Ann Oper Res (2009) 172: 429–446
DOI 10.1007/s10479-009-0644-y

ITC2007 solver description: a hybrid approach

Tomáš Müller

Published online: 18 October 2009

Abstract This paper provides a brief description of a constraint-based solver that was successfully applied by the author to the problem instances in all three tracks of the International Timetabling Competition 2007 (For more details see the official competition website at http://www.cs.qub.ac.uk/itc2007.). The solver presented in this paper was among the finalists in all three tracks and the winner of two.

1 Introduction

The primary objective in the construction of the search algorithm used in this timetabling competition was to demonstrate the feasibility of applying a single solution framework on a variety of important timetabling problems. The Constraint Solver Library (http://cpsolver.sf.net) that was employed contains a constraint-based framework incorporating a series of algorithms based on local search techniques that operate over feasible, though not necessarily complete, solutions. In these solutions some variables may be left unassigned. All hard constraints on assigned variables must be satisfied however. The library is written in Java and is publicly available under GNU's LGPL licence. It has also been successfully applied to several large scale practical timetabling problems, including the course timetabling system that is used at Purdue University (Murray et al. 2007; Müller et al. 2005), see http://www.unitime.org for more details. Currently, algorithms for student sectioning (Müller and Murray 2008) and examination timetabling are being developed using this library. The same algorithm was used for all three tracks of the International Timetabling Competition, with only minimal changes to reflect different problem formulations (e.g., problem model, neighborhoods, and solver parameters).

The remaining sections of this paper briefly describe the competition problems, the search algorithm, the implementation, the neighborhoods employed for the problem in each track, the results obtained for the early and late problem instances, final placement of the solver in the competition, and a few concluding remarks.

T. Müller (✉)
Purdue University, West Lafayette, IN 47907, USA
e-mail: muller@unitime.org

2 Competition

The second International Timetabling Competition consisted of three tracks, each representing a different problem in educational timetabling, namely, examination timetabling, post enrollment based course timetabling, and curriculum based course timetabling. This section provides a brief description of these problems.

2.1 Track 1: examination timetabling

The examination timetabling problem model presented in this track is an extension of the commonly used model. The fundamental problem involves timetabling exams into a set of periods within a defined examination session while satisfying a number of hard constraints. Like other areas of timetabling, a feasible solution is one in which all hard constraints are satisfied. The quality of the solution is measured in terms of soft constraint satisfaction.

The problem consists of the following:

– A list of periods covering a specified length of time. The number and lengths of periods are provided.
– A set of exams that are to be scheduled into these periods.
– For each exam, a set of enrolled students is provided. Each student is enrolled into a number of exams.
– A set of rooms with individual capacities.
– A set of additional period (e.g., exam A after exam B) and room (e.g., exam A must use room R) hard constraints.
– Soft constraints which contribute to a penalty if they are violated (including details on weightings of these constraints).

A feasible timetable is one in which all examinations have a period and a room assigned and the following hard constraints are satisfied:

– No student sits for more than one examination at a time.
– The capacity of individual rooms is not exceeded at any time during the examination session. Note that, unlike course timetabling, exams are explicitly allowed to share rooms.
– Period lengths are not violated.
– Additional hard constraints must be satisfied.

The problem includes the following soft constraints:

– *Two Exams in a Row* The number of occurrences when students have to sit for two exams in a row on the same day.
– *Two Exams in a Day* The number of occurrences when students have to sit for two exams on the same day.
– *Period spread* The number of occurrences when students have to sit for more than one exam during a time interval specified by the institution. This is often used in an attempt to be as fair as possible to all students taking exams.
– *Mixed Durations* The number of occurrences of exams timetabled into rooms along with exams with a different duration.
– *Larger Exams Constraints* The number of *large* exams appearing in the *later portion* of the timetable. Definition of *large* and *later portion* is a part of the description of a particular instance.

- *Room Penalty* The number of times a room is used which has an associated penalty. This is multiplied by the actual penalty as different rooms may have different associated weightings.
- *Period Penalty* The number of times a period is used which has an associated penalty. This is multiplied by the actual penalty as different periods may have different associated weightings.

2.2 Track 2: post enrollment based course timetabling

The timetabling problem in this track is intended to simulate the real-world situation where students are given a choice of lectures that they wish to attend, and the timetable is then constructed according to these choices (i.e., the timetable is to be constructed after students have selected the lectures they wish to attend). This model is based on the model used in the first international timetabling competition http://www.idsia.ch/Files/ttcomp2002, which was run in 2003 in conjunction with PATAT and the Metaheuristics Network.

The problem consists of the following:

- A set of events that are to be scheduled into 45 time slots (5 days of 9 hours each).
- A set of rooms, each of which has a specific seating capacity, in which the events take place.
- A set of room features that are satisfied by rooms and which are required by events.
- A set of students who attend various different combinations of events.
- A set of available time slots for each of the events (i.e. not all events can be placed in all time slots).
- A set of precedence requirements that state that certain events should occur before certain others.

The aim is to try and insert each of the given events into the timetable (that is, assign each event to one of the rooms and one of the 45 time slots) while obeying the following hard constraints:

- No student should be required to attend more than one event at the same time.
- In each case, the room should be big enough for all of the attending students and should satisfy all of the features required by the event.
- Only one event is put into each room in any time slot.
- Events should only be assigned to time slots that are pre-defined as *available* for those events.
- Where specified, events should be scheduled to occur in the correct order during the week.

Note that the first three hard constraints above are exactly the same as the hard constraints used in the first competition. The last two constraints, however, are new additions to the model.

In addition to the five hard constraints that are given above, the following soft constraints are included in the problem:

- *Last Time Slots of a Day* Students should not be scheduled to attend an event in the last time slot of a day (that is, time slots 9, 18, 27, 36, or 45).
- *More than Two in a Row* Students should not have to attend three (or more) events in consecutive time slots occurring in the same day.
- *One Class on a Day* Students should not be required to attend only one event in a particular day.

Note that these three soft constraints are the same as those used in the first competition. The overall solution penalty is the sum of occurrences of a student in the soft constraints that are violated.

In this track, it is allowable to produce an incomplete solution (some events may be left unassigned), however, all hard constraints on the assigned events must be satisfied. Unplaced events are used to calculate a *Distance to Feasibility* measure. This is calculated by identifying the number of students that are required to attend each of the unplaced events and then simply adding these values together.

2.3 Track 3: curriculum based course timetabling

The Curriculum-based timetabling problem consists of the weekly scheduling of lectures for several university courses within a given number of rooms and time periods. Conflicts between courses are determined according to the curricula published by the University and not on the basis of enrolment data.

The problem consists of the following entities:

– *Days, Timeslots, and Periods.* A number of teaching days in the week are given (typically 5 or 6). Each day is split into a fixed number of timeslots, which is equal for all days. A period is a pair composed of a day and a timeslot. The total number of scheduling periods is the product of the number of days times the number of daily timeslots.
– *Courses and Teachers.* Each course consists of a fixed number of lectures to be scheduled in distinct periods, is attended by a given number of students, and is taught by a teacher. For each course there is a minimum number of days over which the lectures of the course should be spread, moreover, there are some periods during which the course cannot be scheduled.
– *Rooms.* Each room has a capacity, expressed as the number of available seats. All rooms are equally suitable for all courses (if large enough).
– *Curricula.* A curriculum is a group of courses such that any pair of courses in the group has students in common. Conflicts between courses, and other soft constraints, are based on curricula.

The solution of the problem is an assignment of a period (day and timeslot) and a room to all lectures of each course. The following hard constraints must be satisfied:

– All lectures of a course must be scheduled, and they must be assigned to distinct periods.
– Two lectures cannot take place in the same room in the same period.
– Lectures of courses in the same curriculum, or taught by the same teacher, must be scheduled in different periods.
– If the teacher of the course is not available to teach that course in a given period, then no lectures of the course can be scheduled in that period.

The problem includes the following soft constraints:

– *Room Capacity* For each lecture, the number of students that attend the course must be less than or equal to the number of seats in all the rooms that host its lectures. Each student above the capacity counts as 1 point of penalty.
– *Minimum Working Days* The lectures of each course must be spread into the given minimum number of days. Each day below the minimum counts as 5 points of penalty.
– *Curriculum Compactness* Lectures belonging to a curriculum should be adjacent to each other (i.e., in consecutive periods). For a given curriculum we account for a violation every time there is one lecture not adjacent to any other lecture within the same day. Each isolated lecture in a curriculum counts as 2 points of penalty.

– *Room Stability* All lectures of a course should be given in the same room. Each distinct room used for the lectures of a course, beside the first, counts as 1 point of penalty.

3 Algorithm

The search algorithm consists of several phases: In the first (construction) phase, a complete feasible solution is found using an Iterative Forward Search (IFS) algorithm (Müller 2005). This algorithm makes use of Conflict-based Statistics (CBS) (Müller et al. 2004) to prevent itself from cycling. In the next phase, the local optimum is found using a Hill Climbing (HC) algorithm. Once a solution can no longer be improved using this method, the Great Deluge (GD) technique (Dueck 1993) is used. The GD algorithm is altered so that it allows some oscillations of the bound that is imposed on the overall solution value. Optionally, Simulated Annealing (SA) (Kirkpatrick et al. 1983) can also be used between bound oscillations of the GD algorithm.

The search ends after a predetermined time limit has been reached. The best solution found within that limit is returned.

3.1 Construction phase

Initially, a complete solution is found using the Iterative Forward Search algorithm (Müller 2005). It starts with all variables being unassigned. During each iteration, an unassigned variable (i.e., a class, or exam) is selected and a value from its domain is assigned to it (assignment of a room and a time). If this causes any violations of hard constraints with existing assignments, the conflicting variables are unassigned. For example, if there is another class in the selected room at the selected time, that class will be unassigned. The search ends when all variables are assigned.

The search is also parametrized by variable and value selection criteria. It first tries to find those variables that are most difficult to assign. A variable is randomly selected among unassigned variables with the smallest ratio of domain size to the number of hard constraints. Other problem-based criteria can be used as well. It then tries to select the best value to assign to the selected variable. A value whose assignment increases the overall cost of the solution the least is selected among values that violate the smallest number of hard constraints (i.e., the number of conflicting variables that need to be unassigned in order to make the problem feasible after assignment of the selected value to the selected variable is minimized). If there is a tie, one of these is selected randomly. It is also possible to completely ignore soft constraints in this phase in order to speed computation of a feasible solution. A value is then selected randomly among values that minimize the number of violated hard constraints.

Conflict-based Statistics (Müller et al. 2004) is used during this process to prevent repetitive assignments of the same values by memorizing conflicting assignments. Conflict-based Statistics is a data structure that memorizes hard conflicts which have occurred during the search together with their frequency and the assignments that caused them. More precisely, it is an array

$$CBS[V_a = v_a \rightarrow \neg V_b = v_b] = c_{ab}.$$

This means that the assignment $V_a = v_a$ has caused a hard conflict c_{ab} times in the past with the assignment $V_b = v_b$. Note that this does not imply that the assignments $V_a = v_a$ and $V_b = v_b$ cannot be used together in the case of non-binary constraints. In the value selection criterion, each hard conflict is then weighted by its frequency, i.e., by the number of past unassignments of the current value of the conflicting variable caused by the selected assignment.

3.2 Hill climbing phase

Once a complete solution is found, a Hill Climbing algorithm is used in order to find the local optimum. In each iteration a change in the assignment of the current solution is proposed by random selection from a problem-specific neighborhood. The generated move is only accepted when it does not worsen the overall solution value (i.e., the weighted sum of violated soft constraints). Only changes that do not violate any hard constraints are considered. This rule applies during all phases. Neighbor assignments are also generated consistently throughout all phases. That is, a problem specific neighborhood is selected randomly (with a given probability among the neighborhoods that have been created for the problem) and is used to generate a random change in the current solution.

The hill climbing phase is finished after a specified number HC_{idle} of idle iterations during which a solution has not improved. The parameter HC_{idle} may be defined differently for the problems in the three competition tracks.

3.3 Great deluge phase

The Great Deluge algorithm (Dueck 1993) uses a bound B that is imposed on the overall value of the current solution that the algorithm is working with. This means that the generated change is only accepted when the value of the solution after an assignment does not exceed the bound. The bound starts at the value

$$B = GD_{ub} \cdot S_{best}$$

where S_{best} is the overall value of the best solution found so far, and GD_{ub} is a problem specific parameter (upper bound coefficient). The bound is decreased after each iteration. This is done by multiplying the bound by a cooling rate GD_{cr}.

$$B = B \cdot GD_{cr}$$

The search continues until the bound reaches a lower limit equal to $GD_{lb}^{at} \cdot S_{best}$, where GD_{lb} is a parameter defining lower bound coefficient. When this lower limit is reached, the bound is reset back to its upper limit of $GD_{ub}^{at} \cdot S_{best}$.

$$B < GD_{lb}^{at} \cdot S_{best} \quad \Rightarrow \quad B = GD_{ub}^{at} \cdot S_{best}$$

The parameter at is a counter starting at 1. It is increased by one every time the lower limit is reached and the bound increased. It is also reset back to 1 when a previous best solution is improved upon. This helps the solver to widen the search when it cannot find an improvement, allowing it to get out of a deep local minimum.

3.4 Simulated annealing phase

The Simulated Annealing algorithm (Kirkpatrick et al. 1983) is applied using a temperature parameter of T. A generated neighbor assignment is accepted if it does not worsen the overall value of the current solution, or with the following probability

$$p_{accept} = e^{-(\Delta/T)}$$

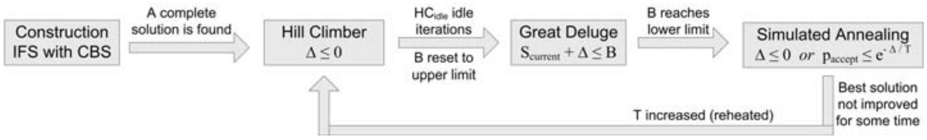
Construction IFS with CBS
A complete solution is found
Hill Climber $\Delta \le 0$
HC_{idle} idle iterations
B reset to upper limit
Great Deluge $S_{current} + \Delta \le B$
B reaches lower limit
Simulated Annealing $\Delta \le 0$ or $p_{accept} \le e^{-\Delta/T}$
Best solution not improved for some time
T increased (reheated)

Fig. 1 Algorithm schema when Simulated Annealing is used

where Δ is the increase in the overall value of the current solution when a detrimental move is assigned. The temperature T starts at initial value of SA_{it}. It is cooled down (multiplied by cooling rate SA_{cr}) after each $SA_{cc} \cdot TL$ iterations (SA_{cc} is a cooling coefficient), where TL is an instance specific number (temperature length), computed as the sum of domain sizes of all variables. If the best solution found is not improved after $SA_{rc} \cdot SA_{cc} \cdot TL$ iterations (SA_{rc} is a reheat coefficient), the temperature is increased to

$$T = T \cdot {SA_{cr}}^{-1.7 \cdot SA_{rc}}$$

In the case when simulated annealing is used, the great deluge phase is stopped after the bound B reaches its lower limit and control is passed to the simulated annealing phase. Similarly, control is passed from the simulated annealing phase back to hill climbing phase just after the system is reheated (see Fig. 1). When simulated annealing is not used, control is never passed from the great deluge phase (GD continues immediately after the bound is increased).

4 Implementation

The solver was implemented using the Constraint Solver Library (http://cpsolver.sf.net). This is a local-search based framework written in Java, where each of the competition problems was modeled using existing primitives such as variable, value, and constraint. The library works only with feasible, though not necessarily complete solutions. This means that no (hard) constraint can be violated, however some variables may be left unassigned. This feasibility is ensured automatically through the framework via notifications sent between variables and their constraints.

Figure 2 shows the core algorithm schema. It is parametrized by a termination criterion, a neighborhood selection and a solution comparator. The algorithm is terminated when a

```
while (termination.canContinue(solution)) {
        Neighbor n = neighborSelection.select(solution);
        if (n != null) n.assign(solution);
        if (solutionComparator.isBetterThanBest(solution)) solution.saveBest();
}
```

Fig. 2 Default search strategy

```
public class HillClimber implements NeighborSelection {
    private Vector<NeighborSelection> iNeighborhoods; // list of neighborhoods
    private int iIdle = 0; // number of idle iterations
    public Neighbor select(Solution solution) {
        while (iIdle < 25000) {
            NeighborSelection neighbor = ToolBox.random(iNeighborhoods);
            Neighbor n = neighbor.select(solution);
            iIdle++;
            if (n == null || iIdle < 25000) continue;
            if (n.value() < 0) { iIdle = 0; return n; }
            else if (n.value() == 0) return n;
        }
        return null;
    }
}
```

Fig. 3 Hill climbing neighborhood selection

perfect solution is found or when a time limit is reached. By default, in each step one variable is selected and re-assigned using the selected variable and value selection heuristics. Conflicting values, if any, are automatically unassigned by the framework. The best found solution is memorized during the search. Additional search strategies can be provided by custom neighborhood selectors. The default solution comparator favors a more complete solution. If two solutions have the same number of variables assigned, the one with lover value of the objective function is preferred.

The competition neighborhood selection combines the phases described in the previous chapter. Each phase is also implemented as a neighborhood selection, returning *null* when there is no neighborhood to select and control is then passed to the following phase. Figure 3 outlines the hill climbing phase. Each such phase employs the custom neighborhoods for each particular problem which are also implemented using the same *NeighborSelection* interface.

Figures 4 and 5 outline how the model of the Track 2: Post Enrollment based Course Timetabling problem is implemented in the above discussed framework. Each event is modeled as a variable with a domain consisting of all possible placements of the event into time and space. Only placements using slots that are available to the event and rooms that have the required features and seating capacity are present in the variable's domain. Other requirements are modeled by the student, precedence, and room constraints. A student constraint is created for each student and this is posted on all events to be attended by the student. It detects conflicts (events attended by the student of the constraint taking place at the same time) by maintaining a table of all currently assigned events of the student, and disallows two such events at the same time. When a variable (i.e., an event) is about to be assigned with a value (i.e., a placement consisting of a slot and a room), all constraints associated with the variable are invoked using the *computeConflict* method and their consistency is checked. All conflicting assignments are automatically unassigned prior to the new assignment. Also, the notification methods *assigned* or *unassigned* are called for all constraints associated with a variable whenever it is assigned or unassigned, respectively. This allows any constraint to keep track of current assignments, which is of particularly use in allowing constraints to find conflicting assignments very quickly. Similarly, a precedence constraint

```
public class TimetablingModel extends Model {
    public Vector<Variable> variables(); // set of events
    public Vector<Constraint> constraints(); // rooms, students, precedences
    public int getTotalValue() { // total score
        int score = 0; // sum of Student.score() over all students
        for (Student constr: constraints())
            if (constr instanceof Student) score + = ((Student)constr).score();
        return score;
    }
}
public class Event extends Variable {
    public Set<Student> students(); // set of students attending this event
    public Set<Room> rooms(); // set of available rooms
    public boolean isAvailable(int slot); // event slot availability
    public Set<Placement> values() { // domain
        Set values<Placement> = new Set();
        for (int slot = 0; slot < 45; slot++) {
            if (!isAvailable(slot)) continue;
            for (Room room : rooms())
                values.addElement(new Placement(this, slot, room));
        }
        return values;
    }
}
public class Placement extends Value {
    public Event variable(); // event of this placement
    public int slot(); // time assignment
    public Room room(); // room assignment
    public int toInt(); //change in score if this assigned
}
```

Fig. 4 Post enrollment based course timetabling model

is created for each precedence requirement between two events, and a room constraint is created for each room on all events that can be placed there disallowing two events from overlapping in the room. As for the objective criteria, all three penalties are computed for each student (*score* method) and aggregated over all students in the model class describing the problem. A useful method *toInt* can be computed on each value estimating the impact of the value being assigned to its variable. This method is used by the custom neighborhoods to quickly evaluate their impact.

Besides the model, a custom set of neighborhoods was implemented for each competition problem. A sample neighborhood is shown in Fig. 6. In each step it tries to move a selected event into a different room. Both an event and a new room are selected randomly. If the new room is not available it tries to select the next available room if such exists.

```
public class Student extends Constraint {
    private Placement[] iTable = new Placement[45]; // current assignment
    public void assigned(long iteration, Placement value) {
        iTable[value.slot()] = value;
    }
    public void unassigned(long iteration, Placement value) {
        iTable[value.slot()] = null;
    }
    public void computeConflicts(Placement value, Set conflicts) {
        // compute conflicting assignments for this student
        if (iTable[value.slot()] != null) conflicts.add(iTable[value.slot()]);
    }
    public int score() { // overall penalty for this student
        int score = 0;
        for (int day = 0; day < 5; day++) { int inRow = 0, eventsADay = 0;
            for (int time = 0; time < 9; time++) {
                if (iTable[9 * day + time] != null) {
                    inRow++; eventsADay++;
                    if (time == 8) score++; // last time slot of a day
                } else inRow = 0;
                if (inRow > 2) score++; // more than two in a row
            }
            if (eventsADay == 1) score++; // one class on a day
        }
        return score;
    }
}
public class Precedence extends BinaryConstraint {
    public void computeConflicts(Placement value, Set conflicts) {
        if (isFirst(value.variable())) {
            if (value.time() >= second().time()) conflicts.add(second);
        } else {
            if (first().time() >= value.time()) conflicts.add(first);
        }
    }
}
public class Room extends Constraint {
    private Placement[] iTable = new Placement[45];
    public void assigned(long iteration, Placement value) {
        if (this.equals(value.room())) iTable[value.time()] = value;
    }
    public void unassigned(long iteration, Placement value) {
        if (this.equals(value.room())) iTable[value.time()] = null;
    }
    public void computeConflicts(Placement value, Set conflicts) {
        if (this.equals(value.room()) && iTable[value.time()] != null)
            conflicts.add(iTable[value.time()]);
    }
}
```

Fig. 5 Student, precedence and room constraints

```
public class RoomMove implements NeighborSelection {
    public Neighbor select(Solution solution) {
        // select an event at random
        Event event = ToolBox.random(solution.model().variables());
        // keep time slot
        int slot = event.assignment().slot();
        // select a room at random (from the rooms where the event can take place)
        int rx = ToolBox.random(event.rooms().size());
        // iterate rooms starting from rx, look for the first available one
        for (int r = 0; r < event.rooms().size(); r++) {
            Room room = event.rooms().get((r + rx) % event.rooms().size());
            // skip currently assigned room
            if (room.equals(event.assignment().room())) continue;
            Placement placement = new Placement(event, slot, room);
            if (solution.model().computeConflicts(placement).isEmpty())
                return new SimpleNeighbor(event,placement); //reassignment of event
        }
        return null; // no room available
    }
}
```

Fig. 6 Select a different room for an event

5 Competition tracks

Value settings for the algorithm parameters used in each of the three competition tracks are listed in Table 1. Simulated annealing is not used on the examination timetabling problem for reasons discussed below. As is mentioned above, in the hill climbing, great deluge, and simulated annealing phases, an assignment change (neighbor assignment) is randomly generated from a set of problem specific neighborhoods. Only moves that do not violate any hard constraints are generated. The first step is selection of a neighborhood. This neighborhood is then used to generate a change that is assigned if accepted by the currently used search strategy (HC, GD, or SA). The individual neighborhoods used in each of the tracks are described below.

The parameter values were selected manually based on series of test runs over the initial instances for each phase (usually containing 1–3 runs on each instance). The key was to find a combination of parameters that would deliver constantly good results as fast as possible.

Table 1 Solver parameters for each competition track

Parameter	Track 1	Track 2	Track 3
HC_{idle}	25000	50000	50000
GD_{ub}	1.12	1.10	1.15
GD_{lb}	0.90	0.90	0.90
GD_{cr}	$1-\frac{1}{9\times10^6}$	$1-\frac{1}{5\times10^6}$	$1-\frac{1}{7\times10^6}$
SA_{it}		1.5	2.5
SA_{cr}		0.97	0.82
SA_{cc}		5	7
SA_{rc}		7	7

The cooling rates in Great Deluge GD_{cr} and Simulated Annealing SA_{cr} play the most important rule in improving the solution in each phase. Too rapidly cooling means overshooting the sweet spot where the solver keeps on improving the best found solution, whereas too slowly cooling means that the solver will take too long to get anywhere. The number of idle iterations in Hill Climbing HC_{idle}, lower bound in Great Deluge GD_{lb}, and cooling coefficient in Simulated Annealing SA_{cc} are important for deciding when to move on to the next phase. The solver should no longer be improving the existing solution, but should not remain stuck for too long. The remaining parameters are important for being able to overcome a local optima the solver might become stuck on during the previous phase, however, it is undesirable to divert from the current solution too much.

The individual neighborhoods for each competition problem were created with the following guidelines in mind:

– *Speed* The neighborhood needs to be generated very quickly. This includes an evaluation of its impact on the current solution. Some longer to compute neighborhoods are used as well, but these are triggered with lower probability to compensate for their speed.
– *Success* It is important for a proposed neighborhood to be accepted fairly often. For instance, rather than trying to move an event to a different randomly selected period or room even when that period or room is not available (and less likely to be accepted), a change neighborhood can try the first available period or room after the randomly selected one.
– *Completeness* It is important to be able to move anywhere in the search space. This is achieved by having a moving neighbor for both time and room and a swapping neighbor between pairs of variables.
– *Optimization criteria* Special neighborhoods were created to cope with some of the optimization criteria, especially for those where multiple traditional move and/or swap neighborhoods are needed to fix one problem in the criteria. Precedence constraints in examination and post enrollment course timetabling, or room stability in curriculum based course timetabling are good examples.

5.1 Track 1: examination timetabling

The following neighborhoods are selected with equal probability during examination timetabling. All of the proposed neighborhoods attempt to change an assignment of an exam or to swap the periods and/or rooms of two exams. If a change cannot be made, it systematically searches for an alternate change by selecting the next feasible assignment that follows the initially proposed change (i.e., it tries to use one of the subsequent periods or rooms in the variable's domain in the order they are loaded from the input file) rather than randomly generating and checking some other change.

Exam swap An examination is randomly selected. A new period and room are randomly selected. If there is no (hard) conflict as a result of assigning the selected exam into the new period and room, the new assignment is returned. If there is only one conflicting examination, and it is possible to swap the selected examination with it, such a swap is returned. The following rooms and periods are tried otherwise (all rooms are first considered for the selected period, then for the next period, etc.). Only periods and rooms that are valid for the selected exam are considered.

Period change An examination and a new period are randomly selected. If no conflict results from assigning the selected exam to the new period (keeping its room assignment), the new assignment is returned. The following periods are tried otherwise. The first available period is returned, another neighborhood is tried if no such period can be found.

Room change An examination and a new room are randomly selected. If no conflict results from assigning the selected exam into the new room (keeping its period assignment), the new assignment is returned. The following rooms are tried otherwise.

Period swap An examination and a new period are randomly selected. If just one conflict results from assigning the selected exam to the new period (keeping its room assignment), and it is possible to swap these two exams (each keeping its room assignment and taking the period assignment of the other exam), such a swap is returned. The following periods are tried otherwise.

Room swap An examination and a new room are randomly selected. If just one conflict results from assigning the selected exam into the new room (keeping its period assignment), and it is possible to swap these two exams (each keeping its period assignment and taking the room assignment of the other exam), such a swap is returned. The following rooms are tried otherwise.

Period and room change An examination and a new period are randomly selected. If no conflict results from assigning the selected exam to the new period (into a randomly selected available room), the new assignment is returned. The following periods are tried otherwise.

The examination timetabling solver does not use the simulated annealing phase. Generation of the above described neighbor changes takes much more time than in other tracks due to the complexity of the imposed soft constraints. When combined with the imposed time limit for each instance, various test runs indicated that the time is better spent using only the great deluge approach.

5.2 Track 2: post enrollment based course timetabling

In order to be able to find a complete solution quickly, soft constraints are ignored during the construction phase. Also, unlike in other tracks, it is allowable to assign an event to a time slot without assigning it into a room (e.g., in the case where there is no room available at the proposed time), or to violate a precedence constraint. Both the case of no room assignment and the violation of precedence constraints are treated as soft constraints. The algorithm starts with the weight of these soft constraints being one (overall solution value is the given score plus the weighted sum of violated no-room and precedence constraints), and these weights are increased by one after every 1,000 iterations during which there is no improvement in the number of violations of these constraints. This helps the solver to gradually decrease the number of these violations while it looks for the best solution score.

The following neighborhoods are used during post enrollment based course timetabling (all are selected with the same probability, except of *Precedence Swap* which is selected with $\frac{1}{10}$ the likelihood of the others).

Time move An event and a new time slot are selected randomly. If it is possible to reassign the event into the new time slot while keeping its room without any conflict, such an assignment is returned. The following time slots are tried otherwise, with the first available time slot being returned if any are available.

Room move An event and a new room are selected randomly. If it is possible to reassign the event into the new room while keeping its current time assignment without any conflict, such an assignment is returned. The following rooms are tried otherwise, with the first available room being returned if any are available. Only rooms that are valid for the selected event are considered (i.e., rooms that are of sufficient size and that have all the required features).

Event move An event is randomly selected. A new time slot and a room are randomly selected. If there is no conflict in assigning the selected event into the new time and room, such an assignment is returned. If there is exactly one event conflicting with the new assignment and it is possible to swap these events, this swap is returned. Otherwise, it tries to use one of the following time slots and rooms (first it keeps the selected time slot and picks another room, then the same with the following time slot, etc.).

Event swap Two events are randomly selected. If it is possible to swap these two events, such a swap is returned. Otherwise it tries to swap the times but pick a different room for these events (in a similar way as *Room Move*).

Precedence swap This neighborhood tries to decrease the number of violated precedence constraints by reassigning an event into a different time and room. A violated precedence constraint is selected randomly, one of its events (randomly selected) is placed in a different time and room that does not violate the selected constraint. The time and room are picked in the same way as in *Event Swap* neighborhood. If no time and room can be found for the selected event, an attempt is made to move the other event as well.

5.3 Track 3: curriculum based course timetabling

The following neighborhoods are used during the curriculum based course timetabling. All are selected with the same probability, except for *Curriculum Compactness Move* which is selected with $\frac{1}{10}$ the likelihood of the others.

Time move A period is changed for a randomly selected lecture. The first non-conflicting period after a randomly selected one is used.

Room move A room is changed for a randomly selected lecture. The first non-conflicting room after a randomly selected one is returned.

Lecture move A lecture is selected randomly, a new time and room are selected for the lecture. If no conflict results from assigning the selected lecture into the selected time and room, the assignment is returned. If there is another lecture conflicting with the time and room and it is possible to swap these two lectures, this swap is returned. The following times and rooms are tried otherwise. Only times that are available for the course of the selected lecture are considered.

Room stability move This neighborhood tries to find a change that decreases the room stability penalty. A course and a room are selected randomly. An attempt is made to assign all lectures of the course into the selected room. If there is already another lecture in the room, it is reassigned to the room of the lecture being moved.

Min working days move This neighborhood tries to find a change that decreases minimum working days penalty. A course with a positive penalty is selected randomly, a day on which two or more lectures are taught is selected, and one of lectures of that day is moved to a day that the course is not being taught on.

Curriculum compactness move This neighborhood tries to find a change that decreases the curriculum compactness penalty. A curriculum is selected randomly, a lecture that is not adjacent to any other in the curriculum is selected and placed into another available period that has an adjacent lecture in the curriculum (if such placement exists and does not create any conflict). A different room may also be assigned to the lecture if the current one is not available.

6 Results

The three tables below contain two distinct sets of results for each competition track. The upper portion of each table presents the best solutions found by the author for the early and late problem instances within the given time limit. One hundred runs were made for each instance. All of these results have zero *Distance to Feasibility* (i.e., a complete feasible solution was found), except for one instance from Track 2 (Post Enrollment based Course Timetabling), see Table 3 for more details. These results were submitted to the competition and were successfully reproduced by the competition organizers. The solver described in this paper was named as a finalist in all three tracks.

Table 2 Submitted results for Track 1 (top), best recorded scores of finalists (bottom)

Instance number	1	2	3	4	5	6	7	8
Two exams in a row	42	0	1275	7533	40	3700	0	0
Two exams in a day	0	10	2070	3245	0	0	0	0
Period spread	2534	0	5193	3958	1361	19900	3628	6718
Mixed durations	100	0	0	0	0	75	0	0
Larger exams constraints	260	380	840	105	1440	375	460	380
Room penalty	1150	0	0	0	0	1250	0	125
Period penalty	270	0	190	1750	100	475	0	342
Overall value	4356	390	9568	16591	2941	25775	4088	7565

Instance number	1	2	3	4	5	6	
T. Müller	**4370**	**400**	**10049**	**18141**	**2988**	**26585**	
C. Gogos	5905	1008	13771	18674	4139	27640	
M. Atsuta et al.	8006	3470	17669	22559	4638	29155	
G. Smet	6670	623	–	–	3847	27815	
N. Pillay	12035	2886	15917	23582	6860	32250	
Instance number	7	8	9	10	11	12	*rank*
T. Müller	**4213**	**7742**	**1030**	16682	**34129**	5535	13.3
C. Gogos	6572	10521	1159	–	43888	–	23.4
M. Atsuta et al.	10473	14317	1737	15085	–	**5264**	28.4
G. Smet	5420	–	1288	**14778**	–	–	28.6
N. Pillay	17666	15592	2055	17724	40535	6310	33.8

Table 3 Submitted results for Track 2 (top), best recorded scores of finalists (bottom)

Instance number	1	2	3	4	5	6	7	8
Distance to feasibility	0	0	0	0	0	0	0	0
More than two in a row	728	1093	73	111	0	8	2	0
One class on a day	23	21	132	283	0	0	3	0
Last time slot of a day	579	1040	0	0	0	5	0	0
Overall value	1330	2154	205	394	0	13	5	0
Instance number	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
Distance to feasibility	0	57	0	0	0	0	0	0
More than two in a row	881	1268	118	169	70	2	0	2
One class on a day	16	33	177	233	1	0	0	4
Last time slot of a day	998	1139	52	51	3	0	0	0
Overall value	1895	2440	347	453	74	2	0	6

Instance number	1	2	3	4	5	6	7	8	9	10	11	12	
H. Cambazard et al.	571	993	**164**	310	**5**	**0**	6	**0**	1560	2163	**178**	146	
M. Atsuta et al.	61	547	382	529	**5**	**0**	**0**	**0**	**0**	**0**	548	869	
M. Chiarandini et al.	1482	1635	288	385	559	851	10	**0**	1947	1741	240	475	
C. Nothegger et al.	**15**	**0**	391	**239**	34	87	**0**	4	**0**	**0**	547	**32**	
T. Müller	1861	–	272	425	8	28	13	6	–	–	263	804	
Instance number	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	***rank***
H. Cambazard et al.	**0**	1	**0**	2	**0**	**0**	1824	**445**	**0**	29	**238**	**21**	13.9
M. Atsuta et al.	**0**	**0**	379	191	1	**0**	–	1215	**0**	**0**	430	720	24.4
M. Chiarandini et al.	675	864	**0**	**1**	5	3	1868	596	602	1364	688	822	28.3
C. Nothegger et al.	166	**0**	**0**	41	68	26	**22**	–	33	**0**	–	30	29.5
T. Müller	285	110	5	132	72	70	–	878	40	889	436	372	31.3

The bottom portion of each table compares the results of all finalists in each competition track.[1] Here, each solver was executed 10 times by the competition organizers on all of the late, early and hidden instances. The finalists are listed here in order of their final placement. A dash indicates that the solver was unable to find a complete feasible solution in any of the 10 trial runs. Note that only the best recorded scores achieved by the finalists are presented in this section, however, all 10 trial runs made by the organizers on each instance were used to produce the final ordering of the finalists. The column named *rank* represents the average rank of the solver across all runs and instances. This average rank was used by the organizers to compare the finalists and to name the winner.

[1] As published at the official competition website http://www.cs.qub.ac.uk/itc2007.

Table 4 Submitted results for Track 3 (top), best recorded scores of finalists (bottom)

Instance number	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Room capacity	4	0	0	0	0	0	0	0	0	0	0	0	0	0
Minimum working days	0	15	10	5	180	15	0	5	35	5	0	255	10	5
Curriculum compactness	0	28	62	30	114	26	14	34	68	4	0	76	56	48
Room stability	1	0	0	0	4	0	0	0	0	0	0	0	0	0
Overall value	5	43	72	35	298	41	14	39	103	9	0	331	66	53

Instance number	1	2	3	4	5	6	7	8	9	10	11
T. Müller	**5**	51	84	37	330	**48**	**20**	41	109	**16**	**0**
Z. Lu et al.	**5**	55	**71**	43	**309**	53	28	49	**105**	21	**0**
M. Atsuta et al.	**5**	**50**	82	**35**	312	69	42	**40**	110	27	**0**
M. Geiger	**5**	111	128	72	410	100	57	77	150	71	**0**
M. Clark et al.	10	111	119	72	426	130	110	83	139	85	3

Instance number	12	13	14	15	16	17	18	19	20	21	*rank*
T. Müller	**333**	**66**	59	84	**34**	**83**	83	**62**	**27**	**103**	12.9
Z. Lu et al.	343	73	**57**	**71**	39	91	69	65	47	106	16.7
M. Atsuta et al.	351	68	59	82	40	102	**68**	75	61	123	17.6
M. Geiger	442	622	90	128	81	124	116	107	88	174	38.2
M. Clark et al.	408	113	84	119	84	152	110	111	144	169	42.2

The solver presented in this paper was named winner of the first track (Examination Timetabling) and the third track (Curriculum based Course Timetabling). It placed 5th in the second track (Post Enrollment based Course Timetabling).

A variety of test runs with different settings and changes to neighborhoods were performed in the process of tuning the solver. The above described algorithm and its parameters represent the best achieved results. However, it is likely that there is still plenty of room for optimization of the solver behavior and parameters.

7 Conclusion

The success of the solver presented here greatly exceeded expectations of what could be achieved applying the same hybrid algorithm to all three tracks of the competition. In doing so, however, it has given some validation to the concept of creating a general solution framework for solving a wide class of problems. This is important to the development of practical solution approaches that do not need to be specifically tailored to the individual problem instance. Since the presented approach is built on a framework and techniques that are currently being used to solve real, large-scale course timetabling problems, the results are also helpful in evaluating the potential effectiveness of the solver for a wider range of applications.

For more information about the presented solver, including source code, please visit http://www.unitime.org/itc2007.

References

Dueck, G. (1993). New optimization heuristics: The great deluge algorithm and the record-to record travel. *Journal of Computational Physics*, *104*, 86–92.

Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, *220*(4598), 671–680.

Müller, T. *Constraint solver library*. GNU Lesser General Public License, SourceForge.net. Available at http://cpsolver.sf.net.

Müller, T. (2005). *Constraint-based timetabling*. PhD thesis, Charles University in Prague, Faculty of Mathematics and Physics.

Müller, T., Barták, R., & Rudová, H. (2004). Conflict-based statistics. In J. Gottlieb, D. Landa Silva, N. Musliu, & E. Soubeiga (Eds.), *EU/ME workshop on design and evaluation of advanced hybrid meta-heuristics*. University of Nottingham.

Müller, T., Barták, R., & Rudová, H. (2005). Minimal perturbation problem in course timetabling. In E. Burke & M. Trick (Eds.), *LNCS: Vol. 3616. Practice and theory of automated timetabling, selected revised papers* (pp. 126–146). Berlin: Springer.

Müller, T., & Murray, K. (2008). Comprehensive approach to student sectioning. In *Proceedings of the 7th international conference on the practice and theory of automated timetabling*.

Murray, K., Müller, T., & Rudová, H. (2007). Modeling and solution of a complex university course timetabling problem. In E. Burke & H. Rudová (Eds.), *LNCS: Vol. 3867. Practice and theory of automated timetabling, selected revised papers* (pp. 189–209). Berlin: Springer.