A Feasibility-preserving Crossover and Mutation Operator for Constrained Combinatorial Problems

Martin Lukasiewycz, Michael Glaß, and Jürgen Teich

Hardware-Software-Co-Design
Department of Computer Science 12
University of Erlangen-Nuremberg, Germany
{martin.lukasiewycz, glass, teich}@cs.fau.de

Abstract. This paper presents a feasibility-preserving crossover and mutation operator for evolutionary algorithms for constrained combinatorial problems. This novel operator is driven by an adapted Pseudo-Boolean solver that guarantees feasible offspring solutions. Hence, this allows the evolutionary algorithm to focus on the optimization of the objectives instead of searching for feasible solutions. Based on a proposed scalable testsuite, six specific testcases are introduced that allow a sound comparison of the feasibility-preserving operator to known methods. The experimental results show that the introduced approach is superior to common methods and competitive to a recent state-of-the-art decoding technique.

1 Introduction and Related Work

Definition 1. *A* constrained combinatorial problem *is defined as:*

$$\begin{aligned}&\textit{minimize } f(x)\\ &\textit{subject to } x \in X_f \textit{ with } X_f \subseteq \{0,1\}^n\end{aligned}$$

The objective function f allows multi-dimensional and non-linear calculations. The search space $X = \{0,1\}^n$ is restricted to binary values, but allows integer values by a binary encoding. The *feasible search space* $X_f \subseteq X$ is restricted by a set of linear constraints which are subsumed in the following matrix inequation:

$$Ax \leq b \tag{1}$$

with $A \in \mathbb{Z}^{m,n}$ and $b \in \mathbb{Z}^m$. Thus, the constraints have to be linear or linearizable[1]. Constraints that are not linearizable have to be handled by common methods like penalty functions. However, many problems have linear constraints only [1] or are dominated by the number of linear constraints [2]. In the following, this paper assumes that all constraints are linear.

In the field of *Evolutionary Computation* different constraint-handling techniques of constrained combinatorial problems as stated in Definition 1 exist. These constraint-handling methods become necessary since a variation by crossover and mutation operators tends to deliver infeasible solutions. A straightforward approach is a *penalty function* that counts the violated constraints and deteriorates the objective function, cf. [3,

[1] linearization substitution rule: $x_1 \cdot x_2 \leftrightarrow x_3$ with $x_1 - x_3 \geq 0 \wedge x_2 - x_3 \geq 0 \wedge x_1 + x_2 - x_3 < 2$

4]. Moreover, a greedy repair algorithm as presented in [5] can be applied to minimize the number of violated constraints. This repair algorithm delivers feasible solutions only for specific problems like, e.g., the *0/1 Knapsack Problem* [5], but it cannot guarantee feasibility for linear constraints as stated in Equation (1). Since this problem is known to be NP-complete in general [1], it cannot be solved by a greedy algorithm (assuming P$\neq$NP).

If the search space is hard constrained, the common constraint-handling methods fail, since they are more focused on the search for feasible solutions than optimizing the objectives. In [2], a decoding strategy [6] known as *SAT decoding* is presented that overcomes these drawbacks. By using a *Pseudo-Boolean (PB) solver* [7] as a decoder, this method always obtains feasible solutions by mapping from a bounded search space to feasible solutions. In combination with an *Evolutionary Algorithm* (EA), a good convergence towards the optimal solutions also on large and complex real-world problems [8] is reached.

Instead of using the PB solver for decoding feasible solutions, our novel approach uses the PB solver within the crossover and mutation operator. Thus, only feasible solutions are obtained already in the variation process of the EA and a decoding becomes unnecessary. To compare this novel approach to common constraint-handling techniques and the SAT decoding, we present a testsuite and six specific testcases. These testcases represent random as well as structured real-world problems.

The remainder of the paper is outlined as follows: Section 2 introduces the scalable testsuite and six specific testcases for the constrained combinatorial problem. In Section 3, the novel feasibility-preserving crossover and mutation operator is presented. Experimental results are given and discussed in Section 4 before the paper is concluded in Section 5.

2 Testsuite

In the following, a scalable testsuite for constrained combinatorial problems is presented. This testsuite is based on the well known *3-SAT* problem [1]. The 3-SAT problem is a special case of the *Satisfiability Problem* where each clause contains exactly three literals. In a nutshell, the 3-SAT problem is to determine if there exists a satisfying solution to a Boolean formula given as a conjunction of clauses. A clause is a disjunction of literals, which are variables or their negations. Each clause can be formulated as a linear greater-or-equal-1 constraint by substituting the *ORs* by plus signs and the negative literals $\overline{x_i}$ by $1 - x_i$. For instance, the clause $(x_1 \vee x_2 \vee \overline{x_3})$ is converted to $x_1 + x_2 + (1 - x_3) \geq 1$ or $x_1 + x_2 - x_3 \geq 0$, respectively. Thus, a transformation to the form in Equation (1) is straightforward.

The testsuite is configured by two parameters:

- n - number of binary variables
- k - number of constraints

In contrast to the original 3-SAT problem, the constructed constrained problem must contain at least one feasible solution. This is ensured by first constructing a random solution $x \in X$ with $\sum_{i=1}^{n} x_i = \frac{n}{2}$ or the same number of $1s$ and $0s$, respectively. Random constraints are generated by constructing clauses with randomly chosen positive (x_i) or negative literals ($\overline{x_i}$) of three randomly selected variables. This generated constraint is only added if it is satisfied by x to guarantee at least a single feasible solution. The procedure is carried out until the desired number of constraints k is reached.

Since this paper deals with constraint-handling, for all following testcases, the objective function f is a simple single linear objective function

$$f(x) = \sum_{i=1}^{n} r_i \cdot x_i$$

with the r_i values randomly distributed in $\mathbb{R}_{[0,1]}$.

T1 Testcase T1 is a random instance with $n = 100$ and $k = 400$. With $\frac{k}{n} \approx 4.3$, hard solvable 3-SAT instances are generated, cf. [9]. The experience shows, $\frac{k}{n} = 4$ tends to construct testcases that have enough feasible solutions for a meaningful optimization but, on the other hand, obtaining a feasible solution is not trivial.

T2 Testcase T2 is a random instance with $n = 150$ and $k = 600$. This testcase, in combination with T1, reveals the scaling of the optimization method for random instances.

T3 Testcase T3 represents a real-world problem with $n = 1000$ and $k = 4000$. For this testcase, a structure is induced by a special scheme to select the variables for the clauses: First, a random variable x_i is selected. The other two variables x_j,x_k are selected by a *Geometric Distribution* with the inverse function $F^{-1}(u) = \lceil ln(1-u)/ln(1-p) \rceil$, with u being a random number in $\mathbb{R}_{[0,1)}$, such that $j = (i + F^{-1}(u))\%n + 1$ and $k = (j + F^{-1}(u))\%n + 1$. A high value for p decreases the pairwise *distance* of the variables that is in average $\frac{1}{p}$. For testcase T3 the value p is set to 0.1. It is known that PB solvers are performing much better on structured problems than on random generated problems [10]. In fact, real-world problems are usually structured due to the problem trait, like, e.g. the system structure, or the construction scheme like, e.g, a hierarchical approach, cf. [9].

T4 Testcase T4 represents a real-world problem with $n = 1000$ and $k = 4100$. Unlike the construction scheme used in T3, a structure is induced by a partitioning approach. First, b constraints are randomly generated among all variables. Second, the variables are uniformly distributed in a partitions such that for each partition $\frac{k-b}{a}$ random constraints are generated. For testcase T4, the parameters are set to $a = 10$ and $b = 100$ and, thus, 10 instances as in testcase T1 are created and interconnected by 100 constraints. These additional 100 constraints further reduce the rate of feasible solutions.

T5 Testcase T5 is constructed the same way as T4 with $n = 2000$, $k = 8100$, $a = 20$ partitions, and $b = 100$. This testcase is used to illustrate the scaling of the optimization methods on structured problems.

T6 Testcase T6 is constructed the same way as T3 with $n = 3000$ and $k = 9000$. With $\frac{k}{n} = 3$ this testcase has a high rate of feasible solutions. This testcase shows the range of applicability of the optimization methods for low constrained problems.

3 Feasibility-preserving Crossover and Mutation

The requirement for a feasibility-preserving crossover and mutation operator for constrained combinatorial problems is to always obtain a feasible offspring solution from two feasible parent solutions. The proposed approach is based on a state-of-the-art *Pseudo-Boolean (PB) solver* [7]. In the following, a specialized crossover and mutation operator is presented that makes use of a PB solver to obtain feasible offspring solutions from two feasible parent solutions. Moreover, a heuristic that aims to improve the quality of the obtained offspring solution in terms of information preservation is presented.

3.1 PB Solver

The task of a PB solver is to find an $x \in X_f$ that satisfies a set of linear constraints as formulated in Equation (1). In fact, this NP-complete problem [1] is an ILP with binary variables and an empty objective function and can be solved by a common ILP solver. However, the specialized PB solvers tend to outrun common ILP solvers on these Boolean-natured problems [11]. These PB solvers are extended *SAT* solvers that are actually used to solve the *Satisfiability problem* and are based on a backtracking strategy. This strategy is known as the DPLL algorithm [12] and is outlined in Algorithm 1. The algorithm efficiently searches for a solution $x \in X_f$ that fulfills all given constraints, cf. [7]:

Algorithm 1 DPLL backtracking algorithm: **solve**

```
Require: $\rho \in \mathbb{R}^n, \sigma \in \{0,1\}^n$
Ensure: $x \in X_f$
1: while true do
2:     branch($\rho, \sigma$)
3:     if CONFLICT then
4:         backtrack()
5:     else if SATISFIED then
6:         return $x$
7:     end if
8: end while
```

Starting with completely unassigned variables, the operation $branch(\rho, \sigma)$ chooses an unassigned variable and assigns it a value (line 2). The rule which variable is chosen and which value is assigned is called *branching strategy*. The branching strategy is guided by the two vectors $\rho \in \mathbb{R}^n$ and $\sigma \in \{0,1\}^n$. Unassigned variables x_i with the highest value ρ_i are prioritized and are set to the value σ_i. After each variable assignment, conflicts are recognized (line 3). If any constraint is not satisfiable anymore, the backtracking is triggered (line 4), i.e., variable assignments are reverted. In case all variables have an assignment and there exists no conflict (line 5), this assignment represents a feasible solution x which is returned (line 6).

3.2 Feasibility-preserving Operator

Algorithm 1 is able to find feasible solutions $x \in X_f$. The backtracking is guided by the branching strategy (ρ, σ) and, thus, these two vectors have a high influence on which solution in X_f is found. Thus, Algorithm 1 is used to generate a feasible initial population by arbitrary random branching strategies.

In Algorithm 2, a feasibility-preserving crossover and mutation operator is presented: Based on the two feasible parent solutions, a branching strategy is derived to obtain a feasible offspring solution using Algorithm 1.

Algorithm 2 Feasibility-preserving crossover and mutation operator

```
Require: $x', x'' \in X_f$; $C \subseteq \{1, ..., n\}$; $r \in \mathbb{R}_{[0,1]}$ (mutation rate)
Ensure: $x \in X_f$
 1: for $i \in \{1, ..., n\}$ do
 2:     if $i \in C$ then
 3:         $\sigma_i = x'_i$
 4:     else
 5:         $\sigma_i = x''_i$
 6:     end if
 7:     $\rho_i = rand(0, 1)$
 8:     if $rand(0, 1) < r$ then
 9:         $\rho_i = \rho_i + 1$
10:         $\sigma_i = \overline{\sigma_i}$
11:     end if
12: end for
13: $x = \mathbf{solve}(\rho, \sigma)$
14: return $x$
```

Algorithm 2 requires two feasible parent solutions $x', x'' \in X_f$, the selection set $C \subseteq \{1, ..., n\}$, and the mutation rate r. A branching strategy (ρ, σ) is generated as follows: The prioritized phase σ_i of the corresponding variable x_i is set to the corresponding value of one of the parent solutions x'_i or x''_i, respectively. This selection is done based on the set C that controls the binary crossover (line 2-6). The priority ρ_i of a variable x_i is randomly chosen in $\mathbb{R}_{[0,1]}$ (line 7). For each variable x_i a mutation is done with the probability r. The mutation increases the priority ρ_i by 1 and flips the prioritized phase value σ_i (line 8-11).

With the given branching strategy (ρ, σ) the PB solver finds a feasible offspring solution $x \in X_f$ (line 13). At this, preserving the information obtained by the parent solutions becomes important. Hence, the *adaption diversity* $div(x, \sigma) = \sum_{i=1}^{n} \frac{|x_i - \sigma_i|}{n}$, that measures the fraction of preserved information of σ, should be kept as small as possible.

Theorem 1. *Given two feasible solutions $x, \tilde{x} \in X_f$, a specific σ, and a random ρ, the probability P to obtain the solutions x in comparison to $\tilde{x}$ is*

$$P(x = \mathbf{solve}(\rho, \sigma)) > P(\tilde{x} = \mathbf{solve}(\rho, \sigma)) \tag{2}$$

if

$$div(x, \sigma) < div(\tilde{x}, \sigma). \tag{3}$$

Proof. Equation (3) implies that $\tilde{x}$ compared to x has a higher count of variables that are different to these corresponding variables of σ. Thus, with a given random ρ, $\tilde{x}$ is excluded with a higher probability earlier in the backtracking search algorithm $\mathbf{solve}(\rho, \sigma)$ compared to x. Hence, x is reached with a higher probability than $\tilde{x}$ as stated in Equation (2).

Thus, the PB solver tends to find a similar offspring solution x compared to σ and preserves the information passed along by the parent solutions x' and x''. By setting C to $\{1, ..., n\}$ or $\{\}$, respectively, Algorithm 2 decays to a feasibility-preserving mutation only operator. This approach is applicable if the *crossover rate* is lower than 1.

3.3 Minimizing the Adaption Diversity

To preserve the information within the feasibility-preserving operator, the adaption diversity has to be minimized. At this, the selection set C that controls the binary crossover has a high influence on this value. We propose a heuristic that finds selection sets based on a graph that represents the constraints of the problem. On the one hand, this heuristic decreases the adaption diversity and, on the other hand, decays to the well known one-point crossover for an unconstrained problem.

Consider the following definitions:

Definition 2 (Constraint-Graph). *A* constraint-graph *$G(V, E)$ is an undirected graph that contains a vertex i for each variable x_i from a problem defined in Equation (1). A function $w : V \times V \to \mathbb{R}$ defines the weight of the edges. For each constraint of the problem an edge between each pair x_i, x_j of variables of the constraint*[2] *is added between the vertices i and j. The weight of the added edge $w(i, j)$ is the reciprocal value of the count of the variables of the constraint. In case there exists already an edge between the vertices i and j, the calculated weight is added to the weight of the existing edge.*

Definition 3 (Cut). *Let $G(V, E)$ denote a graph. A* cut *is a partition of the vertices V in two disjunctive sets C and $\overline{C}$. Any edge $e = (u, v) \in E$ with $u \in C$ and $v \in \overline{C}$ is a* cut edge. *A* weight *of a cut is the sum of the weights of the cut edges.*

[2] Variables are said to be part of a constraint if their corresponding coefficient in C from Equation (1) is non-zero.

A cut on a constraint-graph produces two partitions C and $\overline{C}$. At this, C can be used as the selection set for Algorithm 2. A small cut weight should be aspired since it tends to minimize the number of potentially conflicting constraints and, thus, also tends to minimize the adaption diversity.

A *min-cut algorithm* that finds the minimal cut of an undirected graph is presented in [13]. However, a reasonable crossover operator needs a sufficient number of cuts instead of just a single minimal cut. Based on Algorithm 3, the following proposed heuristic tends to generate $n - 1$ relatively small cuts.

Algorithm 3 Vertex ordering heuristic

```
Require: $G(V, E)$, $w$
Ensure: $P$ is an ordered set
1: while $P \neq V$ do
2:     Select $x_i \in V \backslash P$ with $\sum_{x_j \in P} w(x_i, x_j) = max\{\sum_{x_k \in P} w(x_i, x_k) | x_k \notin P\}$
3:     $P = P \cup \{x_i\}$
4: end while
5: return $P$
```

Given the constraint-graph $G(V, E)$ with the corresponding edge weight function w, the algorithm fills a set P with the vertices V and keeps track of the insertion order. Until the set P contains all vertices from V the algorithm continues (line 1). Each step the most tightly connected vertex x_i with respect to the set P is added to P (line 2-3).

Splitting the ordered set P at one point into two subsets C and $\overline{C}$ generates $n - 1$ cuts with relatively small weights. This is due to the fact that by using the presented heuristic and adding at each step the most tightly connected edge, the weights of the cuts are kept small along the order of P. Thus, we will uses these $n - 1$ C subsets for the selection set C for Algorithm 2. In fact, this approach is similar to the common one-point crossover and decays to it for unconstrained problems.

The time complexity of this algorithm is $O(|E| + |V|log|V|)$, as stated in [13], and with $|V| = n$ and a maximum value of $|E| = n^2$, the aggregated worst-case complexity is $O(n^2)$. However, this algorithm has to be performed only once for each problem. As the experimental results validate, the costs of this heuristic are negligible small compared to the overall runtime of one optimization.

4 Experimental Results

All experimental results were carried out on an Intel Pentium 4 3.20 GHz machine with 1 GB RAM. The implementation of the presented approach is based on the optimization framework OPT4J [14].

4.1 Selection Set

Table 1 presents a comparison of the adaption diversity induced by the feasibility-preserving operator for a completely random selection set C and the selection sets that were obtained by the presented heuristic in Section 3.3.

	T1	T2	T3	T4	T5	T6
random	0.19	0.22	0.19	0.10	0.16	0.18
heuristic	0.12	0.14	0.04	0.02	0.02	0.01

Table 1. Results for the adaption diversity on all testcases with a *random* selection set C and random selection set that was obtained by the presented *heuristic*.

In particular, the induced adaption diversity for the structured testcases T3-T6 is significantly lower for the sets that are obtained by the proposed heuristic. These results validate that the heuristic effectively preserves the information of the parents. Thus, for following experimental results the novel feasibility-preserving technique is performed in combination with the presented adaption diversity minimizing heuristic.

4.2 Optimization

The section compares different *EA* based constraint-handling strategies on the six testcases. The penalty function based approach (*penalty*) tries to minimize the following function (cf. Equation (1)) that prioritizes feasible over infeasible solutions:

$$f'(x) = f(x) + p(x)$$
$$\text{with } p(x) = n \cdot min\{e_1 + ... + e_n\} \text{ such that } Ax \leq b + e \text{ and } e \in \mathbb{N}_0{}^m$$

This approach is extended a by a greedy repair algorithm (*greedy*) [5] that flips each variable trying to minimize $p(x)$. In case of $p(x) = 0$, the greedy algorithm stops since x is a feasible solution. The SAT decoding approach [2] is in the following denoted as *satdec*. The feasibility-preserving approach proposed in this work is denoted as *satop*. For all methods, an elitist EA is used with the population size of 100 individuals, generating 25 offspring from 25 random selected parent solutions. For all methods, the mutation rate is set to $r = \frac{1}{n}$. For the binary vectors, a binary crossover is used, followed by a bit-flip with probability r. For the *satdec* approach, the crossover for the real vector is implemented by the SBX ($\nu = 15$) operator, followed by polynomial mutation ($\eta = 20$) with probability p. For each testcase 10 instances were generated, and for each instance 10 runs were carried out to allow a calculation of am overall meaningful average.

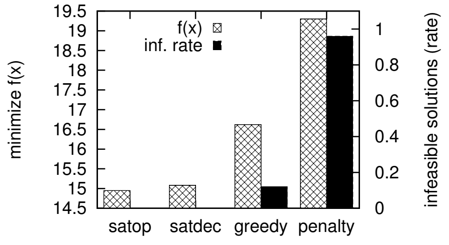
The results of the optimization runs is given in Figure 1. The proposed feasibility-preserving approach *satop* delivers the best solutions for all testcases except T4 where *satdec* is slightly better. This is due to the fact, that *satdec* works best one problems with very few feasible solutions. This is also apparent on testcase T6 with many feasible solutions, where *satdec* is only slightly better than the *greedy* method. However, though *satop* and *satdec* are both based on a PB solver and deliver feasible solutions only, *satop* is about four times faster in average compared to *satdec* and even faster than the *greedy* approach on the structured testcases T3-T6. Except for T4, the *satop* method delivers remarkably better solutions on the testcases T3,T5, and T6 that represent real-world problems. Note that the *greedy* approach becomes remarkably slow on the large problem where obtaining a feasible solution is difficult and fails to find a single feasible solution on the testcases T4 and T5. The method *penalty* is the fastest, but delivers bad results and fails completely to find feasible solutions on the larger problems T3 to T6.

5 Conclusion

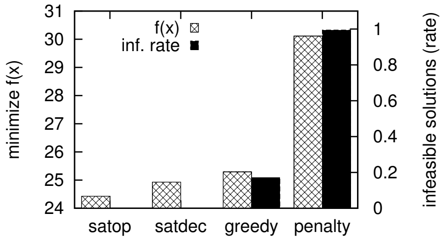
In this paper, a feasibility-preserving crossover and mutation operator for constrained combinatorial problems is presented. This operator allows an Evolutionary Algorithm to perform efficiently also on large and problems with few feasible solutions. The experimental results compare this novel approach to known methods based on a proposed testsuite. The results show that the feasibility-preserving operator is superior to common methods like penalty functions or a greedy repair algorithm. Compared to the state-of-the-art SAT decoding, the novel approach is four times faster and delivers, except for one testcase, better solutions.

References

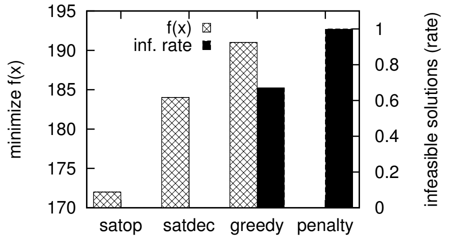
1. Karp, R.M.: Reducibility among combinatorial problems. In Miller, R.E., Thatcher, J.W., eds.: Complexity of Computer Computations, Plenum Press (1972) 85–103
2. Lukasiewycz, M., Glaß, M., Haubelt, C., Teich, J.: SAT-Decoding in Evolutionary Algorithms for Discrete Constrained Optimization Problems. In: Proceedings of CEC '07. (2007) 935–942
3. Coello, C.: Theoretical and numerical constraint handling techniques used with evolutionary algorithms: A survey of the state of the art. Art. Computer Methods in Applied Mechanics and Engineering **191(11-12)** (2002) 1245–1287
4. Michalewicz, Z., Schoenauer, M.: Evolutionary algorithms for constrained parameter optimization problems. Evolutionary Computation **4**(1) (1996) 1–32
5. Zitzler, E., Thiele, L.: Multiobjective Evolutionary Algorithms: A Comparative Case Study and the Strength Pareto Approach. IEEE Transactions on Evolutionary Computation **3**(4) (1999) 257–271
6. Koziel, S., Michalewicz, Z.: A decoder-based evolutionary algorithm for constrained parameter optimization problems. In: Proceedings of PPSN '98. (1998) 231–240
7. Chai, D., Kuehlmann, A.: A fast pseudo-boolean constraint solver. In: Proceedings of DAC '03. (2003) 830–835
8. Lukasiewycz, M., Glaß, M., Haubelt, C., Teich, J.: Efficient symbolic multi-objective design space exploration. In: Proceedings of the ASP-DAC '08. (2008) 691–696
9. Aloul, F.A., Ramani, A., Markov, I.L., Sakallah, K.A.: Solving difficult SAT instances in the presence of symmetry. In: Proceedings of DAC '02. (2002) 731–736
10. Prasad, M.R., Chong, P., Keutzer, K.: Why is ATPG easy? In: Proceedings of DAC '99. (1999) 22–28
11. Aloul, F.A., Ramani, A., Markov, I.L., Sakallah, K.A.: Generic ILP versus specialized 0-1 ILP: an update. In: Proceedings of ICCAD '02. (2002) 450–457
12. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commun. ACM **5**(7) (1962) 394–397
13. Stoer, M., Wagner, F.: A simple min-cut algorithm. J. ACM **44**(4) (1997) 585–591
14. Opt4J: (Java Optimization Framework) http://www.opt4j.org/.



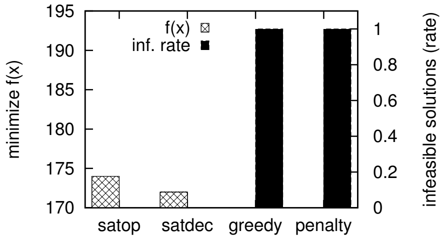
(a) Results for T1.



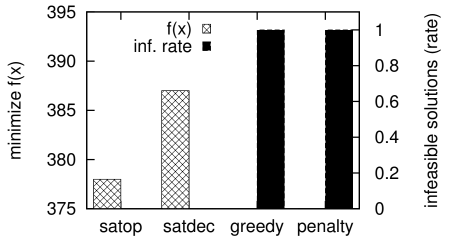
(b) Results for T2.



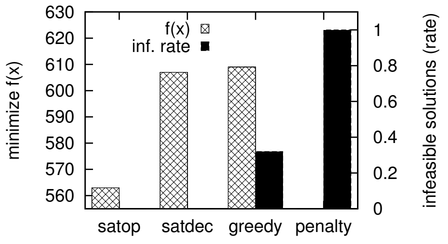
(c) Results for T3.



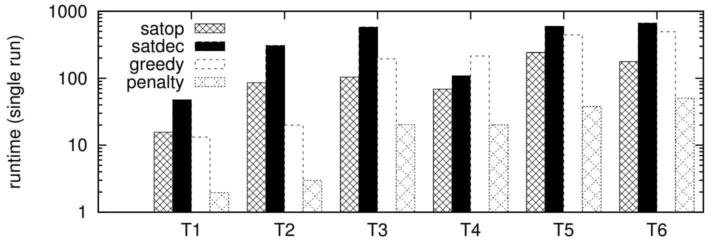
(d) Results for T4.



(e) Results for T5.



(f) Results for T6.



(g) Runtimes for all testcases.

Fig. 1. 1(a) to 1(f) show the results of the four optimization methods on the presented testcases. Given is the reached minimal value of $f(x)$ as well as the rate of infeasible solutions that were obtained throughout the optimization. In 1(g), the runtimes of the optimization methods on the testcases are denoted. Note that these values are given in logarithmic scale.