A Guided Search Genetic Algorithm for the University Course Timetabling Problem

Sadaf Naseem Jat · Shengxiang Yang

Abstract The university course timetabling problem is a combinatorial optimisation problem in which a set of events has to be scheduled in time slots and located in suitable rooms. The design of course timetables for academic institutions is a very difficult task because it is an NP-hard problem. This paper proposes a genetic algorithm with a guided search strategy and a local search technique for the university course timetabling problem. The guided search strategy is used to create offspring into the population based on a data structure that stores information extracted from previous good individuals. The local search technique is used to improve the quality of individuals. The proposed genetic algorithm is tested on a set of benchmark problems in comparison with a set of state-of-the-art methods from the literature. The experimental results show that the proposed genetic algorithm is able to produce promising results for the university course timetabling problem.

1 Introduction

Timetabling is one of the common scheduling problems, which can be described as the allocating of resources for factors under predefined constraints so that it maximises the possibility of allocation or minimises the violation of constraints [14]. Timetabling problems are often complicated by the details of a particular timetabling task. A general algorithm approach to a problem may turn out to be incapable, because of certain special constraints required in a particular instance of that problem. In the university course timetabling problem (UCTP), events (subjects, courses) have to be set into a number of time slots and rooms while satisfying various constraints. Timetabling has become much more difficult to find the general and effective solution due to the diversity of the problem, the variance of constraints, and particular requirements from university to university according to the charac-

Sadaf Naseem Jat
Department of Computer Science, University of Leicester
University Road, Leicester LE1 7RH, United Kingdom
E-mail: snj2@mcs.le.ac.uk

Shengxiang Yang
Department of Computer Science, University of Leicester
University Road, Leicester LE1 7RH, United Kingdom
E-mail: s.yang@mcs.le.ac.uk

teristics. There is no known deterministic polynomial time algorithm for the UCTP. That is, the UCTP is an NP-hard combinatorial optimisation problem [12].

The research on timetabling problems has a long history. Over the last forty years, researchers have proposed various timetabling approaches by using constraint-based methods, population-based approaches (e.g., genetic algorithms (GAs), ant colony optimization, and memetic algorithms), meta-heuristic methods (e.g., tabu search, simulated annealing, and great deluge), variable neighbourhood search (VNS), and hybrid and hyper-heuristic approaches etc. A comprehensive review on timetabling can be found in [8,18] and recent research directions in timetabling are described in [5].

Several researchers have used GAs to solve course timetabling problems [20,21,1,23]. Rossi-Doria et al. [16] compared different meta-heuristics to solve the course timetabling problem. 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. In this paper, a guided search genetic algorithm, denoted *GSGA*, is proposed for solving the UCTP, which consists of a guided search strategy and a local search technique. GAs rely on a population of candidate solutions [22]. If there is a good population, then chances increase to create a feasible and optimal solution. 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 population. In GSGA, a local search technique is also used to improve the quality of individuals through searching in three kinds of neighbourhood structures. In order to test the performance of the proposed GSGA, experiments are carried out on a set of benchmark problems in comparison with a set of state-of-the-art methods from the literature.

The rest of this paper is organised as follows. The next section briefly describes the UCTP. Section 3 presents the genetic algorithm proposed in this paper for the UCTP. Experimental results of comparing the proposed GA and other algorithms from the literature are reported and discussed in Section 4. Section 5 concludes this paper with discussions on the future work.

2 The University Course Timetabling Problem

According to Carter and Laporte [8], the UCTP is a multi-dimensional 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.

In a UCTP, we assign an event (courses, lectures) into a time slot and also assign a number of resources (students, rooms) in such a way that there is no conflict between the rooms, time slots and events. As mentioned by Rossi-Doria et al. [17], the UCTP problem consists of a set of n events (classes, subjects) $E = \{e_1, e_2, ..., e_n\}$ to be scheduled in a set of 45 time slots $T = \{t_1, t_2, ..., t_{45}\}$ (i.e., nine for each day in a five day week), a set of m available rooms $R = \{r_1, r_2, ..., r_m\}$ in which events can take place, a set of k students $S = \{s_1, s_2, ..., s_k\}$ who attend the events and a set of l available features $F = \{f_1, f_2, ..., f_l\}$ that are satisfied by rooms and required by each event.

In addition, interrelationships between these sets are given by five matrices. The first matrix shows which event is attended by which students. The second matrix indicates whether two events can be scheduled in the same time slot or not. The third matrix gives the features that each room possesses. The fourth matrix gives the features required by each event. The last matrix lists the possible rooms to which each event can be assigned.

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 corresponding 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 no 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 [16]. 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 be occurred. 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 $e_i \in E$ ($i = 1, 2, \cdots, n$). For example, the time slots and rooms are allocated to events in an ordered list of pairs like:

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

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

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. students 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. students should not attend more than two classes in a row. In this paper, we will test our proposed algorithm on the problem instances discussed in [16]. We deal with the following hard constraints:

- No student attends more than one events at the same time;
- The room is big enough for all the attending students and satisfies all the features required by the event;
- Only one event is in a room at any time slot.

There are also soft constraints which are penalised equally by their occurrences:

- A student has a class in the last time slot of a day;
- A student has more than two classes in a row;
- A student has a single class on a day.

The goal of the UCTP is to minimise 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 [17], as defined below:

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

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

3 The Guided Search Genetic Algorithm

GAs are a class of powerful general purpose optimisation tools that model the principles of natural evolution [11]. GAs have been used for timetabling since 1990 [10]. Since then,

Algorithm 1 The Guided Search Genetic Algorithm (GSGA)

```
 1: input : A problem instance I
 2: set the generation counter $g := 0$
    {initialize a random population}
 3: for $i := 1$ to population size do
 4:     $s_i \leftarrow$ create a random solution
 5:     $s_i \leftarrow$ solution $s_i$ after applying $LocalSearch()$
 6: end for
 7: while the termination condition is not reached do
 8:     if (g mod $\tau$) == 0 then
 9:         apply $ConstructMEM()$ to construct the data structure $MEM$
10:     end if
11:     $s \leftarrow$ child solution generated by applying $GuidedSearchByMEM()$ or the crossover operator
        with a probability $\gamma$
12:     $s \leftarrow$ child solution after mutation with a probability $P_m$
13:     $s \leftarrow$ child solution after applying $LocalSearch()$
14:     replace the worst member of the population by the child solution $s$
15:     $g := g + 1$
16: end while
17: output : The best achieved solution $s_{best}$ for the problem instance I
```

there are a number of papers investigating and applying GA methods for the UCTP [8]. In this paper, we propose an optimization method based on GAs that incorporates a guided search strategy and a local search operator for the UCTP. The pseudocode of the proposed guided search GA for the UCTP is shown in Algorithm 1.

The basic framework of GSGA is a steady state GA, where only one child solution is generated per iteration/generation. 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, a local search (LS) method as used in [9] is applied to each member of the initial population. The LS method uses three neighbourhood structures, which will be described in section 3.4, to move events to time slots and then uses the matching algorithm to allocate rooms to events and time slots. After the initialization of the population, a data structure (denoted MEM in this paper) is constructed, which stores a list of room and time slot pairs (r, t) for all the events with zero penalty (no hard and soft violation at this event) of selected individuals from the population. After that this 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, 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 improved by a mutation operator followed by the LS method. Finally, the worst member in the population is replaced with the newly generated child individual. The iteration continues until one termination condition is reached, e.g., a preset time limit t_{max} is reached.

In the following sub-sections, we will describe in details the key components of GSGA respectively, including the MEM data structure and its construction, the guided search strategy, the mutation operator, and the local search method.

3.1 The 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

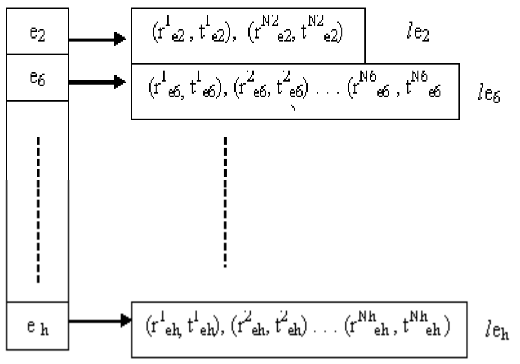

e_2
$(r^1_{e2}, t^1_{e2}), (r^{N2}_{e2}, t^{N2}_{e2})$
l_{e2}
e_6
$(r^1_{e6}, t^1_{e6}), (r^2_{e6}, t^2_{e6}) \ldots (r^{N6}_{e6}, t^{N6}_{e6}$
l_{e6}
e_h
$(r^1_{eh}, t^1_{eh}), (r^2_{eh}, t^2_{eh}) \ldots (r^{Nh}_{eh}, t^{Nh}_{eh})$
l_{eh}

Fig. 1 Illustration of the data structure MEM.

Algorithm 2 $ConstructMEM()$ – Constructing the data structure MEM

```
1:  input : The whole population $P$
2:  sort the population $P$ according to the fitness of individuals
3:  $Q \leftarrow$ select the best $\alpha$ 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 zero penalty) then
8:              add the pair of room and time slot $(r_{e_i}, t_{e_i})$ assigned to $e_i$ into the list $l_{e_i}$
9:          end if
10:     end for
11: end for
12: output : The data structure $MEM$
```

meta-heuristic methods for optimization and search [24,26,25]. In GSGA, we also use a data structure to guide the generation of offspring. Fig. 1 shows the details of the MEM data structure, which is a list of events and each event e_i has again a list l_{e_i} of room and time slot pairs. In Fig. 1, N_i represents the total number of pairs in the list l_{e_i}.

The MEM data structure is regularly reconstructed every τ generations. Algorithm 2 shows the outline of constructing MEM. When MEM is due to be reconstructed, we first select α best individuals from the population P to form a set Q. After that, for each individual $I_j \in Q$, each event is checked by its penalty value(Hard and soft constraints 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_{e_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_{e_2}. If for an event e_i, there is no a list l_{e_i} existing yet, then the list l_{e_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 3 $GuidedSearchByMEM()$ – Generating a child from MEM

```
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 $l_{e_i}$
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 the $MEM$ data structure
```

This MEM data structure is then used to generate offspring for the next τ generations before re-constructed. 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.

3.2 Generating a Child by the Guided Search Strategy

In GSGA, a child is created through the guided search by MEM or a crossover operator with a probability γ. That is, when a new child is to be generated, a random number $\rho \in [0.0, 1.0]$ is first generated. If ρ is less than γ, $GuidedSearchByMEM()$ (as shown in Algorithm 3) will be used to generate the new child; otherwise, a crossover operation is used to generate the new child. Below we first describe the procedure of generating a child through the guided search by MEM and then describe the crossover operator.

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. After that, for each event e_i in E_s, we randomly select a pair of (r_{e_i}, t_{e_i}) from the list l_{e_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 the list l_{ei} 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 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.

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 of size 2. Then, we exchange the time slots between the two parents and allocate rooms to events in each non-empty time slot.

3.3 Mutation

After a child is generated by using either MEM or crossover, a mutation operator is used with a probability P_m. The mutation operator first randomly selects one from three neighbourhood structures N1, N2 and N3, which will be described in Section 3.4, and then make a move within the selected neighbourhood structure.

3.4 Local Search

After mutation, a local search (LS) method is applied on the child solution for possible improvement. Algorithm 4 summarises the LS scheme used in GSGA. LS works on all

Algorithm 4 $LocalSearch()$ – Search the neighbourhood for improvement

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

events. Here, we suppose that each event is involved in soft and hard constraint violations. LS works in two steps and is based on three neighbourhood structures, denoted as N1, N2, and N3. They are described as follows:

– N1: the neighbourhood defined by an operator that moves one event from a time slot to a different one
– N2: the neighbourhood defined by an operator that swaps the time slots of two events
– N3: the neighbourhood 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 (line 2-12 in Algorithm 4), LS checks the hard constraint violations of each event while ignoring its soft constraint violations. If there are hard constraint violations for an event, LS tries to resolve them by applying moves in the neighbourhood structures N1, N2, and N3 orderly[1] until a termination condition is reached, e.g., an improvement is reached 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 (i.e., calculate the hard and soft constraint violations before and after the move). If there is no untried move left in the neighbourhood for an event,

[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, and so on. 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, and so on.

Table 1 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
Per room approximate features	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

LS continues to the next event. After applying all neighbourhood moves on each event, if there is still any hard constraint violation, then LS will stop; otherwise, LS will perform the second step (lines 13-25 in Algorithm 4).

In the second step, after reaching a feasible solution, the LS method is used to deal with soft constraints. LS performs a similar process as in the first step on each event to reduce its soft constraint violations. For each event, LS tries to make moves in the neighbourhood N1, N2, and/or N3 orderly 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 LS finishes, we get a possibly improved and feasible individual.

At the end of each generation, the obtained child solution replaces the worst member of the population to make a better population in the next generation.

4 Experimental Study

The program is coded in GNU C++ with version 4.1 and run on a 3.20 GHz PC. We use a set of benchmark problem instances to test our algorithm, which were proposed by Ben Paechter for the timetabling competition, see [15]. Although these problem instances lack many of the real world problem constraints and issues [13], they allow the comparison of our approach with current state-of-the-art techniques on them.

Table 1 represents the data of timetabling problem instances of three different groups: 5 small instances, 5 medium instances, and 1 large instance. The parameters for GSGA are set as follows: the population size pop_size is set to 50, $\alpha = 0.2 * pop\_size = 10$, $\beta = 0.6$, $\gamma = 0.8$, $\tau = 20$, and $P_m = 0.5$. In the local search, the maximum number of steps per local search s_{max} is set to different values for different problem instances, which are 200 for small instances, 1000 for medium instances, and 2000 for the large instance respectively. There were 50 runs of the algorithm for each problem instance. For each run, the maximum run time t_{max} was set to 90 seconds for small instances, 900 seconds for medium instances, and 9000 seconds for the large instance.

We compare our GSGA with other algorithms on the 11 timetabling problem instances. The algorithms compared in the table are described as follows:

– GSGA: The guided search genetic algorithm proposed in this paper
– RIIA: The randomised iterative improvement method by Abdullah et al. [2]. They presented a composite neighbourhood structure with a randomised iterative improvement algorithm.
– VNS: The variable neighbourhood search by Abdullah et al. [3]. In [3], they used a variable neighbourhood search approach based on the random-descent local search with an exponential Monte Carlo acceptance criteria.

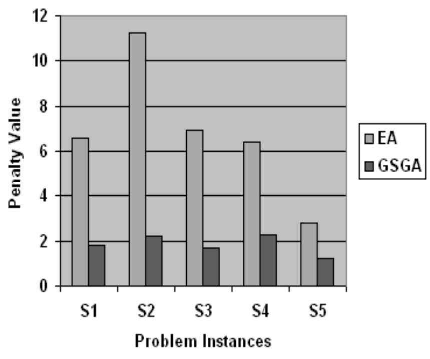
Table 2 Comparison of algorithms on small and medium problem instances

	GSGA		RIIA	GALS	GBHH	VNS	THHS	LS	EA	AA	FA
UCTP	Best	Med	Best	Best	Best	Best	Best	Med	Best	Med	Best
S1	**0**	0	**0**	2	6	**0**	1	8	**0**	1	10
S2	**0**	0	**0**	4	7	**0**	2	11	3	3	9
S3	**0**	0	**0**	2	3	**0**	**0**	8	**0**	1	7
S4	**0**	0	**0**	**0**	3	**0**	1	7	**0**	1	17
S5	**0**	0	**0**	4	4	**0**	**0**	5	**0**	**0**	7
M1	240	242.5	242	254	372	317	**146**	199	280	195	243
M2	**160**	164	161	258	419	313	173	202.5	188	184	325
M3	**242**	245	265	251	359	357	267	77.5%In	249	248	249
M4	**158**	161	181	321	348	247	169	177.5	247	164.5	285
M5	**124**	126.5	151	276	171	292	303	100%In	232	219.5	132
L	**801**	822	100%In	1027	1068	100%In	80%In	100%In	100%In	851.5	1138

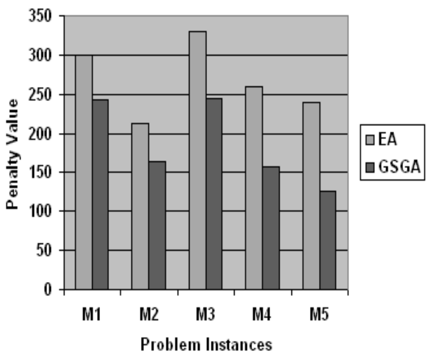
– THHS: The tabu-based hyper-heuristic search by Burke et al. [6]. They introduced a tabu-search hyper heuristics where a set of low level heuristics compete with each other. This approach was tested on the course timetabling and nurse rostering problems.
– EA: The evolutionary algorithm (EA) by Rossi-Doria et al. [16]. They used a local search with the EA to solve the UCTP and also compared several metaheuristics methods on the UCTP.
– GALS: The GA with local seach by Abdullah and Turabieh [1]. They tested a GA with a repair function and local seach on the UCTP.
– LS: The local search method by Socha et al. [19]. They used a random restart local search for the UCTP and compared it with an ant algorithm.
– AA: The ant algorithm used by Socha et al. [19]. They developed a first ant colony optimization algorithm with the help of construction graph and a pheromone model appropriate for the UCTP.
– FA: The fuzzy algorithm by Asmuni et al. [4]. In [4], Asmuni et al. focused on the issue of ordering events by simultaneously considering three different heuristics using fuzzy methods.
– GBHH: The graph-based hyper heuristic by Burke et al. [7]. They employed tabu search with graph-based hyper-heuristics for the UCTP and examination timetabling problems.

Table 2 gives the comparison of the experimental results of our algorithm with the available results of other algorithms in the literature on the small and medium timetabling problem instances. In the table, $S1$ represents small instance 1, $S2$ represents small instance 2, and so on, and $M1$ represents medium problem instance 1, $M2$ represents medium problem instance 2, and so on, and L represents large instance. In Table 2, the term "%ln" 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 2, it can be seen that our proposed GSGA is better than the fuzzy algorithm [4] and graph based approach [7] on all the 11 small, medium, and large problem instances. GSGA outperforms VNS [3], RIIA [2], and EA [16] on all the medium problem instances and ties them on some or all of the small problem instances. It also gives better results than local search [19] on 10 out of the 11 problem instances and is better than the ant algorithm [19] on 9 problem instances and ties it on S5. When comparing with the tabu-based hyper heuristic search [6], GSGA performs better or the same on all the problem instances except



(a)



(b)

Fig. 2 Comparison of GSGA with the EA from [16] on (a) small and (b) medium problems.

Table 3 The t-test results of comparing GSGA against the EA by Rossi-Doria et al. [16]

function	S1	S2	S3	S4	S5
t-test	8.8874	8.0551	9.5634	7.7520	4.5521
function	M1	M2	M3	M4	M5
t-test	6.5912	5.0794	4.5667	17.6783	18.4526

on M1. Finally, the results of our approach is better then the GALS approach [1] on all medium and large instances and ties on S4.

We are interested to compare the results of GSGA with EA to show that the guided serach approach can help to minimise the penalty values and give better results for UCTP in comparison with conventional EAs employed in [16]. Fig. 2 shows the average penalty value over 50 runs of the EA by Rossi-Doria et al. [16] and our proposed GSGA on the small and medium UCTP instances, respectively. The results of the t-test statistical comparison of GSGA against their EA are shown in Table 3. The t-test statistical comparison is carried out with 98 degrees of freedom at a 0.05 level of significance. From Fig. 2 and Table 3, it can be seen that the performance of GSGA is significantly better than the EA by Rossi-Doria et al. on all small and medium problems. These results show that by integrating proper guided search techniques the performance of GAs for the UCTP can be greatly improved.

5 Conclusion and Future work

This paper presents a guided search genetic algorithm, i.e., GSGA, to solve the university course timetabling problem, where a guided search strategy and a local search technique are integrated into a steady state genetic algorithm. 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 next population. In GSGA, a local search technique is also used to improve the quality of individuals through searching three neighbourhood structures. To our knowledge this is the first such algorithm aimed at this problem domain.

In order to test the performance of GSGA, experiments are carried out based on a set of benchmark problems to compare GSGA with a set of state-of-the-art methods from the literature. The experimental results show that the proposed GSGA is competitive and work reasonably well across all problem instances in comparison with other approaches studied in the literature. With the help of the guided search strategy, GSGA is capable of finding (near) optimal solutions for the university course timetabling problem and hence can act as a powerful tool for the UCTP.

Future work includes further analysis of the contribution of individual components (local search and guided search) toward the performance of GSGA. Improvement of genetic operators and new neighbourhood techniques based on different problem constraints will also be investigated. We believe that the performance of GAs for the UCTP can be improved by applying advanced genetic operators and heuristics. The inter-relationship of these techniques and a proper placement of these techniques in a GA may lead to a better performance.

References

1. S. Abdullah and H. Turabieh Generating university course timetable using genetic algorithm and local search. *Proc. of the 3rd Int. conf. on Hybrid Information Technology*, pp. 254-260, 2008.
2. S. Abdullah, E. K. Burke, and B. McCollum. Using a randomised iterative improvement algorithm with composite neighbourhood structures. *Proc. of the 6th Int. Conf. on Metaheuristic*, pp. 153-169, 2007.
3. S. Abdullah, E. K. Burke, and B. McCollum. An investigation of variable neighbourhood search for university course timetabling. *Proc. of the 2nd Multidisciplinary Conference on Scheduling: Theory and Applications*, pp. 413–427, 2005.
4. H. Asmuni, E. K. Burke, and J. M. Garibaldi. Fuzzy multiple heuristic ordering for course timetabling. *Proc. of the 5th UK Workshop on Comput. Intell.*, pp. 302-309, 2005.
5. E. K. Burke and S. Petrovic. Recent research directions in automated timetabling. *European Journal of Operation Research*, **140**(2): 266-280, 2002.
6. E. K. Burke, G. Kendall, and E. Soubeiga. A tabu-search hyper-heuristic for timetabling and rostering. *Journal of Heuristics*, **9**(6): 451-470, 2003.
7. E. K. Burke, B. MacCloumn, A. Meisels, S. Petrovic, and R. Qu. A graph-based hyper heuristic for timetabling problems. *European Journal of Operation Research*, **176**: 177–192, 2006.
8. M. W. Carter and G. Laporte. Recent developments in practical course timetabling. *Proc. of the 2nd Int. Conf. on Practice and Theory of Automated Timetabling*, LNCS 1408, pp. 3–19, 1998.
9. M. Chiarandini, M. Birattari, K. Socha, and O. Rossi-Doria. An effective hybrid algorithm for university course timetabling. *Journal of Scheduling*, **9**(5): 403–432, 2006.
10. A. Colorni, M. Dorigo, and V. Maniezzo. Genetic algorithms - A new approach to the timetable problem. In Akgul et al. (eds.), *NATO ASI Series, Combinatorial Optimization*, Lecture Notes in Computer Science, F(82), pp. 235-239, 1990.
11. L. Davis. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, 1991.
12. S. Even, A. Itai, and A. Shamir. On the complexity of timetable and multicommodity flow problems. *SIAM Journal on Computing*, **5**(4): 691–703, 1976.
13. B. McCollum. University Timetabling: Bridging the Gap between Research and Practice. *Proc of the 6th Int Conf on the Practice and Theory of Automated Timetabling*, pp. 15–35, 2006.
14. N. D Thanh Solving timetabling problem using genetic and heuristics algorithms *Journal of Scheduling*, **9**(5): 403–432, 2006.
15. http://iridia.ulb.ac.be/supp/IridiaSupp2002-001/index.html
16. 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. *Lecture Notes in Computer Science 2740*, pp. 329–351, 2002.
17. O. Rossi-Doria and B. Paechter. A memetic algorithm for university course timetabling. *Proceedings of Combinatorial Optimization (CO 2004)*, pp. 56. 2004.
18. A. Schearf. A survey of automated timetabling. *Artificial Intelligence Review*, **13**(2): 87–127, 1999.
19. K. Socha, J. Knowles, and M. Samples. A max-min ant system for the university course timetabling problem. *Proc. of the 3rd Int. Workshop on Ant Algorithms (ANTS 2002)*, LNCS 2463, pp. 1-13, 2002.

20. W. Erben,J. Keppler. A genetic algorithm solving a weeklycourse timetabling problem. *Proc. of the ist Int. Conf. on Practice and Theory of Automated Timetabling*, LNCS 1153, pp. 198-211, 1995.
21. P. Pongcharoen, W. Promtet, P. Yenradee, and C. Hicks. Schotastic Optimisation Timetabling Tool for University Course Scheduling. *International Journal of Production Economics*, **112**: 903-918, 2008.
22. K. Sastry, D. Goldberg, and G. Kendall. Genetic algorithms. In E. K. Burke and G. Kendall (Eds.), *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*. Chapter 4, pp. 97-125, Springer, 2005.
23. R. Lewis and B. Paechter. Application of the Grouping Genetic Algorithm to University Course Timetabling *Proc. of the 5th European Conf. on Evol. Comput. in Combinatorial Optimization (EvoCOP 2005)*, LNCS 3448, pp. 144-153, 2005.
24. A. Acan and Y. Tekol. Chromosome Reuse in Genetic Algorithms *Proc. of the 2003 Genetic and Evolutionary Computation Conference (GECCO 2003)*, pp. 695-705, 2003.
25. S. Louis and G. Li. Augmenting genetic algorithms with memory to solve traveling salesman problem *Proc. of the 1997 Joint Conference on Information Sciences*, pp. 108-111, 1997.
26. A. Acan. An External Memory Implementation in Ant Colony Optimization. *Proc. of the 4th Int. Workshop on Ant Colony Optimization and Swarm Intelligence (ANTS 2004)*, pp. 73-82, 2004.