An evolutionary solver for mixed integer programming

João Pedro Pedroso

Technical Report Series: DCC-2007-8

Departamento de Ciência de Computadores

Faculdade de Ciências da Universidade do Porto
Rua do Campo Alegre, 1021/1055,
4169-007 PORTO,
PORTUGAL

Tel: 220 402 900 Fax: 220 402 950
http://www.dcc.fc.up.pt/Pubs/

An evolutionary solver for mixed integer programming

João Pedro Pedroso
DCC – FC, Universidade do Porto, Portugal
and
INESC Porto, Portugal
jpp@fc.up.pt

October 2007

Abstract

In this paper we introduce an evolutionary algorithm for the solution of mixed integer programs. The strategy is based on the separation of the set of variables into the integer subset and the continuous subset. The main idea is that if the integer variables are fixed by the evolutionary system, the continuous ones can be determined in function of them by a linear program, which simultaneously provides an evaluation of those variables. We extend this idea to the case were some of the integer variables are fixed by the evolutionary system and the remaining ones, as well as the continuous ones, are determined in function of them. Branch-and-bound and a specialised version of the relax-and-fixed heuristic are used to solve the mixed-integer subproblems.

When a particular assignment of the integer variables set by the evolutionary system leads to a feasible solution, its evaluation is determined directly by the objective function. If the variables correspond to an infeasible solution, the evaluation is measured by the number of variables that could not be fixed, due to infeasibility in the subproblem; solutions with more variables fixed are preferred.

We report results obtained for some standard benchmark instances, and compare them with those obtained by time limited branch-and-bound. For a set of difficult instances, the evolutionary algorithm could almost always improve the solution obtained by branch-and-bound on the same amount of CPU time.

1 Introduction

Integer linear programming problems are widely described in the combinatorial optimisation literature, and include many well-known and important applications. Typical problems of this type include lot sizing, scheduling, facility location, vehicle routing, and more; see, for example, [14]. The problem consists of optimising a linear function subject to a set of linear constraints, in the presence of integer and, possibly, continuous variables. If the subset of continuous variables is empty, the problem is called *pure integer* (IP). In the more general case, where there are also continuous variables, the problem is usually called *mixed integer* (MIP); this work will focus on this general problem.

1.1 The evolutionary structure

Evolutionary algorithm (EA) is a term broadly used to classify problem solving techniques which use the evolution of a set of solutions (the population), with the aim of adapting

them, in such a way that, as the iterative process pursues, the algorithm provides more and more adapted answers to the formulated problem. New solutions are produced based on operations that somehow mimic genetics in natural evolution.

The main idea for the conception of the algorithm described in this paper is that if the integer variables of a MIP are fixed by an evolutionary system, the problem that remains to be solved is a standard linear program (LP); this can be done exactly and efficiently, for example by means of the simplex algorithm or by interior point methods. We are therefore able to make the integer variables evolve through an evolutionary algorithm; after they are fixed by the EA, we can determine the continuous variables and the value of the objective in function of them, by solving a subproblem that is a linear program on the continuous variables.

This idea is extended to the case were the subproblem also determines some integer variables, in addition to the continuous ones. In this case, the subproblem is also a MIP, but with a smaller number of variables, and hence easier to solve than the original problem. The EA fixes most of the integer variables, and the remaining integer variables and the continuous ones are determined in function of them, using branch-and-bound and a specialised version of the relax-and-fixed heuristic.

Notice that this algorithm, as opposed to branch-and-bound, does not work with the solution of continuous relaxations of the initial problem. The solution of MIP subproblems is used for determining the value of a small subset of the integer variables and that of the continuous ones (if some). Additionally, it determines the value of the objective that corresponds to that particular instantiation of the integer variables.

1.2 Background

The most well known algorithm for solving MIPs is branch-and-bound (B&B) (for a detailed description see, for example, [7]). This algorithm starts with a continuous relaxation of the MIP, and proceeds with a systematic division of the domain of the relaxed problem, until the optimal solution is found. In cases where the exploration tree of B&B is small, due to a small number of variables or to a structure of the problem which allows many branches to be pruned, the time that B&B requires to solve a problem may be reasonable for most of the applications. However, for difficult problems the exploration of the tree may take an unacceptable amount of time. In this case, B&B may still provide a solution, the best one found in the time allowed to perform the search.

In cases where B&B fails to find a good feasible solution one can try alternative heuristic procedures. A well-known heuristic for some kinds of problems is relax-and-fix, which fixes only a part of the integer variables on each B&B solution. The EA here proposed makes use of an extended version of this heuristic.

Notice that EAs cannot prove that the solution found is optimal. Moreover, in what concerns convergence, the best that can be proved is that for elitist EAs we obtain a sequence of evaluations that converges to the optimal objective value as the number of generations tends to infinity. Nevertheless, as for many applications the proof of optimality is not required and good feasible solutions are sufficient for practical implementation, the EAs that we propose in this paper may have many suitable uses.

Initial work on random search methods for integer optimisation was presented in [6], where the values of the fractional solutions obtained in the solution of the LP relaxation are randomly rounded up or down, for tentatively obtaining a good feasible solution. An evolution strategy for non linear, unconstrained optimisation in integer variables is described

in [11]. Another heuristic method for solving linear integer problems, where tabu search is associated to a branch-and-cut framework, is proposed in [4]. Local search methods for integer programming based on constraint satisfaction representations are provided in [13], where a problem is represented by a set of variables with finite domains, and a set of linear equality constraints. These are grouped into hard constraints and soft constraints; the former must be verified, whereas the latter may be violated, and the objective is to minimise soft constraint violation. General purpose heuristics for combinatorial problems, exploiting features coming from constraint programming, are presented in [9]. In this work, a tabu search framework is employed as a general purpose approximate solver for problems formulated in a constraint satisfaction setting; a specialisation of the solver for the efficient solution of scheduling problems is also presented. Another approach is that used in local branching [3], where some constraints are added to the initial formulation in order to drive the search of the solution to more promising areas.

In this contribution, our main aim is to provide an approach which is usable for any problem that can be formulated as a mathematical program—ultimately, an MPS file. Hence in this approach the formulation given through a mathematical program is accepted as is, without any modification required. Thus, it can be used as a direct replacement for branch-and-bound when its solution time is not affordable.

1.3 Overview of the paper

In section 2 we present a variant of the relax-and-fix heuristic for MIP, which will be used for solution initialisation and improvement on the subsequent sections.

Section 3 provides a description of the operators required for the evolutionary solver. The initialisation operator fixes the initial values of the integer variables for all the solutions of the population, using the relax-and-fix variant. The strategy used for the evaluation and comparison of solutions is also presented in this section, as well as the reproduction operators, which allow creating new solutions from existing ones.

We then present evolutionary algorithms that make a population of solutions evolve; we provide a simple one, as well as an improved version, in section 4.

We have tested the EA with a subset of the benchmark instances that are available in the *MIPLIB* [1]. We have focused on instances which could not be solved to optimality in one hour of CPU time by branch-and-bound, using the publicly available LP/MIP solver provided in the GLPK [8] software. The results are presented in section 5.

We finish with conclusions and some perspectives on future research.

2 Background algorithms

The mathematical programming formulation of a mixed integer linear program is

$$z = \min_{x,y}\{cx + hy : Ax + Gy \geq b, x \in \mathbb{Z}^n_+, y \in \mathbb{R}^p_+\} \tag{1}$$

where $\mathbb{Z}^n_+$ is the set of nonnegative, integral n-dimensional vectors and $\mathbb{R}^p_+$ is the set of nonnegative, real p-dimensional vectors. A and G are $m \times n$ and $m \times p$ matrices, respectively, where m is the number of constraints. The integer variables are x, and the continuous variables are y. In this paper we assume that there are additional bound restrictions on the integer variables: $l_i \leq x_i \leq u_i$, for $i = 1, \ldots, n$.

2.1 Relax-and-fix

The relax-and-fix heuristic was originally proposed for the lot-sizing problem in [14]. For lot-sizing, the initial problem is divided in periods, each period being treated independently: variables of the current period are determined by B&B (or other more sophisticated approach), variables of the preceding periods are fixed at values found on previous B&B solutions, and integrity of variables corresponding to the subsequent periods is relaxed.

In this work, we extend this idea to a more general case, where there are several stages (playing the role of periods in lot-sizing). In each stage some selected variables are determined by B&B. As in the original relax-and-fix, integer variables concerning previous stages are fixed, and the remaining integer variables are relaxed, as in equation (2).

$$z = \min_{x}\{cx + hy : Ax + Gy \geq b, x_i = \bar{x}_i \; \forall i \in \mathcal{F}, x_i \in \mathbb{Z} \; \forall i \in \mathcal{I}, x_i \in \mathbb{R} \; \forall i \in \mathcal{R}\} \qquad (2)$$

In this equation the set of indices of fixed variables, determined on previous stages, is $\mathcal{F}$; the set of variables currently being determined by B&B is $\mathcal{I}$; the set of the remaining variables, whose integrality is relaxed, is $\mathcal{R}$. The base procedure used for solution construction is depicted in Algorithm 1.

Algorithm 1: Extended relax-and-fix heuristic.
RelaxAndFix()
(1) $\mathcal{R} = \{1, \ldots, n\}$
(2) $\mathcal{F} = \{\}$
(3) **while** $\mathcal{R} \neq \{\}$
(4) fix $x_i = \bar{x}_i, \forall\, i \in \mathcal{F}$, at previously determined values
(5) select a set of variables $\mathcal{I} \subseteq \mathcal{R}$ to determine in this stage
(6) set $x_i, \forall\, i \in \mathcal{I}$, as integer variables
(7) $\mathcal{R} = \mathcal{R} \backslash \mathcal{I}$
(8) relax $x_i, \forall\, i \in \mathcal{R}$ as continuous variables
(9) solve equation (2), determining $\bar{x}_i$, $\forall i \in \mathcal{I}$
(10) $\mathcal{F} = \mathcal{F} \cup \mathcal{I}$
(11) **return** $\bar{x}$

One can consider a variant of this algorithm where the set $\mathcal{I}$ is chosen randomly on each stage. This provides a complete, simple construction method, which is formalised in the next section.

2.1.1 Number of variables to fix

One of the main questions that has to be addressed in order to use the relax-and-fix heuristic when there is not a natural subdivision of the variables, concerns the number K of variables that are to be fixed in each stage (or, in other words, the cardinality of the set $\mathcal{I}$ in Algorithm 1).

Some preliminary tests have shown that the time required for obtaining a complete solution with relax-and-fix is not monotonic on the number of variables fixed per stage: if this number is too small or too large, the construction of a solution requires more time than if an intermediate number is appropriately chosen. For a particular problem, it is easy to choose a good value for the number of variables to fix per stage. However, the optimal value for this parameter is rather problem-dependent.

Preliminary data also shows that, as expected, the quality of the solution generally improves when the number of variables fixed per stage increases. However, if this number is too large, the improvement costs too much in terms of CPU usage.

We have left the number of variables to fix per stage, K, as a parameter of Algorithm 2, described in the next section, which can be used for solution construction by setting the argument $\mathcal{F} = \{\}$ (notice that the parameter $\bar{x}$ is not used for solution construction).

2.1.2 Solution completion

The relax-and-fix construction mechanism can be used in a different context: that of completing a solution that has been partially destructed. For this purpose, all that is required is to send an incomplete solution $\bar{x}$ as a parameter to the algorithm, as well as the set $\mathcal{F}$ of indices that are fixed. All the other variables are either made integer, if they are selected during the construction of the set $\mathcal{I}$, or otherwise relaxed. These ideas are described in Algorithm 2, where parameter K determines the number of variables that are to be made integer on each stage of relax-and-fix. The actual values of the sets $\mathcal{I}$ and $\mathcal{R}$ are constructed on steps (5) to (10). On step (13) the sets $\mathcal{F}, \mathcal{I}$ and $\mathcal{R}$, built up on this algorithm, are used for the construction of a MIP subproblem. Then, B&B is used for solving equation (2), and the resulting solution is stored in the indices $\mathcal{F}$ of variable $\bar{x}$. At the end of this process the set $\mathcal{R}$ is empty, and all the integer variables of the initial problem are fixed.

Algorithm 2: Randomised solution construction or completion by relax-and-fix. For solution construction, $\mathcal{F} = \{\}$ and $\bar{x}$ is not used. For solution completion, $\mathcal{F}$ holds the set of indices of variables that are fixed, and $\bar{x}$ their corresponding values.

RandomRelaxAndFix($\bar{x}, \mathcal{F}, K$)

(1) $\mathcal{R} = \{1, \ldots, n\} \backslash \mathcal{F}$
(2) **while** $\mathcal{R} \neq \{\}$
(3) $\quad \mathcal{I} = \{\}$
(4) $\quad$ fix $x_i = \bar{x}_i, \forall\, i \in \mathcal{F}$, at previously determined values
(5) $\quad$ **for** k=1 **to** K
(6) $\quad\quad$ randomly select $i \in \mathcal{R}$
(7) $\quad\quad \mathcal{I} = \mathcal{I} \cup \{i\}$
(8) $\quad\quad \mathcal{R} = \mathcal{R} \backslash \{i\}$
(9) $\quad\quad$ **if** $\mathcal{R} = \{\}$
(10) $\quad\quad\quad$ **break**
(11) $\quad$ set $x_i, \forall\, i \in \mathcal{I}$, as integer variables
(12) $\quad$ relax $x_i, \forall\, i \in \mathcal{R}$ as continuous variables
(13) $\quad$ solve equation (2), determining $\bar{x}_i$, $\forall i \in \mathcal{I}$
(14) $\quad \mathcal{F} = \mathcal{F} \cup \mathcal{I}$
(15) **return** $\bar{x}$

3 The evolutionary operators

Evolutionary algorithms function by maintaining a set of solutions, generally called a *population*, and making these solutions evolve through operations that mimic the natural evolution: reproduction, and selection of the fittest. These operators were customised for the concrete type of problems that we are dealing with; we focus on each of them in the following sections.

3.1 Initialisation and representation of the solutions

The population that is used at the beginning of an evolutionary process is usually determined randomly, in such a way that the initial diversity is very large. In the case of MIP, it is appealing to bias the initial solutions, so that they are distributed in regions of the search space that are likely to be more interesting. A way to provide this bias is to use the random relax-and-fix heuristic of Algorithm 1 as the initialisation operator.

The part of the solution that is important to keep in the EA is the subset of integer variables, x in equation (1); a particular solution kept in the EA is represented by an n-dimensional vector of integers $\bar{x} = (\bar{x}_1, \ldots, \bar{x}_n)$.

Some integer variables are fixed by the EA, leading to a MIP subproblem with the remaining integer variables, and the continuous variables y, free; these are expected to be determined afterwards, by the relax-and-fix heuristic for solution completion presented in Algorithm 2.

3.2 Evaluation of solutions

The solutions that are kept by the algorithm—or, in other words, the individuals that compose the population—may be feasible or not. For the algorithm to function appropriately it has to be able to deal with both feasible and infeasible solutions coexisting in the population. We will thus assign to each solution a value z corresponding to its objective, and another value ζ corresponding to a measure of its infeasibility (which is zero if the solution is feasible).

Solution evaluation is done throughout the process of relax-and-fix. If all the variables could be fixed, then $\zeta = 0$ and the evaluation is done through the objective function defined in equation (3). This corresponds to the last problem solved by the relax-and-fix procedure, in line (13) of Algorithm 2.

$$z = \max_{y} \{c\bar{x} + hy : Gy \leq b - A\bar{x}, y \in \mathbb{R}^p_+\} \tag{3}$$

It might also happen that during the relax-and-fix process the problem becomes infeasible, and thus not all the integer variables could be fixed. In this case, ζ is made equal to the number of variables that were not yet fixed when the subproblem became infeasible (i.e., it is made equal to the sum of the cardinalities of sets $\mathcal{I}$ and $\mathcal{R}$, on line (13) of Algorithm 2).

3.2.1 Comparison and selection of solutions

For the selection of solutions for reproduction, we propose to rank the solutions according to two criteria: the first criterion is the infeasibility value ζ, and the second is the objective value z. By doing so, feasible solutions are always ranked better than infeasible ones; feasible solutions are ranked according to the objective of the MIP problem, and infeasible solutions are ranked according to their infeasibility measure.

Therefore, for minimisation problems, we say that a solution structure i *is better* than another structure j if $\zeta^i < \zeta^j$ (i is closer to a complete solution than j), or $\zeta^i = \zeta^j$ and $z^i < z^j$ (completion of the solutions is identical, and i has a better objective).

The selection is based on each solution's ranking in the population, which can be determined through the comparison operator defined above (see also section 4.1.1).

3.3 Solution reproduction

3.3.1 Genetic operators

The generation of a new solution from two parent solutions is composed of three steps: recombination, mutation, and local search. The detailed process of reproduction for creating a new genome $\bar{x}$ from two parents $\bar{x}^1$ and $\bar{x}^2$ is presented in Algorithm 3, where a continuous random variable with uniform distribution on $[0, 1]$ is denoted by $\mathscr{R}$, and a discrete random variable with uniform distribution on (a, b) is denoted by $\mathscr{U}(a, b)$. In a glance, what recombination does is to alternately pick parts of the vectors from each of the parents (lines (1) to (5) of Algorithm 3). Mutation adds a random perturbation in an index i of the solution thus obtained; the perturbation consists of assigning to the corresponding variable a random value, drawn with uniform distribution from its lower bound l_i to its upper bound u_i (lines (6) and (7)). Neither recombination nor mutation have parameters.

Local search tries to improve the newly created solution's quality by hill climbing in its neighbourhood, as described below.

Algorithm 3: Generation of a new solution.
GENERATE($\bar{x}^1$, $\bar{x}^2$, K)
(1) **for** i=1 **to** n
(2) **if** $\mathscr{R} < 1/2$
(3) $\bar{x}_i = \bar{x}^1_i$
(4) **else**
(5) $\bar{x}_i = \bar{x}^2_i$
(6) $i = \mathscr{U}(1, n)$
(7) $\bar{x}_i = \mathscr{U}(l_i, u_i)$
(8) **return** LOCALSEARCH($\bar{x}, K$)

3.3.2 Local search

To complement the genetic operators we propose a local search method, for hill climbing in the integer variables space. It tries to find better values for a subset of variables, keeping all the others fixed at their current values. These steps are described in Algorithm 4, which takes as parameter a solution $\bar{x}$. In lines (4) to (6), indices are randomly chosen from the set of fixed variables $\mathcal{F}$ (which initially contains all the indices), and included in the set of integer ones $\mathcal{I}$ (initially empty).

Hence, local search operates by instantiating the sets $\mathcal{F}$ and $\mathcal{I}$. For the current instantiation, we check if the linear relaxation of the problem defined in equation (2) (where variables with indices in $\mathcal{F}$ are fixed, and all the other relaxed) is feasible. If not, additional variables are released, until the relaxation is feasible.

On the solution of the next MIP subproblem (in the function call of line (7)) variables with indices in $\mathcal{I}$ will be treated as integer variables; for indices in $\mathcal{F}$, variables will be kept fixed at their values in $\bar{x}$. Thus, $\mathcal{F}$ and $\mathcal{I}$ are used for creating a MIP subproblem, and the solution is tentatively completed through relax-and-fix, by means of Algorithm 2. The solution obtained (which may be feasible or not) is returned.

The minimum number of variables to be released in this algorithm is controlled by the parameter K, which also sets the number of simultaneous variables to fix on each step of relax-and-fix.

Algorithm 4: The local search procedure.
LocalSearch($\bar{x}, K$)
(1) $\mathcal{F} = \{1, \ldots, n\}$
(2) $\mathcal{I} = \{\}$
(3) **while** card($\mathcal{I}$) $< K$ or linear relaxation of eq. (2) infeasible
(4) randomly select $i \in \mathcal{F}$
(5) $\mathcal{I} = \mathcal{I} \cup \{i\}$
(6) $\mathcal{F} = \mathcal{F} \backslash \{i\}$
(7) **return** RandomRelaxAndFix($\bar{x}, \mathcal{F}, K$)

3.4 LP/MIP solver-dependent features

The operators defined in the preceding sections are all virtually independent of the actual solver used for the LPs and MIPs. However, the optimal number of variables to fix per stage in the relax-and-fix procedure is likely to be dependent of the solver (as well as of the instance being solved).

In many cases, the subproblems passed to the branch-and-bound solver can be rather difficult, since fixing some variables at particular values may create a strange structure for the remaining MIP. As it is not essential to solve these subproblems to optimality, we propose to limit this search; in our implementation, we set a limit to the number of simplex iterations to a predefined value S, which is respected on every solution of equation (2) when it is being used by the genetic operators described in this section.

As when some variables are fixed the LP solution remains dual-feasible, the dual simplex method is likely to be the most appropriate solver to use on LP relaxations.

4 The evolutionary algorithm

The operators described in the previous section can be used in a broad range of evolutionary algorithms. We start this section with a simple algorithm, and then complement it with some improvements.

4.1 A simple algorithm

A simple algorithm, which drives the population operations making use of the solution representation, genetic operators and local search described in the preceding section, is described in Algorithm 5.

The parameters of this algorithm are the number P of solutions to keep in the population, the total CPU time T allowed to the search, the maximum number of simplex iterations for the solution of subproblems in the form of equation (2) (S, described in section 3.4, which is hidden in the algorithms, as it is not required for their explanation) and the number K of simultaneous variables on each relax-and-fix stage. Solutions are initialised in line (2) through the relax-and-fix heuristic defined in section 2.1, and kept in an array of solutions p. The best element of the population (which, after sorting, is p_1) is returned in line (8).

Algorithm 5: A simple evolutionary algorithm for MIP solution.

SIMPLEEA(P, T, K)
(1) **for** $i = 1$ **to** P
(2) $\bar{x}$ =RANDOMRELAXANDFIX$(\bar{x}, \{\}, K)$
(3) $p_i = \bar{x}$
(4) SORT(p)
(5) **while** CPU() $< T$
(6) $p =$ REPRODUCEELITIST(p, K)
(7) SORT(p)
(8) **return** p_1

Algorithm 6: The main reproduction scheme. Elements of the population p are sorted, p_1 being the best element. (For elitist reproduction, p_1 is not changed; for non-elitist reproduction, line (1) is suppressed, and the cycle on line (2) starts for $i = 1$.)

REPRODUCEELITIST(p, K)
(1) $p'_1 = p_1$
(2) **for** $i = 2$ **to** P
(3) e_1 =SELECT(p)
(4) e_2 =SELECT(p)
(5) $\bar{x}$ =GENERATE(e_1, e_2, K)
(6) $p'_i = \bar{x}$
(7) **return** p'

4.1.1 Selection: rank-based fitness

As explained in section 3.2, the solution process is divided into two goals: obtaining feasibility and optimisation. This has motivated the implementation of an order-based scheme, *rank-fitness*, that evaluates solutions on the basis of their ranking, according to the comparison operator defined in section 3.2.1.

One generally wants to normalise the fitnesses, so that their sum for all the elements of the population equals one, and fitnesses can be thought of as probabilities of selection. In this case, the scaled fitness f_i to attribute to the element ranked i (for $i = 1, \ldots, n$) can be determined as

$$f_i = \frac{n - i + 1}{n + (n-1) + \ldots + 1} = \frac{n - i + 1}{n(n+1)/2}$$

The selection of solutions for reproducing, with this probability, is then performed through roulette wheel selection. (See for example [5] for a detailed description of roulette wheel selection.)

4.2 An improved evolutionary algorithm: niche search

In this section we propose some improvements on the evolutionary algorithm, which aim at making a better usage of the elements kept in the population. The main idea, borrowed from [10], is to divide the population, keeping some groups (*niches*) isolated from the others in terms of reproduction, with a simple migration scheme. The claim is that this way, as the global evolutionary search pursues, more localised searches are done inside each of the niches. The algorithm is therefore expected to keep a good compromise between intensification of the

search (inside each niche) and diversification of the population (as there are several niches running simultaneously, each of them possibly searching in disparate regions of the search space). This method has some similarities with that described in [12], where competing subpopulations play a role comparable to that of the niches.

The parameters that must be set by the user for a run of niche search are: the number of niches N, the number of elements on each niche P, the number of simultaneous variables to be used on each stage of relax-and-fix (K), the maximum number of simplex iterations allowed to solve equation (2) (S, again hidden in the algorithms), and the stopping criterion (the allowed CPU time T).

Algorithm 7: An improved evolutionary algorithm for MIP solution. Niches are sorted by the fitness of their best element. The population is kept in an array of arrays p, where p_{ij} is the jth element in niche i. When every p_i, and p, are sorted, p_{11} is the best element in the population.

NICHE(N, P, T, K)
(1) **for** i=1 **to** N
(2) **for** j=1 **to** P
(3) $\bar{x}$ =RANDOMRELAXANDFIX$(\bar{x}, \{\}, K)$
(4) $p_{ij} = \bar{x}$
(5) SORT(p_i)
(6) SORT(p)
(7) **while** CPU() $< T$
(8) **for** i=1 **to** N
(9) **if** $i = 1$ or $p_{i1} \neq p_{i-1,1}$
(10) p_i = REPRODUCEELITIST(p_i, K)
(11) **else**
(12) p_i = REPRODUCENONELITIST(p_i, K)
(13) SORT(p_i)
(14) SORT(p)
(15) **if** p_{11} is not present on other niches
(16) $p_{MN} = p_{11}$
(17) SORT p_M
(18) SORT p
(19) **return** p_{11}

4.2.1 Migration and elitism

Migration of elements between niches is based on a very simple scheme: the best element of the population is copied from the first niche into the last niche (lines (15) and (16) in Algorithm 7), but only in case it does not exist on any other niche. This allows propagation of the best known solution, but avoids over-propagation.

Elitism determines whether the best solution found so far by the algorithm is kept in the population or not. Elitism generally intensifies the search in the region of the best solution. As mentioned before, niche search keeps several groups, or niches, evolving with some independence. Each of these groups may be elitist (keeping *its* best element in its population) or not.

Our objectives are twofold: we want the search to be as deep as possible around good regions, but do not want to neglect other possible regions. For this purpose, niches whose best solution is different of the best solution of other niches are elitist, but when several niches have an identical best solution, only one of them is elitist. This also provides a good interconnection of elitism and migration. With this strategy we hope to have an intensified search in regions with good solutions, and at the same time enforce a good degree of diversification.

5 Numerical results

5.1 Benchmark instances

The instances of MIP problems used as benchmarks are defined in the MIPLIB [1] (see Table 8 on appendix B). The evolutionary system starts by reading an MPS file, and stores the information contained there (the matrices A and G, and the vectors b, c and h in equation (1)) into an internal representation. The number of variables and constraints, their type and bounds, and all the matrix information are, hence, determined at runtime.

The computational environment used is described in appendix A, and the set of benchmark instances and the statistical measures used to report solutions in appendix B. We have selected instances which could not be solved to proven optimality in 3600 seconds of CPU time in our computer by B&B, using the publicly available LP/MIP solver of the GLPK [8] software. These instances come from a wide range of applications; the degree of difficulty of the solution of the LP relaxation also varies widely. Hence, for some of the instances the 3600 seconds allowed evolution for many iterations, whereas for others this time only allowed the initialisation of the population.

5.2 Branch-and-bound

We have used the GLPK implementation, version 4.9, as a basis for comparing our algorithm to branch-and-bound. This software comprises a simplex-based solver for linear programs and an implementation of the branch-and-bound algorithm. GLPK uses a heuristic by Driebeck and Tomlin to choose a variable for branching, and the best projection heuristic for backtracking (see [8] for further details). This was also the software for LP and MIP solution used by relax-and-fix, on the EA.

The results obtained by B&B on the series of benchmark instances selected are provided in Table 1. The time allowed to the search for each instance is 3600 seconds of CPU, and the best solution found within that limit is reported[1].

5.3 Evolutionary algorithms

The simple evolutionary algorithm was parameterised with 12 elements, and the improved algorithm with 4 niches, each with 3 solutions (hence both algorithms kept a population of 12 solutions). Results are reported in tables 2 and 3 for the simple algorithm, and in tables 4 and 5 for the improved one. Both algorithms were allowed to use 3600 seconds of CPU time. Although fine tuning of the parameters could possibly lead to better results, we have made no attempt to do so, and used the same values for all the instances. The number

[1] For one of the instances, `pk1`, this solution is an optimum; this instance was included in the benchmark set because it provides an example where the EAs are considerably worse than B&B.

Instance name	Optimal z	Branch-and-bound best z	% from opt.
air04	56138	56152	0.0249
air05	26374	26415	0.155
bell4	18541484.20	18560472.42	0.102
gt2	21166	21962	3.76
mod011	-54558535	-54107967.58	0.826
modglob	20740508	20815372.17	0.361
noswot	-43	-41	4.65
p6000	-2451377	-2250219	8.21
pk1	11	11*	0
pp08a	7350	7370	0.272
qiu	-132.873137	-132.8731369	7e-08
set1ch	54537.7	60426.250	10.8

Table 1: Optimal solution for each of the benchmark instances, and solutions obtained by branch-and-bound, using GLPK, with the CPU time limited to 3600 seconds. Branch-and-bound did not finish for any of these instances (* indicates that the solution found is an optimum).

of simultaneous variables to fix on each stage of relax-and-fix is $K = 10$, and the limit of simplex iterations allowed for the solution of each subproblem is $S = 10000$.

Instance name	Best solution		Worst solution		Average solution	
	z	%above optim.	z	%above optim.	z	%above optim.
air04	56138.00	0	57159	1.82	56355.44	0.39
air05	26402	0.11	26723	1.32	26517.60	0.54
bell4	18541484.198	0	18541484.20	0	18541484.20	0
gt2	21166	0	21166	0	21166	0
mod011	-54422236.85	0.25	-53556314.31	1.83	-53931032.19	1.15
modglob	20740508.086	0	20740508.09	0	20740508.09	0
noswot	-41	4.65	-41	4.65	-41	4.65
p6000	-2451346	0.0013	-2451128	0.010	-2451208.24	0.0069
pk1	11	0	20	82.82	17	54.
pp08a	7350	0	7500	2.04	7408.40	0.79
qiu	-132.873137	0	-132.873137	0	-132.87	0
set1ch	54718.250	0.33	55586.50	1.92	55772.56	2.26

Table 2: Results obtained for 25 independent runs of the simple EA with a population of 12 elements, allowed to evolve until the CPU time spent reached 3600 seconds.

Instance name	%feas run	Feasibility ($E[t^f]$(s))	%best runs	Best sol. ($E[t^f]$(s))	%opt runs	Optimality ($E[t^f]$(s))
air04	100	735	12	28017	12	28017
air05	100	454	4	86909	0	$\gg$90000
bell4	100	0.35	100	433	100	433
gt2	100	0.26	100	276	100	276
mod011	100	67	4	86776	0	$\gg$90000
modglob	100	0.77	100	34	100	34
noswot	100	11	100	184	0	$\gg$90000
p6000	100	396	4	88399	0	$\gg$90000
pk1	100	0.27	4	87699	4	87699
pp08a	100	0.66	32	8159	32	8159
qiu	100	7.9	100	289	100	289
set1ch	100	17	4	89913	0	$\gg$90000

Table 3: Results obtained for the simple EA (cont.): percent of feasible, best, and optimal runs, and expectations of the CPU time required for reaching feasibility, the best solution found, and an optimal solution.

The comparison of the results of the simple algorithm to those of the improved algorithm,

Instance name	Best solution z	Best solution %above optim.	Worst solution z	Worst solution %above optim.	Average solution z	Average solution %above optim.
air04	56138.00	0	57159	1.82	56357.12	0.39
air05	26402.00	0.11	26723	1.32	26517.60	0.54
bell4	18541484.20	0	18541484.20	0	18541484.20	0
gt2	21166	0	21166	0	21166	0
mod011	-54558535.01	0	-53681175.47	1.61	-54056540.51	0.92
modglob	20740508.09	0	20740508.09	0	20740508.09	0
noswot	-41.00	4.65	-41	4.65	-41.00	4.65
p6000	-2451346.00	0.00127	-2451128	0.0102	-2451208.24	0.0069
pk1	14	27.3	19	72.7	16.56	50.5
pp08a	7350	0	7500	2.04	7381.60	0.430
qiu	-132.873137	0	-132.873137	0	-132.873137	0
set1ch	54632.50	0.17	55632.25	2.01	54895.83	0.657

Table 4: Results obtained for 25 independent runs of the improved EA, with 4 niches, each with 3 elements, allowed to evolve until the CPU time spent reached 3600 seconds.

Instance name	%feas runs	Feasibility ($E[t^f]$(s))	%best runs	Best sol. ($E[t^f]$(s))	%opt runs	Optimality ($E[t^f]$(s))
air04	100	729	12	28013	12	28014
air05	100	426	4	86862	0	$\gg$90000
bell4	100	0.35	100	395	100	395
gt2	100	0.24	100	148	100	148
mod011	100	63	8	44304	8	44304
modglob	100	0.77	100	34	100	34
noswot	100	11	100	121	0	$\gg$90000
p6000	100	396	4	88469	0	$\gg$90000
pk1	100	0.249	4	86408	0	$\gg$90000
pp08a	100	0.602	56	3687	56	3687
qiu	100	7.6	100	154	100	154
set1ch	100	16	4	89851	0	$\gg$90000

Table 5: Results obtained for the improved EA (cont.): percent of feasible, best, and optimal runs, and expectations of the CPU time required for reaching feasibility, the best solution found, and an optimal solution.

with separation in niches, provide evidence of the superiority of the improved version, both in the quality of the solution achieved and in the time required for reaching it. This gives an indication of the importance of making a good usage of the elements kept in the population. The improvements are virtually costless in terms of CPU, and provide some benefits in terms of the results obtained.

From this section on, we will focus on a more profound analysis of the improved version of the EA.

5.3.1 Analysis of the results

A summary of the results obtained is provided in Table 6, where the best solution found during the time-limited branch-and-bound is compared to the average of the solutions of the 25 runs of the improved EA. This shows that in average, for many of the instances, the EA finds better solutions in 3600 seconds than B&B. For most of the cases where B&B does better, the EA was not allowed to generate a meaningful number of solutions within the time limit. For these cases, the results provide an indication of the good quality of the initialisation operators presented in section 3.1.

For having a better insight of the performance of each algorithm, we have plotted, in Figure 1, a log of the objective value of the best solution found as a function of the CPU

	B&B	Improved EA			
Instance	solution z	average z	$E[t^o]$	# sol. generated	# different opt.sols.
air04	56152	56357.12	28014	0	3/3
air05	26415	26517.60	$\gg$90000	0	n.a.
bell4	18560472.42	18541484.20	395	267	2/25
gt2	21962	21166	148	1720	25/25
mod011	-54107967.58	-54056540.51	44304	902	1/2
modglob	20815372.17	20740508.09	34	567	1/25
noswot	-41	-41.00	$\gg$90000	17560	n.a.
p6000	-2250219	-2451208.24	$\gg$90000	1	n.a.
pk1	11	16.56	$\gg$90000	256152	n.a.
pp08a	7370	7381.60	3687	33757	1/14
qiu	-132.8731369	-132.873137	154	164	15/15
set1ch	60426.250	54895.83	$\gg$90000	996	n.a.

Table 6: Summary of the results obtained on 3600 seconds of CPU time. Objective value z of the solution of branch-and-bound. Results for 25 runs of the improved EA: average z, expected CPU time for the EA to reach optimal solutions, average number of solutions generated per run of the EA. Last column reports the number of different optimal solutions found (left) and the number of times an optimal solution was found by the EA (right).

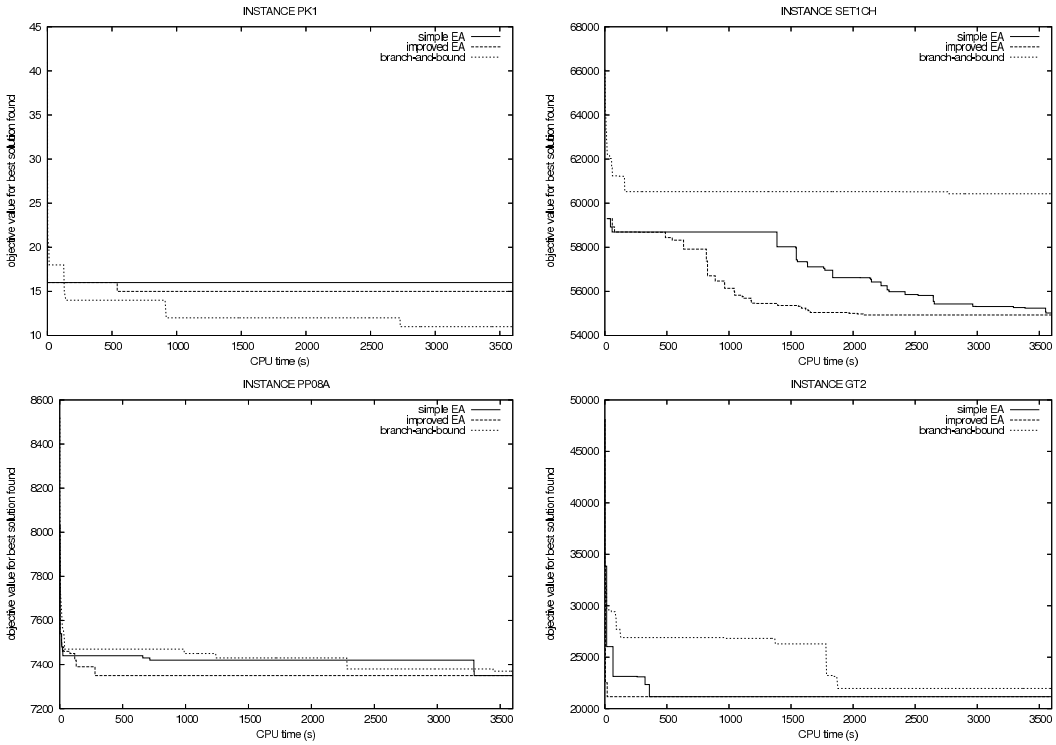

Figure 1: Typical log of the evolution of the best solution found, by branch-and-bound and by each of the EAs, with the CPU time.

time. These logs are plotted for a selection of four instances: gt2 and set1ch (where the EAs are better), pp08a (where the algorithms are approximately equivalent), and pk1 (where B&B is better). Feasible solutions were found very early by all the algorithms. Except for pk1, branch-and-bound tends to take longer to find good solutions that the EAs, and in general, for a given time, the improved EA has a better solution than the other algorithms'.

For the pk1 instance, the EAs are very deeply trapped on local optima, and the evolutionary mechanisms are not sufficient to escape them. On all the other instances, any of the EAs has a better behaviour, with much shorter platforms than B&B.

The pk1 case is somewhat intriguing, as even though the limit time is enough for the generation of many solutions the EA fails to produce good ones. For explaining this behaviour, we provide another set of plots in Figure 2. These were obtained by selecting a typical run of the EA, and recording all the intermediate solutions x^k, obtained after recombination, mutation and local search, until reaching the final solution x^* (this is a global optimum for problems pp08a and gt2, and suboptimal for set1ch and pk1). We then compared the intermediate solutions to the final one, and counted, for each of them, the number of elements x_i^k which were different of the corresponding value in the final solution, x_i^*; then, we set $\delta_i^k = 1$ if $x_i^k = x_i^*$, and $\delta_i^k = 0$ otherwise. This allowed us to calculate, for each solution k, a measure of its distance from the final one: $d_k = \sum_i \delta_i^k$. Finally, for each value of this distance we counted the number of different intermediate solutions that were obtained at that distance from the final one, as well as the average value of their objective, and plotted these two measures in terms of the distance.

Figure 2 indicates that for problems where the EA found good solutions (gt2, set1ch, pp08a), the number of distinct solutions found at a short distance from the final/optimal one is large, and tends to decrease when the distance increases. For these "easy" instances for the EAs, the average value of the objective is very good when the distance is short, and tends to become worse when the distance increases; there is a kind of gradient to the best found solution. On the opposite side, for instance pk1 (where the EA failed severely), the number of solutions different from the best solution found seems to have a peak at relatively large distances, and the average objective function is approximately flat. Likely, there is little hope for finding a smooth path to the final solution, which might have been found fortuitously. (An additional information that the plot for instance pk1 gives concerns the number of distinct solutions found with an odd number of different elements. These values are small, suggesting that for this instance perturbations on the solutions should probably be done on pairs of elements.)

Another indicator of the hardness of an instance to the EA might be given by the number of different optimal solutions found, presented in Table 6. Even though we can draw no conclusions for problems that were not solved to optimality, we observed that for some easy problems the number of different optima found is large. It might be the case that B&B is trapped by degeneration; on the contrary, degeneration gives the EA many possibilities for finding a good solution.

For this reduced set of instances, we have also compared the behaviour of B&B to that of the improved EA in a longer run, allowing 24 hours of CPU time. These results are presented in Table 7. Even though they are not statistically satisfactory for the EA (as they are based on a single run), they consolidate the idea that platforms on the landscape of the best solution found by B&B might be very long, and that providing more time to the EA allows it to find better solutions[2]. Instance pk1 remains a case where the EA did not work. For the pp08a and set1ch cases, the additional time allowed improvement of the solution found. For gt2, the optimal solution was found in less that one hour by the EA, and B&B completed the search in 17 hours.

In order to assess the importance of each of the operators used in the evolutionary system,

[2]For the pp08a instance, we have selected the initialisation of the random number generator that lead to the worst solution presented in table 4.

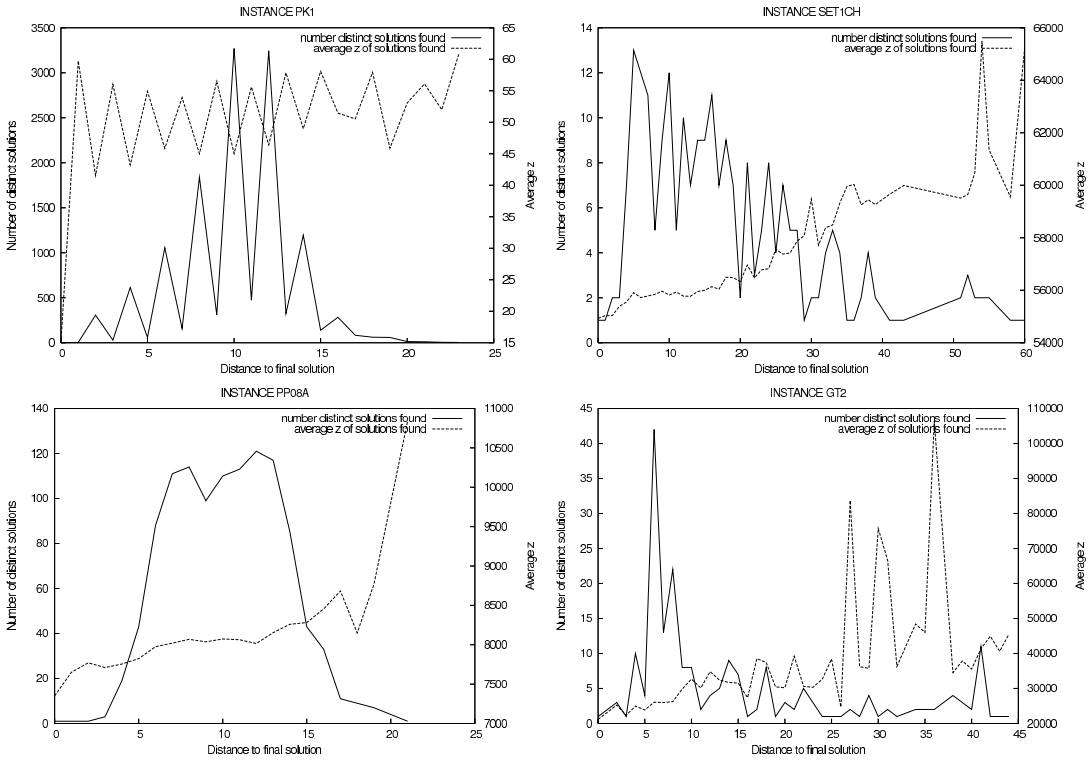

Figure 2: Plot of the number of different solutions obtained (left y axis) and quality of the solutions (right y axis), as functions of the number of indices where values are different of the final solution, for a typical run of the improved EA.

	Branch-and-bound		Improved EA	
Instance	1 hour	24 hours	1 hour	24 hours
gt2	21962	21166 (ended in 17h00m)	21166	21166
pk1	11	11 (ended in 4h43m)	15	15
pp08a	7370	7370	7480	7350 (after 4h27m)
set1ch	60426.250	59610.750	54930	54537.75

Table 7: Summary of the results obtained by branch-and-bound and by (a single run of) the improved EA, allowing 24 hours of CPU time.

we executed some experiments for assessing their efficiency. These experiments consisted of keeping track of which of the operators were responsible for improvements in the solutions, and of analysing the behaviour of the algorithm in their absence. They showed that the two genetic operators, the local search, and the initialisation procedure, were all necessary for a good performance of the algorithm. From these operators the less important one was the recombination, which very seldom found improving solutions, and whose removal led to a relatively small deterioration in the overall performance.

6 Conclusion

We present a system for solving MIPs based on evolutionary computation, which fixes most of the integer variables and evaluates them by solving smaller MIPs, in a process inspired in the relax-and-fix heuristic.

A strategy of dividing the population into several niches improved considerably the performance of the evolutionary algorithm. This is related to the improvement on diversification when there are several niches, which decreases the probability of being trapped in local optima. To this end, a specialised elitist strategy plays an important role, by avoiding keeping good but identical solutions for a long time in the population.

The results obtained with this evolutionary system for some standard benchmark instances were compared to those obtained by B&B. The performance of the evolutionary algorithm is promising, as it generally obtains better solutions than B&B in a limited CPU time, typically with much smaller memory requirements.

For difficult instances, the landscape of the B&B best found solution tends to have large plateaux after the initial search phase, and the solution is likely to remain unchanged for a long time. The plateaux for search with the EA are generally shorter. This indicates that the evolutionary solver might be the right choice when there are constraints either on the time allowed or on the size of memory available, as happens in many real-world, practical situations.

The algorithm proposed does not take into account any particular structure of the instances, and may be used without any modification for any problem which can be formulated as a mathematical program.

There are two interesting future research directions from this work. The first is to analyse if the evolutionary operators defined here are still useful in case the system that solves the MIP subproblems is more powerful than B&B—for example, if one uses branch-and-cut, or a solving black box provided by commercial solvers. The other is to study a heuristic to select an order for fixing the variables in the subproblems solved by relax-and-fix, as this order has a very strong impact on the performance of the EA.

A Computational Environment

The computer environment used in this experiment is the following: a machine with an Intel Pentium 4 at 1600MHz, with 256 KB of cache and 384 MB of RAM, with the Linux Debian operating system.

The LP/MIP solver, GLPK is implemented in the C programming language. The evolutionary algorithms proposed in this paper were implemented mainly in Python, with the CPU intensive parts and the interface to the MIP solver implemented in C. Profiling showed that the CPU time spent on the Python parts is negligible, and hence a direct comparison of these algorithms to GLPK in terms of CPU usage is acceptable.

B Benchmark instances

The instances of MIPs and IPs used as benchmarks are available in [1] and are summarised in Table 8. They provide an assortment of MIP structures, with instances coming from different applications. These are all instances for which branch-and-bound as implemented in GLPK, using default values for its parameters, does not complete the search in 3600 seconds of CPU time, in our computational environment.

Instance name	Application	Number of variables total	integer	binary	Number of constraints	Optimal solution
air04	airline crew scheduling	8904	8904	8904	823	56138
air05	airline crew scheduling	7195	7195	7195	426	26374
bell4	fiber optic net. design	117	64	34	105	18541484.20
gt2	truck routing	188	188	24	29	21166
mod011	unknown	10958	96	96	4480	-54558535
modglob	heating syst. design	422	98	98	291	20740508
noswot	unknown	128	100	75	182	-43
p6000	unknown	6000	6000	6000	2176	-2451377
pk1	unknown	86	55	55	45	11
pp08a	unknown	240	64	64	136	7350
qiu	fiber optic net. design	840	48	48	1192	-132.873137
set1ch	capacitated lot lizing	712	240	240	493	54537.7

Table 8: Set of the MIPLIB benchmark instances used: application, number of constraints, number of variables and optimal solutions.

C Statistics Used

In order to assess the empirical efficiency of the EAs, we provide measures of the expectation of the CPU time required for finding a feasible solution, the best solution found, and the optimal solution, for each of the selected MIP instances.

Let the number of independent runs observed for each benchmark be denoted by K, and let t_k^f be the CPU time required for obtaining a feasible solution in observation k, or the total CPU time in that iteration if no feasible solution was found. Let t_k^o and t_k^b be identical measures for reaching optimality, and the best solution found by the metaheuristic, respectively. Based on these K observations, the expected CPU times required for reaching feasibility, the best solution found in all the observations, and optimality, are respectively:

$$E[t^f] = \sum_{k=1}^{K} \frac{t_k^f}{r^f}, \quad E[t^b] = \sum_{k=1}^{K} \frac{t_k^b}{r^b}, \quad E[t^o] = \sum_{k=1}^{K} \frac{t_k^o}{r^o}.$$

In cases where r^f, r^b, or r^o are equal to 0, $\sum t_k^f$, $\sum t_k^b$, and $\sum t_k^o$ provide lower bounds for the respective expectations.

References

[1] Robert E. Bixby, Sebastiàn Ceria, Cassandra M. McZeal, and Martin W. P. Savelsbergh. An updated mixed integer programming library. Technical report, Rice University, 1998. TR98-03.

[2] Y. Davidor, H.-P. Schwefel, and R. Männer, editors. *Parallel Problem Solving from Nature - PPSN III*, volume 866 of *Lecture Notes in Computer Science*, Berlin, 1994. Springer.

[3] M. Fischetti and A. Lodi. Local branching. *Mathematical Programming*, 98:23–47, 2003.

[4] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, Boston, 1997.

[5] David E. Goldberg. *Genetic Algorithms in Search, Optimization & Machine Learning*. Addison-Wesley, 1989.

[6] R. C. Kelahan and J. L. Gaddy. Application of the adaptive random search to discrete and mixed integer optimization. *International Journal for Numerical Methods in Engineering*, 12:289–298, 1978.

[7] E. L. Lawler and D. E. Wood. Branch-and-bound methods: a survey. *Operations Research*, 14:699–719, 1966.

[8] Andrew Makhorin. *GLPK – GNU Linear Programming Kit*. Free Software Foundation, http://www.gnu.org, 2006. Version 4.11.

[9] Koji Nonobe. *Studies on General Purpose Heuristic Algorithms for Combinatorial Problems*. PhD thesis, Kyoto University, 2000.

[10] João P. Pedroso. Niche search: an application in vehicle routing. In *IEEE International Conference on Evolutionary Computation*, volume 1, pages 177–182, Anchorage, Alaska, 1998. IEEE.

[11] Günter Rudolph. An evolutionary algorithm for integer programming. In Davidor et al. [2], pages 139–148.

[12] Dirk Schlierkamp-Voosen and Heinz Mühlenbein. Strategy adaptation by competing subpopulations. In Davidor et al. [2], pages 199–208.

[13] Joachim P. Walser. Integer optimization by local search – a domain-independent approach. *Lecture Notes in Artificial Intelligence*, LNAI-1637, 1999.

[14] Laurence Wolsey. *Integer Programming*. John Wiley & Sons, 1998.