Soft Comput (2014) 18:669–681
DOI 10.1007/s00500-014-1219-7

FOCUS

Utilising the chaos-induced discrete self organising migrating algorithm to solve the lot-streaming flowshop scheduling problem with setup time

Donald Davendra · Roman Senkerik · Ivan Zelinka · Michal Pluhacek · Magdalena Bialic-Davendra

Published online: 30 January 2014

Abstract The Dissipative Lozi chaotic map is embedded in the discrete self organising migrating algorithm (DSOMA), as a pseudorandom generator. This novel chaotic based algorithm is applied to the constraint based lot-streaming flowshop scheduling problem. Two new and unique data sets generated using the Lozi and Delayed Logistic maps are used to compare the chaos embedded DSOMA and the generic DSOMA utilising the venerable Mersenne Twister. In total, 100 data sets were tested by these two algorithms, for the idling and the non-idling case. From the obtained results, the chaos variant algorithm is shown to significantly improve the performance of generic DSOMA.

Communicated by I. Zelinka.

D. Davendra (✉) · I. Zelinka
Department of Computing Science,
Faculty of Electrical Engineering and Computer Science,
VSB-Technical University of Ostrava,
17. listopadu 15, 708 33 Ostrava-Poruba, Czech Republic
e-mail: donald.davendra@vsb.cz

I. Zelinka
e-mail: ivan.zelinka@vsb.cz

R. Senkerik · M. Pluhacek
Department of Informatics and Artificial Intelligence,
Faculty of Applied Informatics, Tomas Bata University in Zlin,
Nad Stranemi 4511, 760 05 Zlin, Czech Republic
e-mail: senkerik@fai.utb.cz

M. Pluhacek
e-mail: pluhacek@fai.utb.cz

M. Bialic-Davendra
Centre for Applied Economic Research, Faculty of Management and Economics, Tomas Bata University in Zlin,
nam. T. G. Masaryka 5555, 760 01 Zlin, Czech Republic
e-mail: bialic@fame.utb.cz

Keywords Lot-streaming flowshop scheduling · Lozi map · Delayed Logistic map · Discrete Self Organising Migrating algorithm

1 Introduction

One of the core premise of evolutionary algorithms (EA) is their reliance on *stochasticity*, the ability to generate a random events, which in turn, provides the spark of perturbation towards the desired goal. The task of generating this stochasticity is generally in the realm of *pseudorandom number generators* (*PRNG*); a structured sequence of mathematical formulation which tries to yield a generally optimal range of distributed numbers between a specified range.

A wide variety of such random number generators exist, however, the most common in usage is the *Mersenne Twister* (Matsumoto and Nishimura 1998). A number of its variants have been designed; for a full listing please see Matsumoto (2012). Some other common PRNG are the *Mother Of All*, *CryptoAPI*, *Indirection, Shift, Accumulate, Add, and Count* (ISAAC), *Keep it Simple Stupid* (KISS) and *Mutiply-With-Carry* (MWC).

This paper explores a novel approach to generating PRNG, one with a lineage in chaos theory. The term *chaos* describes the complex behaviour of simple, well behaved systems. When casually observed, this behaviour may seem erratic and somewhat random, however, these systems are deterministic, whose precise knowledge of future behaviour is well known. The question is then to reconcile the notion of nonlinearity of these systems.

Sudden and dramatic changes in some nonlinear systems may give rise to complex behaviour called chaos. The noun *chaos* and the adjective *chaotic* are used to describe the time behaviour of a system when that behaviour is aperiodic (it *never* exactly repeats) and appears apparently random or noisy (Hilborn 2000).

This a periodic non-repeating behaviour of chaotic systems is the core foundation of this research. The objective is then to use a valid chaotic system, and embed it in the EA's as a PRNG. Four general branches of chaotic systems exist, which are the dissipative systems, fractals, dissipative and high-dimensional systems and conservative systems. The systems used for this research are the *discrete* dissipative systems of Lozi and Delayed Logistic maps. These two systems are relatively simple in terms of period density, and therefore, easier to obtain data through sectional cropping.

Many chaotic maps in the literature possess certainty, ergodicity and the stochastic property. Recently, chaotic sequences have been adopted instead of random sequences with improved results. They have been used to improve the performance of EA's (Alatas et al. 2009; Caponetto et al. 2003). They have also been used together with some heuristic optimisation algorithms (Davendra et al. 2010; Zuo and Fan 2006) to express optimisation variables. The choice of chaotic sequences is justified theoretically by their unpredictability, i.e. by their spread-spectrum characteristic, non-periodic, complex temporal behaviour, and ergodic properties (Ozer 2010).

A mathematical description of the connection between chaotic systems and random number generators has been given by Herring and Julian (1989). In this paper, a strong linkage has been shown between the Lehmer generator (Lehmer 1951) and the simple chaos dynamical system of Bernoulli shift (Palmore and McCauley 1987). By observation, it is obvious that the Bernoulli shift itself is similar in form to the Lehmer generator. Therefore, it was postulated that prime modulus multiplicative linear congruential generators are implementations of deterministic chaotic processes (Herring and Julian 1989).

A chaotic piecewise-linear one dimensional (PL1D) map has been utilised as a chaotic random number generator in Stojanovski and Kocarev (2001). The construction of the chaos random number system is based on the exploitation of the double nature of chaos, deterministic in microscopic space and by its defining equations, and random in macroscopic space. This new system is mathematically proven to overcome the major drawbacks of classical random number systems, which are its reliance on the assumed randomness of a physical process, inability to analyse and optimise the random number generator, inability to compute probabilities and entropy of the random number generator, and inconclusiveness of statistical tests.

A family of enhanced chaotic pseudo random number generators (CPRNG) has been developed by Lozi (2008), where the main imputes is the generation of very long series of pseudo-random number generations. This is accomplished through what is called the ultra weak coupling of chaotic systems, such as the Tent Map, which is enhanced in order to conceal the chaotic genuine function (Lozi 2009).

Differential evolution (DE), which is a population based global optimization algorithm over continuous spaces applying stochastic state transition rules, first proposed by Price (1999) has been modified using chaotic sequences. Davendra et al. (2010) has applied the canonical DE to solve the PID optimisation problem, whereas Ozer (2010) applied a sequence of chaotic maps to optimise a range of benchmark problems, with the conclusion that the Sinus map and Circle map have somewhat increased the solution quality and with the ability to escape the local optima. The economic dispatch problem was solved by Lu et al. (2011), where the *Tent Map* was utilised as a chaotic local search in order to bypass the local optima.

Concerning combinatorial optimisation problems, Yuan et al. (2008) has developed a chaotic hybrid DE, where the parameter selection and operation are handled by chaotic sequences, whereas Davendra et al. (2010) utilised both the population generation and parameter selection using a hybrid DE-Scatter Search algorithm to solve the travelling salesman problem.

The motivation of this work is to ascertain if any improvement can be attained in an EA by embedding a chaos map. Since current literature has applied chaos to a magnitude of such research with varying application of the chaos maps (population generation, mutation crossover, escaping the local optima etc.), we decided to totally replace the PRNG with a chaotic map. The Lozi map was chosen as it is a very simple map, which can be totally described in a few equations, making it easier to implement. Additionally, the Lozi map has proven to be an excellent generator in the previous work on DE for real value optimisation (Davendra et al. 2010).

This research looks at modifying the discrete self organising migrating algorithm (DSOMA) (Davendra 2009, 2013; Davendra and Bialic-Davendra 2013), yet another population based global optimisation algorithm over discrete spaces, with the Lozi chaotic map and utilising it to solve the lot-streaming flowshop problem with setup time. The primary aim of this work, therefore, is to ascertain if there is any improvement to the algorithm when comparing the chaotic variant to the standard DSOMA variant, utilising the best PRNG; the Mersenne Twister. Secondly, we have generated two unique data sets using the Lozi and Delayed Logistic systems for testing.

2 Lozi map

The Lozi map is a two-dimensional piecewise linear map whose dynamics are similar to those of the better known

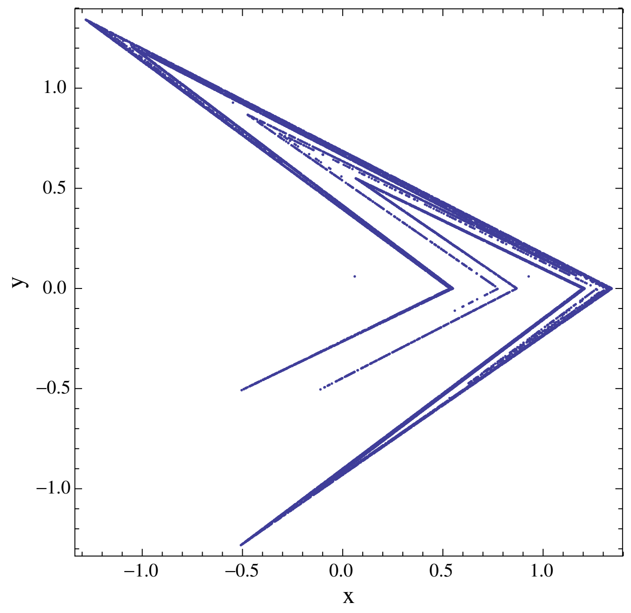

Fig. 1 Lozi map

Henon map (Hennon 1979), and which admits strange attractors.

The advantage of the Lozi map is that one can compute every relevant parameter exactly, due to the linearity of the map, and the successful control can be demonstrated rigorously.

The Lozi map equations are given in (1) and (2).

$$X_{n+1} = 1 - a \cdot |X_n| + b \cdot Y_n \tag{1}$$

$$Y_{n+1} = X_n \tag{2}$$

The parameters used in this work are $a = 1.7$ and $b = 0.5$ as suggested in Caponetto et al. (2003) and the initial conditions are $X_0 = -0.1$ and $Y_0 = 0.1$. The Lozi map is shown in Fig. 1.

3 Delayed logistic map

The Delayed Logistic (DL) map equations are given in (3) and (4).

$$X_{n+1} = A \cdot X_n \cdot (1 - Y_n) \tag{3}$$

$$Y_{n+1} = X_n \tag{4}$$

The parameter used in this work is $a = 2.27$ and the initial conditions are $X_0 = 0.001$ and $Y_0 = 0.001$. The DL map is shown in Fig. 2.

Generally, two different aspects of random number are required, *real* and *integer*. The real number is required within the range of 0–1, whereas the integer is generally required between 1 and some upper bound like the size of the individual or the population size.

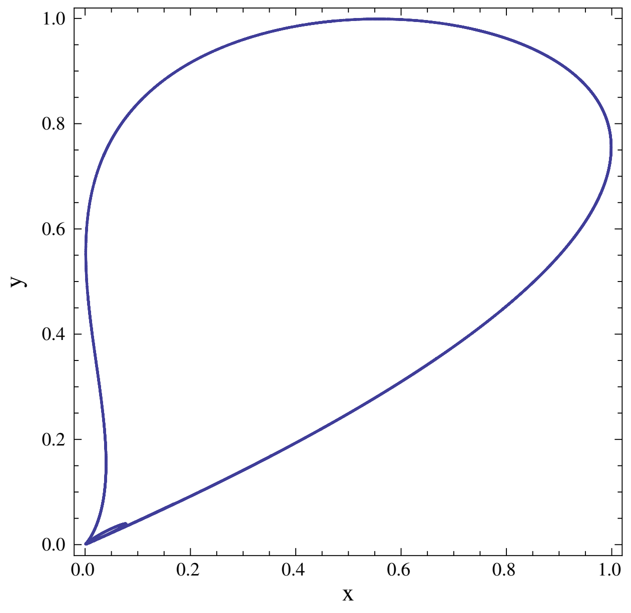

Fig. 2 Delayed logistic

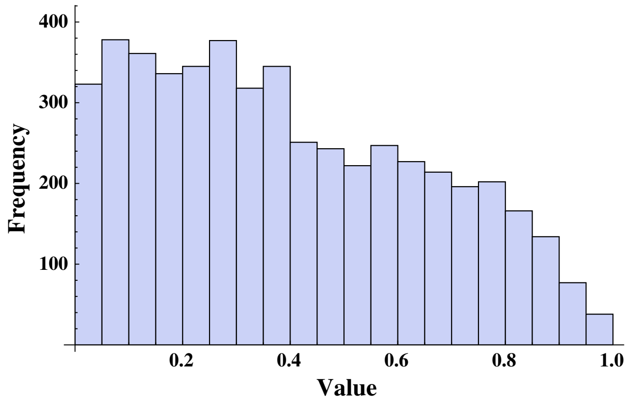

Fig. 3 Lozi map data sample between [0, 1]

Each point of the chaotic system is a two-dimensional point in space. Generally, either of the *x*, or *y* axis values can be used as the random number, however, we have used the data from the *x* axis.

A sample histogram of the data values in the x-axis affixed between 0 and 1 is given in Fig. 3.

In order to obtain numbers between the specified bounds and in the precise order, initially a large number of chaotic numbers are generated using the chaotic map and stored in memory. Based on the requirement, each number is modified sequentially for usage.

4 Discrete self-organising migrating algorithm

DSOMA (Davendra 2009, 2013; Davendra and Bialic-Davendra 2013) is the discrete version of SOMA (Zelinka

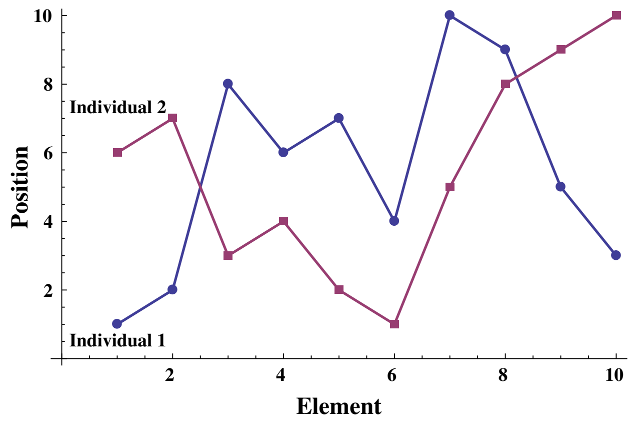

Fig. 4 Two individuals in search space

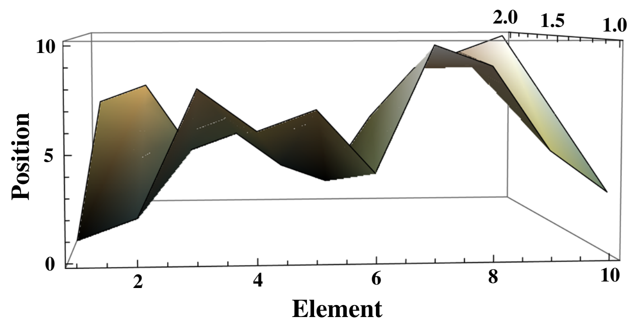

Fig. 5 End view of the two individuals in 3D search space

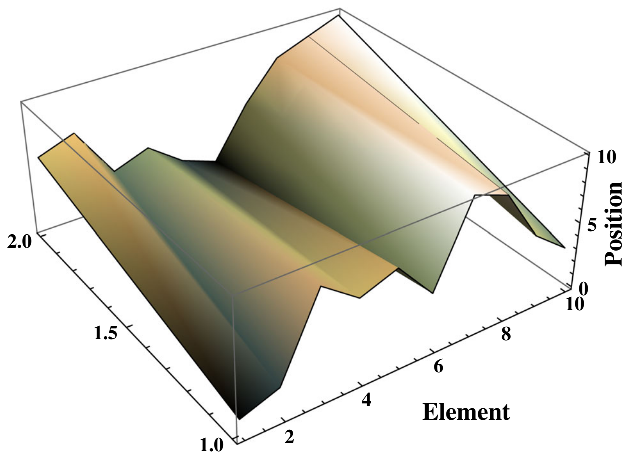

Fig. 6 Isometric view of the two individuals in 3D search space

2004), developed to solve permutation based combinatorial optimisation problem. The SOMA ideology of the sampling of the space between two `individuals` is modified for application in the combinatorial domain. Assume that there are two individuals in a search space. The objective for DSOMA is to transverse from one individual to another, while mapping each discrete space between these two individuals (Fig. 4). Figures 5 and 6 are 3D representations, where the DSOMA mapping is shown as the surface joining these two.

The major input of this algorithm is the sampling of the jump sequence between the individuals in the populations, and the procedure of constructing new trial individuals from these sampled jump sequence elements.

The overall outline for DSOMA can be given as:

1. **Initial phase**

   (a) *Population generation*: An initial number of permutative trial individuals is generated for the initial population.
   (b) *Fitness evaluation*: Each individual is evaluated for its fitness.

2. **DSOMA**

   (a) *Creating jump sequences*: Taking two individuals, a number of possible jump positions is calculated between each corresponding element.
   (b) *Constructing trial individuals*: Using the jump positions; a number of trial individuals is generated. Each element is selected from a jump element between the two individuals.
   (c) *Repairment*: The trial individuals are checked for feasibility and those, which contain an incomplete schedule, are repaired.

3. **Selection**

   (a) *New individual selection*: The new individuals are evaluated for their fitness and the best new fitness based individual replaces the old individual, if it improves upon its fitness.

4. **Generations**

   (a) *Iteration*: Iterate the population till a specified migration.

DSOMA requires a number of parameters as given in Table 1. The major addition is the parameter J_{min}, which gives the minimum number of jumps (sampling) between two individuals. The SOMA variables `PathLength`, `StepSize` and `PRT Vector` which are defined a priori by the user based on the recommendations in Zelinka (2004), are not initialised in DSOMA as they are dynamically calculated by DSOMA using the adjacent elements between the individuals.

4.1 Initialisation

The population is initialised as a permutative schedule representative of the size of the problem at hand (5). As this is

Table 1 DSOMA parameters

Name	Range	Type
J_{min}	(1+)	Minimum number of jumps
Population	10+	Number of individuals
Migrations	10+	Number of iterations

Pseudocode for Generating Initial Population

```
Assume a population $P$ of size $\beta$, a problem of size $N$, and
a schedule given as $X = \{x_1, \ldots, x_N\}$. The fitness can be
given as $C$, while the best individual is represented as $X_L$
with its associated fitness $C_L$ and its index $b$.
1. For $i = 1, 2, \ldots, \beta$ do the following:
    (a) For $j = 1, 2, \ldots, N$ do the following:
        i. Randomly generate a value $x_j$ = rnd int $[1, N]$
        ii. WHILE $x_j \notin X_i$
                Randomly generate a value $x_j$ = rnd int $[1, N]$
        iii. Insert $x_j \rightarrow X_i$
    (b) Insert $X_i \rightarrow P$
    (c) Calculate the fitness of $X_i$ as $C_i = \Im(X_i)$
2. Set $C_L = \min(C)$
3. Set best index $b$ = index of $\min(C)$
4. Set $X_L = X_b$
```

Fig. 7 Pseudocode for generating initial population

the initial population, the superscript of x is initialised to 0. The *rand*() function obtains a value between 0 and 1, and the *INT*() function rounds down the real number to the nearest integer. The *if* condition ensures that each element within the individual is unique.

$$x_{i,j}^0 = \begin{cases} 1 + INT(rand() \cdot (N-1)) \\ \text{if } x_{i,j}^0 \notin \left\{x_{i,1}^0, \ldots, x_{i,j-1}^0\right\} \end{cases} \quad (5)$$
$$i = 1, \ldots, \beta; \; j = 1, \ldots, N$$

Each individual is vetted for its fitness (6), and the best individual, whose index in the population can be assigned as L (leader) and it is designated the leader as X_L^0 with its best fitness given as C_L^0.

$$C_i^0 = \Im\left(X_i^0\right), \quad i = 1, \ldots, \beta \quad (6)$$

The pseudocode for generating a population is given in Fig. 7.

After the generation of the initial population, the migration counter t is set to 1 where $t = 1, \ldots, M$ and the individual index i is initialised to 1, where $i = 1, \ldots, \beta$. Using these values, the following sections (Sects. 4.2, 4.3, 4.4, 4.5) are recursively applied, with the counters i and t being updated in Sects. 4.6 and 4.7 respectively.

4.2 Creating jump sequences

DSOMA operates by calculating the number of discrete jump steps that each individual has to circumnavigate. In DSOMA, the parameter of minimum jumps ($J_{\min}$) is used in lieu of `PathLength`, which states the minimum number of individuals or sampling between two individuals.

Taking two individuals in the population, one as the incumbent (X_i^t) and the other as the leader (X_L^t), the possible number of jump individuals $J_{\max}$ is the mode of the difference between the adjacent values of the elements in the individual (7). A vector J of size N is created to store the difference between the adjacent elements in the individuals. The *mode*() function obtains the most common number in J and designates it as J_{max}.

$$J_j = \left|x_{i,j}^{t-1} - x_{L,j}^{t-1}\right|, \quad j = 1, \ldots, N$$
$$J_{\max} = \begin{cases} mode(J) & if \;\; mode(J) > 0 \\ 1 & otherwise \end{cases} \quad (7)$$

The step size (s), can now be calculated as the integer fraction between the required jumps and possible jumps (8).

$$s = \begin{cases} \left\lfloor \frac{J_{\max}}{J_{\min}} \right\rfloor & \text{if} \quad J_{\max} \geq J_{\min} \\ 1 & \text{otherwise} \end{cases} \quad (8)$$

Create a jump matrix $\mathbf{G}$, which contains all the possible jump positions, that can be calculated as:

$$\mathbf{G}_{l,j} = \begin{cases} x_{i,j}^{t-1} + s \cdot l & \text{if} \quad x_{i,j}^{t-1} + s \cdot l < x_{L,j}^{t-1} \text{ and } x_{i,j}^{t-1} < x_{L,j}^{t-1} \\ x_{i,j}^{t-1} - s \cdot l & \text{if} \quad x_{i,j}^{t-1} + s \cdot l < x_{L,j}^{t-1} \text{ and } x_{i,j}^{t-1} > x_{L,j}^{t-1} \\ 0 & \text{otherwise} \end{cases} \quad (9)$$
$$j = 1, \ldots, N; \; l = 1, \ldots, J_{\min}$$

The pseudocode for creating jump sequences is given in Fig. 8.

4.3 Constructing trial individuals

For each jump sequence of two individuals, a total of $J_{\min}$ new individuals can now be constructed from the jump positions. Taking a new temporary population H ($H = \{Y_1, \ldots, Y_{J_{\min}}\}$), in which each new individual Y_w ($w = 1, \ldots, J_{\min}$), is constructed piecewise from $\mathbf{G}$. Each element $y_{w,j}$

($Y_w = \{y_{w,j}, \ldots, y_{w,N}\}$, $j = 1, 2, \ldots, N$) in the individual, indexes its values from the corresponding jth column in $\mathbf{G}$. Each lth ($l = 1, \ldots, J_{\min}$) position for a specific element is sequentially checked in $\mathbf{G}_{l,j}$ to ascertain if it already

Pseudocode for Creating Jump Sequences

Take the population P with its associated fitness C. The number of jumps (`PathLength`) is given as J_{min} and the step size as s. Assume an empty schedule for storing the possible jump sequences as J and a temporary jump matrix to store the calculated individuals as G, whose size is J_{min} by N, were N is the size of the individual.

```
1. For $i = 1, 2, \ldots, \beta$ do the following:
   (a) For $j = 1, 2, \ldots, N$ do the following:
       i. $J_j = \left| x_{L,j} - x_{i,j} \right|$
   (b) $J_{max} = \text{mode}\ (J)$
   (c) IF $J_{\max} \geq J_{\min}$
           $s = \left\lfloor \frac{J_{\min}}{J_{\max}} \right\rfloor$
       ELSE
           $s = 1$
   (d) For $j = 1, 2, \ldots, N$ do the following:
       i. For $l = 1, 2, \ldots, J_{min}$ do the following:
          A. IF $x_{i,j} < x_{L,j}$
                 $\mathbf{G}_{l,j} = x_{i,j} + s \cdot l$
             ELSE IF $x_{i,j} > x_{L,j}$
                 $\mathbf{G}_{l,j} = x_{i,j} - s \cdot l$
             ELSE
                 $\mathbf{G}_{l,j} = 0$
       ii. Create New Trial Individual as given in Fig.9.
```

Fig. 8 Pseudocode for creating jump sequences

Pseudocode for Constructing Trial Individuals

Take the temporary jump matrix G, of size J_{min} by N, which is now populated with calculated jump sequence elements. Create a temporary population $H = \{Y_1, \ldots, Y_{J_{\min}}\}$, where $Y = \{y_1, \ldots, y_N\}$

```
1. For $w = 1, 2, \ldots, J_{min}$ do the following:
   (a) For $j = 1, 2, \ldots, N$ do the following:
       i. For $l = 1, 2, \ldots, J_{min}$ do the following:
          A. IF
             $\left(\mathbf{G}_{l,j} \notin \{y_{w,1}, \ldots, y_{w,y-1}\}\ \text{AND}\ \mathbf{G}_{l,j} \neq 0\right)$
                 $y_{w,j} = \mathbf{G}_{l,j}$
                 $\mathbf{G}_{l,j} = 0$
       ii. IF $y_{w,j} ==$ NULL
               $y_{w,j} = 0$
2. Repair the temporary population $Y$ in Fig. 10.
```

Fig. 9 Algorithm for constructing trial individuals

exists in the current individual Y_w. If this is a new element, it is then accepted in the individual, and the corresponding lth value is set to zero as $\mathbf{G}_{l,j} = 0$. This iterative procedure can be given as in (10) and the pseudocode for constructing trial individual is represented in Fig. 9.

$$y_{w,j} = \begin{cases} \mathbf{G}_{l,j} \begin{cases} \text{if} & \mathbf{G}_{l,j} \notin \{y_{w,1}, \ldots, y_{w,j-1}\} \\ & \text{and } \mathbf{G}_{l,j} \neq 0; \\ \text{then } \mathbf{G}_{l,j} = 0; \end{cases} \\ 0 \quad \text{otherwise} \end{cases} \tag{10}$$
$$l = 1, \ldots, J_{\min};\ j = 1, \ldots, N;\ w = 1, \ldots, J_{\min}$$

4.4 Repairing trial individuals

Some individuals may exist, which may not contain a permutative schedule. The jump individuals Y_w $(w = 1, 2, \ldots, J_{min})$, are constructed in such a way, that each infeasible element $y_{w,j}$ is indexed by 0.

Taking each jump individual Y_w iteratively from H, the following set of procedures can be applied recursively.

Take A and B, where A is initialised to the permutative schedule $A = \{1, 2, \ldots, N\}$ and B is the complement of individual Y_w relative to A as given in (11).

$$B = A \backslash Y_w \tag{11}$$

If after the complement operation, B is an empty set without any elements; $B = \{\}$, then the individual is correct with a proper permutative schedule and does not require any repairment.

However, if B contains values, then these values are the missing elements in individual Y_w. The repairment procedure is now outlined. The first process is to randomise the positions of the elements in set B. Then, iterating through the elements $y_{w,j}$ $(j = 1, \ldots, N)$ in the individual Y_w, each position, where the element $y_{w,j} = 0$ is replaced by the value in B. Assigning B_{size} as the total number of elements present in B (and hence missing from the individual Y_w), the repairment procedure can be given as in (12).

$$y_{w,j} = \begin{cases} B_h & \text{if } \ y_{w,j} = 0 \\ y_{w,j} & \text{otherwise} \end{cases}$$
$$h = 1, \ldots, B_{size};\ j = 1, \ldots, N \tag{12}$$

After each individual is repaired in H, it is then evaluated for its fitness value as in (13) and stored in γ, the fitness array of size J_{min}.

$$\gamma_w = \Im(Y_w), \quad w = 1, \ldots, J_{\min} \tag{13}$$

The pseudocode for repairing trial individuals is given in Fig. 10.

4.5 Population update

2 Opt local search is applied to the best individual Y_{best} obtained with the minimum fitness value $(\min(\gamma_w))$. After the local search routine, the new individual is compared with the fitness of the incumbent individual X_i^{t-1}, and if it improves on the fitness, then the new individual is accepted in the population (14).

Pseudocode for Repairing Trial Individuals

Take the temporary population H and its associated fitness array γ of size J_{min}. Assume two schedules A and B of maximum size N, where A is initialised to $A = \{1, 2, \ldots, N\}$. The best new trial individual is represented as Y_{best}, with its fitness $\Im(Y_{best})$.

```
1. For $w = 1, 2, \ldots, J_{min}$ do the following:
   (a) $B = A \backslash Y_w$
   (b) IF $B = \{\}$
       Randomise the elements in $B$.
       i. For $j = 1, 2, \ldots, N$ do the following:
          A. SET index $h = 1$
          B. IF $y_{w,j} == 0$
                $y_{w,j} = B_h$
                $h = h + 1$
   (c) Evaluate the fitness of the new trial individual as:
          $\gamma_w = \Im(Y_w)$
   (d) IF $w == 1$
          $Y_{best} = Y_w$
   (e) ELSE IF $\gamma_w < Y_{best}$
          $Y_{best} = Y_w$
```

Fig. 10 Pseudocode for repairing trial individuals

$$X_i^t = \begin{cases} Y_{best} & \text{if } \Im(Y_{best}) < C_i^{t-1} \\ X_i^{t-1} & \text{otherwise} \end{cases} \tag{14}$$

If this individual improves on the overall best individual in the population, it then replaces the best individual in the population (15).

$$X_{best}^t = \begin{cases} Y_{best} & \text{if } \Im(Y_{best}) < C_{best}^t \\ X_{best}^{t-1} & \text{otherwise} \end{cases} \tag{15}$$

4.6 Iteration

Sequentially, incrementing i, the population counter by 1, another individual X_{i+1}^{t-1} is selected from the population, and it begins its own sampling towards the designated leader X_L^{t-1} from Sects. 4.2, 4.3, 4.4 and 4.5. It should be noted that the leader does not change during the evaluation of one migration.

4.7 Migrations

Once all the individuals have executed their sampling towards the designated leader, the migration counter t is incremented by 1. The individual iterator i is reset to 1 (the beginning of the population) and the loop in Sects. 4.2, 4.3, 4.4, 4.5 and 4.6 is re-initiated.

4.8 2 Opt local search

The local search utilised in DSOMA is the 2 Opt local search algorithm. The reason as to why the 2 Opt local search was chosen, is that it is the simplest in the k-opt class of routines. As the DSOMA sampling occurs between two individuals in k-dimension, the local search refines the individual. This, in turn, provides a better probability to find a new leader after each jump sequence. The placement of the local search was refined heuristically during experimentation.

The complexity of this local search is $O(n^2)$. As local search makes up the majority of the complexity time of DSOMA, the overall computational complexity of DSOMA for a single migration is $O(n^3)$.

A schematic of the DSOMA routine is given in Fig. 11, which graphically outlines the procedure for creating jump sequence between two individuals, and constructing trial individuals.

5 Lot-streaming problem

The lot-steaming problem considered in this paper is a subset of the generic flowshop scheduling problem. Whereas, in the permutative flowshop problem, each job n is processed by a single machine m, in a lot-streaming variant, each job is divided into smaller tasks called *lots* (l) (Pan et al. 2011). Once the processing of a sub-lot on its preceding machine is completed, it can be transferred to the downstream machine immediately. However, all $l(j)$ sublots of job j should be processed continuously as no intermingling or exchanging is allowed. A separable sequence-dependent setup time is necessary for the first sub-lot of each job j before it can be processed on any machine k (Pan and Ruiz 2012).

Two different cases of the problem are considered; the idling and the non-idling case. The idling case is a simpler variant of the problem, where only the schedule of the lots are taken into consideration. A non-idling case, on the other hand, is more practical. A non-idle case arise when the machine is not allowed to be idle. This is beneficial, especially in the case where a number of machines are in operation, and resources such as electricity are wasted. Another practical situation is when expensive machinery is employed. Idling on such expensive equipment is often not desired. Other examples come from sectors where less expensive machinery is used but where machines cannot be easily stopped and restarted (Yavuz 2010).

A detailed description of the lot-streaming problem is given in Potts and Baker (1989), Chang et al. (2007) and Sarin and Jaiprakash (2007).

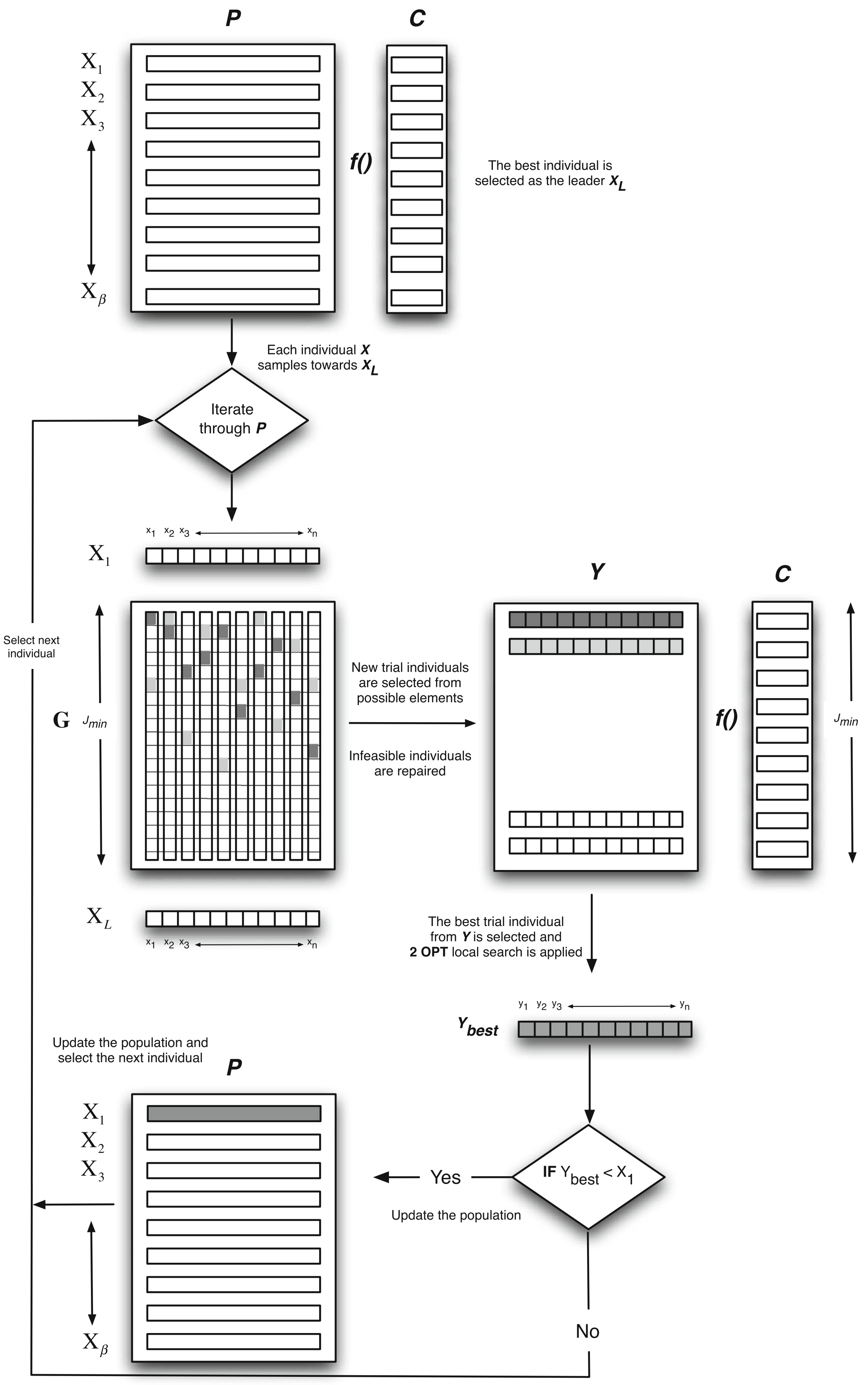
P
C
X_1
X_2
X_3
X_β
f()
The best individual is selected as the leader X_L
Each individual X samples towards X_L
Iterate through P
Select next individual
x_1 x_2 x_3 x_n
X_1
G
J_{min}
New trial individuals are selected from possible elements
Infeasible individuals are repaired
Y
C
f()
J_{min}
X_L
x_1 x_2 x_3 x_n
The best trial individual from Y is selected and 2 OPT local search is applied
y_1 y_2 y_3 y_n
Y_{best}
Update the population and select the next individual
P
X_1
X_2
X_3
X_β
Yes
IF $Y_{best} < X_1$
Update the population
No

Fig. 11 DSOMA schematic

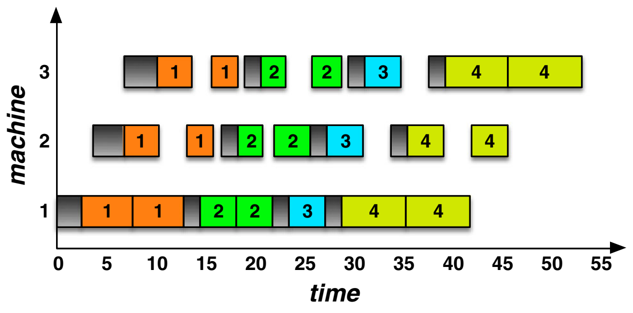

Fig. 12 Idling case of the lot streaming problem

5.1 Idling case

The constraint in this case is that at any given time a machine can process only one sub-lot, and each sub-lot can only be assessed individually. Let the processing time of each sub-lot of job on machine m be $P(m, j)$, and the setup time of job j on machine m, after having processed job j is $s(m, j, j)$, which can also represent the setup time of job j if it is the first job to be proceeded in the machine. The objective is to find a sequence with the optimal sub-lot starting and completion times to minimise the makespan.

The permissible job permutation can be presented as $\pi = \{\pi_1, \pi_2, \ldots, \pi_n\}$, and the earliest start and completion time as $S(m, j, r)$ and $C(m, j, r)$, where r represents the specific sub-lot on job j being processed on machine m.

Initially, the first and third directive of (16) are used to calculate the earliest start time of the first sub-lot of the first job (π_1). The second and fourth directives calculate the completion times, making sure that preemption of jobs is not allowed. Equation (17) controls the earliest start time and the earliest completion time for the successive sub-lots of job π_1, which ensure that sub-lots of the same job are processed continuously.

Equation sets (18) and (20) are used for the calculations for the sub-lots of the following jobs in the π. When calculating the start time for the first sub-lot of a job in (18), the completion time of the previous job on the current machine must be considered with the completion time of the sub-lot on the previous machine, and the setup time of the job on the current machine.

The makespan can be calculated using (20), which is essentially the completion time of the last sub-lot of the last job π_n on the last machine m (Pan and Ruiz 2012).

A schematic of the makespan for the idling case is given in Fig. 12.

$$\begin{aligned}
&S(1, \pi_1, 1) = s(1, \pi_1, \pi_1)\\
&C(1, \pi_1, 1) = S(1, \pi_1, 1) + P(1, \pi_1);\\
&S(w, \pi_1, 1) = \max\{C(w-1, \pi_1, 1), s(w, \pi_1, \pi_1)\}, \qquad (16)\\
&C(w, \pi_1, 1) = S(w, \pi_1, 1) + +P(w, \pi_1),\\
&\quad w = 2, 3, \ldots, m
\end{aligned}$$

$$\begin{aligned}
&S(1, \pi_1, r) = C(1, \pi_1, r-1),\\
&C(1, \pi_1, r) = S(1, \pi_1, r) + P(1, \pi_1),\\
&\quad r = 2, 3, \ldots, l(\pi_1); \qquad (17)\\
&S(w, \pi_1, r) = \max\{C(w-1, \pi_1, r), C(, \pi_1, r-1)\},\\
&C(w, \pi_1, r) = S(w, \pi_1, r) + P(w, \pi_1),\\
&\quad r = 2, 3, \ldots, l(\pi_1),\ w = 2, 3, \ldots, m
\end{aligned}$$

$$\begin{aligned}
&S(1, \pi_1, 1) = C(1, \pi_{i-1}, l(\pi_{i-1})) + s(1, \pi_{i-1}, \pi_i),\\
&C(1, \pi_1, 1) = S(1, \pi_1, 1) + P(1, \pi_1), i = 2, 3, \ldots, n;\\
&S(w, \pi_1, 1) = \max\left\{\begin{array}{c} C(w-1, \pi_1, r),\\ C(w, \pi_{i-1}, l(\pi_{i-1}))\\ +s(w, \pi_{i-1}, \pi_i)\end{array}\right\}, \qquad (18)\\
&C(w, \pi_1, 1) = S(w, \pi_1, 1) + P(w, \pi_i),\\
&\quad i = 2, 3, \ldots, n, w = 2, 3, \ldots, m
\end{aligned}$$

$$\begin{aligned}
&S(1, \pi_1, r) = C(1, \pi_i, r-1),\\
&C(1, \pi_1, r) = S(1, \pi_1, r) + P(1, \pi_1) \qquad (19)\\
&\quad i = 2, 3, \ldots, n,\quad r = 2, 3, \ldots, l(\pi_i);\\
&S(w, \pi_1, r) = \max\{C(w-1, \pi_1, r), C(w, \pi_{i-1}, r-1)\},\\
&C(w, \pi_1, r) = S(w, \pi_1, r) + P(w, \pi_i),\\
&\quad i = 2, 3, \ldots, n,\quad r = 2, 3, \ldots, l(\pi_i), w = 2, 3, \ldots, m
\end{aligned}$$

$$C_{\max}(\pi) = C_T(m, \pi_n, l(\pi_n)) \qquad (20)$$

5.2 Non-idling case

For the non-idling case, the earliest start time for the first sub-lot is given in (21) and (23), where the start time is the maximum of the setup time of the job in the current machine, the completion time of the first sub-lot on the previous machine, and the difference between the completion time of the whole job on the previous machine and the total processing time of the whole job on the preceding machine except the last sub-lot. This ensures that there is no idling time between two adjacent sub-lots. The last two directives of these equations calculate the completion time for the first job.

The subsequent processing time of the next job sequence is given in (23) and (24).

A schematic of the makespan for the non-idling case is given in Fig. 13.

$$\begin{aligned}
&S(1, \pi_1, 1) = s(1, \pi_i, \pi_i)\\
&C(1, \pi_1, l(\pi_1)) = S(1, \pi_1, 1) + l(\pi_1) \times P(1, \pi_1) \qquad (21)\\
&S(w, \pi_1, 1) = \max\left\{\begin{array}{l} s(w, \pi_1, \pi_1), S(w-1, \pi_1, 1)\\ \quad +p(w-1, \pi_1),\\ C(k-1, \pi_1, l(\pi_1))\\ \quad -(l(\pi_1)-1) \times P(1, \pi_1)\end{array}\right\}, \qquad (22)\\
&C(w, \pi_1, l(\pi_1)) = S(w, \pi_1, 1) + l(\pi_1) \times P(w, \pi_i),
\end{aligned}$$

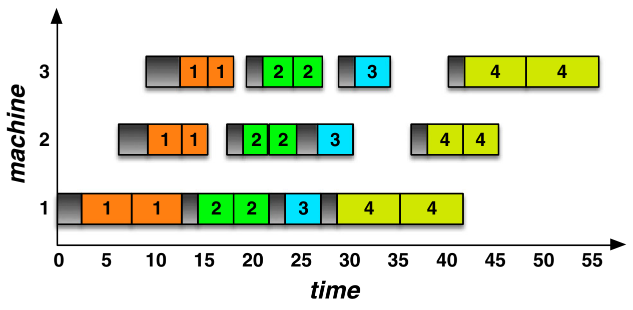

Fig. 13 Non-idling case of the lot streaming problem

$$\begin{aligned}
&w = 2, 3, \dots, m \\
&S(1, \pi_1, 1) = C(1, \pi_{i-1}, l(\pi_{i-1})) + s(1, \pi_{i-1}, \pi_i), \\
&C(1, \pi_1, l(\pi_1)) = S(1, \pi_1, 1) + l(\pi_1) \times P(1, \pi_1), \qquad (23) \\
&i = 2, 3, \dots, n \\
&S(w, \pi_1, 1) = \max \left\{ \begin{array}{l} S(w-1, \pi_i, 1) + P(w-1, \pi_1), \\ C(w-1, \pi_1, l(\pi_1)) - (l(\pi_1) - 1) \\ \quad \times P(1, \pi_1), \\ C(w-1, \pi_{i-1}, l(\pi_{i-1})) \\ \quad + s(1, \pi_{i-1}, \pi_i) \end{array} \right\}, \\
&\qquad i = 2, 3, \dots, n, \quad w = 2, 3, \dots, m \\
&C(w, \pi_1, l(\pi_1)) = S(w, \pi_1, 1) + l(\pi_1) \times P(w, \pi_i), \\
&i = 2, 3, \dots, n, \ w = 2, 3, \dots, m \qquad (24)
\end{aligned}$$

The makespan for the non-idling case can be then calculated as (25).

$$C_{\max}(\pi) = C_T(m, \pi_n, l(\pi_n)) \qquad (25)$$

The objective of the lot-streaming flow shop scheduling problem with makespan criterion is to find a permutation π^* in the set of all permutations $\prod$ which can be given as in (26) (Pan and Ruiz 2012).

$$C_{\max}(\pi^*) \leq C_{\max}(\pi), \quad \forall \pi \in \Pi \qquad (26)$$

6 Data sets generation

In keeping with the theme of utilising the chaotic maps in lieu of PRNG, the data sets have been generated using two unique chaotic maps; the Lozi and the Delayed Logistic map. Five unique sizes of data sets have been generated. They are from 10 job $\times$ 5 machine, 20 job $\times$ 10 machine, 50 job $\times$ 25 machine, 75 job $\times$ 30 machine and 100 job$\times$ 50 machine. There are five instances for each data set, therefore, in total 25 data set instances for each of the Lozi and Delayed Logistic data sets.

In order to have *unique* data sets, each *instance* was initialised from a unique *start* position of the respective chaotic system. Additionally, the map was not allowed to be reinitialised. Two different maps were used in order to have more versatility in the data sets and to remove any particular bias.

The data sets are available at Davendra (2012) for download.

7 Experimentation and analysis

The main emphasis of this work is to show the improvement of applying the chaotic map to DSOMA. Therefore, the experimentation compares the generic and chaos induced DSOMA (DSOMA$_C$) algorithms on the same problem sets, using the identical parameter settings.

The parameter settings for DSOMA and DSOMA$_C$ are given in Table 2. The experiments were conducted on a MacBook Pro, Sandy Bridge 2.3 GHz Intel Core i7, 8 GB 1,333 MHz DDR3 RAM.

7.1 Lozi generated data sets results

The results of the Lozi generated data sets are given in Table 3 for the idling case and Table 4 for the non-idling case. A total of five experiments have been performed on each data set, and the results are presented as the summation of the five experiments for the different problem instance sizes. Four different statistical analysis have been performed on the results. The minimum, maximum, average and execution time computation are presented. Additionally, the *t* test for the different data sets for the two different problems was conducted and is given in Tables 5 and 6. For the idling case in Table 3, DSOMA$_C$ obtains better average minimum (8,684) and average (8,919.84) value compared to 8,899 and 9,248.48 for DSOMA. Furthermore, it obtains more minimum (5) and average values (5) for the individual data sets. In terms of the *t* test, DSOMA$_C$ is significantly better than DSOMA on three data sets of 20 $\times$ 10, 50 $\times$ 25 and 75 $\times$ 30 in the idling case, and both algorithms are significantly equal on the remaining two data sets.

For the non-idling case given in Table 4, the DSOMA and DSOMA$_C$ algorithms performs equally. DSOMA manages to obtain better average minimum (9,863) and average (10,173.24) values compared to 9,878 and 10,179.32 for DSOMA$_C$. In terms of individual instances, DSOMA and DSOMA$_C$ have almost identical performances; average (2) and minimum (3) for DSOMA compared to average (3) and

Table 2 Parameter settings

Parameters	Value
Population	40
Migration	20
J_{min}	20
Local search	2 opt

Table 3 Lozi idling results

Instance	DSOMA				DSOMA$_C$			
	Average	Min	Max	Time (s)	Average	Min	Max	Time (s)
10 × 5	**429.8**	**383**	464	1.87	**429.8**	**383**	464	2.32
20 × 10	1,778	1,680	1,885	45.03	**1,727.8**	**1,644**	1,822	39.96
50 × 25	24,872.2	24,170	25,404	504.7	**24,111.0**	**23,361**	24,569	642.76
75 × 30	7,584.6	7,355	7,714	1,023.7	**7,279.0**	**7,145**	7,324	1,065.32
100 × 50	11,577.8	10,907	11,362	2,953.21	**11,051.6**	**10,887**	11,216	2,855.6
Average	9,248.48	8,899	9,365.8	**905.7**	**8,919.84**	**8,684**	9,079	921.19

Table 4 Lozi non-idling results

Instance	DSOMA				DSOMA$_C$			
	Average	Min	Max	Time (s)	Average	Min	Max	Time (s)
10 × 5	513	431	557	2.22	**511.2**	**425**	552	2.46
20 × 10	**2,107.6**	**1,979**	2,210	39.43	2,108.0	**1,979**	2,210	42.86
50 × 25	**27,852.4**	27,164	28,558	528.32	27,889.6	**27,100**	28,407	536.45
75 × 30	8,067.6	**7,819**	8,230	821.40	**8,063.4**	7,832	8,259	846.43
100 × 50	12,325.6	**11,922**	12,951	3,688.20	**12,324.4**	12,054	12,954	3,865.30
Average	**10,173.2**	**9,863**	10,501.2	**1,015.91**	10,179.3	9,878	10,476.4	1,058.70

Table 5 Paired *t* test for the Lozi idling results

Data set	*t* value	*p* value	$p < 0.05$	Result
10 × 5	–	–	–	–
20 × 10	8.28	0.001	Yes	DSOMA$_c$
50 × 25	14.32	0.0001	Yes	DSOMA$_c$
75 × 30	10.15	0.0005	Yes	DSOMA$_c$
100 × 50	2.22	0.089	No	Same

Table 6 Paired *t* test for the Lozi non-idling results

Data set	*t* value	*p* value	$p < 0.05$	Result
10 × 5	1.15	0.313	No	Same
20 × 10	0.196	0.854	No	Same
50 × 25	0.518	0.613	No	Same
75 × 30	0.188	0.859	No	Same
100 × 50	0.026	0.98	No	Same

minimum (3) for DSOMA$_C$. The *t* test results for all the data instances show that the two algorithms are significantly not different.

7.2 Delayed Logistic generated data sets results

For the Delayed Logistic problem sets, the results are given in Table 7 for the idling case and Table 8 for the non-idling case. The *t* test results for the different data sets are given in Tables 9 and 10. For the idling case, DSOMA$_C$ is better performing on all problem instances having the better average (10,173.24) and minimum (12,176) values. Analysing the *t* test results, DSOMA$_C$ is significantly better than DSOMA on the two instances of 50 × 25 and 100 × 50.

The non-idling case also affirms better performance of the DSOMA$_C$ algorithm. Following the trend in the previous experiment, the chaos based DSOMA has better average minimum (13,379.8) and average (13,963.16) values and individual minimum (5) and average (4) values. From the *t* test results DSOMA$_C$ is significantly better than DSOMA on four instances.

8 Conclusion

This research is based on ascertaining the effectiveness of applying the Lozi map to the DSOMA algorithm in order to solve the lot-streaming flowshop scheduling problem. In total, 50 data sets were generated using the Lozi and Delayed Logistic maps, which were solved for both the idling and non-idling case. In total, five experimentations for each data set were conducted, therefore a total of 500 individual experiments were conducted to validate the finding in this paper.

From the summarised results in Table 11, it is obvious that the DSOMA$_C$ algorithm is better performing than the generic DSOMA algorithm. Of all the valid compared parameters, DSOMA$_C$ obtained more minimum values, 18 compared to 5 and better average values, 17 compared to 4 for the DSOMA algorithm. This shows that DSOMA$_C$ is more robust and has better probability of finding a better solu-

Table 7 Delayed Logistic idling results

Instance	DSOMA				DSOMA$_C$			
	Average	Min	Max	Time (s)	Average	Min	Max	Time (s)
10 × 5	630.6	568	699	2.65	**626.2**	**566**	693	2.34
20 × 10	2,049.8	1,869	2,279	39.54	**2,040.6**	**1,859**	2,264	40.32
50 × 25	13,095.0	11,633	14,392	612.32	**12,789.8**	**11,390**	14,018	623.43
75 × 30	19,256.2	18,789	19,740	967.43	**18,998.8**	**18,493**	19,260	976.32
100 × 50	29,868.8	28,884	31,163	3,145.32	**29,516.8**	**28,572**	30,824	3,212.22
Average	12,980.1	12,348.6	13,654.6	**953.45**	**12,794.4**	**12,176**	13,411.8	970.93

Table 8 Delayed Logistic non-idling results

Instance	DSOMA				DSOMA$_C$			
	Average	Min	Max	Time (s)	Average	Min	Max	Time (s)
10 × 5	**700.6**	**613**	762	2.01	701.2	**613**	762	1.97
20 × 10	2,274.8	2,146	2,461	36.32	**2,234.2**	**2,120**	2,416	37.24
50 × 25	14,524.0	12,897	15,705	540.32	**14,207.0**	**12,816**	15,455	544.23
75 × 30	21,115.0	20,855	21,398	877.43	**20,535.8**	**20,244**	20,903	892.43
100 × 50	32,752.0	31,742	33,622	3,302.32	**32,137.6**	**31,106**	33,539	3,331.4
Average	14,273.3	13,650.6	14,789.6	**951.68**	**13,963.2**	**13,379.8**	14,615	961.45

Table 9 Paired *t* test for the Delayed Logistic idling results

Data set	*t* value	*p* value	$p < 0.05$	Result
10 × 5	1.53	0.2	No	Same
20 × 10	1.23	0.283	No	Same
50 × 25	5.42	0.005	Yes	DSOMA$_c$
75 × 30	1.49	0.209	No	Same
100 × 50	6.04	0.003	Yes	DSOMA$_c$

Table 10 Paired *t* test for the Delayed Logistic non-idling results

Data set	*t* value	*p* value	$p < 0.05$	Result
10 × 5	1	0.37	No	Same
20 × 10	6.38	0.003	Yes	DSOMA$_C$
50 × 25	4.125	0.014	Yes	DSOMA$_C$
75 × 30	7.96	0.0013	Yes	DSOMA$_C$
100 × 50	0.73	0.018	Yes	DSOMA$_C$

Table 11 Summarised results

Instance	DSOMA		DSOMA$_C$	
	Min	Average	Min	Average
10 × 5	2	2	**4**	**3**
20 × 10	1	1	**4**	**3**
50 × 25	0	1	**4**	**3**
75 × 30	1	0	**3**	**4**
100 × 50	1	0	**3**	**4**
Total	5	4	**18**	**17**

tion. In terms of *t* test, DSOMA$_C$ is significantly better than DSOMA on nine data sets and equal in the remaining 11 data sets. This hypothesis raises the main issue regarding the importance of PRNG in stochastic algorithms. Whereas, the defining argument during the past decades has been the application of better crossover and mutation routines, the generator for selection and mutation, PRNG now gains more prominence.

The downside of the experiment invariably has been the execution time, where there was a slight increase for the chaos version. However, this can be improved significantly through utilising the GPU for computing. This will be directed as a further aspect of this research, alongside the application of different chaotic maps as generators. The implementation of different chaos maps is important in the deduction aspect of which maps are compatible for the different problem classes.

Acknowledgments Donald Davendra was supported by the Technology Agency of the Czech Republic under the Project TE01020197 and Michal Pluhacek was supported by the Internal Grant Agency of Tomas Bata University under the project No. IGA/FAI/2013/012.

References

Alatas B, Akin E, Ozer A (2009) Chaos embedded particle swarm optimization algorithms. Chaos Solitons Fractals 40(4):1715–1734

Caponetto R, Fortuna L, Fazzino S, Xibilia M (2003) Chaotic sequences to improve the performance of evolutionary algorithms. IEEE Trans Evol Comput 7(3):289–304

Chang JL, Gong DW, Ma XP (2007) A heuristic genetic algorithm for no-wait flowshop scheduling problem. J China Univ Min Technol 17(4):582–586

Davendra D (2009) Chaotic attributes and permutative optimization. Ph.D. thesis, Tomas Bata University in Zlin, Zlin

Davendra D (2012) Flowshop lot-streaming problem data sets. http://mrl.cs.vsb.cz/people/davendra/research.html

Davendra D, Bialic-Davendra M (2013) Scheduling flow shops with blocking using a discrete self-organising migrating algorithm. Int J Prod Res 51(8):2200–2218. doi:10.1080/00207543.2012.711968

Davendra D, Zelinka I, Senkerik R, Bialic-Davendra M (2010) Chaos driven evolutionary algorithm for the traveling salesman problem. In: Davendra D (ed) Traveling salesman problem, theory and applications. InTech Publishing, Croatia, pp 55–70

Davendra D, Zelinka I, Bialic-Davendra M, Senkerik R, Jasek R (2013) Discrete self-organising migrating algorithm for flow-shop scheduling with no-wait makespan. Math Comput Model 57(12):100–110. doi:10.1016/j.mcm.2011.05.029

Davendra D, Zelinka I, Senkerik R (2010) Chaos driven evolutionary algorithms for the task of pid control. Comput Math Appl 60(4):1088–1104

Hennon M (1979) A two-dimensional mapping with a strange attractor. Commun Math Phys 50:69–77

Herring C, Julian P (1989) Random number generators are chaotic. ACM SIGPLAN 11:1–4

Hilborn R (2000) Chaos and nonlinear dynamics: an introduction for scientists and engineers. OUP, Oxford

Lehmer D (1951) Mathematical methods in large-scale computing units. Ann Comput Lab (Harvard University) 26:141–146

Lozi R (2008) New enhanced chaotic number generators. Indian J Ind Appl Math 1(1):1–23

Lozi R (2009) Chaotic pseudo random number generators via ultra weak coupling of chaotic maps and double threshold sampling sequences. In: ICCSA 2009 the 3rd international conference on complex systems and applications. University of Le Havre, France, pp 1–5

Lu Y, Zhou J, Qin H, Wang Y, Zhang Y (2011) Chaotic differential evolution methods for dynamic economic dispatch with valve-point effects. Eng Appl Artif Intell 24(2):378–387

Matsumoto M (2012) Mersenne twister webpage. http://www.math.sci.hiroshima-u.ac.jp/m-mat/MT/ARTICLES/earticles.html

Matsumoto M, Nishimura T (1998) Mersenne twister: a 623-dimensionally equidistributed uniform pseudorandom number generator. ACM Trans Model Comput Simul 8(1):3–30

Ozer A (2010) Cide: chaotically initialized differential evolution. Expert Syst Appl 37(6):4632–4641

Palmore J, McCauley J (1987) Shadowing by computable chaotic orbits. Phys Lett A 121:399

Pan QK, Fatih Tasgetiren M, Suganthan PN, Chua TJ (2011) A discrete artificial bee colony algorithm for the lot-streaming flow shop scheduling problem. Inf Sci 181(12):2455–2468. doi:10.1016/j.ins.2009.12.025

Pan QK, Ruiz R (2012) An estimation of distribution algorithm for lot-streaming flow shop problems with setup times. Omega 40(2):166–180

Potts CN, Baker KR (1989) Flow shop scheduling with lot streaming. Oper Res Lett 8(6):297–303

Price K (1999) An introduction to differential evolution. In: Corne D, Dorigo M, Glover F (eds) New ideas in optimisation. McGraw Hill International, UK

Sarin SC, Jaiprakash P (2007) Flow shop lot streaming. Springer, Berlin

Stojanovski T, Kocarev L (2001) Chaos-based random number generators part I: analysis. IEEE Trans Circuits Syst I Fundam Theory Appl 48(3):281–288

Yavuz M (2010) Fuzzy lead time management. In: Kahraman C, Yavuz M (eds) Production engineering and management under fuzziness, studies in fuzziness and soft computing, vol 252. Springer, Berlin/Heidelberg, pp 77–94

Yuan X, Cao B, Yang B, Yuan Y (2008) Hydrothermal scheduling using chaotic hybrid differential evolution. Energy Convers Manag 49(12):3627–3633

Zelinka I (2004) Soma—self organizing migrating algorithm. In: Onwubolu G, Babu B (eds) New optimization techniques in engineering. Springer-Verlag, Germany

Zuo X, Fan Y (2006) A chaos search immune algorithm with its application to neuro-fuzzy controller design. Chaos Solitons Fractals 30(1):94–109