2

Index	1	2	3	4	5
Solution 1	1	3	5		4

Step (6) describes the *Standard Mutation* schema. In standard mutation, a single value swap occurs. Assume that a list of random indexes in Table 3.16 are generated which show which values are to be swapped.

It can be seen from Table 3.16, that the values indexed by 4 and 2 are to be swapped in Solution 1 and so forth for all the other solutions. The new *possible* solutions are given in Table 3.17 with their calculated fitness values. The highlighted values are the mutated values.

Only solution 7 improved in the *mutation schema* and replaces the old solution on position 7 in the population. The final population is given in Table 3.18.

Step (7), *Insertion* also requires the generation of random indexes for cascading of the solutions. A new set of random numbers can be visualized as in Table 3.19.

In Table 3.19 the values are presented in ascending order. Taking solution 1, the first process is to remove the value indexed by the first lower index (2) as shown in Table 3.20.

The second process is to move all the values from the upper index (4) to the lower index as in Table 3.21.

The last part is to insert the first removed value from the lower index into the place of the now vacant upper index aas shown in Table 3.22.

Table 3.22. Insertion process 3

Index	1	2	3	4	5
Solution 1	1	3	5	2	4

Table 3.23. Population after *insertion*

Fitness	Population				
31	**1**	**3**	**5**	**2**	**4**
31	4	3	5	2	1
32	**5**	**1**	**2**	**3**	**4**
33	1	4	3	5	2
33	3	4	2	5	1
31	**2**	**1**	**4**	**3**	**5**
33	3	5	1	4	2
32	2	1	5	4	3
33	5	3	4	1	2
34	**1**	**4**	**2**	**3**	**5**

Table 3.24. Final population

Fitness	Population				
31	1	3	5	2	4
31	4	3	5	2	1
32	5	1	2	3	4
33	1	4	3	5	2
33	3	4	2	5	1
31	2	1	4	3	5
33	3	5	1	4	2
32	2	1	5	4	3
33	5	3	4	1	2
34	1	4	2	3	5

Likewise, all the solutions are *cascaded* in the population and their new fitness calculated. The population is then represented as in Table 3.23.

After *Insertion*, four better solutions were found. These solutions replace the older solution in the population. The final population is given in Table 3.24.

DE postulates that each *child* solution replaces it direct *parent* in the population if it has better fitness. Comparing the final population in Table 3.24 with the initial population in Table 3.2, it can be seen that seven solutions produced even or better fitness than the solutions in the old population. Thus these child solutions replace the parent solutions in the population for the next generation as given in Table 3.25.

Table 3.25. Final population with fitness

Fitness	Population				
31	1	3	5	2	4
31	5	4	3	2	1
33	4	5	3	1	2
31	2	1	4	3	5
31	3	5	4	2	1
31	2	1	4	3	5
32	4	3	5	1	2
30	**2**	**1**	**3**	**5**	**4**
32	2	5	1	4	3
31	5	3	1	2	4

The new solution has a fitness of 30, which is a new fitness from the previous generation. This population is then taken into the next generation. Since we specified the $G_{\max} = 1$, only 1 iteration of the routine will take place.

Using the above outlined process, it is possible to formulate the basis for most permutative problems.

3.6 Flow Shop Scheduling

One of the common manufacturing tasks is *scheduling*. Often in most manufacturing systems, a number of tasks have to be completed on every *job*. Usually all these jobs have to follow the same route through the different *machines*, which are set up in a series. Such an environment is called a *flow shop* (FSS) [30].

The standard three-field notation [20] used is that for representing a scheduling problem as $\alpha|\beta|F(C)$, where α describes the machine environment, β describes the deviations from standard scheduling assumptions, and $F(C)$ describes the objective C being optimised. This research solves the generic flow shop problem represented as $n/m/F||F(C_{\max})$.

Stating these problem descriptions more elaborately, the minimization of completion time (makespan) for a flow shop schedule is equivalent to minimizing the objective function $\Im$:

$$\Im = \sum_{j=1}^{n} C_{m,j} \tag{3.8}$$

s.t.

$$C_{i,j} = \max\left(C_{i-1,j}, C_{i,j-1}\right) + P_{i,j} \tag{3.9}$$

where, $C_{m,j}$ = the completion time of job j, $C_{i,j} = k$ (any given value), $C_{i,j} = \sum_{k=1}^{j} C_{1,k}$; $C_{i,j} = \sum_{k=1}^{j} C_{k,1}$ machine number, j job in sequence, $P_{i,j}$ processing time of job j on

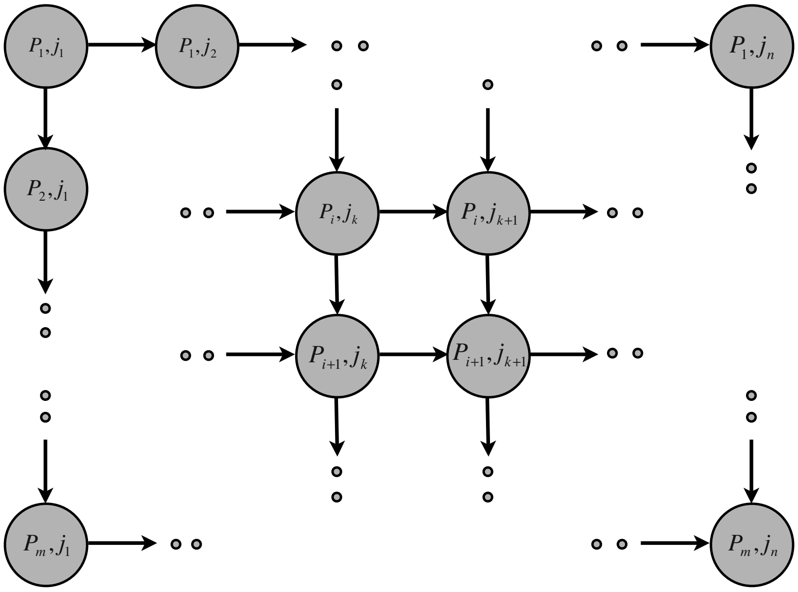

Fig. 3.10. Directed graph representation for the makespan

machine *i*. For a given sequence, the mean flow time, $MFT = \frac{1}{n}\sum_{i=1}^{m}\sum_{j=1}^{n} c_{ij}$, while the condition for tardiness is $c_{m,j} > d_j$. The constraint of Equation 3.9 applies to these two problem descriptions.

3.6.1 Flow Shop Scheduling Example

Generally, two versions of flowshop problems exist. Finding an optimal solution when the sequence changes within the schedule are flexible and changes allowed are generally harder to formulate and calculate. The schedules which are fixed are simpler to calculate and are known as permutative flow shops.

A simple representation of flowshop is given through the *directed graph method*. The *critical path* in the *directed graph* gives the makespan for the current schedule. For a given sequence $j_1, .., j_n$, the graph is constructed as follows: For each operation of a specific job j_k on a specific machine *i*, there is a node (i, j_k) with the *processing time* for that job on that machine. Node (i, j_k), $i = 1, ..., m-1$ and $k = 1, ..., n-1$, has arcs going to nodes $(i+1, j_k)$ and (i, j_{k+1}). Nodes corresponding to machine *m* have only one outgoing arc, as do the nodes in job j_n. Node (m, j_n), has no outgoing arcs as it is the terminating node and the total weight of the path from first to last node is the makespan for that particular schedule [30]. A schmetic is given in Fig 3.10.

Assume a representation of five jobs on four machines given in Table 3.26.

Given a schedule $\{1, 2, 3, 4, 5\}$ which is the schedule $\{j_1, j_2, j_3, j_4, j_5\}$, implying that all jobs in that sequence will transverse all the machines.

Table 3.26. Example of job times

jobs	j_1	j_2	j_3	j_4	j_5
P_{1,j_k}	6	4	4	5	1
P_{2,j_k}	4	6	2	4	3
P_{3,j_k}	3	3	4	1	3
P_{4,j_k}	4	4	5	3	1

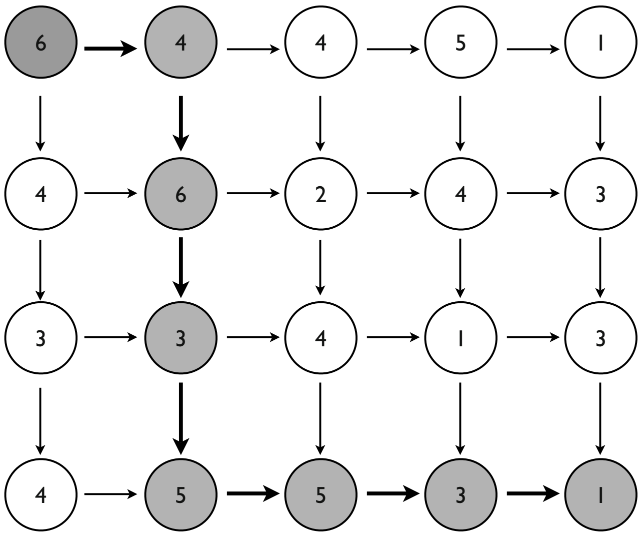

Fig. 3.11. Directed graph representation of the schedule

The directed graph representation for this schedule is given in Fig 3.11.

Each node on the graph represents the time taken to process that particular job on that particular machine. The bold lines represent the *critical path* for that particular schedule.

The Gantt chart for this schedule is represented in Fig 3.12.

The critical path is highlighted The critical path represents jobs, which are not delayed or buffered. This is important for those shops, which have machines with no buffering between them. The total time for this schedule is 34. However, from this representation, it is difficult to make out the time. A better representation of the directed graph and critical path is given in Fig 3.13.

The cumulative time nodes gives the time accumulated at each node. The final node gives the makespan for the total schedule.

The total time Gantt chart is presented in Fig 3.14.

As the schedule is changed, so does the directed graph.

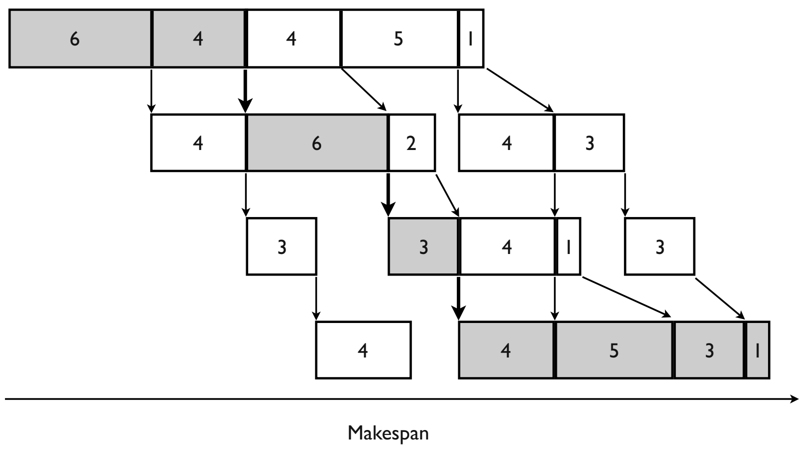

Fig. 3.12. Gantt chart representation of the schedule

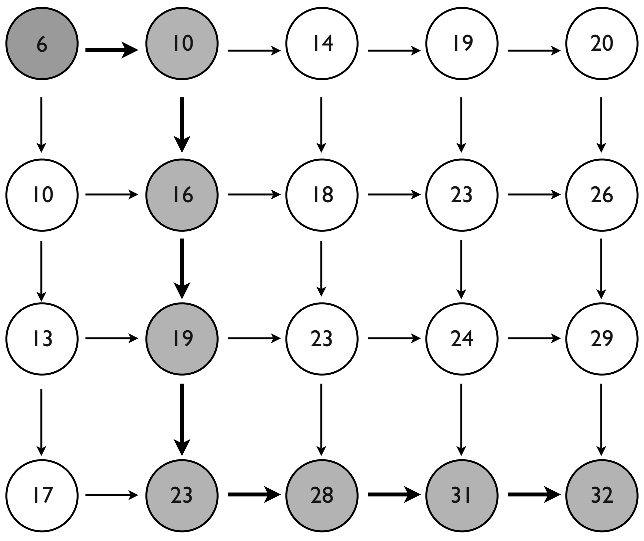

Fig. 3.13. Directed time graph and critical path

3.6.2 Experimentation for Discrete Differential Evolution Algorithm

The first phase of experimentation was used on FSS utilising DDE algorithm. Eight varying problem instances were selected from the literature, which represents a range of problem complexity. The syntax of the problem *n x m* represents *n* machines and *m* jobs. These problem instances were generated randomly for previous tests and range from small problem types (*4x4* to *15x25*), medium problem type (*20x50*) and large problem types (*25x75* and *30x100*).

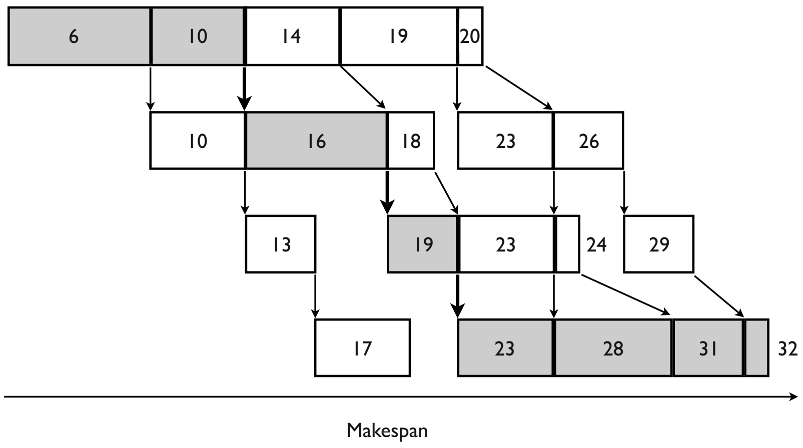

Fig. 3.14. Accumulated time Gantt Chart

Table 3.27. DDE FSS operational values

Parameter	Values
NP	150
CR	0.9
F	0.3

Table 3.28. Comparison of 10 DE-strategies using the *10x25* problem data set

	Strategy									
	1	2	3	4	5	6	7[a]	8	9	10
Makespan	211.8	209.2	212.2	212.4	208.6	210.6	207.8	212.4	210	207.2
Total tardiness	3001.8	3034.6	3021.4	3089.2	3008	2987.8	2936.4	3034.2	2982.8	2990.6
Mean flowtime	105.75	105.11	105.52	107.71	104.68	103.03	103.17	105.32	104.7	104.16

[a]Strategy 7 is the best.

In order to operate, the first phase is to obtain the optimal tuning parameters. All parameters were obtained empirically. The values are given in Table 3.27.

The second phase was to obtain the optimal strategy. Through experience in solving these problems, it became evidently clear that not all the strategies behaved similarly, hence the need to isolate the most promising one from the ten different.

An arbitrary problem of average difficulty was selected, in this case the *10x25* job problem, and using the selected parameters, ten iterations were done. The average values are presented in Table 3.28. Using the multi-objective function of makespan, tardiness and flowtime, Strategy 7 was selected as the optimal.

Table 3.29. DDE FSS makespan

m x *n*	Generated problems	GA	DE	(Solution) GA/DE
4x4	5	44	39	-
5x10	5	79	79	-
8x15	5	143	138	-
10x25	5	205	202	-
15x25	5	248	253	98.02
20x50	5	468	470	99.57
25x75	5	673	715.4	94.07
30x100	5	861	900.4	95.62

Table 3.30. DDE FSS total tardiness

m x *n*	Generated problems	GA	DE	(Solution) GA/DE
4x4	5	54	52.6	-
5x10	5	285	307	92.83
8x15	5	1072	1146	93.54
10x25	5	2869	2957	97.02
15x25	5	3726	3839.4	97.06
20x50	5	13683	14673.6	93.25
25x75	5	30225	33335.6	90.67
30x100	5	51877	55735.6	93.07

With all the experimentation parameters selected, the FSS problems were evaluated. Three different objective functions were to be analysed. The first was the makespan. The makespan is equivalent to the completion time for the last job to leave the system. The results are presented in Table 3.29.

The second objective is the tardiness. Tardiness relates to the number of tardy jobs; jobs which will not meet their due dates and which are scheduled last. This reflects the on-time delivery of jobs and is of paramount importance to production planning and control [30]. The results are given in Table 3.30.

The final objective is the mean flowtime of the system. It is the sum of the weighted completion time of the *n* jobs which gives an indication of the total holding or inventory costs incurred by the schedule. The results are presented in Table 3.31.

Tables 3.29 – 3.31 show the comparison between Genetic Algorithm (GA) developed in a previous study for flowshop scheduling [28], compared with DDE. Upon analysis it is seen that, DE algorithm performs better than GA for small-sized problems, and competes appreciably with GA for medium to large-sized problems. These results are not compared to the traditional methods since earlier study of [4] show that GA based algorithm for flow shop problems outperform the best existing traditional approaches such as the ones proposed by [16] and [39].

Table 3.31. DDE Mean Flowtime

m x *n*	Generated problems	GA	DE	(Solution) GA/DE
4x4	5	21.38	22.11	-
5x10	5	35.3	36.34	97.14
8x15	5	63.09	66.41	95
10x25	5	98.74	103.89	95.04
15x25	5	113.85	122.59	93.03
20x50	5	216	234.32	92.18
25x75	5	317	354.77	89.35
30x100	5	399.13	435.49	91.56

Table 3.32. EDE FSS operational values

Parameter	Values
Strategy	9
NP	150
CR	0.3
F	0.1

These obtained results formed the basic for the enhancement of DDE. It should be noted that even with a very high percentage of in-feasible solutions obtained, DDE managed to outperform GA.

3.6.3 Experimentation for Enhanced Differential Evolution Algorithm

The second phase of experiments outline experimentation of EDE to FSS. As with the DDE, operational parameters were empirically obtained as given in Table 3.32. As can be noticed the parameters are very different from those used in DDE for the same problems. This is attributed to the new routines added to DDE which adds another layer of stochastically to EDE.

The first section of experimentation was conducted on the same group of FSS problems as GA and DDE to obtain comparison results. In this respect, only makespan was evaluated. For all the problem instances, EDE performs optimally compared to the other two heuristics. Columns 5 to 7 in Table 3.33 gives the effectiveness comparisons of EDE, DDE and GA, with EDE outperforming both DDE and GA.

With the validation completed for EDE, more extensive experimentation was conducted to test its complete operational range in FSS.

The second set of benchmark problems is from the three papers of [3], [33] and [15]. All these problem sets are available in the OR Library [29]. The EDE results are compared with the optimal values reported for these problems as given in Table 3.34. The conversion is given in Equation 3.10:

$$\Delta = \frac{(H-U)\bullet 100}{U} \tag{3.10}$$

where H represents the obtained value and U is the reported optimal. For the Car and Hel set of problems, EDE easily obtains the optimal values, and on average around 1% above the optimal for the reC instances.

Table 3.33. FSS comparison

	DDE	GA	EDE	% DDE−GA	% EDE−DDE	% EDE−GA
F 5 x 10	79.4	-	78	-	101.79	-
F 8 x 15	138.6	143	134	103.17	103.43	106.71
F 10 x 25	207.6	205	194	98.74	107.01	105.67
F 15 x 25	257.6	248	240	96.27	107.33	103.33
F 20 x 50	474.8	468	433	98.56	109.65	108.08
F 25 x 75	715.4	673	647	94.07	110.57	104.01
F 30 x 100	900.4	861	809	95.62	111.29	106.42
Ho Chang	213	213	213	100	100	100

Table 3.34. Comparison of FSS instances

Instance	Size	Optimal	EDE	% to Optimal
Car 1	11 x 5	7038	7038	0
Car 2	13 x 4	7166	7166	0
Car 3	12 x 5	7312	7312	0
Car 4	14 x 4	8003	8003	0
Car 5	10 x 6	7720	7720	0
Car 6	8 x 9	8505	8505	0
Car 7	7 x 7	6590	6590	0
Car 8	8 x 8	8366	8366	0
Hel 2	20 x 10	135	135	0
reC 01	20 x 5	1247	1249	0.16
reC 03	20 x 5	1109	1111	0.18
reC 05	20 x 5	1242	1249	0.56
reC 07	20 x 10	1566	1584	1.14
reC 09	20 x 10	1537	1574	2.4
reC 11	20 x 10	1431	1464	2.3
reC 13	20 x 15	1930	1957	1.39
reC 15	20 x 15	1950	1984	1.74
reC 17	20 x 15	1902	1957	2.89
reC 19	30 x 10	2093	2132	1.86
reC 21	30 x 10	2017	2065	2.37
reC 23	30 x 10	2011	2073	3.08

Table 3.35. EDE comparison with DE_{spv} and PSO over the Taillard benchmark problem

	GA		PSO_{spv}		DE_{spv}		$DE_{spv+exchange}$		*EDE*	
	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	0.98	0.66
20x10	5.42	1.72	3.28	1.19	3.71	1.24	2.01	0.93	1.81	0.77
20x20	4.22	1.31	2.84	1.15	3.03	0.98	1.85	0.87	1.75	0.57
50x5	1.69	0.79	1.15	0.7	0.88	0.52	0.41	0.37	0.4	0.36
50x10	5.61	1.41	4.83	1.16	4.12	1.1	2.41	0.9	3.18	0.94
50x20	6.95	1.09	6.68	1.35	5.56	1.22	3.59	0.78	4.05	0.65
100x5	0.81	0.39	0.59	0.34	0.44	0.29	0.21	0.21	0.41	0.29
100x10	3.12	0.95	3.26	1.04	2.28	0.75	1.41	0.57	1.46	0.36
100x20	6.32	0.89	7.19	0.99	6.78	1.12	3.11	0.55	3.61	0.36
200x10	2.08	0.45	2.47	0.71	1.88	0.69	1.06	0.35	0.95	0.18

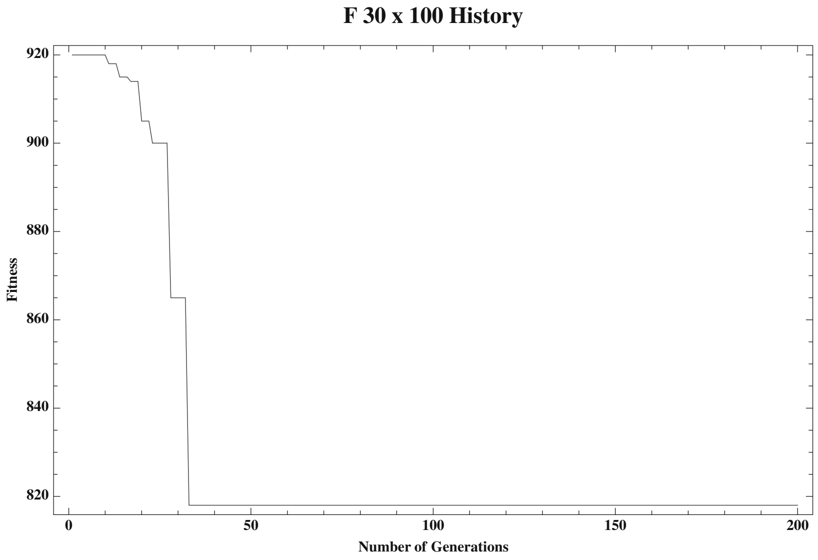

Fig. 3.15. Sample output of the F30x100 FSS problem.

The third experimentation module is referenced from [37]. These sets of problems have been extensively evaluated (see [22, 34]). This benchmark set contains 100 particularly hard instances of 10 different sizes, selected from a large number of randomly generated problems.

A maximum of ten iterations was done for each problem instance. The population was kept at 100, and 100 generations were specified. The results represented in Table 3.35, are as quality solutions with the percentage relative increase in makespan with respect to the upper bound provided by [37] as given by Equation 3.10.

The results obtained are compared with those produced by GA, Particle Swarm Optimisation (PSO_{spv}) DE (DE_{spv}) and DE with local search ($DE_{spv+exchange}$) as in [38]. The results are tabulated in Table 3.35.

It can be observed that EDE compares outstandingly with other algorithms. EDE basically outperforms GA, PSO and DE_{spv}. The only serious competition comes from the new variant of $DE_{spv+exchange}$. EDE and $DE_{spv+exchange}$ are highly compatible. EDE outperforms $DE_{spv+exchange}$ on the data sets of 20x10, 20x20, 50x5 and 200x5. In the remainder of the sets EDE performs remarkbly to the values reported by $DE_{spv+exchange}$. On average EDE displays better standard deviation than that of $DE_{spv+exchange}$. This validates the consistency of EDE compared to $DE_{spv+exchange}$. It should be noted that $DE_{spv+exchange}$ utilises local search routine as its search engine.

A sample generation for the *F 30 x 100* FSS problem is given in Fig 3.15.

3.7 Quadratic Assignment Problem

The second class of problems to be conducted by EDE was the *Quadratic Assignment Problem* (QAP). QAP is a *NP*-hard optimisation problem [35] which was stated for the first time by [18]. It is considered as one of the hardest optimisation problems as general instances of size $n \geq 20$ cannot be solved to optimally [10].

It can be described as follows: Given two matrices

$$A = (a_{ij}) \tag{3.11}$$

$$B = (b_{ij}) \tag{3.12}$$

find the permutation π^* minimising

$$\min_{\pi \in \prod(n)} f(\pi) = \sum_{i=1}^{n} \sum_{j=1}^{n} a_{ij} \bullet b_{\pi(i)\pi(j)} \tag{3.13}$$

where $\prod(n)$ is a set of permutations of n elements.

The problem instances selected for the QAP are from the OR Library [29] and reported in [13]. There are two separate problem modules; *regular* and *irregular*.

The difference between regular and irregular problems is based on the *flow−dominance*. Irregular problems have a flow−dominance statistics larger than 1.2. Most of the problems come from practical applications or have been randomly generated with non-uniform laws, imitating the distributions observed in real world problems.

In order to differentiate among the classes of QAP instances, the flow dominance *fd* is used. It is defined as a coefficient of variation of the flow matrix entries multiplied by 100. That is:

$$fd = \frac{100\sigma}{\mu} \tag{3.14}$$

where:

$$\mu = \frac{1}{n^2} \bullet \sum_{i=1}^{n} \sum_{j=1}^{n} b_{ij} \tag{3.15}$$

$$\sigma = \sqrt{\frac{1}{n^2} \bullet \sum_{i=1}^{n} \sum_{j=1}^{n} (b_{ij} - \mu)^2} \tag{3.16}$$

3.7.1 Quadratic Assignment Problem Example

As example for the QAP is given as the faculty location problem given in Fig 3.16.

The objective is to allocate location to faculties. There is a specific distance between the faculties, and there is a specified flow between the different faculties, as shown by the thickness of the lines. An arbitrary schedule can be $\{2,1,4,3\}$, as given in Fig 3.16. Two distinct matrices are required: one *distance* and one *flow* matrix as given in
Tables 3.36 and 3.37.

Applying the QAP formula, the function becomes:

$$Sequence = \begin{Bmatrix} D(1,2) \bullet F(1,2) + \\ D(1,3) \bullet F(2,4) + \\ D(2,3) \bullet F(1,4) + \\ D(3,4) \bullet F(3,4) \end{Bmatrix}$$

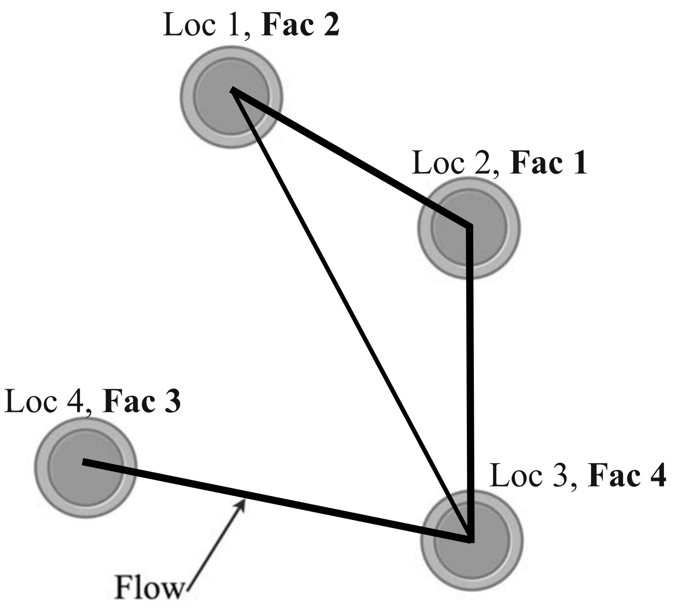

Fig. 3.16. Faculty location diagram for 2, 1, 4, 3

Table 3.36. Distance Matrix

Distance	Value
D(1,2)	22
D(1,3)	53
D(2,3)	40
D(3,4)	55

Table 3.37. Flow Matrix

Flow	Value
F(2,4)	1
F(1,4)	2
F(1,2)	3
F(3,4)	4

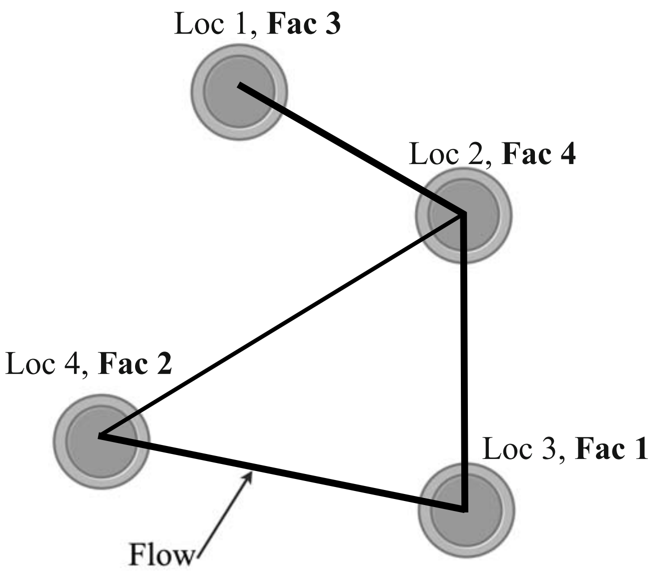

Fig. 3.17. Faculty location diagram for 3, 4, 1, 2

$$Cost = \left\{ \begin{array}{l} (22 \bullet 3) + \\ (53 \bullet 1) + \\ (40 \bullet 2) + \\ (55 \bullet 4) \end{array} \right\} = 419$$

Now, assume a different permutation: $\{3, 4, 1, 2\}$. The faculty location diagram is now given in Fig 3.17.

The solution for this permutation is 395, The flow matrix remains the same, and only the distance matrix changes to reflect the new faculty location.

3.7.2 Experimentation for Irregular QAP

The first phase as with FSS, was to empirically obtain the operational values as given in Table 3.38. These values were used for both regular and irregular instances.

The first set of experimentations was on irregular instances. These are those with flow dominance of greater than 1.

The results are presented in Table 3.39. The results are presented as the factor distance from the optimal: $\Delta = \frac{(H-U)}{U}$; where H is the obtained result and U is the optimal.

Table 3.38. EDE QAP operational values

Parameter	Value
Strategy	1
CR	0.9
F	0.3

Table 3.39. EDE Irregular QAP comparison

Instant	flow dom	n	Optimal	TT	RTS	SA	GH	HAS-QAP	EDE
bur26a	2.75	26	5246670	0.208	-	0.1411	0.012	0	0.006
bur26b	2.75	26	3817852	0.441	-	0.1828	0.0219	0	0.0002
bur26c	2.29	26	5426795	0.17	-	0.0742	0	0	0.00005
bur26d	2.29	26	3821225	0.249	-	0.0056	0.002	0	0.0001
bur26e	2.55	26	5386879	0.076	-	0.1238	0	0	0.0002
bur26f	2.55	26	3782044	0.369	-	0.1579	0	0	0.000001
bur26g	2.84	26	10117172	0.078	-	0.1688	0	0	0.0001
bur26h	2.84	26	7098658	0.349	-	0.1268	0.0003	0	0.0001
chr25a	4.15	26	3796	15.969	16.844	12.497	2.6923	3.0822	0.227
els19	5.16	19	17212548	21.261	6.714	18.5385	0	0	0.0007
kra30a	1.46	30	88900	2.666	2.155	1.4657	0.1338	0.6299	0.0328
kra30b	1.46	30	91420	0.478	1.061	1.065	0.0536	0.0711	0.0253
tai20b	3.24	20	122455319	6.7	-	14.392	0	0.0905	0.0059
tai25b	3.03	25	344355646	11.486	-	8.831	0	0	0.003
tai30b	3.18	30	637117113	13.284	-	13.515	0.0003	0	0.0239
tai35b	3.05	35	283315445	10.165	-	6.935	0.1067	0.0256	0.0101
tai40b	3.13	40	637250948	9.612	-	5.43	0.2109	0	0.027
tai50b	3.1	50	458821517	7.602	-	4.351	0.2124	0.1916	0.001
tai60b	3.15	60	608215054	8.692	-	3.678	0.2905	0.0483	0.0144
tai80b	3.21	80	818415043	6.008	-	2.793	0.8286	0.667	0.0287

The comparison is done with Tabu Search (TT) [36], Reative Tabu Search (RTS) [1], Simulated Annealing (SA) [5], Genetic Hybrid (GH) [2] and Hybrid Ant Colony (HAS) [13].

Two trends are fairly obvious. The first is that for bur instances, HAS obtains the optimal, and is very closely followed by EDE by a margin of only 0.001 on average. For the *tai* instances, EDE competes very well, obtaining the best values for the larger problems and also obtains the best values for the *kra* problems. TT and RTS are shown to be not well adapted to irregular problems, producing 10% worse solution at times. GH which does not have memory retention capabilities does well, but does not produce optimal results with any regularity.

3.7.3 Experimentation for Regular QAP

The second section of QAP problems is discussed in this section. This is the set of regular problem as discussed in [13]. Regular problems are distinguished as having a flow−dominance of less than 1.2.

Comparison was done with the same heuristics as in the previous section. The results are presented in Table 3.40.

Three different set of instances are presented: *nug*, *sko*, *tai* and *wil*. Apart for the *nug20* instance, EDE finds the best solutions for all the reported instances. It can be observed that TT, GH and SA perform best for *sko* problems and RTS performs best for *tai* problems. On comparison with the optimal values, EDE obtains values with tolerance of only 0.01 on average for all instances.

A sample generation for *Bur26a* problem is given in Fig 3.18.

Table 3.40. EDE Regular QAP comparison

Instant	flow dom	n	Optimal	TT	RTS	SA	GH	HAS-QAP	EDE
nug20	0.99	20	2570	0	0.911	0.07	0	0	0.018
nug30	1.09	30	6124	0.032	0.872	0.121	0.007	0.098	0.005
sko42	1.06	42	15812	0.039	1.116	0.114	0.003	0.076	0.009
sko49	1.07	49	23386	0.062	0.978	0.133	0.04	0.141	0.009
sko56	1.09	56	34458	0.08	1.082	0.11	0.06	0.101	0.012
sko64	1.07	64	48498	0.064	0.861	0.095	0.092	0.129	0.013
sko72	1.06	72	66256	0.148	0.948	0.178	0.143	0.277	0.011
sko81	1.05	81	90998	0.098	0.88	0.206	0.136	0.144	0.011
tai20a	0.61	20	703482	0.211	0.246	0.716	0.628	0.675	0.037
tai25a	0.6	25	1167256	0.51	0.345	1.002	0.629	1.189	0.026
tai30a	0.59	30	1818146	0.34	0.286	0.907	0.439	1.311	0.018
tai35a	0.58	35	2422002	0.757	0.355	1.345	0.698	1.762	0.038
tai40a	0.6	40	3139370	1.006	0.623	1.307	0.884	1.989	0.032
tai50a	0.6	50	4941410	1.145	0.834	1.539	1.049	2.8	0.033
tai60a	0.6	60	7208572	1.27	0.831	1.395	1.159	3.07	0.037
tai80a	0.59	80	13557864	0.854	0.467	0.995	0.796	2.689	0.031
wil50	0.64	50	48816	0.041	0.504	0.061	0.032	0.061	0.004

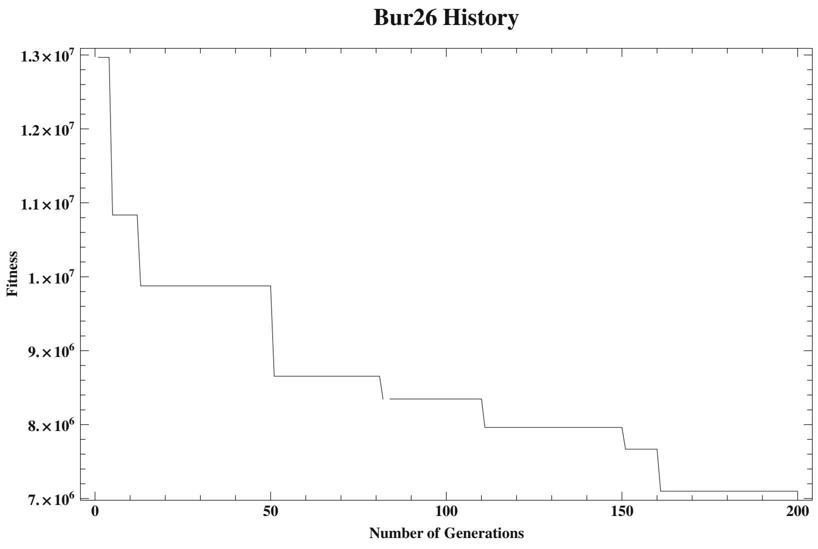

Fig. 3.18. Sample output of the Bur26a problem

3.8 Traveling Salesman Problem

The third and final problem class to be experimented is the *Traveling Salesman Problem* (TSP). The TSP is a very well known optimisation problem. A traveling salesman has a number, *N*, cities to visit. The sequence in which the salesperson visits different cities is called a *tour*. A tour is such that every city on the list is visited only once, except that the salesperson returns to the city from which it started. The objective to is minimise the total distance the salesperson travels, among all the tours that satisfy the criterion.

Several mathematical formulations exist for the TSP. One approach is to let x_{ij} be 1 if city *j* is visited immediately after *i*, and be 0 if otherwise [24, 25]. The formulation of TSP is given in Equations 3.17 to 3.20.

$$\min \sum_{i=1}^{N} \sum_{j=1}^{N} c_{ij} \bullet x_{ij} \tag{3.17}$$

Each city is left after visiting subject to

$$\sum_{j=1}^{N} x_{ij} = 1; \forall i \tag{3.18}$$

Ensures that each city is visited

$$\sum_{i=1}^{N} x_{ij} = 1; \forall j \tag{3.19}$$

No subtours

$$x_{ij} = 0 \quad \text{or} \quad 1 \tag{3.20}$$

Table 3.41. City distance matrix

City	A	B	C	D
E	2	3	2	4
D	1	5	1	
C	2	3		
B	1			

No subtours mean that there is no need to return to a city before visiting all the other cities. The objective function accumulates time as you go from city i to j. Constraint 3.18 ensures that the salesperson leaves each city. Constraint 3.19 ensures that the salesperson enters each city. A subtour occurs when the salesperson returns to a city prior to visiting all other cities. Restriction 3.20 enables the TSP formulation differs from the Linear Assignment Problem programming (LAP) formulation.

3.8.1 Traveling Salesman Problem Example

Assume there are five cities $\{A,B,C,D,E\}$, for a traveling salesman to visit as shown in Fig 3.19. The distance between each city is labelled in the vertex.

In order to understand TSP, assume a tour, where a salesman travels through all the cities and returns eventually to the original city. Such a tour can be given as $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow A$. The graphical representation for such a tour is given in Fig 3.20.

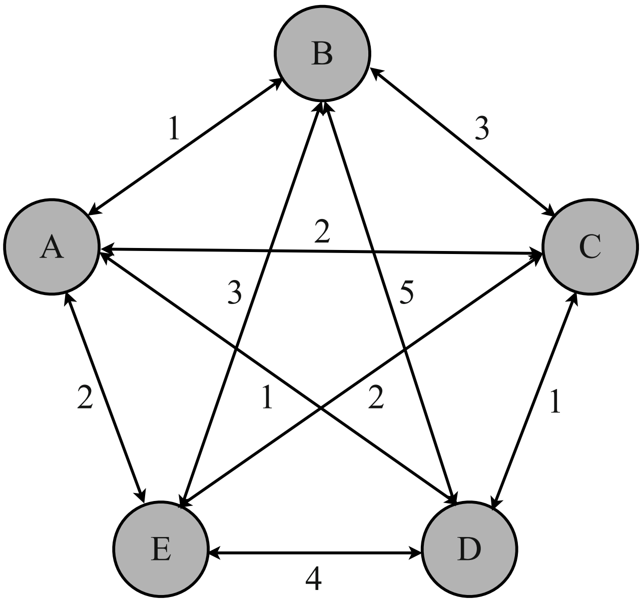

Fig. 3.19. TSP distance node graph

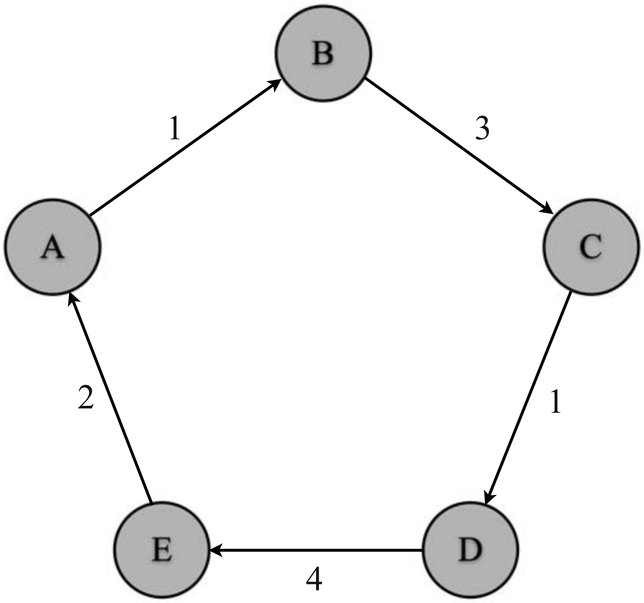

Fig. 3.20. Graphical representation for the tour $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow A$

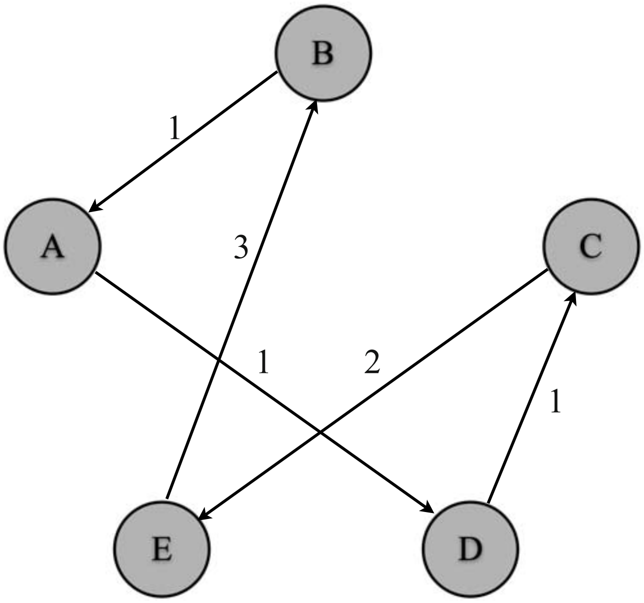

Fig. 3.21. Graphical representation for the tour $A \rightarrow D \rightarrow C \rightarrow E \rightarrow B \rightarrow A$

The total cost for this tour is 11.

The objective of TSP optimisation is to find a tour with the minimal value. Assume now another tour $A \rightarrow D \rightarrow C \rightarrow E \rightarrow B \rightarrow A$. The graphical representation is given in Fig 3.21.

The cost for this new tour is 8, which is a decrease from the previous tour of 11. This is now an improved tour. Likewise many other tours can be found which have better values.

3.8.2 Experimentation on Symmetric TSP

Symmetric TSP problem is one, where the distance between two cities is the same *to* and *fro*. This is considered the easiest branch of TSP problem.

The operational parameters for TSP is given in Table 3.42.

Experimentation was conducted on the *City* problem instances. These instances are of 50 cities and the results are presented in Table 3.43. Comparison was done with Ant Colony (ACS) [11], Simulated Annealing (SA) [21], Elastic Net (EN) [12], and Self Organising Map (SOM) [17]. The time values are presented alongside.

In comparison, ACS is the best performing heuristic for TSP. EDE performs well, with tolerance of 0.1 from the best performing heuristics on average.

3.8.3 Experimentation on Asymmetric TSP

Asymmetric TSP is the problem where the distance between the different cities is different, depending on the direction of travel. Five different instances were evaluated and compared with Ant Colony (ACS) with local search [11]. The experimetational results are given in Table 3.44.

Table 3.42. EDE TSP operational values

Parameter	Value
Strategy	9
CR	0.9
F	0.1

Table 3.43. EDE STSP comparison

Instant	ACS (average)	SA (average)	EN (average)	SOM (average)	EDE (average)
City set 1	5.88	5.88	5.98	6.06	5.98
City set 2	6.05	6.01	6.03	6.25	6.04
City set 3	5.58	5.65	5.7	5.83	5.69
City set 4	5.74	5.81	5.86	5.87	5.81
City set 5	6.18	6.33	6.49	6.7	6.48

Table 3.44. EDE ATSP comparison

Instant	Optimal	ACS 3-OPT best	ACS 3-OPT average	EDE
p43	5620	5620	5620	5639
ry48p	14422	14422	14422	15074
ft70	38673	38673	38679.8	40285
kro124p	36230	36230	36230	41180
ftv170	2755	2755	2755	6902

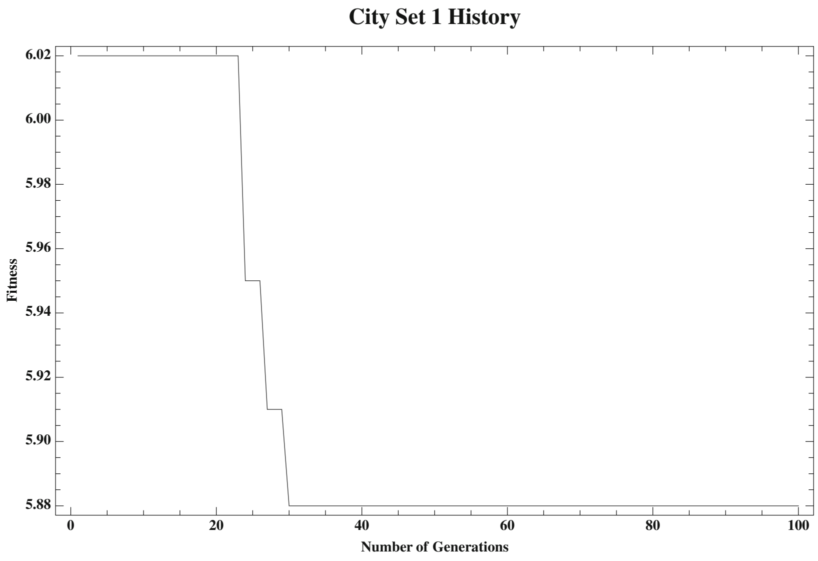

Fig. 3.22. Sample output of the City set 1 problem

ACS heuristic performs very well, obtaining the optimal value, whereas EDE has an average performance. The difference is that ACS employs 3−Opt local search on each generation of its best solution, where as EDE has a 2−Opt routine valid only in local optima stagnation.

A sample generation for *City set 1* problem is given in Fig 3.22.

3.9 Analysis and Conclusion

One the few ways in which the validation of a permutative approach for a real domain based heuristic can be done is empirically; through expensive experimentation's across different problem classes, as attempted here. Through the results obtained, it can be stated that EDE is a valid approach for permutative problems. One of the differing evident features, is that the operating parameters for each class of problems is unique. No definite conclusions can be made on this aspect, apart from the advise for simulations for tuning.

Another important feature of EDE is the level of stochasticity. DE has two levels; first the initial population and secondly the crossover. EDE has five; in addition to the two mentioned, the third is repairment, the fourth is mutation and fifth is crossover. All these three are embedded on top of the DE routine, so the DE routines are a directive search guide with refinement completed in the subsequent routines.

Local search was included in EDE because permutative problems usually require triangle inequality routines. TSP is notorious in this respect, and most heuristics have to employ local search in order to find good solutions. ACS [11], Scatter Search [14] apply local search on each and every solution. This increases computational time and reduces effectiveness of the heuristic for practical applications. The idea of EDE was to only employ local search when stagnation is detected, and to employ the simplest and time economical one.

In terms of produced results, EDE is effective, and more so since it was left in non-altered form for all the problem classes. This is a very important feature since it negates re-programming for other problem instances. Another important feature is that EDE is fairly fast for these problems. Naturally, the increase in problem size increases the execution time, however EDE does not employ any analytical formulation within its heuristic, which keeps down the execution time while producing the same results as with other heuristics.

It is hoped that the basic framework of this approach will be improved to include more problem instances, like Job Shop Scheduling and other manufacturing scheduling problems.

References

1. Battitti, R., Tecchiolli, G.: The reactive tabu search. ORCA Journal on Computing 6, 126–140 (1994)
2. Burkard, R., Rendl, F.: A thermodynamically motivated simulation procedure for combinatorial optimisation problems. Eur. J. Oper. Res. 17, 169–174 (1994)
3. Carlier, J.: Ordonnancements a Contraintes Disjonctives. RAIRO. Oper. Res. 12, 333–351 (1978)
4. Chen, C., Vempati, V., Aljaber, N.: An application of genetic algorithms for the flow shop problems. Eur. J. Oper. Res. 80, 359–396 (1995)
5. Connolly, D.: An improved annealing scheme for the QAP. Eur. J. Oper. Res. 46, 93–100 (1990)
6. Davendra, D.: Differential Evolution Algorithm for Flow Shop Scheduling, Bachelor Degree Thesis, University of the South Pacific (2001)
7. Davendra, D.: Hybrid Differential Evolution Algorithm for Discrete Domain Problems. Master Degree Thesis, University of the South Pacific (2003)
8. Davendra, D., Onwubolu, G.: Flow Shop Scheduling using Enhanced Differential Evolution. In: Proceeding of the 21st European Conference on Modelling and Simulation, Prague, Czech Republic, June 4-5, pp. 259–264 (2007)
9. 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)

10. Dorigo, M., Maniezzo, V., Colorni, A.: The Ant System: optimisation by a colony of co-operating agents. IEEE Trans. Syst. Man Cybern B Cybern. 26(1), 29–41 (1996)
11. Dorigo, M., Gambardella, L.: Ant Colony System: A Co-operative Learning Approach to the Traveling Salesman Problem. IEEE Trans. Evol. Comput. 1, 53–65 (1997)
12. Durbin, R., Willshaw, D.: An analogue approach to the travelling salesman problem using the elastic net method. Nature 326, 689–691 (1987)
13. Gambardella, L., Thaillard, E., Dorigo, M.: Ant Colonies for the Quadratic Assignment Problem. Int. J. Oper. Res. 50, 167–176 (1999)
14. Glover, F.: A template for scatter search and path relinking. In: Hao, J.-K., Lutton, E., Ronald, E., Schoenauer, M., Snyers, D. (eds.) AE 1997. LNCS, vol. 1363, pp. 13–54. Springer, Heidelberg (1998)
15. Heller, J.: Some Numerical Experiments for an MJ Flow Shop and its Decision- Theoretical aspects. Oper. Res. 8, 178–184 (1960)
16. Ho, Y., Chang, Y.-L.: A new heuristic method for the n job, m - machine flow-shop problem. Eur. J. Oper. Res. 52, 194–202 (1991)
17. Kara, L., Atkar, P., Conner, D.: Traveling Salesperson Problem (TSP) Using Shochastic Search. In: Advanced AI Assignment, Carnegie Mellon Assignment, Pittsbergh, Pennsylvania, p. 15213 (2003)
18. Koopmans, T., Beckmann, M.: Assignment problems and the location of economic activities. Econometrica 25, 53–76 (1957)
19. Lampinen, J., Zelinka, I.: Mechanical engineering design optimisation by Differential evolution. In: Corne, D., Dorigo, M., Glover, F. (eds.) New Ideas in Optimisation, pp. 127–146. McGraw Hill, International, UK (1999)
20. Lawler, E., Lensta, J., Rinnooy, K., Shmoys, D.: Sequencing and scheduling: algorithms and complexity. In: Graves, S., Rinnooy, K., Zipkin, P. (eds.) Logistics of Production and Inventory, pp. 445–522. North Holland, Amsterdam (1995)
21. Lin, F., Kao, C., Hsu: Applying the genetic approach to simulated annealing in solving NP-hard problems. IEEE Trans. Syst. Man Cybern. B Cybern. 23, 1752–1767 (1993)
22. Nowicki, E., Smutnicki, C.: A fast tabu search algorithm for the permutative flow shop problem. Eur. J. Oper. Res. 91, 160–175 (1996)
23. Onwubolu, G.: Optimisation using Differential Evolution Algorithm. Technical Report TR-2001-05, IAS (October 2001)
24. Onwubolu, G.: Emerging Optimisation Techniques in Production Planning and Control. Imperial Collage Press, London (2002)
25. Onwubolu, G., Clerc, M.: Optimal path for automated drilling operations by a new heuristic approach using particle swamp optimisation. Int. J. Prod. Res. 42(3), 473–491 (2004)
26. Onwubolu, G., Davendra, D.: Scheduling flow shops using differential evolution algorithm. Eur. J. Oper. Res. 171, 674–679 (2006)
27. Onwubolu, G., Kumalo, T.: Optimisation of multi pass tuning operations with genetic algorithms. Int. J. Prod. Res. 39(16), 3727–3745 (2001)
28. Onwubolu, G., Mutingi, M.: Genetic algorithm for minimising tardiness in flow-shop scheduling. Prod. Plann. Contr. 10(5), 462–471 (1999)
29. Operations Reserach Library (Cited September 13, 2008), http://people.brunel.ac.uk/~mastjjb/jeb/info.htm
30. Pinedo, M.: Scheduling: theory, algorithms and systems. Prentice Hall, Inc., New Jersey (1995)
31. 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)
32. Price, K., Storn, R.: Differential evolution homepage (2001) (Cited September 10, 2008), http://www.ICSI.Berkeley.edu/~storn/code.html

33. Reeves, C.: A Genetic Algorithm for Flowshop Sequencing. Comput. Oper. Res. 22, 5–13 (1995)
34. Reeves, C., Yamada, T.: Genetic Algorithms, path relinking and flowshop sequencing problem. Evol. Comput. 6, 45–60 (1998)
35. Sahni, S., Gonzalez, T.: P-complete approximation problems. J. ACM 23, 555–565 (1976)
36. Taillard, E.: Robust taboo search for the quadratic assignment problem. Parallel Comput. 17, 443–455 (1991)
37. Taillard, E.: Benchmarks for basic scheduling problems. Eur. J. Oper. Res. 64, 278–285 (1993)
38. 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)
39. Widmer, M., Hertz, A.: A new heuristic method for the flow shop sequencing problem. Eur. J. Oper. Res. 41, 186–193 (1989)