A Novel Event Insertion Heuristic for Creating Feasible Course Timetables

Moritz Mühlenthaler · **Rolf Wanka**

Abstract We propose a novel event insertion heuristic for finding feasible solutions for instances of the University Course Timetabling Problem (UCTP). We introduce and apply a new neighbourhood structure on partial timetables that permits to approach a feasible timetable. The key insight is that an event can be inserted in a time slot if all the events conflicting with it are moved to other time slots. In order to prevent our event insertion heuristic from running into local optima, a simple perturbation strategy is employed additionally. Our experimental results show that our event insertion heuristic yields superior results compared to other state-of-the-art feasible solution generation algorithms for a large number of corresponding benchmark instances.

Keywords Course Timetabling Problems · Feasible Solution Generation · Kempe Move

1 Introduction

The task of creating course timetables such that certain constraints are satisfied occurs periodically in all sorts of educational institutions such as high schools and universities. Typically the constraints to be considered are divided into hard and soft constraints. Hard constraints are "must haves," i. e., timetables which violate any of the hard constraints are considered infeasible. On the other hand, soft constraints are "nice to have," i. e., timetables with fewer soft constraint violations are more convenient for staff and students, and are thus preferred. The University Course Timetabling Problem (UCTP) is an NP-hard combinatorial optimisation problem in the setting of a university with the objective of finding a feasible timetable with minimal soft constraint violations.

In this paper, we propose a novel heuristic approach for finding feasible timetables for UCTP instances that is based on a new sophisticated neighbourhood structure for timetables. The research yielding the proposed heuristic was motivated by the authors' university administration, who posed the following question: "Do we have to rent additional rooms for

Research funded in parts by the School of Engineering of the University of Erlangen-Nuremberg.

Moritz Mühlenthaler, Rolf Wanka
Department of Computer Science, University of Erlangen-Nuremberg, Germany
E-mail: {moritz.muehlenthaler,rwanka}@cs.fau.de

a temporarily increased course load, or is there *any* way we can get away with the rooms we have?" No specific soft constraints are imposed in this particular case, the generation of a feasible timetable is sufficient. But feasible timetable generation is also an important part of solving UCTPs when soft constraints are involved. Since hard and soft constraints are typically considered in distinct phases of the optimisation process, a feasible solution is required before even considering the soft constraint violations.

The proposed event insertion heuristic called *Kempe insertion heuristic* is built around a novel neighbourhood structure for timetables, which is closely related to the distance to feasibility. The key feature of this neighbourhood structure is that each move in the neighbourhood brings a timetable closer to feasibility. The experimental results presented in Section 4 show that our Kempe insertion heuristic outperforms current state-of-the-art feasible solution generation algorithms with respect to solution quality and computation time for the small and medium benchmark instances. This means, for such instances, solvers can spend more time on minimising soft constraint violations and thus potentially find better timetables with the same amount of computation time. For the large instances, our results are on par with the currently best performing algorithm by Tuga *et al.* [13], the HSA (hybrid simulated annealing) algorithm. For a considerable number of instances, we only need a fraction of CPU time compared to HSA. The advantage of our approach is that only *one single* sophisticated neighbourhood structure is used whereas HSA uses three different, but simple neighbourhood structures.

The remainder of this paper is organised as follows. In Section 2, the basic definitions concerning the distance to feasibility of timetables and Kempe moves are reviewed. In Section 3, the event insertion heuristic is discussed and we detail how to construct the neighbourhood structure. In Section 4, the performance of the Kempe insertion heuristic is evaluated based on the timetables generated for the 60 benchmark instances proposed by Lewis et al. in [5]. It is compared to the results of the HSA algorithm [13] and the Grouping Genetic Algorithm and the Heuristic Search Algorithm, both from [4].

2 Preliminaries

In this section we give the necessary definitions of structures and operations used by the Kempe insertion heuristic. First, we give a more formal definition of UCTPs, which were informally described above and define the notion of a *partial timetable*. We also give a short review of the *Kempe move*, which is a popular technique in educational timetabling for moving events in a timetable. The Kempe move is the main ingredient of the neighbourhood structure used by our new Kempe insertion heuristic.

2.1 Problem Definition

A UCTP instance I consists of the following data: A set $E = \{c_0, c_1, \ldots\}$ of events (or courses), a set $T = \{t_1, t_2, \ldots\}$ of time slots, and a set $R = \{r_0, r_1, \ldots\}$ of rooms. Additionally, we are given two relations $C \subseteq E \times E$ and $S \subseteq E \times R$. We say that events c and c' are in conflict if $(c, c') \in C$. We say that a room r is suitable for an event c if $(c, r) \in S$.

A *feasible timetable* for a UCTP instance I is an assignment $\tau : E \to R \times T$ of events to $(\textit{room}, \textit{time slot})$-pairs called *resources* such that each of the following *hard constraints* is satisfied:

1. Each event is assigned to a suitable resource.

2. No conflicting events occur in the same time slot.
3. No room is double-booked.

This definition is the basis of our proposed algorithm. It is consistent with the UCTP formulations for the benchmarking instances for feasible timetable generation from [5]. As indicated above, no soft constraints are to be considered for our purpose of finding a feasible timetable.

A *partial timetable* is an assignment of events to resources such that hard constraints 2 and 3 are satisfied. Additionally, a relaxed version of hard constraint 1 is imposed: we require that all assigned resources are suitable for the respective events, but not all events have to be assigned to a resource. Building on the notion of a partial timetable, feasible timetable generation can be turned into an optimisation problem for which the single objective is to minimise the number of events not assigned to a resource. Clearly, if no events remain unassigned, we have found a feasible timetable. The *distance to feasibility* of a partial timetable is the number of events which have not been assigned to a resource.

2.2 The Classical Kempe Move

The *Kempe move* [9] is a technique for moving events between time slots of a timetable. It has been applied with great success in a wide range of feasible timetable generation and soft constraint optimisation algorithms for educational timetabling [1, 7, 8, 12, 13]. The Kempe move is the basic ingredient of the neighbourhood construction in our new Kempe insertion heuristic proposed in the next section. The idea behind using Kempe moves for feasible solution generation is that as long as a few simple requirements are met, applying a Kempe move to a partial timetable results again in a partial timetable. As we will see below, the resulting timetable has the same distance to feasibility as the original timetable. Hence, no additional hard constraint violations are introduced by performing such Kempe moves. For other definitions of the distance to feasibility (see for example [4, 6, 13]), similar properties can be established.

A Kempe move is based on the identification of connected components in a bipartite graph. Let τ be a partial timetable for a UCTP instance I and let

$$E_t = \{c \mid c \in E, \exists r \in R : \tau(c) = (r,t)\}$$

be the events scheduled in time slot t. Now we consider two timeslots s and t along with the bipartite graph $G_{s,t}$ whose nodes are $E_s \cup E_t$ and whose edges are induced by the conflict relation of I. Without loss of generality, let e be any node in E_s called the *trigger event* and $D = D_s \cup D_t$ be the connected component of e with $D_s \subseteq E_s$ and $D_t \subseteq E_t$. None of the events in D_s can be moved to the time slot t as long as the events D_t are present in t without introducing conflicts that would violate the partiality requirements of τ. However, when the events in D_s are swapped with those in D_t, no such conflicts are introduced. A Kempe move between time slots s and t triggered by e swaps the events in D_s with those in D_t and hence, no conflicting events occur in E_s or in E_t after the Kempe move.

As an example, let us consider two timeslots s and t, which are populated by the events $\{c_i\}_{0 \le i < 12}$ as shown in Figure 1. Any two events connected by an edge are in conflict. The connected component of the trigger event c_0 is $\{c_0, c_3\} \cup \{c_6, c_7\}$, so the Kempe move triggered by c_0 exchanges c_0 and c_3 with c_6 and c_7. As indicated in Figure 1, the room assignment may need to be rearranged when exchanging events between timeslots. If, for example, event c_7 can only be scheduled in room 1, c_1 needs to be moved to a different

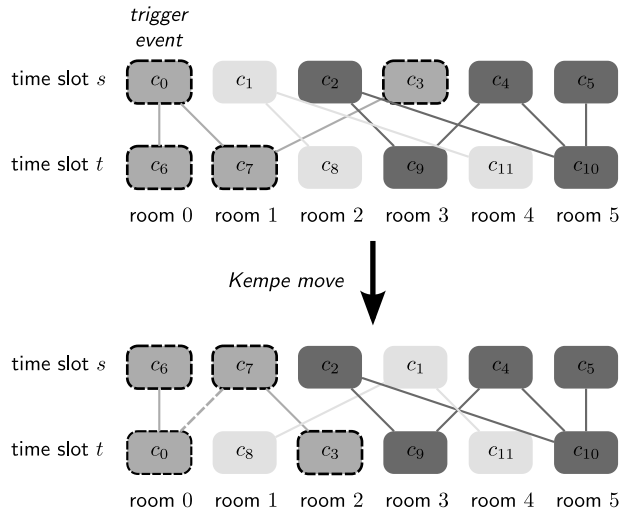

Fig. 1 A Kempe move between time slots s and t which is triggered by the event c_0. Conflicting events are connected by edges.

room. The task of assigning events to suitable rooms for a particular timeslot t can be stated in terms of a maximum cardinality bipartite matching problem, which can be solved, for instance, by the augmenting paths algorithm given in [11] in time $O(\min\{|E_t|,|R|\} \cdot |A|)$, where $|A|$ is the number of suitable room/time slot combinations. Please note that the Kempe move can also be used for soft constraint optimisation with only a slight modification: if the cost of assigning a particular event to a particular room is known, a minimum weight bipartite matching algorithm can be used to determine a room assignment with minimal cost for a particular time slot. See [7] for a UCTP solver using this technique.

For the purpose of finding feasible timetables, we have to consider the following if we want to avoid introducing additional hard constraint violations: when reassigning rooms for the events in a time slot t, each event has to be assigned to a suitable room, i. e., the cardinality of the matching has to be equal to $|E_t|$. If this is the case, we say that a Kempe move is *admissible*. If we perform an admissible Kempe move on a partial timetable, the result is again a partial timetable and its distance to feasibility, as defined in Section 2.1, is not altered by the Kempe move.

3 Feasible Timetable Generation

The new algorithm for generating feasible course timetables consists of two consecutive phases: In the first phase, a simple sequential heuristic, similar to the one used in [13], is employed for quickly creating a partial timetable with as many events assigned to suitable resources as possible. Typically for harder instances, not all events can be assigned to suitable resources by the sequential heuristic. Hence, in the second phase, we try to assign feasible resources to the remaining events using the proposed new *Kempe insertion heuristic* (see Section 3.2). This heuristic uses a computationally heavier neighbourhood exploration scheme based on the Kempe move in order to free suitable resources for the remaining events and bring the timetable gradually closer to feasibility.

3.1 The Sequential Heuristic

Sequential event insertion heuristics generate for a given UCTP instance a partial timetable. The essence of sequential heuristics is that, starting with an empty timetable, events are inserted one by one such that those events are scheduled first, which are likely to be difficult to insert in an already populated timetable. For highly constrained instances, such as the ones proposed by Lewis and Paechter in [5], sequential heuristics are very unlikely to produce feasible timetables. It turns out however that combining the new Kempe insertion heuristic with a sequential heuristic generally yields better timetables within limited computation time than the Kempe insertion heuristic alone because sequential event insertion is, in comparison, very fast.

Algorithm 1 shows the sequential heuristic used in conjunction with the Kempe insertion heuristic for obtaining the experimental results in Table 1. SEQUENTIAL_HEURISTIC takes as input a UCTP instance I, and returns a partial timetable. In the initialisation step, the events to be scheduled are sorted lexicographically according to i) the number of suitable time slots and ii) the number of suitable rooms. Hence, events with the most time slot constraints are scheduled first, and among them the ones with the least number of suitable rooms. Then the sorted list μ of events is traversed in forward order, and for each event a random suitable resource, i. e., a suitable $(\mathit{room}, \mathit{time\ slot})$-pair, is picked. If none is available for a particular event, it remains unassigned. As soon as all events in μ have been processed, the resulting partial timetable is returned.

Algorithm 1: SEQUENTIAL_HEURISTIC

```
input  : I: UCTP instance
output: τ: partial/feasible timetable
τ ← empty timetable
ρ ← list of all resources of I arranged in random order
μ ← list of all events to be scheduled
sort items in μ by i) the number of suitable time slots and ii) the number of suitable rooms
foreach event e in μ do
    if ρ contains a suitable resource for e then
        r ← first suitable resource for e in ρ
        remove r from ρ
        assign e to r
    end
end
return τ
```

The crucial part of the sequential heuristic is the determination of the order in which events are inserted in the timetable. We have tried several approaches, including those proposed by Burke *et al.* in [2], as well as the combination of "least saturation degree first"(LSD) and "largest degree first" (LD) used by Tuga *et al.* in [13]. We found that the sorting criteria as given in Algorithm 1 yield the best results in conjunction with our Kempe insertion heuristic. However, switching to the LSD/LD sequential heuristic used by Tuga *et al.* changes the results only marginally.

3.2 The Kempe Insertion Heuristic

The proposed Kempe insertion heuristic is built around a novel neighbourhood structure for partial timetable transformations, we dub *Kempe insertion neighbourhood*. A *neighbourhood* of a partial timetable τ is a collection of sequences of (admissible) Kempe moves on τ. The Kempe insertion neighbourhood has been specifically designed such that each move in the neighbourhood of a partial timetable decreases its distance to feasibility by one. Current state-of-the-art solvers typically use a combination of different neighbourhoods, each one with its individual strengths and weaknesses [3, 7, 10, 13]. For example the hybrid simulated annealing approach for feasible timetable generation by Tuga *et al.* [13] uses a combination of the *simple*, *swap* and *Kempe chain* neighbourhoods. Our objective however is to show that the Kempe insertion neighbourhood is a very good general purpose neighbourhood for feasible timetable generation, so our feasible solution generation approach relies exclusively on the Kempe insertion neighbourhood.

Let $E^- \subseteq E$ be the set of events which are yet to be scheduled in a partial timetable τ. The key observation behind the Kempe insertion neighbourhood is that an event c can be inserted in the time slot s of a timetable τ if the following two conditions are met: First, all events in s conflicting with c can be moved to a different time slot using admissible Kempe such that no additional events conflicting with c are moved to s. And second, suitable rooms can be assigned to all remaining events in s and c. So for an event c and a time slot s, we can define the set

$$\mathscr{N}_c^\tau(s) = \{K \mid K \text{ is a sequence of admissible Kempe moves involving time slot } s \\ \text{s. t. } c \text{ can be inserted in } s \text{ after performing the moves in } K\}.$$

For an event $c \in E^-$, the objective is to find a time slot s such that $\mathscr{N}_c^\tau(s)$ is non-empty. If such a time slot can be found, then c can be inserted in the timetable. Hence, the Kempe insertion neighbourhood $\mathscr{N}_c^\tau$ of a timetable τ with respect to an event $c \in E^-$ is

$$\mathscr{N}_c^\tau = \bigcup_s \mathscr{N}_c^\tau(s) \ .$$

If $\mathscr{N}_c^\tau$ is non-empty, the event c can be inserted in the timetable and as a consequence, the distance to feasibility of τ decreases by one. Hence the goal of the Kempe insertion heuristic is to make a partial timetable feasible by finding a non-empty $\mathscr{N}_c^\tau$ for each $c \in E^-$.

Figure 2 shows, by example, how a single Kempe move k with $(k) \in \mathscr{N}_c^\tau(s)$ is performed to fit an event c in a time slot s. In the example, the only event in conflict with c in s is c_2, which is moved to a time slot t by k. None of the remaining events in s is in conflict with c, so room assignment can be performed for the events $\{c\} \cup \{c_1, c_3, c_4, c_5\}$ using a maximum cardinality bipartite matching algorithm as described in the previous section.

The full Kempe insertion heuristic is outlined in Algorithm 2. INSERTION_HEURISTIC takes as input a partial timetable τ of some UCTP instance I, as well as the exploration parameter d and a time limit. It returns a feasible timetable as soon as one has been found or the best partial timetable found before the time limit is hit. In each iteration, an event e is picked at random from the list μ of unscheduled events. If some element in $\mathscr{N}_e^\tau$ can be found using NEIGHBOURHOOD_SEARCH (see Algorithm 3), e is added to the timetable, otherwise e remains unscheduled. If less than two events were successfully added to the timetable within $\max\{k, |\mu|\}$ iterations, the search is considered stuck and we insert a randomly chosen event $e \in \mu$ in the timetable "by force." This means a target time slot s is picked for e and all events $C_s(e)$ in conflict with e in s are removed from the timetable so that e can be inserted in s. This perturbation move increases the distance to feasibility by $|C_s(e)| - 1$, and therefore, we

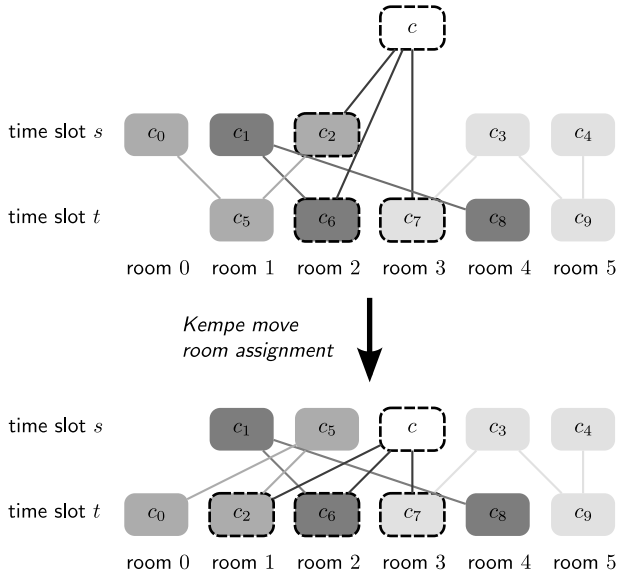

Fig. 2 Inserting an event c in the time slot s: A single Kempe move k with $(k) \in \mathcal{N}_c^\tau$ removes from s all events in conflict with c. When s has been cleared of all such events, c can be inserted in s.

do not want to use this perturbation operation too often. On the other hand, when we have tried to insert a number of events with only little success we have probably wasted CPU time because we are stuck in a local optimum. We can influence how often the perturbation operation is performed by setting the exploration parameter d, which is an upper bound for the number of iterations of NEIGHBOURHOOD_SEARCH before considering the perturbation operation depending on how many events were successfully inserted in the timetable.

The key element of INSERTION_HEURISTIC is the neighbourhood exploration shown in Algorithm 3. In NEIGHBOURHOOD_SEARCH, we try to find for an event c an element of $\mathcal{N}_c^\tau$ in a greedy fashion, so we can add c to the timetable. For each suitable time slot for c, NEIGHBOURHOOD_SEARCH tries to remove events conflicting with c by using Kempe moves. If it succeeds to clean a time slot from all such events, it checks if additional rooms need to be freed using another Kempe move. Now, if rooms can be assigned successfully to all events in $E_s \cup \{c\}$, NEIGHBOURHOOD_SEARCH has found an element of $\mathcal{N}_c^\tau(s) \subseteq \mathcal{N}_c^\tau$ and returns s. It returns "invalid time slot" to indicate that no element of $\mathcal{N}_c^\tau$ has been found. It is possible that NEIGHBOURHOOD_SEARCH cannot find a sequence of moves such that an event e can be inserted in the timetable, although $\mathcal{N}_c^\tau$ is, in principle, non-empty. However, trying to find any sequence of moves such that e can be scheduled is not computationally feasible and, as the experimental results in the next section show, NEIGHBOURHOOD_SEARCH is quite successful in finding such sequences.

In the worst case, NEIGHBOURHOOD_SEARCH tries to fit an event e in every time slot without success. Then for each time slot s and for each event c in the time slot in conflict with e, the algorithm has tried to find a time slot $t \neq s$ such that c can be moved to t without introducing new conflicts. This means, in the worst case, performing NEIGHBOURHOOD_SEARCH results in $O(|T|^2 \cdot |C|)$ attempts to remove events from a time slot using Kempe moves without finding an element of $\mathcal{N}_e^\tau$, where $|T|$ is the number of time slots and $|C|$ the number of conflicts . To increase the likeliness of INSERTION_HEURISTIC to find an

Algorithm 2: INSERTION_HEURISTIC

input : I: UCTP instance
input : k: max. number of iterations until perturbation
in/out : τ: partial timetable

$\tau_{best} \leftarrow \tau$
$\mu \leftarrow$ list of events to be scheduled
while *time limit not hit and* τ *infeasible* **do**
  success $\leftarrow 0$
  for $\max\{k, |\mu|\}$ *iterations* **do**
    e $\leftarrow$ random element from μ
    s $\leftarrow$ NEIGHBOURHOOD_SEARCH(τ, e)
    if s *is a valid time slot* **then**
      insert e in s
      success $\leftarrow$ success $+ 1$
    end
  end
  if $dist(\tau) < dist(\tau_{best})$ **then** $\tau_{best} \leftarrow \tau$
  if success ≤ 1 **then**
    pick e at random from μ
    force insertion of e in τ
    update μ
  end
end
return τ_{best}

Algorithm 3: NEIGHBOURHOOD_SEARCH

input : e: event to be scheduled
in/out : τ: partial timetable

foreach *suitable time slot* s *for* e **do**
  cleanslot $\leftarrow$ s
  `/* try to reschedule all events in s conflicting with e */`
  foreach *event* c *in* s *conflicting with* e **do**
    find a time slot t $\neq$ s s.t. there is an admissible Kempe Move which moves e to t without introducing events conflicting with e in s
    if *suitable* t *was found* **then** KempeMove(s, t, c)
    else cleanslot $\leftarrow$ invalid time slot; break
  end
  `/* if rescheduling failed, try to gather conflicting events in s */`
  if cleanslot $=$ *invalid time slot* **then**
    T $\leftarrow$ suitable time slots not yet processed
    find a time slot t in T and a trigger event c s.t. c triggers an admissible Kempe Move, which increases the number of events conflicting with e in s
    if *suitable* t *and* c *were found* **then**
      KempeMove(s, t, c); break
  end
end
if cleanslot $\neq$ *invalid time slot* **then**
  if *all rooms are booked in* cleanslot **then**
    find a time slot t and a trigger event c s.t. c triggers an admissible Kempe Move, which decreases the number of events in s without introducing events conflicting with e
    if *suitable* t *and* c *found* **then** KempeMove(s, t, c)
  end
  if *suitable rooms can be assigned to events* ($E_{\text{cleanslot}} \cup \{e\}$) **then**
    return cleanslot
end
return invalid time slot

element of $\mathcal{N}_e^\tau$ as early as possible, we do the following: If NEIGHBOURHOOD_SEARCH fails to free a time slot s from all events conflicting with e, we gather events conflicting with e from the time slots which are yet to be processed. More precisely, in the remaining time slots we look for a time slot t such that we can perform an admissible Kempe move which *increases* the number of events conflicting with e in s. Experiments indicated that gathering conflicts in this fashion improves the overall running time of INSERTION_HEURISTIC considerably when feasible solutions are found and also seems to have a beneficial impact on the overall quality of the solutions obtained.

4 Experimental Results

Our experimental results were obtained for the 60 problem instances in [5]. These instances were specifically designed to be hard to solve by sequential heuristics as described in Section 3.1. For each instance however, it is guaranteed that there exists at least one feasible solution. The 60 instances are divided in three categories: Small instances with 200 to 225 events and 5 to 6 rooms, medium instances with 390 to 425 events and 10 to 11 rooms, and large instances with 1000 to 1075 events and 25 to 28 rooms. For all instances, the number of time slots is 45, and all events can be scheduled in any of the 45 time slots if there are no conflicting events in a time slot already.

Our solutions were obtained by running SEQUENTIAL_HEURISTIC 150 times and then using the best partial solution found so far as input for INSERTION_HEURISTIC. Feasible solutions were found by the sequential heuristic for the instances *small* 2, 6, 11, 12 and 20. In comparison, in [13], Tuga *et al.* performed their sequential heuristic 500 times for each instance as a preprocessing step and found feasible solutions for 14 of the 60 instances just using the sequential heuristic. It turned out however, that increasing the number of iterations or modifying the sequential heuristic did not improve the overall solution quality in our experiments. The exploration parameter for INSERTION_HEURISTIC was set to 16 and timeout values were set to 100 s for the small instances, to 200 s for the medium instances and to 500 s for the large instances.

The results shown in Table 1 were obtained by performing 20 consecutive runs for each instance on a *single* core of personal computer equipped with a Intel QuadCore CPU clocked at 3 GHz. Running the sequential heuristic 150 times took between 0.2 s and 1.7 s, depending on the size of the instance. For each instance, the lowest and average distance to feasibility and the average CPU time were recorded. Table 1 shows our results along with the results obtained by Tuga *et al.* in [13] (HSA) on a Pentium IV 3.2 GHz, and Lewis and Paechter in [4] (Lewis I and II) for comparison. Note that different time limits were imposed in the experiments run in [13] and [4]. Tuga *et al.* set the timeouts to 200, 400 and 1000 seconds for the small, medium and large instances, respectively [13]. Lewis and Paechter imposed timelimits of 30, 200 and 800 seconds for small, medium, and large instances to obtain their results [4]. To the knowledge of the authors, no average running times were given in [4].

As shown in Table 1, to combine SEQUENTIAL_HEURISTIC and INSERTION_HEURISTIC consistently outperforms the other algorithms for the small and medium benchmark instances with respect to the distance to feasibility of the obtained solutions and CPU time used. Our Kempe insertion heuristic found feasible timetables for all 40 small and medium instances. Concerning the large instances, our algorithm performs better than both algorithms proposed in [4]. Also, our algorithm is at least as good as the HSA approach by Tuga *et al.* for 13 out of the 20 large instances despite the shorter timeout and uses much less CPU time on average for many instances such as *big* 2, 3, 12, 13, 14 and 16.

	ins. heuristic		HSA		Lewis I	Lewis II
Instance	best(avg)	avg time	best(avg)	time	best(avg)	best(avg)
small 1	0(0)	0.0	0(0)	0	0(0)	0(0)
small 2	0(0)	0.0	0(0)	0	0(0)	0(0)
small 3	0(0)	0.1	0(0)	9	0(0)	0(0)
small 4	0(0)	0.0	0(0)	0	0(0)	0(0)
small 5	0(0)	0.2	0(0)	5	0(1.05)	0(0)
small 6	0(0)	0.0	0(0)	0	0(0)	0(0)
small 7	0(0)	0.1	0(0)	0	0(0)	0(0)
small 8	0(0)	15	0(1.9)	79	4(6.45)	0(1)
small 9	0(0)	1.7	0(3.85)	84	0(2.5)	0(0.15)
small 10	0(0)	0.4	0(0)	15	0(0.1)	0(0)
small 11	0(0)	0.0	0(0)	0	0(0)	0(0)
small 12	0(0)	0.0	0(0)	0	0(0)	0(0)
small 13	0(0)	1.0	0(1)	15	0(1.25)	0(0.35)
small 14	0(0)	33	3(5.95)	136	3(10.5)	0(2.75)
small 15	0(0)	0.0	0(0)	0	0(0)	0(0)
small 16	0(0)	0.0	0(0)	13	0(0)	0(0)
small 17	0(0)	0.1	0(0)	13	0(0.25)	0(0)
small 18	0(0)	0.0	0(0.45)	36	0(0.7)	0(0.2)
small 19	0(0)	0.5	0(1.2)	25	0(0.15)	0(0)
small 20	0(0)	0.0	0(0)	0	0(0)	0(0)
med 1	0(0)	0.21	0(0)	0	0(0)	0(0)
med 2	0(0)	0.15	0(0)	0	0(0)	0(0)
med 3	0(0)	0.73	0(0)	8	0(0)	0(0)
med 4	0(0)	0.30	0(0)	3	0(0)	0(0)
med 5	0(0)	5.2	0(0)	85	0(3.95)	0(0)
med 6	0(0)	4.0	0(0)	20	0(6.2)	0(0)
med 7	0(0)	80	1(4.15)	440	34(51.65)	14(18.5)
med 8	0(0)	4.2	0(0)	12	9(15.95)	0(0)
med 9	0(0.1)	142	0(4.9)	269	17(24.55)	2(9.7)
med 10	0(0)	0.0	0(0)	0	0(0)	0(0)
med 11	0(0)	1.3	0(0)	25	3(13.35)	0(0)
med 12	0(0)	0.2	0(0)	54	0(0.25)	0(0)
med 13	0(0)	1.6	0(0.5)	172	30(43.15)	0(0.5)
med 14	0(0)	1.0	0(0)	59	0(0.25)	0(0)
med 15	0(0)	1.6	0(0.05)	72	0(4.85)	0(0)
med 16	0(0)	7.3	1(5.15)	733	30(43.15)	1(6.4)
med 17	0(0)	1.4	0(0)	39	0(3.55)	0(0)
med 18	0(0)	5.6	0(6.05)	429	0(8.2)	0(3.1)
med 19	0(0)	11	0(5.45)	511	0(9.25)	0(3.15)
med 20	0(0)	15	2(10.6)	457	0(2.1)	3(11.45)
big 1	0(0)	0	0(0)	0	0(0)	0(0)
big 2	0(0.05)	56	0(0)	283	0(0.7)	0(0)
big 3	0(0)	26	0(0)	447	0(0)	0(0)
big 4	0(1.4)	465	0(0)	406	30(32.2)	8(20.5)
big 5	5(8.4)	500	0(1.1)	743	24(29.15)	30(38.15)
big 6	29(40.3)	500	5(8.45)	893	71(88.9)	77(92.3)
big 7	100(109.2)	500	47(58.3)	966	145(157.3)	150(168.5)
big 8	0(0.25)	329	0(0)	210	30(37.8)	5(20.75)
big 9	0(0.7)	434	0(0.05)	419	18(25)	3(17.5)
big 10	9(12.5)	500	0(1.25)	660	32(38)	24(39.95)
big 11	8(10.2)	500	0(0.35)	444	37(42.35)	22(26.05)
big 12	0(0)	37	0(0)	240	0(0.85)	0(0)
big 13	0(0)	70	0(0)	274	10(19.9)	0(2.55)
big 14	0(0)	54	0(0)	271	0(7.25)	0(0)
big 15	0(11.4)	496	0(0)	255	98(113.95)	0(10)
big 16	0(0)	80	0(2)	755	100(116.3)	19(42)
big 17	13(57.5)	500	76(89)	998	243(266.55)	163(174.9)
big 18	0(0)	259	53(62)	764	173(194.75)	164(179.25)
big 19	161(171.5)	500	109(127)	998	253(266.65)	232(247.35)
big 20	6(13.0)	500	40(46.7)	827	165(183.15)	149(164.15)

Table 1 Experimental results for the 60 benchmark instances from [5]. Average time is given in seconds.

5 Conclusions

In this paper, the new Kempe insertion heuristic has been proposed for generating feasible timetables for university course timetabling problems. Our approach is based on exploring a sophisticated neighbourhood structure for partial timetables, the Kempe insertion neighbourhood. Each move in the neighbourhood structure brings the partial timetable computed so far closer to feasibility. In addition, a perturbation strategy has been proposed for preventing the Kempe insertion heuristic from getting stuck in local optima.

The Kempe insertion heuristic has been tested on the 60 benchmark instances from [5]. Our results show that our algorithm consistently outperforms other state-of-the-art algorithms for feasible solution generation [4, 13] for the small and medium benchmark instances with respect to the distance to feasibility of the timetables and CPU time used. For 13 of the 20 large instances, the Kempe insertion heuristic performs at least as good as the Hybrid Simulated Annealing algorithm from [13] despite the shorter timeout and uses much less CPU time on average for many instances. The Kempe insertion heuristic generally outperforms the Grouping Genetic Algorithm and the Heuristic Search Algorithm from [4].

References

1. Edmund K. Burke, Adam J. Eckersley, Barry McCollum, Sanja Petrovic, and Rong Qu. Hybrid variable neighbourhood approaches to university exam timetabling. *European Journal of Operational Research*, 206(1):46–53, 2010.
2. Edmund K. Burke, Barry McCollum, Amnon Meisels, Sanja Petrovic, and Rong Qu. A graph-based hyper-heuristic for educational timetabling problems. *European Journal of Operational Research*, 176(1):177–192, 2007.
3. Luca Di Gaspero and Andrea Schaerf. Neighborhood portfolio approach for local search applied to timetabling problems. *Journal of Mathematical Modeling and Algorithms*, 5(1):65–89, 2006.
4. Rhydian Lewis and Ben Paechter. Finding feasible timetables using group-based operators. *IEEE Transactions on Evolutionary Computation*, 11:397–413, 2007.
5. Rhydian Lewis and Ben Paechter. http://www.emergentcomputing.org/timetabling/harderinstances.htm, accessed 2010.
6. Rhydian Lewis, Ben Paechter, and Barry McCollum. Post enrolment based course timetabling: A description of the problem model used for track two of the second international timetabling competition. Cardiff Accounting and Finance Working Papers A2007/3, Cardiff University, Cardiff Business School, Accounting and Finance Section, July 2007.
7. Zhipeng Lü and Jin-Kao Hao. Adaptive tabu search for course timetabling. *European Journal of Operational Research*, 200(1):235–244, 2010.
8. Liam T. G. Merlot, Natashia Boland, Barry D. Hughes, and Peter J. Stuckey. A hybrid algorithm for the examination timetabling problem. In *Proc. 4th Int. Conf. on the Practice and Theory of Automated Timetabling (PATAT)*, pages 207–231. Springer, 2003.
9. Craig Morgenstern and Harry Shapiro. Coloration neighborhood structures for general graph coloring. In *Proc. 1st ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 226–235, 1990.
10. Tomáš Müller. ITC2007 solver description: A hybrid approach. *Annals of Operations Research*, 172(1):429–446, 2009.
11. Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization; Algorithms and Complexity*. Dover Publications, 1998.
12. Jonathan M. Thompson and Kathryn A. Dowsland. A robust simulated annealing based examination timetabling system. *Computers & Operations Research*, 25(7-8):637 – 648, 1998.
13. Mauritsius Tuga, Regina Berretta, and Alexandre Mendes. A hybrid simulated annealing with Kempe chain neighborhood for the university timetabling problem. In *Proc. 6th ACIS Int. Conf. on Computer and Information Science (ACIS-ICIS)*, pages 400–405, 2007.