Int J Adv Manuf Technol (2007) 34:1016–1029
DOI 10.1007/s00170-006-0655-7

ORIGINAL ARTICLE

Balancing large assembly lines by a new heuristic based on differential evolution method

Andreas C. Nearchou

Received: 19 February 2006 / Accepted: 8 May 2006 / Published online: 8 November 2006
© Springer-Verlag London Limited 2006

Abstract Evolutionary algorithms such as genetic algorithms have been applied on a variety of complex combinatorial optimization problems (COPs) with high success. However, in relation to other classes of COPs, there is little reported experimental work concerning the application of these heuristics on large size assembly line balancing problems (ALBPs). Moreover, very few works in the literature report comparative results on public benchmark instances of ALBPs for which upper bounds on the optimal objective function value exist. This paper considers the simple ALBP of type 2 (SALBP-2), which consists of optimally partitioning the tasks' operations in an assembly line among the workstations with objective the minimization of the cycle time of the line. SALBP-2 is known to be intractable, and therefore the right way to proceed is through the use of heuristic techniques. To that purpose, a novel approach based on the differential evolution method has been developed and tested over public available benchmarks ALBPs. These benchmarks include test instances for several precedence graphs (representing the assembly restrictions) with up to 297 tasks. Extended comparisons with other previously published evolutionary computation methods showed a superior performance for the proposed approach.

Keywords Assembly line balancing · Differential evolution · Evolutionary algorithms · Manufacturing optimization

A. C. Nearchou (✉)
Department of Business Administration, University of Patras, 26500 Patras, Greece
e-mail: nearchou@upatras.gr

1 Introduction

Assembly lines are a traditional widely used type of production systems for mass and large-scale production. They consist of a number of workstations arranged along an automated material handling system such as a conveyor belt. Work pieces are moved along the line from station to station, while each station performs a number of repeated operations necessary to manufacture a desired final product. Each sub-product unit remains at each station for a fixed work rate called the cycle time of the line. The assembly line balancing problem (ALBP) is a decision problem arising when an assembly line has to be configured or redesigned. The problem consists of determining the optimal partitioning (balancing) of the assembly work among the workstations while optimizing one or more objectives without violating the restrictions imposed on the line.

The rapid qualitative and quantitative changes in market demands cause manufacturers to seek the best possible methods for managing their assembly lines so as to produce more sophisticated and more competitive products. To respond to these diverse market needs, modern assembly lines must be highly automated [1] and easily re-configurable [2]. Mixed-model and multi-model lines [3, 4] are today the industrial answer to the growing trend for product variability and shorter production cycles. In mixed-model lines several versions of the basic product are manufactured in the same line; while in multi-model lines several (similar) products are manufactured on one or several lines. BMW (the German car manufacturer) constitutes a characteristic example of a modern assembly line system offering a catalogue of optional features, which theoretically results in 10^{32} different models [5]. The decisions taken to solve ALBP in modern flow-line production systems not only affect the final cost of the products, but also affect the variety of the products manufactured, their final quality, as well as, the time-to-market response. The latter index is strongly depended on the production cycle of the assembly line and constitutes one of the most interesting performance indices in ALBP.

ALBP is classified into the simple ALBP (SALBP), and the generalized ALBP [2, 4, 6]. The latter contains characteristics not contained in the SALBP such as cost objectives, paralleling of stations, mixed-model production, etc. Two formulation types are commonly used with SALBP: SALBP-1 which attempts to minimize the number of stations for a given fixed cycle time, and SALBP-2 which attempts to minimize the cycle time of the line for a given number of stations. The former type is used when a new assembly line has to be implemented and installed, while the latter type is used in an existing assembly line when changes in the production process and manufacturing requirements occur. Any variant of the basic SALBP is of combinatorial nature and belongs to the NP-hard class of combinatorial optimization problems (COPs) [2]. Therefore, exact algorithms can hardly be designed to solve large sizes of any variant of SALBP and consequently the right way to proceed is through the use of heuristics techniques.

This work deals with the deterministic SALBP-2. Traditionally [2], SALBP-2 is addressed through an iterated procedure that solves a corresponding SALBP with a known number of stations and a cycle time value being progressively decreased until reaching a near-optimum value within a specific permitted range. Hence, the way the trial cycle times are examined plays a significant role on the quality of the generated final solutions. SALOME-2 [7] is today the most effective branch and bound method that solves directly SALBP-2.

Recently, some researchers turned their attention to the use of meta-heuristics for the solution of SALBP-2. The most notable of this group of algorithms are evolutionary computation (EC) methods such as genetic and evolutionary algorithms [8], simulated annealing [9] and tabu-search [10, 11]. Genetic algorithms (GAs) for SALBP-2 were presented by [12–14]. Anderson and Ferris [12] developed a GA with an objective function that sums up the maximal station time and put a penalty term for precedence violations. Watanabe et al. [14] showed experimentally that a GA can obtain in reasonably computing time quasi-optimum solutions for large size ALBPs that cannot be solved by ordinary methods. Kim et al. [13] addressed the multi-objective SALBP (including SALBP-2) with additional objectives such as the maximization of workload smoothness and the maximization of work relatedness. Heinrici [15] developed a tabu-search procedure for the solution of SALBP-2, and compared its performance to that of a simulated annealing algorithm. The latter approach directs the search to 'better' solutions using suitable mechanisms of shifts and swaps moves in the search space. Scholl and Voß [16] and Chiang [17] have also developed efficient tabu-search procedures for SALBP-2. Important recent reviews about SALBP can be found in [18–20].

There is a gap in the literature concerning the application of meta-heuristics on public large size SALBP-2 benchmark instances for which upper bounds on the optimal objective function value exist. This paper investigates the use of a new EC method based on the differential evolution algorithm (DEA) for solving SALBP-2. The performance of the proposed method is examined over benchmarks from the open literature and compared to that of other previously published GAs solutions. The benchmarks used are public available at http://www.assembly-line-balancing.de/ and include 302 instances for various ALBPs containing 17 precedence graphs with tasks ranging from 29 to 297.

The rest of the paper is organized as follows: Section 2 states formally SALBP-2. Section 3 describes the standard DEA for function optimization over continuous spaces. Section 4 introduces and analyzes the way DEA can be applied on ALBPs, while Section 5 presents comparative experimental results over known benchmark instances for SALBP-2. Finally, Section 6 summarizes the contribution of the paper and states some directions for future work.

2 Formulation of the problem

SALBP-2 can be formally stated as follows: a set $WS=\{1,\ldots,m\}$ of m workstations are arranged along an assembly line. Manufacturing a single product on the assembly line requires the partitioning of the total assembly work into a set $V=\{1,\ldots n\}$ of n elementary operations called tasks. Each task j $(j\in V)$ is performed on exactly one workstation and requires a deterministic processing time t_j (j=1,...,n). The tasks are partially ordered by precedence relations defining a precedence graph $G=(V,E)$. G is a directed acyclic graph (DAG) with V the set of nodes denoting the tasks in G, and E the set of edges representing the precedence constraints among the tasks. An edge (i, j) denotes that task i must be finished before task j can be started. In this case task i is a direct predecessor of task j. While the edges (q, r) (r, p) (with $q, r, p \in$ V) denote that task q is a direct predecessor of task r and indirect predecessor of task p. The assembly line is associated with a cycle time c denoting the maximum (or average) processing time available for each work cycle. Each workstation can complete its assigned tasks within the specified cycle time. Therefore, given WS, t_j (j=1,...,n) and G, the objective with SALBP-2 is to find a feasible line balance (i.e., an assignment of the n tasks to the m workstations not violating the precedence constraints) that minimizes c. This objective is usually achieved by maximizing the efficiency E of the line given by the relation:

$$E = t_{sum}/m.c \tag{1}$$

where, $t_{sum} = \sum_{j=1}^{n} t_j$ is the sum of the processing times of all the tasks.

Figure 1(a) illustrates an example of a precedence graph for an 11-tasks SALBP having processing times between 2 and 45 time units and t_{sum}=185. The numbers inside the nodes of the graph correspond to the task labels, and those outside the nodes to the processing times. Hence, task 1 has a processing time equal to 4 time units, task 2 a processing time equal to 38 time units, etc. The precedence constraints for example, for task 6 defines that, this task must proceed after the completion of task 4 (direct predecessors), and tasks 1 and 2 (indirect predecessors). While, task 6 must be completed before its direct (or indirect) successors, which are tasks 8, 10 and 11.

Assuming 4 workstations in the line, the optimum line balancing solution for SALBP-2 is (3, 2, 5, 1, 4, 7, 9, 6, 8, 10, 11). Meaning that, the order in which the tasks are to be executed on the assembly line is task 3, followed by task 2, followed by task 5, and so on. This solution corresponds to a minimal cycle time of c=48 time units. The sequence of the workstation loads is thereby (see Fig. 1(b)), WS_1={3}, WS_2={2,5}, WS_3={1,4,7,9,6,8}, WS_4={10,11}.

3 Evolutionary computation (EC) based heuristics

3.1 Overview

EC is a modern class of meta-heuristics inspired by biological evolution. Representative paradigms of EC heuristics are GAs, evolutionary strategies, evolutionary programming, and genetic programming. These techniques have been found to perform better than the classical heuristics or gradient based methods, especially when addressing the problem of optimizing multi-modal, non-differentiable, or discontinuous functions. The state-of-the-art in modern heuristics is to use the term *evolutionary algorithm* (EA) to describe any algorithm that uses population-based random variation and selection [8]. Independently of the form of the optimization problem, any EA undergoes the following mechanism:

(a) Create (usually randomly) a population of individuals that represent potential solutions to the physical problem.
(b) Evaluate the quality of each individual in the population.
(c) Promote individuals of higher quality by introducing selective pressure on the entire population.
(d) Generate new individuals by applying variation operators on the population.

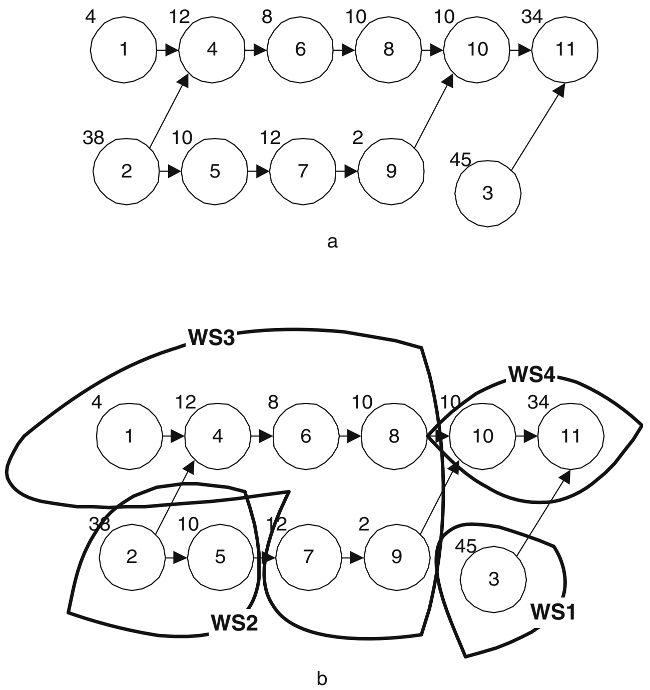


Fig. 1 An 11-tasks ALBP: (**a**) the precedence graph. (**b**) A feasible line balance solution for SALBP-2 with m=4 and c=48

(e) Repeat steps (b)-(d) several times until the satisfaction of a suitable criterion.

As there exists a large number of possible ways to implement the above five steps involved in an EA, the major consideration is to select those heuristics achieving good performance for the problem under consideration. This work addresses SALBP-2 by DEA; a new EC model proposed for optimization over continuous spaces. There are at least three basic features that make DEA attractive to be used for solving complex optimization problems in contrary to the use of other heuristics: (i) It manipulates pure floating-point numbers without any other extra processing, thus, utilizing computer resources efficiently. (ii) It is easy to use and tune, since it requires the settings of very few control parameters. (iii) It is much easier to be implemented than other EAs.

3.2 Differential evolution algorithm (DEA): the basic model

DEA is an EA introduced by Storn and Price [21] for optimization over continuous spaces. Since its invention, DEA has been applied with high success on many numerical optimization problems outperforming other more popular population heuristics including GAs [22, 23]. Recently, some researchers extended with success the application of DEA to complex COPs with discrete decision variables, such as, the machine layout problem [24] and the flow-shop scheduling problem [25].

DEA utilizes Np, D-dimensional parameter vectors $x_{i,k}$, i=1,2,...,Np, as a population to search the feasible region Ω of the optimization problem. The index k denotes the iteration number of the algorithm. The initial population (k=0),

$$S = \{x_{1,0}, x_{2,0}, \ldots, x_{Np,0}\}, \tag{2}$$

is taken to be uniformly distributed in Ω. At each iteration all vectors in S are targeted for replacement. Therefore, Np competitions are held to determine the members of S for the next iterations. This is achieved by using mutation, crossover and acceptance operators. In the mutation phase, for each target vector $x_{i,k}$, i=1,2,..., Np, a mutant vector $\widehat{x}_{i,k}$ is obtained by

$$\widehat{x}_{i,k} = x_{\alpha,k} + F\left(x_{\beta,k} - x_{\gamma,k}\right) \tag{3}$$

where α, β, $\gamma \in \{1,2,...,Np\}$ are mutually distinct random indices and are also different from the current target index i. $x_{\alpha,k}$ is known as the base vector and F>0 is a scaling parameter. The crossover operator is then applied to obtain the trial vector $y_{i,k}$ from $\widehat{x}_{i,k}$ and $x_{i,k}$ using

$$y^j_{i,k} = \begin{cases} \widehat{x}^j_{i,k} \text{ if } R^j \leq C_R \text{ or } j = I_i \\ x^j_{i,k} \text{ if } R^j > C_R \text{ and } j \neq I_i \end{cases}, \tag{4}$$

where I_i is a randomly chosen integer in the set I, i.e., $I_i \in I = \{1,2,...,D\}$; the superscript j represents the j-th component of respective vectors; $R^j \in (0,1)$, drawn randomly for each j. The ultimate aim of the crossover rule is to obtain the trial vector $y_{i,k}$ with components coming from the components of the target vector $x_{i,k}$ and the mutated vector $\widehat{x}_{i,k}$. This is ensured, by introducing C_R (the crossover rate) and the set I. Notice that for C_R=1 the trial vector $y_{i,k}$ is the replica of the mutated vector $\widehat{x}_{i,k}$. The targeting process (mutation and crossover) continues until all members of S are considered. After all Np trial vectors $y_{i,k}$ have been generated, acceptance is applied. In the acceptance phase, the function value at the trial vector, $f(y_{i,k})$, is compared to $f(x_{i,k})$, the value at the target vector and the target vector is updated using

$$x_{i,k+1} = \begin{cases} y_{i,k} \text{ if } f(y_{i,k}) < f(x_{i,k}) \\ x_{i,k} \text{ otherwise} \end{cases} \tag{5}$$

Mutation, crossover and acceptance continue until some stopping conditions are met.

The mechanism described above is one variant of the basic DEA known as scheme DE1 [21]. There are also some other variants, differ in the way they create the mutant vector (Eq. (3)). One such highly effective variation scheme is given by

$$\widehat{x}_{i,k} = \xi \cdot x_{best,\ k} + (1-\xi) \cdot x_{\varepsilon,k} + F\left(x_{\alpha,k} + x_{\beta,k} - x_{\gamma,k} - x_{\delta,k}\right) \tag{6}$$

with α, β, γ, δ, ε different integers taken over $\{1,2,...,Np\}$, and $\xi \in [0,1]$ a coefficient for the convex combination between the best element $x_{best,k}$ of S, and a randomly selected element $X_{\varepsilon,k}$.

4 The proposed DEA-based approach for the solution of SALBP-2

Tuning a DEA for a particular domain needs the specification of the following characteristics:

– A representation mechanism, i.e., a way of encoding ALB solutions to floating-point vectors.
– An evaluation mechanism, i.e., a way of evaluating the quality of each vector.
– A way of initializing the population of vectors.
– The application of mutation and crossover operators on the population in order to generate new ‘better’ populations.
– Values to the three control parameters: population size, crossover rate, and mutation rate.

The first two characteristics are analyzed below in detail. The remaining characteristics are the same as in the standard DEA. A pseudo-code of the proposed DEA for the solution of SALBP-2 is given in the Appendix of this work.

4.1 The representation mechanism

Two different schemes of string representations applicable to ALBPs are mainly reported in the related literature [20]: the station-oriented and the task-oriented representation. Both of them assume strings of integers and a string length equal to the number of tasks to be proceeded in the assembly line. If the *i*-th position of the string has the value *j*, then, using the station-oriented representation, task *i* is assigned to workstation *j*. While, using the task-oriented representation, task *j* in location *i* of the string will be assigned to a workstation before the task in location (*i*+1) of the string. The tasks are allocated into stations starting from the first (*k*=1) and considering the other stations successively. When a station is loaded maximally, it is closed, and a new station (*k*+1) is opened. A solution is feasible when the generated sequence of the tasks in the line does not break the specified precedence constraints. After experimentation with both schemes we found task-oriented representation superior and therefore it was decided to adopt this scheme within DEA.

DEA works with floating-point vectors hence an appropriate mapping is needed from the 'genotypic' state-level (the vectors) to the 'phenotypic' level (the actual ALB solutions). To that purpose, two different encoding schemes namely *random-keys* and *priority-based*, respectively, have been implemented and used with the proposed DEA. Random-keys introduced by Bean [26] in the context of a real-coded GA to address a wide range of sequencing and scheduling problems. Priority-based encoding was used by Gen and Cheng [27] within an order-based GA for the solution of the resource-constrained project scheduling problem.

4.1.1 Encoding ALB solutions with the random-keys method

Let's illustrate how random-keys work through an example concerning the 11-task ALBP of Fig. 1. Any candidate solution (generated by DEA) to this problem is a real-coded vector containing 11 floating-point numbers. Assume the vector

$$\Psi = \begin{pmatrix} 0.69,\ 0.47,\ 0.88,\ 0.38,\ 0.12,\ 0.23, \\ 0.04,\ 0.16,\ 0.33,\ 0.24,\ 0.10 \end{pmatrix}$$

the position and the value of the floating numbers (called keys) in ψ are critical for the interpretation of the sequence into an actual ALB solution. Hence, according to random-keys method we have to identify: first the position of the lowest value in ψ (which is 0.04 at position 7), then, the second lowest value (which is 0.10 at position 11), the third lowest value (0.12 at position 5), and so on. Repeating this ordering for all the keys in ψ we finally get the string

$(7,\ 11,\ 5,\ 8,\ 6,\ 10,\ 9,\ 4,\ 2,\ 1,\ 3)$

For our ALBP this sequence can now be interpreted as: start executing task 7, followed by task 11, followed by task 5, etc. Note that, this solution is clearly illegal for ALBP of Fig. 1 since it violates the specific precedence constraints. When this is the case, a simple repairing procedure is applied on the task sequence and fixes it so that to finally become feasible.

4.1.2 Repairing infeasible ALB solutions generated by random-keys encoding

Precedence relations among tasks are usually presented in a matrix form **M**={M_{ij}}; with M_{ij}=1 if task *i* must be finished before task *j*, otherwise M_{ij}=0. **M** is critical since it is used for checking the feasibility of a solution. In this study, **M** was implemented using an old but efficient algorithm [28] which constructs the transitive closure of a DAG *G*=(*V,E*) using the relation $M_{jk} = M_{jk} \oplus M_{ik}$ (with *i*, *j*, *k* ∈*V*). The behavior of operator ⊕ is given by, the table

⊕	0	1
0	0	1
1	1	1

Therefore, the complete mechanism for constructing the precedence matrix *M* is as follows:

```
Procedure Build_precedence_matrix
begin
Step 1  for i=1 to n do
    for j=1 to n do
     if task i must be finished immediately before task j
     then $M_{ij}$=1 else $M_{ij}$=0
Step 2  for i=1 to n do
    for j =1 to n do
     if $M_{ij}$=1 then
     for k=1 to n do
     $M_{jk} = M_{jk} \oplus M_{ik}$
Step 3  for j=1 to n do
     count=0
    for i=1 to n do
     if $M_{ij}$=1 then count=count+1
     $M_{(n+1)j}$=count

     Return M
end
```

Step 1 builds *M* by putting 1's whenever task *i* must be finished immediately before task *j* and zeros otherwise. In step 2, *M* is updated according to Warshall's algorithm.

That is, whenever a task *j* is indirect predecessor of a task *k* the algorithm sets M_{jk}=1. Step 3 counts for each task *j* the total number of its predecessors, and saves this number in the corresponding column of the last line (*n*+1) of *M*. This information is used in the repairing procedure below.

```
Procedure Repair_infeasible_solution (x)
begin
    M'=M // work with a copy of M //
    j=0 // index for array PS //
    i=1 // iterations' counter //
    repeat
    a=x[i]
    if $M'_{n+1,a}$=0 and task a does not be placed into PS then
    j=j+1, PS [j]=a
    Update M': set $M'_{ak}$=0 and decrease $M'_{(n+1)k}$ by one ∀ k∈[1,n]
    end if
    i=i+1
    if i>n then i=1
    until j=n
    Return PS // the repaired, feasible ALB solution corresponding to x //
end
```

The above procedure corrects an infeasible solution *x* by checking each one of its tasks for feasibility. Each feasible task-*a*, is inserted in the next available position of a new string (initially empty) called partial schedule (*PS*). While, all the outgoing edges from task-*a* to its immediate successors are deleted by updating *M* accordingly. *PS* will finally hold the feasible version of *x*. Table 1 displays the successive passes of the topological sort obtained after applying the above repairing procedure on the infeasible ALB sequence (7, 11, 5, 8, 6, 10, 9, 4, 2, 1, 3) generated by random-keys. A new pass is encountered whenever the repeat/until loop restarts scanning *x* from its first location (*i*=1). The repaired string is given in the last column of Table 1.

4.1.3 Encoding ALB solutions with the priority-based method

Priority-based encoding consists of generating a topological sort of a DAG *G=(V,E)* from a specific *n*-dimensional floating-point vector ψ (*n* denotes the number of the tasks in ALBP). Each vector's component *j* (*j*=1,...,*n*) represents the relative priority of task *j* (*j*∈V). The topological sort is therefore a ranking of all the tasks according to their priorities in an appropriate order to meet the precedence constraints. This mechanism is implemented using the following procedure:

```
Procedure Priority_based_encoding
begin
    Set V'=∅ // with V'⊆V //
    Repeat
    For all j∈V do
    if j has no predecessors then V'=V'∪{j}, i.e., insert j into the set V'.
    Determine the i-th component of ψ with the maximum value for all i∈V'.
    Insert task i into the next available position in the partial schedule (PS).
    V'=V' \ {i}, i.e., remove task i from V'.
    Until PS has been completed
    Return PS
end
```

The above procedure is based on two main repeated steps: (a) construct the set V′ of the candidate tasks to be placed in the next available position of *PS*. (b) Select the highest priority task in V′ as indicated by the values of the corresponding components in ψ. Initially, V′ is empty and iteratively is being updated so as to contain those unscheduled tasks with no predecessors. As it is obvious, all the solutions generated by the specific mechanism are feasible and thus, no repairing mechanism is applied on *PS*.

Table 1 Application of the repairing procedure on an infeasible ALB solution

Pass	Iteration	Infeasible string	PS (repaired string)
1	9	(7, 11, 5, 8, 6, 10, 9, 4, ***2***,1, 3)	(2)
1	10	(7, 11, 5, 8, 6, 10, 9, 4, 2, ***1***, 3)	(2, 1)
1	11	(7, 11, 5, 8, 6, 10, 9, 4, 2, 1, ***3***)	(2, 1, 3)
2	3	(7, 11, ***5***, 8, 6, 10, 9, 4, 2, 1, 3)	(2, 1, 3, 5)
2	8	(7, 11, 5, 8, 6, 10, 9, ***4***, 2, 1, 3)	(2, 1, 3, 5, 4)
3	1	(***7***, 11, 5, 8, 6, 10, 9, 4, 2, 1, 3)	(2, 1, 3, 5, 4, 7)
3	5	(7, 11, 5, 8, ***6***, 10, 9, 4, 2, 1, 3)	(2, 1, 3, 5, 4, 7, 6)
3	7	(7, 11, 5, 8, 6, 10, ***9***, 4, 2, 1, 3)	(2, 1, 3, 5, 4, 7, 6, 9)
4	4	(7, 11, 5, ***8***, 6, 10, 9, 4, 2, 1, 3)	(2, 1, 3, 5, 4, 7, 6, 9, 8)
4	6	(7, 11, 5, 8, 6, ***10***, 9, 4, 2, 1, 3)	(2, 1, 3, 5, 4, 7, 6, 9, 8, 10)
5	2	(7, ***11***, 5, 8, 6, 10, 9, 4, 2, 1, 3)	(2, 1, 3, 5, 4, 7, 6, 9, 8, 10, 11)

Let us see how priority-based encoding works on vector ψ mentioned in sub-section 4.1.1. Again we assume the ALBP with the DAG given in Fig. 1. According to the above procedure, tasks 1, 2, and 3 compete for the first position of array *PS* since they are the only tasks with no predecessors. The priorities for these tasks are 0.69, 0.47, 0.88, respectively. Therefore, *PS*[1]=3 since task 3 has the highest priority. Then, the two tasks 1 and 2 are candidate for the second location of *PS*. Task 1 has the highest priority (=0.69) and thus *PS*[2]=1. Next, task 2 takes the 3rd location of *PS* since is the only task with no predecessors. Consequently, the new candidate tasks for the 4th position of *PS* are the tasks 4 and 5 and thus, *PS*[4] =4. Finally, the ALB solution corresponding to vector ψ will be (3, 1, 2, 4, 6, 8, 5, 7, 9, 10, 11). The interested reader can see in Fig. 2 the detailed step-by-step process for constructing the specific ALB solution. In particular, one can see from Fig. 2 the partial topological sort, the cut (dark long dashed lines) and the eligible nodes, the contents of the partial schedule solution *PS*, as well as, the set V' with the candidate tasks to compete for the next empty position in *PS*.

4.1.4 How a DEA's phenotype is decoded into a SALBP-2 solution?

Once a specific floating-point vector (i.e., a DEA's genotype) is encoded into a feasible ALB solution (DEA's phenotype) then, an appropriate decoding scheme is needed to map this phenotype to an actual solution for SALBP-2. In other words, a method is needed to assign the tasks in the generated task-sequence into the workstations. After experimented with some well known from the literature decoding schemes such as the lower bound and the upper bound search methods [2], we finally decided to adopt a scheme previously proposed by [13]; since this scheme was found in preliminary experiments to be superior. The main idea is to face SALBP-2 through an iterated procedure that solves the corresponding SALBP-1 with a cycle time value being progressively decreased until reaching a near-optimum value within a specific permitted range. This decoding scheme is as follows:

Step 1 Set c initially equal to the theoretical minimum cycle time, i.e., $c=t_{sum}/m$.

Step 2 Assign as many as possible tasks into the first m-1 workstations. Assign all the remaining tasks to the last workstation, m.

Step 3 Calculate the work load W_z for each workstation z (z=1,2,...,m), and the potential workload PW_z (z=1,2,...,m-1) as follows: W_z=the station time St_z (z=1,2,...,m). $PW_z=St_z$ + the processing time of the first task assigned to the (z+1)st workstation (z=1,2,...,m-1).

Step 4 Set $c_W = max\ \{W_1, W_2, ..., W_m\}$ and $c = min\ \{PW_1, PW_2, ..., PW_{m-1}\}$

Step 5 IF ($c_W > c$) THEN GO TO Step 2

Step 6 RETURN c_W as the minimum cycle time and STOP

4.2 Evaluation mechanism

This mechanism corresponds to the computation of the cost function for each phenotype (i.e., for each feasible SALBP-2 solution) of the current population. The objective with SALBP-2 is to minimize c given a fixed m hence the following cost function was adopted:

$$Cost = 1/E \tag{7}$$

where, E is the line efficiency estimated by Eq. (1). Equation (7) aims to minimize cost-function by maximizing the line efficiency E and consequently forcing c to a minimum value.

5 Computational results and discussion

5.1 Experimental setup

The performance of DEA was examined over a large set of benchmark ALB instances taken from the open literature. The precedence graphs, as well as, the existing near optimal solutions for these benchmarks, are available at http://www.assembly-line-balancing.de/. The benchmarks include test instances organized into two data sets. Data set 1 contains 128 instances for 9 precedence graphs with tasks varying from 29 to 111, while data set 2 contains 174 instances concerning 8 precedence graphs with tasks varying from 53 to 297.

Two versions of the proposed DEA were implemented each one corresponding to a distinct encoding scheme either random-keys, or priority-based. We will refer to these heuristics as *DEA_rks* and *DEA_prb*, respectively. For both DEAs the mutant vectors were generated using Eq.(6) with ξ=0. This scheme was experimentally found superior to others more frequently referred to in the literature DE schemes such as DE1 and DE2 [21]. The performance of DEAs was compared against to that of two other EAs previously proposed for the solution of ALBPs; in particular, against the GAs proposed by Kim et al. [13] and Goncalves and Almeida [29], respectively. The former GA utilizes permutation strings to solve SALBP with various objectives. We will refer to this GA as *pGA* (stands for permutation GA). The latter GA is a real-coded GA hybridized by a special parents' selection scheme for creating the population of every new generation, as well as, a suitable local search procedure for the solution of SALBP-1. In order to apply this GA on SALBP-2 we

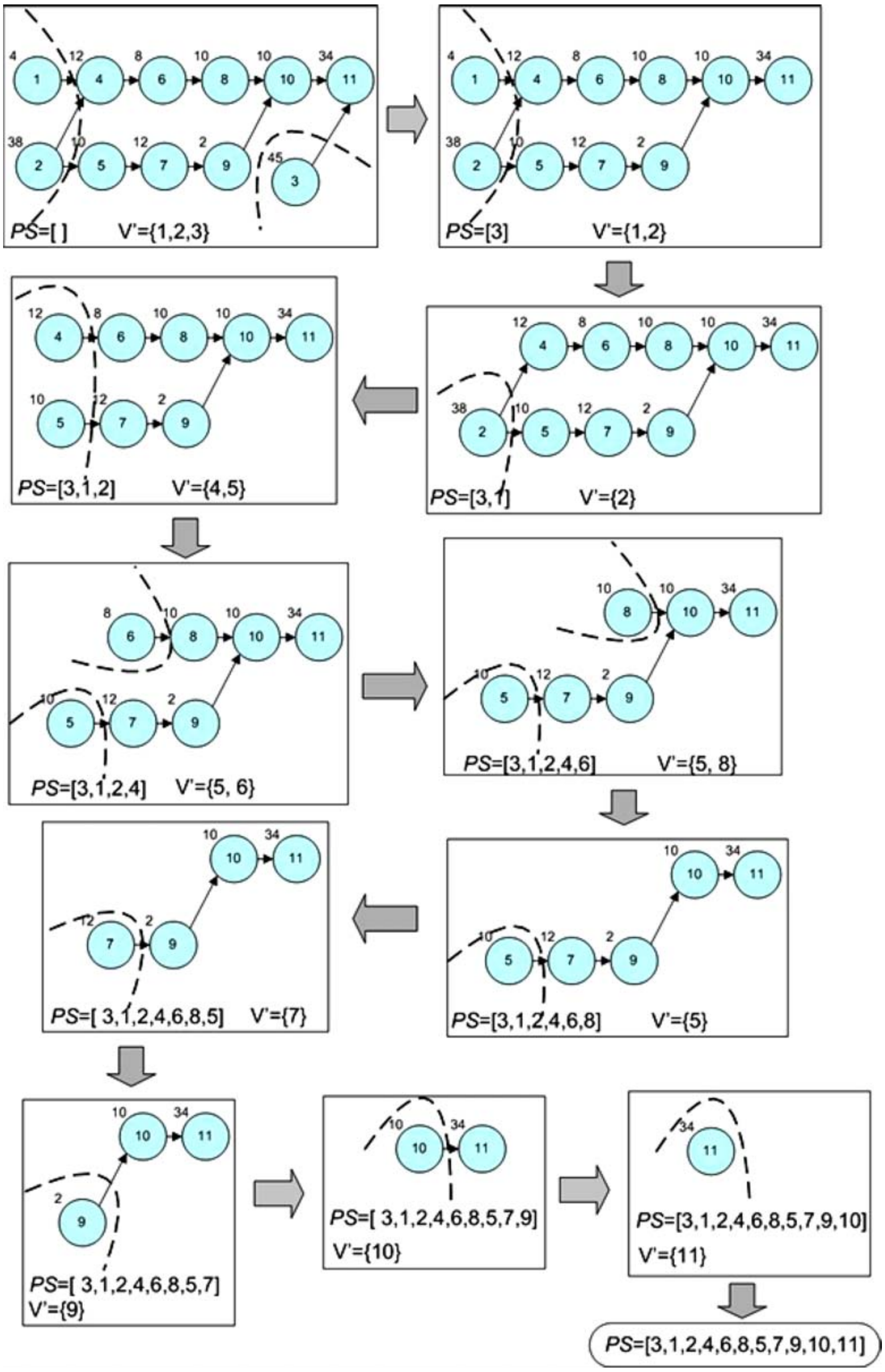

PS=[ ] V'={1,2,3}
PS=[3] V'={1,2}
PS=[3,1] V'={2}
PS=[3,1,2] V'={4,5}
PS=[3,1,2,4] V'={5, 6}
PS=[3,1,2,4,6] V'={5, 8}
PS=[3,1,2,4,6,8] V'={5}
PS=[3,1,2,4,6,8,5] V'={7}
PS=[3,1,2,4,6,8,5,7] V'={9}
PS=[3,1,2,4,6,8,5,7,9] V'={10}
PS=[3,1,2,4,6,8,5,7,9,10] V'={11}
PS=[3,1,2,4,6,8,5,7,9,10,11]

Fig. 2 Step-by-step construction of the ALB solution associated to chromosome ψ using the priority based encoding method

replaced this local search procedure with the decoding scheme presented in sub-section 4.1.4. Random-keys encoding was used to map the floating-point chromosomes to ALB solutions. Consequently, we will refer to this heuristic as *rGA_rks*. Moreover, it was decided to implement an additional version of this GA using priority-based encoding scheme. We will call this heuristic *rGA_prb*. All the algorithms were written in Pascal programming language and run on a Pentium IV 1.7 GHz PC.

5.2 Settings of the control parameters

Much investigation on the selection of the appropriate settings of the control parameters for DEAs was undertaken in preliminary tests. As with any EA, the determination of the correct settings for the control parameters of DEA is a very difficult task. The variety of the control parameters ($Np \geq 4$, $C_R \in [0,1]$, $F \in (0,2)$) included in its components, and. the many possible choices make the determination of the 'perfect' settings almost impossible. In this study 30 trials were performed consisting of three levels of $Np \in (50,100, n)$, with n being the number of the tasks in the corresponding benchmark problem. Five levels of $C_R \in (0.1, 0.3, 0.5, 0.7, 0.9)$, and two levels of F, either being constant and equal to $F=0.9$; or estimated by a rule proposed by Zaharie [30]. This rule says that the values of the control parameters satisfying the equation $2F^2 - 2/Np + CR/Np = 0$ can be considered to be critical for the convergence ability of a DEA. After much experimental effort with the above combination of settings, and taking into account both the quality of the generated solutions, as well as, the processing time spent, the following values were finally adopted: $Np=n$, $C_R=0.9$, $F=0.9$.

All the genetic operators and the settings for the control parameters for the three GAs, (*pGA*, *rGA_rks*, *rGA_prb*) were kept the same as they appeared in the works of their authors. Specifically, for the case of *pGA* we used: tournament policy for parents' selection (tournament size equal to 2), partially mapped crossover (PMX) as the crossover operator, and reciprocal exchange as the mutation operator. Crossover and mutation rates were 0.3 and 0.5, respectively. For *rGA_rks* and *rGA_prb* we used a reproduction scheme which builds the population of the next generation as follows: the 15% top (best) individuals of the current generation are copied to the next (elitist strategy), while the remaining members are created by applying a parameterized uniform crossover operator on the entire population with a rate equal to 0.7. Instead of using a traditional mutation operator, the 20% worst members of the new population are replaced by new randomly generated chromosomes. For fair comparisons, the GAs' population size was defined equal to n (i.e., same as in DEAs). Moreover, all the heuristics are left running until one of the following two criteria is satisfied first: either the existing optimum solution is generated, or a maximum number of $10 \times n$ generations has been surpassed. Table 2 summarizes the basic characteristics upon the development methodology used for configuring each one of the heuristics. Remarks concerning benefits and costs in regard to the representation mechanism adopted are reported in the last two columns of the table.

5.3 Results on public benchmarks

Table 3 displays comparative results obtained by the five heuristics over the benchmarks described in sub-section 5.1. The table provides the following information:

– **ARD**=average relative deviation from the optimum solution in percentage. This performance index is estimated by the relation $((c-c^*)/c^*) \times 100$; with c^* the optimal (or the existing best known) cycle time, and c the cycle time of the best solution generated by a specific heuristic.
– **MRD**=the maximum relative deviation from optimality in percentage.

As one can see from Table 3(a), the best results concerning the benchmarks instances of data set 1 have been obtained by the proposed DEAs with that using the random keys encoding being superior. More specifically, *DE_rks* outperformed all the other heuristics achieving solutions of higher quality with a mean relative offset from the existing optimum approx. equal to 1.7%. The second best performance was achieved by *DE_prb* generating ALB solutions with an average relative deviation from the existing optimum approx. equal to 2.1%. Slightly lower performance was achieved by *pGA* generating solutions with an ARD equal to 2.3%. Even poorer performance is reported in the case of the two real-coded GAs, with *rGA_rks* being the worst of all.

Similar performance is reported in Table 3(b) with a synopsis of the results obtained by the five heuristics over the benchmarks of data set 2. *DE_rks* became again the champion among the five heuristics generating solutions with a mean ARD approx. equal to 3.3%. Very near to this performance is that achieved by *DE_prb*, with a slightly larger ARD, approx. equal to 3.4%.

The interested reader can find in Table 4 detailed results concerning the mean actual processing times (in CPU seconds) spent by each heuristic over the benchmarks problems. The first and second columns of Table 4 display the name of the benchmark problem, and its size (i.e., the number of the tasks in the related precedence graph), respectively. Note that, no results are included for *Mukherje* benchmark problem (an ALBP contained in data set 2 with a 94-tasks DAG) due to the poor performance evaluated by all the heuristics over the instances of this problem. None of the five heuristics was found able to generate ALB solutions with acceptable ARD for the instances included in this benchmarks category. Specifically, the best performance over *Mukherje* instances was encountered by *DE_rks*, which

Table 2 Basic configuration characteristics for each heuristic

Method	genotype	phenotype	Encoding scheme	Operators	Advantages	Disadvantages
DE_prb	FPV[↑]	PV	Priority-based	*Mutation*: using Eq. (6) *Crossover*: Eq. (4) *Acceptance*: Eq. (5)	No repairing. Phenotypes always correspond to feasible ALB solutions	Rather slow topological sort
DE_rks	FPV	PV	Random-keys	Same as in DE_prb	Fast encoding scheme	Phenotypes must be repaired to present feasible ALB solutions
pGA	PV[•]	PV	–	*Mutation*: reciprocal exchange (rate 0.5) *Crossover*: PMX (rate 0.3) *Selection*: Binary tournament	No extra mapping between genotypes and phenotypes	Phenotypes need repairing after crossover
rGA_prb	FPV	PV	Priority-based	*Mutation*: replace randomly the 20% worst members of population *Crossover*: uniform (rate 0.7) *Selection*: the 15% best members always survive	No repairing. Phenotypes always correspond to feasible ALB solutions	Rather slow topological sort
rGA_rks	FPV	PV	Random-keys	Same as in rGA_prb	Fast encoding scheme	Phenotypes must be repaired to present feasible ALB solutions

[↑] FPV=Floating-point vector, [•] PV=Permutation Vector

achieved solutions with a mean ARD approx. equal to 13%. All the other heuristics accomplished worse results. It seems that these benchmarks are too hard to be solved by any one of the heuristics under investigation.

With a deep observation of Table 4 one can easily realize that the use of random-keys encoding within either DEAs, or rGAs results in a much faster optimizer than using priority-based encoding. See for example, the speed of convergence of *DE_prb* and *DE_rks*. It is obvious that the latter method is substantially faster than the former especially, for large size ALBPs. Similarly, the GA with random-keys is faster than the GA with priority-based encoding. This observation is much clearer for the benchmarks of data set 2. Furthermore, one can safely conclude that *pGA* is the fastest heuristic with a mean convergence rate of about 2.7 s of CPU-time for the benchmarks of data set 1, and 3.5 s for the benchmarks of data set 2. Near to this performance but lower comes that of *rGA_rks* with a mean convergence rate approx. equal to 3.3 s (for data set 1) and 6.3 s (for data set 2). Well enough convergence speed is also reported for the champion heuristic *DE_rks*. In particular, for benchmark instances with up to 89-tasks, *DE_rks* was found able to converge to a near-optimum solution in less than 2 s of CPU-time in average. While, for instances of larger sizes (≥111 tasks) *DE_rks* spent in average about 29 s of CPU-time until the convergence. Obviously, the largest processing time spent until the convergence is due to *DE_prb*. Although this heuristic achieved the second best performance (see Tables 3 and 5), it needs much more CPU-time until the convergence than the other heuristics.

Table 5 displays analytically the average relative deviation (ARD) from optimum over the instances of each benchmark problem achieved by the five heuristics. For example, let's take a look at *Buxey* problem, which is an ALBP with precedence constraints given by a 29-tasks DAG. For this problem, the champion method *DE_rks* generated solutions with a mean ARD approx. equal to 1.2%, while the related results for *DE_prb*, *pGA*, *rGA_prb*, and *rGA_rks*, are approx. 2.7%, 3.3%, 2.5%, and 2.2%,

Table 3 A synopsis of the comparative experimental results over the benchmarks of: (a) data set 1 and (b) data set 2

Method	ARD (%)	MRD (%)	ACT (sec)
(a)			
DE_prb	2.07	9.43	29.61
DE_rks	*1.72*	9.43	3.87
pGA	2.31	14.29	2.66
rGA_prb	2.36	9.88	28.86
rGA_rks	2.74	9.43	3.33
(b)			
DE_prb	3.39	12.48	306.57
DE_rks	3.32	11.26	15.79
pGA	4.27	12.94	3.51
rGA_prb	4.38	13.10	112.13
rGA_rks	5.08	14.49	6.32

Table 4 Average running times (in CPU seconds) until the convergence per benchmarks class

Problem name	Problem size	DE_prb	DE_rks	pGA	r GA_prb	r GA_rks
Data set 1						
Buxey	29	0.41	0.80	2.54	1.01	2.21
Sawyer	30	1.42	1.64	0.68	1.51	1.05
Lutz 1	32	0.25	0.88	0.28	1.78	0.49
Gunther	35	0.78	1.08	0.77	1.08	0.79
Kilbridge	45	1.78	1.43	0.98	3.73	1.78
Tonge	70	26.16	3.71	2.76	15.44	4.36
Arcus1	83	48.06	5.29	6.33	36.95	7.97
Lutz 2	89	17.84	1.00	6.08	9.13	1.48
Arcus2	111	169.75	19.02	3.53	126.14	9.81
Average		**29.61**	**3.87**	**2.66**	**21.86**	**3.33**
Data set 2						
Hahn	53	2.81	1.00	0.40	1.67	0.96
Warnecke	58	8.54	3.53	2.53	5.57	2.44
Wee-Mag	75	20.63	3.86	3.24	11.62	4.18
Lutz 3	89	41.88	5.62	6.05	33.49	4.32
Barthold 1	148	380.96	19.47	3.47	200.74	10.08
Barthold 2	148	594.17	33.08	3.56	168.45	12.01
Scholl	297	1096.97	43.95	5.34	363.34	10.27
Average		**306.57**	**15.79**	**3.51**	**112.13**	**6.32**

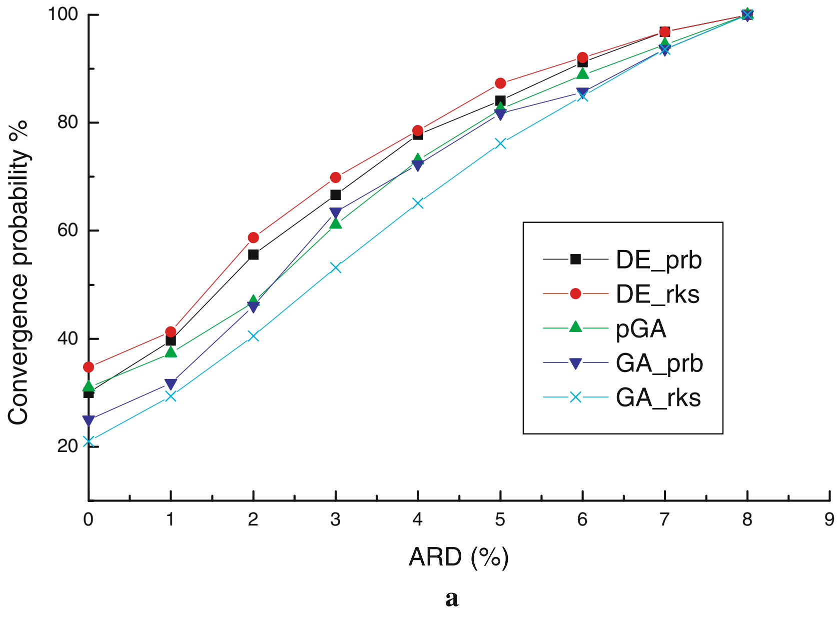
respectively. The worst performance over the benchmarks of data set 1 was encountered in the case of the *Arcus2* problem. Here, both DEAs generated solutions with an ARD<5%, while the three GAs generated solutions with an ARD slightly >5%. *Arcus2* is an ALBP containing a set of 25 test instances represented by a DAG with 111 tasks. Similar performance was also observed in the case of data set 2 benchmarks. For instance, *Hahn* problem (n=53) is an easily solved ALBP by all the heuristics (they achieved the exact optimum for all its test instances) except from *rGA_rks* who generated solutions with an ARD≈1.3%. As it was expected, the largest ARD values (>9%) were generated in the case of the largest ALBP, namely *Scholl* (n=297).

Finally, to be fair with the stochastic behavior of the five heuristics, it was decided to run each of them 10 times

Table 5 Average relative deviations from the existing optimum solution (in percentage) per benchmarks class

Problem name	Problem size	DE_prb	DE_rks	pGA	rGA_prb	r GA_rks
Data set 1						
Buxey	29	2.69	1.16	3.26	2.46	2.15
Sawyer	30	3.52	2.27	2.61	3.67	4.51
Lutz 1	32	0.35	0.32	0.32	0	0.71
Gunther	35	1.02	0.14	0.64	1.21	1.71
Kilbridge	45	0.6	0.66	1.38	0.8	0.96
Tonge	70	2.07	1.88	2.75	3.63	3.63
Arcus1	83	0.83	0.99	1.65	1.48	1.96
Lutz 2	89	2.54	3.08	3.13	2.84	3.76
Arcus2	111	4.98	4.96	5.02	5.11	5.27
Average		**2.07**	**1.72**	**2.31**	**2.36**	**2.74**
Data set 2						
Hahn	53	0.00	0.00	0.00	0.00	1.27
Warnecke	58	3.31	3.74	4.29	5.25	6.40
Wee-Mag	75	1.39	1.23	2.61	2.44	2.37
Lutz 3	89	1.58	1.68	2.54	2.19	3.20
Barthold 1	148	0.24	0.26	0.85	0.58	0.76
Barthold 2	148	7.37	6.85	9.64	9.88	9.77
Scholl	297	9.87	9.51	10.16	10.31	11.76
Average		**3.39**	**3.32**	**4.27**	**4.38**	**5.08**

(starting each time from a different random seed) over each one of the test instances of the benchmarks problems and report the mean values corresponding to the ARD performance index. Hence, each heuristic was run over 10×128=1280 test instances of data set 1, and 10×150=1500 test instances of data set 2. Figure 3 illustrates the convergence probability of the five methods to an ARD less than x% units over the benchmarks instances of the two data sets. As one can see from this figure, *DE_rks* and *DE_prb* heuristics (top curves in Fig. 3 (a),(b)) outperformed all the other methods with the former being the best. More specificly, for the benchmarks of data set 1, *DE_rks* was converged to an ARD less than 1% from the existing optimum in the 41.3% of the instances, to an ARD less than 2% in the 58.7% of the instances, etc. Near to this performance (but lower) is that achieved by *DE_prb*, with an ARD less than 1% in the 39.7% of the experiments, and less than 2% in the 55.6% of the experiments. The corresponding results for the third best heuristic which is *pGA* are: solutions with an ARD<1% in the 37.3% of the



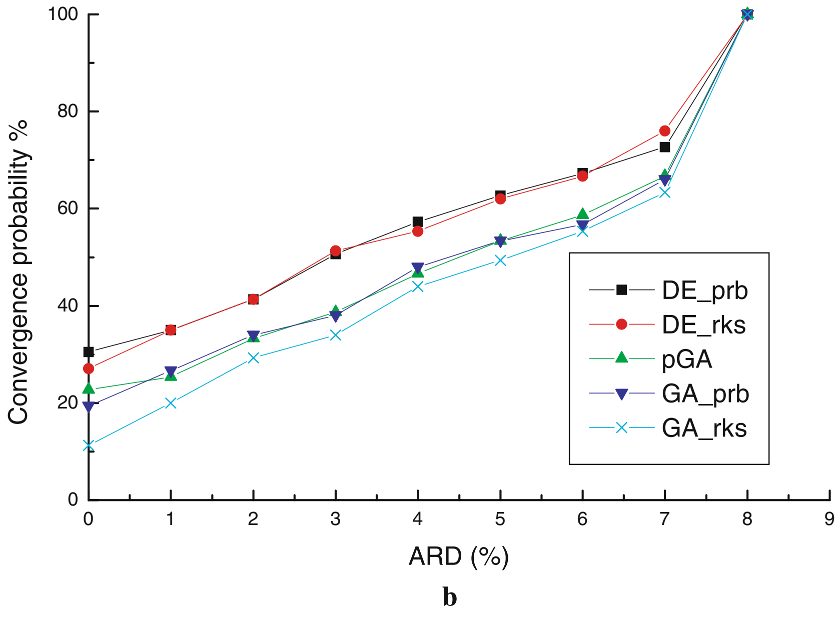


Fig. 3 Convergence probability to an average relative deviation (ARD) from optimum less than x% units. Results concerning the benchmarks of: (**a**) data set 1, (**b**) data set 2

experiments, and solutions with an ARD<2% in the 46.8% experiments. The worst performance (bottom curve in Fig. 3(a)) was encountered by *rGA_rks*. Exact optimum solutions (a zero ARD) have been obtained by *DE_rks* in approx. 34.7% of the experiments, by *DE_prb* in 30%, by *pGA* in 31%, *rGA_prb* in 25%, and *rGA_rks* in 21% of the experiments.

For the case of data set 2 the things with the champion heuristic are not so clear. The two DEAs outperformed all the other heuristics (see top curves in Fig. 3(b)) presenting almost identical performance for solutions with an ARD from the optimum less than 1% and 2%. A slight superiority for *DE_rks* is observed for solutions with an ARD<3%. For solutions with an ARD<4% *DE_prb* managed to perform better than *DE_rks* obtaining solutions below this bound in 57.3% of the experiments. The corresponding performance for the latter heuristic is 55.3% of the experiments. Exact optimum solutions, have been obtained by *DE_prb* in approx., 30% of the experiments, by DE_rks in 27% of the experiments, by *pGA* in 23%, *rGA_prb* in 19%, and *rGA_rks* in 11% of the experiments.

6 Conclusions

This paper investigates the application of differential evolution algorithm (DEA) for the solution of the simple assembly line balancing problem (ALBP) with the objective being to minimize the cycle time of the line. Computational experiments with two different methods for encoding the ALB solutions (namely random-keys and priority-based encoding) have been performed over benchmark test problems from the open literature. The results obtained were quite promising. Comparisons with other previously published evolutionary algorithms showed a superior performance for DEA. The proposed approach is easily implemented and easy to use. Moreover, the overall experimental process showed that DEA could be a promising optimization tool for the solution of many other manufacturing optimization problems of combinatorial nature.

Further research must be done in the direction of improving more the performance of DEA. A first idea is to hybridize DEA with suitable local search techniques such as the simulated annealing algorithm, or the tabu search method. A trend enhanced by the rapidly growing number of papers introducing hybrid meta-heuristics in various production systems. Furthermore, future work will investigate the application of DEA on other more complex ALBPs such as the problem of balancing mixed-model assembly lines. On going research is tackled with the development of a more robust version of the proposed DEA so as to address the multi-objective SALBP. Various objectives are under investigation such as the simultaneously maximization of workload smoothness and the work relatedness of the assembly line.

Acknowledgement This work is integrated in the Innovative Production Machines and Systems (I*PROMS) Network of Excellence.

Appendix

Algorithm Differential Evolution for SALBP-2

```
Pre-processing step:
    Read the necessary input data concerning the DAG $G=(V,E)$ of a specific $n$-tasks ALBP: i.e., the set of tasks $V$, the set of edges $E$, the tasks' processing times $t_j$ ($j$=1,...,$n$), the number of workstations $m$.
Initialization step:
    Set values for the control parameters ($Np$, $F$, $C_R$);
    Initialize generation counter $k$=0;
    Generate a population $S=\{x_{1,k},x_{2,k},...,x_{Np,k}\}$ of $n$-dimensional floating-point vectors;
    The components of $x_{i,k}$ ($i$=1,...,$Np$) are randomly chosen within the range [0,1];
    REPEAT
    for $i$=1 to $Np$ do // create the population $S$ of the new generation //
Mutation step:
    Generate a mutant vector $\hat{x}_{i,k}$ using Eq. (6) and $\xi$=0;
    Crossover step:
    Generate a trial vector $y_{i,k}$ by crossing $x_{i,k}$ and $\hat{x}_{i,k}$ using Eq. (4);
Solution Interpretation Step:
    // Build the phenotypes corresponding to the genotypes $x_{i,k}$ and $y_{i,k}$ by using either
    random-keys, or priority-based encoding scheme (see sub-section 4.1) //
    $P^{x_{i,k}}$=encoding ($x_{i,k}$); $P^{y_{i,k}}$=encoding ($y_{i,k}$);
    // Decode the phenotypes to actual SALBP-2 solutions using the method presented in sub-section 4.1.4. //
    $\mathbf{P}^{x_{i,k}}$=decode_SALBP-2($\mathbf{P}^{x_{i,k}}$);$\mathbf{P}^{y_{i,k}}$=decode_SALBP-2($\mathbf{P}^{y_{i,k}}$);
Acceptance step:
    // Evaluate the phenotypes using Eq. (7), and then apply Eq. (5) //
    if $Cost(P^{y_{i,k}}) < Cost(P^{x_{i,k}})$ then $x_{i,k+1}=y_{i,k}$ else $x_{i,k+1}=x_{i,k}$
    end for
Population Statistics Step:
    Determine the population best ALB solution $P^{best}$;
    if $k$=0 then
    $P^*=P^{best}$ // keep track for the best-so-far solution //
    else if COST($P^{best}$) < COST($P^*$) then
    $P^*=P^{best}$
    end if
    $k$=$k$+1 // increment iteration counter //
```

```
UNTIL $k$ > MAXI // MAXI stands for Maximum
    Iterations //
Return ($P$*)
```

References

1. Kusiak A (2000) Computational intelligence in design and manufacturing. Wiley, New York
2. Scholl A (1999) Balancing and sequencing of assembly lines. Physica, Heidelberg, Germany
3. Becker C, Scholl A (2006) A survey on problems and methods in generalized assembly line balancing. Eur J Oper Res 168:694–715
4. Bukchin J, Dar-El EM, Rubinovitz J (2002) Mixed model assembly line design in a make-to-order environment. Comput Ind Eng 41:405–421
5. Meyr H (2004) Supply chain planning in the German automotive industry. OR Spectrum 26:447–470
6. Baybars I (1986) A survey of exact algorithms for the simple assembly line balancing problem. Manage Sci 32:909–932
7. Klein R, Scholl A (1996) Maximizing the production rate in simple assembly line balancing - a branch and bound procedure. Eur J Oper Res 91:367–385
8. Michalewicz Z, Fogel DB (2000) How to solve it: modern heuristics. Springer, Berlin Heidelberg New York
9. Kirkpatrick S, Gelatt CD Jr, Vecchi MP (1983) Optimization by simulated annealing. Science 220:671–680
10. Glover F (1989) Tabu-search-Part I. ORSA J Comput 1(3):190–206
11. Glover F (1990) Tabu-search-Part II. ORSA J Comput 2(1):4–32
12. Anderson EJ, Ferris MC (1994) Genetic algorithms for combinatorial optimization: the assembly line balancing problem. ORSA J Comput 6:161–173
13. Kim YK, Kim Y-J, Kim Y (1996) Genetic algorithms for assembly line balancing with various objectives. Comput Ind Eng 30/3:397–409
14. Watanabe T, Hashimoto Y, Nishikawa I, Tokumaru H (1995) Line balancing using a genetic evolution model. Control Eng Pract 3:60–76
15. Heinrici A (1994) A comparison between simulated annealing and tabu search with an example from the production planning. In: Dyckhoff H et al (eds) Operations Research Proceedings 1993, Springer, Berlin Heidelberg New York, pp 498–503
16. Scholl A, Voâ S (1996) Simple assembly line balancing - heuristic approaches. J Heuristics 2:217–244
17. Chiang W-C (1998) The application of a tabu search metaheuristic to the assembly line balancing problem. Ann Oper Res 77: 209–227
18. Erel E, Sarin S (1998) A survey of the assembly line balancing procedures. Prod Plan Control 9(5):414–434
19. Rekiek B, Dolgui A, Delchambre A, Bratcu A (2002) State of the art of optimization methods for assembly line design. Annu Rev Control 26:163–174
20. Scholl A, Becker C (2006) State of the art exact and heuristic solution procedures for simple assembly line balancing. Eur J Oper Res 168(3):666–693
21. Storn R, Price K (1997) Differential evolution - a simple and efficient heuristic for global optimization over continuous spaces. J Glob Optim 11(4):341–354
22. Ali MM, Törn A (2004) Population set-based global optimization algorithms: some modifications and numerical studies. Comput Oper Res 31:1703–1725
23. Kaelo P, Ali MM (2006) A numerical study of some modified differential evolution algorithms. Eur J Oper Res 171:674–692
24. Nearchou AC (2006) Meta-heuristics from nature for the loop layout design problem. Int J Prod Econ 101/2:312–328
25. Onwubolu GO, Davendra D (2006) Scheduling flow shops using differential evolution. Eur J Oper Res 169:1176–1184
26. Bean J (1994) Genetics and random keys for sequencing and optimization. ORSA J Comput 6(2):154–160
27. Gen M, Cheng R (2000) Genetic algorithms and engineering optimization. Wiley, New York
28. Warshall S (1962) A theorem of a Boolean matrix. J ACM 9: 11–12
29. Goncalves JF, De Almeida JR (2002) A hybrid genetic algorithm for assembly line balancing. J Heuristics 8(6):629–642
30. Zaharie D (2002) Critical values for the control parameters of differential evolution algorithms. In: Matoušek, Radek, Ošmera, Pavel (eds) Proc. of MENDEL 2002, 8th Int. Mendel Conference on Soft Computing, Brno, Czech Republic, pp 62–67