A Genetic Algorithm Based University Course Timetabling Method by Using Guided and Local Search Strategies

Ravi Teja CH [#1], [#1]M.Tech Scholar

Department of Computer Science & Engineering,
University College Of Engineering,Vizianagaram ,JNTUK
Vizianagaram Dist,AP,India.

Abstract

University course timetabling is one of the important and time consuming issues that each University is involved with it at the beginning of each. 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. There are also a number of hard and soft constraints that must be observed while solving this problem, which makes the solution algorithm a challenge for researchers. 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.

Keywords:
Genetic algorithm (GA), guided search, local search (LS), university course timetabling problem (UCTP).

1. Introduction

The timetabling problem is an important practical problem that is frequently encountered in educational institutions, such as schools and universities. The timetabling problem has received special attention from the scientific community in the last few decades. This is mainly due to the fact that manual generation of timetables is very time consuming and the resulting timetables are usually inefficient and may be costly in terms of money and resources. The interest in timetabling algorithms resulted in the creation of the PATAT series of conferences (Practice and Theory of Automated Timetabling), which sponsors the International Timetabling Competition (ITC). The aim of this competition is to encourage research in the university timetabling domain and bridge the gap between theory and practice, for a better utilization of research techniques in real-world applications.

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 maximizes the possibility of allocation or minimizes the violation of constraints [1]. 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 characteristics. There is no known deterministic polynomial time algorithm for the UCTP. That is, the UCTP is an NP-hard combinatorial optimisation problem [2].

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 [3,4] and recent research directions in timetabling are described in [5].Several researchers have used GAs to solve course timetabling problems [6].Rossi-Doria et al. [7] 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**.** 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.

2. Related Work

Several algorithms have been introduced to solve timetabling problems. The earliest set of algorithms is 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.

A Genetic Algorithm (GA) is a famous optimization tool in computer science. It is an intelligent search method that is inspired from biological evolution and survival of the fittest. It operates on a population of solutions, allocating trials to promising areas of the search space. A GA does not depend heavily on the information available from the underlying problem, and it can be easily hybridized to generate knowledge-augmented GA. Using the operations of selection of the fittest, mutation, and crossover, GAs can quickly reach fit individuals (not always the most fit), but who are usually good enough as solutions to problems of a large magnitude. Crossover is considered as the main GA operator, which requires combining two solutions, while the mutation operator performs some small random change on a single solution. Therefore, designing an appropriate crossover operator is often more challenging than developing a mutation operator or a simple neighborhood move.

This usually makes GAs implementation more difficult compared to other heuristic or meta-heuristic techniques that gradually improve only one problem solution. Using a GA to solve scheduling and timetabling problems is attractive for researchers in the heuristic and meta-heuristic field, since GAs usually performs well in a variety of hard combinatorial optimization problems. For more information about genetic algorithms, the reader is referred to the book of Goldberg.

3. The University Course Timetabling Problem

According to Carter and Laporte [3], 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**.** [8], the UCTP problem consists of a set of n events (classes, subjects) E = {e1, e2, ..., en} to be scheduled in a set of 45 time slots T = {t1, t2, ..., t45} (i.e., nine for each day in a five day week), a set of m available rooms R = {r1, r2, ..., rm} in which events can take place, a set of k students S ={s1, s2, ..., sk} who attend the events and a set of l available features F = {f1, f2, ..., fl} 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 ri and a time slot ti. Each pair of (ri, ti) is assigned a particular number corresponding to an event. If a room ri in a time slot ti 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 constraints will always been satisfied.

For the room assignment we use a matching algorithm described by Rossi-Doria [7]. 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 (ri, ti), of which the index of each pair is the identification number of an event ei € E (i = 1, 2, · · · , 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), · · · , (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 [7]. 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 penalized 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 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 [8], as defined below:

$$f(s) := \#hcv(s) * C + \#scv(s) \quad (1)$$

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

4. The Guided Search Genetic Algorithm

GAs are a class of powerful general purpose optimisation tools that model the principles of natural evolution. GAs has been used for timetabling since 1990. Since then, there are a number of papers investigating and applying GA methods for the UCTP [3].

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.

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: si create a random solution
5: si solution si after applying LocalSearch()
6: **end for**
7: **while** the termination condition is not reached **do**
8: **if** (g mod _) == 0 **then**
9: apply ConstructMEM () to construct the data structure MEM
10: **end if**
11: s child solution generated by applying GuidedSearchByMEM() or the crossover operator with a probability
12: s child solution after mutation with a probability Pm
13: s 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**

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 4.4, 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. 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.

4.1 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 meta-heuristic methods for optimization and search [10]. 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 $\mathbf{e_i}$ has again a list $\mathbf{le_i}$ of room and time slot pairs. In Fig. 1, **Ni** represents the total number of pairs in the list $\mathbf{le_i}$.

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 Ij € 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.

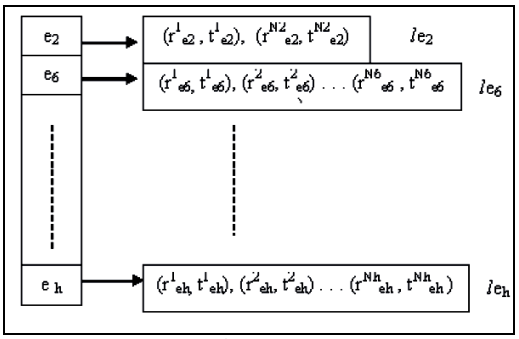

Fig. 1 Illustration of the data structure MEM.

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 select the best _ individuals in P
4: **for** each individual Ij in Q **do**
5: **for** each event ei in Ij **do**
6: calculate the penalty value of event ei from Ij
7: **if** e_i is feasible (i.e., e_i has zero penalty) **then**
8: add the pair of room and time slot (r_{ei} , t_{ei}) assigned to e_i into the list l_{ei}
9: **end if**
10: **end for**
11: **end for**
12: **output:** The data structure MEM

For example, if the event e2 of an individual Ij € 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 le2 . Similarly, the events of the next individual Ij+1 2 Q are also checked by their penalty values. If the event e2 in Ij+1 has a zero penalty, then we add the pair of room and time slot assigned to e2 in Ij+1 into the existing list le2 . If for an event ei, there is no a list lei existing yet, then the list $\mathbf{le_i}$ 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:** TheMEM data structure
2: Es := randomly select β*n events
3: **for** each event ei in Es **do**
4: randomly select a pair of room and time slot from the list lei
5: assign the selected pair to event ei for the child
6: **end for**
7: **for** each remaining event ei not in Es **do**
8: assign a random time slot and room to event ei
9: **end for**
10: **output:** A new child generated using the MEM data structure

4.2 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 τ. That is, when a new child is to be generated, a random number p € [0.0, 1.0] is first generated. If _ is less than, 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 Es of β * n random events to be generated from MEM. After that, for each event ei in Es, we randomly select a pair of (rei , tei) from the list lei that corresponds to the event ei and assign the selected pair to ei for the child. If there is an event ei in Es but there is no the list lei in MEM, then we randomly assign a room

and time slot from possible rooms and time slots to ei for the child. This process is carried out for all the events in Es. For those remaining events that are not present in Es, 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.

4.3 Mutation

After a child is generated by using either MEM or crossover, a mutation operator is used with a probability Pm. The mutation operator first randomly selects one from three neighbourhood structures N1, N2 and N3, which will be described in Section 4.4, and then make a move within the selected neighbourhood structure.

4.4 Local Search

After mutation, a local search (LS) method is applied on the child solution for possible improvement. Algorithm 4 summarizes the LS scheme used in GSGA. LS works on all 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 orderly1 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).

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 ei 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**

If there is no untried move left in the neighbourhood for an event, 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.

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

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.

4.5 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 5, is introduced and combined with LS1 for GSGA. 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 slots3 (e.g., 20% as used in this paper) from the total time slots of *T*. Then, it calculates the penalty of each selected time slot4 and chooses the worst time slot *wt* that has the biggest penalty value for LS. After taking the worst time slot, LS2 tries a move in the neighborhood *N*1 for each event of *wt* and checks the penalty value of each event before and after applying the move. If all the moves in *wt* 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.

Algorithm 5: Local Search Scheme2 (LS2)

```
input : Individual I after LS1 is applied
S := randomly select a preset percentage of time slots from the total time slots of T
for each time slot t_i ∈ S do
    for each event j in time slot t_i do
        calculate the penalty value of event j
    end for
    sum the total penalty value of events in time slot t_i
end for
select the time slot w_t with the biggest penalty value from S
for each event i in w_t do
    calculate a move of event i in the neighbourhood structure N1
    apply the matching algorithm to the time slots affected by the move
    compute the penalty of event i and delta evaluate the result
end for
if all the moves together reduce hard or soft constraint violations then
    apply the moves
else
    delete the moves
end if
output : A possibly improved individual I
```

5. Conclusion

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.

6. Future Work

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.

7. References

[1]. N. D Thanh Solving timetabling problem using genetic and heuristics algorithms *Journal of Scheduling*, **9**(5): 403–432, 2006.

[2]. 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.

[3]. 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.

[4]. A. Schearf. A survey of automated timetabling. *Artificial Intelligence Review*, **13**(2): 87–127, 1999.

[5]. E. K. Burke and S. Petrovic. Recent research directions in automated timetabling. European Journal of Operation Research, **140**(2): 266-280, 2002.

[6]. W. Erben,J. Keppler. A genetic algorithm solving a weeklycourse timetabling problem. *Proc. of the ist Int. onf. on Practice and Theory of Automated Timetabling*, LNCS 1153, pp. 198-211, 1995.

[7]. 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. St¨utzle. A comparison of the performance of different metaheuristics on the timetabling problem. *Lecture Notes in Computer Science 2740*, pp. 329–351,2002.

[8]. O. Rossi-Doria and B. Paechter. A memetic algorithm for university course timetabling. *Proceedings of Combinatorial Optimization (CO 2004)*, pp. 56. 2004.

[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. Schearf. A survey of automated timetabling. *Artificial Intelligence Review*, **13**(2): 87–127, 1999.

8. About the Authors

Ravi Teja CH is currently pursuing his 2 Years M.Tech (CSE) in Computer Science and Engineering at University College of Engineering, Vizianagaram JNTUK. His area of interests includes Data Mining