A Guided Search Non-dominated Sorting Genetic Algorithm for the Multi-Objective University Course Timetabling Problem

Sadaf Naseem Jat[1] and Shengxiang Yang[2]

[1] Department of Computer Science, University of Leicester
University Road, Leicester LE1 7RH, United Kingdom
snj2@mcs.le.ac.uk
[2] Department of Information Systems and Computing, Brunel University
Uxbridge, Middlesex UB8 3PH, United Kingdom
shengxiang.yang@brunel.ac.uk

Abstract. The university course timetabling problem is a typical combinatorial optimization problem. This paper tackles the multi-objective university course timetabling problem (MOUCTP) and proposes a guided search non-dominated sorting genetic algorithm to solve the MOUCTP. The proposed algorithm integrates a guided search technique, which uses a memory to store useful information extracted from previous good solutions to guide the generation of new solutions, and two local search schemes to enhance its performance for the MOUCTP. The experimental results based on a set of test problems show that the proposed algorithm is efficient for solving the MOUCTP.

1 Introduction

In the university course timetabling problem (UCTP), events (subjects, courses) have to be set into a number of time slots and located in suitable rooms while satisfying various constraints. The UCTP is one of the most challenging scheduling problems due to its complexity and highly constrained nature. Usually, the UCTP is NP-hard. It is very difficult to find a general and effective solver for the UCTP due to the diversity of the problem and variance of constraints from institute to institute. Researchers have proposed various approaches, e.g., constraint-based methods, population-based methods, meta-heuristic, and hyper-heuristic approaches, for timetabling. Most research has taken timetabling as a single objective problem by combining multiple criteria into a single scalar value and then minimising the weighted sum of constraint violations as the only objective function. Few work has tackled the multi-objective UCTP (MOUCTP). Burke et al. [3] proposed a hyper-heuristic approach for MOUCTPs. Carrasco and Pato [5] used a bi-objective genetic algorithm (GA) to the class teacher timetabling problem. Datta et al. [6] used the non-dominated sorting GA (NSGA-II) [7] as a university class timetable optimizer. They used a bi-objective model to minimize the soft-constraint violations. A comprehensive review on multi-objective evolutionary algorithms (MOEAs) can be found in [2].

This paper proposes a guided search non-dominated sorting GA (*GSNSGA*) to solve the MOUCTP. GSNSGA integrates a guided search technique [9] and local search (LS) techniques into NSGA-II to solve the MOUCTP. NSGA-II [7] is chosen since it has been successfully used for multi-objective problems in different fields, including timetabling [5]. The guided search technique is used to create offspring to increase the rate of highly fit individuals in the population that lead NSGA-II to find the non-dominated set of solutions and LS techniques are used to enhance the performance of GSNSGA by encouraging better convergence and discovering any missing trade-off space. Experimental results on a set of MOUCTP instances show that GSNSGA is a good solver for the MOUCTP.

2 Description of the MOUCTP

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 events at the same time. Soft constraints should preferably be satisfied, e.g., a student should not attend more than two events in a row. It is very tough or even impossible to satisfy all the soft constraints [6]. This requires us to treat the scheduling of timetable as finding solutions over hard constraints, and optimize them over soft constraints [14].

In this paper, we will test algorithms on the problem instances discussed in [13]. These instances are dealt with as the MOUCTP due to the lack of MOUCTP benchmarks in the literature. 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;
- The room 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 equally penalized by the number of their violations and are described as follows:

- A student has an event in the last time slot of a day;
- A student attends more than two events consecutively;
- A student has a single event on a day.

The number of violations of each of the above three kinds of soft constraints can be taken as one objective function to be minimized. Hence, we have three objective functions, $f_1(x)$, $f_2(x)$, and $f_3(x)$, which are associated with the above three kinds of soft constraints, respectively, in the MOUCTP in this paper.

In a UCTP, we assign an event (course, lecture) into a time slot and also assign a number of resources (students and rooms) such that there is no conflict between the rooms, time slots, and events. The UCTP consists of a set of n events $E = \{e_1, e_2, ..., e_n\}$ to be scheduled into a set of 45 time slots $T = \{t_1, t_2, ..., t_{45}\}$ (9 for each day in a five-day week), a set of m 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

Algorithm 1 Guided Search Non-dominated Sorting Genetic Algorithm

```
1: input: A problem instance I
2: initialise a population P of N solutions
3: apply local search schemes LS1 and LS2 for individuals in P
4: evaluate individuals in P
5: assign rank and crowding distance for individuals in P
6: create data structures
7: set the generation counter g := 0
8: while the termination condition is not reached do
9:    if (g mod τ) == 0 then
10:       apply ConstructMEM() to construct data structures
11:   create a child population Q using GuidedSearch() or Crossover() with a probability γ
12:   apply mutation on individuals in Q with a probability P_m
13:   apply local search schemes LS1 and LS2 for individuals in Q
14:   evaluate the child individuals in Q
15:   merge P and Q and assign rank and crowding distance for individuals
16:   select a new population from the merge of P and Q based on rank and crowding distance
17:   g := g + 1
18: output: A non-dominated set of solutions
```

by rooms and required by events [13]. In addition, the inter-relationships between these sets are given by five matrices, see [9, 13] for details.

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 which 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. 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 room assignment, we use a matching algorithm described in [13]. For every time slot, there is a list of events taking place in it and a pre-processed 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. A solution to a UCTP can be represented as 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: $(2, 4), (3, 30), \cdots, (2, 7)$, where room 2 and time slot 4 are allocated to event 1, room 3 and time slot 30 are allocated to event 2, and so on.

3 The Proposed GSNSGA for the MOUCTP

The framework of GSNSGA, as shown in Algorithm 1, is based on NSGA-II [7]. Initially, a population P of N individuals are randomly generated. For each individual, each event is assigned a random time slot and a room via the matching algorithm. As random solutions have a low chance to be feasible, two LS methods, denoted LS1 and LS2, are used to convert them into feasible or near-feasible solutions. Then, the individuals in P is ranked by the non-dominated sorting as described in [7]. For each individual $I_i \in P$, we calculate the domination count n_i (the number of solutions in P which dominate I_i) and the set S_i of solutions that I_i dominates. Then, we construct the Pareto fronts from the population round by round as follows. All solutions with $n_i = 0$ form the first Pareto front.

For each solution I_i in the first Pareto front, we check each member in the set S_i and reduce its domination count value by one. If the domination count of a member becomes zero, we put it into a list L. After this round of checking, all the members in L form the second Pareto front. The above checking and ranking procedure continues until all Pareto fronts are identified. After ranking, the crowding distance [7] of each front is calculated, which is used for the density estimation for each individual. The crowding distance of a solution I_i is the average side-length of the cube that encloses the solution without including any other individuals in the population. After assigning ranks and crowding distances, GSNSGA constructs three data structures MEM_i $(i = 1, 2, 3)$ to store useful information from the best individuals of the population, which are used to guide the generation of offspring for the following generations.

In each generation, a child population Q is first generated using the data structures MEM_i $(i = 1, 2, 3)$ or crossover, depending on a probability γ. If crossover is applied, we select two parents according to the rank and crowding distance from the parent population P and apply crossover on them. After that, we perform mutation with a probability P_m. Mutation applies a randomly selected neighbourhood structure N1, N2, N3, or N4 to make a move. After mutation, we merge populations Q and P, assign rank and crowding distances for individuals as above, and select the best N solutions based on the ranks and crowding distances to form the population of next generation. The iteration continues until a stop condition is reached, e.g., a time limit t_{max} is reached.

The key components of GSNSGA, including the LS schemes, the data structures, and the guided search strategy, are described respectively as follows.

3.1 The LS Schemes (LS1 and LS2)

In GSNSGA, two LS schemes (LS1 and LS2) are used orderly on each individual in the initial population as well as after a child is created through crossover or the MEM data structure and mutation. The first scheme (LS1), as shown in Algorithm 2, is based on the LS scheme used in [13] with the extension of an additional neighbourhood. LS1 works in two steps based on four neighbourhood structures, denoted as N1, N2, N3, and N4, respectively, where N1 is defined by an operator that moves one event from a time slot to a different one, N2 is defined by an operator that swaps the time slots of two events, N3 is 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, and N4 is defined by an operator that swaps the time slots of two consecutive events with the time slots of another two consecutive events.

In the first step (lines 2-9 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, LS tries to resolve them by applying moves in the neighbourhood structures N1, N2, N3, and N4 orderly, until 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

Algorithm 2 Local Search Scheme 1 (LS1)

```
1: input : Individual I from the population
2: for each event e_i ∈ 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, then N3 if N2 also fails, and finally
               N4 if N3 also fails
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: if no any hard-constraint violations remain then
11:     for each event e_i ∈ E do
12:         if event e_i has soft constraint violation then
13:             if there is untried move left then
14:                 calculate the moves: first N1, then N2 if N1 fails, then N3 if N2 also fails, and finally
                   N4 if N3 also fails
15:                 apply the matching algorithm to the time slots affected by the move to allocate
                   rooms for events
16:                 delta evaluate the result of the move
17:                 if moves reduce soft-constraint violations then
18:                     make the moves and go to line 13
19: output : A possibly improved individual I
```

Algorithm 3 Local Search Scheme 2 (LS2)

```
1: input : Individual I after LS1 is applied
2: while the termination condition is not reached do
3:     S := randomly select a preset percentage of time slots from the total time slots of T
4:     for each time slot t_i ∈ S do
5:         for each event j in time slot t_i do
6:             calculate the penalty value of event j
7:         sum the total penalty value of events in time slot t_i
8:     select the time slot w_t with the biggest penalty value from S
9:     for each event i in w_t do
10:        calculate a move of event i in the neighbourhood structure N1
11:        apply the matching algorithm to the time slots affected by the move
12:        compute the penalty of event i and delta evaluate the result
13:    if all the moves together reduce hard or soft constraint violations then
14:        apply the moves
15:    else
16:        delete the moves
17: output : A possibly improved individual I
```

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, LS1 continues to the next event. After applying all neighbourhood moves on each event, LS1 will perform the second step (lines 10-18 in Algorithm 2). In the second step, after reaching a feasible solution, LS performs a similar process as in the first step on each event to reduce its soft-constraint violations without violating hard constraints. When LS1 finishes, we get a possibly improved feasible individual.

LS2, as shown in Algorithm 3, is used immediately after LS1 on an individual. 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

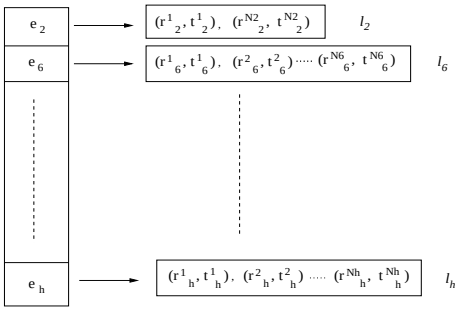

Fig. 1. Illustration of the data structure MEM_i $(i = 1, 2, 3)$.

percentage of time slots[3] (e.g., 30% 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 local search as follows: LS2 tries a move in the neighbourhood 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. 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.

3.2 Data Structures MEM_i $(i = 1, 2, 3)$

Usually, it is assumed that elitism and diversity preservation mechanisms improve the performance of MOEAs [2]. In GSNSGA, we also create extra data structures (memories) to preserve best parts of individuals to guide the generation of offspring. We create three data structures, each of which stores useful information according to one of the three objectives. Figure 1 shows the data structure MEM_i $(i = 1, 2, 3)$, associated with the i-th objective. In MEM_i, there is a list of events and each event e_k has again a list l_{e_k} of room and time slot pairs. In Fig. 1, N_k represents the total number of pairs in the list l_{e_k}. The data structures are regularly reconstructed, e.g., every τ generations.

Algorithm 4 shows the outline of the (re-)construction of the data structures. When the data structures are due to be (re-)constructed, we first select α best individuals from the population P to form a set Q. After that, for each individual $I_j \in Q$, we check its objective values. If any of its objectives, say $f_i(I_j)$, has a zero value, then each event of I_j is checked by its penalty value (hard and soft

[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.

Algorithm 4 *ConstructMEM*() – Constructing data structures

```
1: input : The whole population P
2: Q ← select the best α individuals in P
3: for each individual I_j in Q do
4:     for each objective i do
5:         if f_i(I_j) = 0 then
6:             for each event e_k in I_j do
7:                 calculate the penalty value of event e_k from I_j
8:                 if e_k is feasible (i.e., e_k has zero constraint violation) then
9:                     add the pair of room and time slot (r_{e_k}, t_{e_k}) assigned to e_k into the list l_{e_k} in
                       MEM_i
10: output : The updated data structures MEM_i (i = 1, 2, 3)
```

constraints associated with this event). If an event has a zero penalty value, then we store the information corresponding to this event into corresponding data structure MEM_i. For example, for an individual $I_j \in Q$, assuming $f_1(I_j) = 0$, which means no students have a class in the last time slot of a day in the solution I_j, if the event e_2 of I_j 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} in MEM_1. Similarly, the events of the next individual $I_{j+1} \in Q$ are checked by their penalty values. If $f_1(I_{j+1}) = 0$ and 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} in MEM_1. If an event e_m in an individual $I_k \in Q$ with $f_1(I_k) = 0$ has a zero penalty and there is no list l_{e_m} existing in MEM_1 yet, then the list l_{e_m} is added into MEM_i. Similar process is carried out for each individual in Q. Finally, MEM_i stores a list of pairs of room and time slot for each event with a zero penalty corresponding to the best individuals of the population regarding the i-th objective.

The data structures are then used to generate offspring for the next τ generations before re-constructed. We update the data structures every τ generations instead of every generation in order to make a balance between the solution quality and the computational time cost.

3.3 Generating a Child by the Guided Search Strategy

In GSNSGA, a child population is created by the guided search strategy or crossover with a probability γ. That is, when a child is to be generated, a random number $\rho \in [0.0, 1.0]$ is first generated. If $\rho < \gamma$, the guided search strategy is used to generate the child; otherwise, a crossover operation is used to generate the child. If a child is to be created by the guided search strategy, we first randomly select one data structure MEM_i and then apply Algorithm 5.

In Algorithm 5, we first select a set E_s of $\beta * n$ random events to be generated from MEM_i. Here, β is the percentage of the total number of events. After that, for each event e_k in E_s, we randomly select a pair (r_{e_k}, t_{e_k}) from the list l_{e_k} in MEM_i that corresponds to the event e_k and assign the selected pair to e_k for the child. If an event e_k in E_s has no list l_{e_k} in MEM_i, then we randomly assign a room and a time slot from possible rooms and time slots to e_k for the child. This process is carried out for all events in E_s. For those remaining events not present in E_s, available rooms and time slots are randomly assigned to them.

Algorithm 5 *GuidedSearch*(MEM_i) – Generating a child from MEM_i

1: **input** : The MEM_i 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: **for** each remaining event e_i not in E_s **do**
7: assign a random time slot and room to event e_i
8: **output** : A child generated using the MEM_i data structure

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
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

4 Experimental Study

In this section, we experimentally investigate the performance of GSNSGA and NSGA-II [7] for the MOUCTP. The program was 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 the algorithms, which were proposed for the timetabling competition, see [8]. Table 1 represents the data of the UCTP instances of three different groups: 5 small instances, 5 medium instances, and 1 large instance. According to our preliminary experiments, the parameters for GSNSGA and NSGA-II were set as follows: $N = 50$, $\alpha = 0.2 * N = 10$, $\beta = 0.4$, $\gamma = 0.6$, $\tau = 30$, and $P_m = 0.6$. In the initialisation of the population, the maximum number of steps per LS operation s_{max} was set to 300 for small instances, 1500 for medium instances, and 2500 for the large instance, respectively. There were 20 runs of each algorithm on each problem instance. For each run, the maximum run time t_{max} was set to 100 seconds for small instances, 1000 seconds for medium instances, and 10000 seconds for the large instance.

Firstly, we compare the performance of GSNSGA and NSGA-II regarding the three objective values. The experimental results are shown in Table 2, where $S1$ to $S5$ denote small instance 1 to small instance 5, $M1$ to $M5$ denote medium instance 1 to medium instance 5, and L denotes the large instance, respectively. In Table 2, "Best", "Average", and "Std" mean the best, average, and standard deviation of the three objective values over 20 runs, respectively, "ln" means that over 50% of the results are infeasible. The objective function values of GSNSGA on all problem instances are much smaller than the values for NSGA-II. This shows that local and guided search help the algorithm to find different or unexplored regions of the search space and try to lead the algorithm to global optimum. Figure 2 shows the 3-D and 2-D projections of the objective functions of Pareto front of NSGA-II and GSNSGA on $S1$ and $M1$, respectively. The scale

Table 2. Results of NSGA-II and GSNSGA regarding the three objective values

Algo	MOUCTP	Best			Average			Std		
		f1	f2	f3	f1	f2	f3	f1	f2	f3
NSGA-II	s1	4	5	59	9.43	23.22	87.61	2.23	11.82	11.72
	s2	5	3	72	9.17	21.83	90.94	2.09	9.01	10.01
	s3	5	3	72	8.52	24.42	85.79	2.48	12.31	8.75
	s4	4	1	91	8.88	7.54	111.74	2.07	4.6	13.11
	s5	6	22	52	9.71	48.54	72.06	1.96	15.01	10.82
	m1	38	249	61	44.24	312.78	81.94	2.82	28.29	8.27
	m2	38	277	61	44.67	317.25	82.46	2.93	19.93	8.36
	m3	38	277	58	44.06	346.92	80.94	3.5	26.95	8.47
	m4	38	234	66	44.23	309.79	80.73	2.87	30.09	8.54
	m5	38	234	66	44.23	309.79	80.73	2.87	30.09	8.54
	l	ln	ln	ln	ln	ln	ln	ln	ln	ln
GSNSGA	s1	0	0	0	1.33	6.74	9.91	0.94	3.9	4.66
	s2	0	0	0	1.63	5.75	5.9	1.35	4.12	3.51
	s3	0	0	0	0.65	2.06	7.38	0.76	2.17	5.51
	s4	0	0	0	1.02	1.14	20.46	0.93	1.48	8.86
	s5	0	0	0	1.52	2.04	15.1	1.52	2.22	6.84
	m1	3	95	15	8.74	138.76	32.52	3.75	23.25	11.55
	m2	7	94	3	13	176.6	21	2.73	24.68	6.73
	m3	1	95	5	6.9	145.81	16.69	2.33	20.29	7.88
	m4	0	38	2	7.15	88.81	22.38	5.36	20.29	15.44
	m5	5	94	15	23.15	150.4	43.15	9.72	25.71	9.72
	l	30	221	89	39.72	345.94	124.64	5.46	73.62	21.1

of Fig. 2 is based upon the objective values of non-dominated solutions. From Fig. 2, it can be seen that there is a huge difference between the objective values of the two algorithms. For example, on $M1$, the minimal $f_3(x)$ value of NSGA-II is greater than the maximal $f_3(x)$ value of GSNSGA.

Secondly, we compare the performance of NSGA-II and GSNSGA regarding some other performance measures used for MOEAs. As the true Pareto front of the problems is unknown, we use two performance measures, hypervolume [15] and D metric [15], which are not based on the true Pareto front. The first measure concerns the size of the objective space which is covered by a set of non-dominated solutions. The higher the value, the larger the dominated volume in the objective space and hence the better an algorithm's performance. The D metric measure between two non-dominated sets A and B gives the relative size of the region in the objective space that is dominated by A but not by B, and vice versa. It also gives information about whether either set totally dominates the other set, e.g., $D(A, B) = 0$ and $D(B, A) > 0$ means that A is totally dominated by B. Since in this paper the focus is on finding the Pareto optimal set rather than obtaining a uniform distribution over a trade-off surface, we do not consider the online performance of MOEAs but consider the offline performance. Hence, the Pareto optimal set regarding all individuals generated over all generations is taken as the output of a MOEA. The performance of a particular algorithm on a test problem was calculated by averaging over all 20 runs.

Table 3 shows the values of the hypervolume and D metric of NSGA-II and GSNSGA on the test instances. It can be seen that GSNSGA covers a larger objective value space compared with NSGA-II on all problem instances. It is also evident from the D metric values that there is no objective space that GSNSGA

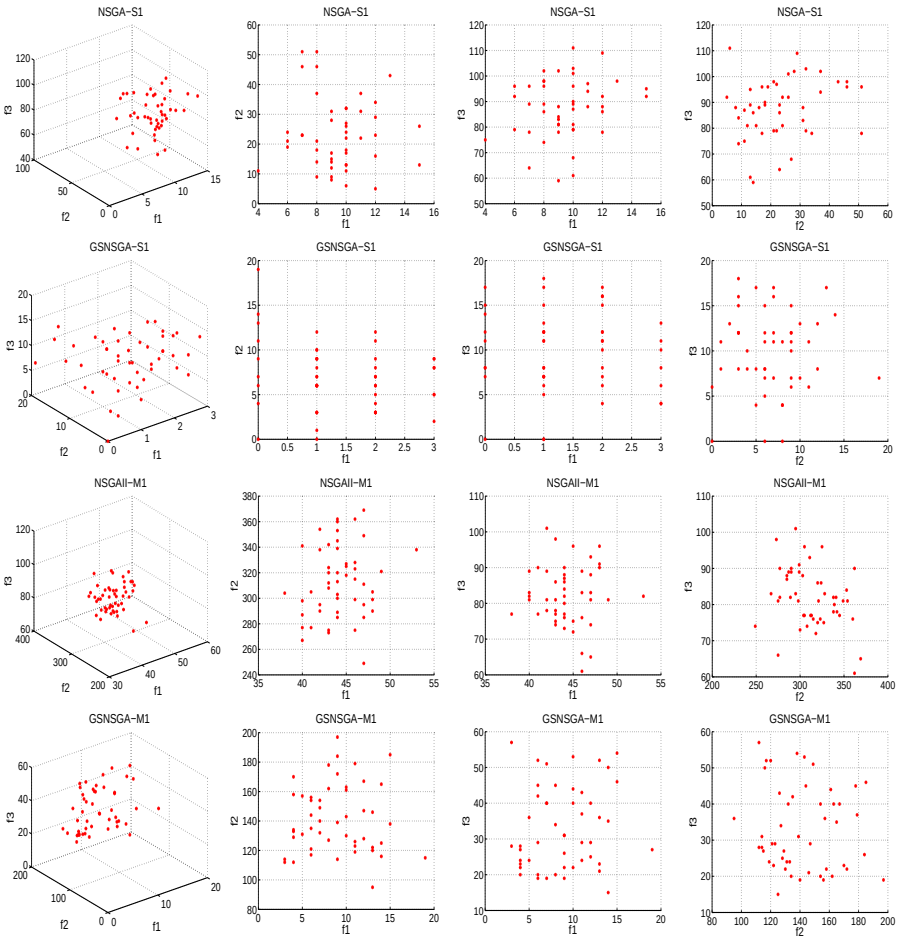

Fig. 2. Comparison of NSGA-II and GSNSGA regarding the three objective values

Table 3. Comparison of algorithms regarding the hypervolume and D metric measures

	Hypervolume		D metric	
MOUCTP	NSGA-II	GSNSGA	NSGA-II vs GSNSGA	GSNSGA vs NSGA-II
S1	5.3×10^6	8.0×10^6	0	7.8×10^6
S2	4.8×10^6	7.9×10^6	0	7.5×10^6
S3	4.8×10^6	8.0×10^6	0	7.4×10^6
S4	4.2×10^6	7.0×10^6	0	6.8×10^6
S5	5.0×10^6	7.9×10^6	0	7.6×10^6
M1	5.0×10^7	9.7×10^7	0	9.5×10^7
M2	4.4×10^7	9.4×10^7	0	9.3×10^7
M3	4.5×10^7	9.9×10^7	0	9.4×10^7
M4	5.3×10^7	1.1×10^8	0	9.2×10^7
M5	5.1×10^7	9.4×10^7	0	9.3×10^7
L	–	3.1×10^8	–	–

Table 4. Comparison of GSNSGA and other algorithms on the test problem instances

MOUCTP	GSNSGA Best	GSGA Best	TSRW Best	VNS Best	GBHH Best	EGD Best	NLGD Best
S1	**0**	**0**	**0**	**0**	6	**0**	3
S2	**0**	**0**	-	**0**	7	**0**	4
S3	**0**	**0**	–	**0**	3	**0**	6
S4	**0**	**0**	–	**0**	3	**0**	6
S5	**0**	**0**	–	**0**	4	**0**	**0**
M1	113	240	–	317	372	**80**	140
M2	**104**	160	173	313	419	105	130
M3	**101**	242	224	357	359	139	189
M4	**42**	158	160	247	348	88	112
M5	114	124	–	292	171	**88**	141
L	**340**	801	–	ln	1068	730	873

is dominated by NSGA-II. On all test problems, GSNSGA outperformed NSGA-II regarding the two performance measures.

Thirdly, a comparison of GSNSGA with other published results was also conducted in order to assess the effectiveness of GSNSGA against other optimisation methods. Since most published results are based on the single objective UCTP, we also compare the results of GSNSGA by aggregating the three objective values into one objective. Table 4 shows the results, where "−" means no result available in the literature, "ln" means no feasible solution for the problem instance, and the best results among all approaches are shown in bold. In Table 4, GBHH [4] denotes a graph-based hyper-heuristics with tabu search for the UCTP, GSGA [9] denotes the guided search GA with LS for the UCTP, VNS [1] denotes the variable neighbourhood search, EGD [11] denotes an extended great deluge method, NLGD [10] denotes a non-linear great deluge algorithm, and TSRW [3] denotes the tabu search roulette wheel hyper-heuristic to solve the three-objective UCTP. From Table 4, it can be seen that GSNSGA obtained the best results for 9 out of 11 problem instances and the second best results for 2 problems. In summary, GSNSGA is able to produce high quality solutions no matter how many objectives the problem has in comparison to other methods.

5 Conclusions and Future work

This paper presents a MOEA that combines guided search and LS techniques with NSGA-II to solve the MOUCTP. NSGA-II gives good results on MOUCTPs, but when it is integrated with the guided search and LS techniques, the improvement is noticeable. The data structures introduced in GSNSGA improve 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 to guide the generation of offspring. This enables GSNSGA to quickly retrieve the best solutions corresponding to previous populations. The experimental results show that GSNSGA is competitive across all test problems. It gives good results by producing a set of non-dominated solutions for the user to choose the most appropriate one rather than restricting to a single solution. It can also be

seen that the addition or deletion of constraints or objectives does not affect the performance of GSNSGA much because each objective function is treated separately. Hence, GSNSGA is appropriate for the MOUCTP.

There are several relevant future works. One would be to check the performance of guided and local search with other MOEAs for the MOUCTP. We also intend to test our approach on other problem instances and devise new genetic operators and neighborhood techniques based on different problem constraints.

References

1. Abdullah, S., Burke, E. K., McCollum, B.: An investigation of variable neighbourhood search for university course timetabling. In: Proc. of the 2nd Multidisciplinary Int. Conf. on Scheduling: Theory and Appl., pp. 413–427 (2005)
2. Abdullah, K., Coit, D. W., Smith, A. E.: Multi-objective optimisation using genetic algorithms: A tutorial. Reliability Engg and System Safty, 91(9), 992–1007 (2006)
3. Burke, E. K., Silva, J. D. L., Soubeiga, E.: Multi-objective hyper-heuristic approaches for space allocation and timetabling. In: Ibaraki, T., Nonobe, K., Yagiuru, M. (eds.), Meta-heuristics: Progress as Real Problem Solvers, Springer, Chapter 6, pp. 129–158 (2003)
4. Burke, E. K., MacCloumn, B., Meisels, A., Petrovic, S., Qu, R.: A graph-based hyper-heuristic for educational timetabling problems. Europ. J. of Oper. Res., 176(1), 177–192 (2007)
5. Carrasco, M. P., Pato, M. V.: A multiobjective genetic algorithm for the class/teacher timetabling problem. In: PATAT III, LNCS 2079, pp. 3–17 (2001)
6. Datta, D., Deb, K., Fonseca, C. M.: Multi-objective evolutionary algorithm for university class timetabling problem. In: Dahal, K. P., Tan, K. C., Cowling, P. I. (eds.), Evolutionary Scheduling, Springer, pp. 197-236 (2007)
7. Deb, K., Agrawal, S., Pratap, A., Meyarivan, T.: A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II. In: Proc. of the 6th Int. Conf. on Parallel Problem Solving from Nature, pp. 849–858 (2000)
8. http://iridia.ulb.ac.be/supp/IridiaSupp2002-001/index.html
9. Jat, S. N., Yang, S.: A guided search genetic algorithm for the university course timetabling problem. In: Proc. of the 4th Multidisciplinary Int. Conf. on Scheduling: Theory and Appl. (2009)
10. Landa-Silva, D., Obit, J.,H.: Great deluge with non-linear decay rate for solving course timetabling problem. In: 4th Int. IEEE conf. on Intelligent Systems, (2008)
11. McMullan, P.: An Extended Implementation of the Great Deluge Algorithm for Course Timetabling. In: Int. Conf. on Computational Science, pp. 538-545 (2007)
12. Paquete, L. F., Fonseca, C. M.: A study of examination timetabling with multiobjective evolutionary algorithms. In: Proc. of the 4th Metaheuristics Int. Conf., pp. 149-154 (2001)
13. Rossi-Doria, O., Sampels, M., Birattari, M., Chiarandini, M., Dorigo, M., Gambardella, L., Knowles, J., Manfrin, M., Mastrolilli, M., Paechter, B., Paquete, L., Stützle, T.: A comparison of the performance of different metaheuristics on the timetabling problem. In: PATAT IV, LNCS 2740, pp. 329–351 (2003)
14. Rudova, H., Murray, K.: University course timetabling with soft constraints. In: PATAT IV, LNCS 2740, pp. 310–328 (2003)
15. Zitzler, E.: Evolutionary Algorithms for Multiobjective Optimization: Methods and Applications. Shaker (1999)