

QuikFix

A Repair-based Timetable Solver

Michael Clark¹, Martin Henz², and Bruce Love³

¹ Metaparadigm Pte Ltd

² National University of Singapore

³ Overseas Family School

Abstract. QuikFix is a software program for solving timetabling problems. The software adapts repair-based heuristic search known in SAT solving to the timetabling domain. A high-level timetabling-specific model enforces structural constraints and allows for meaningful moves in the search space, such as swaps of the time slots or venues of events. QuikFix uses known techniques to improve the search performance, such as multi-starts, tabu lists, and strategic oscillation. The software is easily extensible through the use of object-oriented programming techniques and has been employed for the timetabling of a Singapore K-12 international school, and as an entry to the ITC 2007 timetabling competition.

1 Background

The Overseas Family School, a K-12 international school in Singapore with over 3000 students, faces a complex timetabling problem every year. After employing expert human timetabling expertise, the school's parent company Master Projects Pte Ltd, commissioned and engaged Metaparadigm Pte Ltd for the development of its own bespoke local-search based timetabling solver.

The success of tabu search [GL97] and other meta-heuristics in the timetabling domain made local search an obvious starting point. Instead of exploring a search tree of possible variable assignments, as done by integer and constraint programming, local search quickly constructs an initial state, and then applies so-called *moves* in order to iteratively change it, eventually resulting in an acceptable solution. Domain-independent local search systems such as LOCALIZER [MH97], EASYLOCAL++ [FV01], and HOTFRAME [GS01], follow an abstract view of a state from which possible moves are generated, forming a neighborhood. This approach follows established heuristic search traditions in Operations Research, and has been described in a minimalistic form as GSAT in the domain of boolean satisfiability.

A variant of this approach has proven to be more efficient than GSAT in boolean satisfiability, namely the Walksat strategy [SKC94]. The idea of Walksat is to generate moves not based on a given state, but to select a violated constraint and focus on those moves that “repair” this constraint. The advantage of this approach is that at each step, a smaller number of neighbours needs to be evaluated. In order to translate this approach to

the timetabling domain, QuikFix requires each constraint to elect a move generation strategy that generates repair moves when the constraint is violated. The QuikFix solver is timetabling-specific, which—compared to generic local search frameworks—allows us to exploit domain knowledge effectively, and increase search performance.

In this paper, we explain the QuikFix solver and describe how the problems in Track 3 of the International Timetabling Competition 2007 (ITC 2007) are modeled and solved using this system. Section 2 reviews the problems of the targeted competition track. Section 3 motivates the design of QuikFix by outlining its design goals. Section 4 provides a high-level description of the structure of the engine, Section 5 focuses on the repair-based approach of generating moves based on violated constraints, Section 6 contains the main loop of the solver, and Section 7 describes a number of heuristics that enable QuikFix to achieve competitive results within short running times. Finally, Sections 8 and 9 describe the application of the solver to ITC 2007, and a Singapore International Secondary School, respectively.

2 Problem

Automated timetabling has a long tradition in Operations Research and Artificial Intelligence [Sch99]. This paper focuses on curriculum-based timetabling, as presented in the Second International Timetabling Competition, ITC-2007 [GMS07]. We quote the general problem description, as given by Gaspero, McCollum and Schaerf [GMS07]:

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

In detail, each day is split into a fixed number of time slots, resulting in day/slot pairs called *periods*. Each course consists of a fixed number of lectures to be scheduled in distinct periods, is attended by given number of students, and taught by a teacher. Each teacher is available in a specified set of periods. Each room has a capacity, expressed in terms of the number of available seats. A curriculum is a group of courses such that any pair of courses in the group have students in common. The solution of the problem is a timetable, which is an assignment of a period and a room to all lectures of each course, such that the following hard constraints are met:

- 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 all scheduled in different periods.
- If the teacher of the course is not available to teach that course at a given period, then no lectures of the course can be scheduled at that period.

The objective of timetabling is to generate timetables that meet all hard constraints and minimize the number of violations of the following soft constraints.

- For each lecture, the number of students must be less than or equal to the capacity of its room.
- The lectures of each course must be spread into the given minimum number of days.
- Lectures belonging to a curriculum should be adjacent to each other (i.e., in consecutive periods).
- All lectures of a course should be given in the same room.

Details on how the quality of solutions is quantified with respect to hard and soft constraint violations are given by Gaspero, McCollum and Schaerf [GMS07].

Timetabling at the Overseas Family School shares the structure of these curriculum-based timetabling problems, but requires several additional constraints, as explained in Section 9. Since the constraints of OFS timetabling problems are a superset of the ITC-2007 Track 3, it was easy to use QuikFix to participate in the competition.

3 Design Goals

Before describing the solver, we are listing in this section the main design goals that underlie the QuikFix solver. These principles provide clear guidelines during the development of the development of the solver.

Timetabling focus. The solver must focus on the timetabling domain, and should not be too general in its scope. A timetabling focus will ensure the ability to exploit timetabling-specific optimizations, and provide a data model that enforces strong structural constraints.

Object orientation. The system design should follow the principles of object-oriented design to simplify the data layout of items such as problems, solutions, moves, etc.

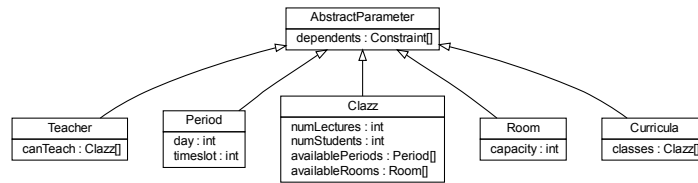
Separation of concerns. As far as possible, the components of the solver should be represented as independent objects with clear interfaces. This separation of concerns will ensure ease of development and maintenance, and provide the basis for extensibility and performance tuning.

Efficiency. Every design decision needs to be analyzed regarding its effect on the runtime of the solver. Real-world and competition requirements demand a rapid response time, and only a carefully engineered system will be able to explore a sufficient search space within the given time constraints.

4 Solver Structure

4.1 Problems

The QuikFix data layout for problems matches closely the format of the ITC 2007 track 3 competition. The principal items are Class, Period (composed of the tuple of day and time slot), Teacher, Room and Curricula.



The main difference lies in the definition of a *Course*. In QuikFix, there is a distinction between a *Course* as a definition of an instructional unit and a *Class* as the scheduled instance of instruction for a particular *Course*. This is to support large courses that need to be split into multiple classes wherein students could equivalently be placed in any one of these classes. The competition problem does not make this distinction so a *Course* can be treated equivalently to a *Class*. For the purposes of this description, the *Class* terminology will be used.

4.2 Solution

A solution is an assignment of each $(Class, lecture\ number)$ tuple to both a $(Teacher, Period)$ and $(Room, Period)$ tuple.

One of the key goals in the design of the QuikFix solver was the use of a high-level structurally constrained timetabling specific model. This structural approach ensures intrinsic satisfaction of basic timetabling constraints:

- no more than one lecture assigned to the same room in the same period
- no more than one lecture assigned to the same teacher in the same period

The solution structure differs from the competition model in that it can also represent a variable assignment of teachers to classes, whereas the teacher allocations are fixed in the competition model.

The solution variable structure follows an object-oriented model containing references to object instances rather than being represented as a matrix of integer or boolean values.

The solution variables (Figure 1) store bi-directional associations between the tuples. The associations are indirected through a tuple pointer rather than referencing the variable directly. This is to reduce the number of variable updates during the layered assignment/move evaluation described later.

The solution variable structure is hidden behind a *TimetableSolution* interface which is used by constraint implementations to evaluate their cost values.

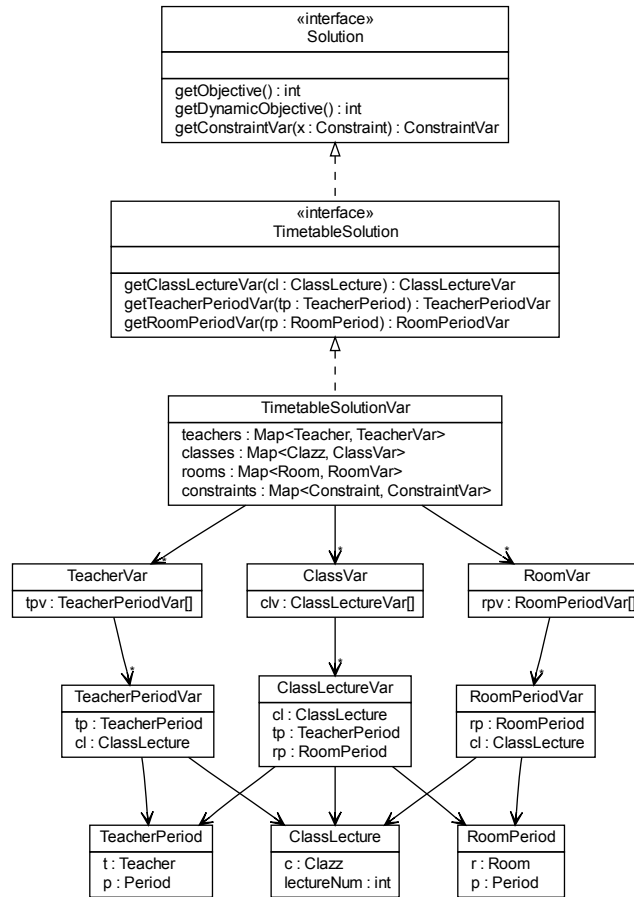


Fig. 1. Solution Structure

5 Constraints and Moves

5.1 Constraints

In QuikFix there is a minimal⁴ distinction between hard and soft constraints. Hard constraints are modeled by setting their weights sufficiently higher. Constraints return an integer value that represents their degree of violation as components of an overall cost function.

An abstract *Constraint* class provides the basis for all Constraint implementations (Figure 2).

The abstract interfaces are as follows:

⁴ The only specific distinction between hard and soft constraints is made in the strategic oscillation heuristic (see Section 7.2).

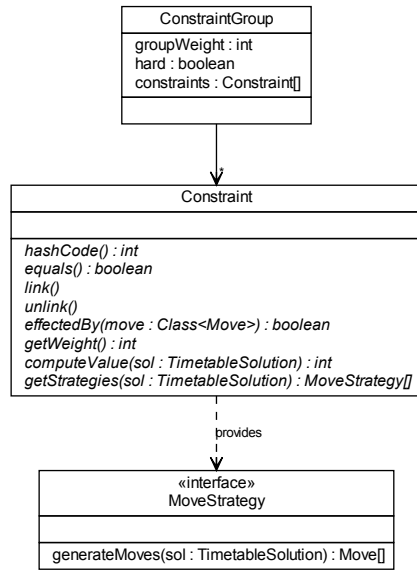


Fig. 2. Constraint interface

`hashCode()` and `equals()` are implemented by constraints so that the intrinsic Java generic collection classes can be used for duplicate constraint elimination.

`link()` and `unlink()` are to allow a `Constraint` implementation to link to its dependent variables to be later used to assess which constraints need to be re-evaluated for a given move/assignment.

`getWeight()` returns the static weight of the constraint.

`computeValue()` computes the constraint value based on the given solution assignment.

`effectedBy()` returns true if the constraint value could be effected by a given type of `Move` (teacher period move or room period move, see Section 5.2).

`getStrategies()` returns a set of move strategies that could repair the given constraint.

Hard Constraints The following implementations of the `Constraint` interface are used to model the hard constraints of the competition problem:

ClassNotConcurrent - given two `Class` parameters, returns the number of periods of the two classes that overlap.

ClassPeriodUnavailable - given a `Class` and `Period`, returns 1 if the class is assigned in the unavailable period.

ClassRoomPeriodInvariant - the solution structure allows the assignment of a $(Class, lecture\ number)$ tuple to a $(Teacher, Period)$ and $(Room, Period)$ tuple with differing periods. This constraint returns the number of lectures where the periods differ.

Soft Constraints The following implementations of the *Constraint* interface are used to model the soft constraints of the competition problem:

CurriculumCompactness: for a given *Curriculum*, returns the number of lectures that are not adjacent to any other lectures out of the set of all classes of the curriculum.

ClassMinimumWorkDays: for a given *Class* and *minimum working days* parameter, if the number of days the class is spread over is less than minimum working days, returns the difference.

ClassRoomCapacity: for a given *Class*, returns the sum over all lectures of the number of students surplus to the size of the assigned room for that lecture.

ClassRoomStability: for a given *Class*, returns the number of rooms the class is assigned to less one.

Constraint Groups Constraints are grouped together into a *ConstraintGroup* to allow setting of a group weight and identifying whether the constraints are hard or soft (Figure 2).

In the final problem formulation, constraint group weights combined with the weighted constraint selection rules and the various move strategies associated with the constraints guides the search through various phases (see Section 5.4).

5.2 Moves and Assignment

In QuikFix, the solution assignments are updated by performing swap moves. Figure 3 illustrates the variable structure and a sample move. These swap moves are the only construct for changing the solution state, which means that all *(Class, lecture number)* tuples are pre-assigned to *(Teacher, Period)* and *(Room, Period)* tuples.

Initial Assignment The initial assignment is done randomly by iterating through all classes in a random order and then for each *(Class, lecture number)* assigning an available *(Teacher, Period)* and *(Room, Period)*. If the required teachers or rooms are exhausted, then virtual teachers and virtual rooms are created to enable complete assignment.

Multiswap Moves A move is composed of a sequence of pair independent transpositions. All moves are thus involutions [BW06]. Each transposition is composed of two tuples for which the assignments are to be swapped. Figure 4 shows the Move interface and implementations⁵. An individual transposition can be performed between two tuples where both of them have an assignment or where only one of the tuples has an assignment (where it effectively acts as a simple move).

⁵ In addition to the pair of tuples to be swapped, the assignments associated with those tuples are also stored. This information is not needed for move evaluation but is used by the tabu mechanism.

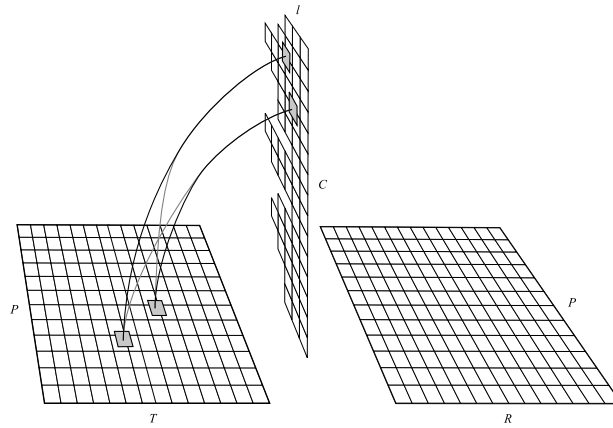


Fig. 3. Example teacher period swap move. The table on the left represents the *teachers* field of *TimetableSolutionVar* in Figure 1, the table on the right represents the *rooms* field, and the vertical table represents the *classes* field.

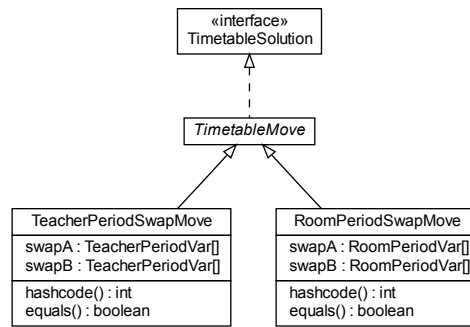


Fig. 4. Move interface

Java generic container classes are used for move comparison and tabu lists. The tuple pairs defining the swap are commutative so *hashCode* must return the same value for an equivalent move where any of the pair representations are transposed and likewise *equals()* must return true. Presently there are two types of moves implemented:

TeacherPeriodSwapMove which defines pairs of $(Teacher, Period)$ tuples for which their associated $(Class, lecture\ number)$ assignments are to be swapped.

RoomPeriodSwapMove which defines pairs of $(Room, Period)$ tuples for which their associated $(Class, lecture\ number)$ assignments are to be swapped.

While at OFS, multiswap moves are required for efficiently and reliably generating timetables, it turns out that single swap moves are most effective for ITC-2007 Track 3 problems.

Move Evaluation Move evaluation takes advantage of object-oriented polymorphism where the *Move* class itself implements the *TimetableSolution* interface (Figure 4). With this mechanism, a move can be evaluated without actually modifying the underlying solution assignments. If the move is chosen, it is applied to the underlying solution by replacing⁶ the modified variables.

To evaluate a move, the following steps are performed:

- Make copies of all variables involved in the move with their associated assignments swapped.
- For each variable changed, add its dependent constraints to a list of constraints that may change for this move.
- Calculate the objective delta by evaluating each of the dependent constraints (using the move itself as the solution instance passed to the constraint’s *compute Value()* method).
- The constraint methods when re-evaluated will call the solution to get the dependent variable’s values.
- The move will respond to the resulting solution interface calls, returning the modified instances for any requests for variables changed by the move or alternatively they will be passed through to the underlying solution.

5.3 Move Strategies

In QuikFix, move strategies are central to the repair-based local search approach whereby a violated constraint is chosen and a move is generated to fix it. A constraint can select a different repair strategy depending on its type. Figure 2 shows the *MoveStrategy* interface returned by the *Constraint* class *getStrategies()* method.

The following move strategies generate *(Teacher, Period)* swap moves:

ClassToNonConcurrentPeriod - given a pair of *Class* as parameters, finds any overlapping periods between the two classes and for each overlapping period generates *n* swap moves to randomly chosen non overlapping periods.

ClassToAvailablePeriod - given a *Class* parameter, finds any periods that are allocated to unavailable periods and generates *n* swap moves to randomly chosen periods that are available.

CompactCurricula - given a *Curriculum* as a parameter, builds a map of all periods of the given curriculum and finds all periods that are not adjacent to any other periods in the curricula. For these non-adjacent periods, *n* swap moves are randomly chosen that move these periods next to other periods in the curriculum.

The following move strategies generate *(Room, Period)* swap moves:

⁶ The variables are replaced so that references to previous assigned states are preserved for moves that are on the tabu list.

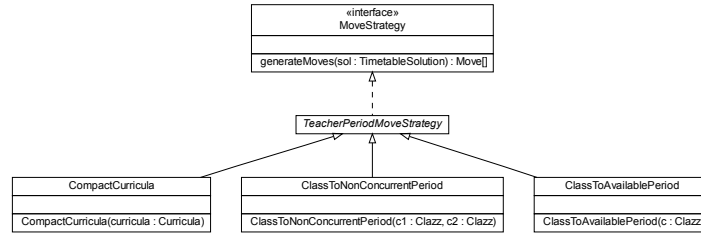


Fig. 5. Teacher period move strategies

ClassToOtherRoom: given a *Class* parameter, for each lecture of the class, generates n swap moves to other rooms within the same period.

ClassToSameRoom: given a *Class* parameter, for each lecture of the class, generates swap moves within the same period to rooms from the set of rooms currently assigned to the lectures of the given class.

This strategy is used to achieve *room stability*.

FixRoomPeriodInvariant: given a *Class* parameter, for each lecture of the class that is assigned to a $(Teacher, Period)$ tuple where the period differs from the assigned $(Room, Period)$ tuple, generates n swap moves to rooms with free periods in the same period as the $(Teacher, Period)$ assignment. This strategy is needed as the solution is not structurally constrained in this dimension. i.e. a teacher can be assigned at a different time to the room assignment.

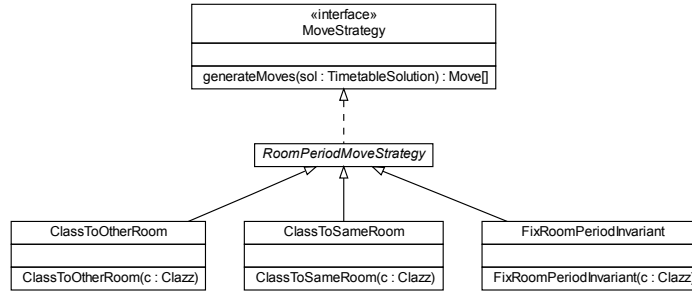


Fig. 6. Room period move strategies

Constraint-specific Strategies In QuikFix, one of the design goals was to be able to easily model new constraints with simple access to solution variables through an abstract high-level timetabling specific solution interface (Figure 1). A second goal in implementing repair-based local search was to allow for a constraint to elect a move generation strategy that may repair itself. This is accomplished through the *Con-*

straint.getStrategies() interface method that is implemented by each constraint class.

We have described the various constraint implementations used by the competition problem followed by a section on the implemented swap move generation strategies. The next step is to show the association between the various constraints and the swap move generation strategies employed for each of them (Table 1).

It is also possible that constraint implementations do not elect a move generation strategy as their objective will be evaluated and guide the search during moves made by other broken constraints. In particular, for the competition problem, no strategy is elected by the *ClassMinimumWorkDays* constraint.

Constraint implementations may also elect multiple strategies. The *ClassRoomCapacity* generates a combinations of room swap moves within the same time period to achieve a better assignment within the same period. This is not always achievable so teacher periods swap ejection moves are generated to move the lecture into another time period as an alternative mechanism to find a better room.

Constraint	Type	Weight	Strategy
ClassNotConcurrent	hard	1000	ClassToNonConcurrentPeriod
ClassPeriodUnavailable	hard	1000	ClassToAvailablePeriod
ClassRoomPeriodInvariant	hard	1000	FixRoomPeriodInvariant
CurriculumCompactness	soft	2	CompactCurricula
ClassMinimumWorkDays	soft	5	<i>none</i>
ClassRoomCapacity	soft	1	ClassToOtherRoom
			ClassToAvailablePeriod
ClassRoomStability	soft	1	ClassToSameRoom

Table 1. Constraint weights and move generation strategies

5.4 Constraint Selection

The key process for repair-based local search involves selecting a broken constraint followed by generating moves to fix it. At each iteration a constraint is selected randomly out of the set of broken constraints with the selection weighted by constraint weightings.

Given a set X containing n broken constraints, with $w(X_j)$ representing the weight of a constraint X_j , the probability p_j of this constraint being selected is:

$$p_j = \frac{w(X_j)}{\sum_{i=1}^n w(X_i)}$$

6 Solver Main Loop

The base logic of the solver main loop is presented here with the following Section 7 detailing the additional heuristics.

The process is to pick a violated constraint, generate moves to repair the constraint, choose a move and conditionally apply it, repeating until some stopping condition is met, similar to the main loop of WSAT(OIP) [Wal98].

There is a single constant *climbProbability* which allows tuning the probability of choosing hill climbing moves as an escape mechanism when no moves exist that improve upon the current value of the objective function.

```
final double climbProbability = 0.10;
```

The Java following code illustrates the main loop algorithm.

```
public void solve(int maxIterations)
{
    TimetableSolution sol = generateInitialSolution();

    for (int iter = 0; i < maxIterations; iter++)
    {
        if (sol.getObjective() <= stoppingObjective)
            return;

        Constraint x = pickViolatedConstraint();
        Move[] moves = generateMovesForConstraint(x);
        Move selectedMove = pickBestMove(moves);

        if (selectedMove == null &&
            Random.nextFloat() < climbProbability)
            selectedMove = pickAnyMove(moves);

        if (selectedMove != null)
            sol.applyMove(selectedMove);

        /* other heuristics here - see Section 7 */
        ...
    }
}
```

7 Heuristics

Without further refinement, the stochastic local search described in the previous two sections typically descends into feasible regions fairly quickly, but then gets trapped in local minima. QuikFix therefore employs the following escape techniques once the feasible region is reached.

7.1 Tabu Lists

QuikFix employs two tabu lists containing moves that are excluded from being chosen as the next step in the main loop of the search.

List of recent moves. For a global constant *recentTabuExpiry*, this list contains the most recent *recentTabuExpiry* moves. This tabu list is essential for escaping local minima.

List of bad moves. For a global constant *badTabuExpiry*, this list contains the most recent *badTabuExpiry* moves that lead to a decrease in the quality of the solution. This tabu list serves as an optimization.

```
final long badTabuExpiry = 100;
final long recentTabuExpiry = 50;
```

In the main loop of the search, any move that is a member of either of the two tabu lists is discarded.

The *TabuList* class makes use of Java Generics to allow tabu of other types (such as *Constraint*).

```
badTabu = new TabuList<Move>(badTabuExpiry);
recentTabu = new TabuList<Move>(recentTabuExpiry);
```

The *pickBestMove()* routine used in the main loop (see Section 6) is described here:

```
Move pickBestMove(Move[] moves)
{
    Move selectedMove = null;
    for (Move move : moves)
    {
        badMoveTabu.runExpiry(iter);
        recentMoveTabu.runExpiry(iter);

        if (badMoveTabu.contains(move) ||
            recentMoveTabu.contains(move)) {
            continue;
        }
        if (move.getObjective() < sol.getObjective()) {
            selectedMove = move;
        }
        else if (move.getObjective() > sol.getObjective()) {
            badTabu.add(move);
        }
    }
    if (selectedMove != null){
        recentTabu.add(selectedMove);
    }
}
```

7.2 Strategic Oscillation

QuikFix adopts strategic oscillation [LM97,AMHV06], a mechanism for escaping from local minima by modifying the weights of constraints, exploring the feasible/infeasible boundary.

After entering the feasible region, any improvement in soft constraints will typically lead to violations of hard constraints, due to the timetabling model adopted by QuikFix. Therefore, our version of strategic oscillation raises the weight of all soft constraints to be equal to that of hard constraints for a fixed number *softBurstOnIters* of moves. Another global parameter *softBurstOffIters* prohibits this behavior for a fixed number of moves after returning to normal weights to allow the solution to settle.

```

final long softBurstOnIters = 15;
final long softBurstOffIters = 30;
final long softBurstWeight = 1000;
long endSoftBurstIter = 0;

```

The following code is added to the main loop.

```

if (sol.isFeasible() &&
     iters > endSoftBurstIter + softBurstOffIters) {
    changeSoftConstraintWeights(sol, softBurstWeight);
    endSoftBurstIters = iters + softBurstOnIters;
}
else if (iters == endSoftBurstIter) {
    changeSoftConstraintWeights(sol, -softBurstWeight);
    endSoftBurstIters = 0;
}

```

7.3 Rapid Restarts

QuikFix starts from a randomly generated assignment and applies repair-based search until it reaches a feasible region. It combines strategic oscillation with multi-starts from the currently best solution [BCFN07], using another global variable *noImproveRestartIters*. For any feasible solution *S*, if the solver is not able to find a feasible solution better than *S* during *noImproveRestartIters* moves after *S* is found, the search abandons that path, and re-starts the search at the best solution found so far.

```

final long noImproveRestartIters = 200;
TimetableSolution savedBestSolution = null;
long lastImproveIter = 0;

```

The following code is added to the main loop.

```

if (iter - lastImproveIter > noImproveRestartIters
     && savedBestSolution != null)
{
    badTabu.clear();
    recentTabu.clear();
    sol = savedBestSolution.clone();
}

if (sol.getObjective() < savedBestSolution.getObjective()) {
    savedBestSolution = sol.clone();
    lastImproveIter = iter;
}

```

8 Benchmarks

The following experiments were conducted on the compute cluster of the School of Computing, National University of Singapore, which comprises more than 100 nodes of 32-bit Intel Xeon processors (2.8GHz) and 64-bit AMD Opteron nodes (2.2 and 2.4GHz).

Each run was performed on one node of the cluster, with a number of iterations that was previously calibrated. Calibration was done by averaging the number of iterations performed during five runs within the allowed competition time. For these five runs, the time was fixed using the calibration software provided by the competition organizers.

Table 2 displays the performance of running QuikFix for the calibrated number of iterations.

9 OFS, A Real-world Application

OFS (Overseas Family School) in Singapore is a school catering for pre-school up to pre-university students. Currently it is managed as four schools: Kindergarten, Elementary School (grades 1 to 5), Middle School (grades 6 to 8), High School (grades 9 to 12). QuikFix is being developed to solve timetabling problems in these schools and to manage resource sharing between these schools. Currently it is being used to develop the High School timetable.

The High School works towards an external qualification of I.G.C.S.E. [IGC07] and M.Y.P. [MYP08] at grade 9 and 10 level (Junior High School) while at grade 11 and 12 level (Senior High School) the external

Instance	Iterations	% Feasible	Best	Median	Worst
comp01	6630000	100	9	16.0	92
comp02	2550000	100	103	139.0	183
comp03	2600000	100	101	137.5	187
comp04	3040000	100	55	84.0	107
comp05	1330000	58	370	510.5	762
comp06	2590000	100	112	146.0	187
comp07	2640000	100	97	142.0	203
comp08	3040000	100	72	95.5	128
comp09	2880000	100	132	150.0	171
comp10	2630000	100	74	107.0	152
comp11	7720000	100	1	5.0	9
comp12	1410000	100	393	452.5	555
comp13	3050000	100	97	124.0	141
comp14	2810000	100	87	109.0	129

Table 2. Average results obtained by QuikFix in one hundred runs, each performed with a number of iterations calibrated to the competition guidelines. The column “Feasible” give the percentage of feasible solutions among all solutions for the respective instance. The last four columns are based on the feasible solutions.

Constraint	Group	Flags	Wt	d(Wt)	VI	r
X Class Group Compactness q002	compact		2	200	2	0.87
X Class Group Compactness q012	compact		2	200	1	1.0
X Class Room Stability c0016	sroom	G	1	100	1	0.9
X Class Room Stability c0002	sroom	G	1	100	1	0.82
X Class Room Capacity c0033 requires 31	nroom	G	1	100	6	1.0
X Class Room Stability c0030	sroom	G	1	100	1	0.78
X Class Room Stability c0066	sroom	G	1	100	2	0.94
X Class Room Stability c0067	sroom	G	1	100	1	0.86
X Class Room Stability c0015	sroom	G	1	100	1	1.0
- Not Concurrent c0032 & c0033	curricula		-	-	-	-
- Not Concurrent c0066 & c0071	curricula		-	-	-	-
Class Spread c0001 over 4 days	spread		5	500	0	0.66
Class Spread c0002 over 4 days	spread		5	500	0	0.5
Class Spread c0004 over 3 days	spread		5	500	0	0.4
Class Spread c0005 over 3 days	spread		5	500	0	0.5

Fig. 7. Constraints interface

qualifications are International Baccalaureate and High School Diploma (accredited by W.A.S.C. [WAS01]).

We briefly highlight constraints and requirements for OFS High School timetabling that go beyond ITC 2007 Track 3. The most important difference between timetabling the High School and the competition problem is that the teachers in the High School are expected to teach about 75% of their available time. Some teachers are given more time depending on their allocated duties. Some courses have several classes (see Section 4.1) and teachers can be used to teach several classes of the same course and the assignment of teachers to classes is done by the solver (see Section 4.2).

In the High School, the students do not have free time. This means that every period of the school week (currently 25) is timetabled with a class/teacher/room. Additional constraints are added to ensure that groups of core courses at Junior High School are placed at the same time to allow students to move freely between them (concurrency constraints). Specialist courses (e.g. science, music, art, computer) must be placed in specialist rooms. Many teachers are allocated rooms and it is important that their teaching is done in that room as much as possible. There are part-time teachers who cannot be allocated rooms and so must use the rooms of full-time teachers when their rooms are not being used.

QuikFix provides extensive support for optimized and interactive timetabling. Constraints can be specified and weighted using constraint interface as shown in Figure 7. Periodic display of the highest-quality timetable uses color-coding of different intensity for constraint violations of corresponding severity, as shown in Figure 8. QuikFix supports interactive timetabling by allowing for fixing teaching event to periods and rooms, and by dynamically changing the constraint weights.

10 Conclusion

The performance of QuikFix on selected (humanly tractable) ITC 2007 competition problems compared with that of an expert human timetable solver (our third author) has shown us that the solver is achieving results

- Curriculum-based course timetabling (track 3). Technical report, DIEGM, University of Udine, August 2007.
- [GS01] L. Di Gaspero and A. Schaerf. Reusable metaheuristic software components and their application via software generators. In *Proceedings of MIC'2001, Meta-heuristics International Conference*, volume 2, pages 637–641, Porto, Portugal, 2001.
- [IGC07] University of Cambridge. International Examinations. Publications catalog available at <http://www.cie.org.uk/profiles/teachers/orderpub>, 2007.
- [LM97] J.-M. Labat and L. Mynard. Oscillation, heuristic ordering and pruning in neighborhood search. In Gert Smolka, editor, *Principles and Practice of Constraint Programming - CP97, Proceedings of the 3rd International Conference*, Lecture Notes in Computer Science 1330, pages 506–518, Linz, Austria, 1997. Springer-Verlag, Berlin.
- [MH97] L. Michel and P. Van Hentenryck. LOCALIZER—a modelling language for local search. In Gert Smolka, editor, *Principles and Practice of Constraint Programming—CP97, Proceedings of the Third International Conference*, Lecture Notes in Computer Science 1330, pages 237–251, Schloss Hagenberg, Linz, Austria, October/November 1997. Springer-Verlag, Berlin.
- [MYP08] International Baccalaureate. Middle Years Programme. Publications available at <http://www.ibo.org/myp/>, 2008.
- [Sch99] A. Schaerf. A survey of automated timetabling. *Artificial Intelligence Review*, 13(2):87–127, 1999.
- [SKC94] B. Selman, H. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proceedings of AAAI-94*, pages 337–343, 1994.
- [Wal98] J. P. Walser. *Domain-Independent Local Search for Linear Integer Optimization*. PhD thesis, Universität des Saarlandes, D 66041 Saarbrücken, Germany, August 1998.
- [WAS01] Western Association of Schools and Colleges. Handbook of Accreditation. Publications catalog available at <http://www.wascsenior.org/wasc/News-Events/NewandRevisedPolicies.htm>, 2001.