Tomas Bata University in Zlin

Faculty of Applied Informatics

Donald Davendra

Chaotic Attributes and Permutative Optimization

Doctoral Thesis

Study-branch: Technical Cybernetics

Supervisors: Assoc. Prof. Ivan Zelinka, *Ph.D*.
Prof. Godfrey Onwubolu, *Ph.D*.

Zlin, Czech Republic, 2009

Contents

List of Figures

List of Tables

Acknowledgements

This thesis has been a 5 years odyssey, which has taken me from the small and remote islands of Fiji, to the Czech Republic, the heart of Europe, half way across the world.

My eternal gratitude will remain with **Prof. Ivan Zelinka**, who during my darkest hour, showed that the fellowship of man still remains in this world. Through him I met a supervisor, a mentor, a friend, and a person who always helped me regardless of the situation I found myself in. This thesis is as much a reflection to his tireless supervison as it is of my research skills.

My gratitude goes to **Prof. Godfrey Onwubolu**, who introduced me to the nobility of Ph.D, as a “contribution to the knowledge of mankind”. His faith and dedication to the cause of research, motivated me to pursue the same path. His herculean efforts to keep in doctoral studies will never be forgotten.

To my parents, **Michael** and **Manjula Davendra** and my sister **Annjelyn Shalvina**, who perservered through so much hardship, I dedicate this thesis to you.

Finally to my darling wife, **Mgr. Magdalena Bialic-Davendra**, who through her perserverence and love, held me together through these 3 years, and made my life a much brighter place.

Abstract

Diversity in evolutionary systems and its application to permutative based combinatorial optimization problems is the core objective of this dissertation.

Stagnation and its implication through chaotic attributes is outlined and new attack strategies are developed to induce viabilty to canonical metaheuristics.

Three new permutative versions of Self Organising Migrating Algorithm (SOMA) are developed, being the Permutative Set Handling, Static Permutative SOMA and Dynamic Permutative SOMA.

Novel clustered population paradigms based loosely around the concept of chaotic attractors and edges are developed and utilised through Differential Evolution (DE) and SOMA. New selection and deletion criteria's are developed and vetted with the canonical algorithms.

Six unique and challenging permutative based combinatorial optimization problems are solved using these heuristics with good results obtained.

Chapter 1

Introduction

One of the most challenging optimization problems is *permutative based* combinatorial optimization. This class of problem harbours some of the most famous optimization problems like travelling salesman and vehicle routing problem.

Another very important branch is that of scheduling, to which a number of manufacturing problems are associated. The most realised and of interest are the shop scheduling problems of flow shop and job shop.

What makes a permutative problem complex is that the solution representation is very concise, since it must have a discrete number of values, and each occupied variable in the solution is unique. Given a problem of size n, a representation can be described as $x = \{x_1, x_2, x_3, ..., x_n\}$, where each value x_i in the solution is unique and the entire set of solutions is an integer representation from $\{1, n\}$.

From an optimization point of view, this represents a number of problems. Firstly, the search space is discrete and a number of validations inevitably have to be conducted in order to have a viable solution. Secondly, the search space is very large, to the scale of $n!$. Consequently, these problems are generally termed *NP* or *NP Hard* [24].

The usual approach is to explore the search space in the neighbourhood of good solutions in order to find better solutions. This unfortunately has the effect of converging the population, which then leads to stagnation. The usual term is *local optima convergence*/*stagnation*. Local minima regions acts as attractor basins, where solutions converge. Diversity in the population decreases and possibility of future evolution diminishes.

This research looks at the *diversity* of the population in order to aid the application of metaheuristics. A permutative solution and its representation present some advantages to this effect. The usual measure of a solution is its fitness, in respect to the problem being solved. In a permutative solution, the distinct ordering of values gives the opportunity to have other measures of diversity.

The second application of this research is the development of viable varients of permutative versions of Self Organising Migrating Algorithm (SOMA) [51]. SOMA is a native heuristic, which is based around the concept of cooperating group of solutions in hyperspace. SOMA is loosly based around the concept of "swamp intelligence". SOMA has been effectivelty applied to a number of real-domian problems, however no application for permutative problems have been published. This research strives to be the first appliction of SOMA to permutative problems and bring completeness to the heuristic.

In order to guage the effectiveness of the developed heuritics, a number of different and difficult permutative based combinatorial optimization problems is solved. A total of six unique problem classes are solved, ranging from logistics, manufacturing and scheduling.

The thesis is divided into two parts; theoratical and practical. The therotical part contains chapters on Differential Evolution (Chapter 2), Self Organising Migrating Algorithm (Chapter 3), Permutative Self Organising Migrating Algorithm (Chapter 4) and Chaotic Signature in Population Dynamics (Chapter 5).

The experimental section contains the chapters of the different problem classes. The chapters include those of Permutative Flowshop Scheduling (Chapter 6), Flow Shop Scheduling with Limited Intermediate Storage (Chapter 7), Flow Shop Scheduling with No Wait (Chapter 8), Quadratic Assignment Problem (Chapter 9), Capacitated Vehicle Routing Problem (Chapter 10) and Job Shop Scheduling (Chapter 11). The dissertation is concluded with the chapter on Analysis and Conclusions.

Theoretical Section

Chapter 2

Differential Evolution

Differential evolution (DE) is one of the evolutionary optimization methods proposed by Storn and Price [38]. DE was first introduced to solve the Chebychev polynomial fitting problem by Storn and Price [38].

DE is a population-based and stochastic global optimizer. In general, the DE algorithm starts with establishing the initial population. Each individual has an **m**-dimensional vector with parameter values determined randomly and uniformly between predefined search ranges. In a DE algorithm, candidate solutions are represented by chromosomes based on floating-point numbers. In the mutation process of a DE algorithm, the weighted difference between two randomly selected population members is added to a third member to generate a mutated solution. Then, a crossover operator follows to combine the mutated solution with the target solution so as to generate a trial solution. Thereafter, a selection operator is applied to compare the fitness function value of both competing solutions, namely, target and trial solutions to determine who can survive for the next generation.

In order to describe DE, a schematic is given in Fig 2.1.

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 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

Canonical Differential Evolution Algorithm

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

Figure 2.1: Canonical Differential Evolution Algorithm

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.

Price [38] has 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.

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

Table 2.1: DE Strategies

Strategy	Formulation
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-to-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-to-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:	$u_{i,G+1} = x_{5,G} + F \bullet (x_{r_1,G} - x_{r_2,G} - x_{r_3,G} - x_{r_4,G})$

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 : $u_{j,i,G+1} = v_{j,i,G+1} = x_{j,r_3,G} + F \bullet \left(x_{j,r_1,G} - x_{j,r_2,G}\right)$.

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 $v_{i,G+1}$.

2.0.1 Tuning Parameters

Outlining an absolute value for *CR* is difficult. It is largely problem dependent. However a few guidelines have been laid down [38]. 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 2.2

Table 2.2: 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

2.1 Enhanced Differential Evolution

Enhanced Differential Evolution (EDE) [8, 9], heuristic is an extension of the Discrete Differential Evolution (DDE) variant of DE [10]. 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.

2.1.1 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 2.1.

$$\begin{aligned} P_G &= \{x_{1,G}, x_{2,G}, ..., x_{NP,G}\},\ x_{i,G} = x_{j,i,G} \\ x_{j,i,G=0} &= (\text{int})\left(rand_j\,[0,1] \bullet \left(x_j^{(hi)} + 1 - x_j^{(lo)}\right) + \left(x_j^{(lo)}\right)\right) \\ &\qquad if\ x_{j,i} \notin \{x_{0,i}, x_{1,i}, ..., x_{j-1,i}\} \\ i &= \{1,2,3,...,NP\},\ j = \{1,2,3,..,D\} \end{aligned} \tag{2.1}$$

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.

2.1.2 Forward Transformation

The transformation schema represents the most integral part of the code. Onwubolu [10] developed an effective routine for the conversion.

Let a set of integer numbers be represented as in Equation 2.2:

$$x_i \in x_{i,G} \tag{2.2}$$

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 of 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 [10].

The formulation of the forward transformation is given as:

$$x_i' = -1 + \frac{x_i \bullet f \bullet 5}{10^3 - 1} \tag{2.3}$$

Equation 2.3 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:

$$x_i' = -1 + \frac{x_i \bullet f \bullet 5}{10^3 - 1} = -1 + \frac{x_i \bullet f \bullet 5}{10^3 - 1} \tag{2.4}$$

2.1.3 Backward Transformation

The reverse operation to forward transformation, backward transformation converts the real value back into integer as given in Equation 2.5 assuming x_i to be the real value obtained from Equation 2.4.

$$\mathrm{int}\,[x_i] = \frac{(1 + x_i) \bullet \left(10^3 - 1\right)}{5 \bullet f} = \frac{(1 + x_i) \bullet \left(10^3 - 1\right)}{500} \tag{2.5}$$

The value x_i is rounded to the nearest integer.

2.1.4 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.

$$x_{i,G+1} = \begin{cases} u_{i,G+1} \text{ if } \begin{cases} u_{j,i,G+1} \neq \left\{u_{1,i,G+1}, \ldots, u_{j-1,i,G+1}\right\} \\ x^{(lo)} \leq u_{j,i,G+1} \leq x^{(lo)} \end{cases} \\ x_{i,G} \end{cases} \tag{2.6}$$

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 2.6.

2.1.5 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.

$$u_{j,i,G+1} = \begin{cases} x^{(lo)} & \text{if } u_{j,i,G+1} < x^{(lo)} \\ x^{(hi)} & \text{if } u_{j,i,G+1} > x^{(hi)} \end{cases} \tag{2.7}$$

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 Figure 2.2

Algorithm for Replication Detection

Assume a problem of size n, and a schedule given as $X = \{x_1,..,x_n\}$. Create a *random* solution schedule $\exists! x_i : R(X) := \{x_1,..,x_i..,x_n\}; i \in Z^+$, where each value is unique and between the bounds $x^{(lo)}$ and $x^{(hi)}$.

1. Create a partial empty schedule $P(X) := \{\}$
2. For $k = 1, 2, ..., n$ do the following:
   (a) Check if $x_k \in P(X)$.
   (b) **IF** $x_k \notin P(X)$
   Insert $x_k \rightarrow P(X_k)$
   ELSE
   $P(X_k) = \emptyset$
3. Generate a missing subset $M(X) := R(X) \backslash P(X)$.

Figure 2.2: Pseudocode for replication detection

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.

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 Figure 2.3.

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.

Algorithm for Random Mutation

Assume a problem of size n, and a schedule given as $X = \{x_1, .., x_n\}$. Assume the missing subset $M(X)$ and partial subset $P(X)$ from Figure 2.2.

1. For $k = 1, 2, ..., n$ do the following:

   (a) **IF** $P(X_k) = \emptyset$
   Randomly select a value from the $M(X)$ and insert it in $P(X_k)$ given as $M(X_{Rnd}) \rightarrow P(X_k)$

   (b) Remove the used value from the $M(X)$.

2. Output $P(X)$ as the obtained complete schedule.

Figure 2.3: Pseudocode for random mutation

2.1.6 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.

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. The routine is shown in Figure 2.4.

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 Figure 2.5.

2.1.7 Local Search

There is always a possibility of stagnation in evolutionary algorithms. DE is no exemption to this phenomenon.

Algorithm for Standard Mutation

Assume a schedule given as $X = \{x_1, .., x_n\}$.

1. Obtain two random numbers r_1 and r_2, where $r_1 = rnd\left(x^{(lo)}, x^{(hi)}\right)$ and $r_2 = rnd\left(x^{(lo)}, x^{(hi)}\right)$, the constraint being $r_1 \neq r_2$
   (a) Swap the two indexed values in the solution
      i. $x_{r_1} = x_{r_2}$ and $x_{r_2} = x_{r_1}$.
   (b) Evaluate the new schedule X' for its objective given as $f(X')$.
   (c) **IF** $f(X') < f(X)$
      i. Set the old schedule X to the new improved schedule X' as $X = X'$.
2. Output X as the new schedule.

Figure 2.4: Pseudocode for standard mutation

Algorithm for Insertion

Assume a schedule given as $X = \{x_1, .., x_n\}$.

1. Obtain two random numbers r_1 and r_2, where $r_1 = rnd\left(x^{(lo)}, x^{(hi)}\right)$ and $r_2 = rnd\left(x^{(lo)}, x^{(hi)}\right)$, the constraints being $r_1 \neq r_2$ and $r_1 < r_2$.
   (a) Remove the value indexed by r_1 in the schedule X.
   (b) For k=r_1,.....,$r_2 - 1$, do the following:
      i. $x_k = x_{k+1}$.
   (c) Insert the higher indexed value r_2 by the lower indexed value r_1 as: $X_{r_2} = X_{r_1}$.
2. Output X as the new schedule.

Figure 2.5: Pseudocode for Insertion

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 a hundred generations. This yields to the maximum of twenty stagnations within one run of the heuristic.

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 [33] as is generally known as a 2-opt local search.

Algorithm for Local Search

Assume a schedule given as $X = \{x_1, .., x_n\}$, and two indexes α and β. The size of the schedule is given as n. Set $\alpha = 0$.

1. **While** $\alpha < n$
   (a) Obtain a random number $i = rand[1,n]$ between the bounds and under constraint $i \notin \alpha$.
   (b) Set $\beta = \{i\}$
      i. **While** $\beta < n$
         A. Obtain another random number $j = rand[1,n]$ under constraint $j \notin \beta$.
         B. **IF** $\Delta(x,i,j) < 0; \begin{cases} x_i = x_j \\ x_j = x_i \end{cases}$
         C. $\beta = \beta \cup \{j\}$
      ii. $\alpha = \alpha \cup \{j\}$

Figure 2.6: Pseudocode for 2 Opt Local Search

The basic outline of a local search technique is given in Figure 2.6. 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 (n) 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.

The complete template of Enhanced Differential Evolution is given in Figure 2.7.

Enhansed Differential Evolution Template

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

Figure 2.7: EDE Template

Chapter 3

Self Organising Migrating Alrogithm

The second utilized heuristic is SOMA [51], which is based on the competitive-cooperative behaviour of intelligent creatures solving a common problem.

In SOMA, individual solutions reside in the optimized model′s hyperspace, looking for the best solution. It can be said, that this kind of behaviour of intelligent individuals allows SOMA to realize very successful searches.

Because SOMA uses the philosophy of competition and cooperation, the variants of SOMA are called strategies. They differ in the way as to how the individuals affect all others. The best operating strategy is called ′AllToAll′ and consists of the following steps:

1. *Definition of parameters*. Before execution, the SOMA parameters (PathLength, Step, PRT, Migrations see Table 3.1) are defined.
2. *Creating of population*. The population *SP* is created and subdivided into clusters.
3. *Migration loop*.
   (a) Each individual is evaluated by the cost function
   (b) For each individual the PRT Vector is created.
   (c) All individuals, perform their run towards the randomly selected according to (3.1). Each solution is selected piecewise. The movement consists of jumps determined by the Step parameter until the individual reaches the final position given by the PathLength parameter. For each step, the cost function for the actual position is evaluated and the best value is saved. Then, the individual returns to the position, where it found the best-cost value on its trajectory.

SOMA, like other evolutionary algorithms, is controlled by a number of parameters, which are predefined. They are presented in Table 3.1.

Mutation

Mutation, the random perturbation of individuals, is applied differently in SOMA compared with other evolutionary strategies. SOMA uses a parameter called PRT to achieve

Table 3.1: SOMA parameters

Name	Range	Type
PathLength	(1.1 − 3)	Control
StepSize	(0.11 − PathLength)	Control
PRT	(0 − 1)	Control

perturbation. It is defined in the range [0, 1] and is used to create a perturbation vector (PRT Vector) as shown in Equation 3.1:

$$\begin{array}{l} if \quad rnd_j < PRT \quad \text{then} \quad PRTVector_j = 1 \\ else \quad 0, \quad j = 1,..,n \end{array} \tag{3.1}$$

The novelty of this approach is that in its canonical form, the PRT Vector is created before an individual starts its journey over the search space. The PRT Vector defines the final movement of an active individual in search space.

The randomly generated binary perturbation vector controls the allowed dimensions for an individual. If an element of the perturbation vector is set to zero, then the individual is not allowed to change its position in the corresponding dimension.

Crossover

In standard evolutionary strategies, the crossover operator usually creates new individuals based on information from the previous generation. Geometrically speaking, new positions are selected from an N dimensional hyper-plane. In SOMA, which is based on the simulation of cooperative behaviour of intelligent beings, sequences of new positions in the N-dimensional hyperplane are generated. The movement of an individual is thus given as follows:

$$\vec{r} = \vec{r}_0 + \vec{m}tPRTVector \tag{3.2}$$

where:

- $\vec{r}$: new candidate solution
- $\vec{r}_0$: original individual
- $\vec{m}$: difference between leader and start position of individual
- t : $\in$ [0 , Path length]
- $PRTVector$: control vector for perturbation

It can be observed from Equation 3.2 that the PRT vector causes an individual to move toward the leading individual (the one with the best fitness) in *N-k* dimensional space. If all *N* elements of the PRT vector are set to 1, then the search process is carried out in an *N* dimensional hyperplane (*i.e.* on a *N+1* fitness landscape). If some elements of the PRT vector are set to 0, then the second terms on the right−hand side of Equation 3.2 equals 0. This means those parameters of an individual that are related to 0 in the PRT vector are not changed during the search. The number of frozen parameters, *k,* is simply the number of dimensions that are not taking part in the actual search process. Therefore, the search process takes place in an *N-k* dimensional subspace.

For each individual, once the final placement is obtained, the values are re−converted into integer format. SOMA conversion is different from that used for DE. The values are simply rounded to the nearest integer and repaired using the repairment procedure. This process was developed and selected during experimentation.

Chapter 4

Permutative Self Organising Migrating Algorithm

SOMA has been applied effectively to a number of differential optimization problems. One of the core objectives of this dissertation work was to develop the "permutative" version of SOMA, which can be applied to permutative based combinatorial problems.

As with the problem enountered with the conversion of DE into combinatorial space, effective conversion strategy had to be developed for SOMA.

The following section outlines the three developed strategies; each unique.

4.1 Discrete Set Handling

Discrete Set Handling (DSH) was the first varient developed by Zelinka and Lampinen to solve the *mixed-integer-discrete* problems encountered in mechanical engineering design.

DSH is employed when a set of values containing discrete values, which are "strict sence"; implying its ridigity in the optimization problem. A "discrete set" is created, which is simply an index to the real set.

A solution in the population can be presented as

$$x_{i,G} = \{x_1, x_2, x_3, ..., x_N\} : i \in NP$$

Each variable x in the solution can be represented by an arbitary set containing totally unrelated variables.

The psedocode representation is given in Figure 4.1.

For example assume a set of values which are totally non-related: $SET\,\{1.2, 3, 4.77, 0.11, True, False, Bool\}$. It is simply not possible to optimize such a set of varibles. DSH creates an arbitary index set, where each value is an index to the set: $DHS : \{1, 2, 3, 4, 5, 6, 7\}$. DHS set is then optimized and during fitness evaluation, the index is simply used to link the actual value. An example is given in Figure 4.2.

The DSH simply creates parity with the "base" optimization problem; which in this case is the differential domain.

Algorithm for Discrete Set Handling

Assume a set X of *arbitary* variables of size n. There are NP solutions in the populations, the the maximm number of jumps is given as $Jmp = PathLength / StepSize$

1. For $k = 1, 2, ..., NP$ do the following:
   (a) Create a *random* solution schedule $\exists! x_i : \{N_k\} := \{x_1, .., x_i .., x_n\}; i \in Z^+$
2. For $k = 1, 2, ..., NP$ do the following:
   (a) Take two solutions from the population, N_1 and N_2.
   (b) For $j = 1, 2, ..., Jmp$ do the following:
      i. Create a temporay schedule matrix $\{T_{Jmp}\}$.
      ii. Calculate the new solution $\{T_j\} := N_1 + (N_2 - N_1) \bullet (j \bullet StepSize) \bullet PRTVector$
3. For $j = 1, 2, ..., NP$ do the following:
   (a) For $k = 1, 2, ..., n$ do the following:
      i. Iterchange the values between the two solutions using the values in $\{T_{j,k}\}$ as the *index* to the values in X given as: $\forall k \{T_{j,k}\} :\Leftrightarrow X_{T_{j,k}}$.
   (b) Calculate the *objective function* of each solution: $f(\{T_j\})$.
   (c) If the new solution improves on the old solution N_1, $f(\{T_j\}) < f(N_1)$ it replaces the old solution in the population: $N_1 = \{T_j\}$.

Figure 4.1: Algorithm for Discrete Set Handling

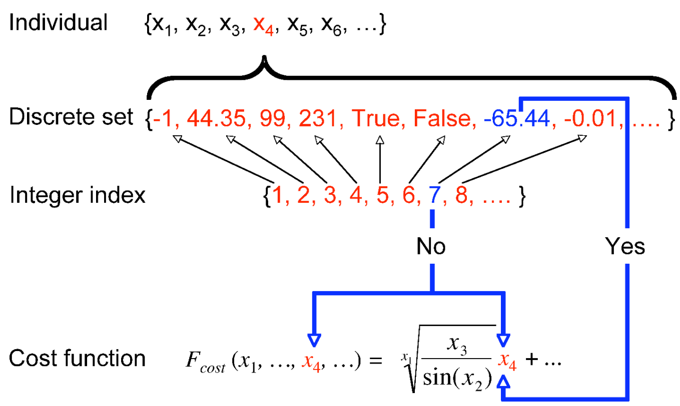

Figure 4.2: Discrete parameter handling

4.2 Permutative Set Handling

DHS is a viable approach when the values of the **initial schedule** are NOT permutative. There is no rule enforcing non-replicaton of the values. Therefore, it becomes possible to have non viable values in the solution. The only approach is to enforce *dual* indexing, first of the schedule and then of the DHS. This is effects duplicates the schedule and adds more *checking* and *correcting* routines.

Permutative Set Handling (PSH) is an approach developed, based on the drawbacks of the DHS. Each solution is created as permutative, similar to as described for EDE, however, no conversion is done to change the variables between the operational domains. Instead, *repairment* is done to each solution. The repairment procedure selected is given as in Figure 4.3.

Algorithm for Random Repair

Assume a problem of size n, and a schedule given as $X = \{x_1,..,x_n\}$. Create a *random* solution schedule $\exists! x_i : R(X) := \{x_1,..,x_i..,x_n\}; i \in Z^+$, where each value is unique and between the bounds.

1. Create a partial empty schedule $P(X) := \{\}$
2. For $k = 1,2,....,n$ do the following:
   (a) Check if $x_k \in P(X)$.
   (b) **IF** $x_k \notin P(X)$
   Insert $x_k \rightarrow P(X_k)$
   ELSE
   $P(X_k) = \emptyset$
3. Generate a missing subset $M(X) := R(X) \backslash P(X)$.
4. For $k = 1,2,....,n$ do the following:
   (a) **IF** $P(X_k) = \emptyset$
   Randomly select a value from the $M(X)$ and insert it in $P(X_k)$ given as $M(X_{Rnd}) \rightarrow P(X_k)$
   (b) Remove the used value from the $M(X)$.
5. Output $P(X)$ as the obtained complete schedule.

Figure 4.3: Algorithm for Random Repair

The outline of PSH is given in below.

---

1. **Initial Phase**
   (a) *Population Generation*: An initial number of permutative trial solutions are generated for the initial population.
2. **SOMA**
   (a) *SOMA Strategy*: The SOMA strategy transforms the parent solution into the child solution using its inbuilt crossover and mutation schemas.

3. **Mutation**

   (a) *Relative Mutation Schema*: Formulates the child solution into a permutative solution of unique values.

4. **Evaluation**

   (a) *Fitness*: Evaluate each solution for its fitness.

5. **Generations**

   (a) *Iteration*: Iterate the solution till a specified generation.

The pseudocode for PSH is given in Figure 4.4

Algorithm for Permutative Set Handling

Assume a schedule of size n. There are NP solutions in the population, and the maximum number of jumps is given as $Jmp = PathLength \big/ StepSize$. The population matrix is given as $\{N_{NP}\}$. The lower bound is given as LB and the upper bound as UB. Create a partial empty schedule $P(X) := \{\}$

1. Create a temporary jump schedule matrix $\left\{T_{Jmp,n}\right\}$.
2. For $k = 1,2,....,NP$ do the following:
   (a) Create a *random* solution schedule $\exists! x_i : \{N_k\} := \{x_1,..,x_i..,x_n\}; i \in Z^+$
3. For $k = 1,2,....,NP$ do the following:
   (a) Take two solutions from the population, one *indexed* and the *best* solution, N_k and N_{best}.
   (b) For $j = 1,2,...,Jmp$ do the following:
      i. Calculate the new solution $\left\{T_j\right\} := N_k + (N_{best} - N_k) \bullet (j \bullet StepSize) \bullet PRTVector$
   (c) For $i = 1,2,....,n$ do the following:
      i. Round each value to the nearest integer $\left\{T_{k,i}\right\} = \left[T_{k,i}\right]$.
      ii. **IF** $\left\{T_{k,i}\right\} < LB$
      Insert $\left\{T_{k,i}\right\} = LB$
      ELSE IF $\left\{T_{k,i}\right\} > UB$
      Insert $\left\{T_{k,i}\right\} = UB$
   (d) For $i = 1,2,....,n$ do the following:
      i. Check if $\left\{T_{k,i}\right\} \in P(X)$.
      ii. **IF** $\left\{T_{k,i}\right\} \notin P(X)$
      Insert $\left\{T_{k,i}\right\} \to P(X_k)$
      ELSE
      $P(X_k) = \emptyset$
   (e) Generate a missing subset $M(X) := \{T_k\} \backslash P(X)$.
   (f) For $i = 1,2,....,n$ do the following:
      i. **IF** $P(X_i) = \emptyset$
      Randomly select a value from the $M(X)$ and insert it in $P(X_i)$ given as $M(X_{Rnd}) \to P(X_i)$
      ii. Remove the used value from the $M(X)$.
   (g) Set $\{T_k\} = P(X)$.
4. Output $\{T\}$ as the obtained complete schedule.

Figure 4.4: Algorithm for Permutative Set Handling

4.3 Static Permutative SOMA

Permutative SOMA is a unique version of SOMA developed on this dissertation work as a complementary approach to solve permutative problems.

Repairement, however effective it may be proven, has a drawback as to that it does not match the *idealogy* of the canonical heuristic. The argument will always be as to how to prove the effectiveness of the underlying heuristic, and the advantage of using repairment strategy.

In EDE, the objective was to have pure conversion between domains. This was feasible due to the vector operations of DE. SOMA, however is a "migrating" algorithm, where the "*space*" between two solutions is mapped in *step-sizes*.

Following this framework, a permutative SOMA; termed *P-SOMA* has been developed for *strict sence* permutative problems.

The first varient is called the ***Static P-SOMA***.

The framework is given below:

---

1. **Initial Phase**
   (a) *Population Generation*: An initial number of permutative trial solutions are generated for the initial population.
   (b) *Fitness Evaluation*: Each soltuion is evaluated fr its fitness.
2. **P-SOMA**
   (a) *Calculate Jump Sequence*: Taking two solutions, the number of possible jumps positions is calculated between each corresponding variable.
   (b) *Generate New Solution*: Using the jump positions; a feasible permutative solution is generated.
   (c) *Recalulate Jump Sequence*: The jump sequence is re calculated taking into consideration the used values.
3. **Selection**
   (a) *New Solution*:The new solutions are evaluated for it fitness and the best new fitness based solution replaces the old solution if it improves upon its fitness.
4. **Generations**
   (a) *Iteration*:Iterate the solution till a specified generation.

The framework is described in detail in the following sub-sections.

4.3.1 Initial Population

The initial population is quite simple to generate. A number of pre-defined variables are required as given in Table 4.1.

The *Population Size* and *Generations* are standard operating parameters of metaheuristics. *Lower bound* refers to the lower limit of the problem being dealt with. The *Upper bound* refers to the upper limit of the solutions.

Table 4.1: Operating variables of P-SOMA

Variable	Syntax	Description
Population Size	NP	The number of solutions
Generations	Gen	Total iteration
Lower bound	LB	Lower limit
Upper Bound	UB	Upper limit
Minimum Jump	MinJ	Minimum number of solutions generated between two solutions
Maximum Jump	MaxJ	Maximum number of solutions generated between two solutions

The *Minimum Jump* and *Maximum Jump* sequences are the equivalent to the *Step-size* in canonical SOMA.

The creation of the initial populaiton is given in Equation 4.1.

$$\text{Initialize}: \begin{cases} \forall i \leq NP \wedge \forall j \leq UB : \begin{cases} x_{i,j,G=0} = LB + rand_j\,[0,1] \bullet (UB - LB) \\ if\; x_{i,j} \notin \{x_{0,i}, x_{1,i}, .., x_{j-1}\} \end{cases} \\ i = \{1,2,...,NP\},\; j = \{1,2,...,UB\},\; Gen = 0,\; rand_j[0,1] \in [0,1] \end{cases} \tag{4.1}$$

4.3.2 P-SOMA

P-SOMA is the routines which calculates the jumps between two solutions in the *k-dimensional* space. In a permutatve setting, a problem is *UB-dimensional*.

Calculating Jump Position

The first part consists of calculating the differences between adjacent solutions as given in Equation 4.2.

$$\begin{aligned} &\text{JumpSeq} = \bigcup_{j=1}^{UB} \left| x_{i,j} - x_{i+n,j} \right|; \\ &i = \{1, .., NP-1\};\; n = \{i+1, ..NP\} \end{aligned} \tag{4.2}$$

JumpPos is a list of values which contain the jump positions between two solutions.

In a static setting, the *MinJumps* is set as a minimum *period* of jump. In this case the *MinJumps* is set by default as 1.

Generating New Solution

The second routine is the selection of the values of the new solution. The idealology of this varient is have as many values as possible within the placement of the two solutions. Starting piecewise from the first solution, the first placed *jump value* is selected for the next solution. The next value is checked for replication and if unique, is selected for the second position as shown in Equation 4.3.

$$
\begin{array}{l}
x_k = \begin{cases} x_{k,j} = \text{ JumpSeq}_{j,l} \\ \text{if JumpSeq}_{j,l} \notin \left\{x_{k,1}, x_{k,2}, .., x_{k,j-1}\right\} \end{cases} \\
j = \{1,2,...,D\}; l = \{MinJumps, MinJumps \bullet 2, .., MinJumps \bullet n\} \\
k = \{1,2.., MaxJumps\}
\end{array}
\tag{4.3}
$$

If a infeasible "JumpSeq" list is encountered, the corresponding value in the new solution is skipped. Once the entire list is filled with the values from the "JumpSeq" list, the remaining values are randomly placed in the solution as given in Equation 4.4.

$$
x_k = \begin{cases} \text{if } x_{k,j} = \emptyset \\ x_{k,j} = Random[LB; UB]; \\ \text{if } Random[LB; UB] \notin, \left\{x_{k,1}, x_{k,2}, .., x_{k,j}\right\} \end{cases}
\tag{4.4}
$$

Re-calculating Jump Position

Once each new solution is created, the corresponding value is removed from the "JumpSeq" list. This way, the corresponding dimension for the particular solution is locked and only through random generation can a dimensional replication be made.

4.3.3 Selection

Each new solution is evalauted for its fitness, and if it improve on the fitness of the *"first"* jump solution, it replaces that particular solution in the population.

4.3.4 Template

The generic template is given in Figure 4.5.

4.3.5 Pseudocode

The pseudocode of the algorithm in given in Figure 4.6.

P-SOMA Template

$$1.\text{Input}: G_{\max}, NP, MinJ \geq 1, MaxJ \geq 1 \text{ and initial bounds}: UB, LB.$$

$$2.\text{Initialize}: \begin{cases} \forall i \leq NP \wedge \forall j \leq UB: \begin{cases} x_{i,j,G=0} = LB + rand_j[0,1] \bullet (UB - LB) \\ if\ x_{i,j} \notin \{x_{1,i}, x_{2,i}, .., x_{j-1}\} \end{cases} \\ i = \{1,2,...,NP\}, j = \{1,2,...,UB\}, G = 0, rand_j[0,1] \in [0,1] \end{cases}$$

$$\begin{cases} 3.\text{While } G < G_{\max} \\ \forall i \leq NP \begin{cases} 4.\quad \text{Calculate Jump Sequence}: \\ \qquad 4.1 \text{ JumpSeq}_k = \bigcup_{j=1}^{UB} \left| x_{i,j} - x_{i+n,j} \right|; i = \{1,2,...,NP\}, j = \{1,2,...,UB\} \\ \forall k \leq MaxJ \begin{cases} 4.2: x_k = \begin{cases} x_{k,j} = \text{JumpSeq}_{j,l} \\ \text{if JumpSeq}_{j,l} \notin \{x_{k,1}, x_{k,2}, .., x_{k,j-1}\} \\ \quad l = \{MinJumps, MinJumps \bullet 2, .., MinJumps \bullet n\} \end{cases} \\ 4.3: x_k = \begin{cases} \text{if } x_{k,j} = \emptyset \\ x_{k,j} = Random[LB; UB]; \\ \text{if } Random[LB; UB] \notin, \{x_{k,1}, x_{k,2}, .., x_{k,j}\} \end{cases} \end{cases} \\ 5.\text{Select} \\ x_{i,G+1} = \begin{cases} x_k \text{ if } f(x_k) \leq f(x_{i,G}) \\ x_{i,G} \text{ otherwise} \end{cases} \end{cases} \\ G = G + 1 \end{cases}$$

Figure 4.5: P-SOMA Template

4.3.6 Worked Example

The ideal explanation of P-SOMA is through the use of a worked example. Consider two random permutative solutions of size 10 which can be represented as in Table 4.2:

Table 4.2: Example of Initial Population

Solutions	Representation
x_1	$\{1,2,8,6,7,4,10,9,5,3\}$
x_2	$\{6,7,3,4,2,1,5,8,9,10\}$

Using Equation 4.2, the jump sequence can be calculated as given in Table 4.3:

Table 4.3: Example of Jump sequence calculation

Solutions	Representation
x_1	$\{1,2,8,6,7,4,10,9,5,3\}$
x_2	$\{6,7,3,4,2,1,5,8,9,10\}$
Jump Sequence	$\{5,5,5,2,5,3,5,1,4,7\}$

Using these values, the *MinJ* can be seen as the lowest value and *MaxJ* as the maximum value. From these values MinJ = 1 and MaxJ = 5. MaxJ is chosen as 5 and not 7, since the frequency of 5 is higher than that of 7.

Using MinJ and MaxJ, the JumpSeq's are generated in Table 4.4.

Table 4.4: Example of Jump sequence generation

x_1	x_2	JumpSeq
1	6	$\{2,3,4,5\}$
2	7	$\{2,3,4,5,6,7\}$
8	3	$\{7,6,5,4\}$
6	4	$\{5\}$
7	2	$\{6,5,3\}$
4	1	$\{3,2\}$
10	5	$\{9,8,7,6\}$
9	8	$\{\ \}$
5	9	$\{6,7,8\}$
3	10	$\{4,5,6,7,8,9\}$

Now, using the selection of closest feasible value, a new solution can be selected as shown in Table 4.5.

The new solution can be represented as in Table 4.6.

The missing values are randomly placed in the solution. From the solution, the number of *fixed* dimension is 8 and 2 dimensions are outside of the two solution settings. These two values are the *overshoot*, which in the canonical SOMA is described as the *PathLength*.

Table 4.5: Example of new solution generation

x_1	x_2	JumpSeq
1	6	$\{\mathbf{2}, 3, 4, 5\}$
2	7	$\{2, \mathbf{3}, 4, 5, 6, 7\}$
8	3	$\{\mathbf{7}, 6, 5, 4\}$
6	4	$\{\mathbf{5}\}$
7	2	$\{\mathbf{6}, 5, 3\}$
4	1	$\{3, 2\}$
10	5	$\{\mathbf{9}, 8, 7, 6\}$
9	8	$\{\ \}$
5	9	$\{6, 7, \mathbf{8}\}$
3	10	$\{\mathbf{4}, 5, 6, 7, 8, 9\}$

Table 4.6: Example of new solution

Solution	$\{\ 2, 3, 7, 5, 6, , 9, , 8, 4\ \}$
Missing Values	$\{1, 10\ \}$
New Solution	$\{\ 2, 3, 7, 5, 6, 10, 9, 1, 8, 4\ \}$

Figure 4.7 shows the two solutions plotted in two dimension, and the feasible jump space between them. Figure 4.8 shows the new solution plotted between the two solutions. As described only two dimensions of the new solutions are outside of the fesible jump space.

Once the solution is plotted, the “JumpSeq” is re-calculated. The values already used are removed from the “JumpSeq” and the second solution is calculated. The re-calculation is given in Table 4.7.

Table 4.7: Example of Jump sequence re-calculation

x_1	x_2	JumpSeq
1	6	$\{3, 4, 5\}$
2	7	$\{2, 4, 5, 6, 7\}$
8	3	$\{6, 5, 4\}$
6	4	$\{\ \}$
7	2	$\{5, 3\}$
4	1	$\{3, 2\}$
10	5	$\{8, 7, 6\}$
9	8	$\{\ \}$
5	9	$\{6, 7\}$
3	10	$\{5, 6, 7, 8, 9\}$

The new selection is now done as in Table 4.8:

The second new solution can be represented as in Table 4.9:

In the second solution, only 7 dimensions are locked, and 3 are open. As the

Table 4.8: Example of new solution selection

x_1	x_2	JumpSeq
1	6	$\{\mathbf{3}, 4, 5\}$
2	7	$\{\mathbf{2}, 4, 5, 6, 7\}$
8	3	$\{\mathbf{6}, 5, 4\}$
6	4	$\{\ \}$
7	2	$\{\mathbf{5}, 3\}$
4	1	$\{3, 2\}$
10	5	$\{\mathbf{8}, 7, 6\}$
9	8	$\{\ \}$
5	9	$\{6, \mathbf{7}\}$
3	10	$\{5, 6, 7, 8, \mathbf{9}\}$

Table 4.9: Example of new solution representation

Solution	$\{ 3, 2, 6, , 5, , 8, , 7, 9 \}$
Missing Values	$\{1, 4, 10 \}$
New Solution	$\{ 3, 2, 6, 4, 5, 1, 8, 10, 7, 9 \}$

solutions are generated, the number of locked dimension reduces and the number of open dimensions increases.

In P-SOMA, only the specified "MaxJ" number of solutions are generated for any two solutions. This gurantees a possible maximum number of **UB** jumps for any two solutions.

4.4 Dynamic Permutative SOMA

Dynamic P-SOMA is a second approach of SOMA. The main difference in Dynamaic P-SOMA is that the *MinJ* and *MaxJ* are self adapting.

Whereas, in the static approach, the MinJ and MaxJ were dependent on the actual ordering of the solutions, in the dynamic approach, they are dependent of the problem size being solved. MinJ is adapted as the jump iteration of at least a fifth of the problem space; hense a fifth of the possible jump space is mapped. MaxJ is usually set to lower than half of the problem size. This is done in order to have more manageable evolution rate and secondly, to induce more randomness into the heuristic as shown in Table 4.10.

Table 4.10: Dynamic P-SOMA parameters

Parameters	Static P-SOMA	Dynamic P-SOMA
MinJ	Minimum difference	$< 1/5$
MaxJ	Maximum difference	$< 50\%$

For larger sized problems, it is more prudent to have an even mapping of the solutions space. The pseudocode for Dynamic P-SOMA is given in Figure 4.9.

Algorithm for Static P-SOMA

Assume a problem of size n, and two solutions $X_1 = \{x_{1,1},..,x_{1,n}\}$ and $X_2 = \{x_{2,1},..,x_{2,n}\}$ in k dimensional space. Create a *random* solution schedule $\exists! x_i : R(X) := \{x_1,..,x_i..,x_n\}; i \in Z^+$,

1. Create a empty schedule for the *jump sequence* $JS := \{\}$.
2. For $k = 1,2,....,n$ do the following:
   (a) Calculate the difference between the adjacent values of X_1 and X_2 given as $JS_k = \left|X_{1,k} - X_{2,k}\right|$.
3. Calculate the *Minimum Jumps* **(MinJ)** and *Maximum Jumps* **(MaxJ)** between the two solutions as $MinJ = \min[JS]$ and $MaxJ = \max[JS]$.
4. Create a *Jump Matrix*, which contains all jump solutions as $\{T_{MaxJ,n}\}$ and a *Jump Sequence Matrix* $\{P_{n,MaxJ}\}$ which contains the *partial* jumps between the two solutions.
5. For $k = 1,2,....,n$ do the following:
   (a) For $j = 1,2,....,JS_k$ do the following:
      i. Generate a list of values between the adjacent values of $X_{1,k}$ and $X_{2,k}$ given as:
      IF $X_{1,k} < X_{2,k}$
      Insert $P_{k,j} = \min\{X_{1,k}, X_{2,k}\} + j$
      ELSE IF $X_{1,k} > X_{2,k}$
      Insert $P_{k,j} = \max\{X_{1,k}, X_{2,k}\} - j$
      ELSE IF $X_{1,k} = X_{2,k}$
      $P_{k,j} = \emptyset$
6. For $k = 1,2,....,MaxJ$ do the following:
   (a) Create a *schedule* for each jump sequence starting from the first feasible value in the partial schedule $\{P_{n,MaxJ}\}$
   (b) For $i = 1,2,....,k$ do the following:
      i. For $j = 1,2,....,P_k$ do the following:
         A. **IF** $P_{i,j} \notin T_k$
         Insert $\{T_{k,i}\} := P_{i,j}$
         ELSE
         $\{T_{k,i}\} = \emptyset$
7. For $k = 1,2,....,MaxJ$ do the following:
   (a) Generate a missing subset $M(X) := R(X) \setminus \{T_k\}$ for each schedule.
   (b) For $i = 1,2,....,k$ do the following:
   (c) **IF** $\{T_{k,i}\} = \emptyset$
   Randomly select a value from the $M(X)$ and insert it in $P(X_k)$ given as $M(X_{Rnd}) \rightarrow \{T_{k,i}\}$
   (d) Remove the used value from the $M(X)$.
8. Output $\{T\}$ as the obtained complete schedule.

Figure 4.6: Algorithm for Static P-SOMA

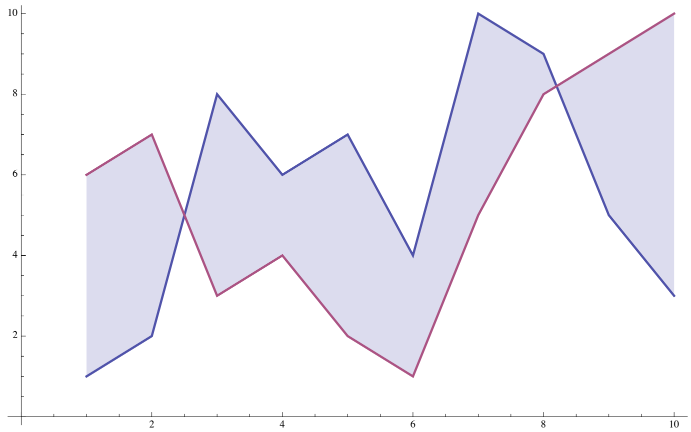

Figure 4.7: Jump space between the two solutions

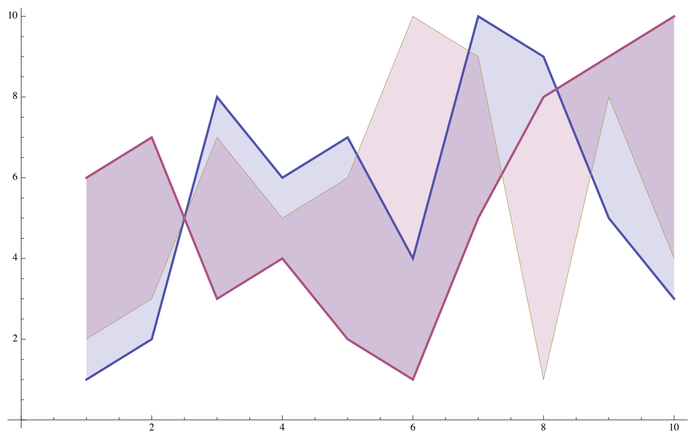

Figure 4.8: New solution in the jump space

Algorithm for Dynamic P-SOMA

Assume a problem of size n, and two solutions $X_1 = \{x_{1,1},..,x_{1,n}\}$ and $X_2 = \{x_{2,1},..,x_{2,n}\}$ in k dimensional space. Create a *random* solution schedule $\exists! x_i : R(X) := \{x_1,..,x_i..,x_n\}; i \in Z^+$.

1. Set the *Minimum Jumps* (**MinJ**) and *Maximum Jumps* (**MaxJ**) between the two solutions as $MinJ = n/\alpha; \alpha \leq 0.2$ and $MaxJ = n\big/\beta; \beta \leq 0.5$.

2. Create a empty schedule for the *jump sequence* $JS := \{\}$.

3. For $k = 1,2,....,n$ do the following:

   (a) Calculate the difference between the adjacent values of X_1 and X_2 given as $JS_k = \left|X_{1,k} - X_{2,k}\right|$.

4. Create a *Jump Matrix*, which contains all jump solutions as $\{T_{MaxJ,n}\}$ and a *Jump Sequence Matrix* $\{P_{n,MaxJ}\}$ which contains the *partial* jumps between the two solutions.

5. For $k = 1,2,....,n$ do the following:

   (a) For $j = 1,2,....,JS_k$ do the following:

      i. Generate a list of values between the adjacent values of $X_{1,k}$ and $X_{2,k}$ given as:
      IF $X_{1,k} < X_{2,k}$
         Insert $P_{k,j} = \min\left\{X_{1,k}, X_{2,k}\right\} + (MinJ \bullet j)$
      ELSE IF $X_{1,k} > X_{2,k}$
         Insert $P_{k,j} = \max\left\{X_{1,k}, X_{2,k}\right\} - (MinJ \bullet j)$
      ELSE IF $X_{1,k} = X_{2,k}$
         $P_{k,j} = \emptyset$

6. For $k = 1,2,....,MaxJ$ do the following:

   (a) Create a *schedule* for each jump sequence starting from the first feasible value in the partial schedule $\{P_{n,MaxJ}\}$

   (b) For $i = 1,2,....,k$ do the following:

      i. For $j = 1,2,....,P_k$ do the following:

         A. **IF** $P_{i,j} \notin T_k$
               Insert $\left\{T_{k,i}\right\} := P_{i,j}$
            ELSE
               $\left\{T_{k,i}\right\} = \emptyset$

7. For $k = 1,2,....,MaxJ$ do the following:

   (a) Generate a missing subset $M(X) := R(X) \setminus \{T_k\}$ for each schedule.

   (b) For $i = 1,2,....,k$ do the following:

   (c) **IF** $\left\{T_{k,i}\right\} = \emptyset$
      Randomly select a value from the $M(X)$ and insert it in $P(X_k)$ given as $M(X_{Rnd}) \rightarrow \left\{T_{k,i}\right\}$

   (d) Remove the used value from the $M(X)$.

8. Output $\{T\}$ as the obtained complete schedule.

Figure 4.9: Algorithm for Dynamic P-SOMA

Chapter 5

Chaotic Signature in Population Dynamics

Population and its application to chaotic systems is well documented. Populations viewed as dynamical systems was first discussed by [30]. Subsequent work by [20], further chronicled the work of viewing populations as number systems. The logisitc map, the simpliest chaotic system is also used for the modeling of population dynamics [30]. Another system is the Voltterra-Lotka equations of biological models.

Chaos in optimization has been largely explored through Neural Networks [27]. The core approach has been to avoid regions of “local optima” or “stagnation” in order to find better solutions. The basic concept has been that chaotic dynamics have been able to search for solutions along the formation of a strange attractor which has fractual structures. These structures are then used to search for solutions in state space along such fratural attractors who’s Legesgue measure is zero.

Nozawa [32] modified the Hopfield-Neural network by the Eulers method to create an equivalent to the chaotic neural network of [2]. A 10 city problem is solved with better results than stochastic models.

Yamada and Aihara [50] solved the Traveling Salesman Problem with chaotic neural networks by computing the largest Lyaponov exponent. They showed that the solving abilities are very high when the largest Lypanov exponent is near zero, which implies that “an edge of chaos ” could have high performance to solve combinatorial problems.

Maintenance scheduling problems were solved by a chaotic simulated annealing approach by [6]. It was also proven of the existance of chaotic dynamics in solving combinatorial problems using chaotic neural networks.

A further exploration of chaos in optimization was done by [28], who proposed a new network model of chaotic potts spin. Using this method the constraint term is always satisfied and feasible solutions are always obtained.

This research takes a similar approach to the ones described, as the main aspect is the avoidance of “local optima” regions in the search space. However, we look upon the population as the driving system behind the convergence of the population.

The usual approach is to visiualize the population as a fitness landscape, where solutions transverge towards global optimal solutions. This approach takes a differnet view of the population. A population is looked upon as an information base, a “genetic code” base where each solution occupies a distinct place in the information space. An example is given in Figure 5.1.

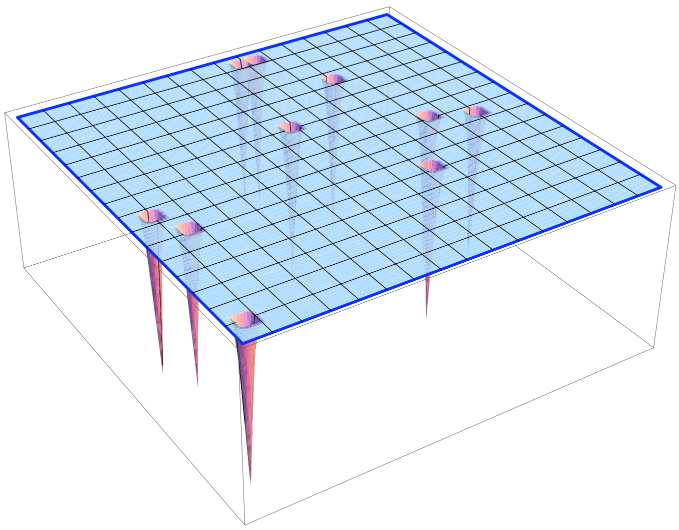

Figure 5.1: Population representation

During successive generations, solutions are mated together, and an exchange of information takes place. Based on selection criteria of different algorithms, new and better performing solutions are accepted in the population after each generation.

However, during evolution, solutions tend to converge towards each other. What this in effect does, is reduce the amount of information available in the information plane of the collective information gene pool. Even if the solution converges towards the global minima, the information left in the population is usually marginal. This is what is termed as "local optima stagnation".

The main input in this research is the creation of a dynamic population which is kept on the threshold of information viability and which can be used by any number of metaheuristics as a population paradigm.

5.1 Population Dynamics

Each solution in a population contains certain information, its own "genetic code" which is used for replication. A way to visualise it is to see a solution as occupying a certain point in the information space as given in Figure 5.2.

The *basin* or trough that the solution occupies is dependent on the number of solutions which occupy the same basin. The basin boundaries are not exactly linear, but rather a contour. This presents the possibility/probability for entry and escape from this specific point as given in Figure 5.3.

As population evolves, the information is shared within the evolving solutions. Within a number of generations, a number of solutions can occupy the same information space. The size of the "basin" increasing and its attraction energy also increases. As more and more solutions are replicated, the number of "evolutionary channels" which exists between the solutions decreases. This gives rise to stagnation of the population, where no new solutions with new/better information is produced.

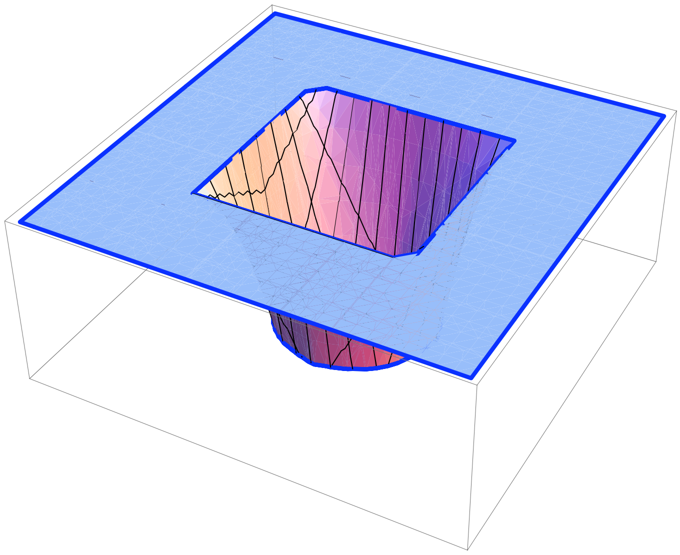

Figure 5.2: Solution in information space

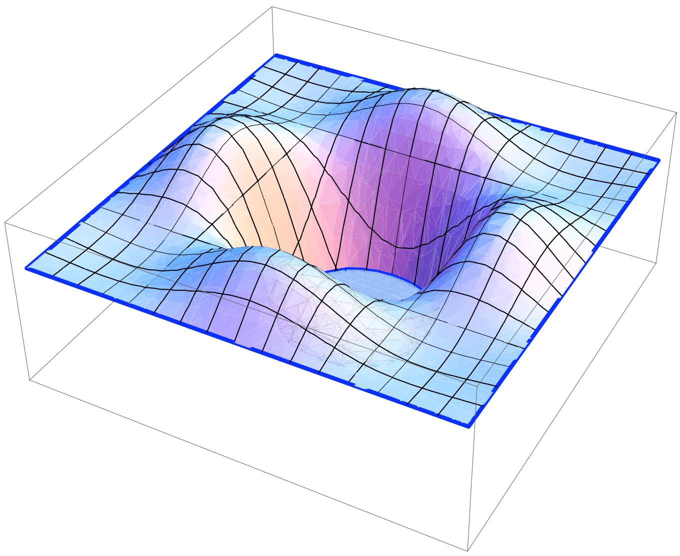

Figure 5.3: Solution boundary in information space

5.1.1 Initial population

The main reason for random population is to provide an initial loose mapping of the solution space. For permutative problems, where solution ordering is stringent, it is often the case that adjacent values are required. A typical approach of using local search heuristics to search in the neighbourhood of the solutions usually yields closely aligned solutions.

The initial population P, for this heuristic is partially stochastic and partly deterministic. The population is divided into two sub-populations, SPs, one randomly generated

(SP_{rand}) and the other structurally generated (SP_{struct}).

The formulation for SP_{rand} is fairly simple. A random permutative string is generated for each solution till a specified number given as P_{size}.

The structured population SP_{struct} is somewhat more complex. It is made of two parts. In the first part, an initial solution is generated with ascending values given as $x_{ascending} = \{1,2,3,..,n\}$, where n is the size of the problem. In order to obtain a structured solution, the first solution is segmented and recombined in different orders to produce different combinations. The first segmentation occurs at $^n/_2$, and the two halfś are swapped to produce the second solution. The second fragmentation occurs by the factor 3; $^n/_3$ Three regions of solutions now exist. The number of possible recombination′s that can exist is 3! = 6. At this point there are nine solutions in the SP_{struct}. The general representation is given as:

$$k \geq 1 + 2! + 3! + ... + z! \tag{5.1}$$

where z is the total number of permutations possible and k is $^{P_{size}}/_2$.

The psedocode of r the populaiton generation is given in Figure 5.4.

Algorithm for Clustered Population Generation

Assume a population given as P which is divided equally into two sub-populations; one random SP_r and one strustured SP_s. The schedule size is n and population size is NP. The maximum catanation of the schedule is given as c and the permutation rate is given as $p_r = c!$.
Generate random population.

1. For $i = 1,2,....,NP/2$ do the following:
   (a) Create a *random* solution schedule $\exists! x_i : SP_r := \{x_1,..,x_i..,x_n\}; i \in Z^+$
2. Create structured population.
   (a) Calculate the trucation point and number as $t_p = \lfloor ^n/_c \rfloor$.
   (b) Generate two schedules, one *forward biased* $X_f = \{1,2,...,n\}$ and the other *reverse biased* $X_r = \{n,n-1,...,1\}$.
   (c) Generate permutation list for forward bias given as:
   $\{X_f\} = \{\{1,..,x_{t_p}\},\{x_{t_p}+1,.....,2 \bullet x_{t_p}\},....,\{c \bullet x_{t_p},...,n\}\}$ and reverse bias as $\{X_r\} = \{\{n,..,c \bullet x_{t_p}\},\{2 \bullet x_{t_p},.....,x_{t_p}+1\},....,\{x_{t_p},...,1\}\}$.
   (d) $i = 1,2,....,p_r$ do the following:
      i. Generate a permutative list based on the truncation points in the solution.
3. Output $P = SP_r \cup SP_s$ as the final population.

Figure 5.4: Algorithm for Clustered Population Generation

5.1.2 Solution Dynamics

A solution represented as $x = \{x_1, x_2, ..., x_n\}$, where n is the number of variables, within a population has a number of attributes. Usually the most visible is its fitness value, by which it is measured within the population. This approach is not so viable in order to measure the diversity of the solution in the population. In retrospect, a single solution is assigned a number of attributes for measure, as given in Table 5.1.

Table 5.1: Solution Parameters

Parameter	Description	Activity
Deviation	Measure of the deviation of the solution	Control
Spread	Alignment of the solution	Control
Life	Number of generation cycles	Selection
Offspring	Number of successful offspring$'$s produced	Selection

The most important attribute is the *deviation* (the difference between successive values in a solution). Since we are using only permutative solutions, deviation or *ordering* of the solution is important. This is due to the fact that each value in the solution is unique. Each value in the solution has a unique footprint in the search space. The formulation for deviation is given as:

$$\delta = \left(\frac{\sum\limits_{i=1}^{n-1} |x_i - x_{i+1}|}{n} \right) x_i \in \{x_1, x_2, ..., x_n\} \tag{5.2}$$

Spread of a solution gives the alignment of the solution. Each permutative solution has a specific ordering, whether it is *forward* aligned or *reverse* aligned. Whereas deviation measures the distance between adjacent solutions, spread is the measure of the hierarchy of subsequent solutions given as:

$$\partial = \begin{cases} +1 & if\,(x_{i+1} - x_i) \geq 1 \\ -1 & if\,(x_{i+1} - x_i) \leq 1 \end{cases} \quad i \in \{1, 2, ..., n\} \tag{5.3}$$

The generalisation of *spread* is given in Table 5.2.

Table 5.2: Spread generalization

Spread	Generalization
> 0	Forward spread
0	Even spread
< 0	Reverse spread

Life is the number of generations the solution has survived in the population and *Offspring* is the number of viable solutions that have been created from that particular solution. These two variables are used for evaluating the competitiveness of different solutions.

The pseudocode is given in Figure 5.5

Algorithm for Solution Dynamics

Assume a problem of size n, and a schedule given as $X = \{x_1, .., x_n\}$. There are NP schedules in the population. Initialize $X_{sprd} = 0$.

1. For $i = 1, 2, ..., NP$ do the following:

   (a) Calculate **deviation**: $X_{i,dev} = \sum_{1}^{n-1} \frac{|x_i - x_{i+1}|}{n}$

   (b) Calculate **spread**: $X_{i,sprd} = X_{sprd} + 1 \Leftrightarrow \sum_{1}^{n-1} (x_i - x_{i+1}) > 1$ and $X_{i,sprd} = X_{sprd} - 1 \Leftrightarrow \sum_{1}^{n-1} (x_i - x_{i+1}) < 1$

Figure 5.5: Algorithm for Solution Dynamics

5.1.3 Chaotic Features

Within the population, certain solutions are seen to exhibit attracting features. These points are usually local optima regions, which draw the solutions together. The approach utilized is to subdivide the population in clusters, each cluster a distinct distance from another.

Figure 5.6 shows a “deviation” space with three clusters. Each cluster contains “n” solutions. At any one time “n” clusters will be in the population, and these clusters share information to create new solutions.

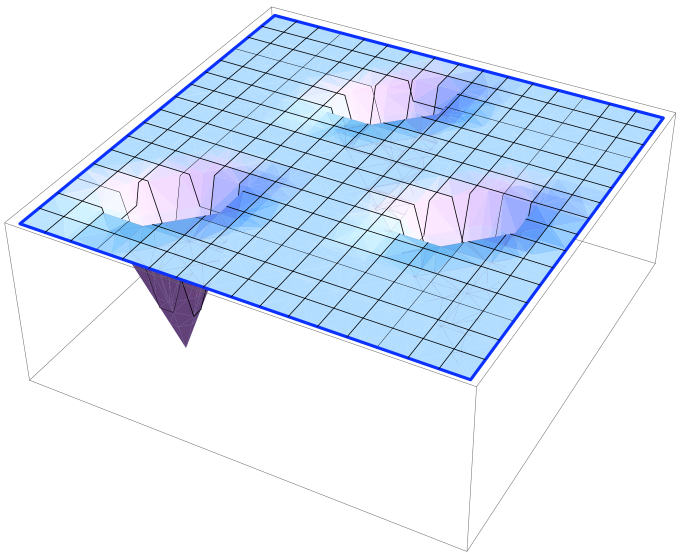

Figure 5.6: Clusters in deviation space

Two controlling parameters are now defined which control the clusters.

Chaos Attractor C_A: The distance that each segment of solution has to differ from each other. The C_A is given in (5.4).

$$C_A \in [0.1, 1+) \tag{5.4}$$

Within the population indexed by the *deviation*, solutions with similar deviation are clustered together, and each cluster is separated by at least a single C_A as seen in (5.5).

$$\begin{aligned}&(\delta_1, \delta_2, ..., \delta_i) \overset{C_A}{\leftrightarrow} (\delta_{i+1}, \delta_{i+2}, ..., \delta_{2i}) \overset{C_A}{\leftrightarrow} \\ &... \overset{C_A}{\leftrightarrow} (\delta_{3i+1}, \delta_{3i2}, ..., \delta_{4i})\end{aligned} \tag{5.5}$$

The second controlling factor is the ***Chaos Edge*** C_E . Whereas C_A is the mapping of individual solutions, C_E is the measure of the entire population. Figure 5.7 shows the deviation space with the boundary outline. The entire "active" solution space is within the region of the outer contours. This is the "chaotic edge" of the current information space.

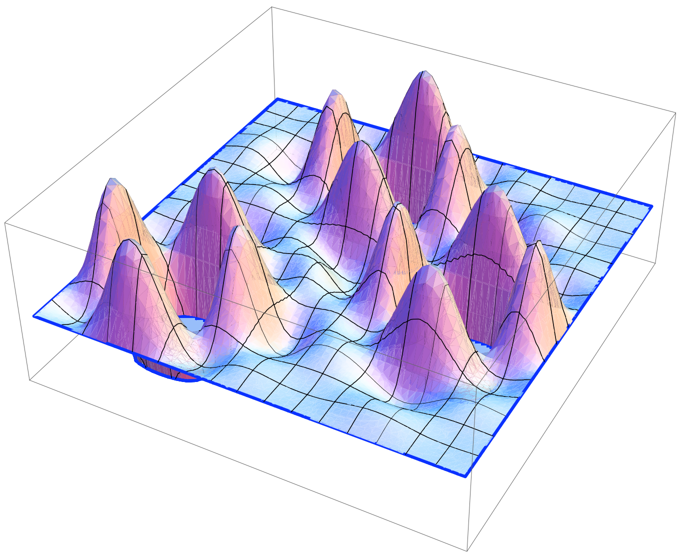

Figure 5.7: Boundary of the clusters

C_E is the measure of the deviation of the fitness of the population and is used to prevent the population from stagnating to any fitness minima. The algorithm is given in FIgure 5.8

5.1.4 Selection and Deletion

Selection of the next generation is based on a tier-based system. If the new solution improves on the global minima, it is then accepted in the solution. Otherwise, competing clusters jokey for the new solution. Initially the solution is mapped for its deviation. This deviation is then mapped to the corresponding cluster.

Within the cluster, the placement of the solution is evaluated. If the new solution corresponds to an existing solution, or reduces the threshold C_A value of the cluster, then it is discarded.

Algorithm for Chaotic Features Calculation

Assume a problem of size n, and a schedule given as $X = \{x_1, .., x_n\}$. There are NP schedules in the population $\{P\}$ and each schedule has a deviation and fitness given by X_{devi} and X_{sprd}. The cluster distance is given by C_A. Initilaize four clusters $\{C_1\}$, $\{C_2\}$, $\{C_3\}$ and $\{C_4\}$.

1. For $i = 1, 2, ..., NP$ do the following:
   (a) Sort the $\{P\}$ in asending order of X_{devi}.
   (b) Divide the population into the four clusters based on X_{devi}.
   (c) For $j = 1, .., 4$ do the following:
      i. Calculate the difference between boundary solutions of each cluster $\{C\}$. $C_{A,j} = X_{\max[X_{devi}],C_j} - X_{\min[X_{devi}],C_j}$
      ii. **IF** $C_{A,j} < C_A$
         A. Dynamic clustering of the boundary solutions of each cluster.
2. Output $\{P_C\}$ as the clustered population.

Figure 5.8: Algorithm for Chaotic Features Calculation

The solution is accepted if it improves on the C_A value of the cluster (hence improving diversity) and also to some extent keeps the balance of the C_E . If the cluster has less than average solutions, then the new solution is admitted.

Table 5.3 gives the selection criteria.

Table 5.3: Selection criteria

Variables	Criteria
Fitness	Improves clusters best solution
C_A	Increases the value of C_A
C_E	Problem dependent

Once the solution is added to the cluster, another solution can be discarded. This solution is usually elected from the middle placed solutions in the cluster, whose fitness is not in the top 5% of the population. If no such solutions exist, then the average rated solution is removed. Solution with high *Life* and low *Offspring* are discarded, since they are considered dormant within the cluster.

Table 5.4 gives the deletion criteria.

5.1.5 Dynamic Clustering

The selection and crossover criteria have now been outlined. After each generation / migration, the clusters are reconfigured. Since, in all heuristics, there is a tendency to converge, it is imperative to keep the solutions unique.

Algorithm for Selection

Assume a problem of size n, and a new schedule given as $X_{new} = \{x_1, .., x_n\}$. There are NP schedules in the population $\{P\}$ and each schedule has a deviation and fitness given by X_{devi} and X_{sprd}. The cluster distance is given by C_A.

1. Calculate the deviation and spread of the solution X_{new} as $X_{new,devi}$ and $X_{new,devi}$.
2. Find the associated cluster $P_{C,X}$ of the new solution X_{new} based on $X_{new,devi}$: $X_{new,devi} \in C$.
3. Calculate the fitness of the new solution $f(X_{new})$.
4. **IF** $X_{new} \rightarrow \{P_{C,X}\} \| X_{new,devi} \cup \{P_{C,X}\} > C_{A,X}$
   (a) Insert the new solution in the associated cluster $X_{new} \rightarrow \{P_{C,X}\}$.
   (b) Update the life X_{life} and offspring $X_{ofsprng}$ value of the parent solution.
   (c) Calculate the $C_{E,X}$ of the new cluster.

Figure 5.9: Algorithm for Selection

Table 5.4: Deletion criteria

Variables	Criteria
Life	High
Offspring	Low
C_A	Decreases

The procedure is to calculate the deviation of the new solutions. Since a mesh of solutions may exist, it is feasible to reconfigure certain boundary solutions. Figure 5.11 can be a representation of a sub-population (SP).

A mutation routine is used to reconfigure the solution. By altering certain positions within the solution it is possible to realign the deviation and spread of the solution. Boundary values within the solutions (usually represented by the upper and lower bound of the solution) are swapped. Another approach is to have two random positions generated and the values in these positions swapped. An illustration is given to describe this process in Table 5.5, Figure 5.12 and Figure 5.13.

Table 5.5: Swap of boundary values

Solution	Deviation	Spread
10 9 6 5 2 **1** 8 7 4 3	2.1	-7
1 9 6 5 2 **10** 8 7 4 3	3.0	-5

Once the boundary values are re-aligned, the second migration/generation loop oc-

Algorithm for Deletion

Assume a problem of size n, and a new schedule given as $X_{new} = \{x_1, .., x_n\}$. There are NP schedules in the population $\{P\}$ and each schedule has a deviation and fitness given by X_{devi} and X_{sprd} and life and offspring given as X_{life} and $X_{ofsprng}$. The cluster distance is given by C_A and the Edge is given as C_E. The active cluster is given as $P_{C,A}$.

1. Randomly select a boundary solution as in the active cluster X_A. If the solution has poor offspring and long life in comparison to the avegare values of the cluster, it is deleted from the population.
2. **IF** $X_{A,ofsprng} < avg\left[P_{C,ofsprng}\right] \| X_{A,life} > avg\left[P_{C,life}\right]$
   (a) Delete X_A.

   If the selected solution increases the C_A value between the clusters, it is selected for deletion.
3. **ELSE IF** $(X_A \not\subset \{P_{C,X}\}) > C_A$
   (a) Delete X_A.
4. Calculate the $C_{E,X}$ of the new cluster.

Figure 5.10: Algorithm for Deletion

curs. The pseudocode is given in Figure 5.14.

5.2 Metaheuristics

The clustered population is designed to be used by any metaheuristic. This is the advantage of this approach, since it is not tied down to a specific method. This section discusses three different heuristics of Genetic Algorithm (GA), Differential Evolution (DE) from Section 2 and Self-Organising Migrating Algorithm (SOMA) from Section 4. Each of these heuristics has been applied to a number of permutative opimization problems.

In each of the heuristics used, the canonical population was removed and replaced with the clustered population and its integrated features.

5.2.1 Genetic Algorithms

Genetic Algorithm (GA) is an adaptive heuristic search algorithm premised on the evolutionary ideas of natural selection and genetics. GA is designed to simulate processes in natural system necessary for evolution, specifically those that follow the principles first laid down by Charles Darwin of survival of the fittest. As such, they represent an intelligent exploitation of a random search within a defined search space to solve a problem [25].

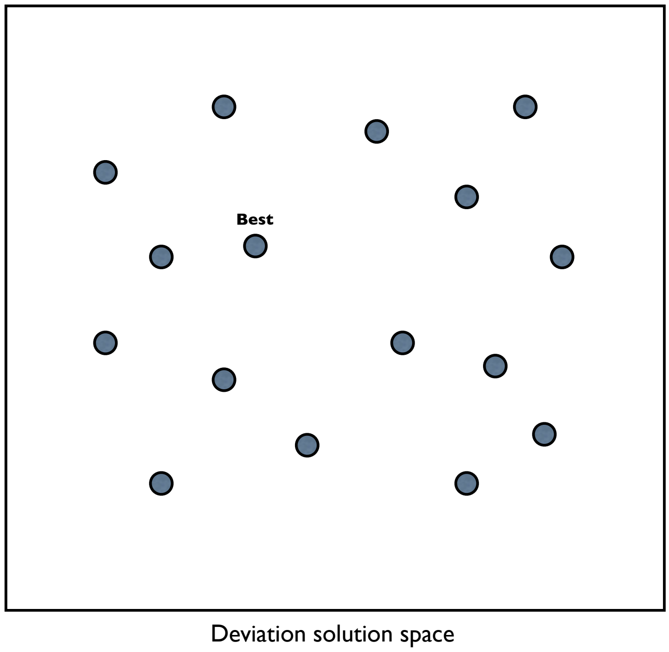

Figure 5.11: Solution space after migration

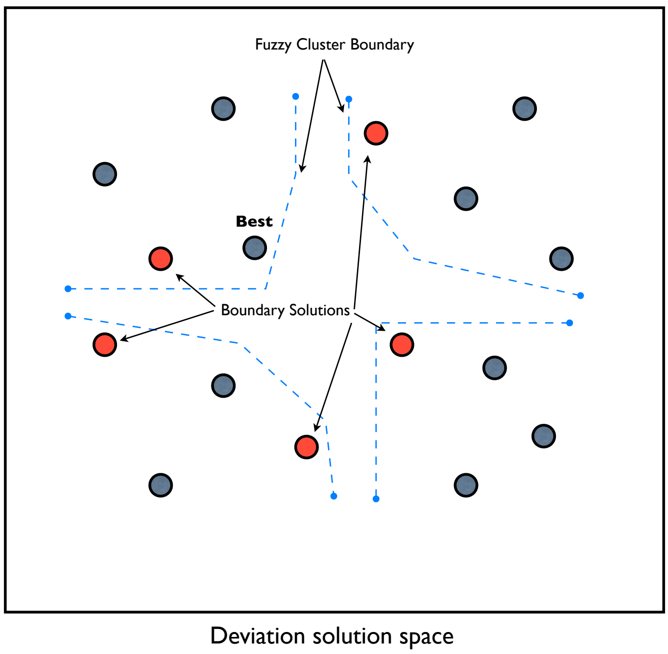

Figure 5.12: Fuzzy clustering and boundary solution isolation

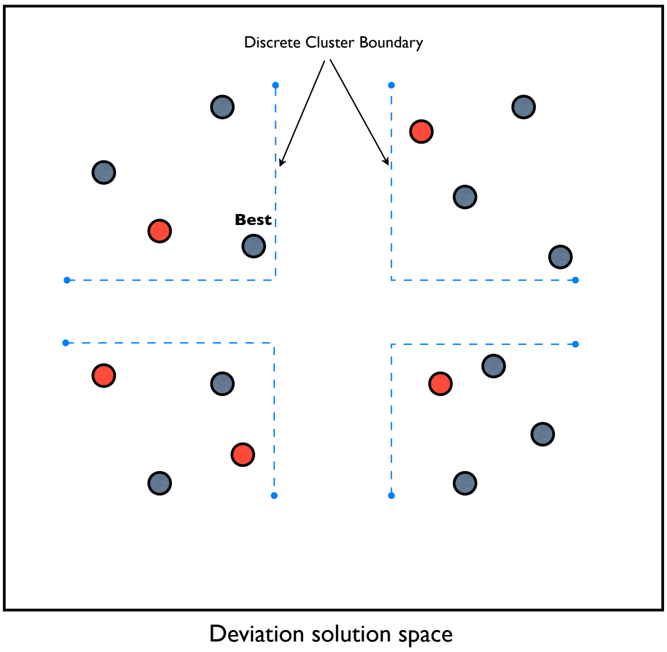

Figure 5.13: Realigned solutions into discrete clusters

Algorithm for Dynamic Clustering

Assume four clusters C_1 - C_4, each with seperation distance $C_{A,i}$, where i refers to the corresponding cluster. Each schedule has n variables.

1. Isolate each schedule in a cluster which has a seperation value less than that of C_A: $X_{devi} < C_{A,X}$.
2. **DO**
   (a) Randomly select two unique random indicies on the schedule $Rnd\,[r_1, r_2] \in n$.
   (b) Using these indicies exchange the values in the solution: $x_{r_1} \Leftrightarrow x_{r_2}$.
   (c) Calculate new deviation of the solution $X_{new,devi}$.
   (d) **IF** $X_{devi} > C_{A,X}$
      i. Accept new schedule in the solution $X_{new} \rightarrow \{P_{C,X}\}$
3. **WHILE** new schedule NOT accepted in cluster

Figure 5.14: Algorithm for Dynamic Clustering

A number of variants of GA exist. For this research, a two-point crossover approach was used as the crossover methodology for the propagation of the population.

A two-point crossover approach is simple to execute. Two solutions from *different clusters* are randomly selected. These solutions are checked to ensure that their *spread* is not equal. This is done to map more diversified solutions. Two crossover positions are randomly selected in the solutions given as $\{CP_1, CP_2\} = Random[n]$, and the two solutions are mated with a possibility of six unique offspring′s being created. An illustration of the selection and crossover is given in Figure 5.15.

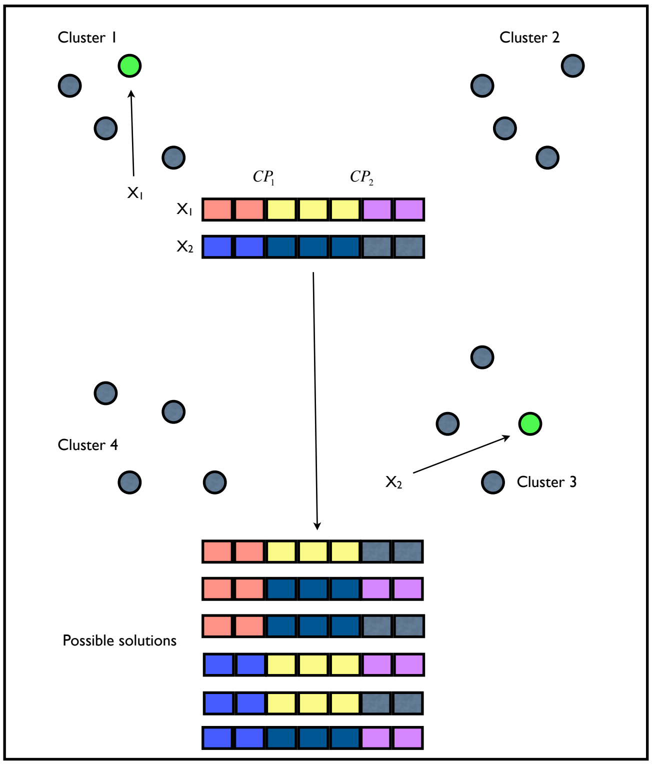

Figure 5.15: GA representation

An example of this process can be shown by having the two values of crossover given as $CP_1 = 2$ and $CP_2 = 4$. The two solutions selected for crossover can be represented as $x_1 = \left\{2,5,\underset{2}{|}4,3,\underset{4}{|}1,6\right\}$ and $x_2 = \left\{3,4,\underset{2}{|}1,2,\underset{4}{|}6,5\right\}$. Three regions exist within each solution. By swapping alternate regions, a total number of possible solutions is now given as in Table 5.6.

With this crossover process, infeasible solutions are usually created. An effective repairment routine is described in the following section that was used to repair the solutions.

Once all the solutions are repaired, their fitness is evaluated and the solution with the best fitness is selected for possible adaptation into the population.

Table 5.6: Possible solutions from crossover

Permutation	Solution
$\{1,1,2\}$	$\{2,5,4,3,6,5\}$
$\{1,2,1\}$	$\{2,5,1,2,1,6\}$
$\{1,2,2\}$	$\{2,5,1,2,6,5\}$
$\{2,1,1\}$	$\{3,4,4,3,1,6\}$
$\{2,1,2\}$	$\{3,4,4,3,6,5\}$
$\{2,2,1\}$	$\{3,4,1,2,1,6\}$

Repairment

The repairment process is given in a number of routines. The first routine is to check the entire solution for repeated values. These repeated values and their positions are isolated in a replicated array $x_{repl} = \left\{x_j, x_{j+n}, .., x\right\}$. The second routine is to find which values are missing from the solutions given as $x_{mis} = \{1, .., n\} \cap \{x_1, x_2, .., x_n\}$.

Since, the replicated array contains a number of sequences of replicated solutions, randomly one solution in each sequence is labelled as feasible and repatriated back into the main solution. This leaves the replicated array containing only infeasible values.

Randomly each value is selected from the missing array and inserted in the position of a replicated value in the replicated array $x_{mis} \overset{random}{\rightarrow} x_{repl}$.

Finally, the replicated array is reinserted in the solution array with all values now feasible $x_{repl} \rightarrow x$.

An illustrative example is given in Table 5.7.

Table 5.7: Illustrative example of repairment.

Routine	**Rand**	x	x_{repl}	x_{mis}
Replicated values		$\{1,3,4,3,4,$ $10,6,7,1,1\}$	$(1,1,1^*)$ $(4^*,4)$	
Missing value				$\{2,8,9\}$
Feasible solution	$\{3,1\}$	$\{*,3,4,3,*,$ $10,6,7,*,1\}$	$(1,1,1^*)$ $(4^*,4)$	
Repair solution	$\{2,3,1\}$ $\{3,1,2\}$		$\{\underset{3}{1},\underset{1}{1},\underset{2}{4}\}$	$\{\underset{2}{2},\underset{3}{8},\underset{1}{9}\}$
Final solution		$\{8,3,4,5,9,$ $10,6,7,2,1\}$		

5.2.2 Differential Evolution Algorithm

Differential Evolution (DE) [38], is the second heuristic selected to be used in conjunction with the clustered population. DE uses a vector perbutation methodology for crossover.

There are ten working strategies for DE, but the one selected for implementation is the DE/rand/2/bin represented as in Equation 5.6.

$$U_{i,G+1} = x_{best,G} + F \cdot (x_{j,r_1,G} - x_{j,r_2,G} - x_{j,r_3,G} - x_{j,r_4,G}) \tag{5.6}$$

This strategy was selected since it maps to the four unique clusters in the *SP*. The best solution is selected from the entire *SP* based on fitness value. Then, each random solution is selected from each distinct cluster. Again the selected values are checked for opposing *spread*. If the spread is identical, then a second round of selection occurs. A schematic is given in Figure 5.16.

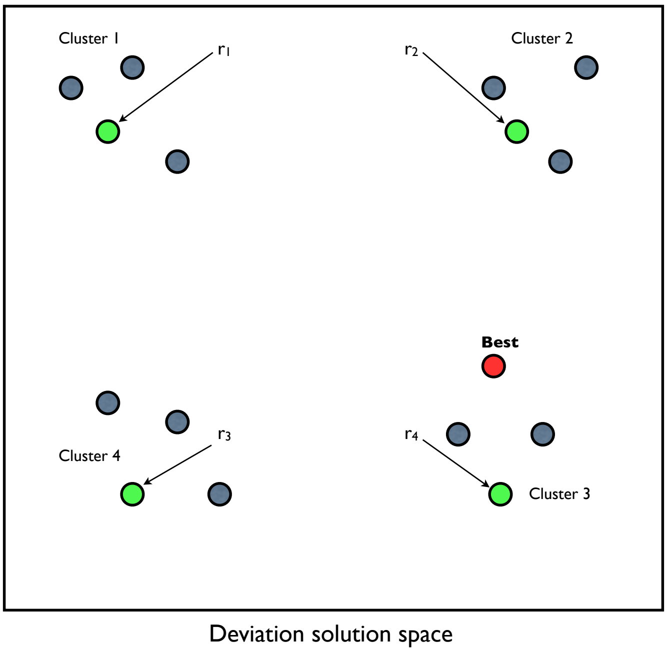

Figure 5.16: DE selection

The selection of the cluster is random, so r_1 can be selected from any cluster with no preference. These values are subtracted given as $x_{j,r_1,G} - x_{j,r_2,G} - x_{j,r_3,G} - x_{j,r_4,G}$. The resulting value is multiplied by the scaling factor F and added to the best solution as given in Fig. 5.17.

The resulting value is only accepted in the new solution if a generated random number is below the given threshold provided by the controlling parameter of *CR*. This procedure provides added stochasticity to the heuristic.

5.2.3 Self Organising Migrating Algorithm

The third utilized heuristic is SOMA [51], which is based on the competitive-cooperative behaviour of intelligent creatures solving a common problem.

In SOMA, individual solutions reside in the optimized model's hyperspace, looking for the best solution.

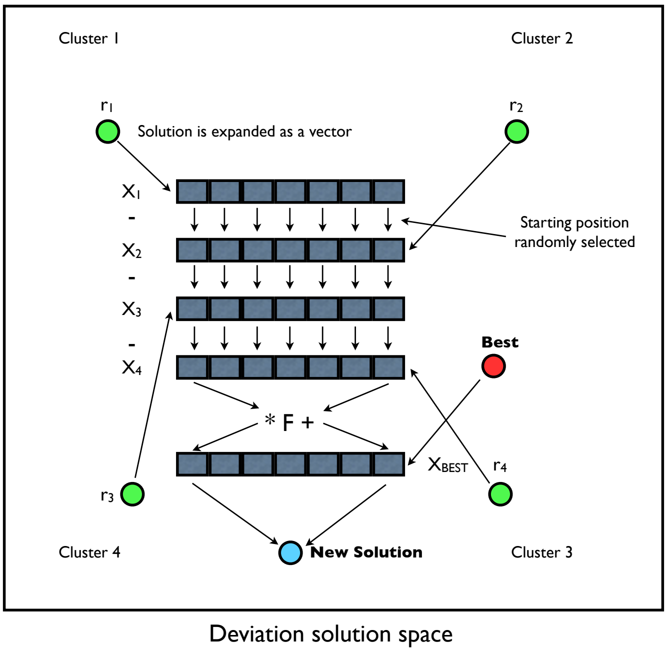

Figure 5.17: DE crossover

Three version of SOMA ave been used; SOMA with PSH, Static P-SOMA and Dynamic P-SOMA.

The schematic of SOMA with clustered population is given in Figure 5.18.

SOMA, like other evolutionary algorithms, is controlled by a number of parameters, which are predefined. They are presented in Table 5.8.

Table 5.8: SOMA parameters for PSH

Name	Range	Type
PathLength	3	Control
StepSize	0.21	Control
PRT	$(0-1)$	Control

For each individual, once the final placement is obtained, the values are re−converted into integer format. SOMA conversion is different from that used for DE. The values are simply rounded to the nearest integer and repaired using the repairment procedure. This process was developed and selected during experimentation.

5.3 General Template

Collating all the piecewise explanation, a general generic template is now described. The conceptual framework of this approach has been published in [14].

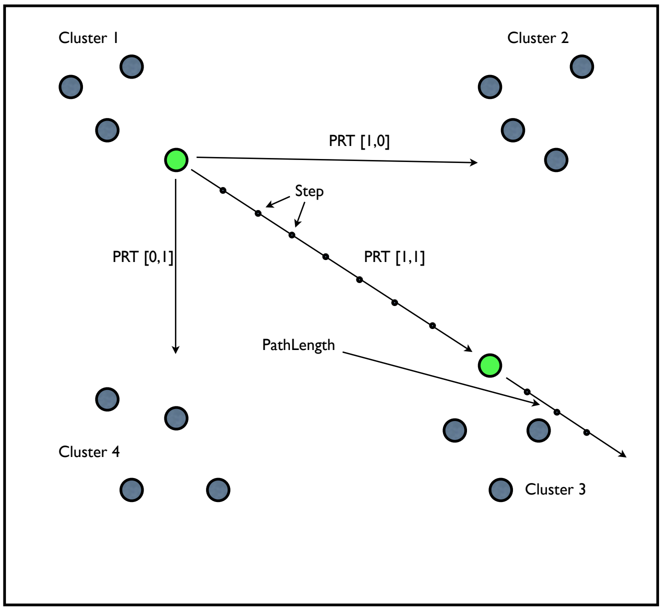

Figure 5.18: SOMA migration utilizing clustered population

1. *Initialize*: Assign the problem size n, population size P_{size}, sub population sizes SP_{struct} SP_{rand}, and the control parameters of C_A and C_E .

2. *Generate:* Randomly create SP_{rand}, half the size of P_{size}, and then structurally create SP_{struct}. These two form the basis of the population.

3. *Calculate:* Calculate the *deviation* and *spread* of each solution in the population. Taking the *deviation* values, configure the population into four clusters. The minimal separation value between the clusters is assigned as C_A. Taking the entire *SP*, the standard deviation of the *fitness* is computed. This is labelled as the C_E.

4. *Generation/Migration*

   (a) Taking each *SP* in turn, the selected heuristic of GA, DE or SOMA is applied to the population.

   (b) The new solution is calculated for its *deviation* and *spread*.

   (c) Using the selection criteria, the solution is placed within the cluster corresponding to its deviation. If replicated solutions exist, then it is discarded. Selection is based on *fitness* and the move of the C_A and C_E.

5. *Re-calculation:* The SP is re-calculated for its cluster boundaries.

6. *Dynamic clustering:* If the value of C_A has deceased, then the boundary solutions are reconfigured. The C_E value is calculated for the new population.

The generic template is given in Figure 5.19

General Template

1. Input: $n, P_{size}, SP_{struct}, SP_{rand}, C_A \in (0.1, 1+), C_E, Gen$

$$2.\ \text{Initialize: } SP_{rand} = \begin{cases} \forall i \le {P_{size}}/{2} \wedge \forall j \le n : x_{i,j,G=0} = rand_j[0,1] \bullet \left(x_j^{(hi)} - x_j^{(lo)}\right) \\ i = \left\{1,2,..,{P_{size}}/{2}\right\}, j = \{1,2,...,n\}, G = 0, rand_j[0,1] \in [0,1] \end{cases}$$

$$SP_{struct} = \begin{cases} k = {P_{size}}/{2} ; z \ni \max\left(\sum_{i=1}^{z} (1 + 2! + ... + z!)\right) \le k; x_{ascend} = \left\{x^{(lo)}, .., x^{(hi)}\right\} \\ x_{descend} = \left\{x^{(hi)}, .., x^{(lo)}\right\} \\ \forall i \le z, \left(x_{ascend}, x_{descend}\right) \subseteq i \xrightarrow{permutate} append\left(SP_{struct}\right) \\ \end{cases}$$

$$3.\ \text{Calculate} \begin{cases} \text{Deviation } \delta = \left(\dfrac{\sum_{j=1}^{n-1} \left| x_j - x_{j+1} \right|}{n} \right) : x_j \in \left\{x_1, x_2, .., x_n\right\} \\ \\ \text{Spread } \partial = \begin{cases} +1 & if \left(x_{j-1} - x_j\right) \ge 1 \\ -1 & if \left(x_{j-1} - x_j\right) \ge 1 \end{cases} \\ C_A = \left(\delta_1, \delta_2, .., \delta_{k/5}\right) \overset{C_A}{\leftrightarrow} \left(\delta_{(k/5)+1}, \delta_{(k/5)+2}, .., \delta_{2(k/5)}\right) \overset{C_A}{\leftrightarrow} ... \overset{C_A}{\leftrightarrow} \left(\delta_{4(k/5)+1}, \delta_{4(k/5)+2}, .., \delta_k\right) \\ C_E = std\left(f\left(x_i\right)\right) : x_i \in \left\{x_1, x_2, ..., x_{P_{size}}\right\} \end{cases}$$

$$\begin{cases} 4.\ \text{While } G < G_{\max} \text{ for each } SP \\ \forall i \le P_{size} \begin{cases} \\ 5.\ \text{Mutate and recombine:} \\ \qquad \vec{u}_{i,G+1} \overset{DE}{\leftarrow} \left\{x_1, x_2, ..., x_i\right\} \\ \qquad 5.1\ \text{ Calculate } \delta \text{ and } \partial \text{ of } \vec{u}_{i,G+1} \\ 6.\ \text{Select} \\ \vec{x}_{i,G+1} = \begin{cases} \vec{u}_{i,G+1} & \text{if } f\left(\vec{u}_{i,G+1}\right) \le f\left(\vec{x}_{best}\right) \\ \vec{u}_{i,G+1} & \text{if } > C_A \\ \vec{x}_{i,G} & \text{otherwise} \end{cases} \\ 7.\ \text{Calculate } C_A, C_E \\ 8.\ \text{Dynamic clustering} \end{cases} \\ G = G + 1 \end{cases}$$

Figure 5.19: General Template

Experimental Section

Chapter 6

Permutative Flow Shop Scheduling

In many manufacturing and assembly facilities, a number of operations have to be done on every job. Often these operations have to be done on all the jobs in the same order implying the jobs have to follow the same route. The machines are assumed to be set up in series and the environment is referred to as a *flow shop* [36].

Flow Shop *Fm*: There are m machines in series. Each job has be pocessed in each one of the m machines. All the jobs have to follow the same route (i.e., they have to processed on Machine 1, and then on Machine 2, etc). After completing on one machine, a job joins the queue at the next machine. Usually all jobs are assumed to operate under the *First In First Out (FIFO)* discipline - that is a job caanot "pass" another while waiting in a queue. Under this effect the envirnment is refereed to as a *permutative* flow shop. the general syntex of this problem as described in the triplet format $\alpha|\beta|\gamma$, is given as

$$Fm\,|Perm\,|C_{\max}$$

The first field denotes the problem being solved, the second field the type of problem (in this case permutative) and the last field denotes the objective being under investigation, which is the makespan (total time taken to complete the job).

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

$$\Im = \sum_{j=1}^{n} C_{m,j} \tag{6.1}$$

s.t.

$$C_{i,j} = \max\left(C_{i-1,j}, C_{i,j-1}\right) + P_{i,j} \tag{6.2}$$

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 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 6.2 applies to these two problem descriptions.

The value of the makespan under a given permutation schedule can also be computed by determining the *critical path* in a directed graph corresponding to the 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 [36]. A schmetic is given in Fig 6.1.

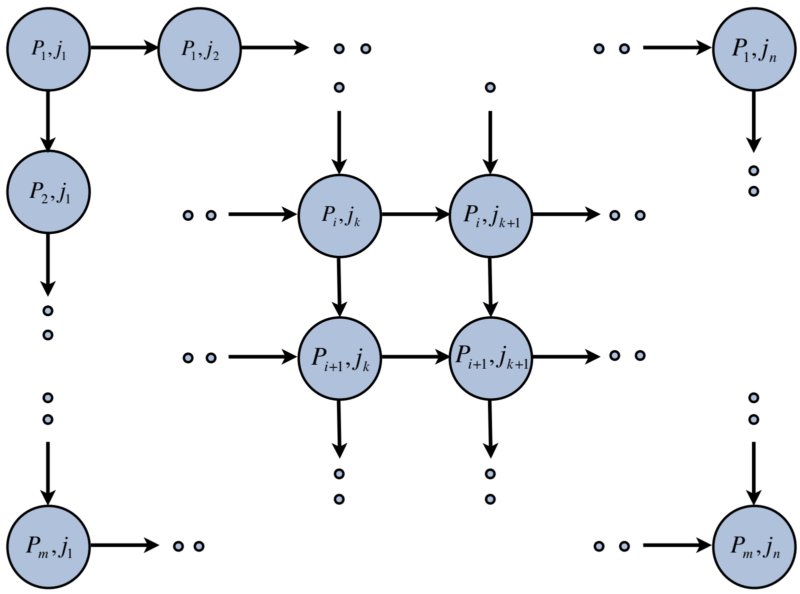

Figure 6.1: Directed graph representation for $Fm|Perm|C_{\max}$

6.1 Experimentation

Two separate phases of experimentation was conducted to show the benefits of clustering of the population. The first set was the application of canonical forms of the heuristics to the problem of flow shop scheduling, in order to set a benchmark from which any improvement can be measured. To this effect, the control parameters and all other operational parameters were kept stagnant.

The control parameters of the population are given in Table 6.1.

P_{size} is generally dependent on the scale of the problem being solved. However the benefits of using a large population is not evident, especially when clustering. Through experimentation, the optimal population cluster was from 200 to 400 solutions. Larger population led to complication in clustering and proved ineffective in improving the heuristic.

Table 6.1: Population operating parameters

Parameter	Value
P_{size}	200 - 400
Generations	> 250/SP
Clusters	4
C_A	> 0.1

Another important fact was that the optimal number of cluster was found to be 4 for best performance of the heuristic.

The control parameters of SOMA and DE are presented in Table 6.2 and Table 6.3.

Table 6.2: SOMA operating parameters

Parameter	Range
PathLength	3
StepSize	0.23
PRT	(0 - 1)

Table 6.3: DE operating parameters

Parameter	Value
F	0.3
CR	0.1

All parameters in Table 6.2 and Table 6.3 were obtained through extensive experimentation.

The experimentation was conducted on a parallel 16 Apple X-Serve cluster at the Tomas Bata University in Zlin, Czech Republic. All codes were written in Mathematica 7 platform. All the mentioned data sets were obtained from [3].

6.1.1 Car, Rec, Hel Benchmark problem sets

The first sets of Flowshop scheduling benchmark problems are Car [5], Rec [41] and Hel [23] benchmark sets. A total of 31 instances exist, each of varying size and difficulty [37].

Table 6.4 gives the results obtained by the heuristics of GA, DE and SOMA. The first phase of comparison is done with the canonical and clustered counterpart of these heuristics in order to show the benefits of using clustering.

The results are presented as percentage increase over the reported optimal value. The results are presented in two formats. The first is the heuristic applied in its canonical form, or without clustering. The second part is the results presented with the clustered population. These heuristics are marked with the subscript *Clus*.

Comparing each heuristic with and without clustering, it is evident that a clustered population improves the heuristic. For GA, the improvement is dramatic. Since only a

Table 6.4: Comparison of canonical and clustered heuristics in Car/Rec/Hel problem

Name	n x m	Cost	GA	GA_{Clus}	DE	DE_{Clus}	SOMA	$SOMA_{Clus}$
Car1	11x5	7038	0	0	0	0	0	0
Car2	13x4	7166	0	0	0	0	0	0
Car3	12x5	7312	0	0	0	0	0	0
Car4	14x4	8003	0	0	0	0	0	0
Car5	10x6	7720	0	0	0	0	0	0
Car6	8x9	8505	0	0	0	0	0	0
Car7	7x7	6590	0	0	0	0	0	0
Car8	8x8	8366	0	0	0	0	0	0
Rec01	20x5	1247	1.04	0	0	0	0	0
Rec03	20x5	1109	1.76	0	0	0	0	0
Rec05	20x5	1242	1.43	0	0	0	0.002	0
Rec07	20x10	1566	1.22	0	0.98	0	0.01	0
Rec09	20x10	1537	1.45	0	0.32	0	0	0
Rec11	20x10	1431	1.32	0	0.54	0	0	0
Rec13	20x15	1930	0.96	0.34	0.45	0.31	0	0
Rec15	20x15	1950	0.87	0.5	0.32	0.28	0.01	0
Rec17	20x15	1902	1.67	0.31	0.29	0.28	0.02	0
Rec19	30x10	2093	1.09	0.41	0.42	0.338	0.02	0
Rec21	30x10	2017	1.68	0.37	0.39	0.38	0.02	0
Rec23	30x10	2011	2.45	0.32	0.21	0.21	0.03	0
Rec25	30x15	2513	2.11	0.43	0.32	0.29	0.03	0
Rec27	30x15	2373	1.2	0.63	0.42	0.27	0.01	0
Rec29	30x15	2287	1.32	0.73	0.61	0.34	0	0
Rec31	50x10	3045	1.91	0.52	0.7	0.32	0.04	0
Rec33	50x10	3114	2.34	0.43	0.84	0.28	0	0
Rec35	50x10	3277	0.43	0.42	0.91	0.27	0	0
Rec37	75x20	4951	3.42	0.9	1.32	0.33	0.09	0.02
Rec39	75x20	5087	2.45	0.89	1.56	0.29	0.06	0.02
Rec41	75x20	4960	3.21	0.92	1.98	0.28	0.09	0.01
Hel01	100x10	513	3.7	0.97	2.1	0.53	0.02	0.01
Hel02	20x10	135	1.21	0	1.97	0	0	0

two point crossover approach was used, the results obtained with GA were not promising, especially for larger problems. However, the solutions improved with clustering, on average all the solutions exhibited optimal values of less than 1% over the optimal. A possible advantage of clustering is that mutation was included in GA through clustering.

The results of DE were obtained from [11]. In 11 instances, the optimal value was obtained, and on average the percentage increase was below 1%. Using clustering, DE_{Clus} markedly improves all the soltuions. This is clearly seen in the large problems sizes of 50 jobs and more. The improvement is clearly in excess of 1.5%.

The final heuristic, SOMA, is the best performing heuristic in these problem instances. The results of SOMA [13] are very close to the optimal, usually in the range of only 0.05 above the optimal. $SOMA_{Clus}$ further improved these results with only four instances failing to find the reported optimal, and all of them at most only 0.02% above the optimal.

The second phase of comparison is done with other published heuristics on the same problem instances. Comparison of the clustered heuristics is done with the Improved Genetic Algorithm (IGA) and Multiagent Evolutionary Algorithm (MAEA) [26] and the Hybrid Genetic Algorithm (H-GA) and Othogonal Genetic Algorithm (OGA) of [48]. The results are given in Table 6.5.

Table 6.5: Comparison of clustered heuristics with other published heuristics

Name	n x m	Cost	H-GA	OGA	IGA	MAEA	GA_{Clus}	DE_{Clus}	$SOMA_{Clus}$
Car1	11x5	7038	0	0	0	0	0	0	0
Car2	13x4	7166	0	0	0	0	0	0	0
Car3	12x5	7312	0	0	0	0	0	0	0
Car4	14x4	8003	0	0	0	0	0	0	0
Car5	10x6	7720	0	0	0	0	0	0	0
Car6	8x9	8505	0	0	0	0	0	0	0
Car7	7x7	6590	0	0	0	0	0	0	0
Car8	8x8	8366	0	0	0	0	0	0	0
Rec01	20x5	1247	0	0.04	0	0	0	0	0
Rec03	20x5	1109	0	0	0	0	0	0	0
Rec05	20x5	1242	0.08	0.21	0	0	0	0	0
Rec07	20x10	1566	0	0.79	0	0	0	0	0
Rec09	20x10	1537	0	0.35	0	0	0	0	0
Rec11	20x10	1431	0	0.91	0	0	0	0	0
Rec13	20x15	1930	0.52	1.08	0.62	0	0.34	0.31	0
Rec15	20x15	1950	0.92	1.23	0.46	0	0.5	0.28	0
Rec17	20x15	1902	1.26	2.08	1.73	0	0.31	0.28	0
Rec19	30x10	2093	0.38	1.76	1.09	0.28	0.41	0.338	0
Rec21	30x10	2017	0.89	1.64	1.44	0.44	0.37	0.38	0
Rec23	30x10	2011	0.45	1.9	0.45	0.44	0.32	0.21	0
Rec25	30x15	2513	1.03	2.67	1.63	0.43	0.43	0.29	0
Rec27	30x15	2373	1.18	2.09	0.8	0.56	0.63	0.27	0
Rec29	30x15	2287	1.05	3.28	1.53	0.78	0.73	0.34	0
Rec31	50x10	3045	0.56	1.49	0.49	0.1	0.52	0.32	0
Rec33	50x10	3114	0	1.87	0.13	0	0.43	0.28	0
Rec35	50x10	3277	0	0	0	0	0.42	0.27	0
Rec37	75x20	4951	2.54	3.41	2.26	2.72	0.9	0.33	**0.02**
Rec39	75x20	5087	1.79	2.28	1.14	1.61	0.89	0.29	**0.02**
Rec41	75x20	4960	2.82	3.43	3.27	2.7	0.92	0.28	**0.01**
Hel01	100x10	513	-	-	-	0.38	0.97	0.53	**0.01**
Hel02	20x10	135	-	-	-	0	0	0	0

In general comparison with published results, the clustered approaches of $SOMA_{Clus}$ and DE_{Clus} are the top two performing heuristics. MAEA approach is the best comparative heuristic, however $SOMA_{Clus}$ is easily the better performing heuristic for large problems. In comparison of MAEA with DE_{Clus}, even though MAEA obtains more optimal solutions, DE_{Clus}, performs more consistently in large problems.

6.1.2 Taillard Benchmark problem sets

The second set of benchmark problems is referenced from [44]. These sets of problems have been extensively evaluated [42]. This benchmark set contains 120 particularly hard instances each of 10 different sizes, selected from a large number of randomly generated problems.

As in the previous case, the first comparison is done with canonical and clustered approaches of GA, DE and SOMA

Table 6.6: Comparison of canonical and clustered heuristics

	GA		GA_{Clust}		*DE*		DE_{Clus}		*SOMA*		$SOMA_{Clus}$	
Problem	Δavg	Δstd	Δavg	Δstd	Δavg	Δstd	Δavg	Δstd	Δavg	Δstd	Δavg	Δstd
20x5	2.12	1.23	2	1.34	0.98	0.66	0.55	0.71	0.42	0.48	0.39	0.6
20x10	3.22	0.76	2.9	0.87	1.81	0.77	1.32	0.98	1.29	0.45	1.28	0.55
20x20	3.42	0.98	1.9	0.76	1.75	0.57	0.98	1.32	1.09	0.34	0.96	0.65
50x5	1.76	0.76	0.56	0.88	0.4	0.36	0.33	0.76	0.41	0.34	0.32	0.29
50x10	4.32	1.53	2.54	1.23	3.18	0.94	3.13	0.77	4.8	1	3.8	0.97
50x20	4.53	1.22	4.22	0.93	4.05	0.65	3.67	0.56	3.9	0.69	3.3	0.56
100x5	2.32	1.43	0.98	1.32	0.41	0.29	0.38	0.54	0.4	0.24	0.21	0.28
100x10	4.43	0.87	3.65	0.76	1.46	0.36	1.31	0.32	3.14	1.4	2.98	0.87
100x20	6.75	1.54	5.32	1.32	3.61	0.36	2.23	0.45	4.96	0.65	3.96	0.56
200x10	2.54	2.67	2.24	1.86	0.95	0.18	0.69	0.54	2.4	1.1	1.78	0.98
200x20	4.53	2.24	3.87	2.03	2.34	0.43	2.32	0.98	3.43	1.42	2.54	0.78
500x10	5.32	2.78	4.98	2.03	3.54	0.76	2.65	1.43	5.64	2.45	3.45	1.87

The results are tabulated in Table 6.6 as quality solutions with the percentage relative increase in makespan with respect to the upper bound provided by [44]. To be specific the formulation is given as:

$$\Delta_{avg} = \frac{(H - U) \times 100}{U} \tag{6.3}$$

where H denotes the value of the makespan that is produced by the utilized algorithm and U is the upper bound or the lower bound as computed.

From the presented results, it is evident that clustered heuristics perform better. The earlier trend continues in these problem instances, with $SOMA_{Clus}$ performing the best over the majority of the instances, followed by DE_{Clus} and GA_{Clus}. DE_{Clus} however performs better for the larger sized instances of 100 jobs. This is attributed to the fact that for the Taillard sets, as in the previous study of D[34], 2 opt local search was employed, and for consistency and comparison basis, local search was employed likewise in the clustered approach of DE_{Clus}.

The benefits of the clustered heuristics are not as marked as in the first set of instances, however on each problem class, an improvement is shown. The average improvements range from around 1% for GA to 0.4% for SOMA.

The second part of the comparison is done with the results obtained for the best-clustered heuristics of $SOMA_{Clus}$ and DE_{Clus} with those produced by GA, Particle Swarm Optimization $PSO_{spv}$and DE with local search $DE_{spv+exchange}$ as in [47] [46] and given in Table 6.7.

$SOMA_{Clus}$is the best performing heuristic in six instances (20x5, 20x10, 20x20, 50x5, 50x20) with DE_{Clus} obtaining better results in the other three instances (100x10, 100x20, 200x10) with one instance of 100x5 drawn and $DE_{spv+exchange}$ performing best in 50x10 instance. The advantage of $DE_{spv+exchange}$ is the fact that it employs local search, whereas $SOMA_{Clus}$does not. However, SOMA is using migration jumps, which also increases the search space fitness evaluations.

In terms of consistency, the average standard deviation of $SOMA_{Clus}$is below 1.0%. This is in line of $DE_{spv+exchange}$ and DE_{Clus}, which goes to show that these heuristics are reliable.

Table 6.7: Comparison of clustered heuristics with other published heuristics

	GA		PSO_{spv}		DE_{spv+ex}		DE_{Clus}		$SOMA_{Clus}$	
Problem	Δavg	Δstd	Δavg	Δstd	Δavg	Δstd	Δavg	Δstd	Δavg	Δstd
20x5	3.13	1.86	1.71	1.25	0.69	0.64	0.55	0.71	**0.39**	0.6
20x10	5.42	1.72	3.28	1.19	2.01	0.93	1.32	0.98	**1.28**	0.55
20x20	4.22	1.31	2.84	1.15	1.85	0.87	0.98	1.32	**0.96**	0.65
50x5	1.69	0.79	1.15	0.7	0.41	0.37	0.33	0.76	**0.32**	0.29
50x10	5.61	1.41	4.83	1.16	**2.41**	0.9	3.13	0.77	3.8	0.97
50x20	6.95	1.09	6.68	1.35	3.59	0.78	3.67	0.56	**3.3**	0.56
100x5	0.81	0.39	0.59	0.34	**0.21**	0.21	0.38	0.54	**0.21**	0.28
100x10	3.12	0.95	3.26	1.04	1.41	0.57	**1.31**	0.32	2.98	0.87
100x20	6.32	0.89	7.19	0.99	3.11	0.55	**2.23**	0.45	3.96	0.56
200x10	2.08	0.45	2.47	0.71	1.06	0.35	**0.69**	0.54	1.78	0.98

The DE results of this chapter have been published in [35], [11], [10], and the PSOMA results have been published in [12] and [13].

Chapter 7

Flow Shop Scheduling with Limited Intermediate Storage

Consider m machines in series with *zero* intermediate storage between sucessive machines. If a given machine finishes the processing of any given job, the job cannot proceed to the next machine while that machine is busy, but must remain on that machine, which therefore remians *idle*. This phenomenon is refered to as **blocking** [36].

In this section only flow shops with zero intermediate storage are considered since any flow shop with positive (but finite) intermediate storage between machines can be modeled as a flow shop with zero intermediate storage.This is due to the fact that the storage space capable of containing one job may be regarded as a machine on which the processing tme of all machines are equal to zero.

The problem of minimizing the makespan in a flow shop with zero intermediate storages is referred to in what follows as

$$Fm\,|block\,|C_{\max}$$

Let D_{ij} denote the time that job j actually departs machine i. Clearly $D_{ij} \geq C_{ij}$. Equality holds that job j is not blocked. The time job j starts its processing at the first machine id denoted by D_{0j}. The following recursive relationship hold under sequence $j_1, \ldots.., j_n$.

$$D_{i,j_1} = \sum_{l=1}^{i} p_{l,j_1} \tag{7.1}$$

$$D_{i,j_k} = \max\left(D_{i-1,j_k} + p_{i,j_k}, D_{i+1,j_{k-1}}\right) \tag{7.2}$$

$$D_{m,j_k} = D_{m-1,j_k} + p_{m,j_k} \tag{7.3}$$

The makespan can also be calculated by determining the critical path in the directed graph. In this graph, node (i, j_k) is the departure time of job j_k from machine i. In contrast with permutative flowshop in Chapter 6, in the graph the arcs, rather than the nodes, have weights. Node (i, j_k), $i = 1,...,m-1; k = 1,....,n-1$, has two outgoing arcs; one arc goes to node $(i+1, j_k)$ and has a weight or distance p_{i+1,j_k}, the other arc goes to node $(i-1, j_{k+1})$ and has weight zero. Node (m, j_k) has only one outgoing arc to node $(m-1, j_{k+1})$ with zero weight. Node (i, j_n) has only one outgoing arc to node $(i+1, j_n)$ with weight p_{i+1,j_n}. Node (m, j_n) has no outgoing arcs. The $C_{\max}$ under sequence $j_1, ..., j_n$ is equal to the length of the maximum weight path from node $(0, j_1)$ to node (m, j_n).

The directed graph is given in Figure 7.1.

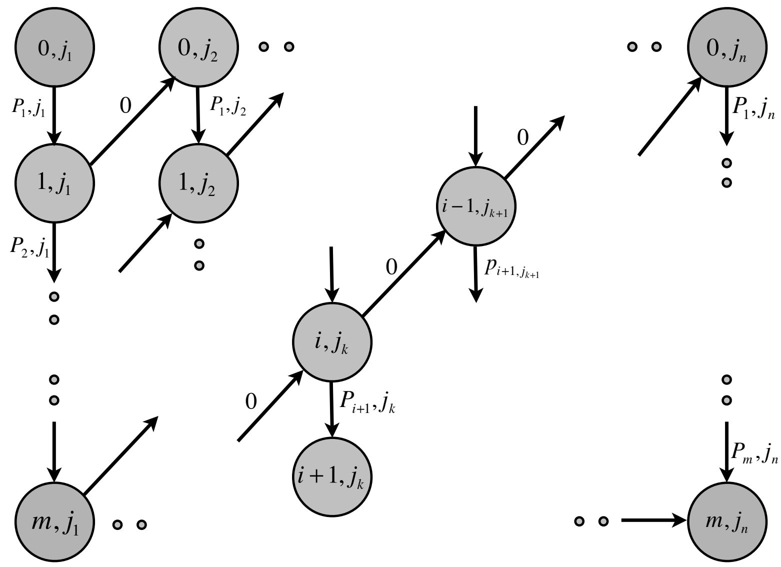

Figure 7.1: Directed graph representation for $Fm|block|C_{\max}$

7.1 Experimentation

The theme of this dissertation is the utilization of the Taillard Problem Sets [44] to solve the different scheduling problems. The flowshop problems from Chapter 6 are used for the simulations for *flowshop with blocking*. This approach allows for an analysis of the difference in makespan for the same problem utilized with different *restrictions*.

The experimentation for $Fm|block|C_{\max}$ was done in two parts.

The first section describes the evaluation of EDE with the taillard benchmark sets alongside that of clustered DE.

The second section outlines the procedure with P-SOMA.

The control parameters of the clustered population for both heuristics are given in Table 7.1.

The control parameters of SOMA and DE are presented in Table 7.2 and Table 7.3.

All parameters in Table 7.2 and Table 7.3 were obtained numerically.

Table 7.1: Population operating parameters

Parameter	Value
P_{size}	200 - 400
Generations	$>$ 250/SP
Clusters	4
C_A	$>$ 0.1

Table 7.2: P-SOMA operating parameters

Parameter	Range
MinJ	Dynamic
MaxJ	(0.2 - 0.5) x Problem size
Version	All-to-One

Table 7.3: DE operating parameters

Parameter	Value
F	0.3
CR	0.1

7.1.1 Differential Evolution

The Tailliard Problem Sets for Flowshop [44] have only been evaluated for "permutative flowshop scheduling" . The *lower bound* values for $Fm|block|C_{\max}$ is **not** provided. Therefore the raw values are provided for all 120 problem instances as the **first lower bound** evaluation of the $Fm|block|C_{\max}$ for DE. The results are tabulated in tables of problem size with avaerage and standard deviation of the specific instances provided. The results for DE and DE_{clust} are given in Tables 7.4 - 7.15.

Table 7.4: 20 job 5 machine $Fm|block|C_{\max}$

Instance	DE	DE_{clust}
Tai01	1694	**1583**
Tai02	1657	**1540**
Tai03	1687	**1502**
Tai04	1682	**1615**
Tai05	1632	**1517**
Tai06	1648	**1614**
Tai07	1685	**1592**
Tai08	1674	**1546**
Tai09	1693	**1540**
Tai10	1605	**1469**
Average	1665.7	**1551.8**
Std Dev	**29.657**	48.661

Table 7.5: 20 job 10 machine $Fm|block|C_{\max}$

Instance	DE	DE_{clust}
Tai11	2084	**1961**
Tai12	2147	**2042**
Tai13	2158	**1824**
Tai14	2047	**1785**
Tai15	1985	**1850**
Tai16	1978	**1809**
Tai17	1965	**1885**
Tai18	2144	**2022**
Tai19	2074	**1966**
Tai20	2106	**2052**
Average	2068.8	**1919.6**
Std Dev	**72.942**	101.321

Table 7.6: 20 job 20 machine $Fm|block|C_{\max}$

Instance	DE	DE_{clust}
Tai21	2698	**2673**
Tai22	2605	**2536**
Tai23	2755	**2692**
Tai24	2684	**2673**
Tai25	2705	**2698**
Tai26	2688	**2544**
Tai27	2610	**2566**
Tai28	2681	**2587**
Tai29	2704	**2662**
Tai30	2688	**2543**
Average	2681.8	**2617.4**
Std Dev	**44.456**	67.781

Table 7.7: 50 job 5 machine $Fm|block|C_{\max}$

Instance	DE	DE_{clust}
Tai31	3856	**3728**
Tai32	4055	**3908**
Tai33	4021	**3708**
Tai34	3965	**3803**
Tai35	3944	**3874**
Tai36	4021	**3848**
Tai37	3893	**3624**
Tai38	3864	**3779**
Tai39	3754	**3536**
Tai40	3952	**3800**
Average	3932.5	**3760.8**
Std Dev	**92.018**	114.914

Table 7.8: 50 job 10 machine $Fm|block|C_{\max}$

Instance	DE	DE_{clust}
Tai41	4306	**4298**
Tai42	4251	**4135**
Tai43	4316	**4196**
Tai44	4481	**4364**
Tai45	4439	**4344**
Tai46	4289	**4154**
Tai47	4455	**4334**
Tai48	4356	**4214**
Tai49	4387	**4282**
Tai50	4361	**4213**
Average	4364.1	**4253.4**
Std Dev	**76.201**	81.732

Table 7.9: 50 job 20 machine $Fm|block|C_{\max}$

Instance	DE	DE_{clust}
Tai51	5191	**5181**
Tai52	4965	**4868**
Tai53	5014	**4911**
Tai54	5048	**4979**
Tai55	5106	**4936**
Tai56	5110	**4996**
Tai57	5184	**4989**
Tai58	5174	**4946**
Tai59	5197	**4960**
Tai60	5163	**5007**
Average	5115.2	**4977.3**
Std Dev	**81.881**	83.127

Table 7.10: 100 job 5 machine $Fm|block|C_{\max}$

Instance	DE	DE_{clust}
Tai61	7865	**7659**
Tai62	7625	**7521**
Tai63	7251	**7179**
Tai64	7305	**7156**
Tai65	7548	**7460**
Tai66	7455	**7386**
Tai67	7694	**7508**
Tai68	7465	**7337**
Tai69	7821	**7740**
Tai70	7764	**7590**
Average	7579.3	**7453.6**
Std Dev	211.323	**192.208**

Table 7.11: 100 job 10 machine $Fm|block|C_{\max}$

Instance	DE	DE_{clust}
Tai71	8400	**8301**
Tai72	8355	**8122**
Tai73	8309	**8263**
Tai74	8641	**8511**
Tai75	8247	**8131**
Tai76	8264	**8107**
Tai77	8382	**8228**
Tai78	8259	**8195**
Tai79	8561	**8406**
Tai80	8457	**8389**
Average	8387.5	**8265.3**
Std Dev	**132.541**	136.100

Table 7.12: 100 job 20 machine $Fm|block|C_{\max}$

Instance	DE	DE_{clust}
Tai81	9198	**9104**
Tai82	9162	**9043**
Tai83	9058	**8956**
Tai84	9164	**9024**
Tai85	9173	**9077**
Tai86	9124	**9089**
Tai87	9226	**9117**
Tai88	9274	**9101**
Tai89	9192	**8983**
Tai90	9451	**9313**
Average	9202.2	**9080.7**
Std Dev	104.659	**97.780**

Table 7.13: 200 job 10 machine $Fm|block|C_{\max}$

Instance	DE	DE_{clust}
Tai91	16587	**16375**
Tai92	16354	**16049**
Tai93	16443	**16304**
Tai94	16985	**16368**
Tai95	16494	**16376**
Tai96	16478	**16134**
Tai97	16678	**16378**
Tai98	16531	**16371**
Tai99	16445	**16166**
Tai100	16543	**16416**
Average	16553.8	**16293.7**
Std Dev	175.262	**128.555**

Table 7.14: 200 job 20 machine $Fm|block|C_{\max}$

Instance	DE	DE_{clust}
Tai101	17204	**17005**
Tai102	17465	**17260**
Tai103	17356	**17204**
Tai104	17223	**17039**
Tai105	17239	**17164**
Tai106	17355	**17243**
Tai107	17648	**17527**
Tai108	17422	**17333**
Tai109	17389	**17203**
Tai110	17524	**17329**
Average	17382.5	**17230.7**
Std Dev	**140.741**	150.018

Table 7.15: 500 job 20 machine $Fm|block|C_{\max}$

Instance	DE	DE_{clust}
Tai111	42687	**41951**
Tai112	42151	**42363**
Tai113	42310	**41800**
Tai114	42573	**42107**
Tai115	42667	**42171**
Tai116	42981	**42372**
Tai117	42982	**42104**
Tai118	42236	**42015**
Tai119	42515	**41755**
Tai120	43517	**42474**
Average	42661.9	**42111.2**
Std Dev	412.562	**241.599**

The bolded values in each table represents the better heuristic for that specific problem instance. Upon analysis, it can be concluded that clustering of the population improves the heuristic, as all the problem instances had the DE_{clust} approach as the better performing heuristic.

7.1.2 Permutative Self Organising Migrating Algorithm

The results for PSOMA and $PSOMA_{clust}$ are also provided as the "raw" results for all 120 problem instances. This is the first evaluation of SOMA with $Fm|block|C_{\max}$ and hence is the first benchmark results for this problem class. The results are tabulated in Tables 7.16 - 7.27 where each table represents a specific problem size. The average and standard deviation values is given for each problem size. The bolded values is the better performing heuristic.

Table 7.16: 20 job 5 machine $Fm|block|C_{\max}$

Instance	PSOMA	$PSOMA_{clust}$
Tai01	1623	**1529**
Tai02	1614	**1568**
Tai03	1623	**1482**
Tai04	1684	**1673**
Tai05	1647	**1553**
Tai06	1664	**1578**
Tai07	1621	**1559**
Tai08	1605	**1544**
Tai09	1594	**1588**
Tai10	1598	**1457**
Average	1627.3	**1533.1**
Std Dev	**29.18**	59.09

Table 7.17: 20 job 10 machine $Fm|block|C_{\max}$

Instance	PSOMA	$PSOMA_{clust}$
Tai11	1956	**1911**
Tai12	2085	**2030**
Tai13	1962	**1882**
Tai14	1803	**1717**
Tai15	1825	**1806**
Tai16	1804	**1759**
Tai17	1865	**1830**
Tai18	2001	**1985**
Tai19	2058	**1917**
Tai20	2108	**1966**
Average	1946.7	**1880.3**
Std Dev	117.047	**101.346**

Table 7.18: 20 job 20 machine $Fm|block|C_{\max}$

Instance	PSOMA	$PSOMA_{clust}$
Tai21	2704	**2635**
Tai22	2546	**2462**
Tai23	2709	**2686**
Tai24	2777	**2606**
Tai25	2841	**2624**
Tai26	2647	**2564**
Tai27	2708	**2590**
Tai28	2664	**2556**
Tai29	2755	**2630**
Tai30	2648	**2573**
Average	2699.9	**2592.6**
Std Dev	81.323	**60.268**

Table 7.19: 50 job 5 machine $Fm|block|C_{\max}$

Instance	PSOMA	$PSOMA_{clust}$
Tai31	3845	**3781**
Tai32	3985	**3900**
Tai33	3844	**3710**
Tai34	3861	**3778**
Tai35	3916	**3856**
Tai36	3952	**3881**
Tai37	3754	**3690**
Tai38	3895	**3840**
Tai39	3645	**3590**
Tai40	3859	**3754**
Average	3855.6	**3778**
Std Dev	97.866	**96.446**

Table 7.20: 50 job 10 machine $Fm|block|C_{\max}$

Instance	PSOMA	$PSOMA_{clust}$
Tai41	4311	**4281**
Tai42	4251	**4143**
Tai43	4289	**4205**
Tai44	4351	**4302**
Tai45	4258	**4318**
Tai46	4374	**4279**
Tai47	4513	**4344**
Tai48	4366	**4200**
Tai49	4308	**4235**
Tai50	4320	**4302**
Average	4334.1	**4260.9**
Std Dev	75.31	**62.964**

Table 7.21: 50 job 20 machine $Fm|block|C_{\max}$

Instance	PSOMA	$PSOMA_{clust}$
Tai51	5212	**5176**
Tai52	5051	**4943**
Tai53	5132	**4951**
Tai54	5028	**4977**
Tai55	5203	**5010**
Tai56	5068	**4961**
Tai57	5124	**5021**
Tai58	5178	**5019**
Tai59	5134	**5047**
Tai60	5146	**5072**
Average	5127.6	**5017.7**
Std Dev	**62.45**	69.791

Table 7.22: 100 job 5 machine $Fm|block|C_{\max}$

Instance	PSOMA	$PSOMA_{clust}$
Tai61	7698	**7621**
Tai62	7605	**7515**
Tai63	7451	**7373**
Tai64	7308	**7288**
Tai65	7655	**7521**
Tai66	7546	**7453**
Tai67	7698	**7583**
Tai68	7642	**7505**
Tai69	7835	**7740**
Tai70	7884	**7723**
Average	7632.2	**7532.2**
Std Dev	169.939	**142.385**

Table 7.23: 100 job 10 machine $Fm|block|C_{\max}$

Instance	PSOMA	$PSOMA_{clust}$
Tai71	8545	**8415**
Tai72	8259	**8192**
Tai73	8437	**8303**
Tai74	8597	**8521**
Tai75	8351	**8266**
Tai76	8236	**8149**
Tai77	8317	**8277**
Tai78	8264	**8127**
Tai79	8467	**8378**
Tai80	8409	**8344**
Average	8388.2	**8297.2**
Std Dev	124.25	**122.97**

Table 7.24: 100 job 20 machine $Fm|block|C_{\max}$

Instance	PSOMA	$PSOMA_{clust}$
Tai81	9146	**9071**
Tai82	9178	**9081**
Tai83	9137	**9077**
Tai84	9087	**9077**
Tai85	9083	**8962**
Tai86	9168	**9094**
Tai87	9321	**9234**
Tai88	9258	**9195**
Tai89	9247	**9101**
Tai90	9367	**9265**
Average	9199.2	**9115.7**
Std Dev	96.002	**90.112**

Table 7.25: 200 job 10 machine $Fm|block|C_{\max}$

Instance	PSOMA	$PSOMA_{clust}$
Tai91	16784	**16333**
Tai92	16343	**16200**
Tai93	16432	**16317**
Tai94	16478	**16388**
Tai95	16984	**16318**
Tai96	16357	**16129**
Tai97	16594	**16480**
Tai98	16946	**16438**
Tai99	16528	**16149**
Tai100	16437	**16387**
Average	16595	**16313.9**
Std Dev	248.901	**119.52**

Table 7.26: 200 job 20 machine $Fm|block|C_{\max}$

Instance	PSOMA	$PSOMA_{clust}$
Tai101	17254	**17102**
Tai102	17554	**17423**
Tai103	17498	**17308**
Tai104	17402	**17265**
Tai105	17365	**17285**
Tai106	17487	**17366**
Tai107	17587	**17476**
Tai108	17448	**17325**
Tai109	17437	**17362**
Tai110	17447	**17330**
Average	17447.9	**17324.2**
Std Dev	**95.067**	100.383

Table 7.27: 500 job 20 machine $Fm|block|C_{\max}$

Instance	PSOMA	$PSOMA_{clust}$
Tai111	42874	**42261**
Tai112	42997	**42265**
Tai113	42551	**41950**
Tai114	42578	**42167**
Tai115	42651	**42087**
Tai116	43015	**42457**
Tai117	43170	**42059**
Tai118	42887	**41975**
Tai119	43008	**42006**
Tai120	43879	**42157**
Average	42961	**42138.4**
Std Dev	382.5	**157.197**

Upon analysis of all the instances, $PSOMA_{clust}$ is seen as the better performing heuristic, as it manages to find the better solution for all the problem instances.

7.2 Analysis

This section comapres the two better performing heuristics from the canonical and clustered approach in order to vet as to which is a better overall heuristic. From the previous results, DE_{clust} and $PSOMA_{clust}$ are the better performing heuristics, and are hense compared. The compared results are given in Tables 7.28 - 7.39. The bolded value is the better perfoming heuristic for the specific problem instance.

Table 7.28: 20 job 5 machine $Fm|block|C_{\max}$

Instance	DE_{clust}	$PSOMA_{clust}$
Tai 01	1583	**1529**
Tai 02	**1540**	1568
Tai 03	1502	**1482**
Tai 04	**1615**	1673
Tai 05	**1517**	1553
Tai 06	1614	**1578**
Tai 07	1592	**1559**
Tai 08	1546	**1544**
Tai 09	**1540**	1588
Tai 10	1469	**1457**
Average	**1551.8**	1553.1
Std Dev	**48.661**	59.09

Table 7.29: 20 job 10 machine $Fm|block|C_{\max}$

Instance	DE_{clust}	$PSOMA_{clust}$
Tai 11	1961	**1911**
Tai 12	2042	**2030**
Tai 13	**1824**	1882
Tai 14	1785	**1717**
Tai 15	1850	**1806**
Tai 16	1809	**1759**
Tai 17	1885	**1830**
Tai 18	2022	**1985**
Tai 19	1966	**1917**
Tai 20	2052	**1966**
Average	1919.6	**1880.3**
Std Dev	**101.321**	101.346

Table 7.30: 20 job 20 machine $Fm|block|C_{\max}$

Instance	DE_{clust}	$PSOMA_{clust}$
Tai 21	2673	**2635**
Tai 22	2536	**2462**
Tai 23	2692	**2686**
Tai 24	2673	**2606**
Tai 25	2698	**2624**
Tai 26	**2544**	2564
Tai 27	**2566**	2590
Tai 28	2587	**2556**
Tai 29	2662	**2630**
Tai 30	**2543**	2573
Average	2617.4	**2592.6**
Std Dev	67.781	**60.268**

Table 7.31: 50 job 5 machine $Fm|block|C_{\max}$

Instance	DE_{clust}	$PSOMA_{clust}$
Tai 31	**3728**	3781
Tai 32	3908	**3900**
Tai 33	**3708**	3710
Tai 34	3803	**3778**
Tai 35	3874	**3856**
Tai 36	**3848**	3881
Tai 37	**3624**	3690
Tai 38	**3779**	3840
Tai 39	**3536**	3590
Tai 40	3800	**3754**
Average	**3760.8**	3778
Std Dev	114.912	**96.446**

Table 7.32: 50 job 10 machine $Fm|block|C_{\max}$

Instance	DE_{clust}	$PSOMA_{clust}$
Tai 41	4298	**4281**
Tai 42	**4135**	4143
Tai 43	**4196**	4205
Tai 44	4364	**4302**
Tai 45	4344	**4318**
Tai 46	**4154**	4279
Tai 47	**4334**	4344
Tai 48	4214	**4200**
Tai 49	4282	**4235**
Tai 50	**4213**	4302
Average	**4253.4**	4256.33
Std Dev	81.732	**62.964**

Table 7.33: 50 job 20 machine $Fm|block|C_{\max}$

Instance	DE_{clust}	$PSOMA_{clust}$
Tai 51	5181	**5176**
Tai 52	**4868**	4943
Tai 53	**4911**	4951
Tai 54	4979	**4977**
Tai 55	**4936**	5010
Tai 56	4996	**4961**
Tai 57	**4989**	5021
Tai 58	**4946**	5019
Tai 59	**4960**	5047
Tai 60	**5007**	5072
Average	**4977.3**	5017.7
Std Dev	83.127	**69.79**

Table 7.34: 100 job 5 machine $Fm|block|C_{\max}$

Instance	DE_{clust}	$PSOMA_{clust}$
Tai 61	7659	**7621**
Tai 62	7521	**7515**
Tai 63	**7179**	7373
Tai 64	**7156**	7288
Tai 65	**7460**	7521
Tai 66	**7386**	7453
Tai 67	**7508**	7583
Tai 68	**7337**	7505
Tai 69	**7740**	**7740**
Tai 70	**7590**	7723
Average	**7453.6**	7532.2
Std Dev	192.208	**142.38**

Table 7.35: 100 job 10 machine $Fm|block|C_{\max}$

Instance	DE_{clust}	$PSOMA_{clust}$
Tai 71	**8301**	8415
Tai 72	**8122**	8192
Tai 73	**8263**	8303
Tai 74	**8511**	8521
Tai 75	**8131**	8266
Tai 76	**8107**	8149
Tai 77	**8228**	8277
Tai 78	8195	**8127**
Tai 79	8406	**8378**
Tai 80	8389	**8344**
Average	**8265.3**	8297.2
Std Dev	136.10	**122.97**

Table 7.36: 100 job 20 machine $Fm|block|C_{\max}$

Instance	DE_{clust}	$PSOMA_{clust}$
Tai 81	9104	**9071**
Tai 82	**9043**	9081
Tai 83	**8956**	9077
Tai 84	**9024**	9077
Tai 85	9077	**8962**
Tai 86	**9089**	9094
Tai 87	**9117**	9234
Tai 88	**9101**	9195
Tai 89	**8983**	9101
Tai 90	9313	**9265**
Average	**9080.7**	9115.7
Std Dev	97.78	**90.11**

Table 7.37: 200 job 10 machine $Fm|block|C_{\max}$

Instance	DE_{clust}	$PSOMA_{clust}$
Tai 91	16375	**16333**
Tai 92	**16049**	16200
Tai 93	**16304**	16317
Tai 94	**16368**	16388
Tai 95	16376	**16318**
Tai 96	16134	**16129**
Tai 97	**16378**	16480
Tai 98	**16371**	16438
Tai 99	16166	**16149**
Tai 100	16416	**16387**
Average	**16293.7**	16313.9
Std Dev	128.555	**119.52**

Table 7.38: 200 job 20 machine $Fm|block|C_{\max}$

Instance	DE_{clust}	$PSOMA_{clust}$
Tai101	17204	**17102**
Tai102	17465	**17423**
Tai103	17356	**17308**
Tai104	**17223**	17265
Tai105	**17239**	17285
Tai106	**17355**	17366
Tai107	17648	**17476**
Tai108	17422	**17325**
Tai109	17389	**17362**
Tai110	17524	**17330**
Average	17382.5	**17324.2**
Std Dev	140.741	**100.38**

Table 7.39: 500 job 20 machine $Fm|block|C_{\max}$

Instance	PSOMA	$PSOMA_{clust}$
Tai 111	**41951**	42261
Tai 112	42363	**42265**
Tai 113	**41800**	41950
Tai 114	**42107**	42167
Tai 115	42171	**42087**
Tai 116	**42372**	42457
Tai 117	42104	**42059**
Tai 118	42015	**41975**
Tai 119	**41755**	42006
Tai 120	42474	**42157**
Average	**42111.2**	42138.4
Std Dev	241.599	**157.19**

The summerised results are given in Table 7.40 for the average and standard deviation values. In general conclusions, DE_{clust} is the better overall heuristic having better overall values in 9 out of 12 problem classes. However, $PSOMA_{clust}$ provides better consistancy with better deviation values in the problem classes.

Table 7.40: DE_{clust} and $PSOMA_{clust}$ summerised results for $Fm|block|C_{\max}$

Instance		**Δavg**		**Δstd**	
job	*mach*	DE_{clust}	$PSOMA_{clust}$	DE_{clust}	$PSOMA_{clust}$
20	5	**1551.8**	1553.1	**48.661**	59.09
20	10	1919.6	**1880.3**	**101.321**	101.346
20	20	2617.4	**2592.6**	67.781	**60.268**
50	5	**3760.8**	3778	114.912	**96.446**
50	10	**4253.4**	4256.33	81.732	**62.964**
50	20	**4977.3**	5017.7	83.127	**69.79**
100	5	**7453.6**	7532.2	192.208	**142.38**
100	10	**8265.3**	8297.2	136.10	**122.97**
100	20	**9080.7**	9115.7	97.78	**90.11**
200	10	**16293.7**	16313.9	128.555	**119.52**
200	20	17382.5	**17324.2**	140.741	**100.38**
500	20	**42111.2**	42138.4	241.599	**157.19**

Chapter 8

Flow Shop Scheduling with No Wait

The third varient of flow shop is also the most challenging and practical [36]. Consider a flow shop with zero intermediate storage subject to different operating procedures. A job, when it goes through the system, is not allowed to *wait* at any machine. For this process, all susequent machines have to be *idle*, at the completion of the job on a machine upstream. This is the *opposite* to the *blocking* case where the jobs are pushed down by machines upstream. In this case the jobs are ***pulled*** down the line by machines which have become idle. This constraint is refered to as the **no-wait** constraint, and minimising the makespan in such a flow shop is referred to as the

$$Fm|nwt|C_{\max}$$

Among all types of scheduling problems, no-wait flowshop owns lots of important applications in different industries such as chemical processing [40], food processing [22], concrete ware production [21], and pharmaceutical processing [39] amongst others.

For the computational complexity of the no-wait flowshop scheduling problem, [19] proves that it is strongly NP-complete. Therefore, only small size instances of the no-wait flowshop problem can be solved with reasonable computational time by exact algorithms.

The no-wait flowshop scheduling problem can be described as follows: Given the processing times p_{jk} for job j and machine k, each of n jobs $(j = 1,2,..,n)$ will be sequenced through m machines $(k = 1,2,..,m)$ Each job j has a sequence of m operations $\left(o_{j1}, o_{j2}, ..., o_{jm}\right)$. To satisfy the no-wait restriction, the completion time of the operation o_{jk} must be equal to the earliest time to start of the operation $o_{j,k+1}$ for $k = 1,2,..,m-1$. In other words, there must not be any waiting times between the processing of any consecutive operation of each of n jobs. The problem is then to find a schedule such that the processing order of jobs is the same on each machine and the maximum completion time should be minimized.

Suppose that the job permutation $x = \{x_1, x_2, ..., x_n\}$ represents the schedule of jobs to be processed. Let $x = \{x_1, x_2, ..., x_n\}$ be the minimum delay on the first machine between the start of job x_j and x_{j-1} restricted by the no- wait constraint when the job x_j is directly processed after the job x_{j-1}. The minimum delay can be computed from the following expression:

$$d\left(x_j,x_{j-1}\right) = p\left(x_{j-1},1\right) + \max\left[0, \max_{2\le k\le m}\left\{\sum_{h=2}^{k} p\left(x_{j-1},h\right) - \sum_{h=1}^{k-1} p\left(x_j,h\right)\right\}\right] \quad (8.1)$$

Then the makespan can be defined as

$$C_{\max}(x) = \sum_{j=2}^{n} d\left(x_j,x_{j-1}\right) + \sum_{k=1}^{m} p\left(x_n,k\right) \quad (8.2)$$

The no-wait flowshop scheduling problem with the makespan criterion is to find a permutation x^* in the set of all permutations X such that

$$C_{\max}(x^*) \le C(x_n,m)\,\forall x \in X \quad (8.3)$$

8.1 Experimentation

The experimentation for $Fm|nwt|C_{\max}$ was done in two parts.

The first section describes the evaluation of DE with the Taillard benchmark sets alongside that of clustered DE. The second section outlines the procedure with P-SOMA, both with and without clustering.

The control parameters of the clustered population for both the experiments is given in Table 8.1.

Table 8.1: Population operating parameters

Parameter	Value
P_{size}	200 - 400
Generations	> 250/SP
Clusters	4
C_A	> 0.1

The control parameters of SOMA and DE are presented in Table 8.2 and Table 8.3.

Table 8.2: P-SOMA operating parameters

Parameter	Range
MinJ	Dynamic
MaxJ	(0.2 - 0.5) x Problem size
Version	All-to-One

All parameters in Table 8.2 and Table 8.3 were obtained numerically.

8.1.1 Differential Evolution

As described in the previous chapter, the Taillard flowshop sets have not been subjected to the $Fm|nwt|C_{\max}$, and the results presented in this section are the "raw"

Table 8.3: DE operating parameters

Parameter	Value
F	0.6
CR	0.1

values, which can be then used as the **benchmark** results for this specialized problem class. The results are presented in Tables 8.4 - 8.15. The bolded values are the better performing heuristic for that specific instance.

Table 8.4: 20 job 5 machine $Fm|nwt|C_{\max}$

Instance	DE	DE_{clust}
Tai01	1793	**1714**
Tai02	1853	**1753**
Tai03	1804	**1704**
Tai04	1984	**1967**
Tai05	1845	**1813**
Tai06	1920	**1878**
Tai07	1842	**1780**
Tai08	1877	**1847**
Tai09	1821	**1811**
Tai10	1842	**1681**
Average	1858.1	**1794.8**
Std Dev	**57.054**	87.929

Table 8.5: 20 job 10 machine $Fm|nwt|C_{\max}$

Instance	DE	DE_{clust}
Tai11	2649	**2516**
Tai12	2954	**2641**
Tai13	2545	**2189**
Tai14	2278	**2278**
Tai15	2603	**2481**
Tai16	2589	**2287**
Tai17	2455	**2216**
Tai18	2631	**2413**
Tai19	2442	**2285**
Tai20	2551	**2494**
Average	2569.7	**2380**
Std Dev	174.714	**149.947**

Table 8.6: 20 job 20 machine $Fm|nwt|C_{\max}$

Instance	DE	DE_{clust}
Tai21	3545	**3419**
Tai22	3612	**3318**
Tai23	3789	**3549**
Tai24	3541	**3287**
Tai25	3861	**3588**
Tai26	3854	**3550**
Tai27	3685	**3634**
Tai28	3512	**3342**
Tai29	3852	**3642**
Tai30	3752	**3666**
Average	3700.3	**3499.5**
Std Dev	**139.834**	144.659

Table 8.7: 50 job 5 machine $Fm|nwt|C_{\max}$

Instance	DE	DE_{clust}
Tai31	4212	**4158**
Tai32	4657	**4518**
Tai33	4365	**4220**
Tai34	4187	**4074**
Tai35	4388	**4291**
Tai36	4587	**4455**
Tai37	4215	**4094**
Tai38	4588	**4286**
Tai39	4256	**3943**
Tai40	4558	**4335**
Average	4401.3	**4237.4**
Std Dev	182.072	**176.82**

Table 8.8: 50 job 10 machine $Fm|nwt|C_{\max}$

Instance	DE	DE_{clust}
Tai41	5633	**5777**
Tai42	5568	**5438**
Tai43	5832	**5761**
Tai44	6105	**5986**
Tai45	5931	**5684**
Tai46	5887	**5706**
Tai47	6254	**6101**
Tai48	5861	**5712**
Tai49	5745	**5596**
Tai50	5848	**5703**
Average	5866.4	**5746.4**
Std Dev	203.188	**185.778**

Table 8.9: 50 job 20 machine $Fm|nwt|C_{\max}$

Instance	DE	DE_{clust}
Tai51	8052	**7968**
Tai52	7564	**7436**
Tai53	7965	**7832**
Tai54	8106	**8004**
Tai55	8154	**7939**
Tai56	8254	**8158**
Tai57	7936	**7850**
Tai58	7941	**7885**
Tai59	7968	**7753**
Tai60	8205	**8088**
Average	8014.5	**7891.3**
Std Dev	**194.94**	200.656

Table 8.10: 100 job 5 machine $Fm|nwt|C_{\max}$

Instance	DE	DE_{clust}
Tai61	9107	**8961**
Tai62	9004	**8608**
Tai63	9054	**8683**
Tai64	8578	**7881**
Tai65	8827	**8732**
Tai66	8964	**8733**
Tai67	8679	**8571**
Tai68	8752	**8501**
Tai69	9147	**9003**
Tai70	9106	**9040**
Average	8921.6	**8671.3**
Std Dev	**199.774**	334.192

Table 8.11: 100 job 10 machine $Fm|nwt|C_{\max}$

Instance	DE	DE_{clust}
Tai71	11589	**11406**
Tai72	11487	**11376**
Tai73	11985	**11587**
Tai74	11754	**11547**
Tai75	11287	**11135**
Tai76	11234	**11185**
Tai77	11884	**11771**
Tai78	11255	**10705**
Tai79	11859	**11665**
Tai80	11883	**11672**
Average	11621.7	**11404.9**
Std Dev	**290.382**	322.882

Table 8.12: 100 job 20 machine $Fm|nwt|C_{\max}$

Instance	DE	DE_{clust}
Tai81	16952	**16425**
Tai82	16151	**15187**
Tai83	16124	**15588**
Tai84	15875	**15142**
Tai85	15486	**14915**
Tai86	15214	**14727**
Tai87	15849	**15535**
Tai88	15879	**15266**
Tai89	16994	**16356**
Tai90	16552	**16303**
Average	16107.6	**15544.3**
Std Dev	**582.762**	618.664

Table 8.13: 200 job 10 machine $Fm|nwt|C_{\max}$

Instance	DE	DE_{clust}
Tai91	23201	**22733**
Tai92	23486	**22804**
Tai93	25611	**23527**
Tai94	25614	**24182**
Tai95	23581	**22350**
Tai96	22518	**22435**
Tai97	23118	**22503**
Tai98	24551	**23483**
Tai99	23198	**22303**
Tai100	23568	**22966**
Average	23844.6	**22928.6**
Std Dev	1061.689	**618.471**

Table 8.14: 200 job 20 machine $Fm|nwt|C_{\max}$

Instance	DE	DE_{clust}
Tai101	31056	**30489**
Tai102	31148	**30388**
Tai103	31089	**28859**
Tai104	32066	**31547**
Tai105	31254	**30665**
Tai106	32117	**31879**
Tai107	34512	**33248**
Tai108	31587	**30267**
Tai109	32628	**31534**
Tai110	31554	**30386**
Average	31901.1	**30926.2**
Std Dev	**1053.972**	1183.314

Table 8.15: 500 job 20 machine $Fm|nwt|C_{\max}$

Instance	DE	DE_{clust}
Tai111	81245	**80536**
Tai112	82224	**81383**
Tai113	80373	**75058**
Tai114	80662	**77444**
Tai115	78095	**75994**
Tai116	79661	**77026**
Tai117	79148	**77455**
Tai118	80154	**78600**
Tai119	78664	**76928**
Tai120	78984	**76771**
Average	79921	**77719.5**
Std Dev	**1261.475**	1953.431

Upon analysis, the clustered approach of DE, DE_{clust} is the better performing heuristic obtaining better values in every problem instance.

8.1.2 Permutative Self Organising Migrating Algorithm

The PSOMA results follow the same outline as the DE results with "raw" data presented for each instance. The results are given in Tables 8.19 - 8.27. The bolded values represent the better performing heuristic for that problem instance.

Table 8.16: 20 job 5 machine $Fm|nwt|C_{\max}$

Instance	PSOMA	$PSOMA_{clust}$
Tai01	1765	**1745**
Tai02	1705	**1694**
Tai03	1764	**1733**
Tai04	1895	**1868**
Tai05	1841	**1799**
Tai06	1818	**1798**
Tai07	1752	**1717**
Tai08	1826	**1810**
Tai09	1814	**1770**
Tai10	1654	**1605**
Average	1783.4	**1753.9**
Std Dev	**70.16**	73.06

Table 8.17: 20 job 10 machine $Fm|nwt|C_{\max}$

Instance	PSOMA	$PSOMA_{clust}$
Tai11	2504	**2429**
Tai12	2548	**2519**
Tai13	2465	**2303**
Tai14	2321	**2233**
Tai15	2405	**2336**
Tai16	2315	**2204**
Tai17	2467	**2407**
Tai18	2548	**2458**
Tai19	2406	**2320**
Tai20	2551	**2462**
Average	2453	**2367.1**
Std Dev	**89.178**	104.16

Table 8.18: 20 job 20 machine $Fm|nwt|C_{\max}$

Instance	PSOMA	$PSOMA_{clust}$
Tai21	3468	**3438**
Tai22	3405	**3366**
Tai23	3564	**3543**
Tai24	3864	**3700**
Tai25	3684	**3577**
Tai26	3741	**3610**
Tai27	3564	**3482**
Tai28	3452	**3344**
Tai29	3687	**3599**
Tai30	3654	**3507**
Average	3608.3	**3516.6**
Std Dev	143.903	**112.265**

Table 8.19: 50 job 5 machine $Fm|nwt|C_{\max}$

Instance	PSOMA	$PSOMA_{clust}$
Tai31	4448	**4365**
Tai32	4616	**4527**
Tai33	4283	**4234**
Tai34	4398	**4323**
Tai35	4563	**4485**
Tai36	4693	**4524**
Tai37	4464	**4363**
Tai38	4509	**4454**
Tai39	4207	**4108**
Tai40	4408	**4343**
Average	4458.9	**4372.6**
Std Dev	146.692	**132.803**

Table 8.20: 50 job 10 machine $Fm|nwt|C_{\max}$

Instance	PSOMA	$PSOMA_{clust}$
Tai41	5714	**5675**
Tai42	5694	**5568**
Tai43	5961	**5857**
Tai44	5997	**5910**
Tai45	6252	**6158**
Tai46	5966	**5829**
Tai47	6352	**6235**
Tai48	5964	**5845**
Tai49	5847	**5786**
Tai50	6152	**6080**
Average	5989.9	**5894.3**
Std Dev	214.039	**209.423**

Table 8.21: 50 job 20 machine $Fm|nwt|C_{\max}$

Instance	PSOMA	$PSOMA_{clust}$
Tai51	8625	**8539**
Tai52	8102	**8077**
Tai53	8424	**8313**
Tai54	8235	**8170**
Tai55	8546	**8401**
Tai56	8424	**8319**
Tai57	8150	**8006**
Tai58	8147	**8094**
Tai59	8269	**8174**
Tai60	8257	**8129**
Average	8317.9	**8222.2**
Std Dev	178.212	**165.979**

Table 8.22: 100 job 5 machine $Fm|nwt|C_{\max}$

Instance	PSOMA	$PSOMA_{clust}$
Tai61	9106	**8995**
Tai62	8759	**8689**
Tai63	8638	**8512**
Tai64	8694	**8590**
Tai65	9018	**8909**
Tai66	8967	**8892**
Tai67	9152	**9009**
Tai68	8692	**8571**
Tai69	9257	**9173**
Tai70	9382	**9221**
Average	8966.5	**8856.1**
Std Dev	261.221	**253.586**

Table 8.23: 100 job 10 machine $Fm|nwt|C_{\max}$

Instance	PSOMA	$PSOMA_{clust}$
Tai71	12084	**11965**
Tai72	11952	**11798**
Tai73	12102	**11964**
Tai74	13257	**12379**
Tai75	12085	**11935**
Tai76	12084	**11750**
Tai77	12345	**12243**
Tai78	11582	**11485**
Tai79	11984	**11805**
Tai80	12184	**12091**
Average	12165.9	**11941.5**
Std Dev	430.339	**256.048**

Table 8.24: 100 job 20 machine $Fm|nwt|C_{\max}$

Instance	PSOMA	*PSOMA$_{clust}$*
Tai81	17468	**16361**
Tai82	15937	**15787**
Tai83	16152	**16081**
Tai84	16558	**16113**
Tai85	16084	**15867**
Tai86	16007	**15996**
Tai87	16753	**16510**
Tai88	16225	**16132**
Tai89	16287	**16127**
Tai90	16794	**16682**
Average	16426.5	**16165.6**
Std Dev	472.814	**278.341**

Table 8.25: 200 job 10 machine $Fm|nwt|C_{\max}$

Instance	PSOMA	*PSOMA$_{clust}$*
Tai91	24158	**24051**
Tai92	24531	**23481**
Tai93	24515	**24401**
Tai94	25168	**24284**
Tai95	24681	**23990**
Tai96	24937	**23733**
Tai97	24967	**24453**
Tai98	24788	**24147**
Tai99	24987	**23698**
Tai100	25843	**24360**
Average	24857.5	**24081.1**
Std Dev	453.614	**345.55**

Table 8.26: 200 job 20 machine $Fm|nwt|C_{\max}$

Instance	PSOMA	*PSOMA$_{clust}$*
Tai101	24158	**31498**
Tai102	33442	**32970**
Tai103	34587	**31334**
Tai104	34688	**32072**
Tai105	33201	**32155**
Tai106	33874	**32836**
Tai107	34087	**33065**
Tai108	35045	**32082**
Tai109	34512	**33073**
Tai110	33781	**31904**
Average	33937.2	**32298.9**
Std Dev	853.6	**647.44**

Table 8.27: 500 job 20 machine $Fm|nwt|C_{\max}$

Instance	PSOMA	*PSOMA$_{clust}$*
Tai111	83547	**82312**
Tai112	84571	**83893**
Tai113	83514	**81677**
Tai114	83648	**81362**
Tai115	83957	**82102**
Tai116	83547	**82218**
Tai117	82657	**81451**
Tai118	83451	**82547**
Tai119	82514	**81547**
Tai120	82668	**81478**
Average	83407.4	**82058.7**
Std Dev	**638.943**	766.952

PSOMA$_{clust}$ is the better performing heuristic which obtains better value for each problem instance. It can be concluded that clustering the population is able to improve the performance of PSOMA.

8.2 Analysis

This section compares the two better performing heuristics from the canonical and clustered approach in order to vet as to which is a better overall heuristic. From the previous results, DE_{clust} and $PSOMA_{clust}$ are the better performing heuristics. The compared results are given in Tables 8.28 - 8.39. The bolded value is the better perfoming heuristic for the specific problem instance.

Table 8.28: 20 job 5 machine $Fm|nwt|C_{\max}$

Instance	DE_{clust}	$PSOMA_{clust}$
Tai 01	**1714**	1745
Tai 02	1753	**1694**
Tai 03	**1704**	1733
Tai 04	1967	**1868**
Tai 05	1813	**1799**
Tai 06	1878	**1798**
Tai 07	1780	**1717**
Tai 08	1847	**1810**
Tai 09	1811	**1770**
Tai 10	1681	**1605**
Average	1794.8	**1753.9**
Std Dev	87.929	**73.06**

Table 8.29: 20 job 10 machine $Fm|nwt|C_{\max}$

Instance	DE_{clust}	$PSOMA_{clust}$
Tai 11	2516	**2429**
Tai 12	2641	**2519**
Tai 13	**2189**	2303
Tai 14	2278	**2233**
Tai 15	2481	**2336**
Tai 16	2287	**2204**
Tai 17	**2216**	2407
Tai 18	**2413**	2458
Tai 19	**2285**	2320
Tai 20	2494	**2462**
Average	2380	**2367.1**
Std Dev	149.947	**104.160**

Table 8.30: 20 job 20 machine $Fm|nwt|C_{\max}$

Instance	DE_{clust}	$PSOMA_{clust}$
Tai 21	**3419**	3438
Tai 22	**3318**	3366
Tai 23	3549	**3543**
Tai 24	**3287**	3700
Tai 25	3588	**3577**
Tai 26	**3550**	3610
Tai 27	3634	**3482**
Tai 28	**3342**	3344
Tai 29	3642	**3599**
Tai 30	3666	**3507**
Average	**3499.5**	3516.6
Std Dev	144.65	**112.26**

Table 8.31: 50 job 5 machine $Fm|nwt|C_{\max}$

Instance	DE_{clust}	$PSOMA_{clust}$
Tai 31	**4158**	4365
Tai 32	**4518**	4527
Tai 33	**4220**	4234
Tai 34	**4074**	4323
Tai 35	**4291**	4485
Tai 36	**4455**	4524
Tai 37	**4094**	4363
Tai 38	**4286**	4454
Tai 39	**3943**	4108
Tai 40	**4335**	4343
Average	**4237.4**	4372.6
Std Dev	176.820	**132.803**

Table 8.32: 50 job 10 machine $Fm|nwt|C_{\max}$

Instance	DE_{clust}	$PSOMA_{clust}$
Tai 41	5777	**5675**
Tai 42	**5438**	5568
Tai 43	**5761**	5857
Tai 44	5986	**5910**
Tai 45	**5684**	6158
Tai 46	**5706**	5829
Tai 47	**6101**	6235
Tai 48	**5712**	5845
Tai 49	**5596**	5786
Tai 50	**5703**	6080
Average	**5746.4**	5894.3
Std Dev	**185.778**	209.423

Table 8.33: 50 job 20 machine $Fm|nwt|C_{\max}$

Instance	DE_{clust}	$PSOMA_{clust}$
Tai 51	**7968**	8539
Tai 52	**7436**	8077
Tai 53	**7832**	8313
Tai 54	**8004**	8170
Tai 55	**7939**	8401
Tai 56	**8158**	8319
Tai 57	**7850**	8006
Tai 58	**7885**	8094
Tai 59	**7753**	8174
Tai 60	**8088**	8129
Average	**7891.3**	8222.2
Std Dev	200.656	**165.979**

Table 8.34: 100 job 5 machine $Fm|nwt|C_{\max}$

Instance	DE_{clust}	$PSOMA_{clust}$
Tai 61	**8961**	8995
Tai 62	**8608**	8689
Tai 63	8683	**8512**
Tai 64	**7881**	8590
Tai 65	**8732**	8909
Tai 66	**8733**	8892
Tai 67	**8571**	9009
Tai 68	**8501**	8571
Tai 69	**9003**	9173
Tai 70	**9040**	9221
Average	**8671.3**	8856.1
Std Dev	334.192	**253.586**

Table 8.35: 100 job 10 machine $Fm|nwt|C_{\max}$

Instance	DE_{clust}	$PSOMA_{clust}$
Tai 71	**11406**	11965
Tai 72	**11376**	11798
Tai 73	**11587**	11964
Tai 74	**11547**	12379
Tai 75	**11135**	11935
Tai 76	**11185**	11750
Tai 77	**11771**	12243
Tai 78	**10705**	11485
Tai 79	**11665**	11805
Tai 80	**11672**	12091
Average	**11404.9**	11941.5
Std Dev	322.881	**256.048**

Table 8.36: 100 job 20 machine $Fm|nwt|C_{\max}$

Instance	DE_{clust}	$PSOMA_{clust}$
Tai 81	16425	**16361**
Tai 82	**15187**	15787
Tai 83	**15588**	16081
Tai 84	**15142**	16113
Tai 85	**14915**	15867
Tai 86	**14726**	15996
Tai 87	**15535**	16510
Tai 88	**15266**	16132
Tai 89	16356	**16127**
Tai 90	**16303**	16682
Average	**15544.3**	16165.6
Std Dev	618.663	**278.3419**

Table 8.37: 200 job 10 machine $Fm|nwt|C_{\max}$

Instance	DE_{clust}	$PSOMA_{clust}$
Tai 91	**22733**	24051
Tai 92	**22804**	23481
Tai 93	**23527**	24401
Tai 94	**24182**	24284
Tai 95	**22350**	23990
Tai 96	**22435**	23733
Tai 97	**22503**	24453
Tai 98	**23483**	24147
Tai 99	**22303**	23698
Tai 100	**22966**	24360
Average	**22928.6**	24059.8
Std Dev	618.471	**332.77**

Table 8.38: 200 job 20 machine $Fm|nwt|C_{\max}$

Instance	DE_{clust}	$PSOMA_{clust}$
Tai 101	**30489**	31498
Tai 102	**30388**	32970
Tai 103	**28859**	31334
Tai 104	**31547**	32072
Tai 105	**30665**	32155
Tai 106	**31879**	32836
Tai 107	33248	**33065**
Tai 108	**30267**	32082
Tai 109	**31534**	33073
Tai 110	**30386**	31904
Average	**30926.2**	32298.9
Std Dev	1183.314	**647.44**

Table 8.39: 500 job 20 machine $Fm|nwt|C_{\max}$

Instance	DE_{clust}	$PSOMA_{clust}$
Tai 111	**80536**	82312
Tai 112	**81383**	83893
Tai 113	**75058**	81677
Tai 114	**77444**	81362
Tai 115	**75994**	82102
Tai 116	**77026**	82218
Tai 117	**77455**	81451
Tai 118	**78600**	82547
Tai 119	**76928**	81547
Tai 120	**76771**	81478
Average	**77719.5**	82058.7
Std Dev	1953.431	**766.95**

The summerised results are given in Table 8.40 for the average and standard deviation values. In general conclusions, DE_{clust} is the better overall heuristic having better overall values in 10 out of 12 problem classes. DE_{clust} also performs better in larger problem sizes. However, $PSOMA_{clust}$ provides better consistancy with better deviation values in 11 out of 12 problem classes.

Table 8.40: DE_{clust} and $PSOMA_{clust}$ summerised results for $Fm|nwt|C_{\max}$

Instance		**Δavg**		**Δstd**	
job	*mach*	DE_{clust}	$PSOMA_{clust}$	DE_{clust}	$PSOMA_{clust}$
20	5	1794.8	**1753.9**	87.929	**73.06**
20	10	2380	**2367.1**	149.947	**104.160**
20	20	**3499.5**	3516.6	144.65	**112.26**
50	5	**4237.4**	4372.6	176.820	**132.803**
50	10	**5746.4**	5894.3	**185.778**	209.423
50	20	**7891.3**	8222.2	200.656	**165.979**
100	5	**8671.3**	8856.1	334.192	**253.586**
100	10	**11404.9**	11941.5	322.881	**256.048**
100	20	**15544.3**	16165.6	618.663	**278.3419**
200	10	**22928.6**	24059.8	618.471	**332.77**
200	20	**30926.2**	32298.9	1183.314	**647.44**
500	20	**77719.5**	82058.7	1953.431	**766.95**

Chapter 9

Quadratic Assignment Problem

QAP is an important problem in theory and practice. Formally, given n facilities and n locations, two n x n matrices $A = [a_{ij}]$ and $B = [b_{rs}]$, where a_{ij} is the distance between locations i and j and b_{rs} is the flow between facilities r and s, the QAP can be stated as follows:

$$\min_{\psi \varepsilon S(n)} \sum_{i=1}^{n} \sum_{j=1}^{n} b_{ij} a_{\psi_i \psi_j} \tag{9.1}$$

where *S(n)* is the set of all permutations (corresponding to the assignments) of the set of integers $\{1,\ldots,n\}$, and the ψ_i gives the location of facility i in the current solution $\psi \varepsilon S(n)$. Here $b_{ij} a_{\psi_i \psi_j}$ describes the cost distribution of simultaneously assigning facility i to location ψ_j and facility j to location ψ_i.

The term *quadratic* stems from the formulation of the QAP as an integer optimization problem with a quadratic objective function. Let x_{ij} be a binary variable which takes value 1 if facility i is assigned to location j and 0 otherwise. Then the problem can be formulated as:

$$\min \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{l=1}^{n} \sum_{k=1}^{n} a_{ij} b_{kl} x_{ik} x_{jl} \tag{9.2}$$

subject to the constraints

$$\sum_{i=1}^{n} x_{ij} = 1, \sum_{j=1}^{n} x_{ij} = 1, x \varepsilon \{0,1\} \tag{9.3}$$

According to [43], the QAP instances found in QAPLIB can be classified into four classes;

- **Unstructured, randomly generated instances**: Instances with the distance and flow matrix entries generated randomly according to an uniform distribution. The *taixxa* is an example of these instances, which are considered the most difficult to solve (we note that $x \equiv$ integer number).
- **Unstructured instances**: Instances with the grid matrix defined as the Manhattan distance between grid points on a n_1 x n_2 grid and with random flows.
- **Real-life instances**: 'Real-life' instances from practical application of the QAP. Amongst them are the layout problem of the hospital (*kra30x*)and instances corresponding to the layout of the typewriter keyboards (*bur26x*). The real-life instances have in common that the flow matrices have (in contrast to the previously

mentioned randomly generated instances) many zero entries and the entries are not uniformly distributed.

- **Real-life like instances**: Because the real life like instances are mainly of small size, [43] proposed the *taixxb* instances in such a way that they resemble the distribution found in real life problems.

In order to differentiate different classes of QAP the flow dominance *fd* is used. It is defined as the coefficient of the flow matrix entries multiplied by the factor of 100 and is represented as:

$$fd(B) = 100 \cdot \frac{\sigma}{\mu} \tag{9.4}$$

where

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

and

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

The general description is that unstructured randomly generated problems with a uniform distribution have a *fd* of less than 1.2 while real life structured problems have a *fd* larger than 1.2.

9.1 Experimentation

This section presents the results obtained from the three different sets of experimentations conducted. Each experiment was repeated 10 times with the same control values. The presented results are the best solutions obtained from these ten simulation on each instant.

All experimentation was conducted on an parallel array of 16 X-Serves with a total of 64 Quad Zeon processors all running on Grid Mathematica platform.

9.1.1 Genetic Algorithm Results

The first set of results is from Genetic Algorithms. The operational parameters of GA is given in Table 9.1.

Table 9.1: GA operational values

Parameter	**Value**
Strategy	2 Point Crossover
Mutation	Single
Population size	500 - 1000
Generations	500 - 1000

The generic and clustered GA results for the irregular problems is presented in Table 9.2.

Table 9.2: Clustered GA Irregular QAP comparison

Instant	**fd**	**n**	**Optimal**	**GA**	GA_{clust}
bur26a	2.75	26	5246670	1.64	**1.25**
bur26b	2.75	26	3817852	1.95	**1.34**
bur26c	2.29	26	5426795	1.75	**1.56**
bur26d	2.29	26	3821225	1.24	**1.21**
bur26e	2.55	26	5386879	1.52	**1.32**
bur26f	2.55	26	3782044	1.62	**1.56**
bur26g	2.84	26	10117172	1.53	**1.42**
bur26h	2.84	26	7098658	1.65	**1.54**
chr25a	4.15	26	3796	2.3	**1.56**
els19	5.16	19	17212548	0.94	**0.91**
kra30a	1.46	30	88900	1.23	**1.12**
kra30b	1.46	30	91420	1.64	**1.34**
tai20b	3.24	20	122455319	1.58	**1.21**
tai25b	3.03	25	344355646	1.61	**0.94**
tai30b	3.18	30	637117113	2.19	**1.24**
tai35b	3.05	35	283315445	2.32	**0.85**
tai40b	3.13	40	637250948	2.54	**1.12**
tai50b	3.1	50	458821517	2.75	**1.24**
tai60b	3.15	60	608215054	2.68	**1.52**
tai80b	3.21	80	818415043	3.11	**1.95**

Table 9.3: Clustered GA Regular QAP comparison

Instant	**fd**	**n**	**Optimal**	**GA**	GA_{clust}
nug20	0.99	20	2570	0.98	**0.85**
nug30	1.09	30	6124	0.84	**0.82**
sko42	1.06	42	15812	0.95	**0.84**
sko49	1.07	49	23386	1.12	**0.93**
sko56	1.09	56	34458	1.35	**0.94**
sko64	1.07	64	48498	1.68	**1.23**
sko72	1.06	72	66256	2.52	**1.54**
sko81	1.05	81	90998	3.21	**2.15**
tai20a	0.61	20	703482	0.98	**0.52**
tai25a	0.6	25	1167256	**0.68**	**0.68**
tai30a	0.59	30	1818146	1.02	**0.95**
tai35a	0.58	35	2422002	1.32	**0.98**
tai40a	0.6	40	3139370	1.54	**1.22**
tai50a	0.6	50	4941410	1.62	**1.31**
tai60a	0.6	60	7208572	2.13	**1.98**
tai80a	0.59	80	13557864	3.21	**2.35**
wil50	0.64	50	48816	1.89	**0.98**

The results of the regular problems in given in Table 9.3.
The results clearly demonstrate that using clustering improves the results of generic

GA. Even though the results obtained for GA are not as competitive for the QAP instances, the main idea of this research of clustering of the population to improve the performance of metaheuristics is validated.

9.1.2 Differential Evolution Results

The second experiment is conducted with Differential Evolution algorithm. Extensive experimentation was conducted with both the regular and irregular QAP problems. Comaprison is done with the DE heuristic without clustering [11].

The operational parameters of DE are given in Table 9.4.

Table 9.4: DE operational values

Parameter	**Value**
Strategy	DE/rand/2/bin
CR	0.9
F	0.3
Population	500 - 1000
Generation	500 - 1000

The first part of the results is on the irregular QAP instances. The results are presented in Table 9.5. The columns represent the name of the problem, its flow dominance, problem size, optimal reported value, DE result and DE with clustering result.

Table 9.5: Clustered DE Irregular QAP comparison

Instant	**fd**	**n**	**Optimal**	**DE**	DE_{clust}
bur26a	2.75	26	5246670	0.006	**0**
bur26b	2.75	26	3817852	0.0002	**0**
bur26c	2.29	26	5426795	0.00005	**0**
bur26d	2.29	26	3821225	0.0001	**0**
bur26e	2.55	26	5386879	0.0002	**0**
bur26f	2.55	26	3782044	0.000001	**0**
bur26g	2.84	26	10117172	0.0001	**0**
bur26h	2.84	26	7098658	0.0001	**0**
chr25a	4.15	26	3796	0.227	**0.07**
els19	5.16	19	17212548	0.0007	**0**
kra30a	1.46	30	88900	0.0328	**0.024**
kra30b	1.46	30	91420	0.0253	**0.015**
tai20b	3.24	20	122455319	0.0059	**0**
tai25b	3.03	25	344355646	0.003	**0**
tai30b	3.18	30	637117113	0.0239	**0**
tai35b	3.05	35	283315445	0.0101	**0.002**
tai40b	3.13	40	637250948	0.027	**0**
tai50b	3.1	50	458821517	0.001	**0**
tai60b	3.15	60	608215054	0.0144	**0.012**
tai80b	3.21	80	818415043	0.0287	**0.014**

Comparing the results of DE and DE_{clust}, it is easy to see that DE_{clust} performs better than DE. Of the 8 bur*xx* instances, the optimal result is obatined for all instances. On the kra*xx* and tai*xx* instances, DE_{clust} outperforms DE marginally.

The second part of the results is on the regular QAP instances as given in Table 9.6.

Table 9.6: Clustered DE Regular QAP comparison

Instant	**fd**	**n**	**Optimal**	**DE**	DE_{clust}
nug20	0.99	20	2570	0.018	**0**
nug30	1.09	30	6124	0.005	**0**
sko42	1.06	42	15812	0.009	**0**
sko49	1.07	49	23386	0.009	**0**
sko56	1.09	56	34458	0.012	**0**
sko64	1.07	64	48498	0.013	**0.006**
sko72	1.06	72	66256	0.011	**0.007**
sko81	1.05	81	90998	0.011	**0.01**
tai20a	0.61	20	703482	0.037	**0**
tai25a	0.6	25	1167256	0.026	**0**
tai30a	0.59	30	1818146	0.018	**0**
tai35a	0.58	35	2422002	0.038	**0**
tai40a	0.6	40	3139370	0.032	**0.019**
tai50a	0.6	50	4941410	0.033	**0.026**
tai60a	0.6	60	7208572	0.037	**0.012**
tai80a	0.59	80	13557864	0.031	**0.021**
wil50	0.64	50	48816	0.004	**0**

DE_{clust} outperfoms DE in regular QAP instances. It manages to find 10 optimal instances out of the 16 tested. Of the remaining 6, DE_{clust} obtains close to 0.01% to the optimal.

9.1.3 Self Organising Migration Algorithm Results

The third and final experiment was conducted with SOMA. The operational parameters of SOMA is given in Table 9.7.

Table 9.7: SOMA operational values

Parameter	**Value**
Strategy	All-to-All
Step Size	0.21
PathLength	3
Population	500 - 1000
Migration	500 - 1000

The results are compared with those of SOMA without clustering of [13] and is given in Table 9.8.

The results of clustered SOMA with regular problems is given in Table 9.9.

Table 9.8: Clustered SOMA Irregular QAP comparison

Instant	**fd**	**n**	**Optimal**	**SOMA**	$SOMA_{clust}$
bur26a	2.75	26	5246670	**0**	**0**
bur26b	2.75	26	3817852	**0**	**0**
bur26c	2.29	26	5426795	**0**	**0**
bur26d	2.29	26	3821225	**0**	**0**
bur26e	2.55	26	5386879	**0**	**0**
bur26f	2.55	26	3782044	0.03	**0.01**
bur26g	2.84	26	10117172	**0**	**0**
bur26h	2.84	26	7098658	**0**	**0**
chr25a	4.15	26	3796	0.129	**0.10**
els19	5.16	19	17212548	**0**	**0**
kra30a	1.46	30	88900	**0.002**	**0.002**
kra30b	1.46	30	91420	0.03	**0.027**
tai20b	3.24	20	122455319	0.004	**0**
tai25b	3.03	25	344355646	**0**	**0**
tai30b	3.18	30	637117113	0.043	**0**
tai35b	3.05	35	283315445	**0**	**0**
tai40b	3.13	40	637250948	0.02	**0**
tai50b	3.1	50	458821517	0.2	**0.2**
tai60b	3.15	60	608215054	0.5	**0.2**
tai80b	3.21	80	818415043	0.8	**0.4**

Table 9.9: Clustered SOMA Regular QAP comparison

Instant	**fd**	**n**	**Optimal**	**SOMA**	$SOMA_{clust}$
nug20	0.99	20	2570	**0**	**0**
nug30	1.09	30	6124	0.02	**0**
sko42	1.06	42	15812	0.01	**0**
sko49	1.07	49	23386	0.005	**0**
sko56	1.09	56	34458	0.01	**0**
sko64	1.07	64	48498	0.06	**0.02**
sko72	1.06	72	66256	0.2	**0.04**
sko81	1.05	81	90998	0.35	**0.05**
tai20a	0.61	20	703482	**0**	**0**
tai25a	0.6	25	1167256	**0**	**0**
tai30a	0.59	30	1818146	0.01	**0**
tai35a	0.58	35	2422002	0.03	**0**
tai40a	0.6	40	3139370	0.623	**0.58**
tai50a	0.6	50	4941410	0.645	**0.42**
tai60a	0.6	60	7208572	0.62	**0.62**
tai80a	0.59	80	13557864	1.05	**0.95**
wil50	0.64	50	48816	**0**	**0**

9.2 Analysis

Comparison of the obtained results is done with some published heuristics. The first comparison is done with the irregular QAP instances. The two best performing results of DE_{clust} and $SOMA_{clust}$ is compared with the Improved Hybrid Genetic Algorithm of [29] shown as GA_1 and the highly refereed Ant Colony approach of [18] given as HAS in Table 9.10.

Table 9.10: Irregular QAP comparison

Instant	**fd**	**n**	**Optimal**	GA_1	**HAS**	DE_{clust}	$SOMA_{clust}$
bur26a	2.75	26	5246670	-	**0**	**0**	**0**
bur26b	2.75	26	3817852	-	**0**	**0**	**0**
bur26c	2.29	26	5426795	-	**0**	**0**	**0**
bur26d	2.29	26	3821225	-	**0**	**0**	**0**
bur26e	2.55	26	5386879	-	**0**	**0**	**0**
bur26f	2.55	26	3782044	-	**0**	**0**	0.01
bur26g	2.84	26	10117172	-	**0**	**0**	**0**
bur26h	2.84	26	7098658	-	**0**	**0**	**0**
chr25a	4.15	26	3796	-	3.082	**0.07**	0.10
els19	5.16	19	17212548	-	**0**	**0**	**0**
kra30a	1.46	30	88900	**0**	0.629	0.024	0.002
kra30b	1.46	30	91420	**0**	0.071	0.015	0.027
tai20b	3.24	20	122455319	-	0.091	**0**	**0**
tai25b	3.03	25	344355646	-	**0**	**0**	**0**
tai30b	3.18	30	637117113	-	**0**	**0**	**0**
tai35b	3.05	35	283315445	-	0.025	0.002	**0**
tai40b	3.13	40	637250948	-	**0**	**0**	**0**
tai50b	3.1	50	458821517	-	0.192	**0**	0.2
tai60b	3.15	60	608215054	-	0.048	**0.012**	0.2
tai80b	3.21	80	818415043	-	0.667	**0.014**	0.4

The best perfomring algorithm is DE_{clust} which obtains the best comparitive result in 17 out of 20 problem instances. $SOMA_{clust}$ obtains the best results in 13 instances and HAS in 12 instances. The hybrid Genetic Algorithm appproach however is able to find the optimal result in the two instances that it is applied, where the other heuristics are not so effective. For the larger size problems, DE_{clust} proves to be a better optimizer.

The second set of comparison is done with the regular QAP instances. Comparison of the clustered SOMA and DE is done with the GA (GA_1) approach of [29], greedy GA (GA_{Greedy}) of [1], GA (GA_2) of [17], Simulated Annealing algorithm (TB2M) of [4], Robust Tabu Search (RTS) of [43], Combined Simulated Annealing and Tabu Search (IA-SA-TS) of [31] and Ant Colony (HAS) of [18]. The results are given in Table 9.11.

As with the irregular problem, DE_{clust} is the best performing algorithm. It manages to find the best value in 16 out of 17 instances, of which 10 are optimal values. $SOMA_{clust}$ is the second best heuristic with 10 best solutions, all of which are optimal values of those particular problems.

The DE results of this chapter have been published in [15] and the PSOMA results have been published in [16].

Table 9.11: Regular QAP comparison

Instant	**fd**	**n**	**Optimal**	GA_1	GA_{Greedy}	GA_2	**TB2M**	**RTS**	**IA-SA-TS**	**HAS**	DE_{clust}	$SOMA_{clust}$
nug20	0.99	20	2570	-	-	-	-	-	-	**0**	**0**	**0**
nug30	1.09	30	6124	**0**	0.07	**0**	0.94	0.73	0.52	0.098	**0**	**0**
sko42	1.06	42	15812	**0**	0.250	**0**	0.66	1.03	0.46	0.076	**0**	**0**
sko49	1.07	49	23386	0.038	0.210	0.009	0.67	0.54	0.46	0.141	**0**	**0**
sko56	1.09	56	34458	**0**	0.02	0.001	0.66	0.53	0.50	0.101	**0**	**0**
sko64	1.07	64	48498	**0**	0.22	**0**	0.57	0.93	0.45	0.504	0.006	0.02
sko72	1.06	72	66256	0.042	0.29	0.014	0.60	0.52	0.48	0.702	**0.007**	0.04
sko81	1.05	81	90998	0.067	0.2	0.014	0.46	0.41	0.40	0.493	**0.01**	0.05
tai20a	0.61	20	703482	-	-	-	-	-	-	0.675	**0**	**0**
tai25a	0.6	25	1167256	-	-	-	-	-	-	1.189	**0**	**0**
tai30a	0.59	30	1818146	-	-	-	-	-	-	1.311	**0**	**0**
tai35a	0.58	35	2422002	-	-	-	-	-	-	1.762	**0**	**0**
tai40a	0.6	40	3139370	-	-	-	-	-	-	1.989	**0.019**	0.58
tai50a	0.6	50	4941410	-	-	-	-	-	-	2.8	**0.026**	0.42
tai60a	0.6	60	7208572	-	-	-	-	-	-	0.313	**0.012**	0.62
tai80a	0.59	80	13557864	-	-	-	-	-	-	1.108	**0.021**	0.95
wil50	0.64	50	48816	0.028	0.07	0.002	0.25	0.55	0.16	0.061	**0**	**0**

Chapter 10

Capacitated Vehicle Routing Problem

The Vehicle Routing Problem (VRP) introduced for the first time by [7] is a complex combinatorial optimization problem, which can be seen as a merge of two well-known problems: the Traveling Salesperson Problem (TSP) and the Bin Packing Problem (BPP).

It can simply be described as follows: given a fleet of vehicles with uniform capacity, a common depot, and several costumer demands, find the set of routes with overall minimum route cost which service all the demands.

Assume a quantity d_i of a single commodity which is to be delivered to each customer $i \in N = \{1, ..., n\}$ from a central depot $\{0\}$ using k independent delivery vehicles of identical capacity C . Delivery is to be accomplished at minimum total cost, with $c_{ij} \geq 0$ denoting the transit cost from i to j , for $0 \leq i, j \leq n$. The cost structure is assumed *symmetric*, i.e., $c_{ij} = c_{ji}$ and $c_{ii} = 0$.

Combinatorially, a solution for this problem consists of a partition of N into k *routes* $\{R_1, ..., R_k\}$, each satisfying $\sum_{j \in R_i} d_j \leq C$, and a corresponding permutation σ_i of each route specifying the service ordering. This problem is naturally associated with the complete undirected graph consisting of nodes $N \cup \{0\}$, edges E, and edge-traversal costs $c_{ij}, \{i, j\} \in E$. In this graph, a solution is the union of k cycles whose only intersection is the depot node. Each cycle corresponds to the route serviced by one of the k vehicles. By associating a binary variable with each edge in the graph, the following integer programming formulation is obtained:

$$\min \sum_{e \in E} c_e x_e$$

$$\sum_{e=\{0,j\} \in E} x_e = 2k \tag{10.1}$$

$$\sum_{e=\{i,j\} \in E} x_e = 2 \;\; \forall i \in N \tag{10.2}$$

$$\sum_{\substack{e=\{i,j\} \in E \\ i \in S, j \notin S}} x_e \geq 2b(S) \;\; \forall S \subset N, |S| > 1 \tag{10.3}$$

$$0 \leq x_e \leq 1 \;\; \forall e = \{i, j\} \in E, \;\; i, j \neq 0 \tag{10.4}$$

$$0 \leq x_e \leq 2 \;\; \forall e = \{0, j\} \in E \tag{10.5}$$

$$x_e \; integral \;\; \forall e \in E \tag{10.6}$$

For ease of computation, $b(S) = \left\lceil \frac{(\sum_{i \in S} d_i)}{C} \right\rceil$ is defined as an obvious lower bound on the number of trucks needed to service the customers in set S. Constraints 10.1 and 10.2 are the degree constraints. Constraints 10.3 is a generalization of the subtour elimination constraints from the TSP and serves to enforce the connectivity of the solution, as well as to ensure that no route has total demand exceeding the capacity C . A (possibly) stronger inequality may be obtained by computing the solution to a Bin Packing Problem (BPP) with the customer demands in set S being packed into bins of size C. Equation 10.3 is the *capacity constraints*.

It is clear from the description that the VRP is closely related to two difficult combinatorial problems. By setting $C = \infty$, the Multiple Traveling Salesman Problem (MTSP) is obtained. An MTSP instance can be transformed into an equivalent TSP instance by adjoining to the graph $k-1$ additional copies of node 0 and its incident edges (there are no edges among the k depot nodes). On the other hand, the question of whether there exists a feasible solution for a given instance of the VRP is an instance of the BPP. The decision version of this problem is conceptually equivalent to a VRP model in which all edge costs are taken to be zero (so that all feasible solutions have the same cost). Hence, the first transformation can be seen as the relaxing the underlying packing (BPP) structure and the second transformation as relaxing the underlying routing (TSP) structure. A feasible solution to the full problem is a TSP tour (in the expanded graph) that also satisfies the packing constraints (i.e., that the total demand along each of the k segments joining successive copies of the depot does not exceed C).

Because of the interplay between the two underlying models, instances of the Vehicle Routing Problem can be extremely difficult to solve in practice. In fact, the largest solvable instances of the VRP are two orders of magnitude smaller than those of the TSP. Exact solution of the VRP thus presents an interesting challenge.

10.1 Experimentation

As with all the other problem classes, experimentation for CVRP was done in two parts.

The first section describes the evaluation of EDE with the Taillard benchmark sets alongside that of clustered DE.

The second section outlines the procedure with P-SOMA.

The control parameters of the clustered population for both are given in Table 10.1.

Table 10.1: Population operating parameters

Parameter	Value
P_{size}	200 - 400
Generations	$>$ 250/SP
Clusters	4
C_A	$>$ 0.1

The control parameters of SOMA and DE are presented in Table 10.2 and Table 10.3.

Table 10.2: P-SOMA operating parameters

Parameter	Range
MinJ	Dynamic
MaxJ	(0.2 - 0.5) x Problem size
Version	All-to-One

Table 10.3: DE operating parameters

Parameter	Value
F	0.6
CR	0.1

All parameters in Table 10.2 and Table 10.3 were obtained numerically.

10.1.1 Differential Evolution Algorithm

A total of 12 problems of the Taillard sets have been experimented. Three different sets exist of four instances of size 75, 100 and 150. The results of canonical and clustered DE are given in Tables 10.4 - 10.6. The bolded values are the best results for that particular instance. The average and standard deviation values are also provided.

Table 10.4: DE VRP 75 tour result

Instance	n	Optimal	DE	DE_{clust}
Tai75a	75	1618.36	1.391	**1.065**
Tai75b	75	1344.62	0.955	**0.828**
Tai75c	75	1291.01	1.401	**1.168**
Tai75d	75	1365.24	1.258	**0.825**
Average			1.251	**0.972**
Std Dev			**0.208**	0.172

Table 10.5: DE VRP 100 tour result

Instance	n	Optimal	DE	DE_{clust}
Tai100a	100	2041.34	1.562	**1.294**
Tai100b	100	1940.61	1.579	**1.173**
Tai100c	100	1406.2	1.475	**1.419**
Tai100d	100	1581.25	1.556	**1.170**
Average			1.543	**1.264**
Std Dev			**0.046**	0.118

Table 10.6: DE VRP 150 tour result

Instance	n	Optimal	DE	DE_{clust}
Tai150a	150	3055.23	2.184	**2.055**
Tai150b	150	2656.47	2.204	**1.833**
Tai150c	150	2341.84	1.991	**1.904**
Tai150d	150	2645.39	2.225	**1.688**
Average			2.150	**1.870**
Std Dev			**0.107**	0.152

The clustered approach of DE, DE_{clust} is the better performing heuristic, obtaining the better value for each problem instance.

10.1.2 Permutative Self Organising Migrating Algorithm

An identical experimentation procedure as the the one described for DE was conducted for PSOMA. The results are tabulated in Tables 10.7 - 10.9, grouped in accordance to their sizes.

Table 10.7: PSOMA VRP 75 tour result

Instance	n	Optimal	PSOMA	$PSOMA_{clust}$
Tai75a	75	1618.36	0.932	**0.928**
Tai75b	75	1344.62	1.005	**0.754**
Tai75c	75	1291.01	1.214	**1.181**
Tai75d	75	1365.24	1.104	**0.950**
Average			1.064	**0.953**
Std Dev			**0.122**	0.175

Table 10.8: PSOMA VRP 100 tour result

Instance	n	Optimal	PSOMA	$PSOMA_{clust}$
Tai100a	100	2041.34	1.688	**1.144**
Tai100b	100	1940.61	1.605	**1.467**
Tai100c	100	1406.2	1.699	**1.414**
Tai100d	100	1581.25	1.476	**1.459**
Average			1.617	**1.371**
Std Dev			**0.103**	0.152

Table 10.9: PSOMA VRP 150 tour result

Instance	n	Optimal	PSOMA	$PSOMA_{clust}$
Tai150a	150	3055.23	2.146	**1.772**
Tai150b	150	2656.47	2.479	**2.217**
Tai150c	150	2341.84	2.145	**1.962**
Tai150d	150	2645.39	2.102	**1.743**
Average			2.218	**1.924**
Std Dev			**0.175**	0.218

As with DE_{clust}, the clustered approach of $PSOMA_{clust}$ is the better performing heuristic, finding better values in all problem instances.

10.2 Analysis

The analysis is done with DE_{clust} and $PSOMA_{clust}$ for the VRP. The results are given in Tables 10.10 - 10.12.

Table 10.10: DE_{clust} $PSOMA_{clust}$ VRP 75 tour result comparison

Instance	n	Optimal	DE_{clust}	$PSOMA_{clust}$
Tai75a	75	1618.36	1.065	**0.928**
Tai75b	75	1344.62	0.828	**0.754**
Tai75c	75	1291.01	**1.168**	1.181
Tai75d	75	1365.24	**0.825**	0.950
Average			0.972	**0.953**
Std Dev			**0.172**	0.175

Table 10.11: DE_{clust} $PSOMA_{clust}$ VRP 100 tour result comparison

Instance	n	Optimal	DE_{clust}	$PSOMA_{clust}$
Tai100a	100	2041.34	1.294	**1.144**
Tai100b	100	1940.61	**1.173**	1.467
Tai100c	100	1406.2	1.419	**1.414**
Tai100d	100	1581.25	**1.170**	1.459
Average			**1.264**	1.371
Std Dev			**0.118**	0.152

Table 10.12: DE_{clust} $PSOMA_{clust}$ VRP 150 tour result comparison

Instance	n	Optimal	DE_{clust}	$PSOMA_{clust}$
Tai150a	150	3055.23	2.055	**1.772**
Tai150b	150	2656.47	**1.833**	2.217
Tai150c	150	2341.84	**1.904**	1.962
Tai150d	150	2645.39	**1.688**	1.743
Average			**1.870**	1.924
Std Dev			**0.152**	0.218

The results are almost evenly split between DE_{clust} and $PSOMA_{clust}$. $PSOMA_{clust}$ obtains 5 out of 12 better results and DE_{clust} obtains 7 out of 12. However, DE_{clust} is a better performing heuristic in the larger problem instances, with better average and standard deviation values.

Chapter 11

Job Shop Scheduling

A ***job shop*** problem (JSP) is different from a *flow shop* scheduling problem, in which all job follow the same route. In the JSP, the *route* of the job is fixed, however not necessarily the same for each job. If a job has to visit certain machines more than once, the job is said to recirculate [36]. This chapter deals with jobs which do not recirculate. The problem designation is

$$Jm \parallel C_{\max}$$

The JSP can be described by a set of n jobs $\{J_i\}_{1 \le j \le n}$ which is to be processed on a set of m machines $\{M_r\}_{1 \le r \le m}$. The problem can be characterized as follows:

1. Each job must be processed on each machine in the order given in a pre-defined technological sequence of machines.
2. Each machine can process only one job at a time.
3. The processing of job J_j on machine M_r is called the operation O_{jr}.
4. Operation O_{jr} requires the exclusive use of M_r for an uninterrupted duration p_{jr}, its processing time; the preemption is not allowed.
5. The starting time and the completion time of an operation O_{jr} is denoted as s_{jr} and c_{jr} respectively. A schedule is a set of completion times for each operation $\left\{c_{jr}\right\}_{1 \le j \le n, 1 \le r \le m}$ that satisfies above constraints.
6. The time required to complete all the jobs is called the *makespan*, which is denoted as $C_{\max}$. By definition, $C_{\max} = \max_{1 \le j \le n, 1 \le r \le m} c_{jr}$.

The problem is "general", in the sense that the technological sequence of machines can be different for each job as implied in the first condition and that the order of jobs to be processed on a machine can be also different for each machine. The predefined technological sequence of each job can be given collectively as a matrix $\left\{T_{jk}\right\}$ in which $T_{jk} = r$ corresponds to the k-th operation O_{jr} of job J_i on machine M_r. The objective of optimizing the problem is to find a schedule that minimizes $C_{\max}$ [49].

11.1 Experimentation

The experiment is conducted on the Taillard benchmark Jobshop scheduling instances [45]. A total of 80 problem instances are available, ranging from 15 job - 15 machine to 100 job - 20 machine problems. The results is presented as the increment on the lower bound provided by Taillard [45]. The equation is given as

$$\Delta = \frac{(H-U)}{U} \tag{11.1}$$

where H is the obtained value and U is the lower bound provided by [45].

The simulations were done in two parts; the first with DE and the second with PSOMA. As with all experiments, two phases of experiment was done with each algorithm, the first with permutative version and the second with the clustered version. The results are given in the subsequent sections.

The control parameters of PSOMA and DE are presented in Table 11.1 and Table 11.2.

Table 11.1: P-SOMA operating parameters

Parameter	Range
MinJ	Dynamic
MaxJ	(0.2 - 0.5) x Problem size
Version	All-to-One

Table 11.2: DE operating parameters

Parameter	Value
F	0.7
CR	0.1

All parameters in Table 10.2 and Table 10.3 were obtained numerically.

11.1.1 Differential Evolution Algorithm

The results obtained for the JSS Taillard instances is given in Tables 11.3 - 11.10. The instances are grouped in respect to their sizes. The bolded values are the better performing heuristic. In addition, the average and standard deviation values are also provided for each problem size.

As with all other experimentation, the reinforced clustered approach of DE_{clust} is the better performing heuristic. However, the canonical approach of DE manages to find similar values in a number of instances.

Table 11.3: 15 job 15 machine $Jm \mid\mid C_{\max}$

Instance	Optimal	DE	DE_{clust}
Tai01	1231	0.451	**0.408**
Tai02	1244	**0.392**	**0.392**
Tai03	1218	0.492	**0.404**
Tai04	1175	0.549	**0.470**
Tai05	1224	0.503	**0.376**
Tai06	1238	**0.330**	**0.330**
Tai07	1227	0.424	**0.374**
Tai08	1217	0.451	**0.391**
Tai09	1274	**0.343**	**0.343**
Tai10	1241	0.411	**0.396**

Table 11.4: 20 job 15 machine $Jm \mid\mid C_{\max}$

Instance	Optimal	DE	DE_{clust}
Tai11	1359	0.641	**0.538**
Tai12	1367	0.691	**0.536**
Tai13	1342	0.503	**0.425**
Tai14	1345	0.552	**0.446**
Tai15	1339	0.525	**0.51**
Tai16	1360	0.525	**0.513**
Tai17	1462	0.501	**0.408**
Tai18	1396	**0.437**	**0.437**
Tai19	1335	**0.431**	**0.431**
Tai20	1348	0.567	**0.526**

Table 11.5: 20 job 20 machine $Jm \mid\mid C_{\max}$

Instance	Optimal	DE	DE_{clust}
Tai21	1644	0.470	**0.427**
Tai22	1600	0.635	**0.443**
Tai23	1557	0.597	**0.489**
Tai24	1646	0.530	**0.495**
Tai25	1595	0.457	**0.396**
Tai26	1645	0.524	**0.424**
Tai27	1680	0.535	**0.457**
Tai28	1603	0.577	**0.487**
Tai29	1625	**0.530**	**0.530**
Tai30	1584	**0.506**	**0.506**

Table 11.6: 30 job 15 machine $Jm \mid\mid C_{\max}$

Instance	Optimal	DE	DE_{clust}
Tai31	1764	**0.434**	**0.434**
Tai32	1795	**0.555**	**0.535**
Tai33	1791	0.572	**0.549**
Tai34	1829	0.473	**0.428**
Tai35	2007	0.425	**0.414**
Tai36	1819	0.597	**0.435**
Tai37	1771	0.570	**0.556**
Tai38	1673	0.605	**0.571**
Tai39	1795	0.484	**0.469**
Tai40	1674	0.676	**0.502**

Table 11.7: 30 job 20 machine $Jm \mid\mid C_{\max}$

Instance	Optimal	DE	DE_{clust}
Tai41	2018	0.656	**0.601**
Tai42	1949	0.698	**0.573**
Tai43	1858	**0.606**	**0.606**
Tai44	1983	**0.560**	**0.560**
Tai45	2000	0.604	**0.554**
Tai46	2015	0.789	**0.563**
Tai47	1903	0.710	**0.609**
Tai48	1949	0.690	**0.589**
Tai49	1967	0.593	**0.560**
Tai50	1926	0.682	**0.645**

Table 11.8: 50 job 15 machine $Jm \mid\mid C_{\max}$

Instance	Optimal	DE	DE_{clust}
Tai51	2760	**0.419**	**0.419**
Tai52	2756	0.458	**0.394**
Tai53	2717	0.385	**0.364**
Tai54	2839	0.357	**0.308**
Tai55	2679	0.478	**0.458**
Tai56	2781	0.436	**0.372**
Tai57	2943	0.397	**0.310**
Tai58	2885	0.431	**0.383**
Tai59	2655	0.458	**0.437**
Tai60	2723	**0.349**	**0.349**

Table 11.9: 50 job 20 machine $Jm \parallel C_{\max}$

Instance	Optimal	DE	DE_{clust}
Tai61	2868	**0.484**	**0.484**
Tai62	2869	0.719	**0.537**
Tai63	2755	0.704	**0.590**
Tai64	2702	0.608	**0.559**
Tai65	2725	0.676	**0.558**
Tai66	2845	**0.571**	**0.571**
Tai67	2825	0.605	**0.530**
Tai68	2784	0.519	**0.496**
Tai69	3071	0.501	**0.438**
Tai70	2995	0.512	**0.445**

Table 11.10: 100 job 20 machine $Jm \parallel C_{\max}$

Instance	Optimal	DE	DE_{clust}
Tai71	5464	0.571	**0.567**
Tai72	5181	0.568	**0.514**
Tai73	5568	**0.586**	**0.586**
Tai74	5339	0.574	**0.551**
Tai75	5392	0.610	**0.602**
Tai76	5342	0.624	**0.598**
Tai77	5436	0.637	**0.615**
Tai78	5394	0.641	**0.632**
Tai79	5358	0.633	**0.620**
Tai80	5183	0.681	**0.667**

11.1.2 Permutative Self Organising Migrating Algorithm

PSOMA was also subjected to the same problem instances as DE. The results are identically grouped in Tables 11.11 - 11.18, according to the problem sizes.

Table 11.11: 15 job 15 machine $Jm \parallel C_{\max}$

Instance	Optimal	PSOMA	$PSOMA_{clust}$
Tai01	1231	0.597	**0.542**
Tai02	1244	0.666	**0.549**
Tai03	1218	**0.496**	**0.496**
Tai04	1175	0.714	**0.660**
Tai05	1224	0.615	**0.602**
Tai06	1238	0.584	**0.560**
Tai07	1227	**0.475**	**0.475**
Tai08	1217	0.675	**0.591**
Tai09	1274	0.658	**0.585**
Tai10	1241	0.560	**0.480**

Table 11.12: 20 job 15 machine $Jm \parallel C_{\max}$

Instance	Optimal	PSOMA	$PSOMA_{clust}$
Tai11	1359	0.598	**0.538**
Tai12	1367	**0.536**	**0.536**
Tai13	1342	**0.425**	**0.425**
Tai14	1345	0.500	**0.446**
Tai15	1339	**0.510**	**0.510**
Tai16	1360	0.608	**0.513**
Tai17	1462	0.479	**0.408**
Tai18	1396	**0.488**	**0.488**
Tai19	1335	0.495	**0.459**
Tai20	1348	**0.526**	**0.526**

Table 11.13: 20 job 20 machine $Jm \parallel C_{\max}$

Instance	Optimal	PSOMA	$PSOMA_{clust}$
Tai21	1644	**0.781**	**0.718**
Tai22	1600	**0.750**	**0.750**
Tai23	1557	0.800	**0.768**
Tai24	1646	0.730	**0.673**
Tai25	1595	0.763	**0.638**
Tai26	1645	0.771	**0.740**
Tai27	1680	0.863	**0.764**
Tai28	1603	0.728	**0.719**
Tai29	1625	0.833	**0.720**
Tai30	1584	**0.777**	**0.777**

Table 11.14: 30 job 15 machine $Jm \parallel C_{\max}$

Instance	Optimal	PSOMA	$PSOMA_{clust}$
Tai31	1764	0.830	**0.689**
Tai32	1795	**0.810**	**0.810**
Tai33	1791	0.860	**0.733**
Tai34	1829	0.724	**0.715**
Tai35	2007	0.607	**0.569**
Tai36	1819	0.794	**0.772**
Tai37	1771	0.884	**0.817**
Tai38	1673	0.866	**0.755**
Tai39	1795	**0.711**	**0.711**
Tai40	1674	0.871	**0.836**

Table 11.15: 30 job 20 machine $Jm \,||\, C_{\max}$

Instance	Optimal	PSOMA	$PSOMA_{clust}$
Tai41	2018	0.910	**0.877**
Tai42	1949	**0.877**	**0.877**
Tai43	1858	0.959	**0.948**
Tai44	1983	0.935	**0.891**
Tai45	2000	0.852	**0.795**
Tai46	2015	0.906	**0.849**
Tai47	1903	**0.849**	**0.849**
Tai48	1949	0.850	**0.813**
Tai49	1967	0.891	**0.828**
Tai50	1926	0.995	**0.886**

Table 11.16: 50 job 15 machine $Jm \,||\, C_{\max}$

Instance	Optimal	PSOMA	$PSOMA_{clust}$
Tai51	2760	0.716	**0.702**
Tai52	2756	**0.643**	**0.643**
Tai53	2717	0.661	**0.621**
Tai54	2839	**0.535**	**0.535**
Tai55	2679	**0.690**	**0.690**
Tai56	2781	**0.675**	**0.675**
Tai57	2943	0.586	**0.574**
Tai58	2885	**0.658**	**0.658**
Tai59	2655	0.732	**0.717**
Tai60	2723	**0.614**	**0.614**

Table 11.17: 50 job 20 machine $Jm \,||\, C_{\max}$

Instance	Optimal	PSOMA	$PSOMA_{clust}$
Tai61	2868	**0.801**	**0.801**
Tai62	2869	**0.808**	**0.808**
Tai63	2755	0.849	**0.840**
Tai64	2702	0.847	**0.830**
Tai65	2725	0.896	**0.875**
Tai66	2845	0.813	**0.794**
Tai67	2825	0.900	**0.856**
Tai68	2784	0.880	**0.806**
Tai69	3071	0.789	**0.676**
Tai70	2995	0.824	**0.795**

Table 11.18: 100 job 20 machine $Jm \,||\, C_{\max}$

Instance	Optimal	PSOMA	$PSOMA_{clust}$
Tai71	5464	0.721	**0.679**
Tai72	5181	0.716	**0.682**
Tai73	5568	0.714	**0.644**
Tai74	5339	0.695	**0.597**
Tai75	5392	0.668	**0.650**
Tai76	5342	0.675	**0.627**
Tai77	5436	0.632	**0.623**
Tai78	5394	0.697	**0.663**
Tai79	5358	0.675	**0.675**
Tai80	5183	0.625	**0.618**

The clustered approach of $PSOMA_{clust}$ is the better performing heuristic. It managaes to find the better value for every instance, however, on some occasions it is unable to improve on the result of the canonical version of PSOMA.

11.2 Analysis

Comparison was done between the clustered approach of DE_{clust} and $PSOMA_{clust}$. The results are given in Tables 11.19 - 11.26 and the comparison result is given in Table 11.27. DE_{clust} is by far the better performing heuristic of the two, managing to find better values for all problem classes. It also manages to find better average and deviation values for the instances.

Table 11.19: 15 job 15 machine $Jm \parallel C_{\max}$

Instance	Optimal	DE_{clust}	$PSOMA_{clust}$
Tai01	1231	**0.408**	0.542
Tai02	1244	**0.392**	0.549
Tai03	1218	**0.404**	0.496
Tai04	1175	**0.470**	0.660
Tai05	1224	**0.376**	0.602
Tai06	1238	**0.330**	0.560
Tai07	1227	**0.374**	0.475
Tai08	1217	**0.391**	0.591
Tai09	1274	**0.343**	0.585
Tai10	1241	**0.396**	0.480
Average	1228.9	**0.388**	0.554
Std Dev	25.141	**0.038**	0.058

Table 11.20: 20 job 15 machine $Jm \parallel C_{\max}$

Instance	Optimal	DE_{clust}	$PSOMA_{clust}$
Tai11	1359	**0.538**	**0.538**
Tai12	1367	**0.536**	**0.536**
Tai13	1342	**0.425**	**0.425**
Tai14	1345	**0.446**	**0.446**
Tai15	1339	**0.510**	**0.510**
Tai16	1360	**0.513**	**0.513**
Tai17	1462	**0.408**	**0.408**
Tai18	1396	**0.437**	0.488
Tai19	1335	**0.431**	0.459
Tai20	1348	**0.526**	**0.526**
Average	1365.3	**0.477**	0.485
Std Dev	38.337	0.051	**0.047**

Table 11.21: 20 job 20 machine $Jm \parallel C_{\max}$

Instance	Optimal	DE_{clust}	$PSOMA_{clust}$
Tai21	1644	**0.427**	0.718
Tai22	1600	**0.443**	0.750
Tai23	1557	**0.489**	0.768
Tai24	1646	**0.495**	0.673
Tai25	1595	**0.396**	0.638
Tai26	1645	**0.424**	0.740
Tai27	1680	**0.457**	0.764
Tai28	1603	**0.487**	0.719
Tai29	1625	**0.530**	0.720
Tai30	1584	**0.506**	0.777
Average	1617.9	**0.466**	0.727
Std Dev	36.570	**0.042**	0.043

Table 11.22: 30 job 15 machine $Jm \parallel C_{\max}$

Instance	Optimal	DE_{clust}	$PSOMA_{clust}$
Tai31	1764	**0.434**	0.689
Tai32	1795	**0.535**	0.810
Tai33	1791	**0.549**	0.733
Tai34	1829	**0.428**	0.715
Tai35	2007	**0.414**	0.569
Tai36	1819	**0.435**	0.772
Tai37	1771	**0.556**	0.817
Tai38	1673	**0.571**	0.755
Tai39	1795	**0.469**	0.711
Tai40	1674	**0.502**	0.836
Average	1791.8	**0.489**	0.741
Std Dev	92.886	**0.060**	0.078

Table 11.23: 30 job 20 machine $Jm \mid\mid C_{\max}$

Instance	Optimal	DE_{clust}	$PSOMA_{clust}$
Tai41	2018	**0.601**	0.877
Tai42	1949	**0.573**	0.877
Tai43	1858	**0.606**	0.948
Tai44	1983	**0.560**	0.891
Tai45	2000	**0.554**	0.795
Tai46	2015	**0.563**	0.849
Tai47	1903	**0.609**	0.849
Tai48	1949	**0.589**	0.813
Tai49	1967	**0.560**	0.828
Tai50	1926	**0.645**	0.886
Average	1956.8	**0.586**	0.861
Std Dev	51.115	**0.029**	0.044

Table 11.24: 50 job 15 machine $Jm \mid\mid C_{\max}$

Instance	Optimal	DE_{clust}	$PSOMA_{clust}$
Tai51	2760	**0.419**	0.702
Tai52	2756	**0.394**	0.643
Tai53	2717	**0.364**	0.621
Tai54	2839	**0.308**	0.535
Tai55	2679	**0.458**	0.690
Tai56	2781	**0.372**	0.675
Tai57	2943	**0.310**	0.574
Tai58	2885	**0.383**	0.658
Tai59	2655	**0.437**	0.717
Tai60	2723	**0.349**	0.614
Average	2773.8	**0.379**	0.643
Std Dev	91.111	**0.049**	0.058

Table 11.25: 50 job 20 machine $Jm \mid\mid C_{\max}$

Instance	Optimal	DE_{clust}	$PSOMA_{clust}$
Tai61	2868	**0.484**	0.801
Tai62	2869	**0.537**	0.808
Tai63	2755	**0.590**	0.840
Tai64	2702	**0.559**	0.830
Tai65	2725	**0.558**	0.875
Tai66	2845	**0.571**	0.794
Tai67	2825	**0.530**	0.856
Tai68	2784	**0.496**	0.806
Tai69	3071	**0.438**	0.676
Tai70	2995	**0.445**	0.795
Average	2843.9	**0.521**	0.808
Std Dev	116.303	**0.052**	0.054

Table 11.26: 100 job 20 machine $Jm \mid\mid C_{\max}$

Instance	Optimal	DE_{clust}	$PSOMA_{clust}$
Tai71	5464	**0.576**	0.679
Tai72	5181	**0.514**	0.682
Tai73	5568	**0.586**	0.644
Tai74	5339	**0.551**	0.597
Tai75	5392	**0.602**	0.650
Tai76	5342	**0.598**	0.627
Tai77	5436	**0.615**	0.623
Tai78	5394	**0.632**	0.663
Tai79	5358	**0.620**	0.675
Tai80	5183	0.667	**0.618**
Average	5365.7	**0.594**	0.646
Std Dev	118.251	**0.043**	0.229

Table 11.27: DE_{clust} and $PSOMA_{clust}$ summerised results for $Jm \mid\mid C_{\max}$

Instance		**Δavg**		**Δstd**	
job	*mach*	DE_{clust}	$PSOMA_{clust}$	DE_{clust}	$PSOMA_{clust}$
15	15	**0.388**	0.554	**0.038**	0.058
20	15	**0.477**	0.485	0.051	**0.047**
20	20	**0.466**	0.727	**0.042**	0.043
30	15	**0.489**	0.741	**0.060**	0.078
30	20	**0.586**	0.861	**0.029**	0.044
50	15	**0.379**	0.643	**0.049**	0.058
50	20	**0.521**	0.808	**0.052**	0.054
100	20	**0.594**	0.646	**0.043**	0.229

Chapter 12

Analysis and Conclusions

12.1 Population Dynamics

In terms of population dynamics, two unique population representations are given. The first set of results are for the QAP problem of "Bur26a", which is first solved by DE_{clust} and the by $PSOMA_{clust}$.

For each set, four graphs are presented, the first two are the initial and final population in "*deviation*" space. The third is the "*Edge*" representation and the final is the "*best solution*" in the population.

The initial popualtion for the DE_{clust} is given in Figure 12.1.

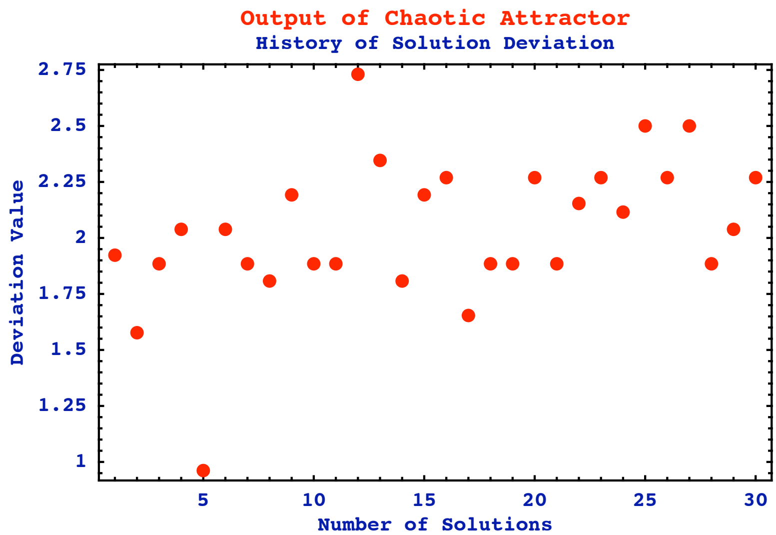

Figure 12.1: Initial Population Clustering for DE_{clust}

The final population clustering is given in Figure 12.2.

The deviation of the solutions is from 1 - 2.75 in the initial population and 5 - 10 in the final population. This shows a *drift* of the solutions in the deviation space. Another point of interest is that the solutions are still diversified in their structure. The solutions within the clusters have converged, however the overall diversity is maintained

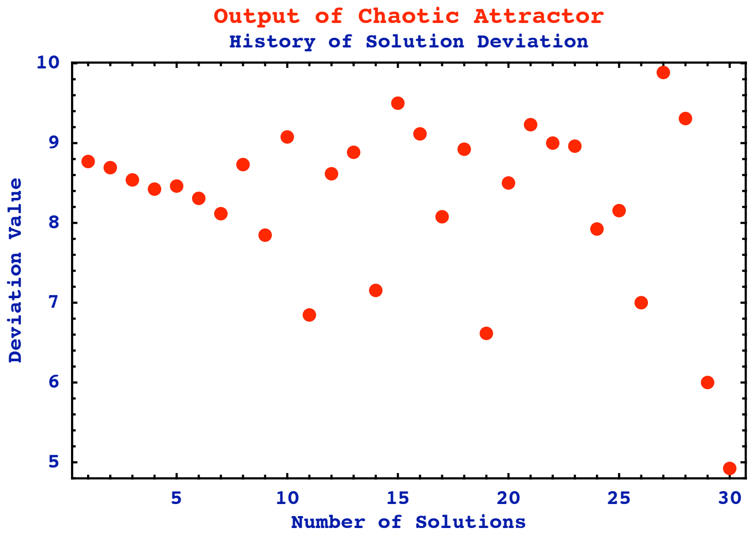

Figure 12.2: Final Population Clustering for DE_{clust}

within the population. This opens more oppertunity to obtain better solutions in next generations.

The *Edge* C_E of the population throughout the population generation (in this case, 200 generations) for DE_{clust} is given in Figure 12.3.

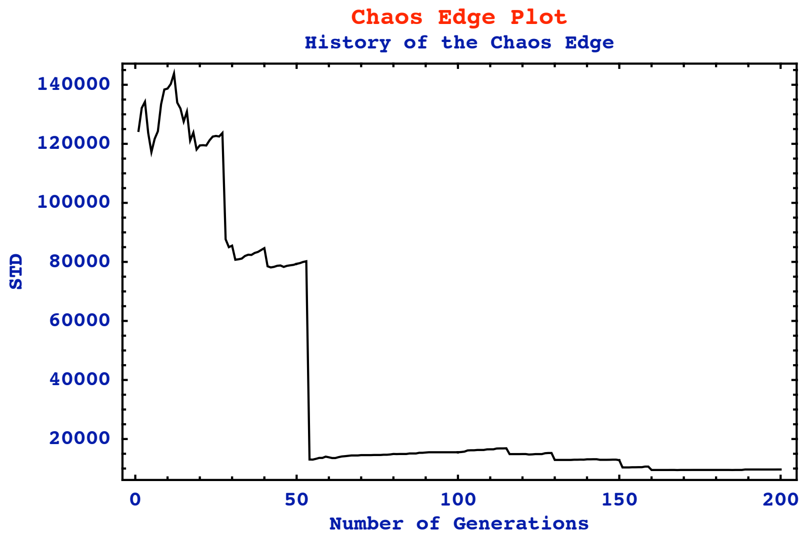

Figure 12.3: Edge for DE_{clust}

A general decline of the spread of the clusters and fitness values is seen. This is typical for a minimising function.

The final graph of the best individual is seen in Figure 12.4.

A direct correlation is seen between the graphs of Edge and Best Individual. The **Edge** is a prelude to a shift in solution space. A shift generally signifies a region of

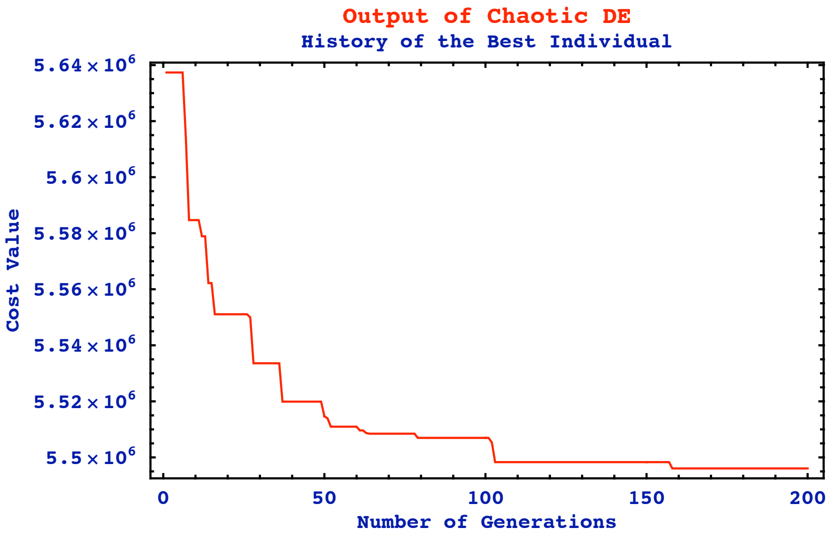

Figure 12.4: Best Individual for DE_{clust}

new solutions, and possibiltiy of further improvement.

A second set of results for the QAP for $PSOMA_{clust}$ is given in Figures 12.5 to 12.8. Figures 12.5 and 12.6 give the initial and final solution represenataion in terms of deviation. As seen for the DE_{clust}, the solution remains diversified for the entirity of the generation. The solution also drift in the deviation space from 2.75 to 8, signifying exploration of the solution space.

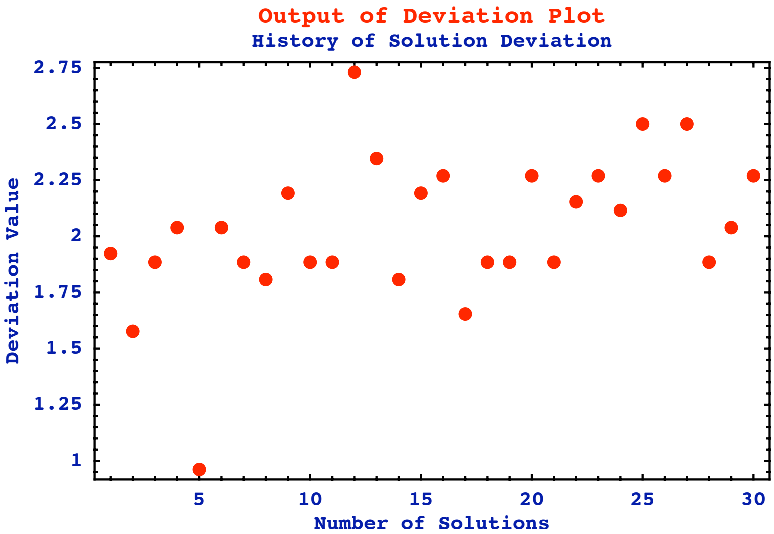

Figure 12.5: Initial Population Clustering for $PSOMA_{clust}$

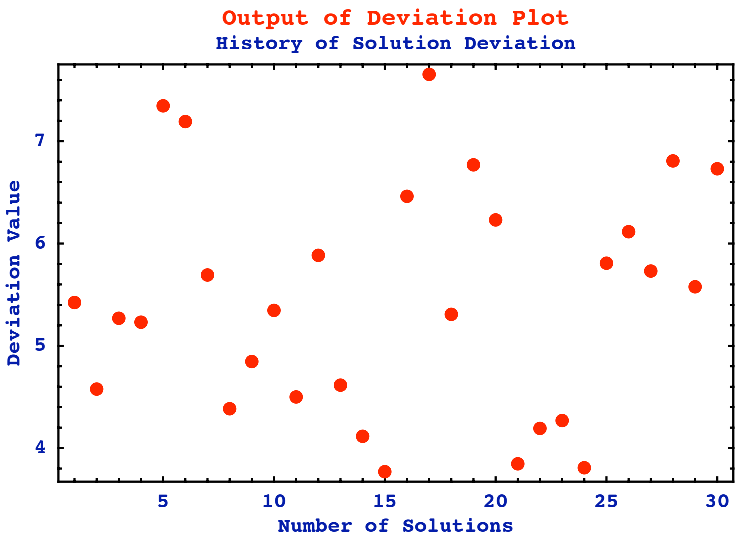

Figure 12.6: Final Population Clustering for $PSOMA_{clust}$

The "Edge" and "Best Individual" graphs are given in Figures 12.7 and 12.8. As with the DE_{clust}, a correlation is seen with the Edge and Best Individual. The Edge is seen as a prelude to the exploration space. The measure of the population provides an indicator as to the shift in the best solution in the population.

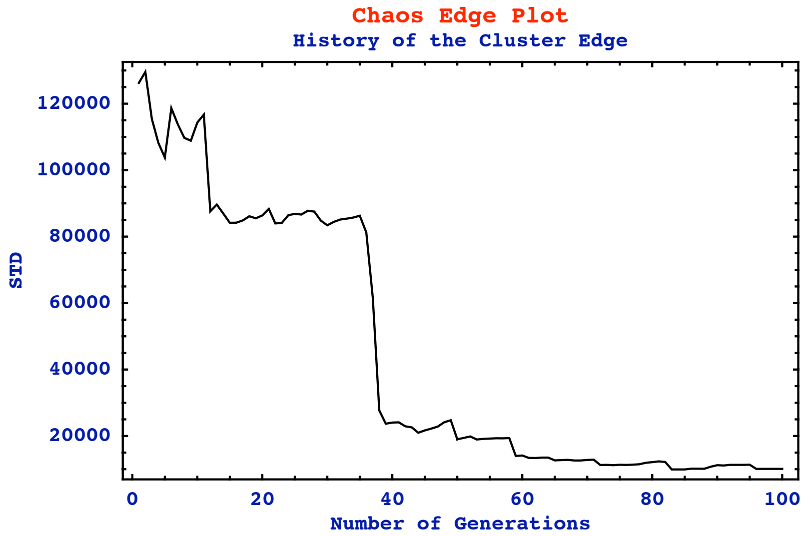

Figure 12.7: Edge for $PSOMA_{clust}$

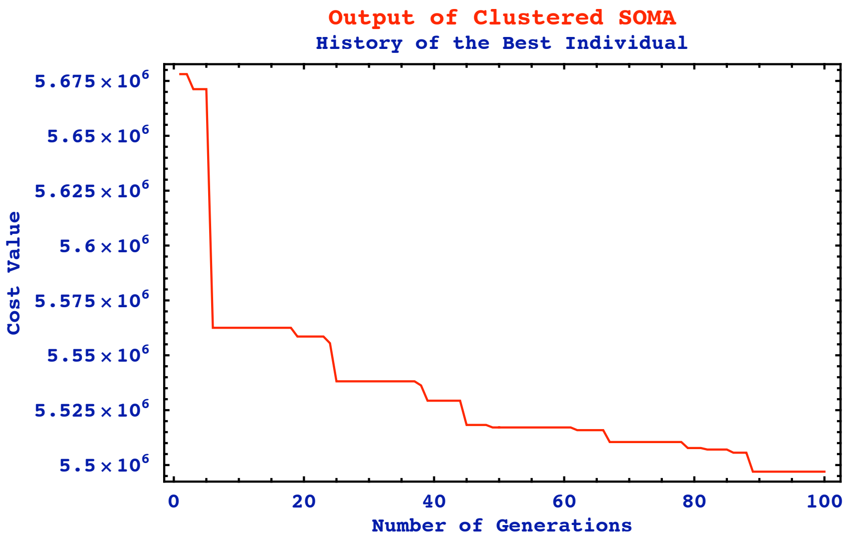

Figure 12.8: Best Individual for $PSOMA_{clust}$

The final set of population dynamics is given for the flowshop scheduling problem of Tai01 in Figures 12.9 to 12.12. The applied heuristic is DE_{clust}. This provides a comparison with another problem class from QAP.

The initial and final solution representation is given in Figures 12.9 and 12.10. As with the representation for QAP, a shift in the deviation space is seen for the population.

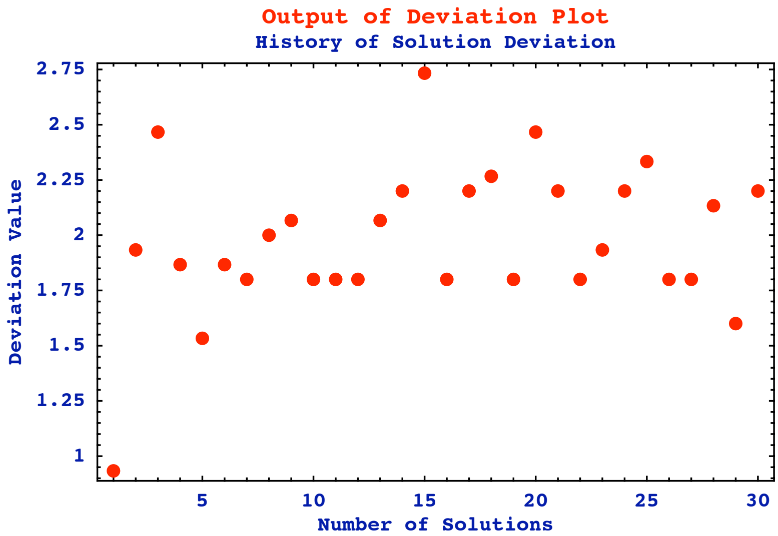

Figure 12.9: Initial Population Clustering for FSS

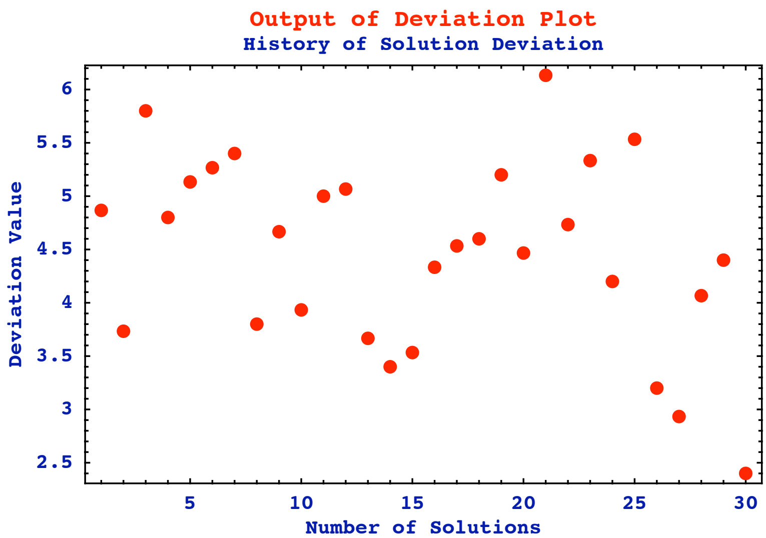

Figure 12.10: Final Population Clustering for FSS

The Edge and Best Individual graphs are also correlated. The Edge graph in Figure 12.11 is a representation of a more hapharzard system. The increase in value is an indication of "stagnation" of the system, where new selection criteria are envoked in order to bypass local optima region. Another indicatior is that even though the Best Individual in Figure 12.12 levels off at generation 45, the Edge graph shows active search indications right up to generation 85.

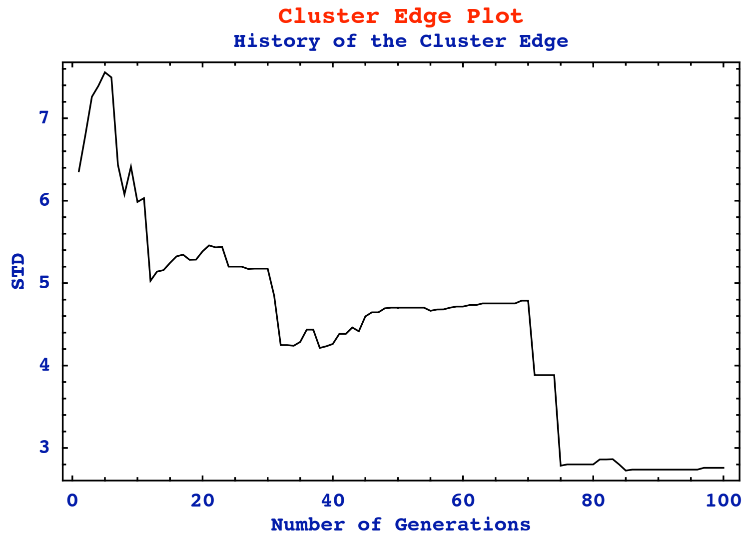

Figure 12.11: Edge for FSS

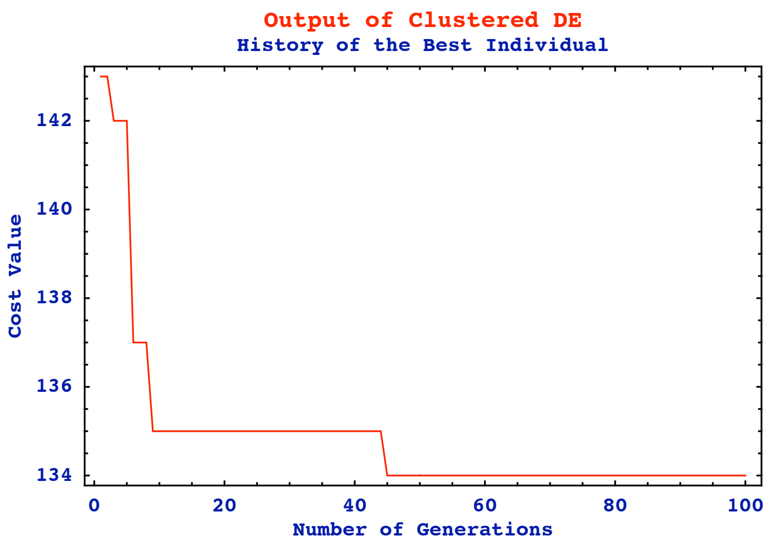

Figure 12.12: Best Individual for FSS

12.2 Conclusion

From the obtained results, it is evident that clustering of the population improves the performance of the applied heuristics. An effort has been made during this research to have a generic form of the clustering, which in effect can be applied to any canonical heuristic.

Clustering can be seen as a tool for the diversification of the solution, and not for propagation. It in effect ensures that unique indicatiors are utilized in order to facilitate the non-convergence of the population. During the initial experimentation it was observed that simple arithematic operators such as "deviation" and "spread" performed exceptionally for the permutative based combinatorial problems.

Clustering inexorably includes new *selection* and *deletion* criteria, which aid and abeit the drift of the clusters in the deviation space.

In order to validate the clustering approach, two unique paradigm heuristical approaches of DE and SOMA have been utilised. For the *premutative flowshop* and *quadratic assignment problem*, Genetic Algorithm (GA) has also been used to provide completeness of the heuristics. DE is a "vector" oriented approach, whereas SOMA is based on "swamp" paradigm.

A total of six unique permutative based combinatorial optimization problem classes have been solved using the clustered approaches of SOMA and DE. In order to have consistancy, the **Taillard benchmark problem sets** have been selected for all these problems, alongside in some cases other problem classes. The Taillard sets are mathematically structured which reflect both problems with good varience and those which reflect practical problem settings. These problems range from small to large in size and difficulty [44].

Permutative flowshop is the generic version of flowshop, which has been solved for a number of years. In this problem class, $PSOMA_{clust}$ performs exceptionally well compared with the optimal solutions and other published heuristics.

The second version of flowshop, **flowshop with limited intermediate storage** or *flowshop wih blocking* is a advanced version of flowshop which reflects a more practical shop floor setting. DE_{clust} is a better performing heuristic for this problem class.

The most current and technologically advanced version of flowshop is **flowshop with no-wait**, where jobs do not wait between machines. This problem class is the most challenging to solve, and has the most practical application in today's manufacturing systems. DE_{clust} is the best performing heuristic for this problem class.

The fourth problem is that of **quadratic assignment problem**. Two unique instances have been solved; regular and irregular. The QAP problem is reflected in the "distance" and "flow" matrix approach with a number of practical applications. $PSOMA_{clust}$ is the better performing heuristic in this problem class, compared with the optimal values and other published heuristics.

The fifth problem is the **capacitated vechicle routing problem**. The CVRP problem is the combined problem of Traveling Salesman (TSP) and Bin Packing Problem (BPP). The difficulty rating of this problem is twice that of TSP, with a more practical setting. DE_{clust} and $PSOMA_{clust}$ are equally impressive for this problem class, with DE_{clust} performing better for the larger sized instances.

The final problem is the **job shop scheduling** problem. JSS is one of the most challenging scheduling problem in manufacturing systems. DE_{clust} is the better performing heuristic for this problem class.

A total of 429 different problem instances have been used with up to 6 unique heuristics. A minimum of 10 experimentation have been conducted for each instance.

An approximate minimum of 10 million generation cycles have been done with an approximate minimum of 200 million objective function evaluations conducted in order to validate the clustered approach.

During the course of this research, five unique heuristics have been developed and one heuristic expanded. The clustering approach is the main heuristical development of this research. SOMA has been applied for the first time to permutative problem with the development of Permutative Set Handling, Dynamic PSOMA and Static PSOMA. Discrete Set Handling has been expanded to include permutative problems.

One of the most impressive feats of this research has been the relative exclusion of "local search" heuristics from the evaluation of the heuristics. DE only incorportaes a 2 opt local search when stagnation is detected, which is very minimal, whereas PSOMA does not incorporate any local search heuristics. This provides a novality to this approach since local search routines have become a hallmark for permutative heuristics in recent years, to an extent that the true effectiveness of the underlying metaheuristics are almost impossible to judge.

The results obtained through the extensive evaluation of the different problem classes validate the clustered approach, and the developed permutative and clustered versions of DE and SOMA.

12.3 Acknowledgement

A research of this scope and complexity could not have been completed without good research funding and resources. The following two grants are acknowledged for the financial support for this research.

1. Grant Agency of the Czech Republic **GX391680170/2602**

2. Grant of the Czech Ministry of Education **MSM 7088352102**

Bibliography

[1] R. Ahuja, J. Orlin, and A. Tiwari. A descent genetic algorithm for the quadratic assignment problem. *Computers and Operations Research*, 27:917–934, 2000.

[2] K. Aihara, T. Takabe, and M. Toyoda. Chaotic neural networks. *Phys.Lett. A*, 6:333–340, 1990.

[3] J. Beasley. Operations reserach library, 2009.

[4] A. Boelte and U. Thonemann. Optimizing simulated annealing schedules with genetic programming. *European Journal of Operations Research*, 92:402–416, 1996.

[5] J. Carlier. Ordonnancements a contraintes disjonctives. *Operations Research*, 12:333–351, 1978.

[6] L. Chen and A. Kazuyuki. Chaotic simulated annealing by a neural network model with transient chaos. *Neural Networks*, 8(6):915–930, 1995.

[7] G. Dantzig and R. Ramser. The truck dispatching problem. *Management Science*, 6:80–91, 1959.

[8] D. Davendra. Differential evolution algorithm for flow shop scheduling. Master's thesis, University of the South Pacific, 2001.

[9] D. Davendra and G. Onwubolu. Enhanced differential evolution hybrid scatter search for discrete optimisation. In *Proc. of the IEEE Congress on Evolutionary Computation*, pages 1156–1162, Singapore, Sept 2007.

[10] D. Davendra and G. Onwubolu. Flow shop scheduling using enhanced differential evolution. In *Proc.21 European Conference on Modeling and Simulation*, pages 259–264, Prague, Czech Rep, Jun 2007.

[11] D. Davendra and G. Onwubolu. Forward backward transformation. In G. Onwubolu and D. Davendra, editors, *Differential Evolution: A Handbook for Global Permutation-Based Combinatorial Optimization*. Springer, Germany, 2009.

[12] D. Davendra and I. Zelinka. Flow shop scheduling using self organising migrating algorithm. In *Proc. 22nd European Conference of Modelling and Simulation*, pages 195–200, Nisosia, Cyprus, June 2008.

[13] D. Davendra and I. Zelinka. Optimization of quadratic assignment problem using self-organinsing migrating algorithm. *Computing and Informatics*, 28:169–180, 2009.

[14] D. Davendra, I. Zelinka, and G. Onwubolu. Chaotic optimization. In *Proc.21 European Conference on Modeling and Simulation*, pages 265–273, Prague, Czech Republic, June 2007.

[15] D. Davendra, I. Zelinka, and G. Onwubolu. Clustered population differential evolution for quadratic assignment problem. In *Proc. 11th IEEE Congress on Evolutionary Computation*, Trondheim, Norway, May 2008.

[16] D. Davendra, I. Zelinka, and R. Senkerik. Clustered self organising migrating algorithm for the quadratic assignment problem. In *Proc. 2nd Global Conference on Power and Optimization*, Bali, Indonesia, June 2009.

[17] Z. Drezne. A new genetic algorithm for the quadratic assignment problem. *INFORMS Journal on Computing*, 115:320–330, 2003.

[18] L. Gambardella, E. Thaillard, and M. Dorigo. Ant colonies for the quadratic assignment problem. *International Journal of Operations Research*, 50:167–176, 1999.

[19] M. Garey and D. Johnson. *Computers and intractability: A guide to the theory of NP-completeness*. Freeman, San Francisco, 1979.

[20] J. Gleick. *Chaos: Making a New Science*. Vintage, USA, 1987.

[21] J. Grabowski and J. Pempera. Sequencing of jobs in some production system. *European Journal of Operational Research*, pages 535–550, 2000.

[22] N. Hall and C. Sriskandarayah. A survey of machine scheduling problems with blocking and no-wait in process. *Operations Research*, pages 510–525, 1996.

[23] J. Heller. Some numerical experiments for an mj flow shop and its decision-theoretical aspects. *Operations Research*, 8:178–184, 1960.

[24] D. Hochbam. *Approximation Algorithms for NP - Hard Problems*. PWS Publishing Company. USA, 1997.

[25] J. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[26] K. Hu, J. Li, J. Liu, and L. Jiao. Permutation flow-shop scheduling based on multiagent evolutionary algorithm. In A. Sattar and B.H. Kang, editors, *AI 2006: Advances in Artificial Intelligence*. Springer-Verlag, Berlin, Germany, 2006.

[27] T. Ikeguchi and Y. Horio. Chaos for avoiding local minima a. mutual connection neural network dynamics. Technical report, Tokyo University of Science, 1999.

[28] S. Ishi and M. Sato. Chaotic potts spin model for combinatorial optimization problems. *Neural Networks*, 10:941–963, 1997.

[29] P. Ji, W. Yongzhong, and L. Haozhao. A solution method for the quadratic assignment problem (qap). In *Proc.6 International Symposium on Operations Research and Its Applications*, pages 106–117, Xinjiang, China, August 2006.

[30] R. May. *Stability and Complexity in Model Ecosystems*. Princeton University Press. Princeton, USA, 2001.

[31] A. Misevicius. An improved hybrid optimization algorithm for the quadratic assignment problem. *Mathematical Modelling and Analysis*, 9:149–168, 2004.

[32] H. Nozawa. Chaos 2. *Physics D*, 2:377, 1992.

[33] G. Onwubolu. *Emerging Optimization Techniques in Production Planning and Control*. Imperial Collage Press, London, England, 2002.

[34] G. Onwubolu and D. Davendra. Scheduling flow shops using differential evolution algorithm. *European Journal of Operations Research*, 171:674–679, 2006.

[35] G. Onwubolu and D. Davendra. *Differential Evolution: A Handbook for Global Permutation-Based Combinatorial Optimization*. Springer, Germany, 2009.

[36] M. Pinedo. *Scheduling: theory, algorithms and systems*. Prentice Hall, Inc., New Jersey, 1995.

[37] S. Ponnambalam, P. Aravindan, and S. Chandrasekhar. Constructive and improvement flow shop scheduling heuristic: an extensive evaluation. *Production Planning and Control*, 12:335–344, 2001.

[38] K. Price. An introduction to differential evolution. In D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimisation*. McGraw Hill, International, UK, 1999.

[39] W. Raaymakers and J. Hoogeveen. Scheduling multipurpose batch process industries with no-wait restrictions by simulated annealing. *European Journal of Operational Research*, pages 131–151, 2000.

[40] C. Rajendran. A no-wait flowshop scheduling heuristic to minimize makespan. *Journal of the Operational Research Society*, pages 472–478, 1994.

[41] C. Reeves. A genetic algorithm for flowshop sequencing. *Computers and Operations Research*, 22:5–13, 1995.

[42] C. Reeves and T.Yamada. Genetic algorithms, path relinking and flowshop sequencing problem. *Evolutionary Computation*, 6:45–60, 1998.

[43] E. Taillard. Robust taboo search for the quadratic assignment problem. *Parallel Computing*, 17:443–455, 1991.

[44] E. Taillard. Benchmarks for basic scheduling problems. *European Journal of Operations Research*, 64:278–285, 1993.

[45] E. Taillard. Job shop lower bounds, 2009.

[46] M. Tasgetiren, Y-C. Liang, M. Sevkli, and G. Gencyilmaz. Particle swamp optimization algorithm for permutative flowshops sequencing problems. In *Differential Evolution Algorithm for Permutative Flowshops Sequencing Problem with Makespan Criterion*, pages 442–452, Sakaraya, Turkey, sept 2004.

[47] M. Tasgetiren, M. Sevkli, Y-C. Liang, and G. Gencyilmaz. Particle swamp optimization algorithm for permutative flowshops sequencing problems. In *4th International Workshops on Ant Algorithms and Swamp Intelligence*, pages 389–390, Brussel, Belgium, sept 2004.

[48] L. Tseng and T. Lin. A hybrid genetic algorithm for the flow-shop scheduling problem. *Lecture Notes in Computer Science*, pages 218–227, 2006.

[49] T. Yamada. *Studies on Metaheuristics for Jobshop and Flowshop Scheduling Problems*. PhD thesis, Kyoto University, 2003.

[50] T. Yamada and K. Aihara. Nonlinear neurodynamics and combinatorial optimization in chaotic neural networks. *Journal of Intelligent Fuzzy Systems*, 5:53–68, 1997.

[51] I. Zelinka. Soma self organizing migrating algorithm. In G. Onwubolu and B. Babu, editors, *New Optimization Techniques in Engineering*. Springer-Verlag, Germany, 2004.