European Journal of Operational Research 263 (2017) 789–804

Contents lists available at ScienceDirect

European Journal of Operational Research

journal homepage: www.elsevier.com/locate/ejor

Discrete Optimization

Adaptive Kernel Search: A heuristic for solving Mixed Integer linear Programs

G. Guastaroba[a,*], M. Savelsbergh[b], M. G. Speranza[a]

[a] *Department of Economics and Management, University of Brescia, C.da S.Chiara 50, Brescia, Italy*
[b] *H. Milton Stewart School of Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta, GA, USA*

ARTICLE INFO

Article history:
Received 1 September 2016
Accepted 2 June 2017
Available online 9 June 2017

Keywords:
Mixed integer linear programming
General-purpose heuristic
Kernel Search
Adaptive heuristic

ABSTRACT

We introduce Adaptive Kernel Search (AKS), a heuristic framework for the solution of (general) Mixed Integer linear Programs (MIPs). AKS extends and enhances Kernel Search, a heuristic framework that has been shown to produce high-quality solutions for a number of specific (combinatorial) optimization problems in a short amount of time. AKS solves a sequence of carefully constructed restricted MIPs (using a commercial MIP solver). The computational effort required to solve the first restricted MIP guides the construction of the subsequent MIPs. The restricted MIPs are constructed around a *kernel*, which contains the variables that are presumably non-zero in an optimal solution. Computational results, for a set of 137 instances, show that AKS significantly outperforms other state-of-the-art heuristics for solving MIPs. AKS also compares favorably to CPLEX and offers more flexibility to trade-off solution quality and computing time.

1. Introduction

A general *Mixed Integer linear Program* (MIP) can be stated as:

$$\begin{aligned} \text{minimize} \quad & z = \mathbf{c}^T\mathbf{x} \\ \text{subject to} \quad & \mathbf{Ax} \geq \mathbf{b} \\ & x_j \geq 0 \quad j \in \mathcal{C} \\ & x_j \in \{0,1\} \quad j \in \mathcal{B} \\ & x_j \in \mathbb{Z}_{\geq 0} \quad j \in \mathcal{I}, \end{aligned}$$

where $\mathbf{x}$ is a vector of n decision variables, $\mathbf{c}$ is a vector of n objective coefficients, $\mathbf{A}$ is an $m \times n$ matrix of technology coefficients, and $\mathbf{b}$ is a vector of m right-hand side coefficients. The set of variables is partitioned into a set of binary variables, $\mathcal{B}$, a set of general integer variables, $\mathcal{I}$, and a set of continuous variables, $\mathcal{C}$. When the set of general integer variables is empty, i.e., set $\mathcal{I} = \emptyset$, we have a 0–1 MIP.

The *focus of our research* is developing a heuristic framework for obtaining high-quality solutions to general MIPs in a reasonable amount of time. In addition, we want the framework to be simple and such that a user can control the trade-off between solution quality and computational effort with just a few parameters.

To do so, we extend and enhance Kernel Search (KS), an integer programming based heuristic framework introduced by Angelelli, Mansini, and Speranza (2010) for the solution of specific (combinatorial) optimization problems formulated as 0–1 MIPs. We called the resulting algorithm *Adaptive Kernel Search* (*AKS*). KS has two phases. In the first phase, KS identifies a subset of variables, called the *kernel*, which should contain most of the variables that are likely to have a non-zero value in an optimal solution, and seeks for an initial feasible solution. In the second phase, KS tries to improve the initial solution by solving a sequence of restricted MIPs (using a commercial MIP solver) involving the variables in the kernel plus a number of additional variables. One of the challenges when applying KS is choosing the "right" kernel size. A small kernel results in smaller restricted MIPs, which can be solved (more) easily, but may not lead to high-quality solutions; a large kernel results in large restricted MIPs, which may lead to high-quality solutions, but cannot be solved in a reasonable amount of time. Choosing an appropriate kernel size is especially challenging when solving general MIPs, because a large variety and diversity of instances have to be accommodated. Upon this observation we designed AKS, which assesses the difficulty of an instance and adjusts its behavior based on its characterization. Hence, the adjective *adaptive* in AKS. For the same reasons, i.e., large variety and diversity of instances, obtaining an initial feasible solution can be (more) challenging. Thus, AKS also includes enhanced functionality for obtaining an initial feasible solution. AKS has three phases. In the first phase, an initial feasible solution is obtained and the diffi-

* Corresponding author.
E-mail addresses: gianfranco.guastaroba@unibs.it (G. Guastaroba), martin.savelsbergh@isye.gatech.edu (M. Savelsbergh), grazia.speranza@unibs.it (M.G. Speranza).

http://dx.doi.org/10.1016/j.ejor.2017.06.005
0377-2217/© 2017 Elsevier B.V. All rights reserved.

culty of the instance is assessed. In the second phase, the kernel is adjusted based on the assessed difficulty of the instance. Finally, in the third phase, as in KS, AKS seeks to improve the initial solution by solving a sequence of restricted MIPs involving the variables in the kernel plus a number of additional variables.

AKS has been tested, first, on a set of 29 benchmark instances that have been used in previous studies on heuristics for general MIPs. Second, AKS has been tested on 108 instances taken from the MIPLIB2010 library. The results demonstrate that AKS outperforms other state-of-the-art heuristics. Furthermore, a comparison with commercial MIP solver CPLEX, when imposing a time limit, also shows that AKS performs well; AKS produced better solutions than CPLEX for 29 instances. Equally important, and maybe even more interesting, by adjusting the parameters of AKS, slightly worse solutions are obtained, on average, but with a remarkable reduction in computing time. The latter highlights the flexibility of the AKS heuristic framework, i.e., a user's ability to control the trade-off between solution quality and computation effort.

The remainder of the paper is organized as follows. Relevant literature is discussed in Section 2 and a review of Kernel Search is provided in Section 3. Adaptive Kernel Search is introduced and described in detail in Section 4. The results of our extensive computational study are presented and discussed in Section 5. Finally, in Section 6, we give some concluding remarks.

2. Literature review

In this section, we provide an overview of the contributions on heuristics for MIPs. We first present the heuristics aimed at quickly finding an initial feasible solution, sometimes called primal heuristics, and then the general-purpose heuristics for the solution of MIPs, at times called improvement heuristics.

A considerable amount of research has focused on developing heuristics aimed at quickly finding an initial feasible solution. The paper by Fischetti, Glover, and Lodi (2005) has been among the first to tackle this problem for a general MIP. The authors propose the Feasibility Pump (FP) heuristic, which is an approach based on generating two trajectories of points, say $\{x^*\}$ and $\{\tilde{x}\}$, that satisfy the MIP feasibility in a complementary way, i.e., x^* is feasible for the Linear Programming (LP) relaxation but not necessarily integer, whereas $\tilde{x}$, which is obtained using a simple rounding procedure, is integer but may not be LP feasible. The basic idea of FP is to iteratively reduce the distance between x^* and $\tilde{x}$ until they coincide, producing an integer feasible solution. The authors then concentrate on 0–1 MIPs, mentioning that, in preliminary experiments, the application of FP to MIPs with integer variables showed an increased probability of cycling. FP has proven to be quite successful and is nowadays a component of commercial solvers, including CPLEX and Gurobi. Several authors developed methods to improve the original FP scheme. Bertacco, Fischetti, and Lodi (2007) extend the original FP heuristic to general MIPs, devising a two-phase heuristic. In the first phase FP algorithm relaxes the integrality condition on all the integer variables, concentrating the search on the binary variables. All binary and integer variables are then considered in the second phase. Finally, a local search phase is carried out if no feasible solution is found during the second phase. The algorithm is stopped as soon as the first feasible solution is found. The probability of cycling is controlled through a restart procedure, which is used whenever a situation of stalling is detected. Fischetti and Lodi (2008) computationally analyze a hybrid algorithm that combines FP as introduced in Fischetti, Glover, and Lodi (2005) with the Local Branching (LB) algorithm proposed in Fischetti and Lodi (2003), and based on the idea of exploring a neighborhood of a solution through the introduction of linear inequalities. This hybrid algorithm starts using the FP procedure to determine an “almost feasible” solution. Subsequently, the original MIP is modified introducing a continuous and a binary variable for each violated constraint. The LB algorithm is then applied to the modified MIP to minimize the number of violated constraints. As the quality of the solution found by the above FP algorithms is sometimes poor, some authors focused on improving the procedure used to choose the LP feasible solution closest to $\tilde{x}$. Among them, the objective FP proposed by Achterberg and Berthold (2007) is a method based on combining, through a weighted function, the original objective function and the distance between the two points. Focusing on 0–1 MIPs, De Santis, Lucidi, and Rinaldi (2013) propose to choose the new LP feasible solution using a nonsmooth concave function which penalizes the presence of fractional variables. Their approach is later extended to general MIPs in De Santis, Lucidi, and Rinaldi (2014). Some other authors concentrated their efforts on studying the use of more elaborated rounding procedure. In particular, borrowing some concepts from the constraint programming literature, Fischetti and Salvagnin (2009) study the effectiveness of replacing the original rounding function present in the FP devised in Bertacco, Fischetti, and Lodi (2007) and Fischetti, Glover, and Lodi (2005) with a diving-like procedure based on rounding and constraint propagation. The benefits of applying a rounding function different from the one adopted in the original FP are also investigated by Baena and Castro (2011). In this paper, the authors consider a rounding scheme where all points lying on a line segment are candidates to be rounded. The end-points of the segment are the feasible points chosen in the original FP and the analytic center of the polytope of the LP relaxation. The line segment is then discretized to choose the points to round. Along the same general lines, Boland et al. (2014) replace the original rounding procedure with the exploration of rounded solutions along a line segment. In contrast with Baena and Castro (2011), the authors show that it is possible to efficiently compute a rounding of every point along the line segment, hence avoiding the need to choose a discretization. A different approach for quickly finding feasible solutions to 0–1 MIPs is proposed by Eckstein and Nediak (2007). Their method is based on the use of concave functions that take zero values for integer feasible solutions, and positive values otherwise. Another method for 0–1 MIPs is introduced in Guzelsoy, Nemhauser, and Savelsbergh (2013). This approach, called restrict-and-relax search, is a branch-and-bound algorithm that, starting from a sub-problem, at any node of the search tree may selectively restrict (or fix) variables, relax (or unfix) previously fixed variables, or simultaneously fix and unfix variables using dual or problem-specific information. Despite the method is, in theory, capable of proving the optimality of a solution, the authors concentrate their study on finding feasible solutions quickly. Note that, in principle, most of the above methods can be extended to be used as general-purpose heuristics for MIPs. To this aim, Fischetti, Glover, and Lodi (2005) devise an extension of the original FP based on introducing a constraint that makes the incumbent solution infeasible, and then applying FP again for a certain number of iterations. However, this is not their main purpose which remains to quickly find an initial feasible solution.

We now review the literature devoted to the design of general-purpose heuristics for MIPs. Balas, Ceria, Dawande, Margot, and Pataki (2001) point out that, in contrast with the many heuristics developed for specific applications, there have been few developments on the design of general-purpose heuristics for MIPs. The authors report that the main proposals appeared in the literature until that moment were no more than half a dozen. Although one decade and a half have passed since that paper was published, the number of papers devising a heuristic for general MIPs is still fairly limited. Glover and Laguna (1997a; 1997b) provide a theoretical framework for designing heuristics for MIPs, but without providing any empirical evidence. Løkketangen and Glover (1998) devise a Tabu Search approach for solving 0–1 MIPs, whereas Balas, Ceria, Dawande, Margot, and Pataki (2001) implement a heuristic performing a local search in the integer neighborhood of a fractional solution of a 0–1 MIP. Glover (2006) presents the general design of a parametric Tabu Search method for the solution of 0–1 MIPs. The approach solves a series of LP problems incorporating branching inequalities as weighted terms in the objective function, and includes some typical components of the Tabu Search metaheuristic, such as an adaptive memory scheme, as well as intensification and diversification procedures. A computational study of the performance of a basic version of the above parametric Tabu Search is provided in Sacchi and Armentano (2011). The currently most effective general-purpose heuristics for MIPs make use of a MIP solver within a heuristic scheme. A survey on these heuristics is provided by Fischetti and Lodi (2011). A significant step forward in the design of heuristics for general MIPs is represented by the already mentioned LB, introduced by Fischetti and Lodi (2003). LB proceeds by defining neighborhoods of a known feasible solution by introducing in the original MIP linear inequalities (called local branching cuts), and then exploring each neighborhood using an MIP solver as a black-box. The LB algorithm presented in Fischetti and Lodi (2003) requires a starting feasible solution and is based on the assumption that the MIP contains a relevant set of binary variables. In the conclusions, the authors provide some guidelines about how to extend their framework to handle the cases of a starting infeasible solution, and of MIPs with (only) integer variables. Balas, Schmieta, and Wallace (2004) show the benefits of combining a heuristic for general MIPs, called pivot and shift, with the XPRESS MIP solver. The Relaxation Induced Neighborhood Search (RINS) framework introduced by Danna, Rothberg, and Le Pape (2005) had a great impact on the evolution of this type of heuristics for MIPs, as witnessed by the number of commercial solvers that are now incorporating it, including CPLEX, XPRESS, and Gurobi. RINS shares some ideas with LB, in particular that of describing a neighborhood as a restricted MIP, and exploring it using a MIP solver. The distinctive characteristic of RINS is that the neighborhoods are defined through the incumbent and the LP relaxation solutions. At specified nodes of the search tree, the current LP relaxation and the incumbent integer solution are compared, and all the integer variables that take the same value in both solutions are fixed. In this way, a simplified sub-problem is obtained restricted to the remaining variables, which is then solved with a MIP solver. Hansen, Mladenović, and Urošević (2006) hybridize LB with a Variable Neighborhood Search (VNS) algorithm, calling the resulting method the VNS Branching (VNSB) algorithm. In their approach, neighborhoods of a given solution are defined as in the LB but are changed more systematically, along the lines of the VNS framework. Rothberg (2007) devises an evolutionary approach for improving, or polishing, solutions to general MIPs. The approach, integrated within a MIP solver, is invoked at specific nodes of the search tree, and involves the classical ingredients of genetic algorithms, including combination and mutation steps which are performed using a large neighborhood search. Fischetti and Monaci (2014) design a method aimed at quickly improving feasible solutions for 0–1 MIPs. The approach, called proximity search, starts with a feasible integer solution, and seeks to improve it using a MIP solver to explore a neighborhood defined by replacing the original objective function with a proximity function that measures the distance from the incumbent solution.

To the best of our knowledge, the current state-of-the-art heuristic for solving 0–1 MIPs is the Variable Neighborhood Decomposition Search (VNDS) introduced in Lazić, Hanafi, Mladenović, and Urošević (2010). VNDS is a two-level VNS heuristic based upon the decomposition of the original problem into subproblems. These problems are obtained by hard fixing a certain number of variables, say k, to their values in the incumbent solution. More precisely, at each iteration of VNDS, all variables are sorted in non-decreasing order of the absolute value of the distance between the value taken in the incumbent solution and in the LP relaxation. The first k variables in the list are then fixed and the resulting sub-problem is solved by means of a MIP solver. If a better incumbent is found, a Variable Neighborhood Descent procedure (i.e., a variant of a VNS where neighborhoods are changed deterministically) is carried out and the process is iterated. If not, the number of variables fixed in the current sub-problem is reduced. Note that if the variables fixed in the sub-problems are only those with zero distance, then VNDS becomes an approach very akin to the RINS. Lazić et al. (2010) provide extensive computational experiments solving 29 benchmark instances, and report the most comprehensive, to our knowledge, performance comparison among general-purpose heuristics designed for solving 0–1 MIPs. In their computational results, the authors compare VNDS with LB by Fischetti and Lodi (2003), RINS by Danna, Rothberg, and Le Pape (2005), and VNSB by Hansen, Mladenović, and Urošević (2006). The computational results indicate that the average quality of the solutions found by VNDS is considerably better than any of the other methods considered, whereas VNDS is the second best methods in terms of computing time, after LB. More recently, Hanafi, Lazić, Mladenović, Wilbaut, and Crévits (2015) devised two heuristics based on the variable neighborhood decomposition search principles, one of which produces results similar to those of the VNDS by Lazić et al. (2010).

Finally, it is worth noting that most of the above general-purpose heuristics cannot exist in isolation but must be seeded with one or more feasible solutions. The latter solutions can be obtained by means of a MIP solver (e.g., terminated once a first feasible solution is found) or using one of the methods described in the first part of this section.

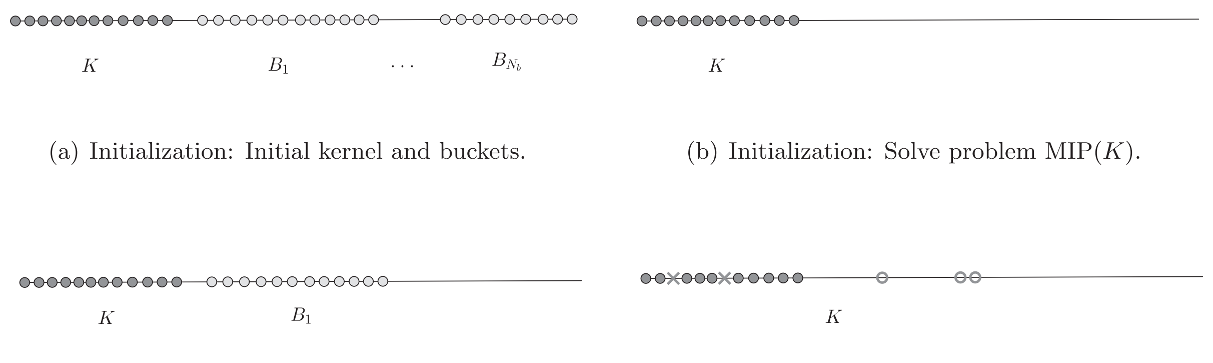
3. Kernel Search

In this section, we introduce the Kernel Search (KS) algorithmic framework and review its applications so far. Our description of the KS algorithmic framework follows Guastaroba and Speranza (2014). The KS algorithmic framework is designed to solve optimization problems that can be formulated as a 0–1 MIP. Most of the notation and definitions introduced in this section will be used throughout the remainder of the paper.

Let $\mathcal{B}$ be the set of binary variables of a 0–1 MIP. Let $U \subsetneqq \mathcal{B}$, then we denote the *restricted* 0–1 MIP in which the binary variables in $\mathcal{B} \setminus U$ are fixed to zero by MIP(U). Note that all continuous variables are always included in a restricted 0–1 MIP. Furthermore, let z^H denote an upper bound on the value of an optimal solution to the original problem, i.e., to MIP($\mathcal{B}$).

We say that a binary variable is *promising* if its value is likely to be one in an optimal solution to the original problem. The KS algorithmic framework aims to always "work" with a restricted 0–1 MIP defined by a *kernel* $K \subset \mathcal{B}$ and a, possibly empty, *bucket* $B \subset \mathcal{B} \setminus K$ with the properties that: (1) an optimal solution to MIP($K \cup B$) is likely to be an optimal solution to the original problem, or will, at least, provide a high-quality solution to the original problem, and (2) an optimal solution to MIP($K \cup B$) can be obtained in a reasonable amount of time. Ideally, all promising variables are in the kernel K and the buckets are used as a diversification mechanism. As it is difficult to determine such a kernel K up front, the kernel is updated using a "learn and adjust" scheme, as sketched in Algorithm 1.

The KS algorithmic framework has two phases. The first phase determines an initial kernel, a ranking of the set of remaining variables, and a partitioning of the, now ordered, set of remaining variables into buckets. The solution to the LP relaxation of the original problem guides the process. The initial kernel K is taken to



K
B_1
$\cdots$
B_{N_b}
(a) Initialization: Initial kernel and buckets.
K
(b) Initialization: Solve problem MIP(K).
K
B_1
(c) Improvement: Solve problem MIP($K \cup B_1$).
K
(d) Improvement: Learn and adjust the kernel.

Fig. 1. An illustrative example of the operation of the KS algorithmic framework.

Algorithm 1 The KS algorithmic framework.

```
   {Initialization}
1: Solve the linear programming relaxation of the 0–1 MIP.
2: Determine an initial kernel $K$ and a sequence of buckets $\{B_i\}_{i=1,\dots,N_b}$ partitioning $\mathcal{B} \setminus K$.
3: Solve MIP($K$).
   {Improvement}
4: while $i \le \min\{\overline{N_b}, N_b\}$ do
5:     Solve MIP($K \cup B_i$).
6:     Analyze the solution and adjust kernel $K$.
7: end while
```

be the set of variables with a positive value in the solution to the LP relaxation. This choice has proven, in a number of settings, to provide a good compromise between solution quality and algorithm efficiency. Next, the remaining variables are sorted in non-decreasing order of their reduced cost and then partitioned into buckets based on an a priori specified bucket length, L_b. Consequently, all but the last bucket will have exactly L_b variables. Thus, if the initial kernel has m variables, then $N_b = \left\lceil \frac{|\mathcal{B}|-m}{L_b} \right\rceil$ buckets will be created. Again, based on experience in different settings, the bucket length L_b is set to m. The final step of the first phase is to solve MIP(K). The solution value, if any, is used to set the (initial) upper bound z^H (otherwise, $z^H = \infty$).

The second phase seeks to improve the solution found in the first phase, if any. A sequence of $\min\{\overline{N_b}, N_b\}$ restricted 0–1 MIPs is solved, where $\overline{N_b}$ is a given non-negative parameter. Specifically, in each iteration i for $i = 1, \dots, \min\{\overline{N_b}, N_b\}$, a slightly modified version of MIP($K \cup B_i$) is solved. First, the upper bound z^H is used to set an objective function cutoff value, because we are only interested in improved solutions. Secondly, we add a constraint requiring that at least one binary variable of B_i has to be one in an optimal solution, i.e., $\sum_{j \in B_i} x_j \ge 1$. By construction, if MIP($K \cup B_i$) is feasible, an optimal solution to it is improving the upper bound z^H. The optimal solution, if it exists, is used to update z^H and to “learn and adjust”. By construction, it contains at least one binary variable from bucket B_i, i.e., at least one new promising variable has been identified. New promising variables are added to the kernel. To control the size of the kernel, variables in the kernel that have not been in the solution to MIP($K \cup B_i$) in the last p iterations, where p is a given parameter, are no longer considered promising and, consequently, removed. After solving $\min\{\overline{N_b}, N_b\}$ restricted 0–1 MIPs, the search for an improving solutions ends.

Fig. 1 illustrates an iteration in the KS algorithmic framework. In the first phase, an initial kernel (solid dark gray circles) and a sequence of buckets (solid light gray circles) are defined. At the end of the first phase, MIP(K) is solved. The first problem solved in the second phase is MIP($K \cup B_1$). If an improved solution is found, the kernel is updated, i.e., new promising variables are added (open circles) and variables that are no longer promising, if any, are removed (crosses). The second phase proceeds with solving MIP($K \cup B_2$), where K is the updated kernel.

It is worth pointing out that even though the KS algorithmic framework has basically only two parameters, i.e., $\overline{N_b}$ and p, the values of these parameters can impact the solution time and the quality of the final solution significantly. Indeed, the smaller the value of p, the sooner variables are removed from the kernel. This results in smaller restricted 0–1 MIPs that are (usually) faster to solve, but often at the expense of a deterioration in the quality of the final solution. Similarly, the smaller the value of $\overline{N_b}$, the poorer the quality of the final solution, but the smaller the computing time, because variables belonging to buckets later in the sequence are not considered and fewer restricted 0–1 MIPs are solved.

The KS algorithmic framework has been used for the solution of variety of (combinatorial) optimization problems, often taking advantage of the specific problem structure. The first time the KS algorithmic framework was applied was for the solution of the multi-dimensional knapsack problem (Angelelli, Mansini, & Speranza, 2010). Subsequently, the same authors successfully deployed the framework for the solution of a portfolio selection problem (Angelelli, Mansini, & Speranza, 2012). Removing variables from the kernel was first introduced in Guastaroba and Speranza (2012a) when the KS was used for the solution of an index tracking problem. Afterward, the KS algorithm has successfully been applied to capacitated facility location problems, both in the multi-source version (Guastaroba & Speranza, 2012b) and in the single-source variant (Guastaroba & Speranza, 2014). Finally, Filippi, Guastaroba, and Speranza (2016) have recently extended the KS algorithmic framework to address the bi-objective enhanced index tracking problem.

4. Adaptive Kernel Search

As indicated above, the KS algorithmic framework has, so far, been used, and tuned, for the solution of instances of specific optimization problems. However, it is clear that, in principle, the framework can be used to solve any instance of a general mixed integer program. Of course, being able to perform well on *any* instance of a general MIP will be challenging, because of the variety and diversity of instances that can be encountered. In this section, we describe enhancements to the basic KS algorithmic framework that, as an extensive computational study will show, result in a powerful tool for efficiently obtaining high-quality solutions to an arbitrary MIP. To avoid being repetitive, we focus only on the enhancements made to the framework. The key changes are detailed below:

- Using the root node LP relaxation instead of the LP relaxation;
- Enhanced functionality for finding an initial feasible solution;
- An adaptivity scheme based on the difficulty of an instance; and
- Accommodating general integer variables.

General integer variables are handled separately, but identically to binary variables. That is, the enhanced KS algorithmic framework maintains two ordered lists of integer variables, one with binary variables and one with general integer variables. Each ordered list is used to define variables that will constitute the initial kernel and the buckets that partition the remaining variables. After the two lists have been created and processed, the binary and general integer variables will be merged to form the actual initial kernel and the buckets. For presentational convenience, we describe the enhancements as if the MIP has only binary variables, and discuss how MIPs with general integer variables are handled at the end of the section.

The *Adaptive Kernel Search* (AKS) algorithmic framework is detailed in Algorithm 2.

Algorithm 2 General scheme of AKS for general MIPs.

```
    {Identify Initial Feasible Solution}
 1: Solve the root node.
 2: Determine an initial kernel $K$ and a sequence of buckets $\{B_i\}_{i=1,\dots,N_b}$ partitioning $\mathcal{B} \setminus K$.
 3: Solve MIP($K$).
 4: if no feasible solution has been found then
 5:     Apply GetFeasible.
 6: end if
    {Assess Difficulty & Adjust the Kernel}
 7: Assess instance difficulty.
 8: if instance is Easy then
 9:     Adjust kernel using Easy procedure.
10: else if instance is Hard then
11:     Adjust kernel using Hard procedure.
12: end if
    {Improvement}
13: while $i \le \min\{\overline{N_b}, N_b\}$ do
14:     Solve MIP($K \cup B_i$).
15:     Analyze the solution and adjust kernel $K$.
16: end while
```

The framework reflects the need to be able to handle a variety and diversity of instances. Specifically, there is a much stronger emphasis on finding an initial feasible solution and on recognizing that the effort required to do so can be used to direct the search for improved solutions. In the first phase, the root node of the search tree is evaluated. The solution to the final linear program solved at the root node is used to determine an initial kernel, a ranking of the remaining variables, and a partition of the (ordered set of) remaining variables into buckets, as in the standard KS algorithmic framework. If a feasible integer solution to the original problem was found at the root node, its value is used to initialize the upper bound z^H, otherwise $z^H = \infty$. Once the initial kernel and buckets are defined, problem MIP(K) is solved, using z^H as a cutoff on the objective function value. In our computational experiments, we found that MIP(K) is feasible in all but a few instances, which, assuming that the original problem does have a feasible solution, is an indication that the initial kernel is too small. To handle this situation, AKS iteratively increases the size of the kernel until MIP(K) is feasible (see Algorithm 3). Specifically, at each iteration, AKS adds the first $m \times w$ binary variables in the ordered list of remaining variables (i.e., the non-kernel variables), where $w \in \mathbb{R}_{>0}$ is

Algorithm 3 GetFeasible.

```
1: while MIP($K$) is not feasible do
2:     Increase the size of kernel $K$.
3:     Solve MIP($K$).
4: end while
5: Determine a sequence of buckets $\{B_i\}_{i=1,\dots,N_b}$ partitioning $\mathcal{B} \setminus K$.
```

a given parameter. As soon as MIP(K) becomes feasible, a new partition of the remaining variables into buckets is created.

In the second phase, AKS assesses the difficulty of the instance by looking at the effort required to obtain a solution to MIP(K), and adjusts the size of the kernel so that it has an “appropriate” size when improving solutions are sought. More specifically, the time taken in the first phase to solve the restricted problem MIP(K) (i.e., either MIP(K) with the initial kernel or the last MIP(K) solved in GetFeasible) is used to assess the difficulty of the instance. AKS classifies an instance as either Easy, Normal, or Hard. Let $t_{MIP(K)}$ denote the time it takes to solve the above MIP(K), then AKS classifies an instance as Easy if $t_{MIP(K)}$ is less than or equal to threshold t_{Easy}, as Hard if $t_{MIP(K)}$ is greater than or equal to threshold t_{Hard}, and as Normal otherwise. When an instance has been classified as Easy, the size of the kernel is iteratively increased as long as problem MIP(K) remains easy to solve, or all binary variables have been added to the kernel. When an instance has been classified as Hard, the algorithm reduces the size of the kernel by permanently fixing a subset of variables. Finally, when an instance has been classified as Normal, the size of the kernel remains unchanged.

4.1. Kernel adjustment for instances classified as Easy

At each iteration, AKS adds the first $m \times q$ variables of the ordered list of remaining (non-kernel) variables to kernel K, where $q \in \mathbb{R}_{>0}$ is a given parameter. Let the set of variables that is added be denoted by K^+. Subsequently, MIP(K) is solved with the objective cutoff value set to z^H and with constraint $\sum_{j \in K^+} x_j \ge 1$ added, i.e., the solution to MIP(K) must include at least one variable from those that have just been added to the kernel. The process continues until the time taken to solve MIP(K) is larger than t_{Easy}, or the kernel contains all binary variables. In the latter case, a solution to MIP(K) is optimal to the original problem and the algorithm stops. For more details, see Algorithm 4.

Algorithm 4 Kernel adjustment for instances categorized as Easy.

```
1: while $t_{MIP(K)} \le t_{Easy}$ and $K \ne \mathcal{B}$ do
2:     Increase the size of kernel $K$.
3:     Solve MIP($K$).
4: end while
5: if $K \ne \mathcal{B}$ then
6:     Determine a sequence of buckets $\{B_i\}_{i=1,\dots,N_b}$ partitioning $\mathcal{B} \setminus K$.
7: end if
```

4.2. Kernel adjustment for instances classified as Hard

Kernel adjustment for instances classified as Hard aims to reduce the computational effort required for solving MIP($K \cup B_i$), when seeking improved solutions, by decreasing the size of the kernel. The size of the kernel is decreased by permanently fixing the value of some binary variables. More specifically, AKS permanently fixes all binary variables with value greater than $1 - \epsilon$ in the solution to the final LP solved at the root node, x^{LP}, to 1, where $\epsilon > 0$ is a pre-specified tolerance. For more details, see Algorithm 5.

Algorithm 5 Kernel adjustment for instances categorized as Hard.

```
1: for $j \in \mathcal{B}$ do
2:     if $x_j^{LP} > 1 - \epsilon$ then
3:         Permanently fix $x_j = 1$.
4:     end if
5: end for
```

4.3. Improvement phase

Once the kernel has been adjusted, AKS commences the improvement phase, which proceeds along the lines outlined above for the standard KS algorithmic framework with the following modification. When MIP($K \cup B_{i-1}$) has not been solved to proven optimality, because a given limit was reached (see Section 5), it is possible that MIP($K \cup B_i$) contains an improving solution in which all variables in B_i are 0. Therefore, the constraint added to MIP($K \cup B_i$) is $\sum_{j \in B_i \cup \{j \in K : x_j^H = 0\}} x_j \geq 1$ rather than $\sum_{j \in B_i} x_j \geq 1$, where x^H represents the solution associated with the upper bound z^H.

4.4. Handling general integer variables

For instances categorized as Hard, adjusting the kernel (of general integer variables) involves permanently fixing a variable $j \in \mathcal{I}$ with $x_j^{LP} \in \left[\left\lfloor x_j^{LP} \right\rceil - \epsilon, \left\lfloor x_j^{LP} \right\rceil + \epsilon \right]$, where $\left\lfloor x_j^{LP} \right\rceil$ denotes the nearest integral value to x_j^{LP}, to $\left\lfloor x_j^{LP} \right\rceil$. Furthermore, because a restricted MIP will include, in most situations, binary as well as integer variables, the constraint that enforces that an improving solution differs in at least one variable now has to require that at least one of the non-kernel binary and integer variables must have a non-zero value. Finally, because AKS maintains separate ordered lists of buckets, and the number of buckets with binary variables is usually different from the number of buckets with general integer variables, it can happen that in a given iteration of the improvement phase one of the two buckets (either related to the binary variables or to the integer variables) is not available. When that happens, AKS considers the one bucket that is available.

5. Experimental analysis

This section is devoted to the presentation and discussion of the computational experiments. They were conducted on a PC Intel Xeon with 3.50 gigahertz 64-bit processor, 16.0 gigabytes of RAM and Windows 7 64-bit as Operating System. The above processor is equipped with 6 cores, and we used 12 threads in all the following experiments. AKS has been implemented in Java and compiled within NetBeans 8.0.2 and the (restricted) MIPs are solved using CPLEX 12.6.0. After preliminary experiments, we decided to use the following CPLEX parameter settings when solving the restricted MIPs: bound strengthening is always active (parameter BndStrenInd=1), and in the improvement phase finding hidden feasible solutions is active (parameter MIPEmphasis=HiddenFeas) as well as the feasibility pump heuristic with an emphasis on finding a feasible solution (parameter FPHeur=1). When CPLEX is used to solve an instance, its default parameter values are used.

In Section 5.1, we describe the data sets that we have used in the computational experiments. In Section 5.2, we discuss the control parameters of three variants of AKS that have been tested. In Section 5.3, we comment on some alternative variants of AKS that we explored, but discarded after preliminary experiments. In Section 5.4, explain the methodology adopted to evaluate the performance of AKS. Finally, the results of the computational experiments are reported in Sections 5.5 and 5.6.

5.1. Testing environment

In the computational experiments we used two data sets. The first data set comprises benchmark instances taken from the literature, whereas the second one is composed of a subset of the test problems composing the MIPLIB2010 library. Altogether, we used 137 instances to assess the performance of the AKS framework.

The first data set, henceforth called *Lazić*, comprises 29 benchmark instances and was used as a test bed by other authors, including Fischetti and Lodi (2003), Danna, Rothberg, and Le Pape (2005), Hansen, Mladenović, and Urošević (2006), and Lazić et al. (2010). In the preliminary experiments, we used this data set to calibrate the parameters of AKS (see Sections 5.2 and 5.3 for further details).

In order to validate the performance of the AKS framework, we composed an additional set of test instances selected from the MIPLIB2010 library version 5 (see Koch et al., 2011 for a detailed description of the instances composing the library). We refer to this second set of instances as the *MIPLIB2010* data set. The MIPLIB2010 library currently comprises 361 instances (the MIPLIB2010 library is publicly available at http://miplib.zib.de/miplib2010.php (last accessed: 14 June 2016)), originating from a variety of academic and industrial applications. The instances are classified according to several characteristics. Regarding the solvability of the instances, they are categorized into “Easy” (224 instances that can be solved within one hour using a MIP solver), “Hard” (58 instances that are not Easy and have been solved to optimality), and “Open” (79 instances whose optimal solution, and even the existence of a feasible solution, is unknown). We first removed from the 361 instances in the library those already included in the Lazić data set. Then, we removed all the test instances that are indicated as to be part of the “Unstable set”. These instances have bad numerical properties and can likely cause numerical troubles to the solvers (see Koch et al., 2011 for further details). The set is included in the library to test solver robustness, which is out of the scope of our experiments. Subsequently, we solved all the remaining instances in the library with CPLEX limiting the computing time to 18,000 seconds. Note that 18,000 seconds is also the time limit imposed in the experiments reported in Lazić et al. (2010). Afterward, in order to have a set of challenging instances, we removed from the set all the instances that were solved by CPLEX to proven optimality within the above time limit. We recognize that removing all instances that were solved to optimality by CPLEX within the time limit may introduce some bias in the comparison with CPLEX presented in the following sections, as we removed the instances where CPLEX performed well. However, despite this possible bias, we prefer to include in the experiments only challenging instances – where we defined challenging as being challenging for CPLEX. We also removed those instances where CPLEX did not find any feasible solution within the time limit (recall that for the Open instances no information is available, even on the existence of a feasible solution). Finally, we removed the instances that showed unstable numerical behavior. We identified these instances as those where CPLEX provided, in two different runs and with the same time limit, solutions whose values were more than 100% different from each other. The remaining set of instances contains 108 instances that compose the MIPLIB2010 data set in our computational experiments. These instances cover a wide range of applications, including crew scheduling, container loading, lot-sizing, network design, portfolio optimization, university course timetabling, and traveling salesman problems. To our knowledge, we are the first authors to extensively test these instances.

To facilitate the interpretation of the results, we divided the MIPLIB2010 data set into two groups. The group for which the value of an optimal solution is indicated on the MIPLIB2010 web-

Table 1
The number of integer, binary, and continuous variables for the instances in the test sets.

Lazić				MIPLIB2010-OS				MIPLIB2010-NOS			
Name	Int.	Bin.	Cont.	Name	Int.	Bin.	Cont.	Name	Int.	Bin.	Cont.
a1c1s1	0	192	3456	berlin_5_8_0	0	794	289	bab1	0	61,152	0
a2c1s1	0	192	3456	bg512142	0	240	552	circ10-3	0	2700	0
arki001	123	415	850	d10200	1267	733	0	d20200	819	3181	0
b1c1s1	0	288	3584	dc1c	0	8380	1659	dano3mip	0	552	13,321
b2c1s1	0	288	3584	dg012142	0	640	1440	dc1l	0	35,638	1659
biella1	0	6110	1218	g200x740i	0	740	740	ex1010-pi	0	25,200	0
danoint	0	56	465	germanrr	5286	5288	239	lectsched-1-obj	482	28,236	0
glass4	0	302	20	germany50-DBM	88	0	8101	liu	0	1089	67
markshare1	0	50	12	go19	0	441	0	methanosarcina	0	7930	0
markshare2	0	60	14	in	0	1489	1,447,585	momentum3	1	6598	6933
mkc	0	5323	2	ivu52	0	157,591	0	n15-3	780	0	152,360
net12	0	1603	12,512	janos-us-DDM	84	0	2100	n3700	0	5000	5000
nsr8k	0	32,040	6316	lotsize	0	1195	1790	n3705	0	5000	5000
nsrand_ipx	0	6620	1	maxgasflow	0	2456	4981	n370a	0	5000	5000
rail507	0	63,009	10	n3-3	366	0	8662	nag	35	1350	1499
rail2536c	0	15,284	9	neos-826841	0	3488	2028	neos-1311124	0	546	546
rail2586c	0	13,215	11	neos-941262	0	6710	2770	neos-937815	0	8876	2770
rail4284c	0	21,705	9	neos-948126	0	6965	2586	ns1456591	19	8000	380
rail4872c	0	24,645	11	neos-984165	0	6478	2405	ns1631475	211	22,470	15
roll3000	492	246	428	ns1111636	0	13,200	347,622	ns1853823	0	213,440	0
seymour	0	1372	0	ns1696083	0	7982	0	ns1854840	474	135,280	0
sp97ar	0	14,101	0	nu120-pr3	61	8540	0	ns1856153	0	11,956	42
sp97ic	0	12,497	0	opm2-z10-s2	0	6250	0	ns1904248	0	38,416	42
sp98ar	0	15,085	0	opm2-z11-s8	0	8019	0	ns2124243	0	16,447	139,636
sp98ic	0	10,894	0	opm2-z12-s14	0	10,800	0	ns4-pr3	0	103,041	320
swath	0	6724	81	opm2-z12-s7	0	10,800	0	ns4-pr9	42	0	7308
tr12-30	0	360	720	p100x588b	0	588	588	ns894236	0	9666	0
umts	72	2802	73	p6b	0	462	0	ns903616	0	21,582	0
van	0	192	12,289	pg	0	100	2600	ns930473	0	11,176	152
				pigeon-12	0	552	108	pb-simp-nonunif	0	23,848	0
				pigeon-13	0	637	117	pigeon-19	0	1273	171
				probportfolio	0	300	20	ramos3	0	2187	0
				queens-30	0	900	0	rmine14	0	32,205	0
				r80x800	0	800	800	rmine21	0	162,547	0
				rail02	0	270,869	0	rmine25	0	326,599	0
				reblock354	0	3540	0	rocII-7-11	0	15,851	250
				reblock420	0	4200	0	rocII-9-11	0	20,361	318
				rmatr200-p10	0	200	35,054	rococoC12-111000	187	8432	0
				rmatr200-p20	0	200	29,405	rvb-sub	0	33,763	2
				rmine10	0	8439	0	sct1	1268	9044	12,574
				set3-10	0	1424	2595	sct32	1332	6396	2039
				set3-15	0	1424	2595	sct5	2302	20,702	14,261
				set3-20	0	1424	2595	shipsched	0	10,549	3045
				seymour.disj-10	0	1209	0	siena1	0	11,775	1966
				shs1023	440,899	1296	2430	sing161	0	733,244	36,858
				toll-like	0	2883	0	sing245	0	220,692	14,454
				triptim2	6548	20,771	7	sing2	0	23,377	8253
				triptim3	6812	21,621	7	sing359	0	674,643	39,119
				tw-myciel4	1	759	0	stockholm	0	962	19,682
				uc-case3	0	11,256	26,493	sts405	0	405	0
				uct-subprob	0	379	1877	sts729	0	729	0
				vpphard2	0	199,999	0	t1717	0	73,885	0
				wnq-n100-mw99-14	0	10,000	0	t1722	0	36,630	0
								uc-case11	302	3898	29,934
								usAbbrv.8-25_70	0	1681	631

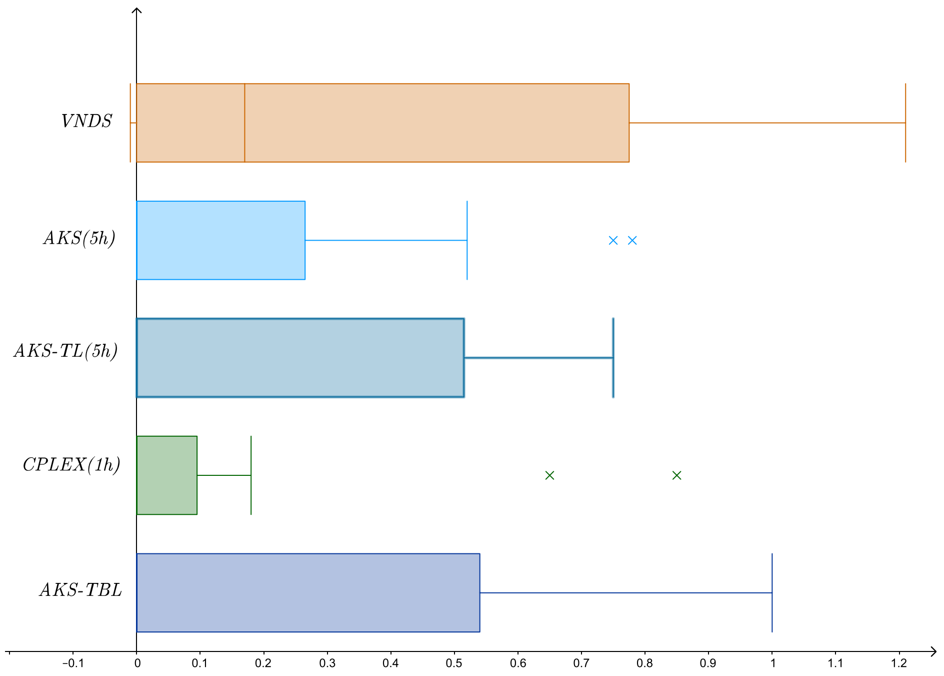
site comprises 53 instances and is from now on called *MIPLIB2010-OS*. The second group of instances, henceforth referred to as *MIPLIB2010-NOS*, contains 55 of the Open instances that were not removed for one of the reasons described above.

Table 1 reports the number of integer, binary and continuous variables, respectively, for each of the instances in the test sets.

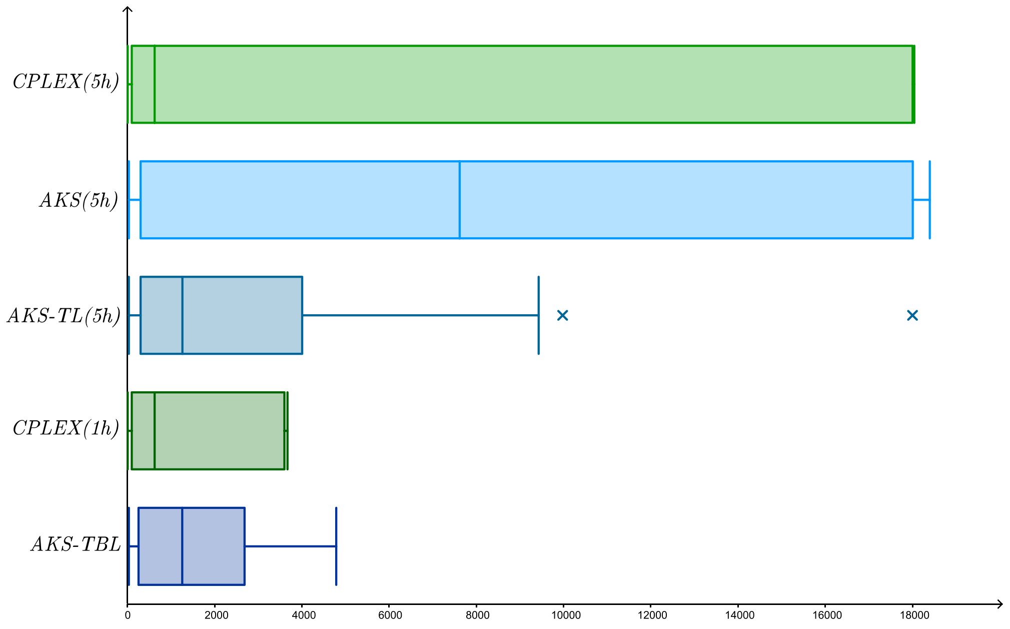
5.2. Adaptive Kernel Search: parameters and variants

An effective implementation of AKS requires the calibration of some control parameters. After some preliminary experiments, we made the following choices. Parameter w, used in GetFeasible, was set to 0.30, whereas parameter q, used when adjusting the kernel for Easy instances was set to 0.35. When adjusting the kernel for Hard instances, the tolerance ϵ was set to 10^{-5} (10 times the value of the integrality tolerance used by CPLEX). Parameters w, q, and ϵ have the same values when used for binary and integer variables. The chosen value of p is such that no variable is ever removed from the kernel, which emphasizes a search for high quality solutions over speeding up the computing time. Moreover, we set $\overline{N_b}$ to a value that ensures that all buckets are considered.

All the instances were solved with a threshold on the total computing time equal to 18,000 seconds, allocated among the restricted MIPs as follows. The root node is solved first. Note that it may happen that this first optimization problem uses all the available computing time. Then, the time allocated to the solution of the first MIP restricted to the initial kernel, i.e., problem MIP(K), is determined dividing the remaining time by $(1 + N_b)$, i.e., the num-



(a) Box-and-whisker plots of z^{UB} Gap % - Outliers smaller than -0.1% and larger than 1.25% are not reported.



(b) Box-and-whisker plots of computing times - VNDS is not reported.

Fig. 2. Box-and-whisker plots for the Lazić data set.

ber of restricted MIPs to be solved if the instance is Normal and GetFeasible is not used. The time allocated to each restricted MIP solved within GetFeasible, if used, is twice that allotted for the above MIP(K), or, if this violates the total limit of 18,000 seconds, equal to the still available computing time. The remaining time is equally allocated to all the other restricted MIPs to be solved. Every time the solution of a restricted MIP takes less time than what allotted, the time saved is equally re-allocated to all the following restricted MIPs. The thresholds used to classify the difficulty of an instance are as follows: t_{Easy} was set equal to 10 seconds and t_{Hard} was set to the time allocated to the solution of the last MIP(K) solved (the restricted MIP for the variables in the initial kernel, or the last restricted MIP solved in GetFeasible). Hence, while parameter t_{Easy} is set to the same value for all instances, parameter t_{Hard} is instance-specific and automatically determined by AKS during execution. The variant of AKS with these control parameters is denoted by *AKS*(5h).

The computing time of AKS(5h) can be substantially reduced at the expense of a small deterioration of the solution quality. It is well-known that integer programming solvers often quickly finds

Table 2
Lazić data set: a comparison of AKS, VNDS and CPLEX.

	CPLEX(5h)		CPLEX(1h)		VNDS		AKS(5h)		AKS-TL(5h)		AKS-TBL	
Inst. name	z^{UB}	CPU (seconds)	z^{UB} Gap %	CPU (seconds)	z^{UB} Gap %	CPU (seconds)	z^{UB} Gap %	CPU (seconds)	z^{UB} Gap %	CPU (seconds)	z^{UB} Gap %	CPU (seconds)
a1c1s1	11503.4	528	0.00	528	0.00	1438	0.00	*T.L.*	0.00	856	0.00	874
a2c1s1	10889.1	307	0.00	307	0.64	2357	0.00	*T.L.*	0.00	916	0.00	917
arki001	7581212.5	8	0.00	8	−0.01	4595	0.00	1369	0.00	612	0.00	612
b1c1s1	24544.3	4888	0.00	*T.L.*	0.42	5347	0.00	*T.L.*	0.00	2119	0.00	2120
b2c1s1	25687.9	16,861	0.09	*T.L.*	1.21	133	0.09	*T.L.*	0.48	2252	0.48	2253
biella1	3065029.8	130	0.00	130	0.00	4452	0.78	6743	0.64	6350	1.00	2860
danoint	65.7	234	0.00	234	0.01	3360	0.00	427	0.00	428	0.00	427
glass4	1200012600.0	459	0.00	459	29.17	3198	0.00	54	0.00	55	0.00	54
markshare1	2.0	*T.L.*	50.00	*T.L.*	50.00	12,592	−50.00	*T.L.*	0.00	2303	0.00	2515
markshare2	4.0	*T.L.*	50.00	*T.L.*	100.00	13,572	0.00	*T.L.*	50.00	2525	50.00	2435
mkc	−563.8	*T.L.*	0.00	*T.L.*	0.34	9003	0.00	*T.L.*	0.00	1568	0.00	1553
net12	214.0	69	0.00	69	0.00	130	0.00	104	0.00	103	0.00	99
nsr8k	18148119.0	*T.L.*	0.18	*T.L.*	14.35	*T.L.*[a]	4.78	*T.L.*	4.85	*T.L.*	7.88	3762
nsrand_ipx	51200.0	34	0.00	34	0.00	10,595	0.00	578	0.00	577	0.00	560
rail507	174.0	27	0.00	27	0.00	2150	0.00	147	0.00	147	0.00	115
rail2536c	689.0	18	0.00	18	0.00	13,284	0.00	43	0.00	43	0.00	43
rail2586c	955.0	*T.L.*	0.10	*T.L.*	0.21	12,822	0.21	*T.L.*	0.52	8096	0.52	3612
rail4284c	1069.0	*T.L.*	0.65	*T.L.*	0.56	17,875	0.75	*T.L.*	0.75	8799	0.56	3631
rail4872c	1538.0	*T.L.*	0.85	*T.L.*	0.91	8349	0.52	*T.L.*	0.52	9429	0.65	3635
roll3000	12890.0	11	0.00	11	0.31	2585	0.00	35	0.00	35	0.00	35
seymour	423.0	*T.L.*	0.00	*T.L.*	0.47	9151	0.00	*T.L.*	0.24	883	0.24	886
sp97ar	661059959.7	*T.L.*	0.00	*T.L.*	0.17	16,933	0.32	180	0.32	179	0.35	85
sp97ic	427684487.7	3188	0.00	3188	0.91	2014	0.32	5673	0.51	3869	0.32	2304
sp98ar	529740623.2	312	0.00	312	0.09	7173	0.03	7619	0.03	3662	0.03	2293
sp98ic	449144758.4	172	0.00	172	0.00	2724	0.02	7646	0.17	1262	0.17	1258
swath	467.4	*T.L.*	0.00	*T.L.*	0.00	901	3.06	4816	3.06	4147	3.06	3011
tr12-30	130596.0	71	0.00	71	0.00	7617	0.00	110	0.00	97	0.00	93
umts	30090583.0	625	0.00	625	0.00	6837	0.00	427	0.00	428	0.00	397
van	4.6	*T.L.*	5.46	*T.L.*	0.00	11,535	0.00	*T.L.*	0.00	9979	11.25	4789
Average		*7795*	*3.70*	*1831*	*6.89*	*8496*	*−1.35*	*9367*	*2.14*	*3094*	*2.64*	*1628*

[a] *For instance* nsr8k, Lazić et al. (2010) *increased the time limit up to 15 hours, since their approach required more than 13,000 seconds to find the first feasible solution.*

an optimal solution, but then spends a large amount of time proving its optimality. Moreover, by design some of the restricted MIPs will be infeasible, and it is also well-known that integer programming solvers struggle to prove infeasibility. To limit the negative impact of these two situations, we impose the following limits. For the MIP restricted to the initial kernel (see Step 3 in Algorithm 2) and each restricted MIP solved in the improvement phase (see Step 14), we limit the time spent without improving the incumbent solution to 1200 seconds, and the time without finding a feasible solution to 700 seconds. For Easy instances, we use slightly different limits for any restricted problem solved in Algorithm 4, namely 1500 and 600 seconds, respectively, and we limit the number of nodes explored without improving the incumbent solution to 80,000,000 and the number of nodes explored without finding a feasible solution to 10,000,000. None of the limits apply to the solution of the root node and any restricted MIP solved in GetFeasible. The variant of AKS with these additional time and node limits is denoted by *AKS-TL(5h)*.

Finally, we also tested the following variant of AKS-TL(5h). We limit the solution of the root node to 7200 seconds and we limit the remainder of AKS to 3600 seconds. Additionally, we limit the number of buckets analyzed to 5 (parameter $\overline{N_b}$). We denote this variant by *AKS-TLB*.

The reason for testing these three AKS variants is to demonstrate the flexibility of the AKS heuristic framework, i.e., by imposing some simple limits and by restricting the number of buckets, we control the trade-off between solution quality and computing time.

5.3. Adaptive Kernel Search: alternative variants

In this section, we briefly describe and discuss some other variants of AKS that we tested in our preliminary experiments and that were outperformed by the AKS variants we decided to use in our final tests.

In a first alternative variant, the LP relaxation was used in the initialization phase instead of the root node relaxation, as was done in the original KS. By using the root node relaxation, we take advantage of the preprocessing and cut generation techniques embedded in CPLEX and the resulting solution (usually) provides more reliable information for the definition of the initial kernel and buckets.

In a second alternative variant, we used the solutions to the LP relaxations of some of the first few nodes explored in the search tree to determine the initial kernel rather than the root node relaxation. More precisely, we considered the LP relaxations corresponding to the first $\bar{n}$ nodes in which the lower bound improved (we tested several values of $\bar{n}$). For each of the $\bar{n}$ nodes, the variables were then sorted to create an ordered list (or two lists if the instance included both binary and integer variables) as described in Section 3. Once $\bar{n}$ nodes were explored, a final ranking of the variables was created based on the ordered lists associated with the explored nodes. The initial kernel and buckets were constructed based on this final ranking. However, we found that there were no remarkable differences in terms of the quality of the best solution found compared to the version that considers only the root node relaxation (and the computing time increased).

In a final alternative variant, we used the optimality gap at the root node to assess the difficulty of an instance. Specifically, if the optimality gap was smaller than a given limit (we tried different values), then the instance was classified as Easy. If the optimality gap was larger than a given limit (here too we tested different values) or no feasible integer solution was found at the root node (and, therefore, no optimality gap was computable), then the instance was categorized as Hard. All the remaining instances were considered Normal. The results of this variant were worse, in

Table 3
MIPLIB2010-OS data set (known optimum): a comparison of AKS, VNDS and CPLEX.

	CPLEX(5h)			CPLEX(1h)		VNDS(5h)		VNDS(1h)		AKS(5h)		AKS-TL(5h)		AKS-TBL	
Inst. name	z^{UB}	z^* Gap %	CPU (seconds)	z^{UB} Gap %	CPU (seconds)	z^{UB} Gap %	CPU (seconds)	z^{UB} Gap %	CPU (seconds)	z^{UB} Gap %	CPU (seconds)	z^{UB} Gap %	CPU (seconds)	z^{UB} Gap %	CPU (sec.)
berlin_5_8_0	62.0	0.00	*T.L.*	0.00	*T.L.*	0.00	*T.L.*	0.00	*T.L.*	0.00	*T.L.*	0.00	2139	0.00	2139
bg512142	184234.0	0.02	*T.L.*	1.95	*T.L.*	1.20	*T.L.*	2.15	*T.L.*	1.51	*T.L.*	1.66	4356	1.66	2508
d10200	12436.0	0.05	*T.L.*	0.00	*T.L.*	*N/A*	*N/A*	*N/A*	*N/A*	−0.03	*T.L.*	−0.02	3786	0.00	2615
dc1c	1778296.5	0.59	*T.L.*	0.14	*T.L.*	0.39	*T.L.*	1.33	*T.L.*	1.39	*T.L.*	1.23	17,291	3.24	3612
dg012142	2549755.3	10.82	*T.L.*	4.70	*T.L.*	−3.98	*T.L.*	28.80	*T.L.*	1.54	*T.L.*	2.33	2638	2.33	2628
g200x740i	30204.0	0.39	*T.L.*	0.00	*T.L.*	11.69	*T.L.*	11.69	*T.L.*	0.57	*T.L.*	0.57	833	0.57	833
germanrr	47095891.1	0.00	*T.L.*	0.00	*T.L.*	*N/A*	*N/A*	*N/A*	*N/A*	0.13	*T.L.*	0.13	5005	0.13	3208
germany50-DBM	473840.0	0.00	*T.L.*	0.51	*T.L.*	*N/A*	*N/A*	*N/A*	*N/A*	0.60	*T.L.*	0.69	3497	0.60	2575
go19	84.0	0.00	*T.L.*	0.00	*T.L.*	1.19	*T.L.*	1.19	*T.L.*	0.00	*T.L.*	0.00	1912	0.00	1912
in	411.0	608.62	*T.L.*	4.14	*T.L.*	*Fails*[a]	*T.L.*	*Fails*[a]	*T.L.*	1.46	*T.L.*	1.46	*T.L.*	1.46	10714
ivu52	481.1	0.02	*T.L.*	0.31	*T.L.*	1.71	*T.L.*	*Fails*[a]	*T.L.*	0.12	15,740	0.56	8570	0.91	3743
janos-us-DDM	1492738.0	0.00	*T.L.*	0.01	*T.L.*	*N/A*	*N/A*	*N/A*	*N/A*	0.00	*T.L.*	0.00	1665	0.00	1663
lotsize	1480242.0	0.00	*T.L.*	0.00	*T.L.*	2.71	*T.L.*	6.02	*T.L.*	0.00	*T.L.*	0.00	3279	0.00	3281
maxgasflow	−44562045.0	0.01	*T.L.*	0.00	*T.L.*	1.74	*T.L.*	1.74	*T.L.*	−0.01	*T.L.*	−0.01	3132	−0.01	3122
n3-3	15921.0	0.04	*T.L.*	0.41	*T.L.*	*N/A*	*N/A*	*N/A*	*N/A*	0.30	*T.L.*	2.06	4350	1.19	3620
neos-826841	29.0	0.00	*T.L.*	0.00	*T.L.*	0.00	*T.L.*	0.00	*T.L.*	0.00	50	0.00	48	0.00	48
neos-941262	2791.0	0.00	*T.L.*	0.00	*T.L.*	0.00	*T.L.*	0.25	*T.L.*	3.33	14,579	3.33	11,117	5.59	3635
neos-948126	2608.0	0.04	*T.L.*	0.19	*T.L.*	1.15	*T.L.*	1.73	*T.L.*	1.46	3580	1.46	3584	1.88	2625
neos-984165	2208.0	0.91	*T.L.*	0.14	*T.L.*	0.68	*T.L.*	0.68	*T.L.*	1.31	14,623	0.72	7777	1.13	3672
ns1111636	166.0	2.47	*T.L.*	0.00	*T.L.*	7.23	*T.L.*	7.23	*T.L.*	0.00	*T.L.*	0.00	*T.L.*	1.20	4417
ns1696083	49.0	8.89	*T.L.*	6.12	*T.L.*	−4.08	*T.L.*	−4.08	*T.L.*	−8.16	10,917	−8.16	10,935	−2.04	3949
nu120-pr3	28585.0	1.62	*T.L.*	0.02	*T.L.*	*N/A*	*N/A*	*N/A*	*N/A*	6.00	14,597	6.00	4722	6.00	3218
opm2-z10-s2	−33337.0	1.45	*T.L.*	0.00	*T.L.*	22.39	*T.L.*	22.39	*T.L.*	−1.40	*T.L.*	−1.40	9882	6.34	3911
opm2-z11-s8	−43476.0	0.02	*T.L.*	2.50	*T.L.*	21.97	*T.L.*	21.97	*T.L.*	0.78	*T.L.*	2.30	9599	17.53	4286
opm2-z12-s14	−63860.0	0.67	*T.L.*	13.87	*T.L.*	23.45	*T.L.*	31.68	*T.L.*	0.17	*T.L.*	−0.64	*T.L.*	15.09	3836
opm2-z12-s7	−64212.0	1.99	*T.L.*	13.57	*T.L.*	26.47	*T.L.*	26.47	*T.L.*	0.31	*T.L.*	0.31	*T.L.*	17.16	4929
p100x588b	47878.0	0.00	*T.L.*	0.00	*T.L.*	3.55	*T.L.*	6.13	*T.L.*	0.96	*T.L.*	0.76	3478	0.76	3154
p6b	−63.0	0.00	*T.L.*	0.00	*T.L.*	4.76	*T.L.*	4.76	*T.L.*	0.00	*T.L.*	1.59	1912	1.59	1911
pg	−8674.3	0.00	*T.L.*	0.00	*T.L.*	4.01	*T.L.*	4.01	*T.L.*	0.07	*T.L.*	0.07	1919	0.07	1919
pigeon-12	−11000.0	0.00	*T.L.*	0.00	*T.L.*	72.73	*T.L.*	72.73	*T.L.*	0.00	*T.L.*	0.00	1537	0.00	1536
pigeon-13	−12000.0	0.00	*T.L.*	0.00	*T.L.*	66.67	*T.L.*	66.67	*T.L.*	0.00	*T.L.*	0.00	1561	0.00	1560
probportfolio	16.7	0.00	*T.L.*	0.85	*T.L.*	3.71	*T.L.*	3.71	*T.L.*	2.46	599	2.46	600	2.46	596
queens-30	−39.0	2.50	*T.L.*	0.00	*T.L.*	33.33	*T.L.*	33.33	*T.L.*	0.00	*T.L.*	0.00	2022	0.00	2000
r80x800	5332.0	0.00	*T.L.*	0.00	*T.L.*	0.69	*T.L.*	0.73	*T.L.*	0.06	*T.L.*	0.41	4973	0.41	3611
rail02	−195.0	2.69	*T.L.*	51.58	*T.L.*	−0.64	*T.L.*	0.45	*T.L.*	−1.69	*T.L.*	−2.76	11,172	−1.61	2153
reblock354	−39277181.5	0.01	*T.L.*	0.01	*T.L.*	1.63	*T.L.*	1.63	*T.L.*	0.00	*T.L.*	0.03	4777	0.03	3031
reblock420	−517792519.4	0.00	*T.L.*	0.00	*T.L.*	4.02	*T.L.*	4.02	*T.L.*	0.00	9425	0.00	5442	0.00	1858
rmatr200-p10	2017.0	0.00	*T.L.*	0.89	*T.L.*	0.84	*T.L.*	0.84	*T.L.*	0.00	*T.L.*	1.34	1337	0.00	1337
rmatr200-p20	838.0	0.12	*T.L.*	0.36	*T.L.*	−0.12	*T.L.*	−0.12	*T.L.*	0.48	*T.L.*	1.67	754	1.67	884
rmine10	−1913.3	0.03	*T.L.*	0.04	*T.L.*	1.06	*T.L.*	1.06	*T.L.*	−0.01	*T.L.*	−0.01	5094	−0.01	2578
set3-10	195772.3	5.72	*T.L.*	1.89	*T.L.*	27.58	*T.L.*	44.38	*T.L.*	0.88	*T.L.*	5.30	1479	5.30	1456
set3-15	132860.9	6.39	*T.L.*	6.99	*T.L.*	20.60	*T.L.*	34.10	*T.L.*	−5.71	*T.L.*	−0.32	1828	−0.32	1744
set3-20	165728.8	3.93	*T.L.*	5.56	*T.L.*	21.66	*T.L.*	21.66	*T.L.*	0.18	*T.L.*	3.37	3186	10.40	2436
seymour.disj-10	287.0	0.00	*T.L.*	0.00	*T.L.*	0.00	*T.L.*	0.00	*T.L.*	0.00	*T.L.*	0.00	2641	0.00	2622
shs1023	13137.9	0.01	*T.L.*	0.49	*T.L.*	*N/A*	*N/A*	*N/A*	*N/A*	0.00	*T.L.*	0.13	12167	0.14	5952
toll-like	611.0	0.16	*T.L.*	0.00	*T.L.*	2.95	*T.L.*	3.60	*T.L.*	−0.16	*T.L.*	−0.16	1550	−0.16	1523
triptim2	12.0	0.00	*T.L.*	0.01	*T.L.*	*N/A*	*N/A*	*N/A*	*N/A*	0.02	8005	0.02	6325	0.17	4010
triptim3	13.5	0.00	*T.L.*	0.00	*T.L.*	*N/A*	*N/A*	*N/A*	*N/A*	0.00	*T.L.*	0.00	10,583	0.00	3962
tw-myciel4	10.0	0.00	*T.L.*	0.00	*T.L.*	*N/A*	*N/A*	*N/A*	*N/A*	0.00	*T.L.*	0.00	618	0.00	619
uc-case3	7204.9	0.00	*T.L.*	0.00	*T.L.*	0.02	*T.L.*	0.02	*T.L.*	0.00	11,355	0.00	6503	0.00	3108
uct-subprob	315.0	0.32	*T.L.*	0.00	*T.L.*	0.95	*T.L.*	2.22	*T.L.*	1.27	*T.L.*	1.27	1983	1.27	1983
vpphard2	82.0	1.23	*T.L.*	0.00	*T.L.*	54.88	*T.L.*	54.88	*T.L.*	−1.22	*T.L.*	−1.22	5014	0.00	4347
wnq-n100-mw99-14	259.0	0.00	*T.L.*	0.00	*T.L.*	5.41	*T.L.*	6.18	*T.L.*	1.93	3741	1.93	3645	1.93	2233
Average		*12.49*	*18,083*	*2.21*	*3607*					*0.21*	*16,008*	*0.58*	*5590*	*2.00*	*2846*
Average no int.		*15.35*	*18,096*	*2.69*	*3608*	*10.62*[b]	*19241*	*12.88*[b]	*4582*	*0.09*	*15,833*	*0.50*	*5664*	*2.28*	*2776*

[a] The method did not find any feasible solution within the time limit.
[b] Not counting the cases of failure.

terms of solution quality, compared to using the time spent solving problem MIP(K) as indicator.

5.4. Assessing performance

In this section, we describe the methodology adopted to assess the performance of the three AKS variants described in Section 5.2.

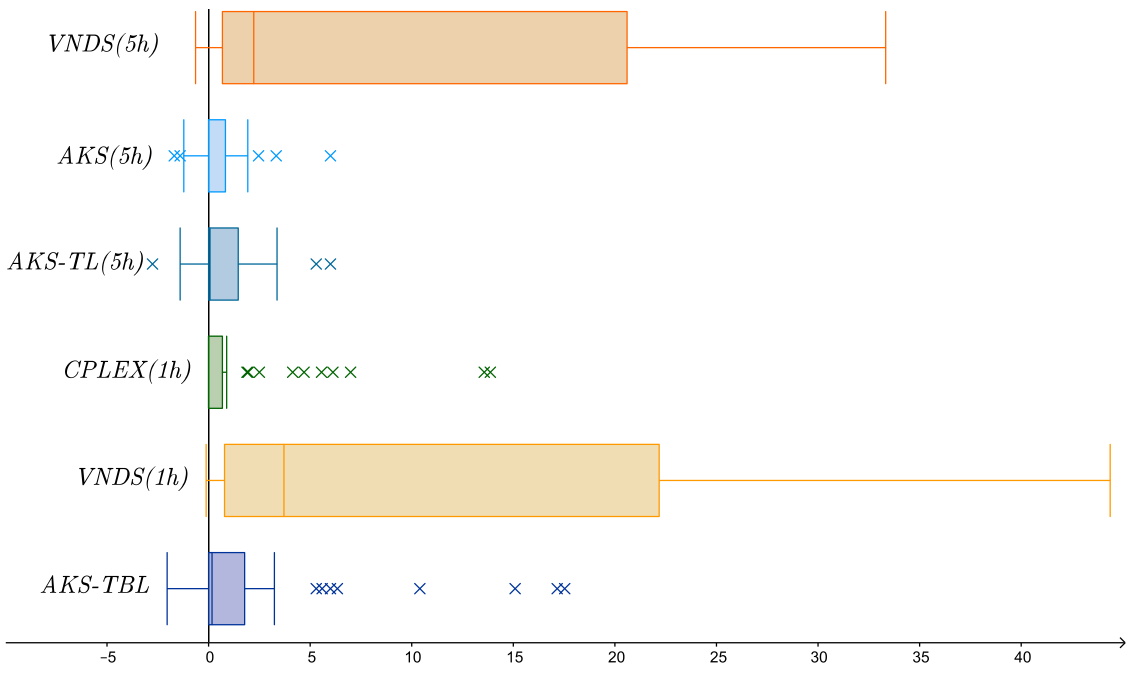
The quality of a solution produced by an AKS variant is evaluated with respect to the best solution found by CPLEX within 5h. In the following, *CPLEX*(5h) refers to CPLEX run with a time limit equal to 18,000 seconds (all other CPLEX parameters have their default values). The value of the best solution found using CPLEX(5h) is denoted by z^{UB}. For a better assessment of the performance of AKS-TLB, we also report the results obtained by CPLEX with a time limit of 3600 seconds, denoted by CPLEX(1h). For methods other than CPLEX(5h), we report the statistic z^{UB} *Gap %* which measures the deterioration (positive value) or the improvement (negative value) with respect to z^{UB}. This statistic is computed as $100 \times \frac{(z^H - z^{UB})}{|z^{UB}|}$, where z^H is the value of the solution found by the method under consideration. For the instances of the MIPLIB2010-OS data set (for which the value of an optimal solution is known), we also compute the deterioration of the solutions obtained by

Table 4
MIPLIB2010-NOS data set (unknown optimum): a comparison of AKS, VNDS and CPLEX.

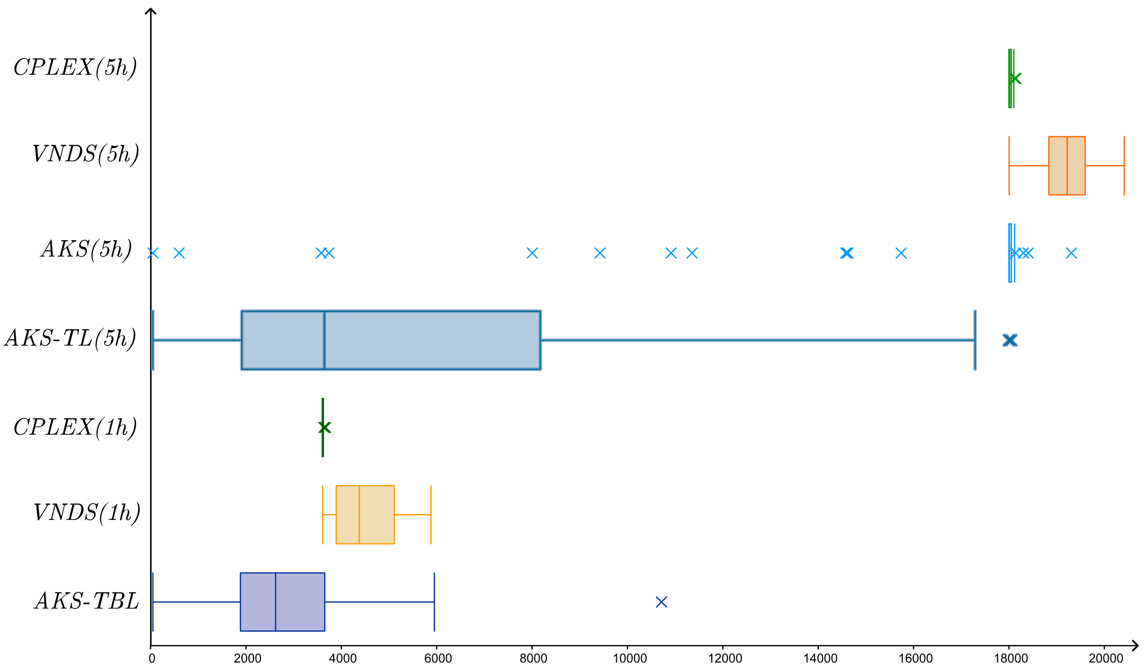
	CPLEX(5h)			CPLEX(1h)		VNDS(5h)		VNDS(1h)		AKS(5h)		AKS-TL(5h)		AKS-TBL	
Inst. name	z^{UB}	LB Gap %	CPU (seconds)	z^{UB} Gap %	CPU (seconds)	z^{UB} Gap %	CPU (seconds)	z^{UB} Gap %	CPU (seconds)	z^{UB} Gap %	CPU (seconds)	z^{UB} Gap %	CPU (seconds)	z^{UB} Gap %	CPU (seconds)
bab1	−218764.9	0.94	*T.L.*	0.00	*T.L.*	1.07	*T.L.*	1.07	*T.L.*	0.06	9443	0.06	2246	0.06	2268
circ10-3	382.0	63.35	*T.L.*	*Fails*[a]	*T.L.*	**−13.09**	*T.L.*	*Fails*[a]	*T.L.*	**−2.62**	*T.L.*	**−2.62**	*T.L.*	*Fails*[a]	3907
d20200	12251.0	0.16	*T.L.*	0.00	*T.L.*	*N/A*	*N/A*	*N/A*	*N/A*	0.06	*T.L.*	0.06	7142	0.06	3613
dano3mip	679.8	14.90	*T.L.*	2.22	*T.L.*	9.03	*T.L.*	9.03	*T.L.*	0.98	*T.L.*	3.61	2590	3.61	2696
dc1l	1781287.1	1.83	*T.L.*	0.18	*T.L.*	**−0.48**	*T.L.*	0.22	*T.L.*	3.95	*T.L.*	3.95	*T.L.*	5.89	3634
ex1010-pi	240.0	7.11	*T.L.*	0.42	*T.L.*	0.42	*T.L.*	0.42	*T.L.*	1.25	*T.L.*	1.25	17,723	1.25	3618
lectsched-1-obj	76.0	34.60	*T.L.*	6.58	*T.L.*	*N/A*	*N/A*	*N/A*	*N/A*	**−3.95**	*T.L.*	**−3.95**	*T.L.*	0.00	3626
liu	1108.0	49.46	*T.L.*	1.08	*T.L.*	3.25	*T.L.*	3.97	*T.L.*	0.72	*T.L.*	**−0.18**	3302	0.36	3090
methanosarcina	2764.0	41.40	*T.L.*	0.00	*T.L.*	**−0.29**	*T.L.*	1.05	*T.L.*	**−1.05**	*T.L.*	**−0.94**	4771	**−0.90**	4711
momentum3	381815.6	74.95	*T.L.*	18.32	*T.L.*	*N/A*	*N/A*	*N/A*	*N/A*	**−12.94**	*T.L.*	5.03	3059	8.15	3592
n15-3	46294.0	27.76	*T.L.*	0.00	*T.L.*	*N/A*	*N/A*	*N/A*	*N/A*	**−2.00**	*T.L.*	**−2.00**	4437	**−2.00**	4593
n3700	1252509.0	15.52	*T.L.*	1.83	*T.L.*	2.43	*T.L.*	2.43	*T.L.*	3.26	*T.L.*	**−1.73**	10,509	**−1.29**	3619
n3705	1254168.0	15.84	*T.L.*	2.48	*T.L.*	4.60	*T.L.*	4.60	*T.L.*	3.59	*T.L.*	3.59	9882	3.59	3621
n370a	1261899.0	15.01	*T.L.*	1.89	*T.L.*	3.30	*T.L.*	3.30	*T.L.*	1.75	*T.L.*	1.75	11,294	1.75	3619
nag	1200.0	61.24	*T.L.*	0.00	*T.L.*	*N/A*	*N/A*	*N/A*	*N/A*	**−6.25**	9941	**−6.25**	4603	8.75	3603
neos-1311124	−181.0	0.55	*T.L.*	0.00	*T.L.*	7.18	*T.L.*	7.18	*T.L.*	0.00	*T.L.*	0.00	1013	0.00	1014
neos-937815	2845.0	0.28	*T.L.*	0.04	*T.L.*	0.32	*T.L.*	0.32	*T.L.*	0.56	1008	0.56	1009	0.56	1008
ns1456591	1118.0	63.04	*T.L.*	0.00	*T.L.*	*N/A*	*N/A*	*N/A*	*N/A*	61.94	*T.L.*	61.94	1810	61.94	1811
ns1631475	15600.0	87.46	*T.L.*	15.38	*T.L.*	*N/A*	*N/A*	*N/A*	*N/A*	**−9.62**	*T.L.*	6.73	3324	6.73	3320
ns1853823	310000.0	80.50	*T.L.*	0.00	*T.L.*	*Fails*[a]	*T.L.*	*Fails*[a]	*T.L.*	**−61.94**	*T.L.*	**−60.65**	*T.L.*	**−54.84**	10332
ns1854840	316000.0	61.25	*T.L.*	0.00	*T.L.*	*N/A*	*N/A*	*N/A*	*N/A*	**−49.37**	7860	**−49.37**	8044	**−49.37**	6093
ns1856153	71.2	100.00	*T.L.*	6.78	*T.L.*	9.11	*T.L.*	9.11	*T.L.*	2.23	*T.L.*	3.53	2961	3.53	2966
ns1904248	90.7	100.00	*T.L.*	0.00	*T.L.*	*Fails*[a]	*T.L.*	*Fails*[a]	*T.L.*	0.00	*T.L.*	0.00	*T.L.*	0.00	7202
ns2124243	77540.0	3.84	*T.L.*	0.00	*T.L.*	1.37	*T.L.*	1.32	*T.L.*	1.12	*T.L.*	1.12	15,154	3.04	3744
ns4-pr3	36140.0	0.03	*T.L.*	0.00	*T.L.*	0.01	*T.L.*	0.01	*T.L.*	0.00	*T.L.*	0.00	2740	0.00	2745
ns4-pr9	35231.0	0.04	*T.L.*	0.00	*T.L.*	*N/A*	*N/A*	*N/A*	*N/A*	0.00	*T.L.*	0.00	1212	0.00	1212
ns894236	17.0	11.76	*T.L.*	0.00	*T.L.*	0.00	*T.L.*	0.00	*T.L.*	0.00	*T.L.*	0.00	696	0.00	690
ns903616	19.0	5.26	*T.L.*	0.00	*T.L.*	0.00	*T.L.*	5.26	*T.L.*	0.00	*T.L.*	0.00	2255	0.00	2343
ns930473	834428.0	100.00	*T.L.*	6.77	*T.L.*	3.42	*T.L.*	3.42	*T.L.*	**−0.01**	*T.L.*	3.55	5334	3.56	3752
pb-simp-nonunif	42.0	16.67	*T.L.*	19.05	*T.L.*	14.29	*T.L.*	14.29	*T.L.*	0.00	*T.L.*	0.00	3995	0.00	3934
pigeon-19	−18000.0	5.56	*T.L.*	0.00	*T.L.*	83.33	*T.L.*	83.33	*T.L.*	0.00	*T.L.*	0.00	3021	0.00	3023
ramos3	222.0	34.17	*T.L.*	1.80	*T.L.*	11.71	*T.L.*	11.71	*T.L.*	2.25	*T.L.*	3.15	6073	3.60	3775
rmine14	−4276.3	0.54	*T.L.*	74.35	*T.L.*	1.31	*T.L.*	1.31	*T.L.*	**−0.03**	*T.L.*	3.91	7359	3.22	4920
rmine21	−2645.0	303.75	*T.L.*	0.00	*T.L.*	*Fails*[a]	*T.L.*	*Fails*[a]	*N/A*	0.00	*T.L.*	0.00	*T.L.*	0.00	7235
rmine25	−2553.5	1342.90	*T.L.*	0.00	*T.L.*	*Fails*[a]	*T.L.*	*Fails*[a]	*T.L.*	0.00	*T.L.*	0.00	*T.L.*	0.00	7326
rocII-7-11	−6.7	75.73	*T.L.*	0.04	*T.L.*	90.01	*T.L.*	90.01	*T.L.*	**−14.55**	*T.L.*	15.21	3561	15.21	3572
rocII-9-11	−4.7	150.36	*T.L.*	0.03	*T.L.*	22.90	*T.L.*	22.90	*T.L.*	**−62.60**	*T.L.*	22.52	5204	22.52	3647
rococoC12-111000	36221.0	7.89	*T.L.*	0.04	*T.L.*	*N/A*	*N/A*	*N/A*	*N/A*	**−1.34**	2344	**−1.34**	2023	**−1.34**	1988
rvb-sub	18.7	37.29	*T.L.*	46.92	*T.L.*	12.55	*T.L.*	175.69	*T.L.*	**−13.25**	13878	**−13.25**	2172	**−13.25**	2176
sct1	−181.5	11.81	*T.L.*	0.05	*T.L.*	*N/A*	*N/A*	*N/A*	*N/A*	0.98	*T.L.*	1.65	15,206	1.11	3624
sct32	−17.2	4.58	*T.L.*	2.75	*T.L.*	*N/A*	*N/A*	*N/A*	*N/A*	37.58	*T.L.*	37.16	11,787	40.34	3617
sct5	−228.1	0.03	*T.L.*	0.00	*T.L.*	*N/A*	*N/A*	*N/A*	*N/A*	**−0.01**	15720	0.01	17443	4.99	1646
shipsched	115180.0	42.01	*T.L.*	0.00	*T.L.*	42.05	*T.L.*	69.81	*T.L.*	16.04	*T.L.*	28.86	5143	28.86	3638
siena1	10885594.6	6.39	*T.L.*	42.44	*T.L.*	23.33	*T.L.*	41.37	*T.L.*	4.40	*T.L.*	6.85	17,430	98.80	3624
sing161	19249035.4	0.12	*T.L.*	0.02	*T.L.*	0.81	*T.L.*	1.16	*T.L.*	0.08	8404	0.11	8421	0.26	8416
sing245	25688258.7	1.13	*T.L.*	0.16	*T.L.*	3.06	*T.L.*	4.22	*T.L.*	**−0.15**	5731	**−0.15**	4763	0.03	2629
sing2	17359206.4	0.93	*T.L.*	0.19	*T.L.*	0.66	*T.L.*	0.66	*T.L.*	**−0.09**	*T.L.*	**−0.07**	10,096	0.12	3659
sing359	23039853.2	1.23	*T.L.*	−0.11	*T.L.*	0.21	*T.L.*	7.09	*T.L.*	**−0.39**	10,109	**−0.26**	9260	**−0.32**	9008
stockholm	127.0	24.47	*T.L.*	1.57	*T.L.*	1.57	*T.L.*	4.72	*T.L.*	0.79	*T.L.*	0.79	4463	0.79	3633
sts405	341.0	38.86	*T.L.*	0.29	*T.L.*	0.88	*T.L.*	0.88	*T.L.*	1.47	*T.L.*	1.47	1510	1.47	1548
sts729	650.0	58.38	*T.L.*	0.31	*T.L.*	0.00	*T.L.*	0.00	*T.L.*	**−0.31**	*T.L.*	**−0.31**	5831	0.31	4025
t1717	182435.0	24.85	*T.L.*	2.11	*T.L.*	2.29	*T.L.*	9.17	*T.L.*	9.79	*T.L.*	9.79	3744	9.79	3145
t1722	116024.0	11.25	*T.L.*	0.90	*T.L.*	3.41	*T.L.*	3.41	*T.L.*	5.55	*T.L.*	6.66	4528	5.26	3070
uc-case11	622232.1	0.75	*T.L.*	0.03	*T.L.*	*N/A*	*N/A*	*N/A*	*N/A*	0.00	*T.L.*	0.08	4071	0.00	3678
usAbbrv.8-25_70	120.0	14.46	*T.L.*	0.00	*T.L.*	2.50	*T.L.*	5.83	*T.L.*	0.83	*T.L.*	0.83	1696	0.83	1726
Average			*18,041*	*4.76*[b]	*3606*					*−1.48*	*16,370*	*1.69*	*7493*	*4.20*[b]	*3715*
Average no int.			*18,036*	*5.34*[b]	*3606*	*9.42*[b]	*19,546*	*16.67*[b]	*4477*	*−2.35*	*16,689*	*1.05*	*7559*	*3.69*[b]	*3861*

[a] The method did not find any feasible solution within the time limit.
[b] Not counting the cases of failure.

CPLEX(5h) with respect to the optimal value published on the MIPLIB2010 website, which we denote by z^*. This deterioration is referred to as z^* *Gap %* and is computed as $100 \times \frac{(z^{UB}-z^*)}{|z^*|}$. The reason for reporting z^* Gap % is to show how close the best solution found by CPLEX(5h) is to the optimal solution, and, consequently, to be able to better assess the performance of variants of AKS. Because for the instances of the MIPLIB2010-NOS data set the optimum value is not known, we report for CPLEX(5h) the gap with respect to the best lower bound found by CPLEX(5h), denoted by *LB Gap %* and computed as $100 \times \frac{(z^{UB}-z^{LB})}{|z^{UB}|}$, where z^{LB} is the best lower bound found by CPLEX(5h). The reason for reporting this statistic is to identify the instances for which CPLEX(5h) found a near-optimal, or even optimal, solution, but for which it was not able to prove the optimality. Obviously, we cannot expect that, on these instances, any heuristic significantly outperforms CPLEX(5h). Regarding computing times, we show for each method the time, measured in seconds and denoted by *CPU* (seconds). We indicate with *T.L.* the cases where the time limit was reached.



(a) Box-and-whisker plots of z^{UB} Gap % - Outliers smaller than -2.5% and larger than 45.0% are not reported.

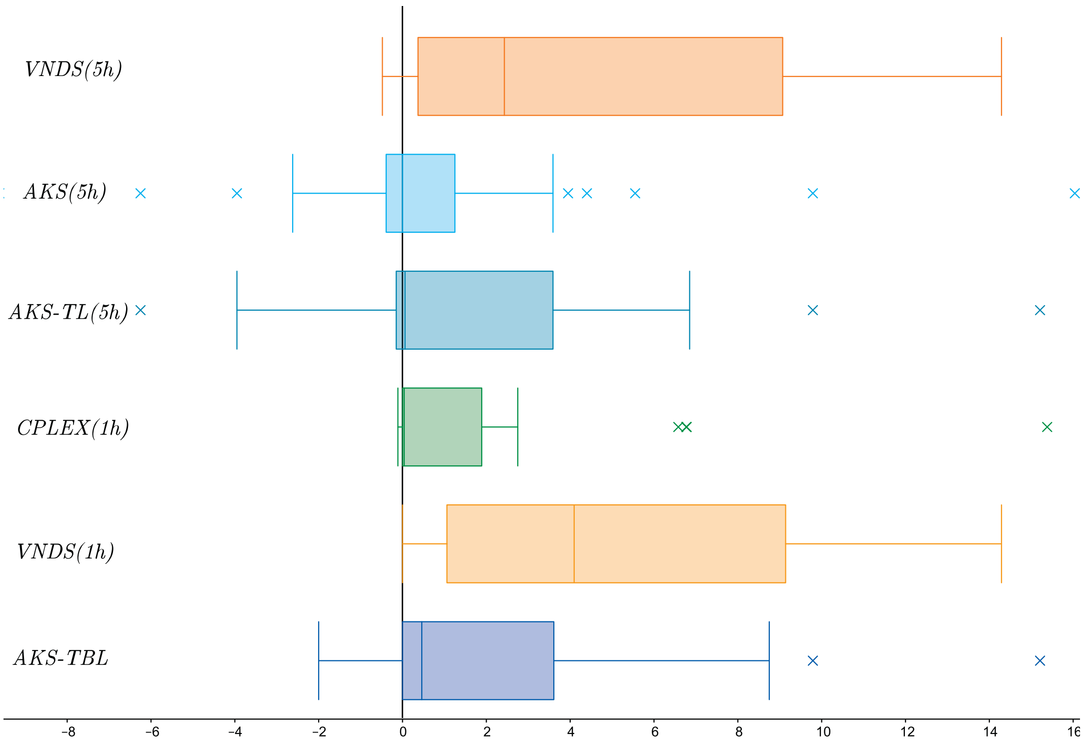


(b) Box-and-whisker plots of computing times.

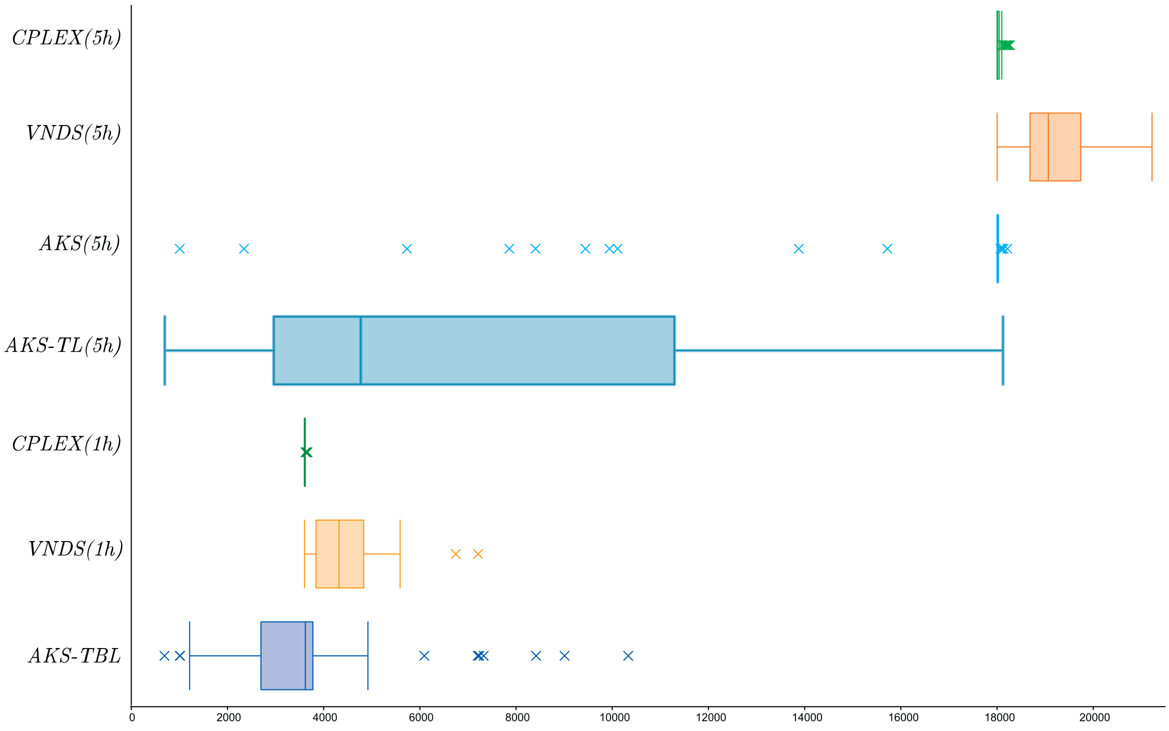
Fig. 3. Box-and-whisker plots for the MIPLIB2010-OS data set (known optimum).

To assess the performance of AKS against heuristics presented in the literature, we used the VNDS proposed in Lazić et al. (2010) as benchmark. To the best of our knowledge, it is the current state-of-the-art heuristic for 0–1 MIPs. In fact, as mentioned above, the authors show that, on the 29 instances in the Lazić data set, the VNDS outperforms Local Branching by Fischetti and Lodi (2003), Relaxation Induced Neighborhood Search by Danna, Rothberg, and Le Pape (2005), as well as the VNSB by Hansen, Mladenović, and Urošević (2006).

As the VNDS was tested only on the 29 instances in the Lazić data set, we organize the results of our computational study as follows. For the Lazić instances, we compare to the results reported in Lazić et al. (2010). As a consequence, for these instances the comparison between their method and the other methods has to be done with some care, since their experiments were conducted on less powerful hardware, a different version of CPLEX was used, and their algorithm was coded in another programming language. More precisely, Lazić et al. (2010) used a computer with a 2.4 gi-



(a) Box-and-whisker plots of z^{UB} Gap % - Outliers smaller than -8% and larger than 16% are not reported.



(b) Box-and-whisker plots of computing times.

Fig. 4. Box-and-whisker plots for the MIPLIB2010-NOS data set (unknown optimum).

gahertz Intel Core 2 Duo E6600 processor and 4.0 gigabytes RAM, the version of CPLEX used was 10.1, and their algorithm was implemented in C++ and compiled within Microsoft Visual Studio 2005. For the MIPLIB2010-OS and MIPLIB2010-NOS instances, we report the results obtained by running the VNDS algorithm provided by the authors on our hardware and using CPLEX 12.6.0. For a better assessment of the different AKS variants, we report the results obtained by VNDS with a time limit of 18,000 seconds, denoted by VNDS(5h), and with a limit of 3600 seconds, denoted by VNDS(1h). Parameter values were set to those suggested in Lazić et al. (2010). Finally, since the VNDS is devised for solving 0–1 MIPs (Lazić et al. 2010), we did not test the VNDS on instances with integer variables – in the computational results tables this is indicated with *N/A*.

5.5. Computational results: the Lazić instances

In this section, we assess the performance of AKS against the results of the VNDS as reported in Lazić et al. (2010), that is the state of the art heuristic on the Lazić data set. The results are given in Table 2. We observed that some of these instances are no longer challenging for the current version of CPLEX. Indeed, CPLEX solved 14 out of 29 instances in less than 10 minutes, and 7 of them in less than 2 minutes. Nevertheless, to provide a complete comparison, we report the results for all the instances.

The results in Table 2 show that any AKS variant performs significantly better than the VNDS of Lazić et al. (2010) in terms of solution quality. In fact, the average value of z^{UB} *Gap %* for the method of Lazić et al. (2010) is 6.89%, while the value is -1.35% for AKS(5h), 2.14% for AKS-TL(5h) and 2.64% for AKS-TBL. The negative value is due to the large improvement achieved on instance MARKSHARE1, one of the most peculiar instances of the data set, while the positive values are mostly due to the large deterioration on instance MARKSHARE2, another peculiar instance of the data set. Note that the large gaps for these two instances are caused by the small value of the solutions found by CPLEX(5h) (equal to 2 and 4, respectively). The average computing time of AKS(5h) is 10% higher than that of the VNDS but is reduced by approximately 60% with AKS-TL(5h) and 80% with AKS-TBL. On average AKS-TBL outperforms CPLEX(1h) both in terms of solution quality and computing time. When solving instances of this set, no AKS variant had to invoke GetFeasible.

Finally, we want to highlight that we solved these instances also with a straightforward extension of the KS described in Section 3 to general MIPs and a time limit of 18,000 seconds, obtaining an average value of z^{UB} Gap % of 7.04% with an average computing time of 7513 seconds. Moreover, on two instances no feasible solution was found. These results indicate that the new features of AKS have a crucial impact on its performance.

The box-and-whisker plots shown in Fig. 2 summarize the computational results for this data set. The left and right edges of each box represent the first (Q_1) and third (Q_3) quartiles of the solutions, while the line inside the box represents the second quartile (i.e., the median). Hence, by construction the middle half of the solutions falls within the box. The lines extending horizontally from each box (whiskers) indicate the variability outside the first and third quartiles. The front whisker goes from Q_1 to the smallest non-outlier in the solutions, while the back whisker goes from Q_3 to the largest non-outlier. As conventional, we consider as outliers (represented by crosses) all solutions whose value is either $1.5 \times (Q_3 - Q_1)$ or more above the third quartile, or $1.5 \times (Q_3 - Q_1)$ or more below the first quartile. Fig. 2(a) summarizes the values of z^{UB} Gap %. The box-and-whisker plots show that each AKS variant outperforms VNDS, and that the deterioration tends to increase moving from AKS(5h) to AKS-TL(5h), and, in turn, to AKS-TBL. Note that the box-and-whisker plot for CPLEX(1h) confirms that many of these instances are no longer challenging for the current version of the solver. Fig. 2(b) summarizes the computing times. No results are shown for VNDS because they were obtained on different hardware and a different programming language was used for its implementation. The box-and-whisker plots highlight the reduction in computing times that can be obtained by using AKS-TL(5h) and, even more clearly, by using AKS-TBL.

5.6. Computational results: the MIPLIB2010 instances

In this section, we assess the performance of AKS on the MIPLIB2010 data set. We first discuss the results obtained on the MIPLIB2010-OS group, and then on the MIPLIB2010-NOS. We recall that the value of an optimal solution is known only on the MIPLIB2010-OS instances, while for the latter data set no information is available.

Table 3 shows the results for the MIPLIB2010-OS data set. Looking at z^* Gap % for CPLEX(5h) it is evident that for several of these instances the solver found an optimal solution, although it did not prove its optimality within the time allocated. CPLEX(5h) found an optimal solution for 23 out of the 53 instances composing this data set. Note also that for the majority of the instances CPLEX(5h) found a near-optimal solution. Statistic z^* Gap % is smaller than 1% on 40 instances for CPLEX(5h) and on 36 instances for CPLEX(1h). The average deterioration of AKS(5h) with respect to CPLEX(5h) is 0.21% with a slightly smaller average computing time (approximately 16,000 seconds instead of 18,000). Interestingly, even though CPLEX(5h) found a near-optimal solution for many of the instances in this data set, AKS(5h) improved upon CPLEX(5h) 9 times, achieving a maximum improvement of 8.16% on instance NS1696083. AKS(5h) found the same solution as CPLEX(5h), or a better one, 27 times out of 53. The maximum deterioration of AKS(5h) is 6.00%. The increase of the average deterioration of AKS-TL(5h) with respect to AKS(5h) is rather small: from 0.21% to 0.58%. On the other hand, the reduction in the average computing time is quite remarkable, since AKS-TL(5h) spent on average slightly more than $\frac{1}{3}$ of the computing time required by AKS(5h). AKS-TL(5h) found the same or a better solution than CPLEX(5h) 24 times out of 53, and reported the same maximum improvement and maximum deterioration as AKS(5h). AKS-TBL achieved an average value of z^{UB} Gap % of 2% with an additional reduction in computing time, approximately half of the time of AKS-TL(5h). Also on this set of instances, AKS-TBL outperformed, on average, CPLEX(1h) both in terms of solution quality and computing time. It is worth observing that only for instance NS1696083 all AKS variants deployed GetFeasible. The first feasible solution was found at the 5th iteration. Note that this instance is also the one where each variant achieved the maximum improvement upon CPLEX(5h). The above observations do not change significantly if the analysis is restricted to the instances without integer variables. For these instances, VNDS(5h) achieves an average value of z^{UB} Gap % of 10.62%, which is worse than the other methods, and did not find a feasible solution for instance IN within the time limit. Whenever VNDS is unable to find a feasible solution, we report *Fails* in column z^{UB} Gap % in the following tables. VNDS(5h) found the same solution as CPLEX(5h), or a better one, 8 times out of 43, achieving a maximum improvement of 4.08% on instance NS1696083, with a maximum deterioration of 72.73%. VNDS(1h) achieves an average value of z^{UB} Gap % of 12.88%, which is worse than both CPLEX(1h) and AKS-TBL, and it does not find a feasible solution for 2 of the instances within the time limit.

We now comment on the results obtained for the MIPLIB2010-NOS data set, which are reported in Table 4. Although the optimum value is not known for these instances, LB Gap % gives an indication of how far the value of the best solution found can be from the optimum value. The gap is smaller than 1% for 11 out of 55 instances, indicating that for these instances CPLEX(5h) found a solution which is very close to the optimum. For the remaining instances, the gap varies considerably. Solving this set of instances, AKS(5h) achieved, on average, a noticeable improvement over CPLEX(5h), i.e., of −1.48%, in, on average, less computing time. Looking in more details at the results, the results shown in Table 4 indicate that AKS(5h) achieved, in some cases, very large improvements, e.g., −62.60% for instance rocII-9-11, but also had some large deteriorations, e.g., 61.94% for instance ns1456591. Overall, AKS(5h) obtained the same or a better solution than CPLEX(5h) 31 times out of 55, being able to improve upon the solution provided by CPLEX(5h) for 20 instances (the related figures are highlighted in bold in Table 4). Using AKS-TL(5h) instead of AKS(5h), the average value of z^{UB} Gap % becomes 1.69% with computing time that is, on average, more than halved. Note that the average deterioration of the solution quality is due, primarily, to 5 instances (momentum3, ns1631475, rocII-7-11, rocII-9-11, and shipsched). On the other hand, AKS-TL(5h) improved upon the solution found by CPLEX(5h) 15 times out of 55, and found a solution that is equal or better for 25 instances. Average computing times can be reduced further using AKS-TBL, at the expense of a further deterioration of the solution quality. Indeed, AKS-TBL spent, on average, half of the computing time required by AKS-TL(5h), with an increase in the average deterioration from 1.69% to 4.20%. When comparing the performance of AKS-TBL with CPLEX(1h), the average deterioration decreases whereas the computing times become slightly higher. Note that both AKS-TBL and CPLEX(1h) did not find any feasible solution for instance circ10-3 within the time limit. All AKS variants deployed GetFeasible on instances circ10-3 and lectsched-1-ob. AKS(5h) and AKS-TL(5h) found the first feasible solution at the 3rd iteration for both instances, while AKS-TBL found the first feasible solution at the 3rd iteration for instance lectsched-1-ob and ran out of computing time in GetFeasible for instance circ10-3. When considering the performance of VNDS(5h), this method achieved an average value of z^{UB} Gap % of 9.42%, and did not find any feasible solution within the time limit 4 times out of 41. VNDS(5h) found the same or a better solution than CPLEX(5h) 6 times, achieving a maximum improvement of 13.09%, and a maximum deterioration of 90.01%. VNDS(1h) achieved an average value of z^{UB} Gap % of 16.67%, and did not find any feasible solution for 5 instances within the time limit.

Figs. 3 and 4 summarize the results for the MIPLIB2010-OS and the MIPLIB2010-NOS data sets, respectively. Note that in both Figs. 3(a) and 4(a), portions of boxes and whiskers as well as outliers to the left of 0% correspond to solutions that improve upon the solution found by CPLEX(5h). These box-and-whisker plots demonstrate, again, that the AKS variants can significantly outperform the VNDS. The pictures also illustrate the impact of the different choices in the three AKS variants regarding the trade-off between solution quality and computational effort. The box-and-whisker plot for CPLEX(1h) in Fig. 3(a) reveals that after 3600 seconds the solver has found an optimal or near optimal solution for many instances in the MIPLIB2010-OS data set. We cannot draw the same conclusion for the MIPLIB2010-NOS instances, as we do not know the value of optimal solutions. Nevertheless, examining Fig. 4(a), it is clear that for many instances the solution found by CPLEX after 1 hour is only slightly improved by CPLEX in the following 4 hours. Indeed, for the middle half of the solutions, CPLEX(1h) has a z^{UB} Gap % ranging from 0% to roughly 2%, with a median (which is hardly visible in Fig. 4(a)) very close to 0%. This suggests that either CPLEX(1h) found an optimal and the solver spent all additional time proving optimality of that solution, or that CPLEX got itself trapped in a region of the search space where no improving solutions can be found (e.g., for instances ns1853823 and ns1854840 – see Table 4).

6. Concluding remarks

We have introduced and computationally tested a heuristic framework for solving general MIPs, called Adaptive Kernel Search. An extensive computational study has demonstrated the benefits of AKS: it significantly outperforms other heuristics for solving 0–1 MIPs and often outperforms CPLEX when CPLEX is run with a time limit. An important feature of AKS is that it allows for easy exploration of the trade-off between solution quality and computing time. As such, AKS is a valuable addition to the toolbox of researchers and practitioners needing to obtain high-quality solutions to MIPs in a short amount of time.

Acknowledgments

We would like to thank Jasmina Lazić for providing us with the source code of the VNDS. We would also like to express our appreciation to the associate editor and three anonymous reviewers for their constructive comments and suggestions which allowed us to improve a former version of the paper.

References

Achterberg, T., & Berthold, T. (2007). Improving the feasibility pump. *Discrete Optimization, 4*(1), 77–86.
Angelelli, E., Mansini, R., & Speranza, M. G. (2010). Kernel search: A general heuristic for the multi-dimensional knapsack problem. *Computers & Operations Research, 37*(11), 2017–2026.
Angelelli, E., Mansini, R., & Speranza, M. G. (2012). Kernel search: A new heuristic framework for portfolio selection. *Computational Optimization and Applications, 51*(1), 345–361.
Baena, D., & Castro, J. (2011). Using the analytic center in the feasibility pump. *Operations Research Letters, 39*(5), 310–317.
Balas, E., Ceria, S., Dawande, M., Margot, F., & Pataki, G. (2001). OCTANE: A new heuristic for pure 0–1 programs. *Operations Research, 49*(2), 207–225.
Balas, E., Schmieta, S., & Wallace, C. (2004). Pivot and shift – A mixed integer programming heuristic. *Discrete Optimization, 1*(1), 3–12.
Bertacco, L., Fischetti, M., & Lodi, A. (2007). A feasibility pump heuristic for general mixed-integer problems. *Discrete Optimization, 4*(1), 63–76.
Boland, N. L., Eberhard, A. C., Engineer, F. G., Fischetti, M., Savelsbergh, M., & Tsoukalas, A. (2014). Boosting the feasibility pump. *Mathematical Programming Computation, 6*(3), 255–279.
Danna, E., Rothberg, E., & Le Pape, C. (2005). Exploring relaxation induced neighborhoods to improve MIP solutions. *Mathematical Programming, Series A, 102*(1), 71–90.
De Santis, M., Lucidi, S., & Rinaldi, F. (2013). A new class of functions for measuring solution integrality in the feasibility pump approach. *SIAM Journal on Optimization, 23*(3), 1575–1606.
De Santis, M., Lucidi, S., & Rinaldi, F. (2014). Feasibility pump-like heuristics for mixed integer problems. *Discrete Applied Mathematics, 165*, 152–167.
Eckstein, J., & Nediak, M. (2007). Pivot, cut, and dive: A heuristic for 0–1 mixed integer programming. *Journal of Heuristics, 13*(5), 471–503.
Filippi, C., Guastaroba, G., & Speranza, M. G. (2016). A heuristic framework for the bi-objective enhanced index tracking problem. *Omega, 65*, 122–137.
Fischetti, M., Glover, F., & Lodi, A. (2005). The feasibility pump. *Mathematical Programming, Series A, 104*(1), 91–104.
Fischetti, M., & Lodi, A. (2003). Local branching. *Mathematical Programming, Series B, 98*(1), 23–47.
Fischetti, M., & Lodi, A. (2008). Repairing MIP infeasibility through local branching. *Computers & Operations Research, 35*(5), 1436–1445.
Fischetti, M., & Lodi, A. (2011). Heuristics in mixed integer programming. In J. Cochran (Ed.), *Wiley encyclopedia of operations research and management science: 3* (pp. 2199–2204). John Wiley & Sons.
Fischetti, M., & Monaci, M. (2014). Proximity search for 0–1 mixed-integer convex programming. *Journal of Heuristics, 20*(6), 709–731.
Fischetti, M., & Salvagnin, D. (2009). Feasibility pump 2.0. *Mathematical Programming Computation, 1*(2), 201–222.
Glover, F. (2006). Parametric tabu-search for mixed integer programs. *Computers & Operations Research, 33*(9), 2449–2494.
Glover, F., & Laguna, M. (1997a). General purpose heuristics for integer programming – Part I. *Journal of Heuristics, 2*(4), 343–358.
Glover, F., & Laguna, M. (1997b). General purpose heuristics for integer programming – Part II. *Journal of Heuristics, 3*(2), 161–179.
Guastaroba, G., & Speranza, M. G. (2012a). Kernel search: An application to the index tracking problem. *European Journal of Operational Research, 217*(1), 54–68.
Guastaroba, G., & Speranza, M. G. (2012b). Kernel search for the capacitated facility location problem. *Journal of Heuristics, 18*(6), 877–917.
Guastaroba, G., & Speranza, M. G. (2014). A heuristic for BILP problems: The single source capacitated facility location problem. *European Journal of Operational Research, 238*(2), 438–450.

Guzelsoy, M., Nemhauser, G., & Savelsbergh, M. (2013). Restrict-and-relax search for 0–1 mixed-integer programs. *EURO Journal on Computational Optimization, 1*(1), 201–218.

Hanafi, S., Lazić, J., Mladenović, N., Wilbaut, C., & Crévits, I. (2015). New variable neighbourhood search based 0–1 MIP heuristics. *Yugoslav Journal of Operations Research, 25*(3), 343–360.

Hansen, P., Mladenović, N., & Urošević, D. (2006). Variable neighborhood search and local branching. *Computers & Operations Research, 33*(10), 3034–3045.

Koch, T., Achterberg, T., Andersen, E., Bastert, O., Berthold, T., Bixby, R. E., Danna, E., Gamrath, G., Gleixner, A. M., Heinz, S., Lodi, A., Mittelmann, H., Ralphs, T., Salvagnin, D., Stey, D. E., & Wolter, K. (2011). MIPLIB 2010. *Mathematical Programming Computation, 3*(2), 103–163.

Lazić, J., Hanafi, S., Mladenović, N., & Urošević, D. (2010). Variable neighbourhood decomposition search for 0–1 mixed integer programs. *Computers & Operations Research, 37*(6), 1055–1067.

Løkketangen, A., & Glover, F. (1998). Solving zero-one mixed integer programming problems using tabu search. *European Journal of Operational Research, 106*(2), 624–658.

Rothberg, E. (2007). An evolutionary algorithm for polishing mixed integer programming solutions. *INFORMS Journal on Computing, 19*(4), 534–541.

Sacchi, L. H., & Armentano, V. A. (2011). A computational study of parametric tabu search for 0–1 mixed integer programs. *Computers & Operations Research, 38*(2), 464–473.