Title	A clique-based algorithm for constructing feasible timetables
Author(s)	Liu, Y; Zhang, D; Chin, FYL
Citation	Optimization Methods And Software, 2011, v. 26 n. 2, p. 281-294
Issued Date	2011
URL	http://hdl.handle.net/10722/129976
Rights	Creative Commons: Attribution 3.0 Hong Kong License

A clique-based algorithm for constructing feasible timetables

Yongkai Liu, Defu Zhang and Francis Y.L. Chin

Department of Computer Science, Xiamen University, Xiamen 361005, China

Department of Computer Science, University of Hong Kong, Hong Kong

Abstract

Constructing a feasible solution, where the focus is on “hard” constraints only, is an important part of solving timetabling problems. For the University Course Timetabling Problem (UCTP), we propose a heuristic algorithm to schedule events to timeslots based on cliques, each representing a set of events that could be scheduled in the same timeslot, which the algorithm constructs. Our algorithm has been tested on a set of well-known instances, and the experimental results show that our algorithm compares favorably with other effective algorithms.

Keywords: timetabling; feasible timetable; heuristic; sequential techniques; clique

1. Introduction

The educational timetabling problem is concerned with the scheduling of a number of events (courses, lectures, examinations) into limited resources, such as rooms and timeslots, subject to a set of constraints [1]. Some events have students in common so that they cannot be scheduled into the same timeslot. This is one instance of a so-called “hard” constraint. Another hard constraint is that the event must be scheduled into a room which satisfies its requirements (e.g. in terms of the room capacity or equipment available in the room). A timetable which satisfies all the hard constraints is called a feasible solution. Once the feasible timetable is constructed, a further step is to improve the quality of the timetable by considering additional requirements called soft constraints, such as spreading out the events which involve the same students or teachers so that students and teachers spread their workload throughout the week. Soft constraints are not compulsory, but they should be satisfied as much as possible.

Many efficient algorithms, such as traditional Integer Programming [2], [3], [4] Constraint Logic Programming [5], [6] and meta-heuristic algorithms [7-23], have been designed to solve the educational timetabling problem. It should be noted that the University Course Timetabling Problem (UCTP) competition, organized by Metaheuristics Network and sponsored by PATAT in 2002 [7], has greatly promoted the application of meta-heuristic algorithms. The first-place winner of the UCTP competition was a Simulated Annealing (SA) based algorithm designed by Philipp Kostuch [9] with a special temperature cooling strategy. The third place winner was also a SA-based strategy designed by Yuri Bykov [19] who modified the SA-variant called the Great Deluge (GD) algorithm, which was first employed by E. Burke [18] to solve examination timetabling problems. To the best of our knowledge, SA was first used by Abramson et al. [8] to

construct timetables in 1991. It has been proved that SA can efficiently solve many kinds of timetabling problems in [9-13]. Tabu Search (TS) based algorithms proposed by Brigitte Jaumard et al.[16] and by Luca Di Gaspero [17] ranked second and fourth in the UCTP competition respectively. While it is true that the TS approach could construct high-quality timetables, more effort was needed to design the Tabu mechanisms. Although genetic algorithms (GA) did not perform well in the UCTP competition, they have been shown to be effective for various other timetabling problems [14], [25]. For instance, Grigorios et al. [14] proposed a special genetic algorithm, which does not use traditional crossover but only adopts the mutation operator, to solve the Greek high-school timetabling problems, and experimental results show that GA was more effective than the column generation approach presented in [15]. In the literature, other algorithms such as Ant Colony Algorithm [20], [21], Neural Networks [22], and Artificial Immune Algorithms [23] were also successfully applied to timetabling problems.

Most of the papers consider both hard constraints and soft constraints together. A popular method [14], [24] is to integrate all the constraints into a single objective function where hard constraints are associated with a much higher cost coefficient than soft constraints. Another method [25] is to employ a two-stage approach, in which the first stage deals with constructing a feasible timetable, and the second stage tries to minimize the violations of soft constraints. In most cases constructing a feasible solution is considered more important than reducing the violations of soft constraints, and the infeasible timetables are rarely acceptable in practice. However, as [26] points out, there are only few papers concentrating solely on constructing feasible solutions. In this paper, we focus on constructing feasible solutions.

Two events are conflicting if there is an overlap in the students attending them. In general, conflicting events are not allowed to be scheduled in the same timeslot, and it is often defined as a hard constraint. If we only consider this constraint, the timetabling problem can be regarded as a particular case of graph coloring [27] of *the conflict graph*, in which events are represented by vertices, conflicts between events are represented by edges, timeslots are represented by the colors, and allocating timeslots to events can be represented by assigning colors to vertices with no adjacent vertices having the same color. Thus unsurprisingly, some timetabling algorithms are based on graph coloring algorithms with additional processing done to take into account the timetabling problem's other constraints (e.g. the suitable room constraint). In this paper, we do not rely on graph coloring but instead introduce a clique-based heuristic. To the best of our knowledge, there are very few papers which employ clique-based heuristics to solve timetabling problems.

The sequential techniques [28], [29] are graph coloring based constructive methods, in which the events are first ordered by some metric [30], [31] (such as Largest Degree first, Least Saturation Degree first or Largest Enrolment first) and then the events are scheduled into the resource one by one. Often not all events can be scheduled and so further processes are required to

handle the unscheduled events. Carter et al. [32] employed a backtracking technique to reverse earlier assignments of events in order to release resources for unscheduled events. Kostuch [9] designed a five-step approach – which included the initial attempt, improvement attempt, shuffling, blow-ups and opening the last slots – to construct a feasible timetable using less than the specified number of timeslots.

Instead of scheduling events one by one, our approach tries to schedule all the independent events at one step by considering cliques of the complement of the conflict graph. These independent events will then be allocated to different rooms by bipartite matching. Our clique-based approach is not only simple but also provides a framework for two heuristic steps, the recombining step and the perturbing step, to efficiently expand the size of the cliques formed and in so doing reduce the number of unscheduled events.

Recently, Lewis et al [1] created 60 test instances which were "harder" than those used in the UCTP competition in the sense that feasible solutions were harder to construct. In [1], the authors reported that some traditional sequential techniques could only schedule about 80% of the events. Thus far, there are no algorithms that can solve all 60 test instances. Since some timetabling problems, upon removing the hard constraint of allocating suitable rooms for events, transform into very hard graph coloring problems, it is not surprising that the sequential techniques may fail to construct feasible timetables for some instances. We have applied our heuristic on the 60 test instances and have compared our results with that of the four algorithms proposed in [26] and [1]. Our algorithm, besides giving comparable, if not better, performance in almost all instances (except only one) takes less running time.

The rest of this paper is organized as follows. In Section 2, we give the problem definition and evaluation criteria. In Section 3, we present our algorithm based on cliques. In Section 4, we analyze the experimental results of our algorithm when tested on the 60 test instances given in [1] and compare such results with the algorithms given in [26] and [1]. Section 5 concludes.

2. Problem definition and evaluation criteria

The UCTP is defined to schedule a set of events E into a set of timeslots T and a set of rooms R, subject to a set of constraints H. The problem considered by this paper is a particular version of UCTP, in which the goal is to construct a feasible timetable where the hard constraints are as follows:

H1. Every event must be scheduled into a suitable room which meets its requirements.

H2. No student is allowed to attend more than one event in the same timeslot.

H3. No room is allowed to be occupied by more than one event in the same timeslot.

Every room possesses a set of features, such as room size and availability of certain

equipment. Correspondingly, every event must be allocated to a room that meets its requirements. So the number of available rooms for each event is fixed for a particular timetabling problem. A feasible timetable should contain all the events without violating any hard constraints. The number of timeslots in UCTP is assumed to be 45 (this parameter, which is based on 9 timeslots per day on a 5-day week, can be readily changed), but when it is hard to schedule all the events into the timetable while keeping its feasibility, some extra timeslots T' (artificial timeslots) may be added to satisfy the assignment of the unscheduled events. There are various ways to calculate the penalty of scheduling events into artificial timeslots. In [1], the cost function is the sum of the number of the artificial timeslots and the number of events scheduled in T'. In [26], the cost function is the sum of the students scheduled in artificial timeslots, but the final goal is to minimize the number of events scheduled in T'. In order to make a comparison with these papers, the evaluation criteria adopted in this paper is the distance to the feasibility [1], i.e., the number of events scheduled in artificial timeslots T'.

3. A clique-based algorithm

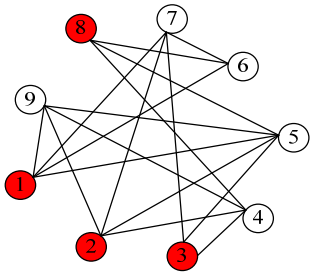
To the best of our knowledge, there are very few papers which employ clique-based heuristics to solve timetabling problems. In [32], Carter et al. pointed out that a clique, a sub-graph where the vertices were adjacent each other, could represent a set of mutually conflicting events in a timetabling problem. This set of events had to be scheduled into different timeslots, so the minimal number of timeslots used would be not less than the size of any clique in the conflict graph. In their algorithm, a large clique was first determined and the examinations in this clique had higher priority to be scheduled. In [33], Carter and Johnson observed that there were many large cliques in the timetabling instances tested by them. They concluded that cliques may help to extend some traditional approaches for timetabling. Inspired by their ideas, we further develop a clique-based algorithm for constructing feasible timetables.

Let $\overline{G}$ be the conflict graph corresponding to a timetabling problem (which essentially captures H2-type hard constraints); G be the complement graph of $\overline{G}$; V be the vertices of G; $N(v)$ be the vertices adjacent to vertex v; $N(c)$ be the vertices adjacent to vertices in clique c; $d(v)$ be the degree of v; $w(v)$ be the weight of v, namely the number of students attending the corresponding event; $d(c)$ be the degree of clique c, namely the sum of the degrees of all vertices in c; $w(c)$ be the weight of clique c, namely the sum of the weights of all vertices in c; and $s(c)$ be the size of clique c, namely the number of vertices in c. The vertices of a clique c in the complement graph G form an independent set s in the original graph $\overline{G}$. The vertices in s are not adjacent to each other, so they are conflict-free and can be colored with the same color; that is, the corresponding set of events can be scheduled into the same timeslot. Notably, our algorithm will consider cliques in complement graph G rather

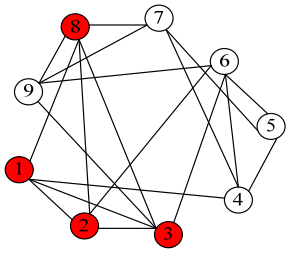
than cliques in graph $\overline{G}$ (unlike [32] and [33]).

One can easily see that a particular timeslot in a feasible timetable contains a set of events t that correspond to a clique of vertices c in the complement graph. So without the constraint of room allocation, the problem of constructing a feasible timetable for UCTP is equivalent to the problem of dividing the graph into 45 non-intersecting cliques. Once a set of conflict-free events t (corresponding to c) is found, the room allocation for events t can be handled by running a maximum matching algorithm $matchRoom(t)$ (corresponding to $matchRoom(c)$) on the bipartite graph of events and rooms ($E \times R$) with edges connecting events with their suitable rooms. Events which cannot be matched to a room can be left for further consideration. We say a clique is *legal* if the corresponding events are conflict-free and all of them are matched to a room after the maximum matching.

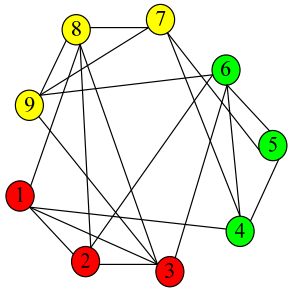
To better explain the relationship between cliques and a feasible timetable, we give an example (see Fig. 1). Suppose there are 9 events $E = \{e_1, e_2, ..., e_9\}$, 3 timeslots $T = \{t_1, t_2, t_3\}$, and 3 rooms $R = \{r_1, r_2, r_3\}$ in a given timetabling problem. Let $\overline{G}$ in Fig.1 (a) be the graph of the timetabling problem and G (which is shown Fig.1 (b)) be the complement graph of $\overline{G}$. The clique $c = \{e_1, e_2, e_3, e_8\}$ in G is an independent set $s = \{e_1, e_2, e_3, e_8\}$ in $\overline{G}$. Note that the corresponding events in s are conflict-free, so they can be scheduled into the same timeslot. We can apply $matchRoom(c)$ to allocate the rooms to the events (having regard to the H1-type hard constraints). Since there are only 3 available rooms, at least one of the 4 events will not be matched to a room. Suppose event e_8 is not matched to a room in the maximum matching. We then remove e_8 from c and consider putting this event into another clique. In Fig.1 (c), we observe that actually G can be divided into 3 cliques $c_1 = \{e_1, e_2, e_3\}$, $c_2 = \{e_4, e_5, e_6\}$ and $c_3 = \{e_7, e_8, e_9\}$. If all of these cliques are legal cliques, then we can easily construct a feasible timetable shown (for example, the one shown in Fig.1 (d)).



(a) The problem graph $\overline{G}$



(b) The complement graph G



(c) The graph is divided into 3 cliques

	t_1	t_2	t_3
r_1	e_1	e_4	e_7
r_2	e_2	e_5	e_8
r_3	e_3	e_6	e_9

(d) A feasible timetable

Fig.1 The relationship between feasible timetable and the cliques of graph

Clearly, one of the key steps of the algorithm is finding cliques. The approach we use to find a clique c in the graph G, $findClique(G,V,c)$, is similar to the algorithm of finding the largest clique proposed in [34]. However, it is not necessary to find the largest clique because the number of events in a timeslot should not exceed the number of rooms. So the backtracking step of the original approach in [34] is eliminated to save processing time. We start with a clique c (which may be empty or may contain some vertices already) and a set of vertices $V \cap N(c)$ (with $N(c) = V$ if c is empty) belonging to the graph G. We repeatedly (i) remove the vertex v with the highest degree from V and add it into c, then (ii) replace V with $V \cap N(v)$ (that is, all the vertices which are not adjacent to v are removed from V**)** until the clique stops expanding when V becomes empty. We finally get a clique c.

For example, in Fig.1 (b), if we begin with an empty clique c, then the vertex e_8 in V with the highest degree (if there are more than one, we randomly select one), is firstly added into c, and at the same time, the former V is replaced by $V \cap N(e_8) = \{e_1, e_2, e_3, e_7, e_9\}$. After this step, e_3 becomes the vertex in V with the highest degree**,** so we add it into c and replace V with $V \cap N(c = \{e_8, e_3\}) = \{e_1, e_2\}$. The next vertex added into c will be e_2, followed by e_1, and so on. We finally obtain a clique $c = \{e_8, e_3, e_2, e_1\}$ when V becomes empty. However, if c is not empty at the beginning with, say, vertex e_5 already in the clique, then the first vertex added into the clique will be e_6, which has the highest degree in $V \cap N(c = \{e_5\}) = \{e_4, e_6, e_7\}$. The clique obtained will be $c = \{e_5, e_6, e_4\}$.

Note that $findClique(G,V,c)$, which is used repeatedly in our algorithm, is not new and may not be the best way to find cliques in a graph, but its simplicity makes our algorithm easier to understand, and it is also very fast, so we adopt it to enhance the computational speed.

We now focus on the problem of dividing the graph G into 45 legal cliques $C = \{c_1, c_2, ..., c_{45}\}$ and describe our algorithm in detail. There are three steps in our algorithm. The pseudo-code of the algorithm is given in Fig.2.

The first step: Initializing

The first step is the initialization of the 45 cliques. At the start, we have 45 empty cliques $C = \{c_1, c_2, ..., c_{45}\}$, a graph G and all vertices V. We will initialize the empty clique in C one by one. For the initialization of clique c_i, instead of selecting the vertex with highest degree as the first vertex, we randomly choose a vertex v_i from V so as to spread out the initialized cliques on the graph. Then the clique is expanded using $findClique(G, V, c_i)$ as described above. (So, the second vertex added into c_i will be the one with the highest degree in $N(v_i)$.) Once the clique c_i is obtained, we run $matchRoom(c_i)$ on it. The unmatched vertices are removed from c_i while the matched vertices are removed from V. If there are no vertices left in V after this initialization, we have succeeded in constructing a feasible timetable. However, it is usually the case that some vertices cannot get into any clique. They will be handled in the next steps.

The second step: Recombining

The second step is to try to recombine the cliques. This step plays a key role in our algorithm. There are two main reasons that a vertex cannot be added into any clique. One reason is that it cannot be matched to a room. The other reason is that it is not adjacent to some vertices in the clique. Based on these situations, we design a process called $recombining(c_i)$ to enlarge a clique c_i. The idea of recombining is to obtain a larger clique by removing a portion of vertices $\bar{c}_i$ from current one c_i and use $findClique(G, V, c_i \setminus \overline{c_i})$ to expand it to be a bigger one. For each vertex v_{ij} in c_i, whether it should be removed or not from the clique is decided by a probability ρ ($0 < \rho < 1$); in particular, the vertex will be removed from c_i and added into $\bar{c}_i$ when ρ is greater than a random real number between 0 and 1(using $rand()$ to implement). The probability ρ is high initially, but it decreases by multiplying a deterioration rate α ($0 < \alpha < 1$) after every N loops, so the vertices being removed become fewer and fewer and the cliques tend to be stable in later phase. However, we may risk getting a smaller clique. To prevent this from happening, an acceptance criterion, in which the new clique c_i' obtained by $recombining(c_i)$ is accepted only when $s(c_i) \le s(c_i')$ or $w(c_i) \le w(c_i')$ or $d(c_i) \le d(c_i')$, is designed to guide the search. The reason we accept the new clique with greater weight is that there are more students scheduled into the timetable. The clique with fewer degrees is also accepted because it seems that a vertex with low degree has fewer cliques to go to, so it should be settled into a clique earlier. Note that the order in which vertices in c_i' are considered has some impact on the maximum matching, so the vertices in c_i' will be randomly ordered before we run $matchRoom(c_i')$. We will try K times until we get an acceptable clique; otherwise, we refuse the new clique c_i'.

In this step, there are a number of inner loops. For each cycle of the inner loop, we randomly order the cliques in C, and run the recombining process on each clique one by one according to that random order. After each inner loop, we lower ρ with the deterioration rate α

($\rho = \rho * \alpha$) and start a new inner loop. This step will end when ρ reaches 0.01 or when there are no vertices left in V .

The third step: Perturbing

The third step will try to swap some vertices between two cliques so that more vertices can be reinserted into these two cliques. The idea of perturbing between bipartite graphs is actually the same as the switching between timeslots. It is not the main part of our algorithm but can be quite helpful when there are still a few vertices that cannot get into any clique after the second step. Recombining is actually a hill climbing exercise: the number of unscheduled events decreases sharply in a short time, but little improvements can be made in later phases when the process drops into a local optimum. Perturbing brings a lot of benefits when this happens. For stubborn cases where there are still vertices which do not belong to any clique after the third step, the idea is that the recombining process and the perturbing process will be run alternatively for as many times as time allows.

The perturbing step consists of L loops. For each loop of perturbing, we randomly select M pairs of cliques and mark all of them unvisited. For each unvisited pair of cliques, we randomly select a vertex v from the first clique c_i and push it into the second clique c_j . The vertices $\bar{v} = c_i \setminus N(v)$ which are not adjacent to v in c_i are popped out and pushed into c_i . After this process, $c_j^{'} = c_j \cup \{v\} \setminus \bar{v}$ is a new clique but $c_i^{'} = c_i \cup \bar{v} \setminus \{v\}$ may not be a clique. However, if both $c_i^{'}$ and $c_j^{'}$ are legal, we accept $c_i = c_i^{'}$ and $c_j = c_j^{'}$ and try to add each of the vertex in V into c_i or c_j one by one, while keeping the legality of these two cliques. Otherwise, we refuse this swap.

Fig.2 The pseudo-code of the algorithm

The first step: Initializing

for $i \leftarrow 1$ **to 45**

$v_i \leftarrow$ randomly select a vertex from V

$c_i \leftarrow c_i \cup \{v_i\}$

$c_i \leftarrow findClique(G, V, c_i)$ //return a legal clique

$c_i \leftarrow matchRoom(c_i)$ //return the matched events

$V \leftarrow V \setminus c_i$

The second step: Recombining

initialize ρ , α , N and K

while (ρ >0.01 **and** V is not empty)

for $n \leftarrow 1$ **to** N

randomly order the cliques in C

for $i \leftarrow 1$ **to** 45

$c \leftarrow c_i$ // backup c_i before recombining

$done \leftarrow false$

for each vertex $v_{ij} \in c_i$

if ($\rho > rand()$)

$\bar{c}_i \leftarrow \bar{c}_i \cup \{v_{ij}\}$ // $\bar{c}_i$ is the collection of removed vertices

$c_i \leftarrow c_i \setminus \{v_{ij}\}$ //remove v_{ij} from c_i

$c_i^{'} \leftarrow recombining(c_i)$ //return the new clique after recombining

for $k \leftarrow 1$ **to** K

$c_i^{'} \leftarrow matchRoom(c_i^{'})$ //return the matched events

if ($s(c_i) \leq s(c_i^{'})$ **or** $w(c_i) \leq w(c_i^{'})$ **or** $d(c_i) \leq d(c_i^{'})$)

$c_i \leftarrow c_i^{'}$

$V \leftarrow V \cup c \setminus c_i$ //update the vertices in V

$done \leftarrow true$

Break

if ($done = false$ **)**

$c_i \leftarrow c$ // we do not accept the new clique

$\rho \leftarrow \rho * \alpha$

The third step: Perturbing

initialize L and M

if (V is not empty)

for $l \leftarrow 1$ **to** L

randomly select M pairs of cliques and set them unvisited

select a pair of unvisited clique (c_i , c_j)

$v \leftarrow$ randomly select a vertex from c_i

$\bar{v} \leftarrow c_j \setminus N(v)$

$c_i^{'} \leftarrow c_i \cup \bar{v} \setminus \{v\}$

$c_j^{'} \leftarrow c_j \cup \{v\} \setminus \bar{v}$

if ($c_i^{'}$ and $c_j^{'}$ are legal)

$c_i \leftarrow c_i^{'}$

$c_j \leftarrow c_j^{'}$

for each vertex $v \in V$

if ($c_i \cup \{v\}$ is legal)

$c_i \leftarrow c_i \cup \{v\}$

$V \leftarrow V \setminus \{v\}$

else if ($c_j \cup \{v\}$ is legal)

$$c_j \leftarrow c_j \cup \{v\}$$
$$V \leftarrow V \setminus \{v\}$$

schedule the remaining events into artificial timeslots T'

if (there is still time) run the **Recombining** and the **Perturbing** alternately for 100 times

return the number of events scheduled in T'

4. Experimental results

Our algorithm was tested on the 60 test instances generated by Lewis et al. in [1], which can be downloaded from http://www.dcs.napier.ac.uk/~benp/centre/timetabling/harderinstances.htm. It is already known that there is at least one feasible solution for each instance. The number of timeslots is fixed to be 45 and these instances are classified into three categories: small, medium and big. As in [1], the limited time for each run is set to be 30, 200, and 800 seconds for each small, medium and big instance respectively. More information about this benchmark can be obtained in [1].

We successfully constructed feasible solutions for 6 instances only after the first step. This showed that our algorithm's initializing step was not as effective when compared with the sequential technique in [26]. However, our initializing step ran in less time and created a good beginning for further improvement steps.

There are six parameters in our algorithm: α, ρ, N, K, L and M. They have a great impact on the performance of our algorithm. In order to find a good combination of values for α and ρ, we first fixed $N = 100, K = 1, L = 1000$ and $M = |E| * |E| / 16$. The initial value of α and ρ were set to be 0.8 and 0.3. At first, we fixed α but increased ρ by 0.05 for each test. From the tests, we found that the algorithm performed quite well when ρ was around 0.5. For tuning α, we also fixed ρ to be 0.6 while increasing α by 0.05 for each test. The values of α and ρ were finally fixed according to the average performance of the tests over all the instances. After we fixed α and ρ, N, K, L and M were carefully tuned according to the running time and the performance. For the recombining process, we first set $N = 100$ and $K = 1$, and then for each test, we increased N by 100 and increased K by 1 alternately. The running time of test for each combination value of N and K was set to be half of the time limit. The combination of values for N and K for each instance was recorded, and the best combination was finally selected according to their average result over all the instances. For the perturbing process, L and M were also tuned in a similar way. But L was set at 1000 and M was set at $|E| * |E| / 16$ at first, and then we increased L by 1000 and increased M by $|E| * |E| / 16$ alternately. However, the running time for each test was half of the recombining process, namely, one quarter of the time limit. In our experiments, we used the same combination of values for α, ρ, N, K, L and M, which had the best average results

over all the instances.

In the process of recombining, the deterioration rate α was fixed to be 0.95 and the probability ρ was set to be 0.6 initially. The number of cycles N for each inner loop and the number of times K for maximum matching on a clique had great impact on the performance of the algorithm. They were carefully tuned and were finally set as $N = 300$ and $K = 5$. The greater N and K were, the better results we obtained, but of course, the more CPU time was used. For the perturbing step, the loops of perturbing L was set as 10^4 and the number of pairs of cliques M was set at $|E|*|E|/4$, that is, about half of the total pairs of cliques have a chance to be perturbed. For small and medium instances, most of the unscheduled events could be inserted into timeslots after perturbing. But for big instances, we still needed to try to run the recombining process and the perturbing process alternately for 100 times.

The proposed algorithm was implemented in C++, and was run on a Pentium IV, 2.60 GHz and 512 Mb of RAM under Windows XP. We carried out 20 runs of our program for each of the 60 instances (i.e. comprising 20 small, 20 medium and 20 big instances). The best and average results were recorded and compared with HAS [26] and Lewis I-III [1]. They are shown in Table 1, Table 2 and Table 3. The column "Min" shows the minimal number of unscheduled events (the events scheduled into artificial timeslots) during the 20 runs of the algorithm. In the brackets of the same column, we give the number of times that the best solution was found over the 20 runs. The column "Ave" denotes the average number of unscheduled events over 20 runs. In the column "CPU(s)" we present the average CPU time (in seconds), but in HSA [26], they provided the minimal CPU time needed to find the best solution, and value 0 means that the constructive heuristic could find a feasible solution using just 45 timeslots after initialization. The last row of each table calculates the number of instances we succeeded in constructing feasible timetables. The columns Lewis I, Lewis II and Lewis III contain the best results from 20 runs of the grouping genetic algorithm (GGA)[1] as given in [35].

Table 1: Small Instances

Instance name	Our algorithm			HSA			Lewis I	Lewis II	Lewis III
	Min	Ave	CPU(s)	Min	Ave	CPU(s)			
S1	0(20)	0	0.05	0(20)	0	0	0	0	0
S2	0(20)	0	0.02	0(20)	0	0	0	0	0
S3	0(20)	0	1.25	0(20)	0	9	0	0	0
S4	0(20)	0	0.05	0(20)	0	0	0	0	0
S5	0(20)	0	4.60	0(20)	0	5	5	0	0
S6	0(20)	0	0.02	0(20)	0	0	0	0	0
S7	0(16)	0.2	0.02	0(20)	0	0	0	0	0

S8	**0(14)**	**0.3**	0.05	0(1)	1.9	79	12	4	0
S9	**0(17)**	**0.15**	0.02	0(2)	3.85	84	4	0	0
S10	0(20)	0	1.25	0(20)	0	15	0	0	0
S11	0(20)	0	0.02	0(20)	0	0	0	0	0
S12	0(20)	0	2.75	0(20)	0	0	0	0	0
S13	**0(20)**	**0**	1.00	0(9)	1	15	0	0	0
S14	**0(7)**	**0.7**	70.55	3(1)	5.95	136	17	3	0
S15	0(20)	0	0.55	0(20)	0	0	0	0	0
S16	0(20)	0	0.35	0(20)	0	13	0	0	0
S17	0(20)	0	2.00	0(20)	0	13	0	0	0
S18	0(6)	0.7	51.65	0(11)	0.45	36	3	3	0
S19	**0(20)**	**0**	1.00	0(11)	1.2	25	3	3	0
S20	0(17)	0.15	70.85	0(20)	0	0	0	0	0
Total	**20**			19			14	18	20

The timetabling problem is an instance of a larger family of grouping problems [1], [36], where a set of items S required to be partitioned into a collection of mutually disjoint groups s_i, such that:

$$\cup s_i = S \quad \text{and} \quad s_i \cap s_j = \phi \quad \text{for} \quad i \neq j$$

Falkenauer [36] pointed out that when the traditional genetic algorithm (GA) is applied to such grouping problems, there may be high redundancy in representations and operators, so they proposed a variation of GA named grouping genetic algorithm (GGA) and first applied it to graph coloring problems. Recently, an improved GGA (Lewis I, Lewis II and Lewis III) proposed by Lewis et al. [1] was also successfully applied to the timetabling problem. In Lewis's algorithm, each timetable is coded as a two dimensional matrix where rows represent rooms and columns represent timeslots. The structure of the algorithm is similar to GA, but they designed a special genetic cross operator. This operator comprises four stages: point selection, injection, removal of duplicates using adaptation, and reconstruction. They also employed a new method for measuring population diversities and distances between individuals with the grouping representation. These mechanisms are well designed according to the features of the grouping problems and it was proved to be efficient in constructing feasible solutions for UCTP.

Table 2: Medium Instances

Instance	Our algorithm			HSA			Lewis	Lewis	Lewis
name	Min	Ave	CPU(s)	Min	Ave	CPU(s)	I	II	III
M1	0(20)	0	5.85	0(20)	0	0	0	0	0
M2	0(20)	0	1.5	0(20)	0	0	0	0	0

M3	0(20)	0	5.05	0(20)	0	8	0	0	0
M4	0(20)	0	2.05	0(20)	0	3	0	0	0
M5	0(20)	0	59.75	0(20)	0	85	8	0	0
M6	0(20)	0	7.95	0(20)	0	20	15	0	0
M7	**0(1)**	**3.55**	134.45	1(1)	4.15	440	41	34	14
M8	0(20)	0	11.35	0(20)	0	12	21	9	0
M9	**0(1)**	**2.15**	123.2	0(1)	4.9	269	30	17	2
M10	0(20)	0	0.35	0(20)	0	0	0	0	0
M11	0(20)	0	3.4	0(20)	0	25	12	0	0
M12	0(20)	0	6.5	0(20)	0	54	0	0	0
M13	**0(20)**	**0**	9.2	0(12)	0.5	172	23	3	0
M14	0(20)	0	10.9	0(20)	0	59	0	0	0
M15	**0(20)**	**0**	7	0(19)	0.05	72	10	0	0
M16	**0(15)**	**0.3**	21.65	1(2)	5.15	733	50	30	1
M17	**0(20)**	**0**	1.8	1(20)	0	239	21	0	0
M18	**0(20)**	**0**	8.65	0(2)	6.05	429	15	0	0
M19	**0(14)**	**0.3**	16.45	0(3)	5.45	511	51	0	0
M20	**0(13)**	**0.65**	24.55	2(1)	10.6	457	15	0	3
Total	**20**			19			7	15	16

HSA [26] initializes the timetable by using two graph coloring heuristic methods: Largest Degree first (LD) and Least Saturation Degree first (LSD). When there are still events that cannot be scheduled within 45 timeslots after initialization, it opens additional timeslots, called artificial timeslots, to hold the remaining events. However, scheduling the events into the artificial timeslots will be punished by a objective function $f(s)$, which calculates the number of events assigned in the artificial timeslots. In the improving phase, HSA tries to reduce $f(s)$ by combining SA and three neighborhood operations: change the assignment of one event, swap two events, and perform the modified kempe chain move. This kempe chain move operates between two randomly selected timeslots. The events between both timeslots are connected if they conflict or they are scheduled in the same room. Before this move is performed, an event is randomly selected, a chain is triggered from this event to its connected events, and this chain is triggered alternately between these two timeslots. From the performance of HSA, one can conclude that this operation works quite well when it is combined with SA.

Table 3: Big Instances

Instance name	Our algorithm			HSA			Lewis I	Lewis II	Lewis III
	Min	Ave	CPU(s)	Min	Ave	CPU(s)			
B1	0(20)	0	18.4	0(20)	0	0	0	0	0

B2	0(20)	0	94.2	0(20)	0	283	0	0	0
B3	0(20)	0	55.4	0(20)	0	447	0	0	0
B4	0(20)	0	133.5	0(20)	0	406	32	30	8
B5	1(2)	3.2	353.1	0(6)	1.1	743	31	24	30
B6	10(1)	15.4	319.25	5(1)	8.45	893	90	71	77
B7	**39(2)**	**46.65**	383.85	47(1)	58.3	966	150	145	150
B8	0(20)	0	136.75	0(20)	0	210	35	30	5
B9	**0(20)**	**0**	122.6	0(19)	0.05	419	26	18	3
B10	0(3)	1.95	319.6	0(6)	1.25	660	36	32	24
B11	0(2)	2.35	271.7	0(14)	0.35	444	43	37	22
B12	0(20)	0	54.55	0(20)	0	240	4	0	0
B13	0(20)	0	84.15	0(20)	0	274	23	10	0
B14	0(20)	0	67.8	0(20)	0	271	8	0	0
B15	0(20)	0	83.95	0(20)	0	255	120	98	0
B16	**0(20)**	**0**	46.85	0(2)	2	755	120	100	19
B17	**0(6)**	**2.05**	554.35	76(1)	89.9	998	260	243	163
B18	**0(4)**	**1.7**	437.95	53(1)	62.6	764	199	173	164
B19	**40(1)**	**53.2**	410.45	109(1)	127	998	262	253	232
B20	**9(2)**	**14.05**	370.5	40(1)	46.7	827	186	165	149
Total	**15**			14			3	5	7

For the big instances, we obtained better results for almost all instances in comparison with Lewis I, Lewis II and Lewis III. Compared with HSA, our algorithm obtained improved results (shown in bold font) for about half of the big instances. But for some small instances, such as S18 and S20, our algorithm returned fewer feasible solutions in 20 runs. For B6, our algorithm dropped into local optima and returned a worse solution in the end. Our algorithm also failed to construct a feasible timetable for B5 while HSA was able to return one. However, it is worth mentioning that, for some of the hardest instances, such as B17 and B18, our algorithm was still able to construct feasible solutions, whereas HSA was not able to do so. For B16, B19 and B20, our algorithm performed better than HSA. This shows the advantage of our algorithm in solving big instances. For other instances, our algorithm also performed no worse than HSA. On the whole, our algorithm successfully constructed feasible timetables for almost all instances in short time. We conclude that our algorithm is robust and efficient in constructing feasible solutions for timetabling problems.

5. Conclusion and future work

Our algorithm has been successfully applied to construct feasible solutions for difficult

timetabling problems. The computational results demonstrate our algorithm's high efficiency, and also proved that it could compete with other effective algorithms. The recombining process introduced in our algorithm can be easily extended and applied to solve more timetabling problems.

Acknowledgments

The authors thank the anonymous referees for their helpful comments and suggestions which contributed to the improvement of the presentation and the contents of this paper. The authors also thank Dr. Bethany Chan for improving the readability and clarity of the paper. This work was supported by the National Nature Science Foundation of China (Grant no. 60773126) and the Province Nature Science Foundation of Fujian (Grant no. A0710023).

References

[1] Rhydian Lewis and Ben Paechter. Finding Feasible Timetables Using Group-Based Operators. IEEE Transactions on Evolutionary Computation, 11(3), 397-413 (2007).

[2] Daskalaki S, Birbas T. Efficient solutions for a university timetabling problem through integer programming. European Journal of Operational Research, 160, 106-120 (2005).

[3] Daskalaki S, Birbas T, Housos E. An integer programming formulation for a case study in university timetabling. European Journal of Operational Research, 153, 117-135 (2004).

[4] Al-Yakoob SM, Sherali HD. A mixed-integer programming approach to a class timetabling problem: A case study with gender policies and traffic considerations. European Journal of Operational Research, 180(3), 1028-1044 (2007).

[5] Gonzalez-del-Campo R, Saenz-Perez F. Programmed Search in a timetabling problem over finite domains. Electronic Notes in Theoretical Computer Science, 177, 253-267 (2007).

[6] C. Valouxis, E. Housos. Constraint programming approach for school timetabling. Computers & Operations Research, 30, 1555-1572 (2003).

[7] http://www.idsia.ch/Files/ttcomp2002/. (2005).

[8] Abramson D. Constructing school timetables using simulated annealing: Sequential and parallel algorithms. Management Science, 37(1), 98-113(1991).

[9] Philipp Kostuch. The University Course Timetabling Problem with a Three-Phase Approach. Proceedings of the 5th International Conference on Practice and Theory of Automated Timetabling V LNCS 1153, Berlin: Springer, 109-125 (2004).

[10] School timetables: A case study in simulated annealing. In V. Vidal (Ed.), Applied simulated annealing (pp. 103-124). Lecture notes in economics and mathematics systems, Berlin: Springer, Chapter 5.

[11] Abramson D, Krishnamoorthy M, Dang H. Simulated annealing cooling schedules for the school timetabling problem. Asia-Pacific Journal of Operational Research, 16, 1- 22(1999).

[12] Yongkai Liu, Defu Zhang, Stephen C.H. Leung. A Simulated Annealing algorithm with a new Neighborhood Structure for the Timetabling Problem. To be published by the 2009 World Summit on Genetic and Evolutionary Computation (2009 GEC Summit).

[13] Pasquale Avella, Bernardo D'Auria Saverio Salerno, Igor Vasil'ev. A computational study of local search algorithms for Italian high-school timetabling. Springer Science Business Media, LLC (2007).

[14] Beligiannis GN, Moschopoulos CN, Kaperonis GP, Likothanassia SD. Applying evolutionary computation to the school timetabling problem: The Greek case. Computers & Operations Research, 35(4), 1265-1280 (2008).

[15] Papoutsis K, Valouxis C, Housos E. A column generation approach for the timetabling problem of Greek high schools. Journal of the Operational Research Society, 54(3), 230-238 (2003).

[16] Jean-François Cordeau, Brigitte Jaumard, Rodrigo Morales. Efficient Timetabling Solution with Tabu Search. http://www.idsia.ch/Files/ttcomp2002/results.htm. (2002).

[17] Luca Di Gaspero and Andrea Schaerf. Timetabling Competition TTComp 2002: Solver Description. http://www.idsia.ch/Files/ttcomp2002/results.htm. (2002).

[18] E. Burke, Y. Bykov, J. Newall, S. Petrovic. A Time-Predefined Local Search Approach to Exam Timetabling Problems. Computer Science Technical Report No. NOTTCS-TR-2001-6, Univ. of Nottingham, (2001).

[19] Yuri Bykov. The Description of the Algorithm for International Timetabling competition. http://www.idsia.ch/Files/ttcomp2002/results.htm. (2002).

[20] M. Dorigo and C. Blum. Ant colony optimization theory: A survey. Theoretical Computer Science, 344(2-3), 243-278 (2005).

[21] M. Eley. Ant algorithms for the exam timetabling problem. In: E.K. Burke and H. Rudova (eds). Practice and Theory of Automated Timetabling: Selected Papers from the 6th International Conference. Lecture Notes in Computer Science, 3867, 364-382 (2007).

[22] Smith KA, Abramson D, Duke D. Hopfield neural networks for timetabling: formulations, methods, and comparative results. Computers and Industrial Engineering, 44(2), 283-305 (2003).

[23] M.R. Malim, A.T. Khader and A. Mustafa. Artificial immune algorithms for university timetabling. In: E.K. Burke and H. Rudova (eds.): Proceedings of the 6th International Conference on Practice and Theory of Automated Timetabling, 234-245. August 2006, Brno, Czech Republic. (2006).

[24] Avella P, D'Auria B, Salerno S, Vasil'ev I. A computational study of local search algorithms for Italian high-school timetabling. Journal of Heuristics, 13, 543-556 (2007).

[25] Rossi-Doria, M. Samples, M. Birattari, M.Chiarandini, J. Knowles, M. Manfrin, M. Mastrolilli, L.Paquete, B. Paechter, and T. Stuitzle. A comparison of the performance of different metaheuristics on the timetabling problem. In Practice and Theory of Automated Timetabling (PATAT) IV, ser. Lecture Notes in Computer Science, E. Burke and P. de Causmaecker, Eds. Berlin, Germany: Springer-Verlag, 2740, 329-351(2003).

[26] Mauritsius Tuga, Regina Berretta and Alexandre Mendes. A Hybrid Simulated Annealing with Kempe Chain Neighborhood for the University Timetabling Problem. 6th IEEE/ACIS International Conference on Computer and Information Science (2007).

[27] D.J.A. Welsh and M.B. Powell. The upper bound for the chromatic number of a graph and its

application to timetabling problems. The Computer Journal, 11: 41-47 (1967).

[28] D. Brelaz. New methods to color the vertices of a graph. Communication of the ACM, 22(4), 251-256 (1979).

[29] West, D.B. Introduction to Graph Theory (2nd edition). Prentice Hall (2001).

[30] S. Broder. Final examination scheduling. Communications of the ACM, 7, 494-498 (1964).

[31] E. Burke, B. McCollum, A. Meisels, S. Petrovic, and R. Qu. A graph-based hyper-heuristic for educational timetabling problems. European Journal of Operational Research, 176,177-192 (2007).

[32] M.W. Carter, G. Laporte and S.Y. Lee. Examination timetabling: Algorithmic strategies and applications. Journal of Operational Research Society, 47(3), 373-383 (1996).

[33] M.W. Carter and D.G. Johnson. Extended clique initialisation in examination timetabling. Journal of Operational Research Society, 52, 538-544 (2001).

[34] Patric R. J. Östergård. A fast algorithm for the maximum clique problem. Discrete Appl. Math., 120, 197-207 (2002).

[35] http://www.dcs.napier.ac.uk/~benp/centre/timetabling/experimentalresults2.htm

[36] Falkenauer, E.: Genetic Algorithms and Grouping Problems. Wiley, Chichester (1998)