


SUBMITTED TO COMPUTATIONAL OPTIMIZATION AND APPLICATIONS JOURNAL A GENETIC ALGORITHM TO SOLVE THE TIMETABLE PROBLEM

Ganchudur Bayandelger

Related papers

[Download a PDF Pack](#) of the best related papers 



[A genetic algorithm to solve the timetable problem](#)

Marco Dorigo

[Metaheuristics for high school timetabling](#)

Alberto Colorni

[Genetic algorithms and highly constrained problems: The time-table case](#)

Alberto Colorni

A GENETIC ALGORITHM TO SOLVE THE TIMETABLE PROBLEM

Alberto Colorni

Centro di Teoria dei Sistemi del CNR
Dipartimento di Elettronica e Informazione
Politecnico di Milano
Piazza Leonardo da Vinci 32
20133 Milano
tel. +39-2-2399-3567

Marco Dorigo

Politecnico di Milano Artificial Intelligence & Robotics Project
Dipartimento di Elettronica e Informazione
Politecnico di Milano
Piazza Leonardo da Vinci 32
20133 Milano
tel. +39-2-2399-3622
fax +39-2-2399-3411
e-mail: dorigo@elet.polimi.it

Vittorio Maniezzo

Politecnico di Milano Artificial Intelligence & Robotics Project
Dipartimento di Elettronica e Informazione
Politecnico di Milano
Piazza Leonardo da Vinci 32
20133 Milano
tel. +39-2-2399-3622
fax +39-2-2399-3411
e-mail: maniezzo@elet.polimi.it

Subject categories:

Programming: heuristic, stochastic;

Mathematics: combinatorics.

The paper presents an application of an adapted genetic algorithm to a real world instance of the timetable problem. The results of its application are described. We also compare our results with simulated annealing and tabu search.

Other keywords: timetable problem, genetic algorithms

Abstract

In this paper we present the results of an investigation of the possibilities offered by genetic algorithms to solve the timetable problem. This problem has been chosen since it is representative of the class of multi-constrained, NP-hard, combinatorial optimization problems with real-world application. First we present our model, including the definition of a hierarchical structure for the objective function and the generalized genetic operators which can be applied to matrices representing timetables. Then we report about the outcomes of the utilization of the implemented system to the specific case of the generation of a school timetable. We compare two versions of the genetic algorithm (GA), with and without local search, both to a handmade timetable and to two other approaches based on simulated annealing and tabu search. Our results show that GA with local search and tabu search with relaxation both outperform simulated annealing and handmade timetables.

(Introduction)

Evolutionary algorithms^[14] constitute a class of computational paradigms useful for function optimization inspired from the study of natural processes. They use a population of possible solutions, which are concurrently subject to modifications aimed at the determination of the optimal solution. A particularly efficient instantiation of evolutionary algorithms is represented by the *genetic algorithm* (GA)^[22], in which the natural analogy is population genetics. In the GA every possible solution is represented by a "digital individual" and after the generation of an initial set of feasible solutions (a *population*), individuals are randomly mated allowing the recombination of genetic material. The resulting individuals can then be mutated with a specific mutation probability. The new population so obtained undergoes a process of natural selection which favors the survival of the fittest individuals (the best solutions), and provides the basis for a new evolutionary cycle. The fitness of the individuals is made explicit by means of a function, called the *fitness function* (*f.f.*), which is related with the objective function to optimize. The *f.f.* quantifies how good a solution is for the problem faced. In GAs individuals are sometimes called *chromosomes*, and positions in the chromosome are called *genes*. The value a gene actually takes is called an *allele* (or *allelic value*). Allelic values may vary on a predefined set, that is called *allelic alphabet*.

Let P be a population of N chromosomes (*individuals* of P). Let $P(0)$ be the initial population, randomly generated, and $P(t)$ the population at time t . The GA generates a new population $P(t+1)$ from the old population $P(t)$ applying some *genetic operators*. The new population is created by means of the *reproduction* operator, that gives higher reproduction probability to higher fitted individuals, and by the subsequent application of the *crossover* and of the *mutation* operators, which modify randomly chosen individuals of population $P(t)$ into new ones. The overall effect of GAs work is to move the population P towards areas of the solution space with higher values of the fitness function.

The computational speed-up that we obtain using GAs with respect to random search is due to the fact that our search is directed by the fitness function. This direction is not based on whole chromosomes, but on their parts which are strongly related to high values of the fitness function; these parts are called *building blocks* [4,17]. It has been demonstrated^[3] that GAs are very efficient at processing building blocks. GAs are therefore useful for every problem where an optimal solution is composed of a collection of building blocks. This computational paradigm allows an effective search in very large search spaces. It has been recently applied to various kinds of optimization problems, including NP-complete problems^[15], e.g. Traveling Salesman^[24] and Satisfiability^[10], with satisfactory results.

The three basic genetic operators are:

- *reproduction*, an operator which allocates in the population $P(t+1)$ an increasing number of copies of the individuals with a *f.f.* above the average in population $P(t)$;
- *crossover*, a genetic operator activated with a probability p_c , independent of the specific individuals on which it is applied; it takes as input two randomly chosen individuals (parents) and combines them to generate two sons;
- *mutation*, an operator that causes, with probability p_m , the change of an allelic value of a randomly chosen gene; for instance, if the alphabet were $\{0,1\}$, an allelic value of 0 would be modified into 1 and vice versa.

The main goal of our research is to understand the limitations of the GA and its potentialities in addressing highly constrained problems, that is optimization problems where a minimal change to a feasible solution is very likely to generate an infeasible one. As a test problem we have chosen the timetable problem (TTP), that is known to be NP-hard^[12], but which has been intensively investigated given its great practical relevance^[1,5,6,7,9,11,13,16,20,21,25].

1. The timetable problem

We will use the construction of a timetable, or schedule of classes, for an Italian high school as the medium for our investigation. The ideas introduced in this paper can be applied, of course, to the solution of other, and possibly very different, instances of the timetable problem. The possibility of "on field" testing has been the main reason for the choice of this particular problem example. (In a typical Italian high-school, a class receives five hours of lessons, six days a week. Teachers may teach one or more subjects, usually in two or more classes. In addition to their eighteen-hour teaching demand, they have other activities, as described in the paper. Also, every teacher has the right to take one day-off per week, in addition to Sundays.)

The construction of the lesson timetable for an Italian high school may be decomposed in the formulation of several interrelated timetables. In fact, sections are always coupled in pairs, with a

couple of sections sharing many teachers and resources (e.g. laboratories). Two coupled sections can therefore be processed as an "atomic unit", not further decomposable given its high internal dependencies, but relatively isolated from other sections.

Given these premises, the problem is described by:

- a list of m teachers (20-24 in our case);
- a list of p classes involved (10 for the two coupled sections);
- a list of n weekly teaching hours for each class (30);
- the *curriculum* of each class, that is the list of the frequencies of the teachers working in the class;
- some external conditions (for example the hours during which some teachers are involved in other sections or activities).

A formal representation of the TTP is the following. Given the 5-tuple $\langle \mathbf{T}, \mathbf{A}, \mathbf{H}, \mathbf{R}, f \rangle$ where

\mathbf{T} is a finite set $\{T_1, T_2, \dots, T_i, \dots, T_m\}$ of m resources (teachers);

\mathbf{A} is a set of jobs (teaching in the p classes and other activities) to be accomplished by the teachers;

\mathbf{H} is a finite set $\{H_1, H_2, \dots, H_j, \dots, H_n\}$ of n time-intervals (hours);

\mathbf{R} is a $m \cdot n$ matrix of $r_{ij} \in \mathbf{A}$ (a timetable);

f is a function to be minimized, $f: \mathbf{R} \Rightarrow \mathbb{R}$;

we want to compute

$$\text{MIN } f(\sigma, \Delta, \Omega, \Pi)$$

where

- σ is the number of *infeasibilities*, as defined in the following;
- Δ is the set of didactic costs (e.g., not having the hours of the same subject spread over the whole week);
- Ω is a set of organizational costs (e.g., not having at least one teacher available for possible temporary teaching posts);
- Π is a set of personal costs (e.g., not having a specific day-off).

Every solution (timetable) generated by our algorithm is *feasible* if it satisfies the following constraints:

- every teacher and every class must be present in the timetable in a predefined number of hours;
- there may not be more than one teacher in the same class in the same hour;

- no teacher can be in two classes in the same hour;
- there can be no "uncovered hours" (that is, hours when no teacher has been assigned to a class).

The problem has been approached by means of linear programming with binary variables, using some heuristics^[11]. In fact, if it were approached with standard algorithms, i.e. defining binary variables x_{ijk} (where, according to the parameters previously specified, i identifies a teacher, j identifies a time-interval and k identifies a class) the problem would be represented by 6000 variables ($i = 1, \dots, 20$; $j = 1, \dots, 30$; $k = 1, \dots, 10$), which makes it intractable^[2,8,23]. We have decided to approach it by means of an evolutionary stochastic algorithm, namely a Genetic Algorithm (GA), introduced in the next section.

2. The genetic approach to the timetable problem

Some difficulties are encountered when applying GAs to constrained combinatorial optimization problems. The most relevant of them is that crossover and mutation operators, as previously defined, may generate infeasible solutions.

The following corrections to this drawback have been proposed.

- change the representation of a solution in such a way that crossover can be applied consistently;
- define new crossover and mutation operators which generate only feasible solutions;
- apply the crossover and mutation operators and then make some kind of *genetic repair* that changes the infeasible solutions to feasible ones through the use of a filtering algorithm.

In the traveling salesman case the most successful approaches have been the introduction of a new crossover operator^[18] and the application of genetic repair^[24]. The redefinition of mutation is in this case particularly straightforward: it is sufficient to exchange the position of two cities in the string. In the TTP on the other hand, even after the redefinition of both crossover and mutation, it has been necessary to implement genetic repair.

We now describe how we approached the problem of generating a school timetable for a pair of sections of an Italian high school, as described in Section 1.

The alphabet we chose is the set **A** of the jobs that teachers have to perform: its elements are the classes to be covered and other activities. We indicate:

- with the characters 1,2,3, .. ,0 the ten classes where the lessons have to be taught;
- with the character D the hours at disposal for temporary teaching posts;
- with the character A the hours for the professional development;

- with the character S the hours during which lessons are taught in classes of sections different from the two considered; this hours are fixed in the initialization phase and are called *fixed hours*;
- with the character ♦ the hours in which the teacher does not have to work;
- with the characters ----- the teacher's day-off.

Our alphabet is therefore $\mathbf{A} = \{1,2,3,4,5,6,7,8,9,0,D,A,S,\diamond,-\}$.

This alphabet allows us to represent the problem as a matrix \mathbf{R} (an $m \cdot n$ matrix of $r_{ij} \in \mathbf{A}$) where each row corresponds to a teacher and each column to a hour. Every element r_{ij} of the matrix \mathbf{R} is a gene; its allelic value may vary on the subset of \mathbf{A} specific to the teacher corresponding to the row containing the gene.

The problem is therefore represented by matrices similar to that proposed in Figure 1. To be a feasible timetable a matrix must satisfy the constraints discussed in Section 1.

Teacher-Subject	Mon	Tue	Wed	Thu	Fri	Sat
Literature - 1	♦11♦1	112♦♦	♦♦♦11	2212♦	11111	-----
Literature - 2	♦6♦♦6	7777♦	♦♦♦77	66♦♦7	-----	7777♦
Literature - 3	♦♦♦2♦	666♦6	2♦♦22	-----	6266♦	6622♦
Literature - 4	♦8♦♦♦	44♦♦♦	-----	♦♦4♦8	84888	88444
Literature - 5	-----	♦5555	♦♦355	♦♦353	3♦33♦	33♦♦♦
Literature - 6	000♦0	-----	0♦999	0099♦	9♦♦♦♦	♦9♦♦♦
English	152♦5	32411	53♦♦♦	♦♦♦♦5	43422	-----
German	77997	98800	607♦6	-----	♦6♦♦♦	♦♦08♦
History and Philosophy - 1	5♦33♦	♦3343	-----	55♦44	♦♦♦4♦	4555♦
History and Philosophy - 2	9♦♦8♦	-----	♦88♦0	990♦♦	0♦009	908♦8
Math and Physics - 1	-----	5♦♦♦♦	45434	4453♦	5♦55♦	♦4333
Math and Physics - 2	♦9♦09	09998	♦♦♦08	88800	♦9♦♦0	-----
Math - 1	SSSS2	2S1AA	112♦♦	♦♦♦1♦	-----	22♦1♦
Math - 2	6S66S	SS♦AA	♦7S6♦	-----	7777♦	♦♦♦6♦
Natural sciences	33444	80022	-----	7378♦	♦8995	5♦9♦♦
Art	84518	-----	96643	37279	2♦♦♦♦	01♦05
Experimental Physics	2277S	S♦♦67	-----	1166♦	♦♦2SS	SS1♦♦
Gymnastic - 1	SSS♦♦	♦♦♦34	345SS	♦♦♦SS	S5♦♦♦	-----
Gymnastic - 2	SSS♦♦	♦♦♦89	890SS	♦♦♦SS	S0♦♦♦	-----
Religion	4S853	♦♦♦♦♦	721S♦	SS♦♦♦	-----	SS690

Fig.1 - Example of a matrix representing a timetable.

The constraints are managed as follows.

- by the *genetic operators*, so that the set of hours to be taught by each teacher, allocated in the initialization phase, cannot be changed by the application of the genetic operators (which have been specifically redefined for this purpose);
- by the *filtering algorithm*, so that the infeasibilities caused by the application of genetic operators are, totally or partially, eliminated by filtering;
- by the *objective function*, so that selective pressure is used to limit the number of individuals with infeasibilities (infeasibilities are explicitly considered in the objective function, by means of high penalties).

It is possible to distinguish between two kinds of constraints: rows and columns. *Row constraints* are incorporated in the genetic operators and are therefore always satisfied; *column constraints* (infeasibilities due to superimpositions or uncovered classes) are managed by means of a combination of fitness function and genetic repair. Single-teacher solutions (i.e. solutions which satisfy a single teacher) are constrained each other by column constraints. Genetic repair must convert infeasible timetables into feasible ones, modifying them as little as possible.

We decided to manage the infeasibilities by means of both filtering and fitness function penalties because in this way the algorithm has a greater degree of freedom in moving through the search space. This choice is due to the difficulty of the problem: in our application, in fact, every teacher represents a TSP-like problem (consisting of the analysis of the permutations of a predefined symbol set), which has a search space of dimension

$$k = \frac{\left(\sum_h n_h \right)!}{\prod_h (n_h!)} = \frac{n!}{\prod_h (n_h!)}$$

where n_h is the number of repetitions of the h -th character in the row representing the teacher, and n is the total number of weekly hours. A teacher working in several classes has a value of k greater than that of a teacher working in fewer classes, the total number of teaching hours being the same.

3. The hierarchical structure of the objective function

The objective function is the basis for the computation of the *f.f.*, which provides the GA with feedback from the environment. The feedback is used to direct the population towards areas of the search space characterized by better solutions.

The need to distinguish between objective function (*o.f.*) and fitness function (*f.f.*) comes from the necessity to define the *o.f.* with reference to a cost minimization problem, while the GA has a structure which allows it to solve maximization problems. The translation from *o.f.* into *f.f.* is obtained by simply mapping the numeric values of the former into those of the latter, by means of a monotonically decreasing function. In particular, our system is based on a *linear dynamic fitness scaling* procedure^[19]. At each generation the maximum and the minimum objective function values of the individuals of the population are computed (max *o.f.* and min *o.f.* in Figure 2): they define an interval on the *o.f.* axis which is linearly mapped onto an interval of the *f.f.* axis, limited by two system constants MINFIT and MAXFIT, in such a way that min *o.f.* corresponds to MAXFIT and max *o.f.* corresponds to MINFIT. This procedure attains two objectives: first it minimizes the *o.f.* while it maximizes the *f.f.*, second, it strongly discriminates solutions belonging to populations whose individuals have very small variations of *o.f.* values, which is often the case in the late stages of the search process.

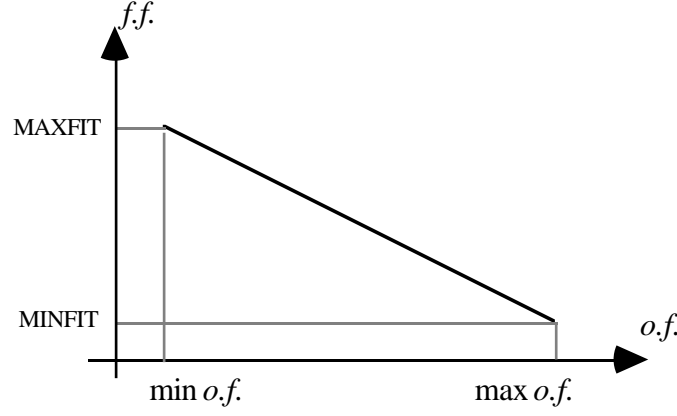


Fig.2 - Mapping the *o.f.* into the *f.f.*

The *o.f.* for our problem measures a generalized cost which represents the gap between the timetable considered and an *ideal* timetable, i.e. a timetable which satisfies all the different constraints. The *o.f.* for timetable \mathbf{R} is

$$z(\mathbf{R}) = \alpha \cdot \#_{\text{inf}} + \beta_1 \cdot s_{\Delta} + \beta_2 \cdot s_{\Omega} + \beta_3 \cdot s_{\Pi}$$

where

- $\#_{\text{inf}}$ is the number of infeasibilities in the matrix \mathbf{R} ;
- s_{Δ} measures the rate of dissatisfaction of didactic requirements in the matrix \mathbf{R} ;
- s_{Ω} measures the rate of dissatisfaction of organizational requirements in the matrix \mathbf{R} ;
- s_{Π} measures the rate of dissatisfaction of teachers' requirements in the matrix \mathbf{R} ;
- α, β_1, β_2 , and β_3 are weights chosen by the user to bias the timetable towards different aspects.

Choosing $\alpha \gg \beta_1, \beta_2, \beta_3$, we induce a *hierarchical structure* in the *o.f.*, so that we acknowledge the different relevance of the several groups of problem conditions. In our application, the following structure has been chosen:

- level 1*, feasibility conditions σ ;
- level 2*, management conditions (Δ, Ω, Π);
- level 3*, single teachers conditions.

At *level 1* we handle possible superimpositions of teachers (two or more teachers during the same hour in the same class) and "uncovered hours" for the classes (hours when a class is not covered by any teacher).

At *level 2* we consider the three following requirement typologies.

Δ - didactic requirements

- no more than 4 teaching hours a day for each teacher (Δ_1);
- not the same teacher every day at the last hour (Δ_2);
- uniform distribution of the hours of the same subject over the week (Δ_3);
- pairs of hours for classworks (Δ_4);

Ω - organizational requirements

- concentration on the same day (as much as possible) for parent-teacher meeting hours (Ω_1);
- no less than 2 teaching hours a day for each teacher (Ω_2);
- as few holes in a teacher's schedule as possible (Ω_3);

Π - teachers' requirements

- a weight is chosen, for the full set of requirements of the different teachers (successively defined at level 3) with respect to the other requirements of level 2; moreover, a teacher ranking (based on criteria such as seniority, external engagements, etc.) is defined.

At *level 3* we consider the preferences expressed by each teacher for his/her specific timetable. Each teacher assesses his/her personal requirements, such as the day-off desired or not having the first or the last hours, and so on. These assessments are then normalized, so that each teacher takes part with a specific quota to the determination of the requirements of the whole teaching staff.

Finally, an additional procedure was designed to compute the *f.f.* incrementally. In several instances in fact, a timetable of known fitness is modified by simply swapping, for example, two hours or two days. It is therefore worthwhile to maintain the values of some intermediate fitness components, so that only those directly affected by the modifications are computed again.

4. The algorithm

The algorithm, in Pascal-like notation, is the following:

```
initialization      {this routine creates a population of N individuals, satisfying for every individual a
                      set of constraints:
                      - every teacher (row) is given the right amount of hours to be taught.
                      - some hours are set to the "fixed hour status", which means they cannot be moved.}

while (NOT_VERIFIED_END_TEST) do {the end test is on the number of iterations performed}
  begin
    apply reproduction;
    apply crossover;
    for l:=1 to N do
      begin
        apply mutation of order k;
        apply day mutation;
        if (LOCAL_ON) then apply local search {LOCAL_ON is a boolean variable}
        if (num_infeasibilities > MAX_INFEASIBILITIES) then apply filter
                                {MAX_INFEASIBILITIES is a system constant}
      end;
    end.
```

We examine now the operators used by the GA and their computational complexity. This complexity is a function of:

- the number N of individuals composing the population,
- the activation probabilities chosen for each genetic operator,
- the computational complexity of the *f.f.*, of the local *f.f.* (defined below), and of the genetic repair (filter) algorithm.

We call **FF** the fitness function evaluation complexity, and **GR** the genetic repair (filter) complexity (see Appendix).

We have chosen to explicitly represent the activation probabilities in the complexity formulae because they are user-defined variables which can be set to zero in specific situations, thus heavily affecting the complexity of the operator they refer to.

Reproduction. This is the classical reproduction operator that promotes individuals with an above average value of the *f.f.* It gives every individual h a reproduction probability $p_r(h)$ equal to its fitness divided by the total fitness of the population. New populations are generated by using these reproduction probabilities in conjunction with Monte Carlo methods.

The complexity of one application of the reproduction operator to the whole population is then $O(\mathbf{FF} \cdot N)$, where \mathbf{FF} is the complexity of computing the fitness function.

Crossover. The task of this operator is that of efficiently recombining building blocks (defined below for our case), so that, given two parents, it is possible to generate two offspring with better *f.f.* values (or at least with one of them with a significantly better *f.f.* value). We call the *local fitness function* (*l.f.f.*) that part of the fitness function due only to characteristics specific to each teacher. Given two individuals (timetables) of the population, \mathbf{R}_1 and \mathbf{R}_2 , the rows of \mathbf{R}_1 are sorted in order of decreasing *l.f.f.*, and the best k_1 rows are taken as a building block. Then, the remaining $m - k_1$ (where m is the number of teachers) rows are taken from \mathbf{R}_2 to generate the first son. The second son is obtained from the non-utilized rows of \mathbf{R}_1 and \mathbf{R}_2 . The value of k_1 is determined by the program on the basis of the *l.f.f.* of both parents. This operator is applied in probability to each selected pair of potential parents: the probability of its application is the system parameter p_c .

The crossover operator is implemented by means of the following algorithm:

pair randomly the individuals of the population

for each pair of individuals **do**

with probability p_c **do** **begin** { p_c is a control parameter}

compute the *l.f.f.* of the rows of the two individuals;

sort by decreasing values of the *l.f.f.* the rows of the two individuals;

create two sons merging twice the two individuals

{the first son is generated taking the best k_1 rows from the better parent and the worst rows from the worse parent; the second son is generated using the remaining unused rows from both the parents};

end;

The complexity of one application of the crossover operator is $O[(N \cdot p_c) \cdot (m \cdot \mathbf{LFF})]$, where \mathbf{LFF} is the complexity of computing the *l.f.f.*

Mutation of order k. This operator takes k contiguous genes and swaps them with another k contiguous non-overlapping ones belonging to the same row. Mutation of order one is a special case of this operator. It cannot be applied when, among the genes to be mutated, there are some special characters, like A or S (these fixed hours have in fact been allocated, during the initialization phase, to unconvertible activities). This operator is applied in probability to each row of each individual, the probability of its application is the system parameter p_{mk} (p_{m1} in the case of mutation of order one).

A particular kind of mutation is *day mutation*, which takes one day and swaps it with another one belonging to the same row. The i -th day, in a teacher timetable, is a substring containing genes that codify five contiguous hours, from the first to the fifth of a same day. It is a special case of mutation

of order k and has been introduced for efficiency reasons (with special reference to day-off allocation) and it is controlled by a specific application probability parameter, p_{md} .

The complexity of one application of the mutation of order k operator to the population is $O(N \cdot n \cdot m \cdot p_{mk})$, where p_{mk} is the probability of application of the mutation operator.

Local search. This operator moves a solution to its local optimum. It works in two stages. In the first it tries to eliminate infeasibilities without worsening second-level costs. This is done using a procedure taken from the filter algorithm – see appendix – that identifies the causes of the infeasibilities and removes them by swaps of hours. The second stage is a 2-opt that swaps hours and days until no better solutions are present in the neighborhood of the current timetable. The operator thus moves a solution to a point of the search space that is locally optimal with respect to a neighborhood defined by every possible swap of hours and of days.

Filter. The filter operator takes as input an infeasible solution and returns as output a feasible one. It is used to ensure global feasibility to a timetable and is based on the observation that in each column (hour) of the matrix every class must be present once and only once. If it were present twice or more times, there would be teacher superimpositions. If it were not present, the class would be uncovered. It is based on four procedures of increasing computational cost which are applied sequentially. The first two procedures identify swaps of hours of one teacher that eliminate infeasibilities. The other two procedures modify the schedules of more than one teacher.

The detailed steps comprising the filter algorithm are shown in the Appendix.

Table 1 gives the main properties of the considered genetic operators.

Tab.1 - Genetic operators and feasibility in the TTP case

Operator name	Global feasibility	Row feasibility
<i>Reproduction</i>	Maintained	Maintained
<i>Crossover</i>	Not maintained	Maintained
<i>Mutation of order k</i>	Not maintained	Maintained
<i>Day mutation</i>	Not maintained	Maintained
<i>Filter</i>	Recovered	Maintained

Row feasibility is maintained by all the operators presented, while this is not the case for column feasibility: the goal of filtering will, therefore, be that of recovering column - hence global - feasibility for any given timetable.

Summarizing, the operators have the following computational complexities (when applied to a population of dimension N):

Reproduction $O(\mathbf{FF} \cdot N)$
Crossover $O(\mathbf{LFF} \cdot N \cdot m \cdot p_c)$

<i>Mutation of order k</i>	$O(N \cdot m \cdot n \cdot p_{mk})$
<i>Local search</i>	$O(\mathbf{FF} \cdot N \cdot m \cdot n)$
<i>Filter (Genetic Repair)</i>	$O(N \cdot m^6 \cdot n^3)$ {see Appendix}
<i>Fitness function FF</i>	$O(m^2 \cdot n^2)$
<i>Local fitness function LFF</i>	$O(m \cdot n^2)$

5. Computational results

The program we used for our experiments was written using the C language on an IBM PC with standard configuration. The model and the program were tested by defining the timetable for a large high school in Milan, in which the timetable was handmade by a group of teachers. In the academic year 1992-93 the handmade timetable had a cost of 234 (as measured with our objective function). We cooperated with the teachers who usually define the timetable. This collaboration has been ongoing, from the design (for requirement definition) until the validation phase, which gave the results reported below. Moreover, to assess the relative efficacy of our approach, we compared our approach with simulated annealing (SA), adapting the indications of Abramson^[1], and with tabu search (TS), adapting the approach of Hertz and de Werra^[21].

Parameter settings

We conducted a number of experiments to determine an efficient parameter settings for our GA. The control parameters tested were:

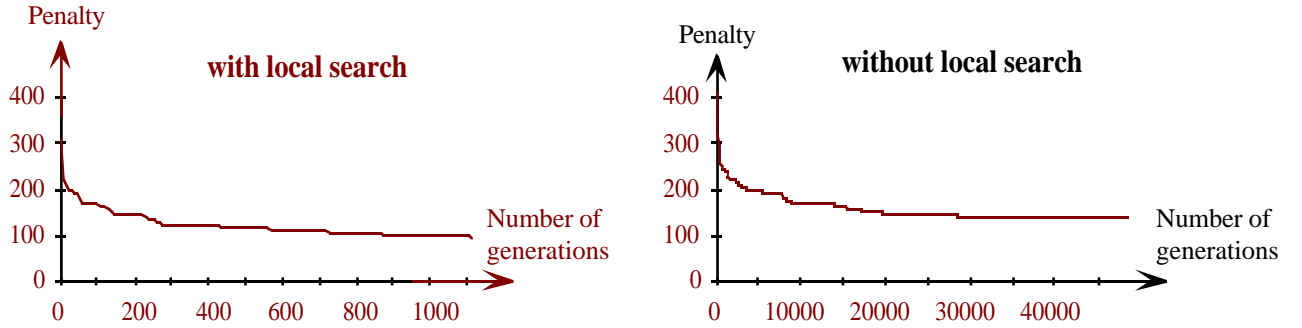
- i* control probabilities: p_{m1} , p_{mk} and p_{md} of the different mutation operators, p_c of the crossover;
- ii* local search enabled/disabled;
- iii* cost of infeasibilities: it has been set to a high value (1000, one order of magnitude greater than weights for management conditions) or to a low value (3, to ease explorations of infeasible regions).

Point (*i*) was tested considering 5 sets of values for each parameter chosen at regular intervals. The range of the intervals would be enlarged if the best performing value resulted to be at one extreme of the range. Five runs were carried out for each value combination, with only one change from the initial setting, performed on a parameter per run. Each run lasted eight hours on an IBM PC 486. In this way, we could identify a good parameter setting, and the values are reported in Table 2. The algorithm was found somewhat insensitive when changes of p_{m1} and p_c were held to within ± 0.2 . however, this insensitivity was not evident for changes of p_{md} (day mutation probability) or p_{mk} , the values of which should be kept very small.

Tab.2 - Parameter settings

parameter	value
p_{m1}	0.30
p_{mk}	0.01
p_{md}	0.01
p_c	0.80

Point (ii) was tested running the algorithm 10 times with local search enabled and 10 times with local search disabled, always using the optimal parameter setting found in point *i*. The average cost of the timetable found without local search was 160.1 (variance $\sigma=14.4$) with a best cost 138. Meanwhile, the average cost with local search was 111.1 ($\sigma=16.2$) with a best cost 91. The GA with local search is therefore definitely superior to its counterpart, in accordance with the indications expressed by Mühlenbein^[24]. Figure 3 presents the evolution of the best values in the two cases. Both runs lasted eight hours. The difference in the number of generations is due to the computational cost of local search. As a further test, we applied the local search operator to the best timetables found without local search, thus carrying them to their local optima. The average cost of these locally optimized timetables was 157.7 ($\sigma=14.4$), with a best cost 136. This shows that it is worthwhile to search in the space of the local optima: local optimization of the result of genetic search is not as effective as genetic search among local optima.

**Fig.3 - Best values with and without local search.**

Finally, tests on point (iii) suggest that a high value for the cost of infeasibilities should be used when local search is off, a low value used when local search is on. In the case in which local search is on, swaps are applied which maximally decrease the cost of the timetable; i.e., the highest-cost, unsatisfied requirements are removed first. When very low infeasibility costs are used, local search tries first to assemble a good timetable from the point of view of second level requirements, and only later it tries to make it feasible. Since we have an explicit filter operator applied after local search, we observe that, on the average, local search coupled with low infeasibility cost is much more effective than local search coupled with high infeasibility costs. In fact, if infeasibilities had a high cost, local search would first try to recover feasibility, even though this process leads to a great increase of

second-level costs. Subsequent local optimization, performed on feasible or quasi-feasible timetables, cannot change these second-level costs much. A further benefit of low infeasibility costs results from the fact that in the first iterations more infeasible timetables survive in the population. This broadens the search region which can be reached from the current population, thus allowing the exploration of otherwise unreachable areas.

The case of GA without local search yields opposite results. With low infeasibility costs, filtering is always applied intensively, thus dramatically slowing down the search process. The surviving infeasible solutions however are usually the best-rated ones and generate many offspring.

As a final observation, note also that, while exploring promising zones of the search space, the algorithm always identified feasible solutions within very few iterations; very good solutions, however, needed many iterations to be devised.

Comparison with other approaches

We provide a comparison with two alternative approaches to evaluate how well our algorithm performs with respect to the state of the art of automatic timetable design. Again, the algorithm was implemented and tested in a high school in Milan. Since we could use any predefined population of solutions as our starting point, we chose the previous year's timetables and adjusted them to meet the new needs. This gave us a proven basis from which to begin, thereby saving computing time and leading to improved solutions with respect to the handmade ones. The PC 386 environment allowed on-site runs, and the carefully designed menu-based user interface allowed the users both to directly interact with the computer in the input phase and to slightly edit the final solution: all these features, beside increasing the effectiveness of the interaction, greatly helped the acceptance of the package by the users.

When we applied our algorithm to the timetable used in the previous year in the school, we were able to better arrange many lessons, so that both some didactical requirements and several teachers' preferences were better satisfied. The total cost of the hand-designed timetable was in fact of 234, while our GA has been able to lower it to 91. Note that we computed for our problem a lower bound of 54 for the cost function, due to the presence of inconsistent single teachers requirements and fixed hours. For example, so many teachers chose Saturday as desired day-off, which made it impossible to satisfy all of them within a feasible timetable. Therefore, any timetable was bound to reflect costs due to unsatisfied day-off requests. Similarly, some teachers had fixed hours that did not meet their requirements, which again introduced unavoidable costs.

A comparison was carried out with respect to a Simulated Annealing (SA) based timetable designer. We implemented a system following the indications presented by Abramson^[1], using the non-genetic part of our system as objective function evaluator. Given the differences between the specific problems undertaken by Abramson and by us, we had to adapt the SA to work with our fitness function. In particular, we defined the neighborhood of a timetable as we had done in the GA

case. Also, the SA was asked to minimize not just the number of infeasibilities, but also didactical, organizational and teachers' requirements.

The use of this algorithm in our example problem led to timetables consistently worse than the handmade ones: it could not even recover the initial cost level when it was let evolve from the previous year timetable (final results all had a cost well over 300). And while SA was very efficient in recovering feasible timetables from infeasible ones, it could not effectively optimize feasible timetables. This was true both in the case of high and of low infeasibility costs.

We therefore modified the basic SA in two ways. First, we tested a SA with *reinitialization* of the temperature. As soon as a timetable got frozen, i.e., when the SA was unable to improve the current timetable since no proposed swap was accepted anymore, the temperature was set to its starting level. The annealing process could then start again from the previously frozen timetable. The best timetable we found with this modified algorithm had a cost of 183. A graph of the cost evolution for the corresponding run is presented in Figure 4, where the lower line plots the evolution of the best so far solution and the higher one the evolution of the current solution.

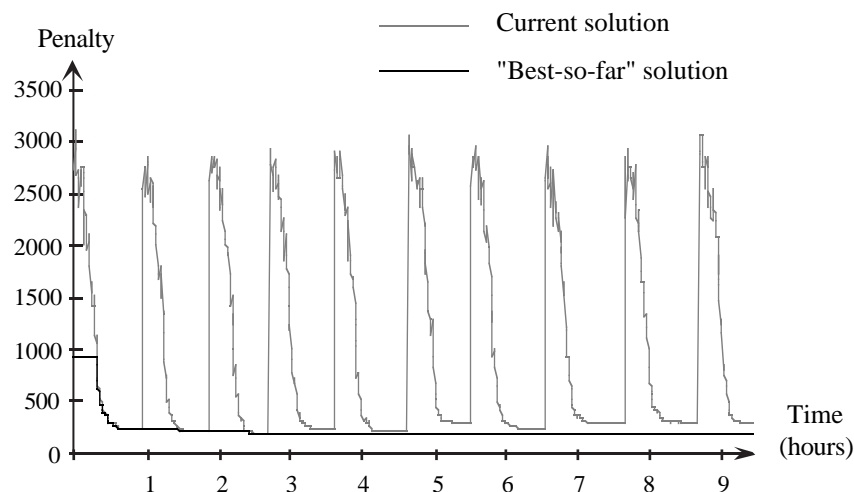


Fig.4 - SA with temperature reinitialization.

Alternatively, we tried to relax the problem when the timetable became frozen. We set the infeasibility costs to 3 for a given (parametric) number of iterations after which the normal cost function was applied again. This approach, coupled with a cooling rate of 0.9, yielded a timetable with a cost of 164. A graph of the cost evolution of the "best-so-far" and current solutions for the corresponding 10-hour-long run is presented in Figure 5.

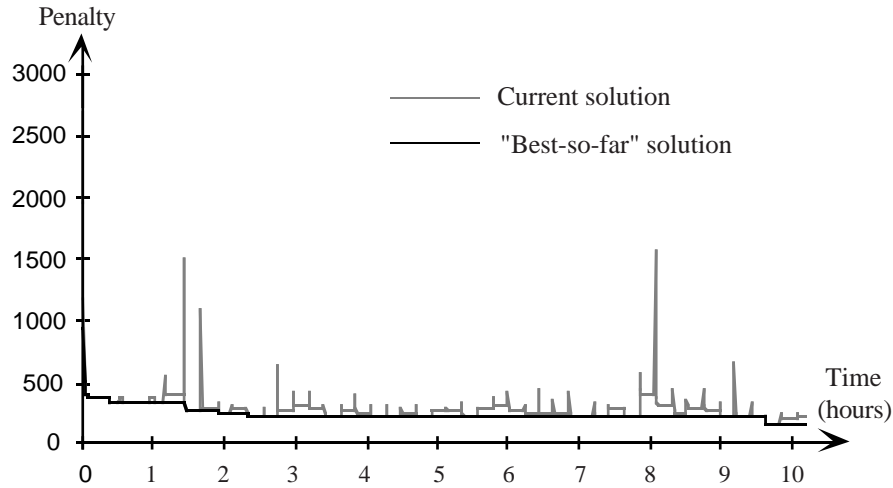


Fig.5 - SA with temporary relaxations.

Finally, we tested tabu search (TS), following the indications of Hertz and De Werra^[21]. We implemented a variable-length tabu list and used part of our system to assess the cost of the timetables, as in the case of SA. The results achieved with this simple model were not very satisfying, so we implemented a relaxation procedure identical to that used with SA. The results so obtained were very good indeed, reliably producing timetables with costs lower than 100, with the best one 85.

In summary, we obtained the following results:

Algorithm	Best	Average	Std. dev.
Handmade	234	234	//
GA	138	160	15
GA with local search	91	111	16
TS	184	196	13
TS with relaxation	85	97	12
SA with reinitialization	183	262	57
SA with relaxation	164	174	25

6. Conclusions

In this paper we have presented a model, a class of algorithms and a computing program for the timetable problem, with special reference to a real world application (the timetable of an Italian high school). We compared our GA-based approach with various versions of simulated annealing and tabu search. In our experiments GAs produced better timetables than simulated annealing, but slightly worse timetables than tabu search. An advantage of GAs over both SA and TS is that GAs

give the user the flexibility of choosing within a set of different timetables. This is an important feature as the evaluation of a timetable is done by an objective function which can miss some characterizing aspects. This, in turn, can make a timetable with a slightly higher cost more desirable than one with a lower cost.

The main contributions of our work are:

- the definition of genetic operators that minimize generalized cost functions (which penalize the possible infeasibilities of the generated solutions);
- the distribution over genetic operators, fitness function and genetic repair of the management of the infeasibilities;
- the hierarchical structuring (in our case on three levels) of the *o.f.*, in order to allow an easy and effective definition of the relevance of the different objectives used;
- the filtering algorithm, i.e. an algorithm capable of recovering from infeasibilities, converting an infeasible solution into a feasible one.

We believe that our approach is a useful generalization of the GA and can be applied to other highly constrained combinatorial optimization problems.

Acknowledgments

We would like to thank Dan Jurafsky, Chris Meluish, and Ed Watkins for helpful comments on a draft version of this paper.

References

1. A. ABRAMSON, 1991. Constructing school timetables using simulated annealing: sequential and parallel algorithms, *Management Science* 37, 98–113.
2. E.A. AKKONYUNLU, 1973. A linear algorithm for computing the optimum of university timetable, *Computer Journal* 16, 347–350.
3. A. BERTONI and M. DORIGO, 1993. Implicit parallelism in genetic algorithms, to appear in *Artificial Intelligence*.
4. L. BOOKER, D.E. GOLDBERG and J.H. HOLLAND, 1989. Classifier systems and genetic algorithms. *Artificial Intelligence* 40, 235–282.
5. M.W. CARTER, 1986. A survey of practical applications of examination timetabling algorithms, *Operations Research* 34, 193–202.
6. N. CHAHAL and D. DE WERRA, 1989. An interactive system for constructing timetables on a PC, *European Journal of Operational Research* 40, 32–37.

7. A. COLORNI, M. DORIGO and V. MANIEZZO, 1992. Gli algoritmi genetici e il problema dell'orario. *Rivista di Ricerca Operativa* 60, 5–31, (in Italian).
8. J. CSIMA and C.C. GOTLEIB, 1961. Tests on a computer method for construction of school timetables, *Communications of the ACM* 7, 160–163.
9. L. DAVIS and F. RITTER, 1987. Schedule optimization with probabilistic search, *Proceedings of the Third IEEE Conference on Artificial Intelligence Applications*.
10. K.A. DE JONG and W.M. SPEARS, 1989. Using genetic algorithms to solve NP-complete problems, *Proceedings of the Third International Conference on Genetic Algorithms*, Morgan Kaufmann.
11. D. DE WERRA, 1985. An introduction to timetabling, *European Journal of Operational Research* 19, 151–162.
12. S. EVEN, A. ITAI and A. SHAMIR, 1976. On the complexity of timetable and multicommodity flow problems, *SIAM Journal of Computing* 5 :4, 691–703.
13. J.A. FERLAND and S. ROY, 1985. Timetabling problem for university as assignment of activities to resources, *Computers & Operations Research* 12, 207–218.
14. D.B. FOGEL, 1992. A brief history of simulated evolution, *Proceedings of the First International Conference on Evolutionary Programming*, D.B. Fogel & J.W. Atmar (eds.), Evolutionary Programming Society, La Jolla, CA.
15. M.R. GAREY and D.S. JOHNSON, 1979. *Computers and intractability*, W.H. Freeman & Company.
16. P. GIANOGLIO, 1990. Application of neural networks to timetable construction, *Proceedings of the Third International Workshop on Neural Networks & Their Applications*.
17. D.E. GOLDBERG, 1989. *Genetic algorithms in search, optimization & machine learning*, Addison-Wesley, Reading, Massachussets.
18. D.E. GOLDBERG and R. LINGLE, 1985. Alleles, loci, and the traveling salesman problem, *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, Lawrence-Erlbaum.
19. J.J. GREFENSTETTE and J.E. BAKER, 1989. How Genetic Algorithms Work: A Critical Look at Implicit Parallelism, in Schaffer (ed.), *Proceedings of the Third International Conference on Genetic Algorithms*, Morgan Kaufmann.

20. A. HERTZ, 1989. La coloration des sommets d'un graphe et son application a la confection d'horaires, *Ph.D. Thesis No 785*, Ecole Polytechnique Federale de Lausanne, Switzerland.
21. A. HERTZ and D. DE WERRA, 1989. Informatique et horaires scolaires, *Output 12*, 53–56.
22. J.H. HOLLAND, 1975. *Adaptation in natural and artificial systems*, The University of Michigan Press, Ann Arbor, Michigan. Reprinted by MIT, 1992.
23. N.H. LAWRIE, 1969. An integer programming model for a school timetabling problem, *Computer Journal 12*, 307–316.
24. H. MÜHLENBEIN, 1989. Parallel genetic algorithms, population genetics and combinatorial optimization, *Proceedings of the Third International Conference on Genetic Algorithms*, Morgan Kaufmann.
25. J.M. MULVEY, 1982. A classroom/time assignment model, *European Journal of Operational Research 9*, 64–70.

Appendix - The filtering procedure

The steps composing the filter algorithm (which actually are as many filters themselves) are the following.

Step 0: for each column h ($h=1, \dots, n$) of the matrix:

- compute the set of classes having superimpositions and put its elements in the list **over**(h) together with their relative matrix coordinates;
- compute the set of uncovered classes and put its elements in the list **miss**(h) together with their relative matrix coordinates.

Step 1: until there exist pairs of classes c_i and c_j such that:

- c_i and c_j are present in the same row (that is, classes c_i and c_j have a common teacher)
- in one of the rows where they are both present, they occupy columns h and k , with $c_i \in \mathbf{over}(h)$, $c_i \in \mathbf{miss}(k)$, $c_j \in \mathbf{over}(k)$, $c_j \in \mathbf{miss}(h)$

swap c_i and c_j .

The effect of the algorithm is shown in Figure 6.

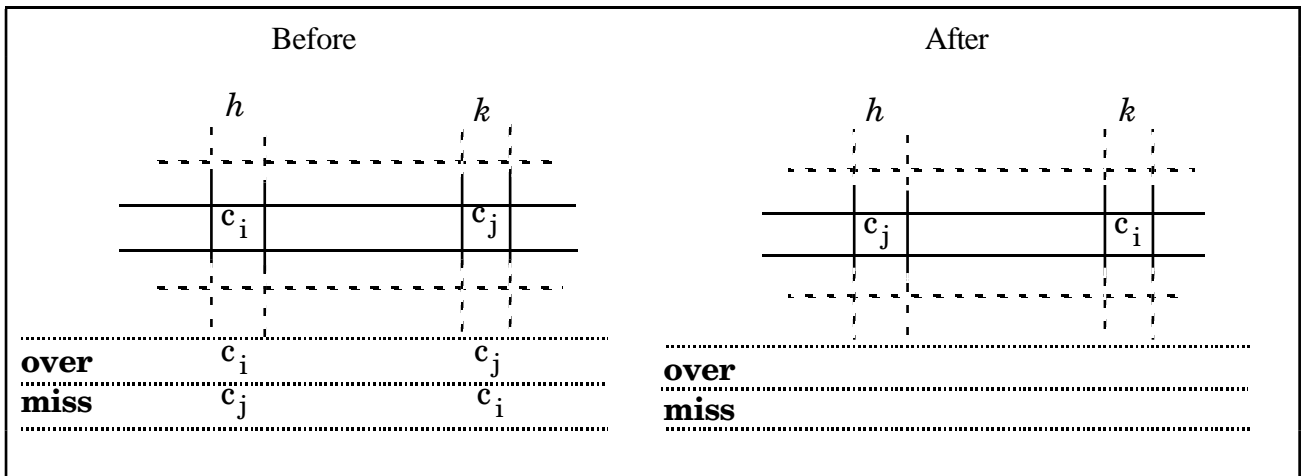


Fig.6 - Swapping two classes.

Step 2: until there exist pairs of elements, constituted by a class c_i and a *movable gene* e_j (that is, a gene that is either a character D or \diamond), such that:

- c_i and e_j are present in the same row
- in one of the rows where they are both present, they occupy columns h and k with $c_i \in \mathbf{over}(h)$, $c_i \in \mathbf{miss}(k)$

swap c_i and e_j .

The effect of the algorithm is shown in Figure 7.

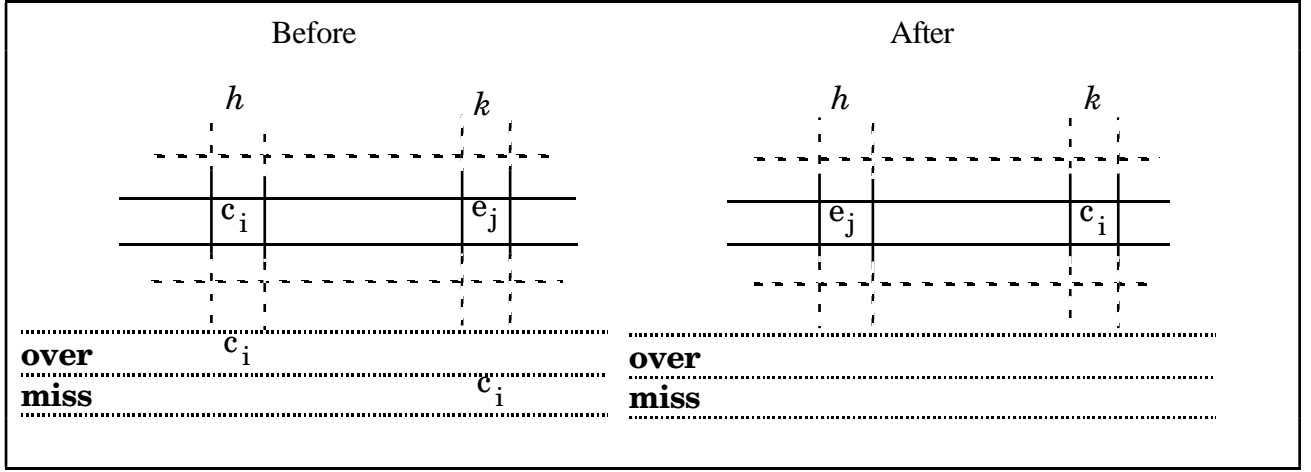


Fig.7 - Swapping two elements (a class and a movable gene).

Step 3: until there exist *transitive paths among classes*, that is paths connecting classes $c_i, c_j, \dots, c_s, c_t$, such that

- the classes $c_i, c_j, \dots, c_s, c_t$ are two by two in the same row
- $c_i \in \mathbf{over}(h), c_i \in \mathbf{miss}(k), c_j \in \mathbf{over}(k), \dots, c_s \in \mathbf{miss}(l), c_t \in \mathbf{over}(l), c_t \in \mathbf{miss}(h)$

swap each class with the following one belonging to the transitive path and present in the same row.

The effect of the algorithm is shown in Figure 8, where only a two-step path is considered. Since the complexity of the algorithm grows exponentially with the path length, we set the maximum length path to two. However, the algorithm can be generalized for handling longer paths.

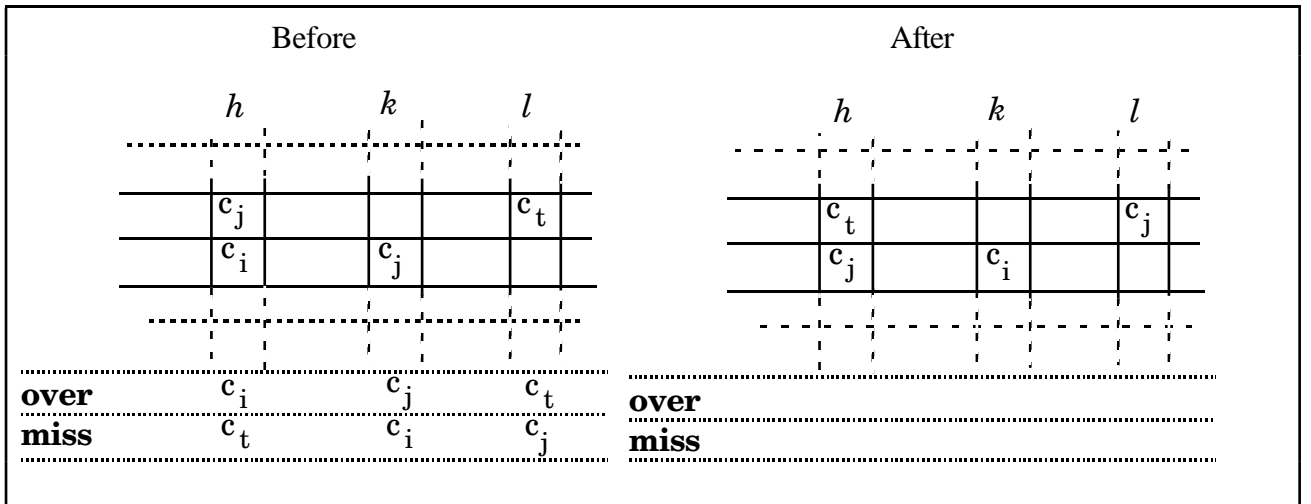


Fig.8 - Swapping classes in a transitive path.

Step 4: until there exist *transitive paths among elements*, where an element can be a class or a movable gene in the sense defined in step 2, behave as in step 3 (allowing any class to be substituted by a movable gene).

The effect of the algorithm is shown in Figure 9.

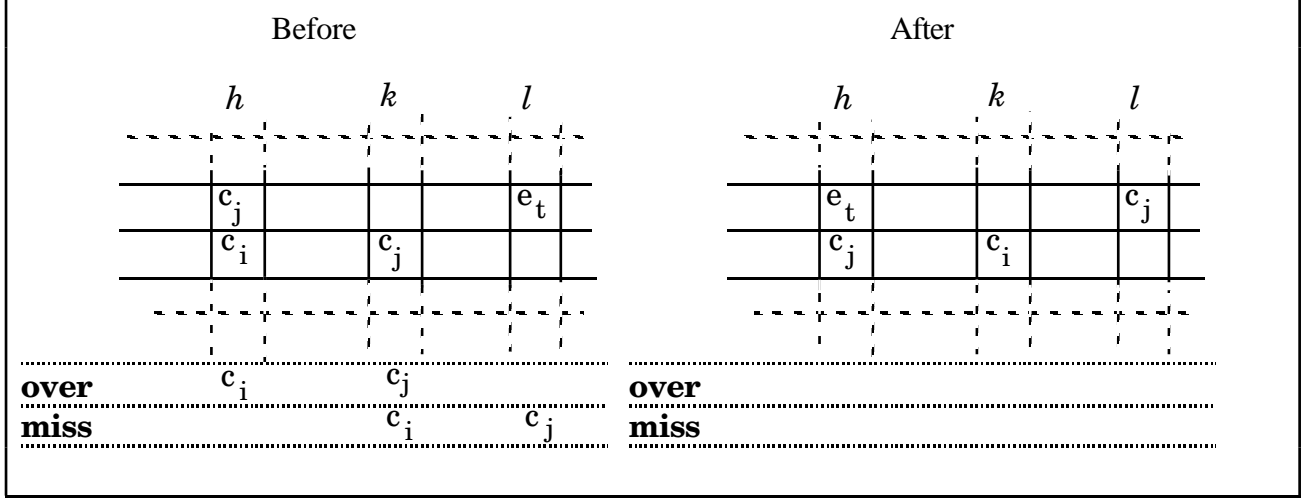


Fig.9 - Swapping elements in a transitive path.

The computational complexities of the steps are the following.

Step 0: $O(m \cdot n)$

Step 1: $O(m^3 \cdot n^2)$

Step 2: $O(m^3 \cdot n^2)$

Step 3: $O(m^6 \cdot n^3)$

Step 4: $O(m^6 \cdot n^3)$

Steps 3 and 4 imply a search in the set of links among superimposed and uncovered classes; it has a complexity that grows exponentially with the length of the path. For this reason we have (parametrically) limited to two the length of the path: the remaining infeasibilities (very few, as was found experimentally) are left to the penalty/reproduction mechanism. With such a restriction the complexity of the procedure is polynomial, in our case $m^6 \cdot n^3$.

We propose, as an example, an outline of the computation of the complexity for Step 1. In this case, we have to find all the columns where an element is missing in one and superimposed in the other and vice-versa. Therefore, for each column h (complexity n), for each element $c_i \in \mathbf{over}(h)$ (complexity at most m) and for each element $c_j \in \mathbf{miss}(h)$ (complexity at most m) we have to identify another column k (complexity n) having $c_i \in \mathbf{miss}(k)$, $c_j \in \mathbf{over}(k)$ (complexity $m+m$). The complexity is thus $n \cdot m \cdot m \cdot n \cdot 2 \cdot m = 2 \cdot n^2 \cdot m^3$, that is $O(m^3 \cdot n^2)$.

Observation 1. Note that m is a large overestimate of the length of the over and miss lists, which in the real cases have usually lengths 0, 1 or 2. Therefore, the factor maximally affecting the computational speed of the filter operator is the power of n .

Observation 2. Steps 1 and 2 are special cases respectively of steps 3 and 4. They have been introduced for efficiency reasons, and could have been obtained from steps 3 or 4 setting the path length to one.