Robotics and Computer-Integrated Manufacturing 29 (2013) 21–30

Contents lists available at SciVerse ScienceDirect

Robotics and Computer-Integrated Manufacturing

journal homepage: www.elsevier.com/locate/rcim

A comparison of two chromosome representation schemes used in solving a family-based scheduling problem

Chen-Fu Chen, Muh-Cherng Wu *,[1], Yi-Hsun Li, Pang-Hao Tai, Chie-Wun Chiou

Department of Industrial Engineering and Management, National Chiao Tung University, 1001 University Road, Hsinchu, Taiwan 300, ROC

ARTICLE INFO

Article history:
Received 16 November 2011
Received in revised form
24 April 2012
Accepted 29 April 2012
Available online 26 May 2012

Keywords:
Ant Colony optimization
Chromosome representation
Genetic algorithm
Scheduling

ABSTRACT

Meta-heuristic algorithms have been widely used in solving scheduling problems; previous studies focused on enhancing existing algorithmic mechanisms. This study advocates a new perspective—developing new chromosome (solution) representation schemes may improve the performance of existing meta-heuristic algorithms. In the context of a scheduling problem, known as permutation manufacturing-cell flow shop (PMFS), we compare the effectiveness of two chromosome representation schemes (S_{old} and S_{new}) while they are embedded in a meta-heuristic algorithm to solve the PMFS scheduling problem. Two existing meta-heuristic algorithms, genetic algorithm (GA) and ant colony optimization (ACO), are tested. Denote a tested meta-heuristic algorithm by *X_Y*, where *X* represents an algorithmic mechanism and *Y* represents a chromosome representation. Experiment results indicate that *GA_* S_{new} outperforms *GA_*S_{old}, and *ACO_*S_{new} also outperforms *ACO_*S_{old}. These findings reveal the importance of developing new chromosome representations in the application of meta-heuristic algorithms.

1. Introduction

In the past two decades, meta-heuristic algorithms have been widely used in solving complex space-search problems, which are mostly of NP-hard and cannot be solved easily by exact algorithms. A meta-heuristic algorithm is composed essentially of two parts: (1) *algorithmic mechanism* and (2) solution representation (also known as *chromosome representation*). In meta-heuristic applications, previous studies focused on enhancing existing algorithmic mechanisms to improve their performance. Yet, developing methods to improve *chromosome representation* has been rarely noticed.

This study advocates a new perspective—developing new chromosome representation may improve the performance of existing meta-heuristic algorithms. Such a research claim is based on experiment findings obtained from solving a scheduling problem, known as permutation manufacturing-cell flow shop (PMFS). We compare the effectiveness of two chromosome representation schemes (S_{old} and S_{new}) while they are embedded in a particular meta-heuristic algorithm to solve the scheduling problem.

In this study, we tested two existing meta-heuristic algorithms: genetic algorithm (GA) and ant colony optimization (ACO). The tested meta-heuristic algorithms are denoted by *X_Y*, where *X* represents an algorithmic mechanism and *Y* represents a chromosome representation. The experiment results reveals that *GA_*S_{new} outperforms *GA_*S_{old} and *ACO_*S_{new} also outperforms *ACO_*S_{old}. This finding sheds a light on the track of developing new chromosome representations in the application of meta-heuristic algorithms.

The remainder of this paper is organized as follows. Section 2 describes the PMFS scheduling problem and reviews relevant literature. Section 3 presents the two solution representation scheme (S_{old} and S_{new}). Section 4 describes the commonality and distinction of the two ACO algorithms (*ACO_*S_{new} and *ACO_*S_{old}). The commonality and distinction of *GA_*S_{new} and *GA_*S_{old} are presented in Section 5. Section 6 presents the numerical experiments and results. In Section 7, we propose some conjectures for explaining why *GA_*S_{new} outperforms *GA_*S_{old}, and why *ACO_*S_{new} outperforms *ACO_*S_{old}. Concluding remarks are in the last section.

2. Problem and literature review

The scheduling context is a flow shop in which each job must go through the same process sequence. In the process sequence, there is only one machine at each stage and one buffer is equipped for each machine. Four distinct features of the flow shop are introduced below.

* Corresponding author. Tel.: +886 03 5731913; fax: +886 03 5729101.
E-mail addresses: tom.iem96g@nctu.edu.tw (C.-F. Chen), mcwu@mail.nctu.edu.tw (M.-C. Wu), k9592250@gmail.com (Y.-H. Li), supertk1031@gmail.com (P.-H. Tai), cw_chiou@yahoo.com.tw (C.-W. Chiou).
[1] Postal/Present address: 1001 University Road, Hsinchu, Taiwan 30010, ROC

0736-5845/$ - see front matter © 2012 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.rcim.2012.04.009

First, the shop follows a *family-based scheduling paradigm*, due to the adoption of *manufacturing cell* concept (also known as *grouped technology*). That is, all jobs are pre-grouped into various families. Jobs within a family are similar in process requirements; therefore, no setup time is required if two jobs of the same families are consecutively processed. However, significant setup times are required if the two jobs are in different families. Therefore, each job family is processed in a *group manner*. Once a job family starts processing in a stage, we cannot switch to process any other family unless all jobs in the present family have completed their operations.

Second, each job is *individually transported*. Noticeably, the jobs of a family are not transported in a group manner. That is, when a job completes its operation at a stage, it is *immediately* and *individually* transported to the buffer in the next stage.

Third, setup times among families are *sequence-dependent*. The setup time required to switch to process a new family depends upon the difference between the two consecutive families. The greater the difference between the two families, the longer the setup time that is required, and vice versa.

Fourth, the shop is a *permutation flow* shop with *no breakdown*. That is, while jobs traveling through each stage of the flow shop, the job sequence within each family and the sequence among families keep the same. Each machine is reliable and has no breakdown in the scheduling horizon.

The scheduling problem leads to two sequencing decisions: *among-family* sequencing and *within-family* sequencing. Within-family sequencing is the sequence of jobs within each family. Among-family sequencing is the sequence of families. Noticeably, the within-family sequencing decision is essential only when the *individual-transportation feature* is strictly imposed in the scheduling context. That is, the *within-family* sequencing decision would not be needed if jobs are transported in a group manner because in that case changing sequencing decision within a family would not change the scheduling performance.

An actual example of this scheduling context is the SMT (surface mounting machine) process for printed circuit board [1]. On a printed circuit board (PC board), many electronic parts need to be mounted on the surface of the board. A flow shop, involving a sequence of SMT machines, is typically designed for the surface mounting tasks. Each SMT machine is a workstation responsible for mounting a particular group of electronic parts on the PC board, and the group of parts can be changed by a setup. A PC board is a job, which shall pass through each workstation of the flow shop to complete all its part mountings. Two jobs with the same or highly similar part profiles can consecutively pass through the flow shop without requiring any setup. Therefore, PC boards are grouped into families and the family-based scheduling paradigm is usually adopted. For a workstation, a significant setup time is required if switching to process a new family. The greater the difference between two consecutive families, the longer the setup time that is required. Therefore, family setup time is *sequence-dependent*.

In previous studies, this scheduling problem is NP-hard in the strong sense [2,3]. Exact optimization procedures can only be used to solve small-size problem, but are not suitable for solving realistically sized problems because they require extensive computational efforts. Therefore, previous studies developed *approximation* algorithms for finding a near-optimum solution at lower computational expense. Approximation algorithms for solving this scheduling problem are either heuristic approach [1,4–6] or meta-heuristic approach. The meta-heuristic approach includes genetic algorithm [7], tabu search [8,9], memetic algorithm [7], and simulated annealing algorithm [10].

These prior meta-heuristic algorithms are distinct in developing various *evolutionary mechanisms* for generating better solutions, but their chromosome representation schemes are essentially the same. Aside from the traditional track, this study examines the effect of using new chromosome representation while we apply existing meta-heuristic algorithms to solve the scheduling problem.

3. Chromosome representations

As stated, two chromosome representation schemes are used to solve the PMFS scheduling problem. One (called S_{old}) is adopted from previous studies [3,7,11], and the other (called S_{new}) is developed by this study. Before introducing S_{old} and S_{new}, readers are reminded that the PMFS scheduling problem includes two decisions—job sequencing within each family and sequencing among families. Therefore, a chromosome representation must be eligible for accommodating the two sequencing decisions.

3.1. S_{old} chromosome representation

To accommodate the two sequencing decisions, S_{old} chromosome representation has two distinct features: *clustering* and *multiple-segments*. Consider a scheduling context with n jobs (*i*.e., $J_1, J_2, \ldots, J_n$) that have been grouped into q job families (*i*.e., $f_1, f_2, \ldots, f_q$). The *multiple-segments* feature indicates that a chromosome includes $q+1$ segments. The clustering feature indicates that the $q+1$ segments are categorized into two clusters. The first cluster involves *only one* segment, which represents the sequence among the q families. The second cluster involves q segments, each of which represents the job sequence within a particular family.

S_{old} chromosome representation is explained by an example problem with 10 jobs and 3 families. See Fig. 1, the chromosome involves two clusters. The first cluster involves only one segment, which implies that the sequence among families is $f_3 \rightarrow f_2 \rightarrow f_1$. The second cluster involves three segments; the first segment indicates that family f_1 has 3 jobs and their processing sequence is $J_1 \rightarrow J_3 \rightarrow J_2$. Accordingly, the second and the third segments respectively represent the job sequence within family f_2 and f_3.

3.2. S_{new} chromosome representation

In contrast, S_{new} has two other distinct features: *single segment* and *decoding mechanism*. That is, an S_{new} chromosome is a *single*

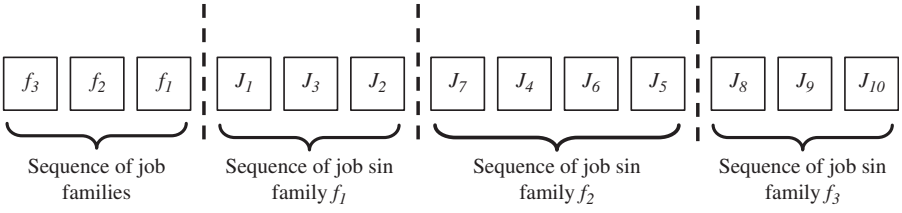

Fig. 1. S_{old} chromosome representation.

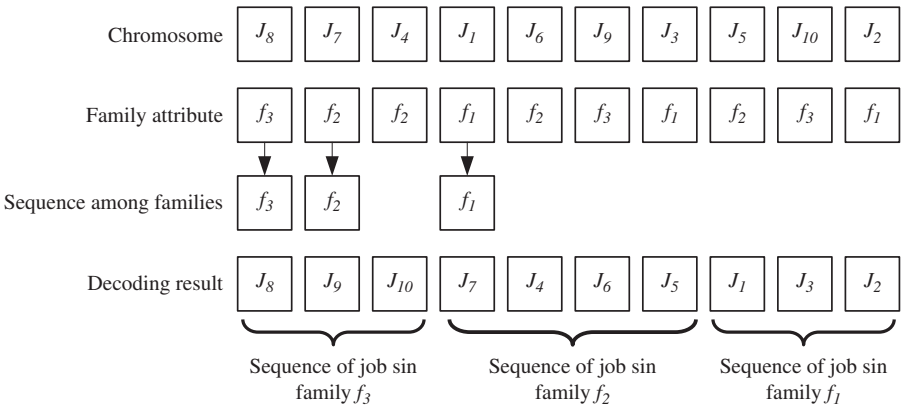

Fig. 2. S_{new} chromosome representation.

segment which comprises a sequence of jobs. By *decoding* the S_{new} chromosome, we can obtain the two scheduling decisions—job sequence within each family and the sequence among families.

Consider a scheduling context with n jobs (*i*.e., $J_1, J_{2,\ldots}, J_n$) that are grouped into q families (*i*.e.,$f_1, f_{2,\ldots}, f_q$). The S_{new} chromosome is a *single segment* that has a sequence of n jobs. To obtain the among-family sequence decision, the decoding mechanism is designed by traveling through the S_{new} chromosome (i.e., the job sequence) and reviewing each job's affiliated family. While traveling through the chromosome, a particular family may appear several times. Each family is only recorded when it appears the first time. As a result, the recorded sequence of these families shows an *among-family* sequence. Alternatively, for jobs within a particular family, their sequencing decision is determined by the appearing sequence of each job in the S_{new} chromosome.

As shown in Fig. 2, an example with 10 jobs and 3 families is used to explain S_{new} chromosome representation scheme. The S_{new} chromosome indicates that the sequence of the first four jobs is $J_8 \rightarrow J_7 \rightarrow J_4 \rightarrow J_1$ and their affiliated family sequence is $f_3 \rightarrow f_2 \rightarrow f_2 \rightarrow f_1$. This implies that the resulting family sequence is $f_3 \rightarrow f_2 \rightarrow f_1$. By conforming to the job precedence relationships of the S_{new} chromosome, the job sequence within each family can be easily obtained. Namely, the job sequence within familyf_3 is $J_8 \rightarrow J_9 \rightarrow J_{10}$, that within f_2 is $J_7 \rightarrow J_4 \rightarrow J_6 \rightarrow J_5$, and that within f_1 is $J_1 \rightarrow J_3 \rightarrow J_2$.

4. ACO algorithms

As stated, the two ACO algorithms (ACO_S_{old} and ACO_S_{new}) are distinct in using different chromosome representations. Yet, their algorithmic mechanisms are essentially the same. Herein, we first describe ACO_S_{new}, and proceed to the required embellishments for developing ACO_S_{old}.

4.1. ACO_S_{new} algorithm

Consider a problem with N jobs to be scheduled. See Fig. 2, an S_{new} chromosome is a *single* segment, which represents a sequence of the N jobs. The chromosome, by a decoding method, can be used to obtain two sequencing decisions—job sequence within each family and the sequence among families. A better chromosome means that it results in a smaller makespan (scheduling performance).

The purpose of ACO_S_{new} algorithm is to obtain a good chromosome (i.e., a good job sequence). To do so, finding a good job sequence is considered as a traveling salesman problem (TSP). Given a virtual start node (say, Node_0), we have to travel through N existing nodes. The purpose is to find a good traveling route (i.e., a traveling sequence of these N nodes), which also denotes a good chromosome (a sequence of N jobs).

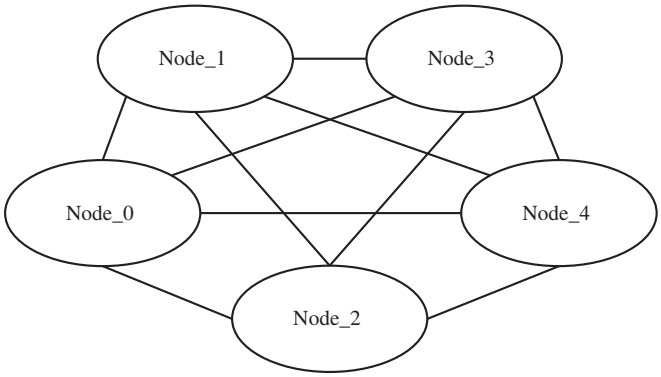

Fig. 3. An example ACO network, with C (N, 2) paths in total.

To illustrate the generation of a traveling route, we model the $N+1$ nodes (including Node_0) as a network. In the network, there exists a path between any two nodes; this yields C (N, 2) paths in total as shown in Fig. 3. Each path has an *aggregate attribute* (also called *preference index*), which denotes the degree of preference for traveling through the path. Let λ_{ij} denote the *preference index* on the path that connects node i and j. The larger is λ_{ij}, the higher is the probability of traveling through the path. Noticeably, λ_{ij}, an *aggregate attribute*, is the combination of two other attributes τ_{ij} and η_{ij} that shall be explained later in this section.

Given such an ACO network, the procedure of generating a traveling route is explained below. Consider that a salesman is now at node i, and there are m remaining nodes to be traveled through. This implies that the salesman has m alternatives (paths) for choosing the next node. In ACO_S_{new}, we use the tournament method [12] to select the next node; that is, the probability of choosing node k as the next node is $p_{ik} = \lambda_{ik}/(\Sigma_{j=1}^{j=m} \lambda_{ij})$. Starting from Node_0, a salesman, by repeatedly following the above *path selection* method, could ultimately find a traveling route. Suppose there are S salesmen designated to travel the network, we *very likely* generate S different traveling routes, due to the *probabilistic* nature of the tournament method. In literature, such a salesman is also called an *ant*, which explains why such an algorithm is named ACO (*ant* colony optimization).

In ACO_S_{new}, the aggregate attributes (λ_{ij}) on the network are iteratively updated. We denote the updated ACO network by W_t, where W_0 represents the initial network and t represents total number of network updating. Notice that W_t and W_{t+k} are of the

same network architecture but are different in path attribute (λ_{ij}). Given W_t, the idea for generating W_{t+1} is based on the feedback of multiple travelers. That is, we first send S ants (salesmen) to travel through the W_t network. This yields S different traveling routes; each route is very likely with different performance (makespan). Then, the performances of these S traveling routes are aggregated to change λ_{ij} for obtaining W_{t+1} as explained below.

As stated, λ_{ij} is the combination of two other attributes τ_{ij} and η_{ij}. The attribute η_{ij} is a *static* attribute, which denotes the relative importance of path ij and shall not be changed during the update of W_t. In *ACO_S*$_{new}$, we define $\eta_{ij}=1/(s_{ij}+p_j)$ where p_j denotes the processing time of job j, and s_{ij} denotes the setup time required for the transition from processing job i to job j. That is, if jobs i and j are of the same family, then $s_{ij}=0$.

In contrast, τ_{ij} is a dynamic attribute, which shall be *dynamically* changed during the update of W_t. Denote the performance of sth traveling route by L_s. By the inclusion of L_s, the attribute τ_{ij} (also called *pheromone*) is updated as follows.

$$\tau_{ij}(t+1)=(1-\rho)\tau_{ij}(t)+\sum_{s=1}^{S}\Delta\tau_{ij}^{s}$$

where

$$\Delta\tau_{ij}^{s}=\begin{cases}\frac{Q}{L_s} & \text{when } s\text{-th traveler passes through path ij}\\ 0 & \text{otherwise}\end{cases}$$

Q: a constant (parameter, $Q>0$); ρ: evaporation rate (parameter, $1>\rho>0$).

In the above formulation, $\tau_{ij}(t+1)$ denotes the *pheromone* on path ij in network W_{t+1}. Notice that the *pheromone* on each path of W_t shall evaporate with a certain percentage (ρ) while we transit from W_t to W_{t+1}. The term $(1-\rho)\tau_{ij}(t)$ denotes the remaining *pheromone* after such evaporation. The term $\Delta\tau_{ij}^{s}$ denotes the feedback of s-th ant about the preference path ij. While sth ant did travel through path ij, the ant suggests the *pheromone* on path ij shall be increased by this amount $\Delta\tau_{ij}^{s}=Q/L_s$. The smaller is L_s (smaller makespan), the larger is $\Delta\tau_{ij}^{s}$. In contrast, if sth ant did not travel through path ij, then the *pheromone* on path ij shall not be increased ($\Delta\tau_{ij}^{s}=0$).

The above formula for updating $\tau_{ij}(t)$ is essentially a recursive form. To carry out the recursive form, we need to define $\tau_{ij}(0)$, the initial value of *pheromone* on path ij. In *ACO_S*$_{new}$, we use the criterion of SPT (shortest processing time) to generate a job sequence. That is, in sequencing N jobs, the shorter is the processing time, the higher is the sequence priority. We call such a chromosome the *initial chromosome* ω_0, denote its makespan by L_0, and set $\tau_{ij}(0)=1/(S\cdot L_0)$ where S is the total number of ants (travelers).

Given $\tau_{ij}(t)$ and η_{ij}, the aggregate attribute $\lambda_{ij}(t)$ is defined as follows.

$\lambda_{ij}(t)=[\tau_{ij}(t)]^{\alpha}[\eta_{ij}]^{\beta}$, where α, β are positive integers.

Given W_t, in which $\lambda_{ij}(t)$ is available, an ant travels through the network in a *probabilistic* manner. As stated, while s-th ant is now at node i, its probability of proceeding to node k can be defined below.

$$p_{ik}^{s}(t)=\begin{cases}\frac{\lambda_{ik}(t)}{\sum_{j\in J_s}\lambda_{ij}(t)} & \text{if node } k\in J_s\\ 0 & \text{if node } k\notin J_s\end{cases}$$

where J_s denotes the set of nodes that have not been travelled through by sth ant. Due to the probabilistic feature of the tournament method, for any two ants s and q, we would find that $J_s\neq J_q$ in most cases; in turn, this leads to $p_{ik}^{s}(t)\neq p_{ik}^{q}(t)$. The variable p_{ik}^{s} is also called *transition probability* for ant s.

While W_t is obtained in the ACO network updating process, the best chromosome ever found must be recorded. We denote such a chromosome $\omega_{best}(t)$ and its makespan $L_{best}(t)$. The ACO updating process shall terminate while the best solution keeps unchanged for T_f generations; that is, $L_{best}(t_n)=L_{best}(t_n+1)=\ldots=L_{best}(t_n+T_f)$.

The components of the *ACO_S*$_{new}$ have been comprehensively presented above. To facilitate readers understand the algorithmic architecture, the procedure of *ACO_S*$_{new}$ is summarized below.

Procedure ACO_S$_{\text{new}}$

Step 1: Initialization
- Input parameters ρ,α,β,S,T_f;
- Set $t=0$.

Step 2: Compute $\tau_{ij}(0)$
- Generate *initial chromosome* ω_0;
- Compute its makespan L_0; Set $\tau_{ij}(0)=1/S\cdot L_0$;
- Create ACO network W_0.

Step 3: Update ACO network W_t
- Send S ants to travel through network W_t;
- Obtain S traveling routes (chromosomes) and their makespans;
- Obtain W_{t+1} by using the ACO network updating method;
- Record the best chromosome $\omega_{best}(t)$ and its makespan $L_{best}(t)$.

Step 4: Termination Check
- If $L_{best}(t_n)=L_{best}(t_n+1)=\ldots=L_{best}(t_n+T_f)$, STOP;
- Else, Go to Step 3.

4.2. *ACO_S*$_{old}$ *algorithm*

See Fig. 1, an S_{old} chromosome is a $F+1$ segment. Of these segments, the first one represents the sequence among families, and each of the remaining F segments represents the job sequence with a particular family.

In *ACO_S*$_{old}$, a chromosome is modeled by a *composite* ACO network, which is composed of $F+1$ *sub-networks*. That is, we model each segment by a sub-network; the resulting $F+1$ sub-networks are then connected together to yield a composite ACO network. Taking the chromosome in Fig. 1 as an example, its composite ACO network involves four sub-networks (see Fig. 4). Each sub-network (representing a segment of the chromosome) is obtained by adopting the aforementioned method for creating a network in *ACO_S*$_{new}$. That is, each sub-network starts with a virtual node; a traveler is requested to travel through the sub-network. The traveling route in turn represents the node sequence of the segment.

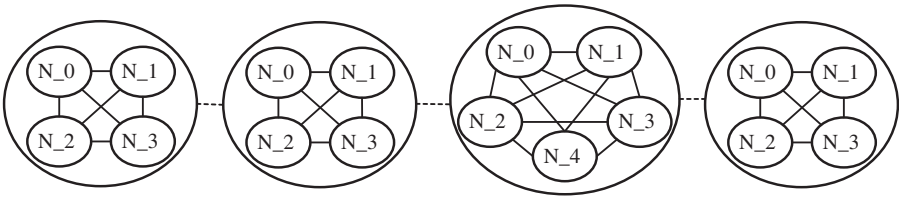

Fig. 4. A composite ACO network.

In *ACO_S$_{old}$*, a chromosome is obtained by requesting an ant travel through the composite ACO network. That is, the ant has to travel through all sub-networks in order. Once traveling through a sub-network, the ant is automatically sent to the *start node* of the next sub-network. Therefore, the ultimate traveling route of the composite ACO network represents a solution.

As stated, *ACO_S$_{old}$* and *ACO_S_$_{new}$* are essentially the same in their algorithmic mechanisms, but are distinct in including different chromosome representations,. That is, Procedure *ACO_S$_{old}$* is exactly the same as Procedure *ACO_S$_{new}$* except that we shall consider W_t as a composite ACO network. In *ACO_S$_{old}$*, we also set $\eta_{ij}=1/(s_{ij}+p_j)$ and set $\tau_{ij}(0)$ for each sub-network based on the SPT criteria (i.e., the smaller is p_j, the higher is the sequence priority). Noticeably, for sub-network 1, p_j denotes the total processing times of jobs in a family.

5. Genetic algorithms

As stated, the two genetic algorithms (*GA_S$_{old}$* and *GA_S$_{new}$*) are distinct in using different chromosome representations. Yet, their algorithms are essentially the same. In fact, *GA_S$_{new}$* is a special case of *GA_S$_{old}$*. Therefore, we first describe the *GA-S$_{old}$* algorithm, and proceed to the required embellishments for developing *GA-S$_{new}$*.

5.1. *GA_S$_{old}$ algorithm*

The *GA_S$_{old}$* algorithm is adapted from Lin et al. [3]. Like the two above ACO algorithms, the scheduling objectives in the two GA algorithms are defined as makespan. The smaller is makespan, the better is the solution.

5.1.1. *Three genetic operators*

In *GA_S$_{old}$*, three types of *genetic operators* are used to create new chromosomes (called *off-springs*) from existing chromosomes (called *parents*). To obtain a parent chromosome, the *binary tournament selection* method [13] is applied, which denotes that we randomly select two chromosomes from a set of chromosomes and pick the better one of the two as a parent.

Of the three types of genetic operators, the *position-based crossover* operator [14] is a 2-*to*-1 operator; that is, two parents are used to create one offspring. The other two types, the *swap* and the *insertion* operators [15], are both 1-to-1 type; that is, one offspring is created by only one parent. Both the swap and insertion operators are regarded as the *mutation* operator.

An example of the *position-based crossover* operator—for a particular chromosome *segment* is shown in Fig. 5. In the figure, there are two parents (Parent 1 and Parent 2) and each parent comprises a distinct sequence of 10 jobs, where the location for accommodating a job is called a *gene*. Three steps are carried out. First, a binary number (0 or 1) is randomly generated for each gene of Parent 1, and the resulting sequence of generated binary numbers is called *Mask*. Second, the offspring inherits the genes (jobs) of Parent 1 that have "1" as binary numbers in the Mask. Third, for the jobs in Parent 2 that have not been assigned to the offspring, we assign each of them *sequentially* to the remaining genes of the offspring.

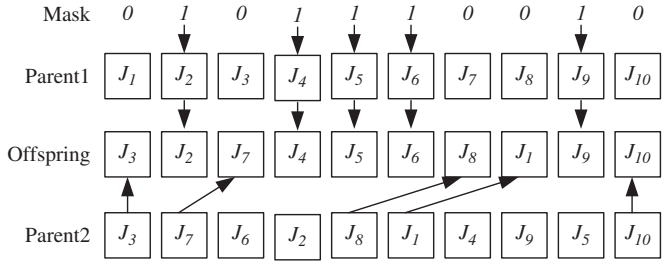

Fig. 5. Position-based crossover operator.

The *swap* operator—for a particular segment is a single step procedure as shown in Fig. 6. Of the segment, two genes are randomly selected and their jobs (J_4 and J_7) are exchanged. The *insertion* operator—for a particular segment is a two-step procedure as shown in Fig. 7. First, we randomly select two genes (J_4 and J_7). Second, of the two jobs, place the latter one (J_7) immediately before the preceding one (J_4).

5.1.2. *Procedure of GA_S$_{old}$*

The procedure of *GA_S$_{old}$* is listed below. As stated, an *S$_{old}$* chromosome involves $F+1$ segments. To generate a new chromosome, each of these $F+1$ segments is *independently* manipulated by applying genetic operators; in turn, the resulting new *F+1* segments are combined together to obtain a new chromosome.

Procedure *GA_S$_{old}$*

Step 1: *Initial setting*
- Input parameters T_f, P_{size}, $0 \le P_c \le 1$, $0 \le P_m \le 1$;
- Set $t=0$; (t denotes the age of population P_t)
- Form a population P_0 by randomly generating P_{size} chromosomes.

Step 2: *Record the best solution in P_t*
- Find $\omega_{best}(t)$; (i.e., the best solution in P_t)
- Denote $L_{best}(t)$ as the makespan of $\omega_{best}(t)$.

Step 3: *Apply crossover operation to update P_t*
- *Skip check*: Generate a random number r, if $r > p_c$ then skip the crossover operation and go to Step 4;
- *Select parents:* Carry out the binary tournament section method twice to pick two chromosomes from P_t as a pair of parents, which are respectively called X_{p1} and X_{p2} (X_{p2} denotes the inferior one in terms of fitness values). Each segment i ($1 \le i \le F+1$) in X_{p1} and X_{p2} is, respectively called $S_{i,p1}$ and $S_{i,p2}$.
- *Create new chromosome*
  - Create offspring for each segment i
    For segment $i=1, \ldots, F+1$
      Apply the position-based crossover operator on $S_{i,p1}$ and $S_{i,p2}$ to create one new segment S_i.
    Endfor
  - Combine S_i ($1 \le i \le F+1$) to form a new chromosome $X_{\text{offspring}}$

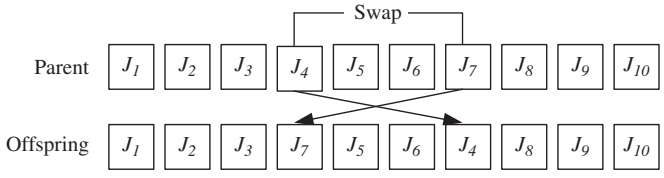

Fig. 6. Swap operator.

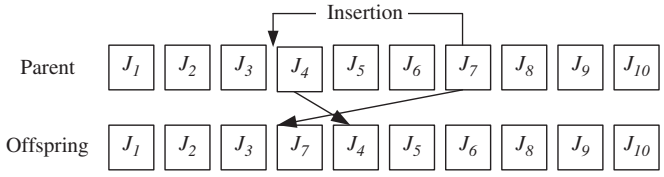

Fig. 7. Insertion operator.

- *Update population* P_t
  - If X_{p2} is inferior to $X_{\text{offspring}}$, then X_{p2} is removed from P_t and $X_{\text{offspring}}$ is placed into P_t; else go to Step 4.

Step 4: *Apply mutation operation to update* P_t

- *Skip check*: Generate a random number r, if $r > p_m$ then skip the mutation operation and go to Step 5;
- *Select parent*: Use the binary tournament selection method to pick one parent, denoted by X_{parent} and each of its segment is denoted by $S_{i,p}$.
- *Select operator:* Of the swap and insertion operators, randomly determine which one to use—each one is with the same probability (0.5).
- *Create new chromosome*
  - Create offspring for each segment i
    For segment $i=1, \ldots, F+1$
      Apply the swap/insertion operator on $S_{i,p}$ to create one new segment S_i.
    Endfor
  - Combine S_i $(1 \le i \le F+1)$ to form a new chromosome $X_{\text{offspring}}$
- *Update Population* P_t: If X_{parent} is inferior to $X_{\text{offspring}}$, then X_{parent} is removed from P_t and $X_{\text{offspring}}$ is placed into P_t; else go to Step 4.

Step 5: Termination check

- If $L_{best}(t) = L_{best}(t+1) = \ldots = L_{best}(t+T_f)$, Then STOP;
- Else, go to Step 2.

5.2. GA_S_{new} algorithm

The procedure GA_S_{new} can be easily obtained because it is essentially a special case of GA_S_{old}. As stated, a scheduling solution in GA_S_{new} is represented by a *single-segment* chromosome, while that in GA_S_{old} is represented by $F+1$ *segments*. To adapt to the single-segment scenario, we can easily develop GA_S_{new} by only setting $F=0$ in the procedure GA_S_{old}.

6. Numerical experiments

Numerical experiments are carried out to compare the effectiveness and efficiency of the aforementioned meta-heuristic algorithms. In the following, we first describe the experiment scenarios. Second, we define three indices for comparing the performance between two algorithms. Third, we compare the experiment results of the two GA algorithms, and finally compare that of the two ACO algorithms.

6.1. Experiment scenarios

The four meta-heuristic algorithms are coded in C++ programming languages, running on personal computers equipped with AMD Athlon(tm) II ⁎4640 3.0 GHz CPU and 4 G RAM.

Data sets for the experiments are adopted from prior studies [1,3,7,9]. The data sets are categorized into 30 scenarios, and each scenario includes 30 problem instances. In total, there are 900 (30×30) problem instances; each problem instance essentially denotes a unique scheduling problem.

Of the 30 *scenarios*, each one is designated by *X-F-m*, where *X* denotes the type of setup times (LSU, MSU, SSU), *F* is the number of families, and *m* is the number of machines. In addition, SSU denotes small setup times, MSU denotes medium setup times, and LSU denotes large setup times. For example, as shown in Table 1, LSU33 denotes a scenario with large setup times, 3 families, and 3 machines.

Of the 30 problem instances in a scenario, each one is varied by randomly changing the following types of parameters: *the number of jobs per family*(n_f), *processing times*, and *family setup times*. These parameters are so designed: n_f is randomly generated from a discrete uniform distribution U [1,10]. The processing times at each stage are randomly generated from U [1,10]. Three different cases of setup times were randomly generated, where U [1,20] is used to model SSU, U [1,50] is used to model MSU, and U [1,100] is used to model LSU.

Table 1
Experiment results of GAs.

Scenario	Makespan							Computation time	
	N	*Nw+Ne*	*Ne*	*Nw*	*Lnew*	*Lold*	*γ (%)*	*Tnew*	*Told*
SSU33	30	30	30	0	134.47	134.47	0.00	14.88	18.54
SSU34	30	28	25	3	150.98	150.99	0.01	16.84	20.50
SSU44	30	28	24	4	185.64	185.64	0.00	21.17	24.56
SSU55	30	29	19	10	241.94	242.11	0.07	30.63	32.30
SSU56	30	28	12	16	253.36	253.71	0.14	31.95	34.81
SSU65	30	28	15	13	285.78	285.96	0.06	36.41	37.27
SSU66	30	28	13	15	294.90	295.14	0.08	40.16	40.03
SSU88	30	22	7	15	402.60	403.14	0.12	67.49	58.19
SSU108	30	16	0	16	481.76	481.94	0.04	95.61	72.24
SSU1010	30	23	1	22	521.90	522.45	0.10	107.94	83.84
MSU33	30	30	30	0	160.00	160.00	0.00	13.75	17.91
MSU34	30	30	29	1	184.68	184.69	0.01	15.70	19.59
MSU44	30	30	28	2	237.60	237.61	0.00	21.01	24.23
MSU55	30	29	24	5	306.09	306.16	0.02	29.64	31.42
MSU56	30	30	24	6	317.47	317.68	0.06	30.04	32.98
MSU65	30	28	23	5	362.63	362.66	0.01	35.92	36.68
MSU66	30	28	19	9	380.02	380.07	0.02	38.55	39.33
MSU88	30	21	7	14	510.39	510.44	0.01	65.15	56.55
MSU108	30	14	2	12	623.95	623.51	(0.08)	89.16	70.74
MSU1010	30	21	1	20	655.17	655.52	0.05	99.95	79.66
LSU33	30	30	30	0	228.03	228.03	0.00	15.44	18.79
LSU34	30	30	30	0	239.00	239.00	0.00	14.94	18.96
LSU44	30	29	29	0	323.44	323.43	(0.00)	19.99	23.81
LSU55	30	29	28	1	415.97	415.97	(0.00)	28.00	30.95
LSU56	30	30	24	6	436.97	437.17	0.04	30.72	33.24
LSU65	30	30	27	3	491.92	492.02	0.02	35.95	36.48
LSU66	30	28	24	4	514.32	514.34	0.00	36.37	38.44
LSU88	30	23	10	13	701.63	701.11	(0.08)	60.78	54.90
LSU108	30	16	3	13	847.62	847.36	(0.04)	85.85	68.94
LSU1010	30	22	1	21	907.13	908.67	0.17	101.14	78.04
Average	**30**	**26.27**	**17.97**	**8.30**	**393.25**	**393.37**	**0.03**	**44.37**	**41.13**

Noticeably, in each problem *instance*, 15 experiments *runs* are carried out. Using a different *random number*, each run generates a different initial solution. In turn, the finally obtained solution in each *run* may be different. Therefore, in each problem *instance*, the average of its 15 experiment runs is taken as the performance of the instance. Furthermore, in each *scenario*, the average of its 30 problem instances is taken as its *ultimate performance measure*. In summary, to compare the two algorithms, we totally carry out 27,000 experiment runs (2 algorithms × 30 scenarios/algorithm × 30 instances/scenario × 15 runs/instance).

6.2. Performance measures

Consider that the experiment results of two algorithms X_S_{old} and X_S_{new} are to be compared, where *X* denotes either *GA* or *ACO*. We use three alternative indices (γ, N_w+N_e, t_0) to compare the performance of the two algorithms.

The first index γ is defined below. Consider that the experiment results of two algorithms X_S_{old} and X_S_{new} are to be compared, where *X* denotes either *GA* or *ACO*. The average performance (makespan) of a particular scenario obtained from X_S_{old} is denoted by L_{old} and that obtained from X_S_{new} is denoted by L_{new}; accordingly, the computation times are respectively denoted by T_{old} and T_{new}. The performance difference between X_S_{old} and X_S_{new} is denoted by $\gamma=(L_{old}-L_{new})/L_{old}$. Given a scenario, while we are comparing the two ACO algorithms, $\gamma > 0$

implies that ACO_S_{new} outperforms ACO_S_{old}. Likewise, $\gamma > 0$ implies that GA_S_{new} outperforms GA_S_{old} while the two GA algorithms are compared in this scenario.

The second index N_w+N_e is explained below. Since there are 30 scenarios, $N=30$ is used to denote the total number of instances in a scenario. In turn, N_e denotes the number of instances with $\gamma=0$ and N_w denotes the number of instances with $\gamma > 0$. As a result, N_w+N_e denotes the number of instances that X_S_{new} either outperforms or performs equally well as X_S_{old}. The higher is N_w+N_e, the better is X_S_{new} comparing against X_S_{old}.

The third index t_0 is explained below. To compare the performance difference between X_S_{new} and X_S_{old}, we carried out a paired t-test for the 900 problem instances (30 scenarios × 30 instances/scenario). For each problem instance, the test statistic for modeling the performance difference is defined as $d=(\overline{L}_{old}-\overline{L}_{new})/\overline{L}_{old}$, where $\overline{L}_{old}$ and $\overline{L}_{new}$, respectively denotes the average performance of the 15 runs of the two algorithms. The t-value is $t_0=\overline{d}/(S_d/\sqrt{n})$ where $\overline{d}$ is the *mean* and S_d is the *standard deviation* of the 900 problem instances. The obtained t-value can be used to justify if X_S_{new} outperforms X_S_{old}, in the sense of statistical significance.

6.3. Experiment results comparison

In the experiments, we set the parameters of the two GA algorithms as follows: $P_{size}=1000$, $p_c=0.95$, $p_m=0.10$, $T_f=4000{,}000$. And the parameters of the two ACO algorithms are set as follows: $\rho=0.8$, $S=C(N,2)$, where N is the total number jobs, $\alpha=1$, $\beta=2$, and $T_f=10{,}000$.

Table 1 shows a comparison between the experiment results of GA_S_{new} and GA_S_{old}. The first index γ ranges from -0.08% to 0.17%, and its average is 0.03%. The second index N_w+N_e ranges from 14 to 30, and its average is 26.27. These two indices indicate that GA_S_{new} on average outperforms GA_S_{old}. The third index $t_0=3.16 > t_{0.025,899}=1.96$, which implies that GA_S_{new} also outperforms GA_S_{old} in the sense of statistical significance. The average of T_{old} is 41.13 s., and the average of T_{new} is 44.37 s. Both the two GA algorithms appear to be computationally inexpensive.

Table 2 shows a comparison between the experiment results of ACO_S_{new} and ACO_S_{old}. The first index γ ranges from 0.56% to 3.08%, and its average is 1.70%. The second index N_w+N_e ranges from 22 to 29, and its average is 26.80. These two indices indicate that ACO_S_{new} on average outperforms ACO_S_{old}. The third index $t_0=30.88 > t_{0.025,899}=1.96$, which implies that ACO_S_{new} also outperforms ACO_S_{old} in the sense of statistical significance. The average of T_{old} is 13.76 s., and the average of T_{new} is 70.36 s. Likewise, the two ACO algorithms are computationally inexpensive.

In summary, in terms of solution quality, Tables 1 and 2 reveal that chromosome representation S_{new} outperforms S_{old} while they embedded in a particular meta-heuristic algorithm (either GA or ACO). Thus, the choice of chromosome representation could have a significant effect on the performance of meta-heuristic algorithm. This advocates the value of developing new chromosome representation scheme in the application of meta-heuristic algorithms.

7. Analyses of experiment result

In this section, we endeavor to explain why GA_S_{new} outperforms GA_S_{old} and why ACO_S_{new} outperforms ACO_S_{old}. To reveal the underlying reasons, we extensively trace and analyze the intermediate and resulting outcomes for each of the four evolutionary processes. Based on some interesting findings, we propose some conjectures on the underlying reasons of why GA_S_{new} and ACO_S_{new} are superior to their counterparts.

Table 2
Experiment results of ACOs.

Scenario	Makespan							Computation time	
	N	$Nw+Ne$	Ne	Nw	L_{new}	L_{old}	γ (%)	T_{new}	T_{old}
SSU33	30	25	4	21	134.96	136.27	0.97	3.35	0.64
SSU34	30	28	1	27	152.56	154.46	1.26	5.15	1.28
SSU44	30	28	0	28	187.63	190.84	1.71	9.77	2.14
SSU55	30	24	0	24	246.79	250.53	1.45	19.49	4.69
SSU56	30	25	1	24	259.57	262.97	1.32	21.03	6.70
SSU65	30	27	0	27	292.51	297.35	1.64	31.09	7.36
SSU66	30	27	0	27	302.18	308.71	2.14	31.67	7.72
SSU88	30	24	0	24	418.97	424.28	1.24	57.37	18.29
SSU108	30	25	0	25	506.84	512.10	1.01	100.11	25.48
SSU1010	30	22	0	22	551.92	554.87	0.56	98.01	29.51
MSU33	30	26	6	20	160.76	162.74	1.23	3.38	0.51
MSU34	30	28	4	24	185.23	187.42	1.18	5.13	0.97
MSU44	30	26	0	26	239.03	242.31	1.32	12.76	2.16
MSU55	30	28	0	28	309.46	315.96	2.07	31.04	5.66
MSU56	30	29	0	29	321.72	329.32	2.32	30.41	4.90
MSU65	30	28	0	28	367.47	375.24	2.07	49.92	7.51
MSU66	30	28	0	28	386.52	396.05	2.42	49.08	8.39
MSU88	30	29	0	29	529.09	546.14	3.08	102.58	20.49
MSU108	30	29	0	29	655.03	667.53	1.88	175.06	39.82
MSU1010	30	25	1	24	687.61	698.20	1.52	164.46	40.30
LSU33	30	22	5	17	228.67	229.98	0.56	4.98	0.80
LSU34	30	27	5	22	239.38	241.19	0.76	5.50	0.83
LSU44	30	28	4	24	324.13	327.27	0.93	16.36	1.95
LSU55	30	28	0	28	419.19	426.50	1.70	48.32	4.75
LSU56	30	27	0	27	440.70	449.42	1.95	50.98	5.80
LSU65	30	29	0	29	496.02	507.07	2.17	92.34	6.75
LSU66	30	27	0	27	518.88	531.96	2.48	87.37	8.96
LSU88	30	29	0	29	715.14	736.53	2.90	180.73	22.54
LSU108	30	27	0	27	883.67	905.50	2.40	328.03	64.14
LSU1010	30	29	0	29	947.02	972.93	2.66	295.34	61.76
Average	**30**	**26.80**	**1.03**	**25.77**	**403.62**	**411.39**	**1.70**	**70.36**	**13.76**

7.1. GA_S_{new} vs GA_S_{old}

Our conjectures on why GA_S_{new} outperforms GA_S_{old} are two-fold. The first conjecture is that the first segment of the S_{old} chromosome, which models a sequencing decision among families, is a *dominant segment*. Namely, the first segment appears to have much more influence on the solution quality than the other segments. By reviewing the GA_S_{old} algorithm presented in Section 5.1.2, the chromosome population is continually updated by replacing inferior chromosomes by relative superior ones. Such a replacement mechanism leads to that the chromosomes in the dominant segment tend to approach a *homogeneous* state. That is, in GA_S_{old}, most chromosomes in the population ultimately tend to have the same dominant segment, as illustrated in Fig. 8 which is obtained from tracing a problem instance in scenario MSU9. Notice that this problem instance is referred for all experiment discussions of this section.

In Fig. 8, the horizontal axis denotes T_f (the parameter for program termination); and of the two vertical axes, the right one denotes makespan (solution quality) and the left one denotes the percentage of dominant segments that have become homogeneous (i.e., with exactly the same appearance); herein such a percentage is denoted by θ_h. Notice that while T_f increases (i.e., we relax the program termination condition), the solution quality initially improves but gradually tends to approach a steady state (i.e., cannot be further improved). In addition, while T_f increases, θ_h gradually increases and reaches toward almost 100%. This implies that the dominant segment now has very rare probability to change. That is, if the dominant segment of the GA_S_{old} is by chance trapped into a *local optimum solution* too early, the *GA_Sold* would be very likely trapped there and could not be further improved.

The second conjecture on why GA_S_{new} outperforms GA_S_{old} is that the chromosome representation of S_{new} is essentially

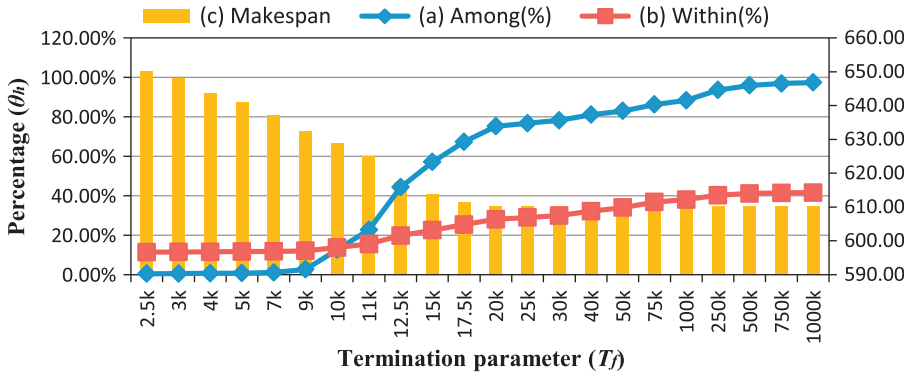

Fig. 8. Tracing results of GA_S_{old} in a problem instance of MSU9 at various T_f: (a) QUOTE of the first segment moves toward 100%, (b) average QUOTE of the other segments moves toward 40%, (c) histogram of solution quality.

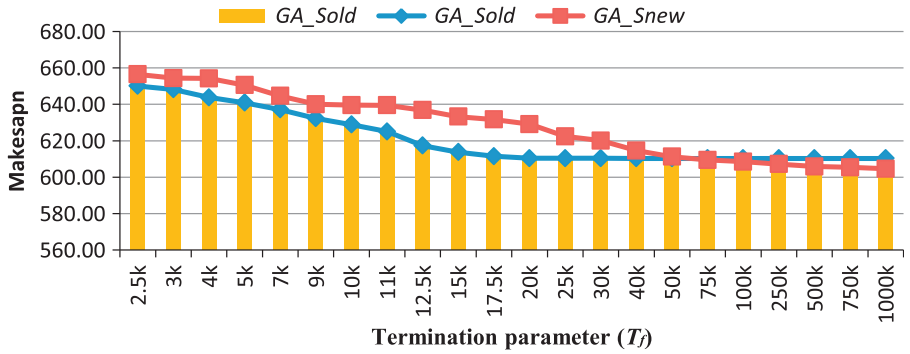

Fig. 9. A comparison of solution quality between GA_S_{old} and GA_S_{new} at various T_f < none > . < /none > .

redundant; that is, one scheduling decisions can be modeled by more than one chromosomes. This *one-to-many* feature has two pros and cons impacts. The search of GA_S_{new} is less efficient due to redundancy representation. Yet it tends to be more effective due to flexibility in representation, because the S_{new} chromosomes in the population tend not to be homogenized. Fig. 9 compares the solution quality of GA_S_{old} and GA_S_{new}. The figure reveals that GA_S_{new} is inferior to GA_S_{old} while at lower T_f, and GA_S_{new} becomes superior while at higher T_f. This implies that if we provide enough computational time resources, GA_S_{new} could keep improving solution quality yet GA_S_{old} would tend to be trapped into a local optimum and cannot be improved further. Therefore, GA_S_{new} tend to outperform GA_S_{new} while we have enough computation time resources.

7.2. ACO_S_{new} vs ACO_S_{old}

Our conjectures on why ACO_S_{new} outperforms ACO_S_{old} are two-fold. The first conjecture is that the sequencing decision among families is a *dominant decision*. That is, the sequencing decision among families is much more important than the sequencing decisions of jobs within each family. In fact, this conjecture has been proposed and well supported by the empirical tracing results illustrated in Fig. 8, at the time we endeavor to explain why GA_S_{new} would outperform GA_S_{old}. We therefore adopt this conjecture in the context of explaining why ACO_S_{new} outperforms ACO_S_{old}. Such a conjecture adoption implies that we shall pay attention to the analysis of the dominant decision.

The second conjecture is that the *average ant traffic intensity* on an ACO_S_{old} network is much higher than that on an ACO_S_{new} network. For ACO_S_{old}, consider a simple scheduling problem as an example in which there are 4 families and 20 jobs. Then we have five sub-networks in ACO_S_{old} and the first sub-network (i.e., the dominant decision or the sequencing decision among families) is as shown in Fig. 3. In this sub-network, the number of travelling paths is C (5, 2)=10. The number of ants to be selected in each evolutionary iteration is C (20, 2)=190. Then the average traffic intensity for the ACO_S_{old} is 190/10=19 ants per path. In contrast, for the ACO_S_{new} counterpart, we have *only one* big network which involves 21 nodes (i.e., 20 job nodes and 1 starting nodes). In each evolutionary iteration, the number of paths is C (21, 2)=210; and the number of ants is likewise C (20, 2)=190. As a result, the average traffic intensity for the ACO_S_{new} is 190/210= 0.90 ants per path, which is far less than that of ACO_S_{old}.

Before explaining why ant traffic intensity is an appropriate conjecture, we define two ACO network attributes ($\hat{\tau}_{ij}$ and $\overline{\tau}_{ij}$) as pre-requisites. For an ACO program, while its evolutionary process is terminated, the resulting pheromone (τ_{ij}) on each travelling path (a path connecting node i and node j) can be recorded. Define $\hat{\tau}_{ij} = \tau_{ij} / \Sigma_{k \neq i / k \in S} \tau_{ik}$ where S denotes the set of all nodes in the concerned ACO network. The higher is $\hat{\tau}_{ij}$, the higher probability is the path selected by travelling ants. For the problem instance of MSU9, we can obtain a from-to-matrix as shown in Table 3. Based on the from-to-matrix, we define a *dominant route* by sorting $\hat{\tau}_{ij}$ in descending order. Then, out of the C (N, 2) paths, we select N distinct paths as the *dominant route* (refer to the highlighted paths in Table 3). For the dominant route, we further define an aggregate attribute $\overline{\tau}_{ij} = \Sigma_{\hat{\tau}_{ij} \in D} \hat{\tau}_{ij} / N$ where D denotes a set that includes all paths of the dominant route. Notice that $\overline{\tau}_{ij}$ essentially denotes the traffic intensity of the dominant route.

We now justify the second conjecture by analyzing $\overline{\tau}_{ij}$ on the ACO_S_{old} and ACO_S_{new} networks. As shown in Fig. 10, $\overline{\tau}_{ij}$ of ACO_S_{old} and ACO_S_{new} are compared while they are used to solve the problem instance of MSU9. From the figure, we could see that $\overline{\tau}_{ij}$ of ACO_S_{old} is around 50% and that of ACO_S_{new} is around 20%. This implies that the *composing paths* on the dominant route in

Table 3
From-to-matrix of ACO_S_{old} in a problem instance of MSU9 scenario.

From-to-matrix	To									
	F1	**F2**	**F3**	**F4**	**F5**	**F6**	**F7**	**F8**	**F9**	**F10**
From										
FStarting node	**99.97%**	0.00%	0.00%	0.00%	0.03%	0.00%	0.00%	0.00%	0.00%	0.00%
F1	–	0.07%	0.00%	0.00%	**68.94%**	12.22%	11.22%	7.25%	0.25%	0.04%
F2	0.09%	–	89.74%	1.98%	3.01%	0.40%	**4.49%**	0.02%	0.00%	0.27%
F3	0.00%	**49.79%**	–	50.21%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
F4	0.00%	1.10%	**50.12%**	–	48.79%	0.00%	0.00%	0.00%	0.00%	0.00%
F5	49.97%	1.66%	0.00%	**48.37%**	–	0.01%	0.00%	0.00%	0.00%	0.00%
F6	12.60%	0.31%	0.00%	0.00%	0.01%	–	38.30%	48.51%	**0.23%**	0.04%
F7	10.50%	3.18%	0.00%	0.00%	0.00%	34.77%	–	**50.34%**	1.01%	0.19%
F8	6.62%	0.02%	0.00%	0.00%	0.00%	**42.96%**	49.11%	–	0.28%	1.02%
F9	0.51%	0.00%	0.00%	0.00%	0.00%	0.44%	2.17%	0.62%	–	**96.25%**
F10	0.08%	0.41%	0.00%	0.00%	0.00%	0.08%	0.42%	2.26%	96.74%	–

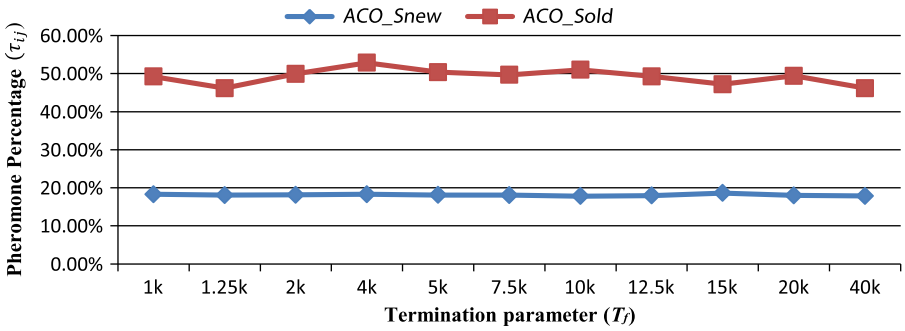

Fig. 10. A comparison of QUOTE between ACO_S_{old} and $ACO\text{-}S_{new}$ at various T_f.

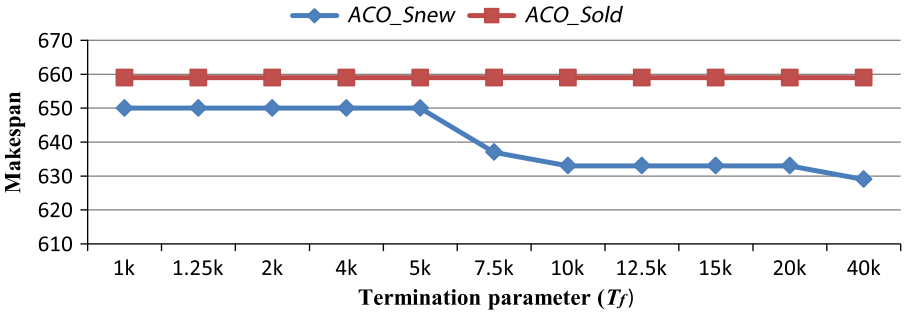

Fig. 11. A comparison of solution quality between ACO_S_{old} and ACO_S_{new} at various T_f.

ACO_S_{old}, compared against that of ACO_S_{new}, have higher probability of being travelled. That is, ACO_S_{new} is more *divergent* than ACO_S_{old} in creating new scheduling solutions. Suppose the dominant route is a local optimum solution, ACO_S_{old} shall have a lower probability than ACO_S_{new} in an attempt to generate a new solution better than the local optimum. Such a local optimum conjecture can be empirically supported by comparing Tables 1 and 2, in which GA-based algorithms significantly outperforms ACO-based algorithms; this implies that ACO-based algorithms tend to be trapped into a local optimum.

We further have empirical findings to support the aforementioned conjecture—ACO_S_{new} is more *divergent* than ACO_S_{old} in creating new scheduling solutions and more likely to generate a new solution better than the current local optimum. See Fig. 11, while T_f increases, ACO_S_{new} keeps improving in the obtained optimum solution. In contrast, the obtained optimum solution of ACO_S_{old} tends to quickly become stable; that is, the optimum solution obtained at T_f=1k is close to that obtained at T_f=40k. This implies that ACO_S_{old} may have been trapped into a local optimum in a very early stage.

8. Concluding remarks

This paper attempts to highlight the importance of developing new chromosome representation in the application of meta-heuristic algorithms. Such a research claim is justified by solving a scheduling problem, called permutation manufacturing-cell flow shop (PMFS). We compare the effectiveness of two chromosome representation schemes (S_{old} and S_{new}) while they are embedded in a particular meta-heuristic algorithm to solve the scheduling problem.

We first compare the effectiveness of S_{old} and S_{new} while they are embedded in a GA algorithmic mechanism, which in turn yields two algorithms GA_S_{new} and GA_S_{old}. Experiment results indicate that GA_S_{new} outperforms GA_S_{old}. Second, we proceed to compare the effectiveness of S_{old} and S_{new} while they are embedded in ACO algorithmic mechanism. Experiment results indicate that ACO_S_{new} also outperforms ACO_S_{old}. These findings reveal that chromosome representation S_{new} appears to be better than S_{old}. This also implies that developing appropriate chromosome representations might be very important in the application of meta-heuristic algorithms.

In addition, we propose some conjectures for explaining why GA_S_{new} outperforms GA_S_{old}, and why ACO_S_{new} also outperforms ACO_S_{old}. These conjectures are supported by empirical findings. The three main conjectures are summarized here. First, the sequencing decision among families is a dominant decision. Second, in the dominant decision, GA_S_{old} tends to generate a homogeneous chromosome population and is likely be trapped into a local optimum. Third, in the dominant decision, ACO_S_{old} tend to have a higher traffic intensity which in turn reduces the probability of generating a new better solution.

Some possible extensions of this research are being considered. First, we attempt to compare the effectiveness of some other meta-heuristic algorithms while they are embedded with S_{old} and S_{new}. Second, we attempt to analyze the underlying reasons why S_{new}-based algorithms and S_{old}-based algorithms would be different in their obtained solutions. Third, we attempt to use these underlying reasons as a guide to modify and enhance existing meta-heuristic algorithms.

Acknowledgements

This work is financially supported by a research contract NSC 99-2221-E-009-110-MY3.

References

[1] Schaller JE, Gupta JND, Vakharia AJ. Scheduling a flowline manufacturing cell with sequence dependent family setup times. European Journal of Operational Research 2000;125(2):324–339.
[2] Gupta JND, Stafford EF. Flowshop scheduling research after five decades. European Journal of Operational Research 2006;169:699–711.
[3] Lin SW, Ying KC, Lee ZJ. Metaheuristics for scheduling a non-permutation flowline manufacturing cell with sequence dependent family setup times. Computers & Operations Research 2009;36:1110–1121.
[4] Ruben RA, Mosier CT, Mahmoodi F. A comprehensive analysis of group scheduling heuristics in a job shop cell. International Journal of Production Research 1993;31(4):1343–1369.
[5] Frazier GV. An evaluation of group scheduling heuristics in a flow-line manufacturing cell. International Journal of Production Research 1996;34(4):959–976.
[6] Reddy V, Narendran TT. Heuristics for scheduling sequence-dependent set-up jobs in flow line cells. International Journal of Production Research 2003;41(1):193–206.
[7] Franca PM, Gupta JND, Mendes AS. Evolutionary algorithms for scheduling a flowshop manufacturing cell with sequence dependent family setups. Computers and Industrial Engineering 2005;48(3):491–506.
[8] Logendran R, DeSzoeke P, Barnard F. Sequence-dependent group scheduling problems in flexible flow shops. International Journal of Production Economics 2006;102(1):66–86.
[9] Hendizadeh SH, Faramarzi H, Mansouric SA, Gupta JND, ElMekkawy TY. Meta-heuristics for scheduling a flowline manufacturing cell with sequence dependent family setup times. International Journal of Production Economics. 2008;111(2):593–605.
[10] Lin SW, Gupta JND, Ying KC, Lee ZJ. Using simulated annealing to schedule a flowshop manufacturing cell with sequence dependent family setup times. International Journal of Production Research 2009;47(12):3205–3217.
[11] Ying KC, Gupta JND, Lin SW, Lee ZJ. Permutation and nonpermutation schedules for the flowline manufacturing cell with sequence dependent family setups. International Journal of Production Research 2009;48(8):2169–2184.
[12] Dorigo M, Gambardella LM. Ant colony system: a cooperative learning approach to the traveling salesman problem, Evolutionary Computation. IEEE Transactions on 2002;1(1):53–66.
[13] A. Brindle, Genetic Algorithms for Function Optimization. Doctoral Dissertation, University of Alberta, Edmonton, Canada, 1981, unpublished doctoral dissertation.
[14] G. Syswerda, Uniform crossover in genetic algorithms, in: Proceedings of the Third International Conference on Genetic Algorithms, Schaffer, J. (Ed.), Morgan Kaufmann Publishers, Los Altos, CA, 1989, pp. 2–9.
[15] Wang CS, Uzsoy R. A genetic algorithm to minimize maximum lateness on a batch processing machine. Computers & Operations Research 2002;29:1621–1640.