DSOMA—Discrete Self Organising Migrating Algorithm

Donald Davendra, Ivan Zelinka, Michal Pluhacek and Roman Senkerik

Abstract A discrete Self Organising Migrating Algorithm (DSOAM) is described in this chapter. This variant is specifically designed for the permutative based combinatorial optimisation problem, where the problem domain in generally NP-Hard. Specific sampling between individuals in the search space is introduced as a means of constructing new feasible individuals. These feasible solutions are improved using 2-Opt routines. DSOMA has proven successful in solving manufacturing scheduling and assignment problems.

1 Introduction

Complex engineering problems can be loosely defined into three main domains. Unimodal and multimodal real domain problems generally deal with floating point values. Combinatorial optimisation problems on the other hand deal with integer based values. In *strict sense* combinatorial optimisation problems, the values are

D. Davendra (✉)
Department of Computer Science, Central Washington University, 400 E. University Way, Ellensburg, WA 98926-7520, USA
e-mail: DonaldD@cwu.edu; donald.davendra@vsb.cz

D. Davendra · I. Zelinka
Department of Computer Science, Faculty of Electrical Engineering and Computer Science, VSB-Technical University of Ostrava, 17. Listopadu 15, 708 33 Ostrava-Poruba, Czech Republic
e-mail: ivan.zelinka@vsb.cz

M. Pluhacek · R. Senkerik
Department of Informatics and Artificial Intelligence, Faculty of Applied Informatics, Tomas Bata University in Zlin, Nad Stranemi 4511, 76005 Zlin, Czech Republic
e-mail: pluhacek@fai.utb.cz

R. Senkerik
e-mail: senkerik@fai.utb.cz

D. Davendra and I. Zelinka (eds.), *Self-Organizing Migrating Algorithm*, Studies in Computational Intelligence 626, DOI 10.1007/978-3-319-28161-2_2

inherently permutative. *Wide sense* combinatorial optimisation problems deal with integer values within specific ranges. The third domain in simply a mixture of the first two, and is generally called mixed integer-real optimisation.

Combinatorial optimisation problems are generally considered *NP-Hard*, with the associated decision problems formulated as *NP-Complete*. A range of different problems in engineering have been classified as combinatorial optimisation problems. The most common ones are scheduling problems, routing and assignment problems amongst others.

Scheduling problems such as flowshop, openshop and jobshop are readily identifiable as the most common manufacturing problems whereas routing problems such as vehicle routing and traveling salesman problems are classic mathematical problems, and forms the general basis for the *P* versus *NP Problem* Millennium problem [7].

Since these problems are *NP-Hard*, specific deterministic and stochastic algorithms have been developed to solve these problems in reasonable time with given resources. One of the powerful stochastic algorithms are evolutionary algorithms (EA). However, general canonical EA's for real-domain problems cannot be used to solve combinatorial optimisation problems. Therefore, a separate class of *discrete* EA's have been developed to deal with this domain of problem.

All established metaheuristics have developed discrete variants such as: Genetic algorithms [4, 8, 13], Ant Colony Optimisation [5], Tabu Search [14], Particle Swarm [11, 15], Differential Evolution [9, 10], Artificial Bee Colony [12, 16] and Harmony Search [1, 6].

A discrete variant of the Self-Organising Migrating Algorithm (SOMA) has been developed to solve combinatorial optimisation problems [2, 3]. This chapter details the discrete Self-Organising Migrating Algorithm (DSOMA), with population initialisation, creating jump sequences, constructing trial individual, repairment and selection.

2 Discrete Self-organising Migrating Algorithm

Discrete Self-Organising Migrating Algorithm (DSOMA) [3] is the discrete version of SOMA, developed to solve permutation based combinatorial optimisation problem. The same ideology of the sampling of the space between two individuals is retained. Assume that there are two individuals in a search space as given in Fig. 1. The objective for DSOMA is to transverse from one individual to another, while mapping each discrete space between these two individuals. Figures 2 and 3 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.

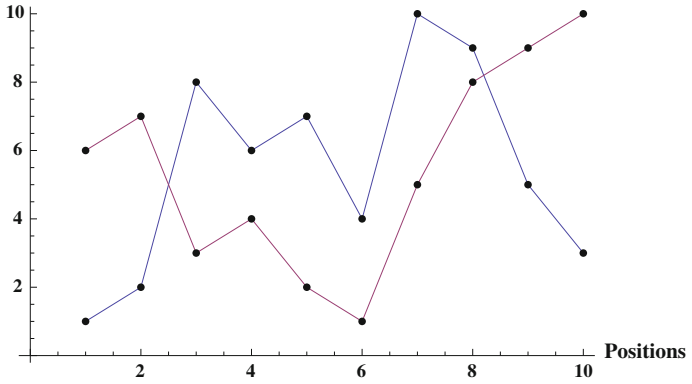

Fig. 1 Two individuals in search space

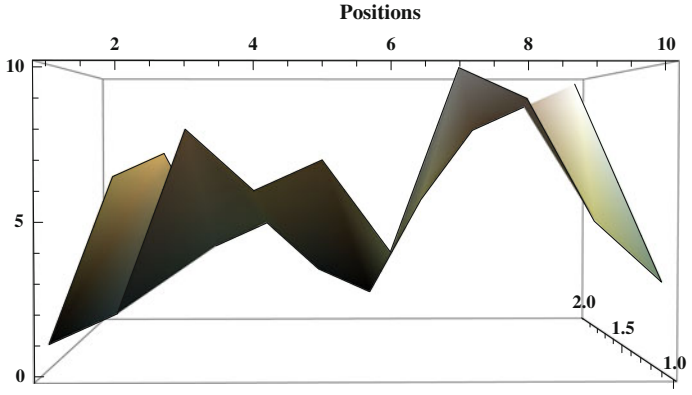

Fig. 2 End view of the two individuals in 3D search space

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.

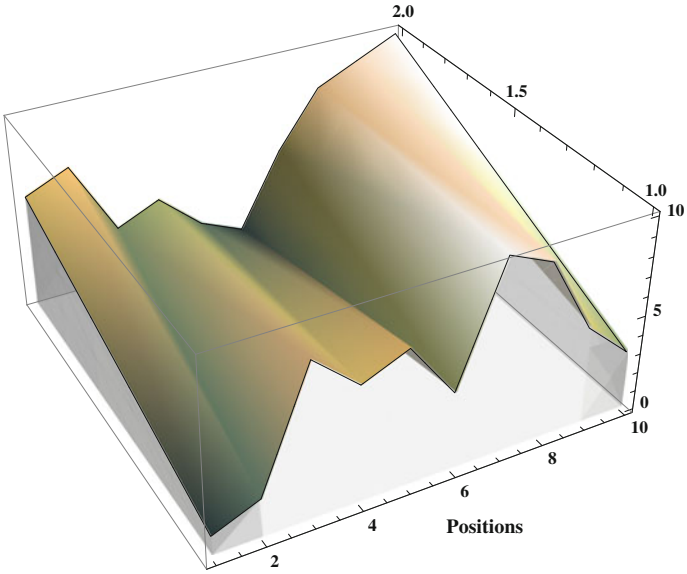

Fig. 3 Isometric view of the two individuals in 3D search space

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 are not initialised as they are dynamically calculated by DSOMA using the adjacent elements between the individuals.

Table 1 DSOMA parameters

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

3 Initialisation

The population is initialised as a permutative schedule representative of the size of the problem at hand (1). As this is the initial population, the superscript of x its set 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 checks to ensure that each element within the individual is unique.

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

Each individual is vetted for its fitness (2), 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.

$$C_i^0 = \Im\left(X_i^0\right), \quad i = 1, \ldots, \beta \tag{2}$$

The pseudocode for generating a population is given in Fig. 4.

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. 4–7) are recursively applied, with the counters i and t being updated in Sects. 8 and 9 respectively.

Pseudocode for Generating Initial Population

Assume a population P of size β, 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 $x_j = \text{rnd int } [1, \text{N}]$
      ii. **WHILE** $x_j \notin X_i$
          Randomly generate a value $x_j = \text{rnd int } [1, \text{N}]$
      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. 4 Pseudocode for generating initial population

4 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 the 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 (3). 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 $J_{\max}$.

$$\begin{aligned} J_j &= \left| x_{i,j}^{t-1} - x_{L,j}^{t-1} \right|, \quad j = 1, \ldots, N \\ J_{\max} &= \begin{cases} mode(J) & if\ mode(J) > 0 \\ 1 & otherwise \end{cases} \end{aligned} \tag{3}$$

The step size (s), can now be calculated as the integer fraction between the required jumps and possible jumps (4).

$$s = \begin{cases} \left\lfloor \frac{J_{\max}}{J_{\min}} \right\rfloor & \text{if } J_{\max} \geq J_{\min} \\ 1 & \text{otherwise} \end{cases} \tag{4}$$

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

$$\begin{aligned} \mathbf{G}_{l,j} &= \begin{cases} x_{i,j}^{t-1} + s \cdot l & \text{if } x_{i,j}^{t-1} + s \cdot l < x_{L,j}^{t-1} \text{ and } x_{i,j}^{t-1} < x_{L,j}^{t-1} \\ x_{i,j}^{t-1} - s \cdot l & \text{if } x_{i,j}^{t-1} + s \cdot l < x_{L,j}^{t-1} \text{ and } x_{i,j}^{t-1} > x_{L,j}^{t-1} \\ 0 & \text{otherwise} \end{cases} \\ & j = 1, \ldots, N;\ l = 1, \ldots, J_{\min} \end{aligned} \tag{5}$$

The pseudocode for creating jump sequences is given in Fig. 5.

5 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 $y_{w,j}$ $\left(Y_w = \{y_{w,j}, \ldots, y_{w,N}\},\ j = 1, 2, \ldots, N\right)$ in the individual, indexes its values from the corresponding jth column

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** $J_{\max} \geq J_{\min}$

      $s = \left\lfloor \frac{J_{\max}}{J_{\min}} \right\rfloor$

      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 Figure 2.6.

Fig. 5 Pseudocode for creating jump sequences

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 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 Eq. (6) and the pseudocode for constructing trial individual is represented in Fig. 6.

$$y_{w,j} = \begin{cases} \mathbf{G}_{l,j} \begin{cases} \text{if} \quad \mathbf{G}_{l,j} \notin \left\{ y_{w,1}, \ldots, y_{w,j-1} \right\} \text{ and } \mathbf{G}_{l,j} \neq 0; \\ \text{then } \mathbf{G}_{l,j} = 0; \end{cases} \\ 0 \quad \text{otherwise} \end{cases} \tag{6}$$
$$l = 1, \ldots, J_{\min};\ j = 1, \ldots, N;\ w = 1, \ldots, J_{\min}$$

6 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.

Pseudocode for Constructing Trial Individuals

Take the temporary jump matrix G, of size is 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** $(\mathbf{G}_{l,j} \notin \{y_{w,1}, \ldots, y_{w,y-1}\}$ AND $\mathbf{G}_{l,j} \neq 0)$
            $y_{w,j} = \mathbf{G}_{l,j}$
            $\mathbf{G}_{l,j} = 0$
      ii. **IF** $y_{w,j} ==$ NULL
            $y_{w,j} = 0$
2. Repair the temporary population Y in Figure 2.7.

Fig. 6 Algorithm for constructing trial individuals

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 Eq. (7).

$$B = A \backslash Y_w \tag{7}$$

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 Eq. (8).

$$y_{w,j} = \begin{cases} B_h & \text{if } y_{w,j} = 0 \\ y_{w,j} & \text{otherwise} \end{cases} \qquad h = 1, \ldots, B_{size};\ j = 1, \ldots, N \tag{8}$$

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. 7 Pseudocode for repairing trial individuals

After each individual is repaired in H, it is then evaluated for its fitness value as in Eq. (9) and stored in γ, the fitness array of size $J_{\min}$.

$$\gamma_w = \Im(Y_w), \quad w = 1, \ldots, J_{\min} \tag{9}$$

The pseudocode for repairing trial individuals is given in Fig. 7.

7 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 (10).

$$X_i^t = \begin{cases} Y_{best} & \text{if } \Im(Y_{best}) < C_i^{t-1} \\ X_i^{t-1} & \text{otherwise} \end{cases} \tag{10}$$

If this individual improves on the overall best individual in the population, it then replaces the best individual in the population (11).

$$X_{best}^{t} = \begin{cases} Y_{best} & \text{if } \Im(Y_{best}) < C_{best}^{t} \\ X_{best}^{t-1} & \text{otherwise} \end{cases} \tag{11}$$

8 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. 4–7. It should be noted that the leader does not change during the evaluation of one migration.

9 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. 4–8 is re-initiated.

10 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. 8, which graphically outlines the procedure for creating jump sequence between two individuals, and constructing trial individuals.

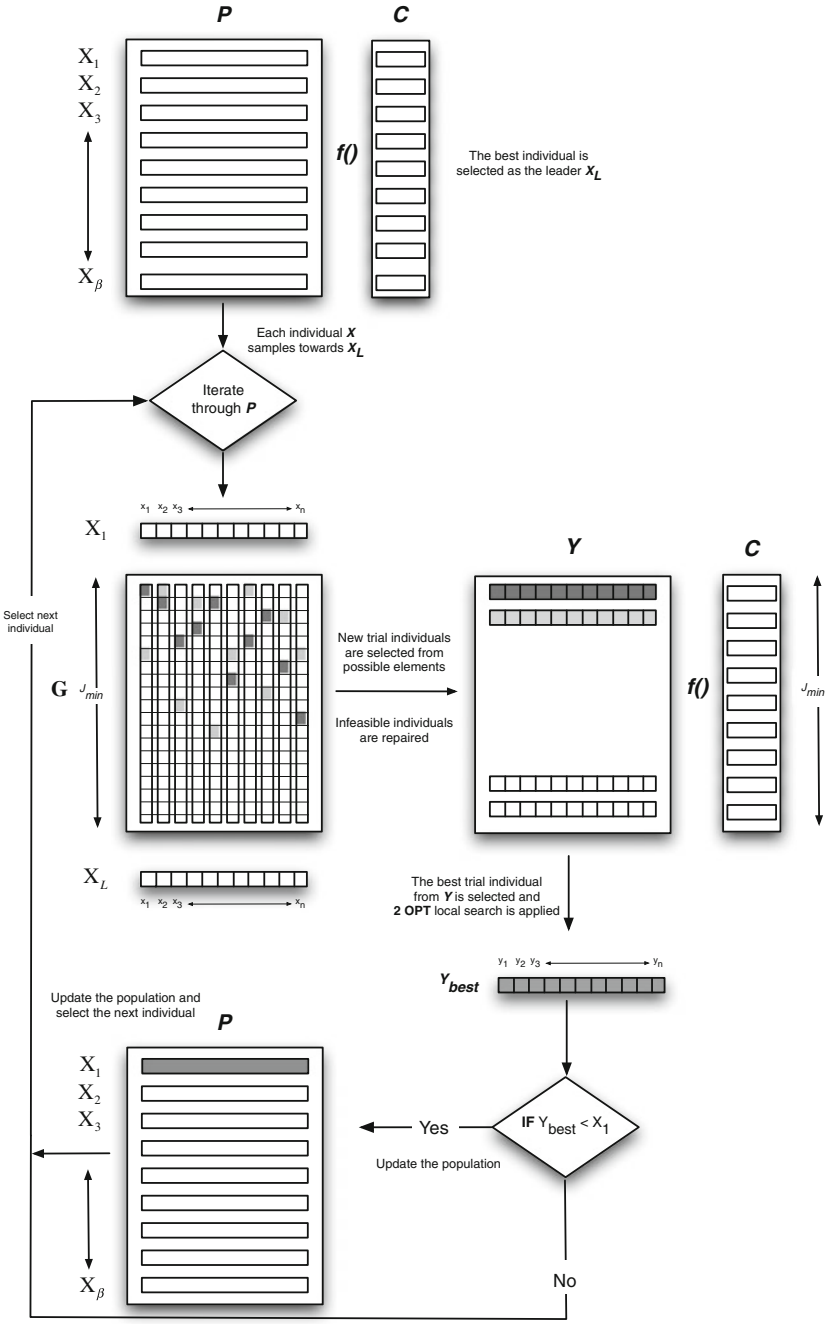

Fig. 8 DSOMA schematic

11 Conclusion

The discrete Self Organising Migrating Algorithm is described in this chapter. Using specific sampling of individuals in the search space, new individuals are constructed which assists in driving the population towards the leader.

DSOMA has proven successful in solving problem in the combinatorial optimisation domain. As a relatively new algorithm, it has a lot of scope for further development and refinement, especially in individual sampling and parallelisation aspects.

Acknowledgments The following grants are acknowledged for the financial support provided for this research: Grant Agency of the Czech Republic—GACR P103/15/06700S, VSB SGS grants of SP2015/141 and SP2015/142, research project NPU I No. MSMT-7778/2014 by the Ministry of Education of the Czech Republic, European Regional Development Fund under the Project CEBIA-Tech No. CZ.1.05/2.1.00/03.0089 and partially by the Internal Grant Agency of Tomas Bata University under the project No. IGA/FAI/2015/057.

References

1. Askarzadeh, A.: A discrete chaotic harmony search-based simulated annealing algorithm for optimum design of pv/wind hybrid system. Sol. Energy **97**(0), 93–101 (2013)
2. Davendra, D., Zelinka, I.: Optimization of quadratic assignment problem using self-organising migrating algorithm. Comput. Inform. **28**, 169–180 (2009)
3. Davendra, D., Zelinka, I., Bialic-Davendra, M., Senkerik, R., Jasek, R.: Discrete self-organising migrating algorithm for flow-shop scheduling with no-wait makespan. Math. Comput. Modell. **57**(12), 100–110 (2013) (Mathematical and Computer Modelling in Power Control and Optimization)
4. Drezne, Z.: A new genetic algorithm for the quadratic assignment problem. INFORMS J. Comput. **115**, 320–330 (2003)
5. Gambardella, L., Taillard, E., Dorigo, M.: Ant colonies for the quadratic assignment problem. Int. J. Oper. Res. **50**, 167–176 (1999)
6. Gao, K., Suganthan, P., Pan, Q., Chua, T., Cai, T., Chong, C.: Pareto-based grouping discrete harmony search algorithm for multi-objective flexible job shop scheduling. Inf. Sci. **289**, 76–90 (2014)
7. Institute, C.M.: Millennium problems (2015). http://www.claymath.org/millennium-problems
8. Lin, F., Kao, C.: Hsu: applying the genetic approach to simulated annealing in solving np-hard problems. IEEE Trans. Syst. Man Cybern. B Cybern. **23**, 1752–1767 (1993)
9. Onwubolu, G., Davendra, D.: Scheduling flow shops using differential evolution algorithm. Euro. J. Oper. Res. **171**, 674–679 (2006)
10. Onwubolu, G., Davendra, D.: Differential evolution: a handbook for global permutation-based combinatorial optimization. Springer, Germany (2009)
11. Pan, Q., Tasgetiren, M., Liang, Y.: A discrete particle swarm optimization algorithm for the no-wait flowshop scheduling problem. Comput. Oper. Res. **35**(9), 2807–2839 (2008)
12. Pan, Q.K., Wang, L., Li, J.Q., Duan, J.H.: A novel discrete artificial bee colony algorithm for the hybrid flowshop scheduling problem with makespan minimisation. Omega **45**, 42–56 (2014)
13. Reeves, C.: A genetic algorithm for flowshop sequencing. Comput. Oper. Res. **22**, 5–13 (1995)

14. Taillard, E.: Robust taboo search for the quadratic assignment problem. Parallel Comput. **17**, 443–455 (1991)
15. Tasgetiren, M., Sevkli, M., Liang, Y.C., Gencyilmaz, G.: Particle swamp optimization algorithm for permutative flowshops sequencing problems. In: 4th International Workshops on Ant Algorithms and Swamp Intelligence, pp. 389–390. Brussel, Belgium (2004)
16. Tasgetiren, M.F., Pan, Q.K., Suganthan, P., Chen, A.H.L.: A discrete artificial bee colony algorithm for the total flowtime minimization in permutation flow shops. Inf. Sci. **181**(16), 3459–3475 (2011)

http://www.springer.com/978-3-319-28159-9

Self-Organizing Migrating Algorithm
Methodology and Implementation
Davendra, D.; Zelinka, I. (Eds.)
2016, XVIII, 289 p. 128 illus., 87 illus. in color.,
Hardcover
ISBN: 978-3-319-28159-9