A comparison of the performance of different metaheuristics on the timetabling problem

Olivia Rossi-Doria[1], Michael Sampels[2], Mauro Birattari[3], Marco Chiarandini[3], Marco Dorigo[2], Luca M. Gambardella[4], Joshua Knowles[2], Max Manfrin[2], Monaldo Mastrolilli[4], Ben Paechter[1], Luis Paquete[3], Thomas Stützle[3]

[1] School of Computing, Napier University,
10 Colinton Road, Edinburgh, EH10 5DT, Scotland
{o.rossi-doria|b.paechter}@napier.ac.uk

[2] IRIDIA, Université Libre de Bruxelles, CP 194/6,
Av. Franklin D. Roosevelt 50, 1050 Bruxelles, Belgium
{msampels|mdorigo|jknowles|mmanfrin}@ulb.ac.be

[3] Intellektik, Technische Universitaet Darmstadt,
Alexanderstr. 10, 64283 Darmstadt, Germany
{mbiro|machud|lpaquete|tom}@intellektik.informatik.tu-darmstadt.de

[4] IDSIA, Galleria 2, 6928 Manno, Switzerland
{luca|monaldo@idsia.ch}

Abstract. The main goal of this paper is to attempt an unbiased comparison of the peformance of straightforward implementations of five different metaheuristics on a university course timetabling problem. In particular the metaheuristics under consideration are Evolutionary Algorithms, Ant Colony Optimization, Iterated Local Search, Simulated Annealing, and Tabu Search. To attempt fairness the implementations of all the algorithms use a common solution representation, and a common neighbourhood structure or local search. The results show that no metaheuristic is best on all the timetabling instances considered. Moreover, even when instances are very similar, from the point of view of the instance generator, it is not possible to predict the best metaheuristic, even if some trends appear when focusing on particular instance classes. These results underline the difficulty of finding the best metaheuristics even for very restricted classes of timetabling problem.

1 Introduction

This work is part of the Metaheuristic Network*, a European Commission project undertaken jointly by five European institutions, whose aim is to empirically compare and analyse the performance of various metaheuristics on different combinatorial optimization problems including timetabling.

* http://www.metaheuristics.net/

Course timetabling problems arise periodically at every educational institution, as schools and universities. A general problem consists in assigning a set of events (classes, lectures, tutorials, etc) into a limited number of timeslots, so that a set of constraints are satisfied. Constraints are usually classified as hard or soft. Hard constraints are constraints that must not be violated under any circumstances, *e.g.* students cannot attend two classes at the same time. Soft constraints are constraints that should preferably be satisfied, but can be accepted with a penalty associated to their violation, *e.g.* students should not attend three classes in a row. The general course timetabling problem is NP-hard. A considerable amount of research has dealt with the problem, comprehensive reviews can be found in [6, 21].

We consider here a reduction of a typical university timetabling problem. We aim at an unbiased comparison of the performance of straightforward implementations of five different metaheuristics on this problem. In order to attempt a fair and meaningful analysis of the results of the comparison we have restricted all the algorithms to the use of a common direct solution representation and search landscape. Moreover all use the same library, programming language and compiler, and experiments are run on the same hardware.

The stress here is on the comparison of the different methods under similar conditions. More freedom in the use of more efficient representations and more heuristic information may give different results.

The rest of the paper is organized as follows: In Section 2 a description of the particular timetabling problem considered is given and the classes of instances used for the experiments are presented. In Section 3 we describe the common representation and search landscape used for all the implementations of metaheuristics. In Section 4 we give a description of the general features of each metaheuristic under consideration and details of our implementations. Finally in Section 5 we outline the results and conclusions of our study.

2 The University Course Timetabling Problem

The timetabling problem considered here is a reduction of a typical university course timetabling problem. It has been introduced by Ben Paechter to reflect aspects of Napier University's real timetabling problem.

2.1 Problem description

The problem consists of a set of events or classes E to be scheduled in 45 timeslots (5 days of 9 hours each), a set of rooms R in which events can take place, a set of students S who attend the events, and a set of features F satisfied by rooms and required by events. Each student attends a number of events and each room has a size. A feasible timetable is one in which all events have been assigned a timeslot and a room so that the following hard constraints are satisfied:

- no student attends more than one event at the same time;

– the room is big enough for all the attending students and satisfies all the features required by the event;
– only one event is in each room at any timeslot.

In addition, a candidate timetable is penalised equally for each occurrence of the following soft constraint violations:

– a student has a class in the last slot of a day;
– a student has more than two classes in a row;
– a student has a single class on a day.

Note that the soft constraints have been chosen to be representative of three different classes: the first one can be checked with no knowledge of the rest of the timetable; the second one can be checked while building a solution, taking into account the events assigned to nearby timeslots; and finally the last one can be checked only when the timetable is complete, and all events have been assigned a timeslot.

The objective of the problem is to minimize the number of soft constraint violations in a feasible solution. All infeasible solutions are considered worthless.

2.2 Problem Instances

A generator is used to produce problem instances with different characteristics for different values of given parameters. All instances produced have a perfect solution, *i.e.* a solutions with no constraint violations, hard or soft. The generator takes eight command line parameters which specify various characteristics of the instance, and a random seed. Using the same seed will produce the same problem instance - a different seed will produce a different instance with the same characteristics.

Three classes of instances of different size have been selected for comparison purposes, respectively called `small`, `medium` and `large`. They are generated with the sets of parameters reported in Table 1.

Class	`small`	`medium`	`large`
Num_events	100	400	400
Num_rooms	5	10	10
Num_features	5	5	10
Approx_features_per_room	3	3	5
Percent_feature_use	70	80	90
Num_students	80	200	400
Max_events_per_student	20	20	20
Max_students_per_event	20	50	100

Table 1. Parameters used to produce the different instance classes.

Each class of problem has been determined experimentally to be given a specified time limit. The time limits for the problem classes are respectively 90,

900, and 9000 seconds for the `small`, `medium` and `large` class. These timings refer to durations on a specific piece of hardware (see Section 5).

3 Common search landscape

All metaheuristics developed here for the Metaheuristics project employ the same direct solution representation and search landscape, as described in the following (see also [20]). In particular we used a common local search in an evolutionary algorithm, an ant colony optimization algorithm, and an iterated local search. A simulated annealing, and a tabu search were restricted to the same neighbourhood structure.

3.1 The solution representation

We chose a direct solution representation to keep things as simple as possible. A solution consists of an ordered list of length $|E|$ where the positions correspond to the events (position i corresponds to event i for $i = 1, ..., |E|$). An integer number between 1 and 45 (representing a timeslot) in position i indicates the timeslot to which event i is assigned.

The room assignments are not part of the explicit representation; instead we use a matching algorithm to generate them. For every timeslot there is a list of events taking place in it, and a preprocessed list of possible rooms to which these events can be assigned according to size and features. The matching algorithm gives a maximum cardinality matching between these two sets using a deterministic network flow algorithm. If there are still unplaced events left, it takes them in label order and puts each one into the room of correct type and size which is occupied by the fewest events. If two or more rooms are tied, it takes the one with the smallest label. This procedure ensures that each event-timeslot assignment corresponds uniquely to one timetable, *i.e.* a complete assignment of timeslots and rooms to all the events.

3.2 The neighbourhood structure and local search

The solution representation described above allows us to define a neighbourhood using simple moves involving only timeslots and events. The room assignments are taken care of by the matching algorithm.

The neighbourhood is the union of two smaller neighbourhoods, N_1 defined by an operator that moves a single event to a different timeslot, and N_2 defined by a second operator that swaps the timeslots of two events.

The Local Search is a stochastic first improvement local search based on the described neighbourhood. It goes through the list of all the events in a random order, and tries all the possible moves in the neighbourhood for every event involved in constraint violations, until improvement is found. It solves hard constraint violations first, and then, if feasibility is reached, it looks at soft constraint violations as well. Delta evaluation of solutions is extensively used

to allow a faster search through the neighbouring timetables. A more detailed description of the local search is outlined in the following:

1. Ev-count $\leftarrow$ 0;
   Generate a circular randomly-ordered list of the events;
   Initialize a pointer to the left of the first event in the list;
2. Move the pointer to the next event;
   Ev-count $\leftarrow$ Ev-count + 1;
   if (Ev-count = $|E|$) {
   Ev-count $\leftarrow$ 0;
   goto 3.; }
   (a) **if** (current event NOT involved in hard constraint violation (hcv))
   { **goto** 2.; }
   (b) **if** ($\nexists$ an untried move for this event) { **goto** 2.; }
   (c) Calculate next move (first in N_1, then N_2)** and generate resulting potential timetable;
   (d) Apply the matching algorithm to the timeslots affected by the move and delta-evaluate the result;
   (e) **if** (move reduces hcvs) {
   Make the move;
   Ev-count $\leftarrow$ 0;
   goto to 2.;}
   (f) **else goto** 2.(b);
3. **if** ($\exists$ any hcv remaining) END LOCAL SEARCH;
4. Move the pointer to the next event;
   Ev-count $\leftarrow$ Ev-count + 1;
   if (Ev-count = $|E|$) END LOCAL SEARCH;
   (a) **if** (current event NOT involved in soft constraint violation (scv))
   { **goto** 4.; }
   (b) **if** ($\nexists$ an untried move for this event) { **goto** 4.; }
   (c) Calculate next move (first in N_1, then N_2)** and generate resulting potential timetable;
   (d) Apply the matching algorithm to the timeslots affected by the move and delta-evaluate the result;
   (e) **if** (move reduces scvs without introducing a hcv) {
   Make the move;
   Ev-count $\leftarrow$ 0;
   goto 4.; }
   (f) **else goto** 4.(b);

Since the described local search can take a considerable amount of CPU time, it could be more effective within the context of some of the metaheuristics to

** That is, for the event being considered, potential moves are calculated in strict order. First, we try to move the event to the next timeslot, then the next, then the next etc. If this search through N_1 fails then we move through the N_2 neighbourhood, by trying to swap the event with the next one in the list, then the next one, and so on.

use this time in a different way. We therefore introduced in the local search a parameter for the maximum number of steps allowed, which was left free for the different metaheuristic implementations.

4 Metaheuristics

In the following we briefly describe the basic principles of each metaheuristic under consideration and give details of the implementations for the timetabling problem described in Section 2 which are used for the comparison.

4.1 Evolutionary Algorithm

Evolutionary Algorithms (EAs) are based on a computational model of the mechanisms of natural evolution [3]. EAs operate on a population of potential solutions and comprise three major stages: selection, reproduction and replacement. In the selection stage the fittest individuals have a higher chance than those less fit of being chosen as parents for the next generation, as in natural selection. Reproduction is performed by means of recombination and mutation operators applied to the selected parents: recombination combines parts of each of two parents to create a new individual, while mutation makes usually small alterations in a copy of a single individual. Finally, individuals of the original population are replaced by the new created ones, usually trying to keep the best individuals and deleting the worst ones. The exploitation of good solutions is ensured by the selection stage, while the exploration of new zones of the search space is carried out in the reproduction stage, based on the fact that the replacement policy allows the acceptance of new solutions that do not necessarily improve existing ones.

EAs have been successfully used to solve a number of combinatorial optimization problems, including timetabling. State-of-the-art algorithms often use problem-specific information to enhance their performance, such as heuristic mutation [19] or some heuristically guided constructive technique [17].

Here, for the benefit of the comparison and understanding of the role of each component of the algorithm, we propose a basic implementation that uses only the problem-specific heuristic information coming from the local search. It is characterized by a steady-state evolution process, *i.e.* at each generation only one couple of parent individuals is selected for reproduction. A generational genetic algorithm, where the entire population is replaced at each generation, was also implemented, but the steady-state scheme gave better results. Tournament selection is used, that is a number of individuals are chosen randomly from the current population and the best one in terms of fitness function is selected as parent. The fitness function $f(s)$ for a solution s is given by the weighted sum of the number of hard constraint violations hcv and soft constraint violations scv

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

where C is a constant larger than the maximum possible number of soft constraint violations. The crossover used is a uniform crossover on the solution representation, where for each event a timeslot's assignment is inherited either from the first or from the second parent with equal probability. The timeslots assignment corresponds uniquely to a complete timetable after applying the matching algorithm. Mutation is just a random move in the neighbourhood defined by the local search extended with 3-cycle permutations of the timeslots of three distinct events, which corresponds to the complete neighbourhood defined in [20]. The offspring replaces the worst member of the population at each generation. The algorithm is outlined in Algorithm 1.

Algorithm 1 Evolutionary Algorithm

input: A problem instance I
for $i = 1$ **to** n **do**
  {generate a random population of solutions}
  $s_i \leftarrow$ random initial solution
  $s_i \leftarrow$ solution s_i after local search
  sort population by fitness
end for
while time limit not reached **do**
  Select two parents from population by tournament selection
  $s \leftarrow$ child solution after crossover with probability α
  $s \leftarrow$ child solution after mutation with probability β
  $s \leftarrow$ child solution after applying local search
  $s_n \leftarrow$ child solution s replaces worst member of the population
  sort population by fitness
  $s_{best} \leftarrow$ best solution in the population s_1
end while
output: An optimized solution s_{best} for I

The algorithm is a memetic algorithm using the local search described in Section 3. The local search is run with maximum number of steps 200, 1000, and 2000 respectively for the `small`, `medium` and `large` instances. The problem is to find a balance between a reasonable number of steps for the local search and a sufficient number of generations for the evolutionary algorithm to evolve while the local search is not abruptly cut too often and can effectively help to reach local optima.

The initial population is built assigning randomly, for each individual, a timeslot to each event according to a uniform distribution, and applying the matching algorithm. Local search is then applied to each member of the initial population. The population size n is 10, the tournament size is 5, crossover rate is $\alpha = 0.8$ and mutation rate is $\beta = 0.5$.

4.2 Ant Colony Optimization

Ant Colony Optimization (ACO) is a metaheuristic proposed by Dorigo et al. [10]. The inspiration of ACO is the foraging behavior of real ants. The basic

ingredient of ACO is the use of a probabilistic solution construction mechanism based on stigmergy. ACO has been applied successfully to numerous combinatorial optimization problems including the quadratic assignment problem, satisfiability problems, scheduling problems etc. The algorithm presented here is the first implementation of an ACO approach for a timetabling problem. It follows the ACS branch of the ACO metaheuristic, which is described in detail in [4] and which showed good results for the traveling salesman problem [9].

Algorithm 2 Ant Colony System

$\tau(e,t) \leftarrow \tau_0 \; \forall \; (e,t) \in E \times T$
input: A problem instance I
calculate $c(e,e') \; \forall \; (e,e') \in E^2$
calculate $d(e), f(e), s(e) \; \forall \; e \in E$
sort E according to $\prec$, resulting in $e_1 \prec e_2 \prec \cdots \prec e_n$
$j \leftarrow 0$
while time limit not reached **do**
  $j \leftarrow j + 1$
  for $a = 1$ **to** m **do**
    {construction process of ant a}
    $A_0 \leftarrow \emptyset$
    for $i = 1$ **to** n **do**
      choose timeslot t randomly according to probability distribution P for event e_i
      perform local pheromone update for $\tau(e_i, t)$
      $A_i \leftarrow A_{i-1} \cup (e_i, t)$
    end for
    $s \leftarrow$ solution after applying matching algorithm to A_n
    $s \leftarrow$ solution after applying local search for $h(j)$ steps to s
    $s_{best} \leftarrow$ best of s and C_{best}
  end for
  global pheromone update for $\tau(e,t) \; \forall \; (e,t) \in E \times T$ using C_{best}
end while
output: An optimized candidate solution s_{best} for I

The basic principle of an ACS for tackling the timetabling problem is outlined in Algorithm 2: At each iteration of the algorithm, each of m ants constructs, event by event, a complete assignment of the events to the timeslots. To make a single assignment of an event to a timeslot, an ant takes the next event from a pre-ordered list, and probabilistically chooses a timeslot for it, guided by two types of information: (1) heuristic information, which is an evaluation of the constraint violations caused by making the assignment, given the assignments already made, and (2) stigmergic information in the form of a 'pheromone' level, which is an estimate of the utility of making the assignment, as judged by previous iterations of the algorithm. The stigmergic information is represented by a matrix of 'pheromone' values $\tau : E \times T \rightarrow \mathbf{R}_{\geq 0}$, where E is the set of events and T is the set of timeslots. These values are initialized to a parameter τ_0, and then updated by local and global rules; generally, an event-timeslot pair which has been part of good solutions in the past will have a high pheromone value, and

consequently it will have a higher chance of being chosen again in the future. At the end of the iterative construction, an event-timeslot assignment is converted into a candidate solution (timetable) using the matching algorithm. This candidate solution is further improved by the local search routine. After all m ants have generated their candidate solution, a global update on the pheromone values is performed using the best solution found since the beginning. The whole construction phase is repeated, until the time limit is reached.

The single parts of Algorithm 2 are now described in more detail. The following data are precalculated for events $e, e' \in E$:

$$\begin{aligned} c(e,e') &:= \#\text{ students attending both } e \text{ and } e', \\ d(e) &:= |\{e' \in E \setminus \{e\} \mid c(e,e') \neq 0\}|, \\ f(e) &:= \#\text{ features required by } e, \\ a(e) &:= \#\text{ students attending } e. \end{aligned}$$

We define a total order[‡] $\prec$ on the events by

$$\begin{aligned} e \prec e' \Leftrightarrow\ & d(e) > d(e') \vee \\ & d(e) = d(e') \wedge f(e) < f(e') \vee \\ & d(e) = d(e') \wedge f(e) = f(e') \wedge a(e) > a(e') \vee \\ & d(e) = d(e') \wedge f(e) = f(e') \wedge a(e) = a(e') \wedge l(e) < l(e') \ . \end{aligned}$$

Here, $l : E \to \mathbf{N}$ is an injective function that is only used to handle ties. We define $E_i := \{e_1, \ldots, e_i\}$ for the totally ordered events denoted as $e_1 \prec e_2 \prec \ldots \prec e_n$.

For the construction of an event-timeslot assignment each ant assigns sequentially timeslots to the events, which are processed according to the order $\prec$. This means, it constructs assignments $A_i : E_i \to T$ for $i = 0, \ldots, n$.

We start with the empty assignment $A_0 = \emptyset$. After A_{i-1} has been constructed, the assignment A_i is constructed as $A_i = A_{i-1} \cup \{(e_i, t)\}$ where t is chosen randomly out of T with the following probabilities:

$$P(t = t' \mid A_{i-1}, \tau) = \frac{\tau(e_i,t')^\alpha \cdot \eta(e_i,t')^\beta \cdot \pi(e_i,t')^\gamma}{\sum_{u \in T} \tau(e_i,u) \cdot \eta(e_i,u)^\beta \cdot \pi(e_i,u)^\gamma} \ .$$

The parameters β and γ control the weight of the heuristic information corresponding to hard and soft constraint violations, respectively. These heuristic functions η and π are defined as follows.

$$\eta(e_i, t') := \frac{1}{1 + \sum_{e \in A_{i-1}^{-1}(t')} c(e_i, e)}$$

is used to give higher weight to those timeslots that produce fewer student clashes. In order to give higher weight to those timeslots that produce fewer

[‡] We are aware of the fact that it could make more sense to order the events with respect to the features the other way around, but actually in this case it doesn't make significant difference in terms of results.

soft constraint violations, we use:

$$\pi(e_i, t') := \frac{1}{1 + L + S + R_{before} + R_{around} + R_{after}} \text{ , with}$$

$$\begin{aligned}
L &:= \begin{cases} a(e_i) \text{ if } t' \text{ is the last timeslot of the day} \\ 0 \quad \text{otherwise,} \end{cases} \\
S &:= \#\text{ students attending event } e_i\text{, but no other events belonging} \\
&\quad \text{to the same day as } t' \text{ in } A_{i-1}, \\
R_{before} &:= \#\text{ students attending event } e_i \text{ and also events in the two} \\
&\quad \text{timeslots before } t' \text{ on the same day,} \\
R_{around} &:= \#\text{ students attending event } e_i \text{ and also events in both timeslots} \\
&\quad \text{before and after } t' \text{ on the same day,} \\
R_{after} &:= \#\text{ students attending event } e_i \text{ and also events in the two} \\
&\quad \text{timeslots after } t' \text{ on the same day.}
\end{aligned}$$

After each construction step, a local update rule on the pheromone matrix is applied for the entry corresponding to the current event e_i and the chosen timeslot t_{chosen}:

$$\tau(e_i, t_{chosen}) \leftarrow (1 - \psi) \cdot \tau(e_i, t_{chosen}) + \psi \cdot \tau_0 \ .$$

The parameter $\psi \in [0, 1]$ is the pheromone decay parameter, which controls the diversification of the construction process. The higher its value the smaller the probability to choose the same event-timeslot pair in forthcoming steps.

After the assignment A_n has been completed, the matching algorithm for the assignment of rooms is executed, in order to generate a candidate solution s. The local search routine is applied to s for a number of steps $h(j)$ depending on the current iteration number $j \in \mathbf{N}$.

The global update rule for the pheromone matrix τ is performed after each iteration as follows: Let A_{best} be the assignment of the best candidate solution s_{best} found since the beginning. For each event-timeslot pair (e, t) we update:

$$\tau(e, t) \leftarrow \begin{cases} (1 - \rho) \cdot \tau(e, t) + \rho \cdot \frac{Q}{1 + q(s_{best})} & \text{if } A_{best}(e) = t \\ (1 - \rho) \cdot \tau(e, t) & \text{otherwise,} \end{cases}$$

where Q is a parameter controlling the amount of pheromone laid down by the update rule, and the function q measures the quality of a solution s as the sum of hard constraint violations hcv and soft constraint violations scv:

$$q(s) := \#\ hcv(s) + \#\ scv(s) \ .$$

The parameters for the algorithm described above were chosen after several experiments on the given test problem instances and are reported in Table 2.

	small	medium	large
m	15	15	10
τ_0	0.5	10	10
ρ	0.1	0.1	0.1
α	1	1	1
β	3	3	3
γ	2	2	2
ψ	0.1	0.1	0.1
$h(j)$	$\begin{cases} 5\,000 & j = 1 \\ 2\,000 & j \geq 2 \end{cases}$	$\begin{cases} 50\,000 & j \leq 10 \\ 10\,000 & j \geq 11 \end{cases}$	$\begin{cases} 150\,000 & j \leq 20 \\ 100\,000 & j \geq 21 \end{cases}$
Q	10^5	10^{10}	10^{10}

Table 2. Parameters for the ACO algorithm.

4.3 Iterated Local Search

ILS [15] is based on the simple yet powerful idea of improving a local search procedure by providing new starting solutions obtained from perturbations of a current solution, often leading to far better results than when using random restart [12, 15, 16, 18, 22]. To apply ILS, four components have to be specified. These are a GenerateInitialSolution procedure that generates an initial solution s_0, a Perturbation procedure, that modifies the current solution s leading to some intermediate solution s', a LocalSearch procedure that returns an improved solution s'', and a procedure AcceptanceCriterion that decides to which solution the next perturbation is applied. A scheme for ILS is given below.

Algorithm 3 Iterated Local Search

```
s_0 = GenerateInitialSolution()
s = LocalSearch(s_0)
while termination condition not met do
  s' = Perturbation(s, history)
  s'' = LocalSearch(s')
  s = AcceptanceCriterion(s, s'', history)
end while
```

In our implementation for the university course timetabling problem the LocalSearch procedure was the common local search described in Section 3. GenerateInitialSolution generates initial random solutions according to a uniform distribution, so that no problem-specific information is used. We implemented the following three types of perturbation moves:

P1. choose a different timeslot for a randomly chosen event;
P2. swaps the timeslots of two randomly chosen events;
P3. choose randomly between the two previous types of moves and a 3-exchange move of timeslots of three randomly chosen events.

All random choices were taken according to a uniform distribution. Each of these different moves is applied k times, where k is chosen of the set $\{1, 5, 10, 25, 50, 100\}$.

Hence, it determines the strenght of the perturbation. The Perturbation is applied to the solution returned by the AcceptanceCriterion.

We considered three different methods for accepting solutions in AcceptanceCriterion:

Random Walk: This method always accepts the new solution s'' returned by LocalSearch.

Accept if Better: The new solution s'' is accepted if it is better than s. This leads to a first improvement descent in the space of the local optima.

Simulated Annealing: The new solution s'' is always accepted if it is better than the current one. Otherwise s'' is accepted with a probability based on the evaluation function $f(s)$, but infeasible new solutions are never accepted when the current one is feasible. $f(s)$ is the number of hard constraint violations if both s and s'' are infeasible, or the number of soft constraint violations if they are both feasible. Two methods for calculating this probability were applied:

SA1. $P_1(s, s'') = e^{-\frac{(f(s)-f(s''))}{T}}$

SA2. $P_2(s, s'') = e^{-\frac{(f(s)-f(s''))}{T \cdot f(s_{best})}}$

where T is a parameter called *temperature* and s_{best} is the best solution found so far. The value of T is kept fixed during the run, and it is chosen from $\{0.01, 0.1, 1\}$ for **SA1** and $\{0.05, 0.025, 0.01\}$ for **SA2**.

Finally we even considered applying ILS without LocalSearch. In this implementation, the Perturbation switched between moves **P1** and **P2**. The values of k tested were chosen from the set $\{25, 50, 100, 200\}$. This implementation is also known as Reduced Variable Neighbourhood Search.

We ran all combinations of parameters for the `small` and `medium` instances in an automated parameter tuning procedure, the racing algorithm proposed by Birattari et al. [2]. This method, based on the Friedman statistical test, empirically evaluates a set of candidate configurations discarding bad ones as soon as statistically sufficient evidence is gathered against them. The instances used in the race were generated with the problem instance generator described in Section 2.2. The best resulting configurations of parameters for each instance class are summarized as follows.

Small instances
Type of Perturbation: **P1**
$k = 1$
AcceptanceCriterion: **SA2** with $T = 0.025$

Medium instances
Type of perturbation: **P1**
$k = 5$
AcceptanceCriterion: **SA1** with $T = 0.1$

For the `large` instances the automated tuning would have required a very large amount of time given the actual computational environment available.

Consequently the same parameter setting found for the `medium` instances is used also for the `large` ones.

4.4 Simulated Annealing

Simulated Annealing is a local search inspired by the process of annealing in physics [7, 14]. It is widely used to solve combinatorial optimization problems, especially to avoid getting trapped in local optima when using simpler local search methods [1]. This is done as follows: an improving move is always accepted while a worsening one is accepted according to a probability which depends on the amount of deterioration in the evaluation function value, such that the worse a move is, the less likely it is to accept it. Formally a move is accepted according to the following probability distribution, dependent on a virtual temperature T, known as the Metropolis distribution:

$$p_{accept}(T, s, s') = \begin{cases} 1 & \text{if} \quad f(s') \leq f(s) \\ e^{-\frac{(f(s')-f(s))}{T}} & \text{otherwise} \end{cases}$$

where s is the current solution, s' is the neighbour solution and $f(s)$ is the evaluation function. The temperature parameter T, which controls the acceptance probability, is allowed to vary over the course of the search process.

We tackle the course timetabling problem in two distinct phases. In a first phase only hard constraints are considered and reduced. When a feasible solution is reached, which means no hard constraints are violated anymore, a second phase starts and tries to minimize the number of soft constraints violations. Going back from a feasible to an infeasible solution is not allowed. In the first phase the evaluation function f is given by the number of hard constraints violations, *hcv*, while in the second phase by the number of soft constraints violations, *scv*.

Algorithm 4 outlines the global procedure with the two phases Simultaed Annealing, where T_h is the temperature in the infeasible region, and T_s is the temperature in the feasible region.

We implemented versions of Simulated Annealing that differ in the following components: neighbourhood exploration strategy, initial temperature, cooling schedule and temperature length. The different variants and different parameters for these components have been object of an automated tuning for finding the best configuration, using the same racing algorithm as described in Section 4.3. The considered choices for the four components are the following.

Neighbourhood exploration We considered two strategy for generating a neighboring solution:

1. Strategy 1 considers moves from neighbourhoods N_1 and N_2 in the same order as in the local search outlined in Section 3.2. Only events involved in constraint violations are considered. Yet, different from the local search procedure, after trying all possible moves for each event of the timetable, the algorithm does not end but continues with a different order of events.

Algorithm 4 Simulated Annealing

input: A problem instance I
$s \leftarrow$ random initial solution
{Hard Constraints phase}
$T_h \leftarrow T_{h0}$;
while time limit not reached and $hcv > 0$ **do**
  Update temperature;
  $s' \leftarrow$ Generate a neighbouring solution of s
  if $f(s') < f(s)$ **then**
    $s \leftarrow s'$;
  else
    $s \leftarrow s'$ with probability $p(T, s, s') = e^{-\frac{(f(s')-f(s))}{T}}$
  end if
  $s_{best} \leftarrow$ best between s and s_{best}
end while
{Soft Constraints phase}
$T_s \leftarrow T_{s0}$
while time limit not reached and $scv > 0$ **do**
  Update temperature
  $s' \leftarrow$ Generate a neighbouring solution of s
  if $hcv = 0$ in s' **then**
    if $f(s') < f(s)$ **then**
      $s \leftarrow s'$
    else
      $s \leftarrow s'$ with probability $p(T, s, s') = e^{-\frac{(f(s')-f(s))}{T}}$
    end if
    $s_{best} \leftarrow$ best between s and s_{best}
  end if
end while
output: An optimized solution s_{best} for I

2. Strategy 2 abandons the local search framework and uses a completely random move selection strategy. At each step the proposed move is generated randomly from the union of N_1 and N_2.

Initial temperature Two possibilities were considered:

1. Use the temperature that provides a probability of $1/e$ for accepting a move that worsens by 2% the evaluation function value of a randomly generated solution s_r. Formally, choose T such that

$$p = \frac{1}{e} = e^{-(\frac{0.02 \cdot f(s_r)}{T})}$$

   i.e. $T = 0.02 \cdot f(s_r)$
2. Sample the neighbourhood of a randomly generated initial solution, compute the average value of the variation in the evaluation function produced by the sampled neighbours, and multiply this value by a given factor to obtain the initial temperature. We fix the size of the sample to 100 neighbors.

Because the first method produce initial temperatures which does not scale well with the hardness of the instances, the latter one is preferred. Two different multiplier factors (`TempFactHcv` and `TempFactScv`) has to be considered in the tuning of the algorithm, one for the initial value of T_h and the other one for the initial value of T_s.

Cooling schedule We use a non monotonic temperature schedule realized by the interaction of two strategies: a standard geometric cooling and a temperature re-heating. The standard geometric cooling computes the temperature T_{n+1} in iteration $n+1$ by multiplying the temperature T_n in iteration n with a constant factor α (cooling rate):

$$T_{n+1} = \alpha \times T_n, \qquad 0 < \alpha < 1$$

This schedule is expected to be competitive with the adaptive cooling as proposed for the Graph Partitioning Problem by Johnson et al. [12] and in many successful implementations for timetabling where parameters were obtained by experimentations. A sort of adaptation to the behaviour of the search process, however, is included in our implementation by re-heating the temperature when the search seems to be stagnating. Indeed, according to a rejection ratio given by the number of moves rejected on the number of moves tested, the temperature is increased to a value equal to the initial one when the ratio exceeds a given limit. This inspection is done every fixed number of iterations, in our case three times the temperature length. Cooling rate and rejection ratio are thus other parameters which need tuning; effectively these are four parameters α_h, α_s, `RejLimHcv`, `RejLimScv` w.r.t. the phase undertaken. For the sake of simplicity we fix $\alpha_h = \alpha_s = \alpha$.

Temperature length The number of iterations at each temperature is kept proportional to the size of the neighbourhood, as suggested by Johnson et al. [13], who remarked that this seems to be necessary in order to obtain high quality solutions. The rate of the neighbourhood size (`NeighRate`) is another parameter to optimize. We keep it the same for the two phases.

SA in LS					
NeighRate	α	TempFactHcv	TempFactScv	RejLimHcv	RejLimScv
0.1	0.8	0.1	0.3	0.98	0.97
0.2	0.9	0.2	0.62		
	0.97		0.7		
SA in LS with Fixed Temperature					
NeighRate	α	TempFactHcv	TempFactScv	AccLimHcv	AccLimScv
–	–	0.1	0.05	–	–
		0.2	0.1		
			0.15		
			0.2		
			0.3		
SA random					
NeighRate	α	TempFactHcv	TempFactScv	AccLimHcv	AccLimScv
0.1	0.8	*0.1*	***0.3***	***0.98***	***0.97***
0.2	***0.9***	**0.2**	0.62		
	0.95				

Table 3. Simulated Annealing parameters for the three versions considered. `NeighRate` is the proportion of the Neighbourhood examined at each temperature, α is the cooling rate, `TempFactHcv` and `TempFactScv` are the multiplier factors for the initial temperature, respectively for hard and soft constraints, `AccLimHcv` and `AccLimScv` are the acceptance ratio limits respectively for the hard and the soft constraints loops.

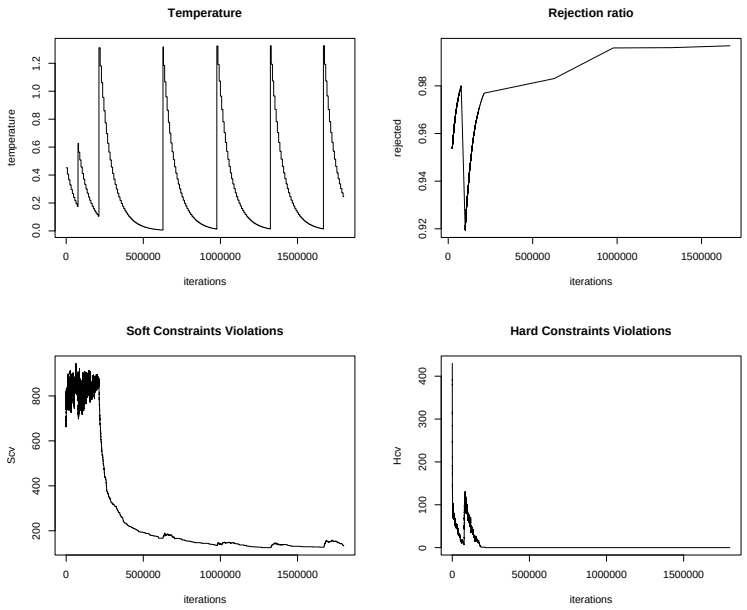

Fig. 1. Behaviour of Simulated Annealing components over the search process in a run of the best configuration found on a medium instance. We remark that we check the rejection rate only every three times the temperature length.

Suggested by experimental observations we tested also a version of simulated annealing in which the temperature is kept constant over the whole search process. The idea is that the simulated annealing acceptance criterion is useful at the end of the search for getting out from local optima, allowing the acceptance of worsening moves. Therefore maintaining a certain probability of accepting worsening moves during the whole search and above all during the end of the process could produce competitive results with less effort for properly tuning the parameters since in this case only the constant temperature needs to be considered. We tested this version using the first neighbourhood exploration strategy.

In all the cases the stopping criterion is the time limit imposed from the experiments set up.

Table 3 summarizes the components and the values of the different parameters that were tuned with the racing algorithm for the simulated annealing. The 70 different configurations tested in the race were generated from all the possible combinations of these values.

The result of the race is that the best implementation, out of the three described, for both the `small` and `medium` instances, is the simulated annealing with complete random move selection. In Table 3 the values for the winning configurations are indicated by italic face for the `small` instances and by bold face font for the `medium` instances. As in the ILS case, the configuration proposed for the `large` instances is the same used for the `medium` instances.

Figure 1 shows the behaviour over the search process for the temperature, the acceptance ratio, soft constraints violations and hard constraints violations of the best configuration found in a run on a `medium` instance. The behaviour on `small` instances is similar.

4.5 Tabu Search

Tabu search (TS) is a local search metaheuristic which relies on specialized memory structures to avoid entrapment in local minima and achieve an effective balance of intensification and diversification. TS has proved remarkably powerful in finding high-quality solutions to computationally difficult combinatorial optimization problems drawn from a wide variety of applications [1, 11]. More precisely, TS allows the search to explore solutions that do not decrease the objective function value, but only in those cases where these solutions are not forbidden. This is usually obtained by keeping track of the last solutions in term of the move used to transform one solution to the next. When a move is performed the reverse move is considered *tabu* for the next l iterations, where l is the tabu list length. A solution is forbidden if it is obtained by applying a tabu move to the current solution.

The implementation of TS proposed for the university course timetabling problem is described in the following. According to the neighbourhood operators described in Section 3.2, a move is defined by moving one event or by swapping two events. We forbid a move if at least one of the events involved has been moved less than l steps before. The tabu status length l is set to the number of events divided by a suitable constant k (we set $k = 100$). With the aim of decreasing

the probability of generating cycles and reducing the size of the neighbourhood for a faster exploration, we consider a variable neighbourhood set: every move is a neighbour with probability 0.1. Moreover, in order to explore the search space in a more efficient way, tabu search is usually augmented with some aspiration criteria. The latter are used to accept a move even if it has been marked tabu. We perform a tabu move if it improves the best known solution.

The TS algorithm is outlined in Algorithm 5, where L denotes the tabu list. In summary, it considers a variable set of neighbours and performs the best move that improves the best known solution, otherwise it performs the best non-tabu move chosen among those belonging to the current variable neighbourhood set.

Algorithm 5 Tabu Search

input: A problem instance I
$s \leftarrow$ random initial solution
$L \leftarrow \emptyset$
while time limit not reached **do**
  for i=0 to 10% of the neighbours **do**
    $s_i \leftarrow s$ after i-th move
    compute fitness $f(s_i)$
  end for
  if $\exists s_j | f(s_j) < f(s)$ and $f(s_j) \leq f(s_i) \forall i$ **then**
    $s \leftarrow s_i$
    $L \leftarrow L \cup E_i$ where E_i is the set of events moved to get solution s_i
  else
    $s \leftarrow$ best non tabu moves between all s_i
    $L \leftarrow L \cup E_b$ where E_b is the set of events moved by the best non-tabu move
    $s_{best} \leftarrow$ best solution so far
  end if
end while
output: An optimized solution s_{best} for I

5 Evaluation

We tested the five algorithms on a PC with an AMD Athlon 1100 Mhz on five `small` instances with running time 90 seconds, in 500 independent trials per metaheuristic per instance; five `medium` instances with running time 900 seconds, in 50 independent trials per metaheuristic per instance; and two `large` instances with running time 9000 seconds, in 20 independent trials per metaheuristic per instance.

The complete results of the experiments, the test instances and all the algorithms can be found at `http://iridia.ulb.ac.be/~msampels/ttmn.data`.

Results for one instance of each class `small` and `medium` are summarized in Figures 2 and 3. The results of all trials on a single instance are ordered by the quality of the solution (number of soft constraint violations) and the rank of the solution in all solutions. An invalid solution (with hard constraint violations) is considered to be worse than any valid solution. Thus it is ordered behind

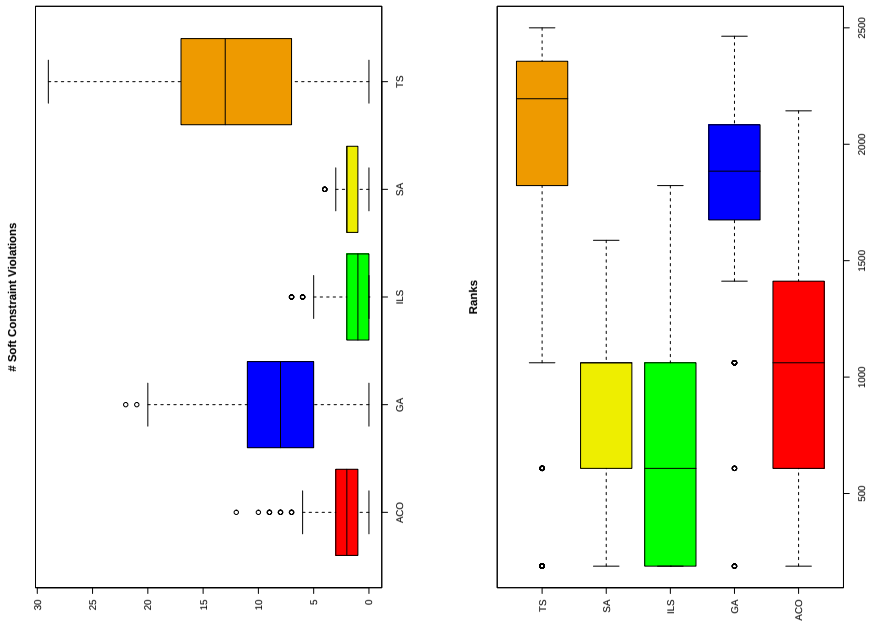

Fig. 2. Results for the `small03` instance.

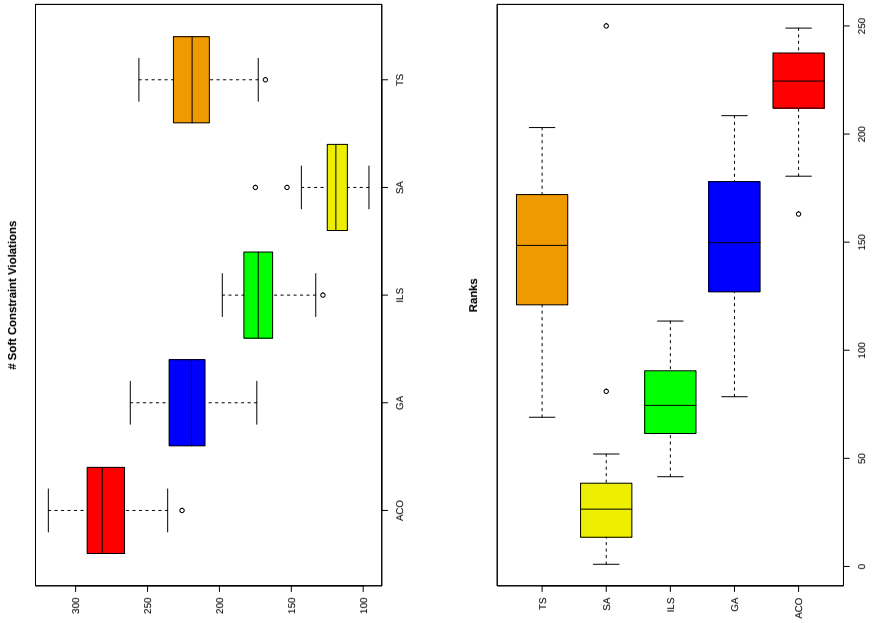

Fig. 3. Results for the `medium01` instance.

them. The solutions are grouped by the metaheuristic used. In the boxplots a box shows the range between the 25% and the 75% quantile of the data. The median is indicated by a bar. The whiskers extend to the most extreme data point which is no more than 1.5 times the interquantile range from the box. Outliers are indicated as circles.

Figures 4 and 5 show the results for the two `large` instances with additional diagrams to report the distribution of the valid solutions and the percentage of invalid solutions that were found in the 20 independent trials of each implemented metaheuristics.

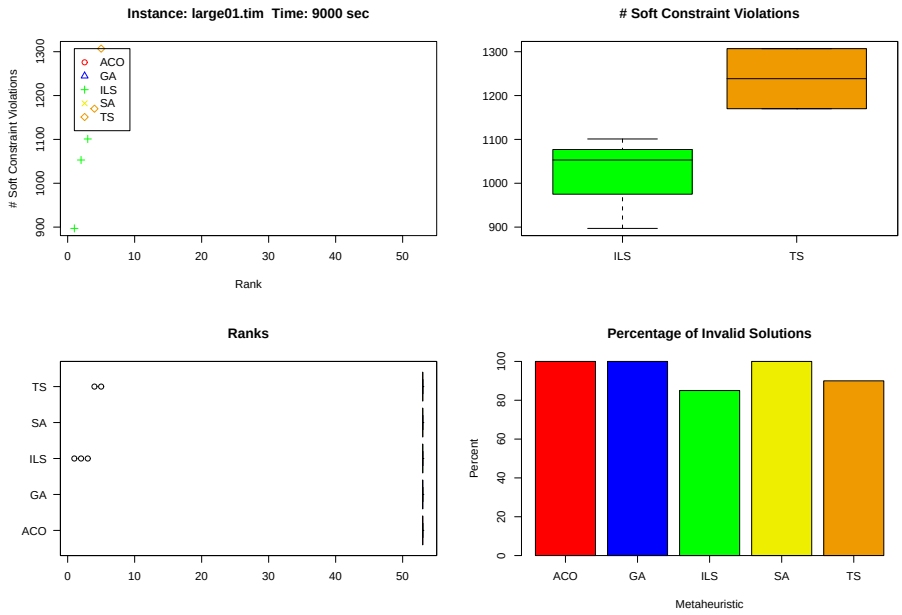

Fig. 4. Results for the `large01` instance.

In the following we give a few highlights and comments on the results on each class of instances.

On the `small` instances all the algorithms reach feasibility in every run. ILS generally performs best, followed closely by SA and ACO. GA is definitely worse, but TS shows the worst overall performance.

SA is best on `medium` instances, even if it does not reach feasibility in some runs. ILS is still very good and more reliable in terms of feasibility. GA and TS give similar worse results, and ACO shows the worst performance.

For the first `large` instance `large01` most metaheuristics do not even find feasibility. TS reaches feasibility for about 8% of the trials, ILS for a bit more, and, when it does, results for soft constraints are definitely better than the TS ones. ILS is again best for the `large02` instance, where it finds feasibility for

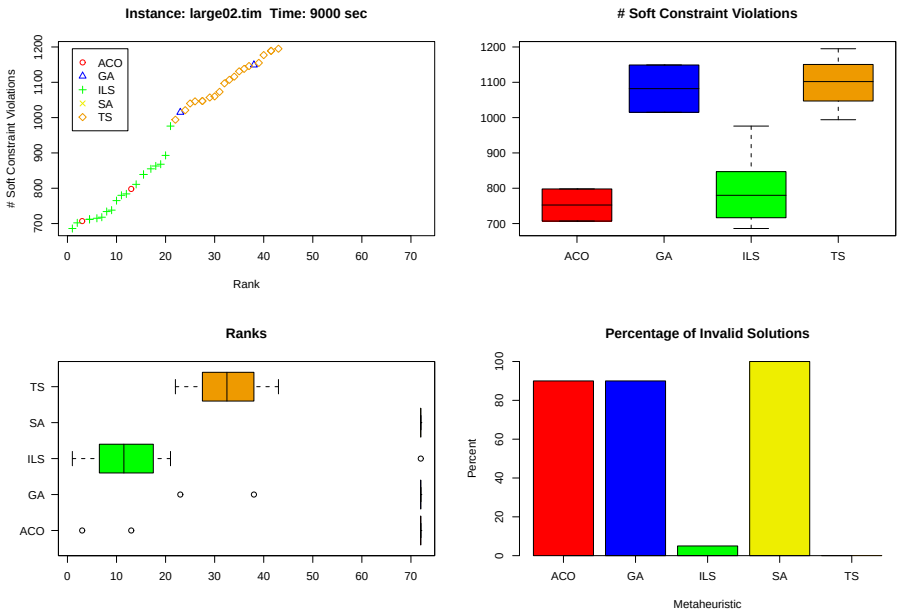

Fig. 5. Results for the `large02` instance.

about 97% of the trials against only 10% of ACO and GA. SA never reaches feasibility, while TS gives always feasible solutions but with worse results than ILS and ACO in terms of soft constraints.

The results presented here have to be read bearing in mind the context to which they belong. Strong restrictions have been made on the implementations of the metaheuristics, as the use of a single representation and a single search landscape, and a minimal use of problem specific heuristics. The use of a different representation, indirect and/or constructive, a different neighbourhood structure and local search, or more freedom in the use of additional heuristic information might give different results.

6 Conclusion

Based on the full set of results presented in the previous section, we can make the following general conclusions, also confirmed by the analysis made in [8]:

1. problem instance difficulty varies (sometimes significantly) between problem instances, across categories, and to a lesser extent, within a category (i.e. where all parameters to the generator except the random seed were the same), in terms of the observed aggregated performance of the metaheuristics. This is what we had expected, and reflects real data, where some specific problem (e.g. a particular choice of subjects by a particular student) can make timetabling much more difficult in a particular year;

2. the absolute performance of a single metaheuristic varies (sometimes significantly) between instances, within and, to a lesser extent, across categories;
3. the relative performance of any two metaheuristics varies (sometimes significantly) between instances, within and, to a lesser extent, across categories;
4. the performance of a metaheuristic with respect to satisfying hard constraints and satisfying soft constraints may be very different.

These conclusions lead us to believe that it will be very difficult to design a metaheuristic that can tackle general instances, even from the restricted class of problems provided by our generator. However, our results suggest that a hybrid algorithm consisting of at least two phases, one taking care of feasibility, the other taking care of minimising the number of soft constraint violations, is a promising research direction.

Additionally, we confirmed that knowing certain aspects of an instance does not guarantee that we will know about the structure of the search space, nor does it suggest *a priori* that we will know which metaheuristic will be best. This suggests the importance, in future, of trying to measure the search space characteristics directly; the aim here would be to try and match algorithms/parameter settings based on measurements of these characteristics, so that some of the *a priori* uncertainty of performance is removed.

This is ongoing work. In order to progress further in understanding these problems we have organised an international competition [23] based on our generator. We hope that this will shed more light on these difficult problems. We have also begun to analyse search space characteristics and relate these to metaheuristic performance.

Acknowledgments: This work was supported by the *Metaheuristics Network*, a Research Training Network funded by the Improving Human Potential program of the CEC, grant HPRN-CT-1999-00106. The information provided is the sole responsibility of the authors and does not reflect the Community's opinion. The Community is not responsible for any use that might be made of data appearing in this publication.

References

1. E. H. L. Aarts, J. K. Lenstra (eds.). Local Search in Combinatorial Optimization. John Wiley & Sons, Chichester, 1997.
2. M. Birattari, T. Stützle, L. Paquete, and K. Varrentrapp. A racing algorithm for configuring metaheuristics. Technical report, Intellektik, Technische Universität Darmstadt, Darmstadt, Germany, 2002.
3. T. Baeck, D. Fogel, and Z. Michalewicz. Evolutionary Computation 1: Basic Algorithms and Operators Institute of Physics, 2000.
4. E. Bonabeau, M. Dorigo, and G. Theraulaz. From Natural to Artificial Swarm Intelligence. Oxford University Press, 1999.
5. E. K. Burke, M. Carter (eds.), The Practice and Theory of Automated Timetabling: Selected Papers from the Second International Conference. Lecture Notes in Computer Science 1408, Springer-Verlag, Berlin, 1997.

6. M. W. Carter and G. Laporte. Recent developments in practical course timetabling. In [5]. 3–19, 1997.
7. V. Cerný. A Thermodynamical Approach to the Traveling Salesman Problem. Journal of Optimization Theory and Applications, 45:41–51, 1985.
8. M. Chiarandini and T. Stützle. Experimental Evaluation of Course Timetabling Algorithms. Technical Report, FG Intellektik, TU Darmstadt, 2002.
9. M. Dorigo and L. M. Gambardella. Ant Colony System: A cooperative learning approach to the traveling salesman problem. IEEE Transactions on Evolutionary Computation, 1(1):53–66, 1997.
10. M. Dorigo, V. Maniezzo, and A. Colorni. The Ant System: Optimization by a colony of cooperating agents. IEEE Transactions on Systems, Man, and Cybernetics, 26:29–41, 1996.
11. F. Glover and M. Laguna. Tabu Search. Kluwer Academic Publishers, Boston et al., 1998.
12. D. S. Johnson and L. A. McGeoch. The traveling salesman problem: A case study in local optimization. In E. H. L. Aarts and J. K. Lenstra (eds), Local Search in Combinatorial Optimization, New York, 215–310, 1997. John Wiley & Sons.
13. D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon. Optimization by simulated annealing: an experimental evaluation; partI, graph partitioning. Operations Research, 37(6):865–892, November-December 1989.
14. S. Kirkpatrick, C. D. Gelatt and M. P. Vecchi. Optimization by Simulated Annealing. Science, Number 4598(220):671–680, 1983.
15. H.R. Lourenço, O. Martin and T. Stützle Iterated Local Search In F. Glover and G. Kochenberger (eds), Handbook of Metaheuristics, Volume 57 of International Series in Operations Research & Management, 321–353, 2002, Kluwer Academic Publishers.
16. O. Martin and S.W. Otto. Partitioning of unstructured meshes for load balancing. Concurrency: Practice and Experience, 7:303–314, 1995.
17. B. Paechter, R. C. Rankin, A. Cumming, and T. C. Fogarty. Timetabling the classes of an entire university with an evolutionary algorithm. Parallel Problem Solving from Nature (PPSN) V. Lectures Notes in Computer Science 1498, Springer-Verlag, Berlin, 865–874, 1998.
18. L. Paquete and T. Stützle. Experimental investigation of iterated local search for coloring graphs. In S. Cagnoni, J. Gottlieb, E. Hart, M. Middendorf and G. Raidl (eds), Applications of Evolutionary Computing, Proceedings of EvoWorkshops 2002. Lectures Notes in Computer Science 2279, Springer-Verlag, Berlin, 122–131, 2002.
19. P. Ross, D. Corne, and H. Fang. Improving evolutionary timetabling with delta evaluation and directed mutation. In H. P. Schwefel, Y. Davidor, R. Manner (eds.), Parallel Problem Solving from Nature(PPSN) III. Lectures Notes in Computer Science, Springer-Verlag, 560–565, 1994.
20. O. Rossi-Doria, C. Blum, J. Knowles, M. Sampels, K. Socha, and B. Paechter. A local search for the timetabling problem. In Proceedings of the 4th international conference on the Practice And Automated Timetabling (PATAT 2002), Gent, Belgium, 124–127, 2002.
21. A. Schaerf. A survey of Automated Timetabling. Artificial Intelligence Review, 13:87–127, 1999.
22. T. Stützle. Local search Algorithms for Combinatorial Problems – Analysis, Improvements, and New Applications. PhD thesis, TU Darmstadt, Germany, 1998.
23. http://www.idsia.ch/Files/ttcomp2002.