Genetic Algorithms With Guided and Local Search Strategies for University Course Timetabling

Shengxiang Yang, *Member, IEEE*, and Sadaf Naseem Jat

***Abstract*—The university course timetabling problem (UCTP) is a combinatorial optimization problem, in which a set of events has to be scheduled into time slots and located into suitable rooms. The design of course timetables for academic institutions is a very difficult task because it is an NP-hard problem. This paper investigates genetic algorithms (GAs) with a guided search strategy and local search (LS) techniques for the UCTP. The guided search strategy is used to create offspring into the population based on a data structure that stores information extracted from good individuals of previous generations. The LS techniques use their exploitive search ability to improve the search efficiency of the proposed GAs and the quality of individuals. The proposed GAs are tested on two sets of benchmark problems in comparison with a set of state-of-the-art methods from the literature. The experimental results show that the proposed GAs are able to produce promising results for the UCTP.**

***Index Terms*—Genetic algorithm (GA), guided search, local search (LS), university course timetabling problem (UCTP).**

I. Introduction

TIMETABLING is one of the common scheduling problems, which can be described as the allocation of resources for tasks under predefined constraints so that it maximizes the possibility of allocation or minimizes the violation of constraints [40]. Timetabling problems are often made complicated by the details of a particular timetabling task. A general algorithm approach to one problem may turn out to be incapable for another problem, because certain special constraints are required in a particular instance of that problem. In the university course timetabling problem (UCTP), events (subjects and courses) have to be allocated into a number of time slots and rooms while satisfying various constraints. It is very difficult to find a general and effective solution for timetabling due to the diversity of the problem, the variance of constraints, and particular requirements from university to university according to the characteristics. There is no known deterministic polynomial time algorithm for the UCTP, since it is an NP-hard combinatorial optimization problem [18].

Manuscript received September 8, 2009; revised December 21, 2009; accepted March 26, 2010. This work was supported by the Engineering and Physical Sciences Research Council of U.K. under Grant EP/E060722/1. This paper was recommended by Associate Editor J. Lazansky.

S. Yang is with the Department of Information Systems and Computing, Brunel University, Uxbridge, UB8 3PH, U.K. (e-mail: shengxiang.yang@brunel.ac.uk).

S. N. Jat is with the Department of Computer Science, University of Leicester, Leicester, LE1 7RH, U.K. (e-mail: snj2@le.ac.uk).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TSMCC.2010.2049200

The research on timetabling problems has a long history of more than 40 years, starting with Gotlieb in 1962 [22]. Researchers have proposed various timetabling approaches by using graph coloring methods, constraint-based methods, population-based approaches (e.g., genetic algorithms (GAs), ant-colony optimization, and memetic algorithms), metaheuristic methods (e.g., tabu search (TS), simulated annealing (SA), and great deluge), variable neighborhood search (VNS), hybrid and hyperheuristic approaches, etc. A comprehensive review and recent research directions in timetabling can be found in [13], [27], and [35]. GAs have been used to solve the UCTP in the literature [3], [28], [34]. Rossi-Doria *et al.* [37] compared different metaheuristics to solve the UCTP. They concluded that conventional GAs do not give good results among a number of approaches developed for the UCTP. Hence, conventional GAs need to be enhanced to solve the UCTP.

Recently, a guided search GA, denoted as *GSGA*, has been proposed for solving the UCTP [23]. GSGA consists of a guided search strategy and a local search (LS) technique. One of the important concept of GAs is the notion of population. Unlike traditional search methods, GAs rely on a population of candidate solutions [38]. In GSGA, a guided search strategy is used to create offspring into the population based on an extra data structure. This data structure is constructed from the best individuals from the population, and hence, stores useful information that can be used to guide the generation of good offspring into the next populations. The main advantage of this data structure is that it maintains partial information of good solutions, which otherwise may be lost in the selection process. In GSGA, an LS technique is also used to improve the quality of individuals through searching in three kinds of neighborhood structures. GSGA has shown some promising results based on some preliminary experiments in [23].

In this paper, we further investigate the performance of GSGA with LS strategies for the UCTP. Here, a unified framework of combining standard GA and LS strategies is used. In addition to the original LS strategy used in GSGA [23], a new LS strategy is introduced into GSGA, which leads to an extended GSGA (EGSGA) for the UCTP. In order to investigate the effect of parameters on the performance of GSGA for the UCTP, a sensitivity analysis of key parameters of GSGA is carried out by systematic experiments based on a set of benchmark UCTP instances. In order to test the performance of GSGAs, experiments are also carried out to compare GSGAs with other variants of GAs and a set of state-of-the-art methods from the literature on a set of benchmark UCTP instances.

The rest of this paper is organized as follows. Section II briefly describes related work on the UCTP. The UCTP studied in this paper is described in Section III. Section IV presents the proposed method and the common framework for all methods described in this paper. Experimental results of the sensitivity analysis of key parameters of GSGA and comparing the proposed GSGAs with other algorithms from the literature are reported and discussed in Section V. Finally, Section VI concludes this paper with some discussions on the future work.

II. Related Work

Several algorithms have been introduced to solve timetabling problems. The earliest set of algorithms are based on graph coloring heuristics. These algorithms show a great efficiency in small instances of timetabling problems, but are not efficient in large instances. Later, stochastic search methods, such as GAs, SA, TS, etc., were introduced to solve timetabling problems.

Generally speaking, there are two types of metaheuristics algorithms [5]. The first type are local-area-based algorithms and the second are population-based algorithms. Each type has some advantages and disadvantages. Local-area-based algorithms include SA [41], very large neighborhood search [1], TS [30], and many more. Usually, local-area-based algorithms focus on exploitation rather than exploration, which means that they move in one direction without performing a wider scan of the search space. Population-based algorithms start with a number of solutions and refine them to obtain global optimal solution(s) in the whole search space, and hence, are global-area-based algorithms. Population-based algorithms that are commonly used to tackle timetabling problems include evolutionary algorithms (EAs) [14], particle swarm optimization [19], ant-colony optimization [39], artificial immune system [31], etc.

In recent years, several researchers have used GAs to solve the UCTP. They enhanced the performance of GAs by using modified genetic operators, heuristics operators, and LS techniques. Generally speaking, when a simple GA is employed, it may generate illegal timetables that have duplicate and/or missing events. The quality of a solution produced by population-based algorithms may not be superior to local-area-based algorithms mainly due to the fact that population-based algorithms are more concerned with exploration than exploitation [5]. Population-based algorithms scan solutions in the whole search space without focusing on the individuals of good fitness within a population. Furthermore, population-based algorithms may experience premature convergence, which may lead to them being trapped into local optima. Population-based algorithms have another drawback of requiring more time [33]. However, EAs have several advantages when compared with other optimization techniques [34]. For example, GAs can perform a multidirectional search using a set of candidate solutions [21].

Various combinations of local-area- and global-area-based algorithms have been reported to solve timetabling problems in the literature [32], [37], [40]. In addition, it is also being increasingly realized that EAs without incorporation of problem-specific knowledge do not perform as well as mathematical programming-based algorithms on certain classes of timetabling problems [8]. In this paper, we want to combine the good properties of local- and global-area-based algorithms to solve the UCTP. We try to make a balance between the exploration ability (global improvement) of GAs and exploitation ability (local improvement) of LS. In addition, an external memory data structure is introduced to store parts of previous good solutions and reintroduce these stored parts into offspring in order to enable the proposed GAs to quickly locate the optimum of a UCTP.

III. University Course Timetabling Problem

A. Problem Description

According to Carter and Laporte [13], the UCTP is a multidimensional assignment problem, in which students and teachers (or faculty members) are assigned to courses, course sections, or classes and events (individual meetings between students and teachers) are assigned to classrooms and time slots. The real-world UCTP consists of different constraints: some are hard constraints and some are soft constraints. Hard constraints must not be violated under any circumstances, e.g., a student cannot attend two classes at the same time. Soft constraints should preferably be satisfied, but can be accepted with a penalty associated to their violation, e.g., a student should not attend more than two classes in a row.

In this paper, we will test our proposed algorithms on the problem instances discussed in [37]. We deal with the following hard constraints:

1) no student attends more than one events at the same time;
2) the room is big enough for all the attending students;
3) the room satisfies all the features required by the event;
4) only one event is in a room at any time slot.

There are also soft constraints, which are equally penalized by the number of their violations and are described as follows:

1) a student has a class in the last time slot of a day;
2) a student has more than two classes in a row;
3) a student has a single class on a day.

B. Problem Formulation

In a UCTP, we assign an event (course and lecture) into a time slot and also assign a number of resources (students and rooms) in such a way that there is no conflict between the rooms, time slots, and events. As mentioned by Rossi-Doria *et al.* [37], the UCTP consists of a set of n events (classes and subjects) $E = \{e_1, e_2, \ldots, e_n\}$ to be scheduled into a set of 45 time slots $T = \{t_1, t_2, \ldots, t_{45}\}$ (i.e., nine for each day in a five day week), a set of m available rooms $R = \{r_1, r_2, \ldots, r_m\}$ in which events can take place, a set of k students $S = \{s_1, s_2, \ldots, s_k\}$ who attend the events, and a set of l available features $F = \{f_1, f_2, \ldots, f_l\}$ that are satisfied by rooms and required by events.

In addition, interrelationships between these sets are given by five matrices. The first matrix $A_{k,n}$, called *Student-Event* matrix, shows which event is attended by which students. In $A_{k,n}$, the value of $a_{i,j}$ is 1 if student $i \in S$ should attend event $j \in E$; otherwise, the value is 0. The second matrix $B_{n,n}$, called *Event-Conflict* matrix, indicates whether two events can be scheduled in the same time slot or not. It helps to quickly identify events that can be potentially assigned to the same time slot. The third matrix $C_{m,l}$, called *Room-Features* matrix, gives the features that each room possesses, where the value of a cell $c_{i,j}$ is 1 if $i \in R$ has a feature $j \in F$; otherwise, the value is 0. The fourth matrix $D_{n,l}$, called *Event-Features* matrix, gives the features required by each event. It means that event $i \in E$ needs feature $j \in F$ if and only if $d_{ij} = 1$. The last matrix $G_{n,m}$, called *Event-Room* matrix, lists the possible rooms to which each event can be assigned. Through this matrix, we can quickly identify all rooms that are suitable in size and feature for each event. Usually, a matrix is used for assigning each event to a room r_i and a time slot t_i. Each pair of (r_i, t_i) is assigned a particular number that corresponds to an event. If a room r_i in a time slot t_i is free or no event is placed, then "-1" is assigned to that pair. In this way, we assure that there will be not more than one event assigned to the same pair so that one of the hard constraint will always been satisfied.

For the room assignment, we use a matching algorithm described by Rossi-Doria *et al.* [37]. For every time slot, there is a list of events taking place in it and a preprocessed list of possible rooms to which the placement of events can occur. The matching algorithm uses a deterministic network flow algorithm and gives the maximum cardinality matching between rooms and events.

In general, the solution to a UCTP can be represented in the form of an ordered list of pairs (r_i, t_i), of which the index of each pair is the identification number of an event $e_i \in E$ ($i = 1, 2, \ldots, n$). For example, the time slots and rooms are allocated to events in an ordered list of pairs like

$$(2, 4), (3, 30), (1, 12), \ldots, (2, 7)$$

where room 2 and time slot 4 are allocated to event 1, room 3 and time slot 30 are allocated to event 2, etc.

The goal of the UCTP is to minimize the soft-constraint violations of a feasible solution (a feasible solution means that no hard-constraint violation exists in the solution). The objective function $f(s)$ for a timetable s is the weighted sum of the number of hard-constraint violations #hcv and soft-constraint violations #scv, which was used in [35], as defined as follows

$$f(s) := \#\text{hcv}(s) * C + \#\text{scv}(s) \tag{1}$$

where C is a constant, which is larger than the maximum possible number of soft-constraint violations.

IV. Investigated GAs for the UCTP

In this section, the GAs that are investigated in this paper to solve the UCTP are described in detail. We first present the framework of these GAs with the integration of LS schemes, and then, describe the recently proposed GSGA and its extension, which is introduced in this paper.

A. Common Framework

GAs are a class of general-purpose optimization tools that model the principles of natural evolution [16]. GAs are population-based heuristic methods, which start from an initial population of randomly generated solutions of a problem. Each solution in a population is called an individual of the population. Each individual is evaluated according to a problem-specific objective function, usually called the fitness function. After evaluation, there is a selection phase in which possibly good individuals are chosen by a selection operator to undergo the recombination process. In the recombination phase, crossover and mutation operators are used to create new individuals in order to explore the solution space. The newly created individuals replace old individuals, usually the worst ones, of the population based on the fitness. This process is repeated until a stopping criterion is reached, which may be the maximum number of generations or a time limit. GAs were first used for timetabling in 1990 [14]. Since then, there have been a number of papers investigating and applying GA methods for the UCTP [13].

Algorithm 1 Steady State Genetic Algorithm (SSGA)

```
1:  randomly initialize a population of solutions
2:  evaluate the individuals in the population
3:  apply local search to each individual of the population
4:  while the termination condition is not reached do
5:      select two parents through tournament selection
6:      create a child by crossover with a probability $P_c$
7:      apply mutation to the child with a probability $P_m$
8:      apply local search to the child
9:      replace the worst member of the population by the child
10: end while
```

The basic framework of GAs investigated in this paper is based on the steady-state GA (i.e., one individual is created during each generation) together with LS schemes, which is denoted as *SSGA* in this paper and is shown in Algorithm 1. With this basic framework, GAs start from an initial population of individual solutions that are randomly generated (i.e., events are randomly assigned to rooms and time slots for each solution). It is reasonable to expect that the quality of the initial solutions would affect the quality of the final solutions. However, we start from random initial solutions in this paper. Then, in each generation, one individual is generated as follows. First, two parents are selected using the tournament selection (of a tournament size 2 in this paper). Then, crossover is carried out with a probability P_c to generate one child via exchanging the time slots between the two selected parents and allocating rooms to events in each nonempty time slot. After crossover, the child undergoes the mutation operation with a probability P_m. The mutation operator first randomly selects one of the three neighborhood structures $N1$, $N2$, and $N3$ (to be described next in this section), and then, makes a move within the selected neighborhood structure. After mutation, LS is performed on the child. Finally, the newly generated child is used to replace the worst individual from the population.

1) LS and Neighborhood Structures: In the basic framework of GAs investigated in this paper, an LS method, as used in [36] and denoted as *LS1* in this paper, is applied on an individual for possible improvement. Algorithm 2 summarizes the LS1 scheme used in the basic framework. LS1 works on all events. Here, we suppose that each event is involved in soft- and hard-constraint violations. LS1 works in two steps and is based on three neighborhood structures $N1$, $N2$, and $N3$, respectively, which are described as follows:

1) *N1:* the neighborhood defined by an operator that moves one event from a time slot to a different one;

Algorithm 2 Local Search Scheme 1 (LS1)

```
1: input : Individual I from the population
2: for each event $e_i \in E$ do
3:   if event $e_i$ is infeasible then
4:     if there is untried move left then
5:       calculate the moves: first N1, then N2 if N1 fails,
         and finally N3 if N1 and N2 fail
6:       apply the matching algorithm to the time slots
         affected by the move to allocate rooms for events
7:       delta evaluate the result of the move
8:       if moves reduce hard constraints violation then
9:         make the moves and go to line 4
10:      end if
11:    end if
12:  end if
13: end for
14: if no any hard constraints remain then
15:   for each event $e_i \in E$ do
16:     if event $e_i$ has soft constraint violation then
17:       if there is untried move left then
18:         calculate the moves: first N1, then N2 if N1 fails,
            and finally N3 if N1 and N2 fail
19:         apply the matching algorithm to the time slots
            affected by the move to allocate rooms for
            events
20:         delta evaluate the result of the move
21:         if moves reduce soft constraints violation then
22:           make the moves and go to line 17
23:         end if
24:       end if
25:     end if
26:   end for
27: end if
28: output : A possibly improved individual I
```

2) *N2:* the neighborhood defined by an operator that swaps the time slots of two events;
3) *N3:* the neighborhood defined by an operator that permutes three events in three distinct time slots in one of the two possible ways other than the existing permutation of the three events.

In the first step (lines 2–13 in Algorithm 2), LS1 checks the hard-constraint violations of each event while ignoring its soft-constraint violations. If there are hard-constraint violations for an event, LS1 tries to resolve them by applying moves in the neighborhood structures $N1$, $N2$, and $N3$ orderly[1] until a stop condition is reached, e.g., an improvement or the maximum number of steps $s_{\max}$ is reached, which is set to different values for different problem instances. After each move, we apply the matching algorithm to the time slots affected by the move and try to resolve the room allocation disturbance and delta evaluate the result of the move.[2] If there is no untried move left in the neighborhood for an event, LS1 continues to the next event. After applying all neighborhood moves on each event, if there is still any hard-constraint violation, then LS1 will stop; otherwise, LS1 will perform the second step (lines 14–27 in Algorithm 2).

In the second step, after reaching a feasible solution, LS1 deals with soft constraints in a similar way, as in the first step. For each event, LS1 tries to make moves in the neighborhood $N1$, $N2$, and/or $N3$ orderly in order to reduce its soft-constraint violations without violating the hard constraints. For each move, the matching algorithm is applied to allocate rooms to affected events and the result is delta-evaluated. When LS1 finishes, we get a possibly improved individual.

B. Guided Search Genetic Algorithm

The GSGA proposed in [23] incorporates a guided search strategy and the aforementioned LS scheme LS1 into the GA to solve the UCTP. The pseudocode of GSGA for the UCTP is shown in Algorithm 3. In GSGA, we first initialize the population by randomly creating each individual via assigning a random time slot for each event according to a uniform distribution and applying the matching algorithm to allocate a room for the event. Then, the LS1 method is applied to each member of the initial population using the three neighborhood structures described in Section IV-A.

Algorithm 3 Guided Search Genetic Algorithm (GSGA)

```
1: input : A problem instance I
2: randomly initialize a population of solutions
3: evaluate the individuals in the population
4: apply the local search scheme LS1 to each individual of
   the population
5: set the generation counter $g := 0$
6: while the termination condition is not reached do
7:   if $(g \bmod \tau) == 0$ then
8:     apply $ConstructMEM()$ to construct $MEM$
9:   end if
10:  create a child using $GuidedSolutionConstruction()$
     or $Crossover()$ with a probability $\gamma$
11:  apply mutation to the child with a probability $P_m$
12:  evaluate the child solution
13:  apply local search LS1 to the child
14:  replace the worst member of the population by the child
15:  $g := g + 1$
16: end while
17: output : The best solution $s_{best}$ achieved for I
```

After the initialization of the population, a data structure (denoted as MEM in this paper) is constructed, which stores a list of room and time slot pairs (r, t) for each event that has a zero penalty (i.e., no hard and soft violation at this event) of individuals selected from the population. After that, MEM can be used to guide the generation of offspring for the following generations. The MEM data structure is reconstructed regularly, e.g., every τ generations. In each generation of GSGA,

[1]For the event being considered, potential moves are calculated in a strict order. First, we try to move the event to the next time slot, then the next, then the next, etc. If this search in $N1$ fails, we then search in $N2$ by trying to swap the event with the next one in the list, then the next one, etc. If the search in $N2$ also fails, we try a move in $N3$ by using one different permutation formed by the event with the next two events, then with the next two, etc.

[2]A delta evaluation means that we only calculate the hard- and soft-constraint violations of those events involved in a move within an individual and work out the change of the objective value of the individual before and after the move accordingly. On the contrast, a complete evaluation of an individual involves the calculation of the hard- and soft-constraint violations of all events in the individual. A delta evaluation is much cheaper than a complete evaluation, and hence, is not counted into the number of evaluations in the experimental study of this paper, as in other works from the literature.

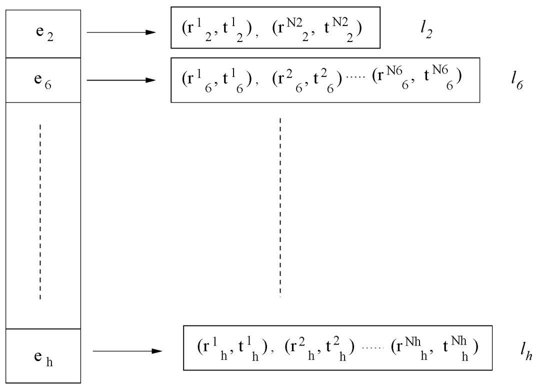

e2
e6
eh
(r¹₂, t¹₂), (r^N2₂, t^N2₂)
(r¹₆, t¹₆), (r²₆, t²₆)..... (r^N6₆, t^N6₆)
(r¹ₕ, t¹ₕ), (r²ₕ, t²ₕ) ..... (r^Nh ₕ, t^Nh ₕ)
l2
l6
lh

Fig. 1. Illustration of the data structure MEM.

one child is first generated either by using MEM or by applying the crossover operator, depending on a probability γ. After that, the child will be mutated by a mutation operator with a probability p_m followed by the LS1 local search operation. Finally, the worst member in the population is replaced by the newly generated child individual. This iteration continues until one termination condition is reached, e.g., a preset time limit $t_{\max}$ is reached.

In the following sections, we describe in detail the key components of GSGA, including the MEM data structure and its construction and the guided search strategy, respectively.

1) MEM Data Structure: There have been a number of researches in the literature on using extra data structure or memory to store useful information in order to enhance the performance of GAs and other metaheuristic methods for optimization and search [4], [29]. In GSGA, we also use a data structure MEM to guide the generation of offspring. This MEM is used to provide further direction of exploration and exploitation in the search space. It aims to increase the quality of a child solution by reintroducing part of best individuals from previous generations. Fig. 1 shows the details of the MEM data structure, which is a two-level list. The first level is a list of events and the second level is a list l_i of room and time slot pairs corresponding to each event e_i in the first level list. In Fig. 1, N_i represents the total number of pairs in the second level list l_i.

The MEM data structure is regularly reconstructed every τ generations. Algorithm 4 shows the outline of constructing MEM. When MEM is due to be reconstructed, we first select $\alpha \times N$ best individuals from the population P to form a set Q, where N denotes the population size. After that, for each individual $I_j \in Q$, each event is checked by its penalty value, i.e., the hard- and soft-constraint violations associated with this event. If an event has a zero penalty value, then we store the information corresponding to this event into MEM. For example, if the event e_2 of an individual $I_j \in Q$ is assigned room 2 at time slot 13 and has a zero penalty value, then we add the pair $(2, 13)$ into the list l_2. Similarly, the events of the next individual $I_{j+1} \in Q$ are also checked by their penalty values. If the event e_2 in I_{j+1} has a zero penalty, then we add the pair of room and time slot assigned to e_2 in I_{j+1} into the existing list l_2. If for an event e_i, there is no list l_i existing yet, then, the list l_i is added into the MEM data structure. Similar process is carried out for the selected Q individuals, and finally, the MEM data structure stores pairs of room and time slot corresponding to those events with zero penalty of the best individuals of the current population.

Algorithm 4 $ConstructMEM()$
```
1:  input : The whole population $P$
2:  sort the population $P$ according to the fitness of individuals
3:  $Q \leftarrow$ select the best $\alpha \times N$ individuals in $P$
4:  for each individual $I_j$ in $Q$ do
5:      for each event $e_i$ in $I_j$ do
6:          calculate the penalty value of event $e_i$ from $I_j$
7:          if $e_i$ is feasible (i.e., $e_i$ has a zero penalty) then
8:              add the room and time slot pair $(r_i, t_i)$ assigned to $e_i$ into the list $l_i$
9:          end if
10:     end for
11: end for
12: output : The data structure $MEM$
```

This MEM data structure is then used to guide the generation of offspring for the next τ generations. We update MEM every τ generations instead of every generation in order to make a balance between the solution quality and the computational time cost of the proposed GSGA.

2) Generating a Child by the Guided Search Strategy: In GSGA, a child is created through the guided search strategy by MEM (see Algorithm 5) or the crossover operator (see Algorithm 6) with a probability γ. When a new child is to be generated, a random number $\rho \in [0.0, 1.0]$ is first generated. If $\rho < \gamma$, Algorithm 5 (i.e., *GuidedSolutionConstruction*()) will be used to generate the new child; otherwise, a crossover operation is used to generate the new child. Next, we first describe the procedure of generating a child through the guided search strategy by MEM, and then, describe the crossover operator.

Algorithm 5 $GuidedSolutionConstruction()$
```
1:  input : The $MEM$ data structure
2:  $E_s :=$ randomly select $\beta * n$ events
3:  for each event $e_i$ in $E_s$ do
4:      randomly select a pair of room and time slot from the list $l_i$
5:      assign the selected pair to event $e_i$ for the child
6:  end for
7:  for each remaining event $e_i$ not in $E_s$ do
8:      assign a random time slot and room to event $e_i$
9:  end for
10: output : A new child generated using $MEM$
```

If a child is to be created using the MEM data structure, we first select a set E_s of $\beta * n$ random events to be generated from MEM. Here, β is a percentage value and n is the total number of events. After that, for each event e_i in E_s, we randomly select a pair of (r_i^j, t_i^j), $j = 1, \ldots, N_i$, from the list l_i that corresponds to the event e_i and assign the selected pair to e_i for the child. If there is an event e_i in E_s, but there is no list l_i in MEM, then we randomly assign a room and time slot from possible rooms and time slots to e_i for the child. This process is carried out

Algorithm 6 $Crossover()$

```
1: input : The current population
2: Select parents $P1$ and $P2$ by the tournament selection
3: for each event $e_i$ of the child $Ch$ do
4:    if $rand(0,1) < 0.5$ then
5:       $e_i$ of $Ch \leftarrow$ the time slot allocated to $e_i$ of P1
6:    else
7:       $e_i$ of $Ch \leftarrow$ the time slot allocated to $e_i$ of P2
8:    end if
9: end for
10: allocate rooms to all occupied time slots using the matching algorithm
11: output : A new child generated using $Crossover()$
```

for all the events in E_s. For those remaining events that are not present in E_s, they are assigned random rooms and time slots.

Algorithm 6 describes the crossover operator. If a child is to be generated using the crossover operator, we first select two individuals from the population as the parents by the tournament selection with the tournament size 2. Then, for each event of the child, we randomly select a parent and allocate the corresponding time slot for that event. Finally, we allocate rooms to events in each nonempty time slot using the matching algorithm.

After a child is generated by using either MEM or crossover, it goes over the mutation operation with a probability P_m, as described earlier. After mutation, we apply the LS scheme LS1 to improve the child individual. Finally, the worst individual in the population is replaced by the new child.

C. Extended GSGA

In this paper, we propose an extended version of GSGA, denoted EGSGA, for the UCTP. In EGSGA, a new LS scheme, denoted *LS2* in this paper and described in Algorithm 7, is introduced and combined with LS1 for GSGA.

Algorithm 7 Local Search Scheme 2 (LS2)

```
1: input : Individual $\mathbf{I}$ after LS1 is applied
2: $S :=$ randomly select a preset percentage of time slots from the total time slots of $T$
3: for each time slot $t_i \in S$ do
4:    for each event $j$ in time slot $t_i$ do
5:       calculate the penalty value of event $j$
6:    end for
7:    sum the total penalty value of events in time slot $t_i$
8: end for
9: select the time slot $w_t$ with the biggest penalty value from $S$
10: for each event $i$ in $w_t$ do
11:    calculate a move of event $i$ in the neighbourhood structure N1
12:    apply the matching algorithm to the time slots affected by the move
13:    compute the penalty of event $i$ and delta evaluate the result
14: end for
15: if all the moves together reduce hard or soft constraint violations then
16:    apply the moves
17: else
18:    delete the moves
19: end if
20: output : A possibly improved individual $\mathbf{I}$
```

In EGSGA, LS2 is used immediately after LS1 on random solutions of the initial population as well as after a child is created through crossover or the MEM data structure and mutation. The basic idea of LS2 is to choose a high-penalty time slot that may have a large number of events involving hard- and soft-constraint violations and try to reduce the penalty values of involved events.

LS2 first randomly selects a preset percentage of time slots[3] (e.g., 20% as used in this paper) from the total time slots of T. Then, it calculates the penalty of each selected time slot[4] and chooses the worst time slot w_t that has the biggest penalty value for LS. After taking the worst time slot, LS2 tries a move in the neighborhood $N1$ for each event of w_t and checks the penalty value of each event before and after applying the move. If all the moves in w_t together reduce the hard- and/or soft-constraint violations, then we apply the moves; otherwise, we do not apply the moves. In this way, LS2 can not only check the worst time slot, but also reduce the penalty value for some events by moving them to other time slots.

In general, LS2 aims to help in improving the solution obtained by LS1. LS2 is expected to enhance the individuals of the population and increase the quality of the feasible timetable by reducing the number of constraint violations.

D. GA With Both LS Schemes (GALS)

In order to investigate the effect of the guided search strategy in GSGAs, in this paper we also investigate a SSGA with two aforementioned LS strategies LS1 and LS2 and without the guided search strategy. This GA is denoted as *GALS* in this paper. In each generation of GALS, one child is first generated through selection, crossover (Algorithm 6), and mutation. Then, two LS methods, LS1 (Algorithm 2) and LS2 (Algorithm 7), are orderly applied to the child. In the end, the worst individual in the population is replaced with this new child. Hence, GALS is an SSGA with the integration of LS2. This will also allow us to check the effect of LS2 on the performance of SSGA for the UCTP. GALS differs from EGSGA with $\gamma = 0.0$ in that in GALS, crossover is carried out with a crossover probability P_c, while in EGSGA with $\gamma = 0.0$ crossover is carried out with a probability 1.0. In other words, EGSGA with $\gamma = 0.0$ is equivalent to GALS with $P_c = 1.0$.

V. Experimental Study

In this section, we experimentally investigate the performance of the proposed methods GALS, GSGA, and EGSGA in addition with SSGA described in Algorithm 1 and a TS method proposed by Rossi-Doria *et al.* [37] for the UCTP. TS is a powerful tool for solving difficult optimization problems. It is a local-area-based algorithm that has specialized memory structures to avoid being trapped in local minima and achieve an effective balance of intensification and diversification [37]. A TS method maintains a list of tabu moves that represent timetables, which have been

[3]Rather than choosing the worst time slot out of all the time slots, we randomly select a set of time slots, and then, choose the worst time slot. This is because, for each selected time slot, we need to calculate its penalty value, which is time consuming. By selecting a set of time slots instead of all time slots, we try to balance between the computational time and the quality of the algorithm.

[4]The penalty of a time slot is the sum of the penalty values of all the events that occur in the time slot.

TABLE I
THREE GROUPS OF PROBLEM INSTANCES

Class	Small	Medium	Large
Number of events	100	400	400
Number of rooms	5	10	10
Number of features	5	5	10
Approximate features per room	3	3	5
Percentage (%) of features used	70	80	90
Number of students	80	200	400
Maximum events per student	20	20	20
Maximum students per event	20	50	100

visited recently and are forbidden, in order to prevent the search from staying in the same region, and thus, enable the TS method to escape from local optima [10].

All algorithms were coded in GNU C++ under version 4.1 and run on a 3.20 GHz PC. We use a set of benchmark UCTP instances to test the algorithms, which were proposed in [37] for the timetabling competition.[5] Although these problem instances lack many of the real-world problem constraints and issues [33], they allow the comparison of our approaches with current state-of-the-art techniques on them. Table I presents the data of these UCTP instances in three different groups: five small instances, five medium instances, and one large instance. As mentioned earlier, the basic framework of all GAs we investigate (SSGA, GALS, GSGA, and EGSGA) is a steady-state GA. The basic parameters for GAs were set as follows: the population size N was set to 50, the mutation probability P_m was set to 0.5, and the crossover probability P_c (only used in SSGA and GALS) was set to 0.8. The value of the constant C in the objective function was set to 10^6.

Two sets of experiments were carried out in this study. The first set of experiments are devoted to analyze the sensitivity of parameters for the performance of GSGA for the UCTP. The second set of experiments compare the performance of investigated GAs with or without the guided search strategy on the test UCTPs. For both sets of experiments, there were 50 runs of each algorithm on each problem instance. Following other works [37], [39], for each run of an algorithm on a problem, the maximum run time $t_{\max}$ was set to 90 s for small instances, 900 s for medium instances, and 9000 s for the large instance based on the fact that larger UCTP instances are more complex and have more conflicting constraints and a larger search space as compared to smaller UCTP instances, and therefore, requires more processing time.

In the end, we compare our experimental results of EGSGA with a set of current state-of-the-art methods from the literature on the aforementioned set of test UCTP instances and another set of UCTP instances taken from the 2002 International Timetabling Competition (TTC 2002).[6]

A. Sensitive Analysis of Key Parameters of GSGA

The performance of the guided search strategy method depends on the parameters and operators used. Through our previous work [23], we found that α, β, γ, and τ are key parameters

[5]For details, see http://iridia.ulb.ac.be/supp/IridiaSupp2002-001/index.html.
[6]For details, see http://www.idsia.ch/Files/ttcomp2002/oldindex.html.

TABLE II
PARAMETER SETTINGS IN GSGA

Parameter	Settings					
α	0.2	0.4	0.6	0.8		
β	0.1	0.3	0.5	0.7	0.9	
γ	0.0	0.2	0.4	0.6	0.8	1.0
τ	20	40	60	80	100	

that can greatly affect the performance of GSGA for the UCTP, where α is the percentage of best individuals selected from the current population for creating the data structure MEM, β is the percentage value of the total number of events that are used to create a child through the data structure MEM, γ is the probability that indicates whether a child is created through MEM or crossover, and τ decides the frequency of updating MEM (i.e., MEM is updated every τ generations). Hence, we test our algorithm GSGA with different settings of these parameters.

Table II shows different parameters and their settings that are tested in our experiments. In order to find out which parameter settings have a great effect on the performance of GSGA, we run GSGA 50 times for all parameter combinations in Table II. In Total, 600 combinations of parameter settings were observed. Here, we only present some of them that seem to have a great effect on the performance of GSGA. We chose two α values 0.2 and 0.6, three β values 0.1, 0.3, and 0.7, three γ values 0.2, 0.4, and 0.8, and two τ values 20 and 60. The experimental results with respect to the average objective value of GSGA with these selected parameter settings are presented in Table III. In Table III (and other tables and figures in this paper), "S1" represents small problem instance 1, "S2" represents small problem instance 2, etc., "M1" represents medium problem instance 1, "M2" represents medium problem instance 2, etc., and "L" represents the large problem instance.

In order to help to understand the experimental results, Fig. 2 shows the dynamic performance regarding the average objective value against the number of evaluations over 50 runs of GSGA with one parameter changing while the other parameters kept constant on different UCTP instances. Fig. 2(a) shows the effect of changing α on M1 with $\beta = 0.3$, $\gamma = 0.8$, and $\tau = 20$. Fig. 2(b) shows the effect of changing β on S2 with $\alpha = 0.2$, $\gamma = 0.8$, and $\tau = 20$. Fig. 2(c) shows the effect of changing γ on S5 with $\alpha = 0.2$, $\beta = 0.3$, and $\tau = 20$. Fig. 2(d) shows the effect of changing τ on S4 with $\alpha = 0.2$, $\beta = 0.3$, and $\gamma = 0.8$. From Table III and Fig. 2, several results can be observed and are analyzed next.

First, the parameter α has a significant effect on the performance of GSGA for the UCTP. The performance of GSGA drops when the value of α increases from 0.2 to 0.8 (see Fig. 2(a) for reference). This occurs because when we choose a small part of the population to create the MEM data structure, MEM can provide a strong guidance during the genetic operations and help GSGA to exploit the area of the search space that corresponds to the best individuals of the population sufficiently. This sufficient exploitation can ensure that GSGA quickly achieves better solutions. In the contrast, when a large part of the population is taken to create or update MEM, then MEM will lose its effect of guiding GSGA to exploit promising areas of the search

TABLE III
AVERAGE BEST VALUE OF 50 RUNS OF GSGA WITH DIFFERENT PARAMETER SETTINGS ON THE TEST PROBLEM INSTANCES

α	β	γ	τ	S1	S2	S3	S4	S5	M1	M2	M3	M4	M5	L
0.2	0.1	0.2	20	6	9	5	9	4	221	146	211	178	126	726
0.2	0.1	0.2	60	11	10	11	21	5	247	148	217	178	142	803
0.2	0.1	0.4	20	2	3	6	14	3	271	101	192	143	112	716
0.2	0.1	0.4	60	6	10	4	11	3	225	133	201	166	114	806
0.2	0.1	0.8	20	0	2	4	3	1	192	110	203	98	107	645
0.2	0.1	0.8	60	2	3	5	4	0	195	125	221	101	115	669
0.2	0.3	0.2	20	7	6	8	4	4	178	126	183	116	108	722
0.2	0.3	0.2	60	11	11	10	9	4	190	130	187	163	126	812
0.2	0.3	0.4	20	2	6	4	3	4	176	119	178	153	113	647
0.2	0.3	0.4	60	6	4	8	4	5	182	124	185	161	118	654
0.2	0.3	0.8	20	0	0	1	2	0	152	108	121	101	96	637
0.2	0.3	0.8	60	1	2	1	1	2	161	109	155	121	111	675
0.2	0.7	0.2	20	8	6	8	4	5	181	172	162	321	139	856
0.2	0.7	0.2	60	12	11	8	4	5	192	210	175	343	142	881
0.2	0.7	0.4	20	14	7	13	6	4	183	161	154	219	128	778
0.2	0.7	0.4	60	9	2	9	5	5	190	184	162	232	132	811
0.2	0.7	0.8	20	2	0	1	3	5	170	139	218	123	116	664
0.2	0.7	0.8	60	8	5	6	5	6	191	142	225	135	118	679
0.6	0.1	0.2	20	11	8	21	13	7	322	152	223	181	128	834
0.6	0.1	0.2	60	13	15	24	13	8	340	156	231	182	162	888
0.6	0.1	0.4	20	22	17	12	14	3	271	101	192	143	112	716
0.6	0.1	0.4	60	12	10	6	11	3	228	142	211	164	117	764
0.6	0.1	0.8	20	5	3	4	6	5	196	111	220	101	112	652
0.6	0.1	0.8	60	11	12	8	6	3	201	122	247	121	126	689
0.6	0.3	0.2	20	11	10	9	8	5	181	127	198	132	128	722
0.6	0.3	0.2	60	11	11	10	9	4	190	130	200	163	126	812
0.6	0.3	0.4	20	7	7	11	9	5	179	125	181	173	153	647
0.6	0.3	0.4	60	9	11	17	10	5	184	129	185	219	164	712
0.6	0.3	0.8	20	2	4	3	5	0	156	113	132	116	103	645
0.6	0.3	0.8	60	5	3	4	6	3	159	128	161	124	123	663
0.6	0.7	0.2	20	13	21	12	9	7	209	180	167	245	184	912
0.6	0.7	0.2	60	15	21	21	22	9	234	194	197	289	190	934
0.6	0.7	0.4	20	14	9	11	8	7	189	175	161	189	134	781
0.6	0.7	0.4	60	14	11	13	9	10	211	191	186	194	143	792
0.6	0.7	0.8	20	5	4	6	4	5	163	151	192	126	112	670
0.6	0.7	0.8	60	6	4	8	6	10	178	172	212	129	128	705

space. In other words, when α is set to large values, GSGA tends to be SSGA, and hence, the performance will drop or become weak. This can be observed from Fig. 2(a): when the value of α increases, the best solution of GSGA cannot improve after a certain number of evaluations, e.g., after about 4000 evaluations when $\alpha = 0.6$ and after about 1500 evaluations when $\alpha = 0.8$.

Second, regarding the effect of β, it can be seen that setting this parameter to a very small or very large value affects the penalty value. This result can be observed from the interesting behavior of GSGA on the S2 problem instance with $\alpha = 0.2$, $\gamma = 0.8$, and $\tau = 20$, and different β values in Fig. 2(b). From Fig. 2(b), it can be seen that when the value of β increases from 0.1 to 0.3, the performance of GSGA improves due to the enhanced effect of the MEM data structure. However, when the value of β is further raised, the performance of GSGA drops. This occurs because if a large portion of individuals is created through MEM, e.g., when $\beta = 0.9$, the chance of creating a similar child may be increased every generation, and after a few generations, GSGA may be trapped in a suboptimal state, and hence, cannot obtain the optimal solution. From Fig. 2(b), it can be seen that by setting the value of β to 0.7 or 0.9 leads to an earlier stagnation in the performance of GSGA in the solving process.

Third, regarding the effect of γ on the performance of GSGA, from Table III, it can be easily seen that increasing the value of γ results in near-optimal solutions. The reason lies in the fact that a small value of γ leads to the proposed GSGA algorithm acting as the conventional SSGA. Fig. 2(c) shows the behavior of GSGA on the S5 problem instance with $\alpha = 0.2$, $\beta = 0.3$, and $\tau = 20$, and different values of γ. It is quite obvious that a large value of γ, e.g., $\gamma = 0.8$, leads to an optimal solution quickly. The effect of γ also shows the importance of the MEM data structure. From Fig. 2(c), it can also be seen that when $\gamma = 1.0$, the performance of GSGA drops in comparison with when $\gamma = 0.8$. This result shows that the use of crossover helps to improve the performance of GSGA for the UCTP.

Fourth, regarding the effect of τ, it can be seen from Table III and Fig. 2(d) that setting τ to 20 gives a better objective value than setting τ to other values (i.e., 40, 60, 80, and 100). Hence, updating the MEM data structure every small number of generations gives a better performance of GSGA. This is because when MEM is updated more frequently, the information extracted from the best individuals of the population can be more timely used to guide the generation of offspring, and hence, gives a greater chance to create better individuals. The difference is significant when τ is set to 20 over 100. The effect of τ also shows the importance of the MEM data structure for the performance of GSGA.

Based on the aforementioned parameter analyses, in the following experiments, we set the parameters for GSGA and EGSGA as follows: $\alpha = 0.2$, $\beta = 0.3$, $\gamma = 0.8$, and $\tau = 20$.

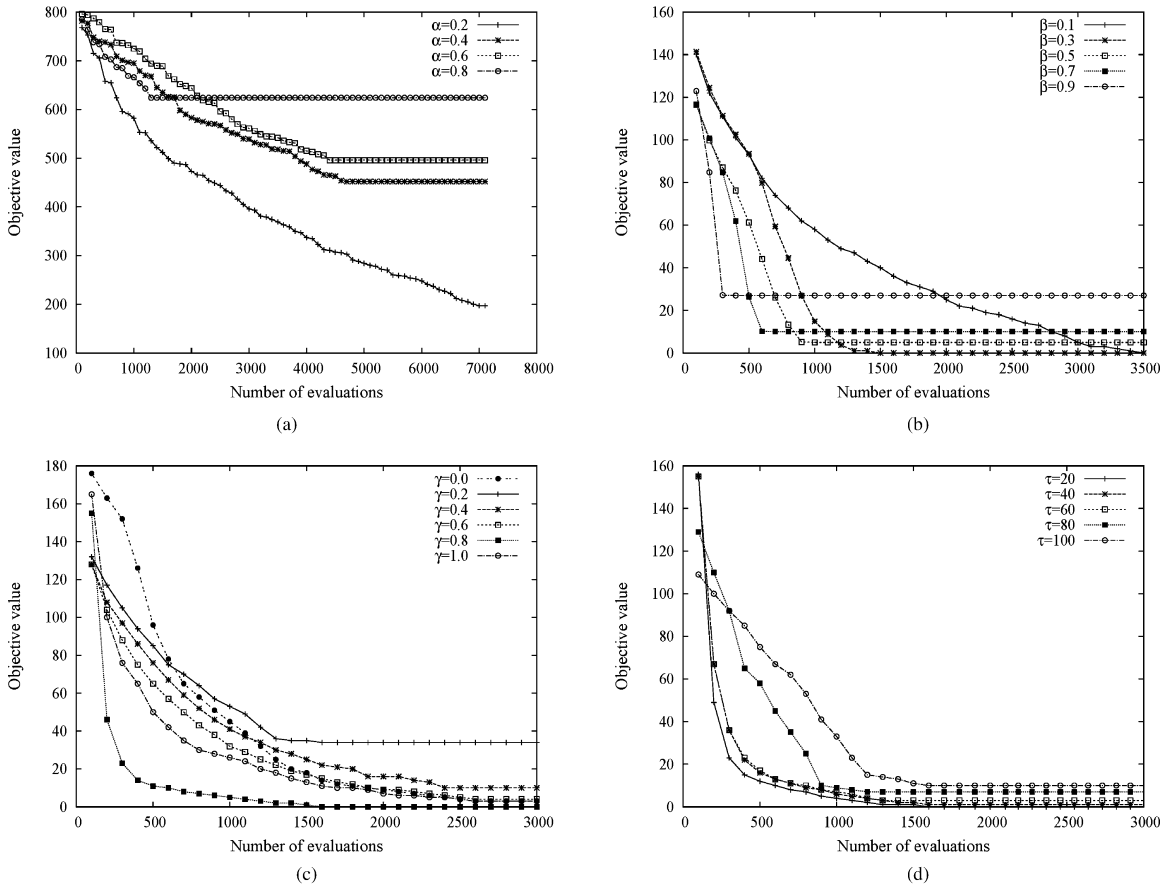


Fig. 2. Comparison on the effect of parameters on the performance of GSGA on different problem instances. (a) M1 with $\beta = 0.3$, $\gamma = 0.8$, and $\tau = 20$. (b) S2 with $\alpha = 0.2$, $\gamma = 0.8$, and $\tau = 20$. (c) S5 with $\alpha = 0.2$, $\beta = 0.3$, and $\tau = 20$. (d) S4 with $\alpha = 0.2$, $\beta = 0.3$, and $\gamma = 0.8$.

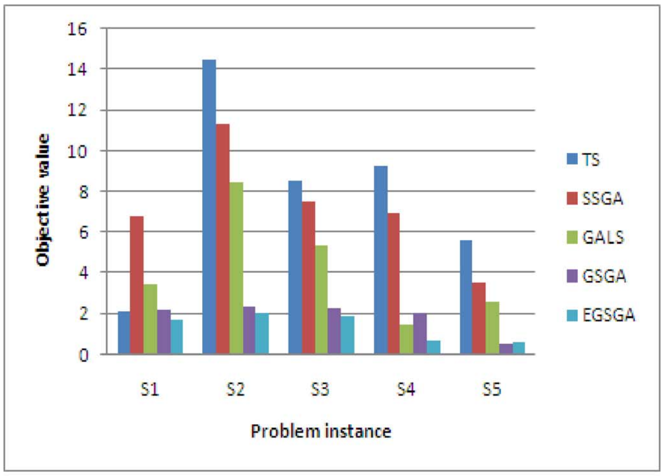
B. *Comparative Experiments*

This set of experiments compares the performance of EGSGA with our implemented algorithms (TS, SSGA, GALS, and GSGA). The parameter settings identified by the previous experiments were used for all results presented in this section. The same set of parameters were used for GAs in order to have a fair comparison of the performance of algorithms.

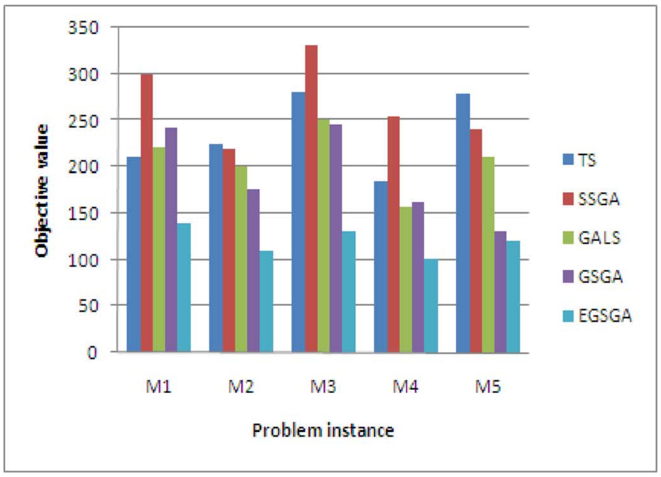
Fig. 3 presents the comparison of EGSGA with other algorithms with respect to the average performance over 50 runs on small and medium test UCTP instances. Table IV compares all algorithms in terms of the best, average, standard deviation, and worst penalty value over the 50 runs on the 11 problem instances, where "–" means that no feasible solution was found by the corresponding method. From Fig. 3 and Table IV, it can be seen that EGSGA produces a lower average and standard deviation of the objective value on most of the UCTP instances and that the worst objective values produced by EGSGA are by far the best among all algorithms. This is a really good result, which means that EGSGA is much more reliable than the others. EGSGA produces good solutions due to the usage of the MEM data structure and LS schemes. As mentioned earlier, this is due to the fact that we assign to an event a pair of room and time slot that was extracted from one of the best individuals of a previous population. This means that the pair satisfies different constraints that are suitable to that event. The LS technique further helps EGSGA to find the local optimum of an individual. By doing so, we increase the chance of getting better and better solutions during the solving process.

Fig. 4 shows the dynamic behavior of algorithms against the number of evaluations on some problem instances, where the *x*-axis represents the number of evaluations and the *y*-axis represents the average objective value over 50 runs. Fig. 4(a) and (b) show the performance of different algorithms on small UCTP instances S1 and S3, respectively. Fig. 4(c) represents the performance of algorithms on the medium problem instance M5, where the *y*-axis shows the objective value expressed in the log scale. Fig. 4(d) shows the performance of GSGA and EGSGA on the large problem instance, where the *y*-axis is also expressed in the log scale.

From Fig. 4, it can be seen that on the small instances, SSGA and TS reach near-optimal solutions as the number of evaluations increases and that GALS and GSGA perform similar to each other. We notice that when the two techniques in EGSGA are used independently, GALS and GSGA are not significantly better than each other, but when the two techniques are combined in EGSGA, we see a great improvement in the performance of EGSGA. The penalty value of EGSGA is reduced at the beginning of the search and gives the optimal solution between 1000 and 1500 evaluations, while GSGA and GALS give near-optimal solutions after 2000 evaluations. On the M5 medium



(a)



(b)

Fig. 3. Comparison of EGSGA with other algorithms regarding the average performance on (a) small instances and (b) medium instances.

TABLE IV
COMPARISON OF ALGORITHMS ON DIFFERENT PROBLEM INSTANCES

UCTP	Alg	Best	Ave	Std	Worst
S1	TS	0	2.06	3.35	9
	SSGA	0	6.75	8.27	15
	GALS	0	3.43	5.12	15
	GSGA	0	2.11	3.33	9
	EGSGA	0	1.71	2.42	4
S2	TS	1	14.5	9.6	27
	SSGA	3	11.3	8.66	32
	GALS	0	8.43	5.21	19
	GSGA	0	2.32	5.59	16
	EGSGA	0	2.01	3.71	11
S3	TS	0	8.53	7.56	22
	SSGA	0	7.26	5.40	26
	GALS	0	5.32	6.60	19
	GSGA	0	2.2	3.21	11
	EGSGA	0	1.8	1.53	2
S4	TS	2	9.26	7.34	22
	SSGA	0	6.81	7.01	24
	GALS	0	1.24	2.41	7
	GSGA	0	1.84	2.20	11
	EGSGA	0	0.63	1.89	5
S5	TS	0	5.58	6.42	16
	SSGA	0	3.49	7.00	19
	GALS	0	2.53	2.89	7
	GSGA	0	0.51	1.86	5
	EGSGA	0	0.55	0.82	3
M1	TS	211	220.5	27.64	267
	SSGA	280	302	36.117	321
	GALS	227	229.5	10.65	256
	GSGA	240	247	9.02	260
	EGSGA	139	142	6.384	202
M2	TS	185	230.5	21.59	273
	SSGA	188	225.2	20.01	290
	GALS	180	203	20.62	256
	GSGA	162	172.4	14.49	209
	EGSGA	92	112	10.96	134
M3	TS	280	286	8.170	301
	SSGA	249	330	23.45	389
	GALS	235	249.2	10.21	300
	GSGA	242	247	6.021	290
	EGSGA	122	128.4	4.832	160
M4	TS	176	187	18.38	241
	SSGA	247	256	21.86	321
	GALS	142	160.1	16.90	203
	GSGA	158	162.7	17.01	212
	EGSGA	98	100.2	5.451	112
M5	TS	255	276	20.45	365
	SSGA	232	245.6	15.32	343
	GALS	200	212.2	24.77	298
	GSGA	124	128.5	23.67	200
	EGSGA	116	121.3	13.29	151
L	TS	–	–	–	–
	SSGA	–	–	–	–
	GALS	–	–	–	–
	GSGA	801	858.2	40.35	921
	EGSGA	615	648.5	19.11	670

problem instance, a great fall in the penalty value can be noticed in the performance of EGSGA. EGSGA quickly generates a feasible solution after a few evaluations and makes positive movement toward the near-optimal solution by exploring the search space as the number of evaluations increases. On the other hand, GALS and GSGA achieve a feasible solution over 1000 evaluations. It can be observed from Fig. 4(d) that the search speed of EGSGA on the large problem instance is better than that of GSGA. We anticipate this result because partial solutions from good individuals provide more efficient solutions when combined with the LS technique LS2.

The t-test results of statistically comparing investigated algorithms are shown in Table V. The t-test statistical comparison was carried out with 98 DOF at a 0.05 level of significance. In Table V, the t-test results of comparing two algorithms is shown as "$s+$," "$s-$," "$+$," or "$-$" when the first algorithm is significantly better than, significantly worse than, insignificantly better than, or insignificantly worse than the second algorithm, respectively. In Table V, "In" means that one or both of the algorithms being compared failed to find a feasible solution of the corresponding problem.

From Table V, it can be seen that the performance of EGSGA is significantly better than the performance of all other algorithms on all medium and large problem instances and that the performance of EGSGA is significantly better than the performance of GSGA on most small, medium, and large problem instances. These results show that the integration of proper LS techniques with the guided search strategy can greatly improve the performance of GAs for the UCTP.

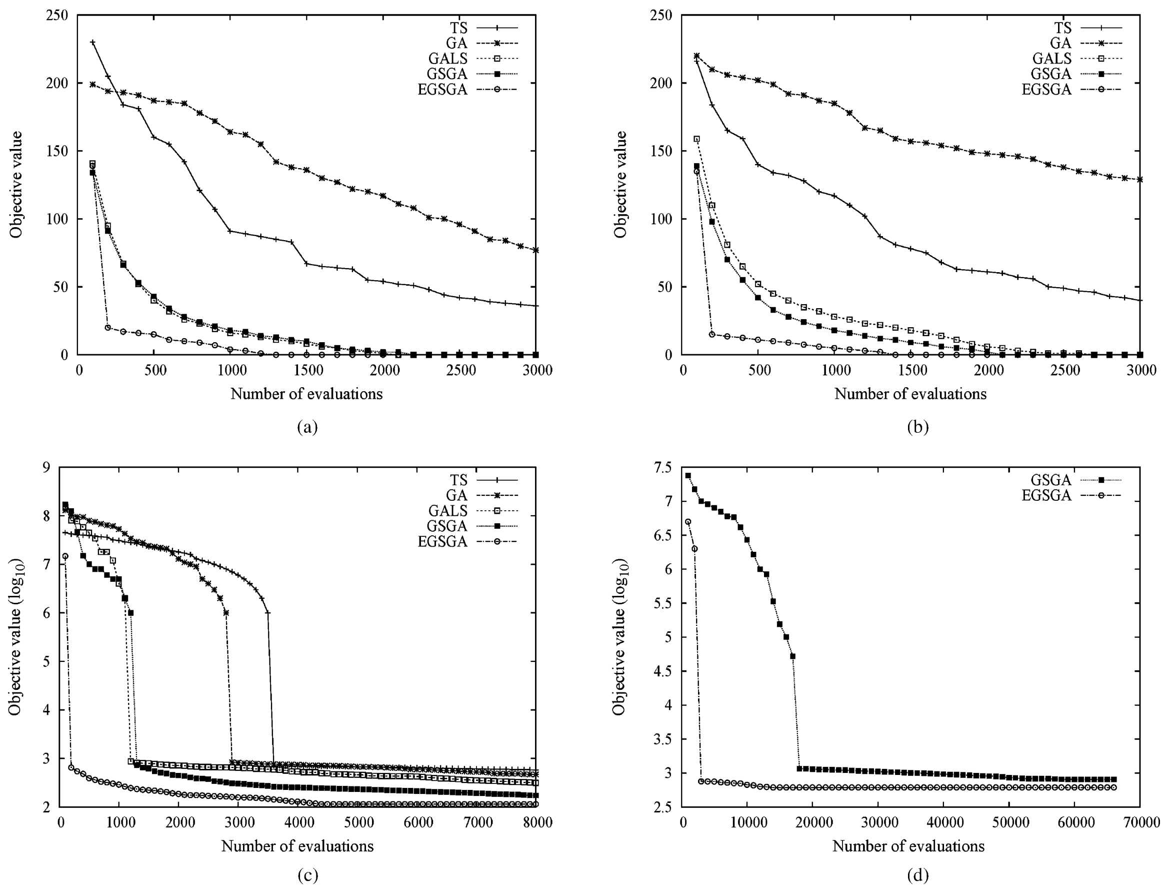


Fig. 4. Dynamic performance of algorithms on different problem instances: (a) S1, (b) S3, (c) M5, and (d) L.

TABLE V
t-TEST VALUES OF COMPARING ALGORITHMS ON DIFFERENT PROBLEM INSTANCES

UCTP	S1	S2	S3	S4	S5	M1	M2	M3	M4	M5	L
SSGA − TS	$s-$	$s+$	$+$	$s+$	$s+$	$s-$	$+$	$s-$	$s-$	$s+$	In
SSGA − GALS	$s-$	$s-$	$s-$	$s-$	$-$	$s-$	$-$	$s-$	$s-$	$s-$	In
SSGA − GSGA	$s-$	$s-$	$s-$	$s-$	$s-$	$s-$	$s-$	$s-$	$s-$	$s-$	In
SSGA − EGSGA	$s-$	$s-$	$s-$	$s-$	$s-$	$s-$	$s-$	$s-$	$s-$	$s-$	In
GALS − GSGA	$-$	$s-$	$s-$	$s+$	$s-$	$s+$	$s-$	$+$	$+$	$s-$	In
EGSGA − GALS	$s+$	$s+$	$s+$	$+$	$s+$	$s+$	$s+$	$s+$	$s+$	$s+$	In
EGSGA − GSGA	$+$	$+$	$s+$	$s+$	$+$	$s+$	$+$	$s+$	$s+$	$+$	$s+$
TS − GALS	$-$	$s-$	$s-$	$s-$	$s-$	$s+$	$s-$	$s-$	$s-$	$s-$	In
TS − GSGA	$-$	$s-$	$s-$	$s-$	$s-$	$s+$	$-$	$s-$	$-$	$s-$	In
TS − EGSGA	$-$	$s-$	$s-$	$s-$	$s-$	$s-$	$s-$	$s-$	$s-$	$s-$	In

C. *Comparison With Other Algorithms From the Literature*

In this section, we compare the experimental results of our algorithms with the available results of other algorithms in the literature on the 11 aforementioned test UCTP instances. The algorithms compared and the conditions under which their results were reported are described as follows.

1) *EGSGA:* The EGSGA proposed in this paper. The results reported here were out of 50 runs with each run lasting for 90 s for small UCTP instances, 900 s for medium instances, and 9000 s for the large instance.
2) *Randomized Iterative Improvement Algorithm (RIIA):* Abdullah *et al.* [1] proposed the RIIA. They presented a composite neighborhood structure with the RIIA . The results were reported out of five runs with each run lasting for 200 000 evaluations.
3) *Nonlinear Great Deluge (NLGD):* Landa-Silva and Obit [26] proposed a modified great deluge algorithm by using a nonlinear decay of water level. They successfully improved the performance of the great deluge algorithm on medium UCTP instances. The results were reported out of ten runs with each run lasting for 3600 s for small UCTP instances, 4700 s for medium instances, and 6700 s for the large instance.
4) *GA With LS (GAWLS):* Abdullah and Turabieh [3] proposed GAWLS. They tested a GA with a repair function and LS on the UCTP. The results were reported out of five runs and the stopping criterion for each run was not clearly mentioned in the paper.
5) *Hybrid EA (HEA):* Abdullh *et al.* [2] proposed HEA that consists of an EA that uses a light mutation operator followed by an RIIA. The results were reported out of five runs with 200 000 evaluations per run.
6) *Graph-Based Hyperheuristic (GBHH):* Burke *et al.* [11] proposed GBHH. They employed TS with GBHHs for the UCTP and examination of timetabling problems. The results were reported out of five runs with 12 000 evaluations per run for small UCTP instances, 1200 evaluations per run for medium instances, and 5400 evaluations per run for the large instance.
7) *TS Hyperheuristics (THHS):*Burke *et al.* [9] introduced a TSHH for the UCTP, where a set of low-level heuristics

TABLE VI
COMPARISON OF ALGORITHMS ON DIFFERENT PROBLEM INSTANCES

	EGSGA		RIIA	NLGD	GAWLS	HEA	GBHH	THHS	LS	GA	AA	FA
UCTP	Best	Med	Best	Best	Best	Best	Best	Best	Med	Best	Med	Best
S1	**0**	0	**0**	3	2	0	6	1	8	**0**	1	10
S2	**0**	0	**0**	4	4	0	7	2	11	3	3	9
S3	**0**	0	**0**	6	2	0	3	**0**	8	**0**	1	7
S4	**0**	0	**0**	6	**0**	0	3	1	7	**0**	1	17
S5	**0**	0	**0**	**0**	4	0	4	**0**	5	**0**	**0**	7
M1	**139**	143	242	140	254	221	372	146	199	280	195	243
M2	**92**	96.5	161	130	258	147	419	173	202.5	188	184	325
M3	**122**	124	265	189	251	246	359	267	77.5%In	249	248	249
M4	**98**	101	181	112	321	165	348	169	177.5	247	164.5	285
M5	**116**	119.5	151	141	276	135	171	303	100%In	232	219.5	132
L	615	622.5	100%In	876	1027	**529**	1068	80%In	100%In	100%In	851.5	1138

compete with each other. This approach was tested on the course timetabling and nurse rostering problems. The results were reported out of five runs with 12 000 evaluations per run for small UCTP instances, 1200 evaluations per run for medium instances, and 5400 evaluations per run for the large instance.

8) *LS:* Socha *et al.* [39] introduced an LS method. They used a random restart LS method for the UCTP and compared it with an ant algorithm (AA). The results were reported out of 50 runs with each run lasting for 90 s for small UCTP instances, 40 runs with each run lasting for 900 s for medium instances, and 10 runs with each run lasting for 9000 s for the large instance.
9) *GA:* Rossi-Doria *et al.* [37] proposed a GA. They used a LS method with the GA to solve the UCTP and also compared several metaheuristics methods on the UCTP. The results were reported out of 50 runs with each run lasting for 90 s for small problem instances, 900 s for medium instances, and 9000 s for the large instance.
10) *AA:* Socha *et al.* [39] used AA. They developed an ant-colony optimization algorithm with the help of a construction graph and a pheromone model appropriate for the UCTP. The results were reported out of 50 runs with each run lasting for 90 s for small UCTP instances, 40 runs with each run lasting for 900 s for medium instances, and 10 runs with each run lasting for 9000 s for the large instance.
11) *Fuzzy Algorithm (FA):* Asmuni *et al.* [7] proposed FA. Asmuni *et al.* [7] focused on the issue of ordering events by simultaneously considering three different heuristics using fuzzy methods. The results were reported out of one run for each problem instance, and the stopping criterion for each run on a problem instance was not clearly mentioned in the paper.

One thing to notice is that the aforementioned algorithms compared and the conditions under which their results were reported have been frequently used by other researchers to compare the performance of their algorithms. Strictly speaking, it is not fully fair to use the results reported in the literature, since the conditions involved are not the same for all algorithms. However, the results reported can give us a rough understanding on how good or bad an algorithm is in comparison with existing methods. Hence, we also follow the trend in the literature and roughly compare our algorithm EGSGA with the aforementioned state-of-the-art methods using the reported results.

Table VI gives the comparison results, where the term “%In” represents the percentage of runs that failed to obtain a feasible solution. “Best” and ‘med” mean the best and median result among 50 runs, respectively. We present the best result among all algorithms for each UCTP instance in the bold font. From Table VI, it can be seen that EGSGA performs better than the FA [7], LS [39], and graph-based approach [11] on all the 11 problem instances. EGSGA outperforms RIIA [1], and GA [37] on all the medium problem instances and ties them on some or all of the small problem instances. EGSGA also gives better results than the AA [39] on ten problem instances and ties it on S5. When comparing with the tabu-based hyperheuristic search [9], EGSGA performs better or the same on all the problem instances. The results of our approach is better than GAWLS [3] and NLGD [26] on all medium and large instances and ties them on one small instance. Finally, the result of EGSGA is better than that of HEA [2] on all medium problem instances and ties it on small instances. On the whole, EGSGA beats all algorithms on medium problem instances and gives promising results on the large problem.

We also run EGSGA according to the TTC 2002 rules on the set of 20 UCTP instances. There were 50 runs of EGSGA on each problem instance. For each run on a problem instance, the maximum run time $t_{\max}$ was set to 900 s. The results of the algorithms we compared are taken from the TTC 2002 Website. These algorithms are briefly described as follows.

1) *SA:* Kostuch [25] proposed an SA-based heuristic. This approach is divided into two stages. First, it finds a feasible timetable. Second, it uses an SA scheme to improve the timetable, according to an objective function value.
2) *Efficient Timetabling Solution (ETTS):* Cordeau *et al.* [15] proposed ETTS with TS. They developed a tabu heuristic that first finds a feasible solution, and then, improves the quality of the solution by reducing soft constraints.
3) *Great Deluge LS (GDLS):* Bykov [12] used a GDLS algorithm to solve the problem.
4) *Three Stage LS (TSLS):* Gaspero and Schaerf [20] used a TSLS paradigm. Their LS method consists of hill climbing, TS, and multiswap shake stages.
5) *Adaptive Memory LS (AMLS):* Arntzen and Løkketangen [6] proposed a simple AMLS method to improve the

TABLE VII
COMPARISON OF ALGORITHMS ON TTC 2002 PROBLEM INSTANCES

UCTP	EGSGA	SA	ETTS	GDLS	TSLS	AMLS	DTS
comp01	54	**45**	61	85	63	132	148
comp02	**25**	**25**	39	42	46	92	101
comp03	**44**	65	77	84	96	170	162
comp04	132	**115**	160	119	166	265	350
comp05	97	102	161	**77**	203	257	412
comp06	**3**	13	42	6	92	133	246
comp07	**12**	44	52	**12**	118	177	228
comp08	**23**	29	54	32	66	134	125
comp09	21	**17**	50	184	51	139	126
comp10	**53**	61	72	90	81	148	147
comp11	46	**44**	53	73	65	135	144
comp12	96	107	110	**79**	119	290	182
comp13	**69**	78	109	91	160	251	192
comp14	**13**	52	93	36	197	230	316
comp15	35	**24**	62	27	114	140	209
comp16	**12**	22	34	300	38	114	121
comp17	104	86	114	**79**	212	186	327
comp18	39	**31**	38	39	40	87	98
comp19	63	**44**	128	86	185	256	325
comp20	2	7	26	**0**	17	94	185

quality of an initial solution. The search is guided by TS mechanisms based on recency and frequency of certain attributes of previous moves.

6) *Dubourg et al. TS (DTS):* Dubourg *et al.* [17] proposed a TS approach to solve the UCTP.

Table VII shows the comparison of EGSGA with other results from the literature on the TTC 2002 test set, where "comp01" represents the first benchmark instance, "comp02" represents the second benchmark instance, etc. Table VII shows the best result (soft-constraints violation) achieved by each algorithm, where the best result among all algorithms for each UCTP instance is represented in the bold font. From Table VII, it can be seen that EGSGA is able to produce good results on all problem instances. It gives the best result on nine out of 20 problem instances. But, there is still a chance of improvement of the proposed approach to get an optimal solution on hard problem instances.

From the aforementioned experimental results, it can be seen that the guided search strategy and proper LS techniques used in GSGAs can help to minimize the objective function value and give better results for the UCTP compared to other population-based algorithms employed in the literature. The experimental results also shows that due to the good solutions that are created through the MEM data structure in GSGAs, the chance of getting feasible and optimal solutions is increased.

VI. Conclusion and Future Work

This paper presents a GSGA to solve the UCTP, where a guided search strategy and LS techniques are integrated into a steady-state GA. The guided search strategy uses a data structure to store useful information, i.e., a list of room and time slot pairs for each event that is extracted from the best individuals selected from the population and has a zero penalty value. This data structure is used to guide the generation of offspring into the following populations. The main advantage of this data structure lies in that it improves the quality of individuals by storing part of former good solutions, which otherwise would have been lost in the selection process, and reusing the stored information in the following generations. This can enable the algorithm to quickly retrieve the best solutions corresponding to the previous and new populations. In the original GSGA [23], a LS technique is used to improve the quality of individuals through searching three neighborhood structures. This paper also proposes an EGSGA for the UCTP by adding another LS method into GSGA.

In order to test the performance of GSGAs for the UCTP, experiments were carried out to analyze the sensitivity of parameters and the effect of the guided search strategy for the performance of GSGAs based on a set of benchmark UCTP instances. The experimental results of EGSGA were also compared with several state-of-the-art methods from the literature on the tested UCTP instances. The experimental results show that the proposed GSGA is competitive and works reasonably well across all problem instances in comparison with other approaches studied in the literature. Generally speaking, with the help of the guided search and LS strategies, EGSGA is able to efficiently find optimal or near-optimal solutions for the UCTP, and hence, can act as a powerful tool for the UCTP.

To our knowledge, this study is the first time to apply guided search GAs to address timetabling problems. There are several works to pursue in the future. One future work will be to further analyze the neighborhood techniques toward the performance of EGSGA. We also intend to test our approach on the 2007 timetabling competition benchmarks, in particular, and other problem instances that are available in the literature, in general. Improvement of genetic operators and new neighborhood techniques based on different problem constraints will also be investigated. We believe that the performance of GSGAs for the UCTP can be improved by applying advanced genetic operators and heuristics. For example, using adaptive techniques to adjust the key parameters of GSGAs with the searching progress may further improve the performance of GSGAs for the UCTP. The understanding of the interrelationship of these techniques and a proper placement of these techniques in GSGA may lead to better performance of GSGAs for the UCTP.

References

[1] S. Abdullah, E. K. Burke, and B. McCollum, "Using a randomized iterative improvement algorithm with composite neighbourhood structures," in *Proc. 6th Int. Conf. Metaheuristic*, 2007, pp. 153–169.

[2] S. Abdullah, E. K. Burke, B. McCollum, and H. Turabieh, "A hybrid evolutionary approach to the university course timetabling problem," in *Proc. 2007 Congr. Evol. Comput.*, 2007, pp. 1764–1768.

[3] S. Abdullah and H. Turabieh, "Generating university course timetable using genetic algorithm and local search," in *Proc. 3rd Int. Conf. Hybrid Inform. Tech.*, 2008, pp. 254–260.

[4] A. Acan and Y. Tekol, "Chromosome reuse in genetic algorithms," in *Proc. 2003 Genetic Evol. Comput. Conf.*, 2003, pp. 695–705.

[5] M. A. Al-Betar, A. T. Khader, and A. T. Gani, "A harmony search algorithm for university course timetabling," in *Proc. 7th Int. Conf. Pract. Theory Automated Timetabling*, 2008, pp. 1–12.

[6] H. Arntzen and A. Løkketangen, "A local search heuristic for a university timetabling problem," *the 2002 International Timetabling Competition (TTC 2002)*, 2003.

[7] H. Asmuni, E. K. Burke, and J. M. Garibaldi, "Fuzzy multiple heuristic ordering for course timetabling," in *Proc. 5th UK Workshop Comput. Intell.*, 2005, pp. 302–309.

[8] P. P. Bonissone, R. Subbu, N. Eklund, and T. R. Kiehl, "Evolutionary algorithms + domain knowledge = real-world evolutionary computation," *IEEE Trans. Evol. Comput.*, vol. 10, no. 3, pp. 256–280, Jun. 2006.

[9] E. K. Burke, G. Kendall, and E. Soubeiga, "A tabu-search hyper-heuristic for timetabling and rostering," *J. Heuristics*, vol. 9, no. 6, pp. 451–470, 2003.

[10] E. K. Burke, J. Kingston, K. Jackson, and R. Weare, "Automated university timetabling: The state of the art," *Comput. J.*, vol. 40, no. 9, pp. 565–571, 1997.

[11] E. K. Burke, B. MacCloumn, A. Meisels, S. Petrovic, and R. Qu, "A graph-based hyper-heuristic for educational timetabling problems," *Eur. J. Oper. Res.*, vol. 176, no. 1, pp. 177–192, Jan. 2007.

[12] Y. Bykov, "Algorithm description," *the 2002 International Timetabling Competition (TTC 2002)*, 2003.

[13] M. W. Carter and G. Laporte, "Recent developments in practical course timetabling," in *Proc. 2nd Int. Conf. Pract. Theory Automated Timetabling*, (Lecture Notes in Computer Science), vol. 1408, 1998, pp. 3–19.

[14] A. Colorni, M. Dorigo, and V. Maniezzo, "Genetic algorithms—A new approach to the timetable problem," in *Lecture Notes in Computer Science* (NATO ASI Series, Combinatorial Optimization), Akgul *et al.*, Eds., vol. F-82, 1990, pp. 235–239.

[15] J.-F. Cordeau, B. Jaumard, and R. Morales, "Efficient timetabling solution with tabu search," *the 2002 International Timetabling Competition (TTC 2002)*, 2003.

[16] L. Davis, *Handbook of Genetic Algorithms*. New York: Van Nostrand Reinhold, 1991.

[17] A. Dubourg, B. Laurent, E. Long, and B. Salotti, "Algorithm description," *the 2002 International Timetabling Competition (TTC 2002)*, 2003.

[18] S. Even, A. Itai, and A. Shamir, "On the complexity of timetable and multicommodity flow problems," *SIAM J. Comput.*, vol. 5, no. 4, pp. 691–703, 1976.

[19] S. F. H. Irene, S. Deris, and S. Z. M. Hashim, "A study on PSO-based university course timetabling problem," in *Proc. Int. Conf. Adv. Comput. Control*, 2009, pp. 648–651.

[20] L. D. Gaspero and A. Schaerf, "Algorithm description," *the 2002 International Timetabling Competition (TTC 2002)*, 2003.

[21] M. Gen and R. Cheng, *Genetic Algorithms and Engineering Design*. New York: Wiley, 1997.

[22] C. C. Gotlieb, "The construction of class-teacher timetables," in *Proc. IFIP Congr.*, C. M. Popplewell, Ed., 1962, pp. 73–77.

[23] S. N. Jat and S. Yang, "A guided search genetic algorithm for the university course timetabling problem," in *Proc. 4th Multidisciplinary Int. Conf. Scheduling: Theory Appl.*, 2009, pp. 180–191.

[24] S. N. Jat and S. Yang, "A memetic algorithm for the university course timetabling problem," *Proc. 20th IEEE Int. Conf. Tools Artif. Intell.* 2008, vol. 1, pp. 427–433.

[25] P. A. Kostuch, "SA-based heuristic," *the 2002 International Timetabling Competition (TTC 2002)*, 2003.

[26] D. Landa-Silva and J. H. Obit, "Great deluge with non-linear decay rate for solving course timetabling problems," in *Proc. 4th IEEE Int. Conf. Intell. Syst.*, 2008, pp. 811–818.

[27] R. Lewis, "A survey of metaheuristic based techniques for university timetabling problems," *OR Spectrum*, vol. 30, no. 1, pp. 167–190, 2008.

[28] R. Lewis and B. Paechter, "Application of the grouping genetic algorithm to university course timetabling," in *Proc. 5th Eur. Conf. Evol. Comput. Combinat. Optim.* (Lecture Notes in Computer Science), vol. 3448, 2005, pp. 144–153.

[29] S. Louis and G. Li, "Augmenting genetic algorithms with memory to solve travelling salesman problem," in *Proc. 1997 Joint Conf. Inf. Sci.*, 1997, pp. 108–111.

[30] Z. Lü and J. K. Hao, "Adaptive tabu search for course timetabling," *Eur. J. Oper. Res.*, vol. 200, no. 1, pp. 235–244, Jan. 2010.

[31] M. R. Malim, A. T. Khader, and A. Mustafa, "Artificial immune algorithms for university timetabling," in *Proc 6th Int. Conf. Pract. Theory Automated Timetabling*, 2006, pp. 234–245.

[32] A. Masri and J. Ghaith, "Hybrid ant colony systems for course timetabling problems," in *Proc. 2nd Int. Conf. Data Mining Optim.*, 2009, pp. 120–126.

[33] B. McCollum, "University timetabling: Bridging the gap between research and practice," in *Proc. 6th Int. Conf. Pract. Theory Automated Timetabling*, 2006, pp. 15–35.

[34] P. Pongcharoen, W. Promtet, P. Yenradee, and C. Hicks, "Schotastic optimization timetabling tool for university course scheduling," *Int. J. Prod. Econ.*, vol. 112, no. 2, pp. 903–918, Apr. 2008.

[35] R. Qu, E. K. Burke, B. McCollum, and L. T. G. Merlot, "A survey of search methodologies and automated system development for examination timetabling," *J. Sched.*, vol. 12, no. 1, pp. 55–89, 2009.

[36] O. Rossi-Doria, C. Blum, J. Knowles, M. Sampels, K. Socha, and B. Paechter, "A local search for the timetabling problem," in *Proc. 4th Int. Conf. Pract. Theory Automated Timetabling* (Lecture Notes in Computer Science), vol. 2740, 2003, pp. 124–127.

[37] O. Rossi-Doria, M. Sampels, M. Birattari, M. Chiarandini, M. Dorigo, L. Gambardella, J. Knowles, M. Manfrin, M. Mastrolilli, B. Paechter, L. Paquete, and T. Stützle, "A comparison of the performance of different metaheuristics on the timetabling problem," in *Proc. 4th Int. Conf. Pract. Theory Automated Timetabling* (Lecture Notes in Computer Science), vol. 2740, 2003, pp. 329–351.

[38] K. Sastry, D. Goldberg, and G. Kendall, "Genetic algorithms," in *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques,* E. K. Burke and G. Kendall Eds. New York: Springer-Verlag, 2005, ch. 4, pp. 97–125.

[39] K. Socha, J. Knowles, and M. Samples, "A max-min ant system for the university course timetabling problem," in *Proc. 3rd Int. Workshop Ant Algorithms* (Lecture Notes in Computer Science), vol. 2463, 2002, pp. 1–13.

[40] N. D. Thanh, "Solving timetabling problem using genetic and heuristics algorithms," *J. Sched.*, vol. 9, no. 5, pp. 403–432, 2006.

[41] M. Tuga, R. Berretta, and A. Mendes, "A hybrid simulated annealing with kempe chain neighborhood for the university timetabling problem," in *Proc. 6th IEEE/ACIS Int. Conf. Comput. Inf. Sci.*, 2007, pp. 400–405.

Shengxiang Yang (M'00) received the B.Sc. and M.Sc. degrees in automatic control and the Ph.D. degree in systems engineering from Northeastern University, Shenyang, China, in 1993, 1996, and 1999, respectively.

From October 1999 to October 2000, he was a Postdoctoral Research Associate with the Algorithm Design Group, Department of Computer Science, King's College London, London, U.K. From November 2000 to June 2010, he was a Lecturer in the Department of Computer Science, University of Leicester, Leicester, U.K. He is currently a Senior Lecturer in the Department of Information Systems and Computing, Brunel University, Uxbridge, Middlesex, U.K. He has authored or coauthored more than 100 publications. He has given invited keynote speeches in several international conferences and coorganized several workshops and special sessions in conferences. He is an Area Editor, an Associate Editor, or an Editorial Board Member for four international journals. He is the Co-Editor of several books and conference proceedings and the Co-Guest Editor of several journal special issues. His research interests include evolutionary algorithms, swarm intelligence, metaheuristics and hyperheuristics, artificial neural networks, computational intelligence in dynamic and uncertain environments, scheduling, network flow problems and algorithms, and real-world applications.

Dr. Yang is a Member of the Association of Computing Machinery Special Interest Group on Genetic and Evolutionary Computation. He is a Member of the Task Force on Evolutionary Computation in Dynamic and Uncertain Environments, the Evolutionary Computation Technical Committee, and the IEEE Computational Intelligence Society. He is the founding Co-Chair of the IEEE Symposium on Computational Intelligence in Dynamic and Uncertain Environments.

Sadaf Naseem Jat received the B.Sc. and M.Sc. degrees in computer science from University of Sindh, Jamshoro, Pakistan, in 2000 and 2001, respectively. She is currently working toward the Ph.D. degree from the Department of Computer Science, University of Leicester, Leicester, U.K.

Her current research interests include evolutionary algorithms, memetic computing, and their applications for timetabling problems.