Comput Optim Appl (2018) 69:1–24
https://doi.org/10.1007/s10589-017-9934-5

Alternating criteria search: a parallel large neighborhood search algorithm for mixed integer programs

Lluís-Miquel Munguía[1] · Shabbir Ahmed[2] · David A. Bader[1] · George L. Nemhauser[2] · Yufen Shao[3]

Received: 6 June 2016 / Published online: 8 August 2017

Abstract We present a parallel large neighborhood search framework for finding high quality primal solutions for general mixed-integer programs (MIPs). The approach simultaneously solves a large number of sub-MIPs with the dual objective of reducing infeasibility and optimizing with respect to the original objective. Both goals are achieved by solving restricted versions of two auxiliary MIPs, where subsets of the variables are fixed. In contrast to prior approaches, ours does not require a feasible starting solution. We leverage parallelism to perform multiple searches simultaneously, with the objective of increasing the effectiveness of our heuristic. We computationally compare the proposed framework with a state-of-the-art MIP solver in terms of solution quality, scalability, reproducibility, and parallel efficiency. Results show the efficacy of our approach in finding high quality solutions quickly both as a standalone primal heuristic and when used in conjunction with an exact algorithm.

Keywords MIPs · Parallel algorithms · Primal heuristics · LNS

This research has been supported in part by ExxonMobil Upstream Research Company, the National Science Foundation, the Office of Naval Research and the Air Force Office of Scientific Research.

✉ Lluís-Miquel Munguía
lluis.munguia@gatech.edu

[1] College of Computing, Georgia Institute of Technology, Atlanta, GA 30332, USA

[2] School of Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta, GA 30332, USA

[3] ExxonMobil Upstream Research Company, Houston, TX 77098, USA

1 Introduction

We present *Parallel Alternating Criteria Search*, a parallel large neighborhood search (LNS) heuristic designed for finding high quality feasible solutions to general MIPs. In discrete optimization, high quality feasible solutions are valuable assets in the optimization process, and constructing them has been one of the main focuses of research for the past two decades. Berthold [9] and Fischetti et al. [22] present comprehensive literature reviews on primal heuristics and their applications to MIPs.

Starting heuristics and improvement heuristics are two classes of primal heuristics that differ in whether they require a starting feasible solution or not. The feasibility pump [8,19] is a widely used starting heuristic for finding feasible solutions to MIP instances quickly. It consists of an iterative algorithm that combines linear program (LP) relaxations and roundings to enforce LP and integer feasibility until a feasible solution is found. Successive works built on the original feasibility pump to improve the solution quality [2] and its overall success rate [5,14,25,41]. Structure-based heuristics [26] and RENS [11] (relaxation enforced neighborhood search) are other compelling starting heuristics for finding feasible solutions. RENS belongs to the class of LNS heuristics, which entail solving carefully restricted sub-MIPs derived from the original problem. Their effectiveness relies on the ability to use the full power of a MIP solver to optimize the subproblem. LNS approaches differ in how the search neighborhood is defined. RENS attempts to find the best possible rounding when a fractional solution is given. Based on domain propagation and rounding, shift-and-propagate [12] and ZI rounding [46] are additional successful starting heuristics which are computationally tested in [3]. More algorithms for finding solutions to MIP instances can be found in the literature [6,7,28–30].

Improvement heuristics, on the other hand, require a feasible starting solution and their focus is to improve its quality with respect to the objective. Simple improvement algorithms include the 1-opt [1] and 2-opt [34] heuristics. A large number of improvement heuristics found in the literature rely on LNS ideas. Successful LNS heuristics include local branching [20] and subsequent LNS heuristics that use local branching [33], RINS [18] (relaxation induced neighborhood search) , DINS [27] (distance induced neighborhood search), proximity search [24] and evolutionary algorithms [42]. The neighborhoods within these heuristics are usually defined using branch-and-bound information, such as the best available solutions or the LP relaxation at the current tree node. Because of this requirement, they must be executed as part of the node processing routine during the branch-and-bound process. Due to this input dependence, many nodes must be explored before these heuristics become effective at exploring diversified neighborhoods. Thus, high quality upper bound improvements are rarely found early in the search. In order to address this issue, the search neighborhoods used in our heuristic are defined using randomization instead of branch-and-bound dependent information. This allows us to obtain a wide range of diverse search neighborhoods from the beginning of the search, thus increasing the heuristic's effectiveness.

1.1 Parallel computing applied to integer optimization

Due to recent trends in processor design, parallelism has become ubiquitous in today's computers. With the advent of multi-core CPUs, it has become necessary to rethink most conventional algorithms in order to leverage the benefits of parallel computing. In the field of discrete optimization, Bader et al. [4] and Koch et al. [38] discuss potential applications of parallelism. The most widely used strategy entails exploring the branch-and-bound tree in parallel by solving multiple subproblems simultaneously. Due to its simplicity, most state-of-the-art MIP solvers incorporate this technique, such as CPLEX [16], GUROBI [13], and ParaSCIP [44]. However, studies have suggested that parallelizing the branch-and-bound search may not scale well to a large number of cores [38]. Alternative parallelizations have been proposed in [23] and [15], where parallelism is used to explore different perturbations of the same problem. Other options include the application of parallelism to cut generation, primal heuristics, preprocessing and branching. To our knowledge, the work of Koc et al. [36] are the only effort to parallelize primal heuristics for general MIPs. In this work, the authors present a parallelization of the Feasibility Pump [19]. Therefore, Parallel Alternating Criteria Search is the first parallel algorithm to combine features from starting and improvement heuristics, offering the possibility of generating starting solutions and improving them with respect to the original objective.

LNSs are some of the most computationally expensive heuristics, since they are based on the optimization of sub-MIPs. A strategy to leverage parallelism is to perform a large number of LNS simultaneously over a diversified set of neighborhoods with the objective of increasing the chances of finding better solutions. To some degree, the parallelization of the branch-and-bound tree already provides this diversification and improvement in performance, since the exploration of multiple nodes in parallel includes the simultaneous execution of multiple heuristics with a diverse set of inputs. Our heuristic builds upon a similar parallelization strategy, and expands it by adding an additional algorithmic step that combines and consolidates the improvements found in parallel.

Parallel Alternating Criteria Search combines parallelism and diversified large neighborhood searches in order to deal with large instances. Our approach is suitable for MIPs belonging to all kinds of applications, since it does not require knowledge or assumptions regarding the underlying structure of the problem. Although most of the algorithmic components present in Parallel Alternating Criteria Search have been introduced in the literature before, we demonstrate their great effectiveness when put together in a parallel algorithm. We find our approach to be competitive or better than CPLEX at finding solutions for more than 90% of the instances in the MIPLIB2010 library [37]. The improvement of the proposed method becomes more pronounced on harder instances. Additionally, we present a parallel scheme that combines the use of Alternating Criteria Search in conjunction with an exact algorithm. Results show that alternative parallelizations of the branch-and-bound process can be more efficient than traditional methods, especially when large-scale instances are considered.

We introduce our primal heuristic in Sect. 2, where we present components that define it and give further details on the parallel implementation. Section 3 presents computational experiments and results on standard instances from the literature. Section 4 provides some concluding remarks.

2 Parallel alternating criteria search

We define a mixed-integer program (MIP) as:

$$\min \left\{ c^t x | Ax = b, l \leq x \leq u, x_i \in \mathbb{Z}, \forall i \in \mathcal{I} \right\} \tag{MIP}$$

where $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, and $\mathcal{I} \subseteq \{1, \ldots, n\}$ is the subset of integer variable indices. The decision vector x is bounded by $l \in \overline{\mathbb{R}}^n$ and $u \in \overline{\mathbb{R}}^n$, where $\overline{\mathbb{R}}$ is the extended set of real numbers $\mathbb{R} \cup \{-\infty, \infty\}$.

Our intent is to satisfy a twofold objective: to find a feasible starting solution and to improve it with respect to the original objective. We introduce an LNS heuristic, in which two auxiliary MIP subproblems are iteratively solved to attain both goals. The process requires an initial vector, which is not required to be a feasible solution. As seen in Fig. 1, this vector is improved by solving sub-MIPs, in which a subset of the variables are fixed to its input values.

For linear programs, the feasibility problem is solved via the two-phase Simplex method, in which an auxiliary optimization problem is developed in order to find a feasible starting basis. This approach was introduced for the case of 0–1 MIPs in [21]. In a similar fashion, the following auxiliary MIP, denoted as FMIP, poses the problem of finding a feasible starting solution as an optimization problem:

$$\begin{aligned} &\min \sum_{i=0}^{m} \Delta_i^+ + \Delta_i^- \\ &\text{s.t.} \\ &\quad Ax + I_m \Delta^+ - I_m \Delta^- = b \\ &\quad x_i = \hat{x}_i, \forall i \in \mathcal{F} \\ &\quad l \leq x \leq u \\ &\quad x_i \in \mathbb{Z}, \forall i \in \mathcal{I} \\ &\quad \Delta^+ \geq 0, \Delta^- \geq 0 \end{aligned} \tag{FMIP}$$

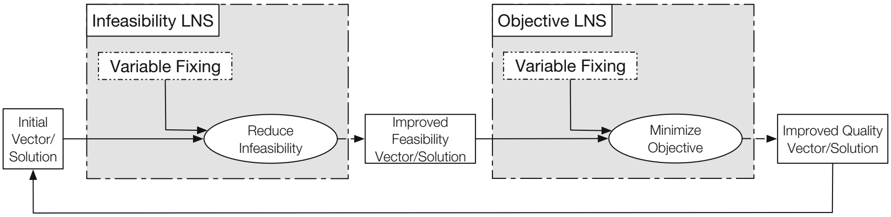


Fig. 1 High level depiction of the sequential heuristic

where I_m is an $m \times m$ identity matrix and Δ^+, Δ^- are two vectors of continuous variables of size m corresponding to the m constraints. procedure, auxiliary variables are introduced as slack for each constraint and the sum of their value is minimized. A decision vector is feasible to a MIP if and only if it can be extended to a solution of value 0 to the associated FMIP. Instead of directly solving FMIP, neighborhoods are restricted by fixing a given subset $\mathcal{F}$ of the integer variables to the values of an input vector $[\hat{x}, \hat{\Delta}^+, \hat{\Delta}^-]$. Due to the addition of slack variables, $\hat{x}$ is not required to be a feasible solution vector. However, it must be integer and within the variable bounds in order to preserve the feasibility of the model: $l \leq \hat{x} \leq u$ and $\hat{x}_i \in \mathbb{Z}, \forall i \in \mathcal{I}$.

FMIP ensures that feasibility is preserved under any arbitrary variable fixing scheme. This represents a departure from most LNS heuristic improvement approaches such as RINS, DINS, local branching, and proximity search, where the choice of variable fixings is tied to the availability of a feasible solution. In the context of our heuristic, variable fixings become a viable tool for reducing the complexity of the problem.

Using a similar approach, we introduce a second auxiliary problem aimed at improving a partially feasible vector $[\hat{x}, \hat{\Delta}^+, \hat{\Delta}^-]$ with respect to the original objective:

$$\begin{aligned}
\min \quad & c^t x \\
\text{s.t.} \quad & \\
& Ax + I_m\Delta^+ - I_m\Delta^- = b \\
& \sum_{i=0}^{m} \Delta_i^+ + \Delta_i^- \leq \sum_i \hat{\Delta}_i^+ + \hat{\Delta}_i^- \\
& x_i = \hat{x}_i, \forall i \in \mathcal{F} \\
& l \leq x \leq u \\
& x_i \in \mathbb{Z}, \forall i \in \mathcal{I} \\
& \Delta^+ \geq 0, \Delta^- \geq 0
\end{aligned} \tag{OMIP}$$

OMIP is a transformation of the original MIP model, in which auxiliary slack variables are introduced in each constraint. Achieving and preserving the feasibility of the incumbent is our primary concern. In order to ensure that the optimal solution to OMIP remains at most as infeasible as the input solution $\hat{x}$, an additional constraint that limits the amount of slack is added, where the degree of infeasibility is bounded by $\sum_i \hat{\Delta}_i^+ + \hat{\Delta}_i^-$.

By iteratively solving subproblems of both auxiliary MIPs, the heuristic will hopefully converge (although its convergence is not guaranteed) to a high quality feasible solution. By construction, infeasibility decreases monotonically after each iteration. On the other hand, the solution quality may fluctuate with respect to the original objective. Figure 2 depicts the expected behavior of the algorithm. A similar approach for achieving feasibility via the FMIP model is presented in [21], although our work differs in several key aspects. The authors of the aforementioned work limit their scope to 0-1 problems and use local branching to explore FMIP. Our approach accepts general-integer variables and uses variable fixings as a means of exploiting parallelism and to reduce infeasibility. Another differentiating factor of our approach is the transition

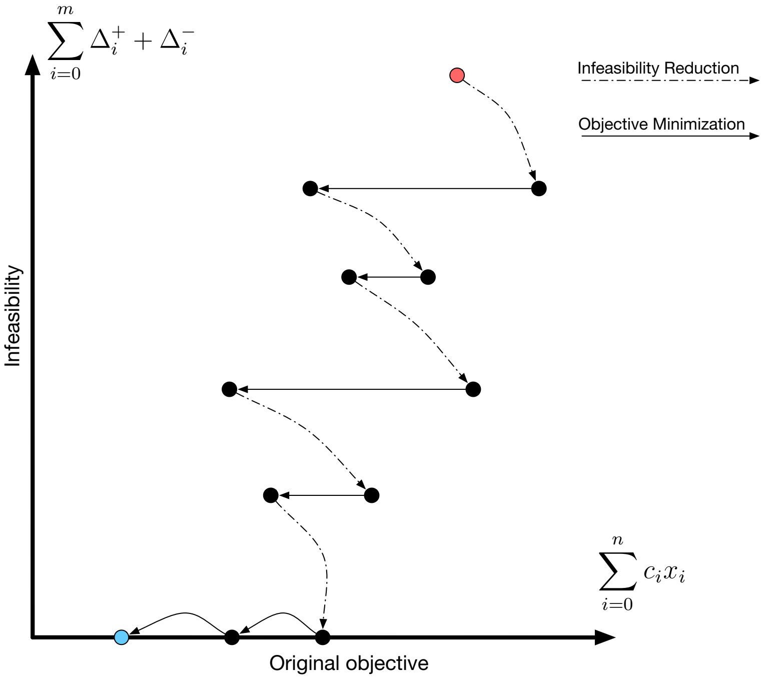


Fig. 2 Transition to a high quality feasible solution

and use of an auxiliary MIP model, OMIP, to provide improvements with respect to the original objective.

2.1 Parallelization of alternating criteria search

We leverage parallelism by generating a diversified set of large neighborhood searches, which are solved simultaneously. By exploring a large number of different search neighborhoods in parallel, we hope to increase the chances of finding solution improvements, hence speeding up the overall process. After this exploration phase, improvements found in parallel are combined efficiently. For this purpose, an additional search subproblem is generated, in which the variables that have the same value across the different solutions are fixed. A similar approach has been previously described in the literature [9,42]. In this context, the solution recombination provides the ability to merge the improvements found in parallel, providing a speedup in the process. A pseudocode version is given in Algorithm 1. Each parallel processor iteratively generates a set of randomized variable fixings and solves the associated sub-MIP of either FMIP or OMIP until the allowed time limit. Upon termination, all solutions are exchanged and the set $\mathcal{U}$ containing the indices of the variables with identical values across solutions is determined. The solution recombination MIP consists of a subproblem, in which the variables present in $\mathcal{U}$ are fixed. The size of $\mathcal{U}$ is dependent on the similarities between the solutions to be merged. If the set is too large and no

Algorithm 1 Parallel Feasibility Heuristic

```
Ensure: Feasible solution $\hat{x}$ if found
  $initialize\ [\hat{x}, \Delta^{+}, \Delta^{-}]$ as an integer solution
  $T := numThreads()$
  while time limit not reached do
    if $\sum_i \Delta_i^{+} + \Delta_i^{-} > 0$ then
      for all threads $t_i \in \{0, T-1\}$ in parallel do
        $\mathcal{F}_{t_i}$ := randomized variable index subset, $\mathcal{F}_{t_i} \subseteq \mathcal{I}$     ▷ Variable Fixings are diversified
        $[x^{t_i}, \Delta^{+t_i}, \Delta^{-t_i}]$ :=FMIP_LNS($\mathcal{F}_{t_i}, \hat{x}$)     ▷ $FMIP$ LNS are solved concurrently
      end for
      $\mathcal{U} := \{j \in \mathcal{I} | x_j^{t_i} = x_j^{t_k}, 0 \leq i < k < T\}$
      $[\hat{x}, \Delta^{+}, \Delta^{-}]$ :=FMIP_LNS($\mathcal{U}, x^{t_0}$)     ▷ The recombination step differs in variable fixings
    end if
    $\Delta^{UB} := \sum_i \Delta_i^{+} + \Delta_i^{-}$
    for all threads $t_i \in \{0, T-1\}$ in parallel do
      $\mathcal{F}_{t_i}$ := randomized variable index subset, $\mathcal{F}_{t_i} \subseteq \mathcal{I}$
      $[x^{t_i}, \Delta^{+t_i}, \Delta^{-t_i}]$ :=OMIP_LNS($\mathcal{F}_{t_i}, \hat{x}, \Delta^{UB}$)
    end for
    $\mathcal{U} := \{j \in \mathcal{I} | x_j^{t_i} = x_j^{t_k}, 0 \leq i < k < T\}$
    $[\hat{x}, \Delta^{+}, \Delta^{-}]$ :=OMIP_LNS($\mathcal{U}, x^{t_0}, \Delta^{UB}$)
  end while
  $return\ [\hat{x}, \Delta^{+}, \Delta^{-}]$

  function FMIP_LNS($\mathcal{F}, \hat{x}$)
    return $min\{\sum_i \Delta_i^{+} + \Delta_i^{-} | Ax + I_m\Delta^{+} - I_m\Delta^{-} = b, x_j = \hat{x}_j\ \forall j \in \mathcal{F}, x_j \in \mathbb{Z}\ \forall j \in \mathcal{I}\}$
  end function

  function OMIP_LNS($\mathcal{F}, \hat{x}, \hat{\Delta}$)
    return $min\{c^t x | Ax + I_m\Delta^{+} - I_m\Delta^{-} = b, \sum_i \Delta_i^{+} + \Delta_i^{-} \leq \hat{\Delta}, x_j = \hat{x}_j\ \forall j \in \mathcal{F}, x_j \in \mathbb{Z}\ \forall j \in \mathcal{I}\}$
  end function
```

improvements can be found, the best solution used in the recombination is returned. The best solution will be the most feasible or the most optimal, depending on whether a recombination FMIP or OMIP is being optimized. Every solution used as input can be also added as a MIP start, since they remain feasible under the set of variable fixings. Figure 3 depicts an example for a simple 0–1 knapsack instance. Firstly, the Feasibility MIP is derived from the original problem instance. Next, two subproblems characterized by different fixings are solved in parallel. In a final step, the variables with coinciding values are fixed and a feasible solution is found.

Distributed-memory parallelism is the main paradigm for large-scale parallel computer architectures. One of the defining characteristics is the fact that memory is partitioned among parallel processors. As a result, processor syncronization and memory communication must be done via a message passing interface, such as MPI [32]. Processor coordination and communication becomes a problematic element in the algorithm design, inducing occasional overheads. Our approach requires the exchange of solutions between processors before and after each solution recombination. The arrangement is such that every processor communicates its best solution to every other processor during an all-to-all synchronous exchange. Unlike point-to-point communications, all-to-all collective communication primitives take advantage of the

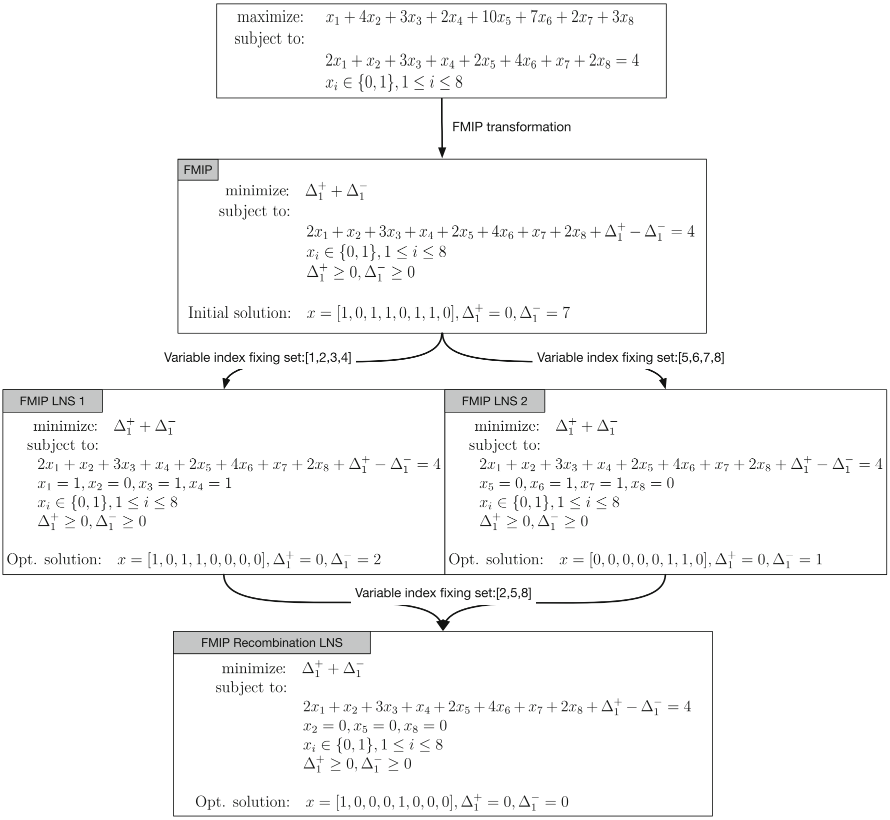

maximize: $x_1 + 4x_2 + 3x_3 + 2x_4 + 10x_5 + 7x_6 + 2x_7 + 3x_8$
subject to:
$2x_1 + x_2 + 3x_3 + x_4 + 2x_5 + 4x_6 + x_7 + 2x_8 = 4$
$x_i \in \{0,1\}, 1 \le i \le 8$
FMIP transformation
FMIP
minimize: $\Delta_1^+ + \Delta_1^-$
subject to:
$2x_1 + x_2 + 3x_3 + x_4 + 2x_5 + 4x_6 + x_7 + 2x_8 + \Delta_1^+ - \Delta_1^- = 4$
$x_i \in \{0,1\}, 1 \le i \le 8$
$\Delta_1^+ \ge 0, \Delta_1^- \ge 0$
Initial solution: $x = [1,0,1,1,0,1,1,0], \Delta_1^+ = 0, \Delta_1^- = 7$
Variable index fixing set:[1,2,3,4]
Variable index fixing set:[5,6,7,8]
FMIP LNS 1
minimize: $\Delta_1^+ + \Delta_1^-$
subject to:
$2x_1 + x_2 + 3x_3 + x_4 + 2x_5 + 4x_6 + x_7 + 2x_8 + \Delta_1^+ - \Delta_1^- = 4$
$x_1 = 1, x_2 = 0, x_3 = 1, x_4 = 1$
$x_i \in \{0,1\}, 1 \le i \le 8$
$\Delta_1^+ \ge 0, \Delta_1^- \ge 0$
Opt. solution: $x = [1,0,1,1,0,0,0,0], \Delta_1^+ = 0, \Delta_1^- = 2$
FMIP LNS 2
minimize: $\Delta_1^+ + \Delta_1^-$
subject to:
$2x_1 + x_2 + 3x_3 + x_4 + 2x_5 + 4x_6 + x_7 + 2x_8 + \Delta_1^+ - \Delta_1^- = 4$
$x_5 = 0, x_6 = 1, x_7 = 1, x_8 = 0$
$x_i \in \{0,1\}, 1 \le i \le 8$
$\Delta_1^+ \ge 0, \Delta_1^- \ge 0$
Opt. solution: $x = [0,0,0,0,0,1,1,0], \Delta_1^+ = 0, \Delta_1^- = 1$
Variable index fixing set:[2,5,8]
FMIP Recombination LNS
minimize: $\Delta_1^+ + \Delta_1^-$
subject to:
$2x_1 + x_2 + 3x_3 + x_4 + 2x_5 + 4x_6 + x_7 + 2x_8 + \Delta_1^+ - \Delta_1^- = 4$
$x_2 = 0, x_5 = 0, x_8 = 0$
$x_i \in \{0,1\}, 1 \le i \le 8$
$\Delta_1^+ \ge 0, \Delta_1^- \ge 0$
Opt. solution: $x = [1,0,0,0,1,0,0,0], \Delta_1^+ = 0, \Delta_1^- = 0$

Fig. 3 Example depicting a feasibility improvement iteration for a 0–1 knapsack sample instance

underlying network structure to communicate synchronously among a large number of processors in a more efficient manner [45].

2.2 Finding a starting point

Alternating Criteria Search only requires a starting vector that is integer feasible and within variable bounds. However, given that feasibility is one of the primary objectives of the heuristic, it is proposed to choose a starting point that is as feasible as possible with respect to the objective function of FMIP. Many strategies can provide a start for the algorithm. A common strategy solves the LP relaxation and rounds every fractional variable to the nearest integer. Since LP relaxations can potentially be very costly to solve, we propose a quick heuristic, Algorithm 2, that tries to minimize the infeasibility of a starting point. Within each iteration, subsets of variables are fixed to random integer values within bounds while the remaining ones are optimized towards feasibility. The algorithm terminates once all integer variables are fixed.

Starting with a sorted list of variables by increasing bound range and an input parameter θ, the algorithm proceeds to fix the top $\theta\%$ of variables to a random integer value within their bounds. It is possible that a variable may have an infinite bound, and

Algorithm 2 Starting vector heuristic

```
Require: Percentage of variables to fix $\theta$, $0 < \theta \leq 100$, Fixed bound constant $c_b$
Ensure: Starting integer-feasible vector $\hat{x}$
$V$:=list of integer variables sorted by increasing bound range $u - l$
$\mathcal{F} := \emptyset$
while $\hat{x}$ is not integer feasible and $\mathcal{F} \neq \mathcal{I}$ do
    $\mathcal{K}$:= top $\theta$ % of unfixed variables from $V$
    for $k \in \mathcal{K}$ do
        $\hat{x}_k$ :=random integer value between $[\max(l_k, -c_b), \min(u_k, c_b)]$
    end for
    $\mathcal{F} := \mathcal{F} \cup \mathcal{K}$
    $[x, \Delta^+, \Delta^-] := min\{\sum_i \Delta_i^+ + \Delta_i^- | Ax + I_m\Delta^+ - I_m\Delta^- = b, x_j = \hat{x}_j \ \forall j \in \mathcal{F}\}$
    $\mathcal{Q}$:= index set of integer variables of $x$ with integer value
    $\hat{x}_q = x_q, \forall q \in \mathcal{Q}$
    $\mathcal{F} := \mathcal{F} \cup \mathcal{Q}$
end while
$return\ \hat{x}$
```

therefore, an infinite range. In this case, the infinite bound is replaced by a constant input parameter c_b. For our practical purposes, this was determined to be 10^6. With sorting, the goal is to drive binary variables integer first, given that binary decisions force integrality on other variables. Until all integer variables are fixed, the LP relaxation of FMIP is solved in order to optimize the unfixed variables towards feasibility. Consecutively, the variables that become integer are fixed. A minimum of $\theta\%$ variables are fixed at every iteration. Hence, the algorithm will require at most $\lceil \frac{100}{\theta} \rceil$ iterations to converge to a starting solution. The θ parameter controls a tradeoff of difficulty of the LP relaxations against the quality of the starting feasible solution.

2.3 The variable fixing scheme

The variable fixing scheme determines the difficulty and the efficacy of each large neighborhood search. A desirable quality of the fixings is that they must not be too restrictive, as very few improvements will be found. At the same time, if not enough variables are fixed, the search space might be too large to find any improvements in a small amount of time. Neighborhood diversification is another key property, since parallel efficiency depends on it. In order to generate a large number of diverse neighborhoods early in the search, we use randomization instead of branch-and-bound information.

Devising a general method for fixing variables may be challenging, since there are many problem structures to consider. Problems range in structure, constraint matrix shape, number, and kinds of variables. Given these requirements, we propose a simple, yet intuitive variable fixing algorithm. It incorporates randomness, in order to satisfy the need for diversity and it allows the fixing of an adjustable number of variables. As shown in Algorithm 3, fixings are determined by selecting a random integer variable x' and fixing a consecutive set of integer variables starting from x' up to a certain cap determined by an input parameter ρ. If the end of $\mathcal{I}$ is reached before enough variables are chosen, the algorithm continues the selection starting from the beginning of the set in a circular way.

Algorithm 3 Variable Fixing Selection Algorithm

```
Require: Fraction of variables to fix $\rho$, $0 < \rho < 1$
Ensure: Set of integer indices $\mathcal{F}$
function RandomFixings($\rho$)
    $i$:= random element in $\mathcal{I}$
    $\mathcal{F}$:= first $\rho \cdot |\mathcal{I}|$ consecutive integer variable indices starting from $i$ in a circular fashion
    return $\mathcal{F}$
end function
```

For any MIP we consider, its formulation is usually structured, and its variables are arranged consecutively depending on their type and their logical role. Consecutive sets of variables often belong to cohesive substructures within a problem. This is the case in network flow and routing problems where flow assignations for a particular entity are formulated successively. Similar properties can be found in formulations for scheduling problems. In our experience, our proposed variable selection often produces an easier subMIP, in which the fixings affect a subset of contiguous substructures and the remaining ones are left unfixed. Due to its simplicity, it is an efficient variable selection strategy. However, any permutation of rows and columns alters its effectiveness as well as the solving process, as discussed in [17].

2.4 Framing alternating criteria search within an exact algorithm

In the current parallel branch-and-bound paradigm, threads are focused on solving multiple subproblems in parallel, which has shown poor scalability [38]. We propose an alternative use of the parallel resources by decoupling the search for high quality solutions from the lower bound improvement. Thus, a subset of the threads are allocated to an instance of the branch-and-bound solver focused on improving the lower bound, and our Alternating Criteria Search replaces the traditional primal heuristics. In the process, our parallel heuristic searches for solutions to the entire problem regardless of the variable fixings produced in the branch-and-bound tree, and supplies them to the solver. Both algorithms proceed to run concurrently until a time limit or optimality is reached. Communication between the parallel heuristic and the branch-and-bound is performed via MPI collective communications and new feasible solutions are added back via callbacks.

3 Experimental results

In this section, we evaluate the performance and behavior of Parallel Alternating Criteria Search (PACS) in terms of solution quality, scalability, reproducibility and parallel efficiency. The framework is implemented in C++, using CPLEX 12.6.1 as a backbone solver. We compare our framework against different configurations of the state-of-the-art general purpose MIP solver CPLEX 12.6.1. Out of the 361 instances from the MIPLIB 2010 library [37], we select as a benchmark those 333 for which feasibility has been proven as a benchmark. MIPLIB classifies such instances by difficulty based on the time required to reach optimality. 206 easy instances are defined as the subset in which optimality has been proven in less than 1 h by at least one of the tested MIP solvers. 54 additional instances have also been solved, but not under the previous conditions (hard instances). The remaining 73 unsolved instances are classified as open. All of our computations are performed on an 8-node computing cluster, each with two Intel Xeon X5650 6-core processors and 24 GB of memory.

3.1 Automating the choice of parameters

Three main parameters regulate the difficulty and the solution time of each LNS within the heuristic, and their appropriate selection is crucial for performance. We heuristically calibrate the settings for each instance automatically by executing a single iteration of the algorithm under multiple independent sample configurations and with the same initial solution. Each iteration run is performed in parallel and the best performing one is selected for the full run. The parameter θ regulates the number of variables to be fixed during the initial solution generation process. Depending on the difficulty of the instance, θ is chosen from the set $\{1, 5, 10, 20, 50, 100\%\}$. θ aside, parameters $[\rho, t]$ determine the percentage of variables to be fixed and the time limit in each LNS. In this case, the configurator chooses a subset of the permutations $\rho \in \{5, 10, 20, 50, 75, 95\%\}$ and $t \in \{5, 20, 50\,\text{s}\}$. A total of 6 initial configurations for θ are tested, in addition to a total of 16 permutations of $[\rho, t]$. The time required for calibration is not counted in the final execution time. The chosen configuration is the one which delivers the largest improvement per unit of time. On average, the calibration process takes 126s due to the fact that the calibration is run in parallel. With the intention of drawing a fair comparison, CPLEX is set to its parallel distributed-memory setting using 96 cores and allowed instance-specific tuning for the same amount of calibration time prior to the search. In our experience, instance-specific tuning certainly helps CPLEX improve its performance in most of the small instances. However, the time allowed seems to be insufficient to correctly tune for a subset of larger instances. In such cases, the parameters chosen result in inferior runs compared to the default setting. We opt to combine all results, and select the best run of CPLEX: either default or tuned. When comparing primal bounds, the default setting is on primal solutions (the emphasis on *hidden feasible solutions* setting). Settings are default otherwise.

3.2 Evaluation of primal solution quality

We evaluate the quality of primal solutions in terms of the metrics introduced in [10]. Given a solution x for a MIP with an optimal solution $\hat{x}$, the primal gap $\gamma(x) \in [0, 1]$ of x is defined as:

$$\gamma(x) = \begin{cases} 0 & \text{if } |c^T\hat{x}| = |c^T x| = 0 \\ 1 & \text{if } \quad c^T\hat{x} \cdot c^T x < 0 \\ \frac{|c^T\hat{x} - c^T x|}{\max\{|c^T\hat{x}|, |c^T x|\}} & \text{else.} \end{cases} \tag{1}$$

Given a time limit t_{max}, we define the primal gap function $p : [0, t_{max}] \mapsto [0, 1]$ as:

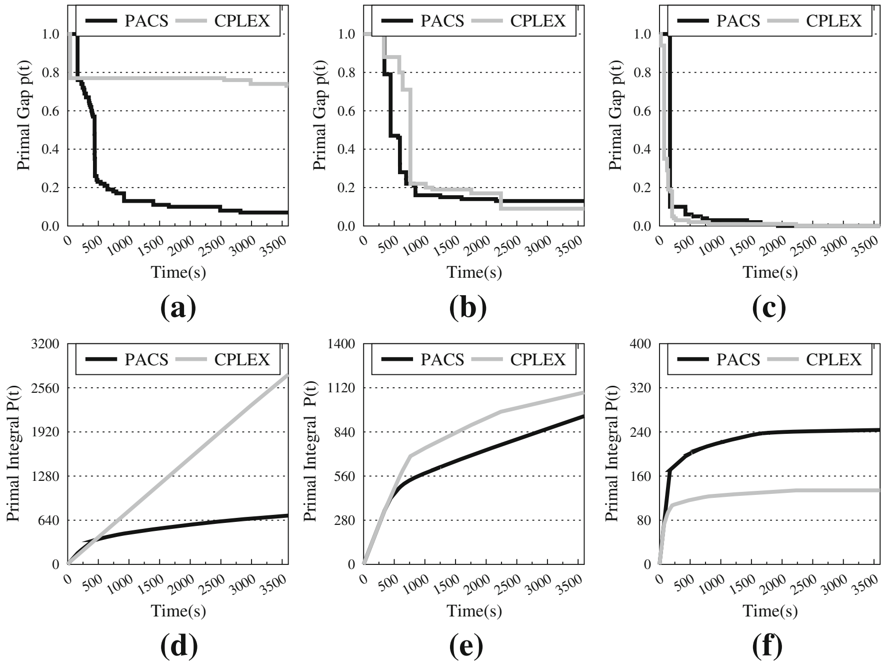
$$p(t) = \begin{cases} 1 & \text{if no solution is found until point } t \\ \gamma(x(t)) & \text{with x(t) being the incumbent solution at point } t\text{, else.} \end{cases} \tag{2}$$

The primal gap function is monotonically decreasing and allows to depict the progress of the optimization towards the optimal solution. The primal integral is defined as:

$$P(t) = \int_{t=0}^{T} p(t)\mathrm{d}t. \tag{3}$$

The primal integral $P(t)$ captures the notion of how early solutions are found. Both $p(t)$ and $P(t)$ promise to be powerful metrics to evaluate the performance of finding high quality primal solutions. To illustrate their use, we introduce three examples in Fig. 4, in which the performance of both CPLEX and PACS are plotted with respect to both metrics.

For rail03, PACS is able to find primal solutions of higher quality than those found by CPLEX. This results in a lower primal gap profile, as well as a slower increasing primal integral function. In the second problem, CPLEX finds a better solution at the end of the optimization. Its primal integral function value, however, is higher than



(a) (b) (c)

(d) (e) (f)

Fig. 4 Primal gap function p(t) and primal integral P(t) of the solutions provided by CPLEX and PACS for the **a**, **d** rail03, **b**, **e** shs1023 and **c**, **f** sing245 problems

the one for PACS because PACS is able to find relatively better solutions earlier in the search. For sing245, both schemes find the optimal solution, as shown by the flat profile of the primal integral. However, CPLEX does so earlier in the search, thus resulting in a lower integral value.

The following set of results evaluate the quality of the provided solutions for the proposed set of MIPLIB2010 instances. In Fig. 5, we show a performance profile that illustrates the differences in primal gap output of both schemes. Let $p(t)_{\mathrm{CPX}}$ and $p(t)_{\mathrm{PACS}}$ be the primal gap functions of CPLEX and our parallel heuristic respectively after a time limit t. We report improvements in terms of the difference:

$$Improvement = (p(t)_{\mathrm{CPX}} - p(t)_{\mathrm{PACS}}) \times 100.$$

Figure 5 displays the differences between the primal gap functions found by both methods after different time limits. Specifically, each performance curve measures the cumulative percentage of instances among the total that reach or surpass certain amounts of difference in favor of PACS (Improvement) or CPLEX (Deterioration). For example, PACS produced solutions with a primal gap that was at least 10% better after 180 seconds for 13% of the problem instances. In contrast, only 4% of the instances yield a worse primal gap of 10% or more in the same amount of time. Both schemes tie or show differences between −0.1 and 0.1% for the remaining percentage of instances.

Results show that our heuristic performs better for a substantial number of instances, whereas worse solutions are found in a relatively smaller subset of cases. One of our primary focuses is the ability to provide high quality solutions early in the search. After

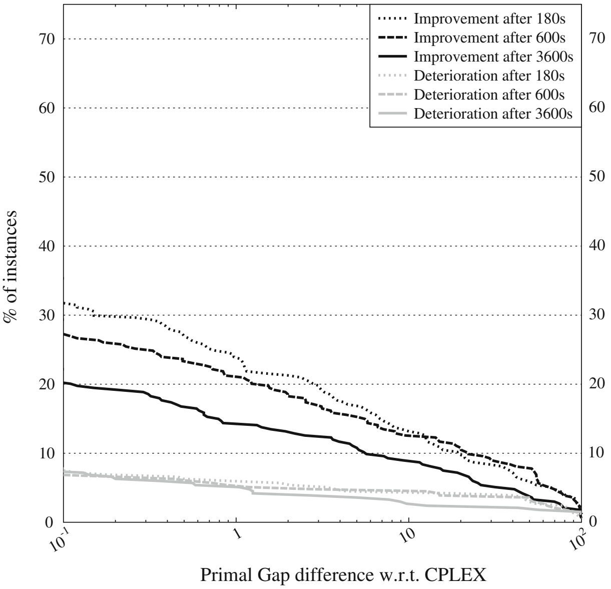


Fig. 5 Improvement and deterioration of primal gap w.r.t. CPLEX for all instances

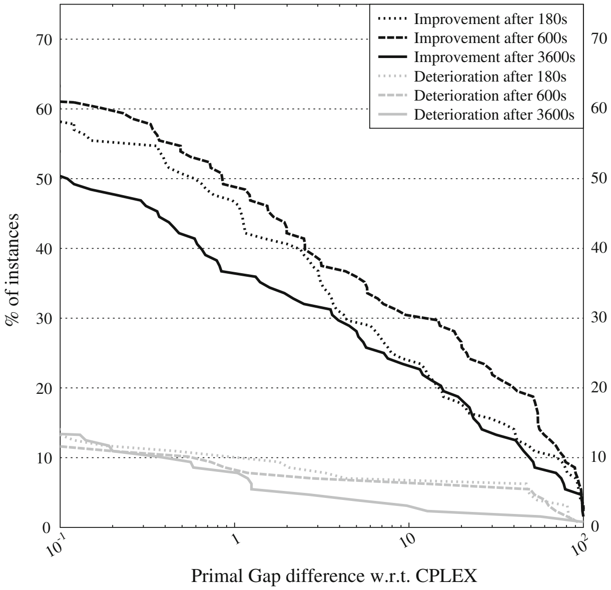


Fig. 6 Improvement and deterioration of primal gap in hard and open instances w.r.t. CPLEX

3 min of execution, PACS provides solution improvements for 32% of the instances. At the time limit of 1 h, this percentage decreases to 20% after CPLEX neutralizes the advantage for some of the instances. In contrast, worse solutions are only obtained in 7% of the cases.

In Fig. 6, the comparison is restricted exclusively to hard and open instances. Results show the scalability of our heuristic in hard MIPs, as more than 58% of the instances yield improvements after 3 min of execution. A comparison between the 180 and the 600 s profiles indicate that the competitive advantage of PACS is sustained and increased throughout a large portion of the runtime.

Figure 7 depicts a comparison of the primal integral. Given a time t, we define $P(t)_{\text{CPX}}$ and $P(t)_{\text{PACS}}$ as the primal integrals provided by CPLEX and PACS respectively at time t. For improved readability, we compare the improvements in terms of the scaled difference:

$$Improvement = \frac{P(t)_{\text{CPX}} - P(t)_{\text{PACS}}}{t} \times 100$$

The difference between primal integrals is scaled by t in order to be able to plot profiles belonging to different time cutoffs.

The performance profiles indicate that PACS finds solutions earlier in the search, resulting in significantly lower primal integral profiles for a majority of the instances. After 600 s, the primal integral values show an improvement for 61% of the instances. The performance profiles shift to the left as time advances, indicating that the advantage of PACS is reduced. When only hard or open instances are considered, performance profiles are clustered together and further to the right, indicating that

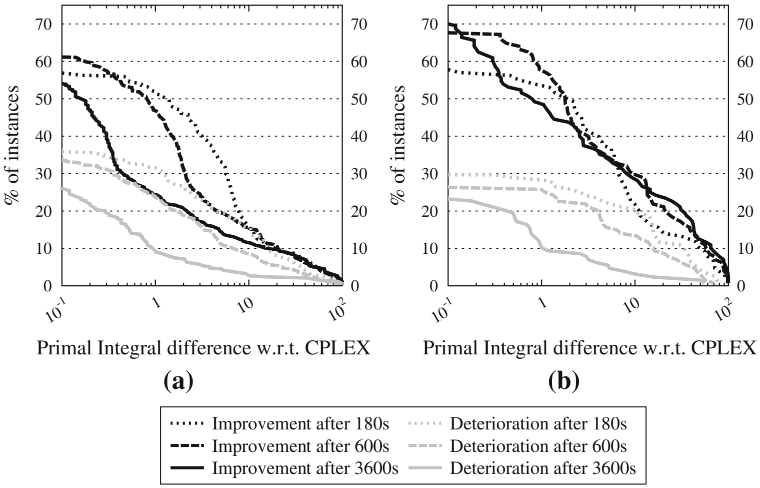


Fig. 7 Improvement and Deterioration in primal integral w.r.t. CPLEX for **a** all and **b** hard and open instances

the advantage of PACS is sustained through time. After 1 h, PACS showed better primal integrals for more than 54% of the instances. The plots suggest that, not only does PACS find better solutions for most of the instances, but it does so faster.

3.3 Framing alternating criteria search within an exact algorithm

We test the performance of our heuristic when run in combination with an exact branch-and-bound algorithm. In this hybrid scheme, a fraction of the cores are dedicated to improving the primal incumbent and the rest are commited to computing the lower bound in the tree search.

We give a direct performance comparison between parallel memory-distributed CPLEX using 96 threads, and our combined scheme. In the latter, 84 cores are allocated to the parallel heuristic while the remaining 12 threads are allocated to CPLEX in shared-memory parallel mode. The comparison is in terms of the difference between the optimality gap provided by both algorithms: $Gap_{\mathrm{CPX}} - Gap_{\mathrm{PACS}}$. For any of the two algorithms X, its optimality gap may be defined in terms of its best found upper and lower bound, UB_X and LB_X:

$$Gap_X = \frac{UB_X - LB_X}{UB_X} \times 100.$$

Figure 8 shows multiple performance profiles for different time cutoffs. Both approaches outperform each other for different instance subsets, but our combined scheme performs better in more instances. After 10 min, our algorithm provides a better gap for over 30% of the instances, while the opposite is true for 18% of the instances. We observe a similar shift in performance when only the hard and open

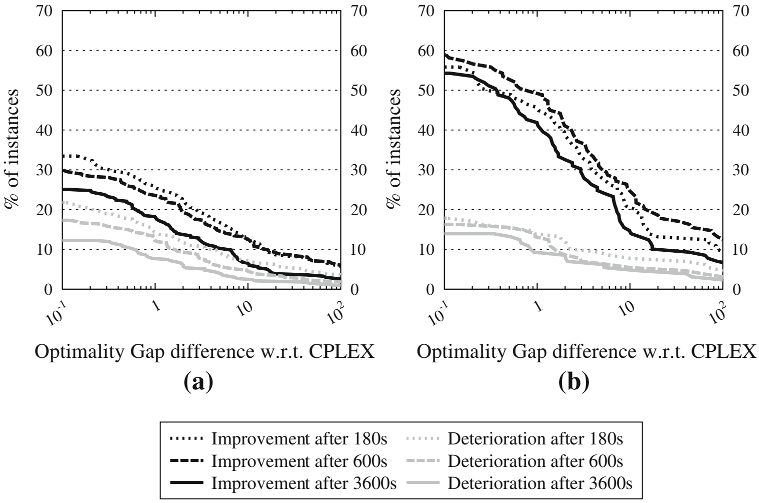


Fig. 8 Improvement and deterioration of optimality gap w.r.t. CPLEX for **a** all and **b** hard and open instances

instances are considered. In this case, our ability to produce solutions of high quality early in the search allows us to achieve a smaller optimality gap for over 55% of the instances after 10 min. At termination, over 14% of the instances show a competitive advantage of over 10% of the gap. In contrast, CPLEX shows better performance for at most 5% of the instances.

The two tables shown below give an overview of the performance of all tested methods. Table 1 highlights the number of instances where the best solution was found, and the subset of instances for which the best solution was also optimal. PACS is able to find the optimal solution in more instances, and delivers the best solution for a larger number of instances. This advantage also translates to hard and open instances.

Table 2 compares performance from the perspective of optimality. CPLEX is able to prove optimality for a larger number of instances, given that it has more processors dedicated to the branch-and-bound. Because better primal solutions are found by the combined scheme, however, smaller gaps are obtained for more instances in which optimality can't be proven.

Table 1 Primal solution performance comparison

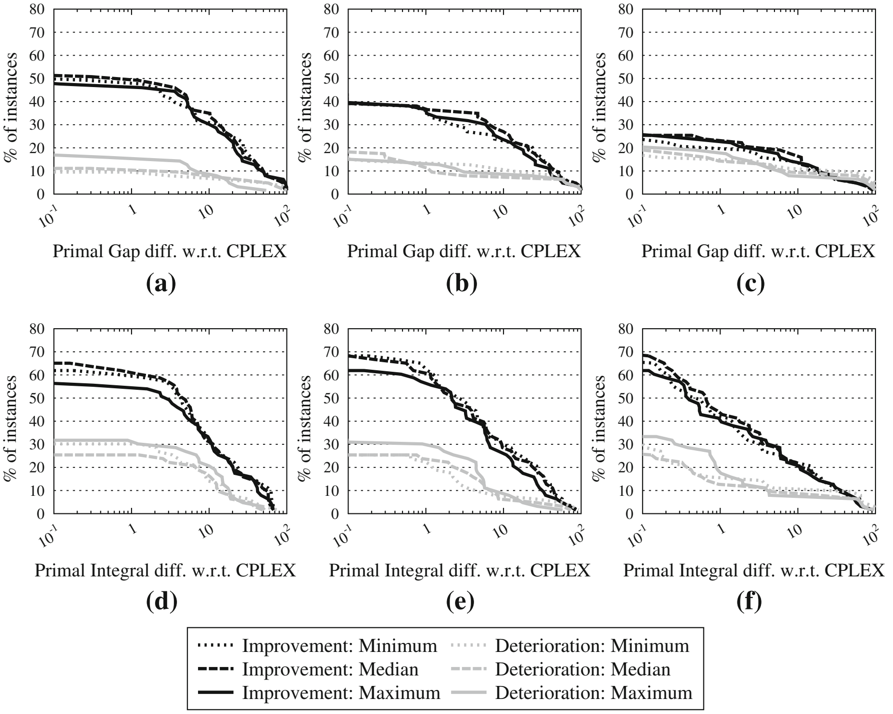
Computing method	All problems		Hard and open problems	
	Optimal solutions	Best solutions	Optimal solutions	Best solutions
PACS	219	260	28	69
Combined scheme	194	224	25	55
CPLEX (emph. on hidden sols)	183	193	16	26
CPLEX (balanced emphasis)	183	194	17	32

Table 2 Optimality performance comparison

Time cutoff (s)	Combined scheme			CPLEX (balanced emphasis)		
	180	600	3600	180	600	3600
Avg. gap (%) (all inst.)	22.74	16.58	12.87	25.59	21.44	14.67
Avg. gap (%) (hard and open inst.)	42.16	33.49	28.32	50.34	45.34	33.76
Instances solved (all inst.)	79	93	103	107	114	124
Avg. time to opt. (s) (all inst.)			242			143

3.4 The impact of nondeterminism

Our approach is nondeterministic by nature due to the different time limits and the parallel synchronization required for the solution recombination. We have demonstrated its performance in comparison to CPLEX, which is deterministic in its default setting. When in deterministic mode, most parallel MIP solvers must sacrifice a fraction of the performance in order to ensure a proper repeatibility of the results. In order to assess the impact of nondeterminism, we compare the performance of our approach with CPLEX running in Opportunitistic mode, which is non-deterministic. In the experiments shown below, we evaluate the consistency of 5 runs for each feasible instance



(a) **(b)** **(c)**

(d) **(e)** **(f)**

Fig. 9 Performance profiles for all instances in the Reoptimize set. In each of the charts *multiple lines* are drawn for the best, mean, and worst executions. Charts show the performance for different cutoffs: **a**, **d** 180 s, **b**, **e** 600 s, and **c**, **f** 3600 s

Table 3 Performance comparison of non-deterministic approaches

Time cutoff (s)			Parallel alternating criteria search			CPLEX (opportunistic mode)		
			180	600	3600	180	600	3600
Avg.	P. gap (%)	All inst.	24.86	17.01	11.47	35.97	22.95	10.58
		H&O inst.	31.77	21.50	12.39	45.73	35.35	16.06
	P. integral	All inst.	77.20	162.60	528.50	91.11	210.18	648.57
		H&O inst.	88.75	99.29	602.31	103.99	277.88	954.45
SD	P. gap(%)	All inst.	3.16	2.73	1.32	3.23	0.71	1.74
		H&O inst.	3.48	3.78	1.10	0.89	0.95	1.53
	P. integral	All inst.	5.64	17.35	58.85	3.26	9.10	45.99
		H&O inst.	5.26	20.00	57.00	1.59	4.96	38.28
Min.	P. gap (%)	All inst.	22.14	14.56	10.50	32.80	21.92	8.86
		H&O inst.	28.33	17.51	11.21	44.77	33.86	14.19
	P. integral	All inst.	70.98	146.27	479.46	87.54	200.55	591.79
		H&O inst.	83.02	179.23	539.64	101.74	271.08	904.34
Med.	P. gap(%)	All inst.	24.03	16.59	11.06	35.65	23.13	9.95
		H&O inst.	30.94	21.31	12.22	45.77	35.49	16.02
	P. integral	All inst.	76.84	160.06	516.95	91.42	208.43	651.69
		H&O inst.	88.67	197.20	600.70	104.18	278.34	957.46
Max.	P. gap (%)	All inst.	29.25	20.67	13.49	39.69	23.68	12.48
		H&O inst.	36.05	25.96	13.75	46.65	36.36	17.79
	P. integral	All inst.	84.23	187.19	617.50	95.02	222.05	699.69
		H&O inst.	95.17	227.01	678.11	105.73	283.32	998.57

in the Reoptimize set, which is a subset of 63 easy, hard, and open instances from the MIPLIB2010 library.

Figure 9 depicts a comparison of the primal gap and primal integral obtained with both methods. For each of the charts shown, multiple lines are depicted, showing the performance when the the best, the mean, and the worst run is selected. When all instances in the Reoptimize set are considered, PACS shows better performance earlier in the search, as reflected in the primal integral. After 1 h, PACS shows a better primal integral for 62% of the instances, in the worst execution. CPLEX is competitive in finding solutions at the end of the execution, as reflected by chart 1(c), as it finds better or equal solutions for 75% of the instances, where PACS does so for 80% of them.

Table 3 reports the statistical results in terms of the average, standard deviation, and median of the primal gap and primal integral for different time cutoffs.

3.5 Additional performance tests: scalability and parallel efficiency

We examine the performance of Alternating Criteria Search from the perspective of scalability and parallel efficiency. Due to the large amount of experiments required to

Table 4 Selected instances for scaling experiments

Easy instances	Hard instances	Open instances
a1c1s1	atm20-100	momentum3
bab5	germanrr	nsr8k
csched007	n3-3	pb-simp-nonunif
danoint	rmatr200-p5	t1717
map14	set3-20	ramos3

evaluate this set of metrics, we reduce the test bed to a representative subset of the instances chosen randomly. The 15 instances shown in Table 4 are selected from all three difficulty categories.

3.5.1 Parallel scalability

Strong scalability is the ability to increase the algorithm's performance proportionally when parallel resources are incremented while the problem size remains fixed, and a particularly relevant metric is the speedup to cut off: a comparison of the time required by different processor configurations to reach certain solution quality. Let T_c^p be the time required for the algorithm to reach a cut-off c when p processing nodes are used. We define the parallel speedup to cut off S_{SC} with respect to a baseline configuration B as $S_{SC} = \frac{T_c^B}{T_c^p}$. In Fig. 10, we show the speedup demonstrated by the heuristic for several processor configurations ranging from a baseline of 12 cores to a total of 96. Among the runs for different processor configurations, the cutoff was determined to be the objective value of the best solution achieved by the worst performing run.

In terms of scaling, our heuristic shows a variable performance dependent on the characteristics of each individual instance. This behavior is expected since increasing the number of simultaneous searches does not guarantee a translation to faster improvements. In general, however, speedups are achieved more consistently as the difficulty of the problem increases. The addition of more cores sometimes exhibits a

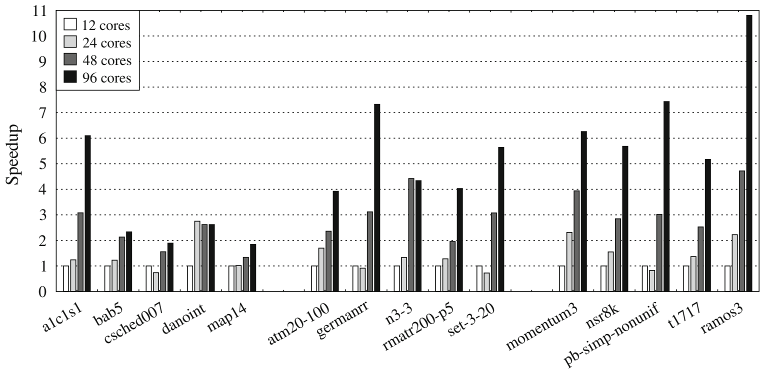


Fig. 10 Parallel strong-scaling results

multiplicative effect. In most small instances, optimality is achieved quickly for all processor configurations, thus resulting in small speedup values.

3.5.2 Parallel load balancing

Load balancing is the property that measures the degree of uniformity of the work distribution among processors. Given the synchronous nature of our approach, an even difficulty of the subproblems is essential in order to ensure all processors optimize for an equivalent amount of time. In the case of the parallel heuristic, the size of each subproblem is regulated by the number of fixed variables and the imposed search time limit. Hence, a proper calibration of these parameters must ensure an even distribution of the workload.

The load balance of a parallel application can be evaluated as follows: Let the total execution time of a processor P_i be defined as the sum of the time spent performing useful computations (TU_{P_i}), communications (TC_{P_i}) and the synchronization time TS_{P_i} spent waiting for other processors to complete their computations. Then, we characterize the utilization of a processor U_{P_i} as the ratio of useful computation time over the total execution time:

$$U_{P_i} = \frac{TU_{P_i}}{TU_{P_i} + TS_{P_i} + TC_{P_i}}.$$

We believe hard instances represent the worst-case scenario, since these are the ones that require the most computational effort and prolonged optimization times. Figure 11 shows the average core utilization for the hard instance momentum3. Performance results are displayed for different processor configurations as well as different time limit parameters. Time limit configurations are denoted as $C_{T_{LNS}-T_R}$, where T_{LNS} is the time allowed for each search and T_R is the time allowed to the recombination step. Results show that processors sustain high utilizations (above 95%) throughout the execution, even when large processor configurations are used. The setting with the shortest solution times remains the most efficient because smaller time penalties are paid due to early terminations.

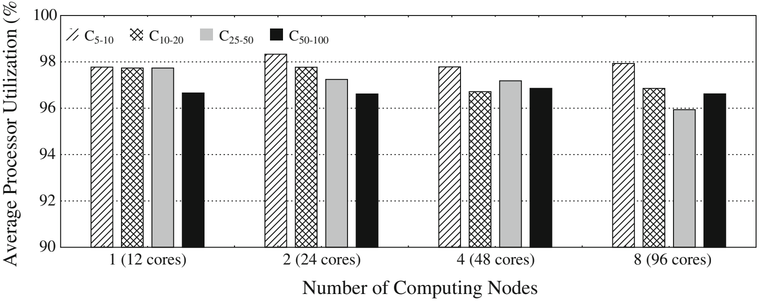


Fig. 11 Parallel synchronization overhead in terms of average processor utilization for different parameter configurations

3.5.3 The impact of the starting heuristic

In Sect. 2.2, we described a quick heuristic for finding a starting solution suitable for Alternating Criteria Search. The objective of this strategy is to provide a starting point for the algorithm that is as feasible as possible with respect to the objective function of FMIP. An iterative algorithm is proposed, in which successive restricted LP relaxations are solved, the difficulty of which depend on a variable fixing parameter ρ.

Figure 12 illustrates the behavior of the starting heuristic for four problem instances, as the variable fixing parameter ρ varies. For the instances shown, the infeasibility of the provided starting solution increases as more variables are randomly fixed per iteration. However, the algorithm also becomes faster, as a direct consequence of reducing the difficulty of the LP relaxations. For the evaluated test set, a good compromise can usually be found, in which better solutions than the ones provided by random fixings ($\rho = 1$) are found, at the expense of slightly longer runtimes.

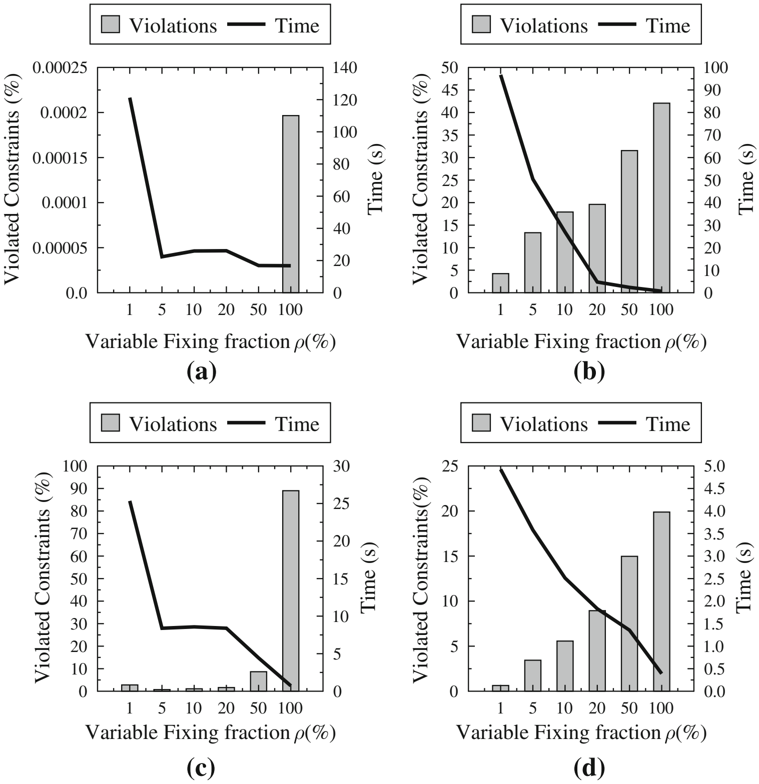


Fig. 12 Performance tradeoffs shown by the starting heuristic, in which the time to reach the initial solution is contraposed to its quality as a function of the variable fixing parameter ρ for the **a** in, **b** rail03, **c** shs1023 and **d** sing245 problems

4 Conclusions

The combination of parallelism and simple large neighborhood search schemes can provide a powerful tool for generating high quality solutions. The heuristic becomes especially useful in the context of large instances and time-sensitive optimization problems, where traditional branch-and-bound methods may not be able to provide competitive upper bounds and attaining feasibility may be challenging. Our method is a highly versatile tool, as it can be applied to any general MIP as a standalone heuristic or in the context of an exact algorithm. Many algorithmic ideas contribute to the competitiveness of our approach, such as the use of two auxiliary MIP transformations and fixing diversifications as the main source of parallelism. The proposed algorithm could benefit from further improvements. Besides the parallelization, we introduced general strategies for addressing the choice of starting solutions and variable fixings in the context of general MIPs. However, their specification is independent of the parallel framework. Thus, it could be possible to substitute them for more effective fixings when considering specific classes of problems with defined structures, such as capacitated network design [35,40], lot sizing [39] or Maritime Inventory Routing Problems [31, 43]. We hope this work sparks interest and motivation to investigate whether better parallelizations are possible in order to speed up the optimization process. In the future, we plan to evaluate other components of the branch-and-bound search that may profit from parallelism.

Acknowledgements We wish to thank the referees, whose comments led to an improved version of the paper.

References

1. Achterberg, T.: Constraint integer programming. Ph.D. thesis (2007)
2. Achterberg, T., Berthold, T.: Improving the feasibility pump. Discrete Optim. **4**(1), 77–86 (2007)
3. Achterberg, T., Berthold, T., Hendel, G.: Rounding and propagation heuristics for mixed integer programming. In: Operations Research Proceedings, 2011, pp. 71–76. Springer (2012)
4. Bader, D.A., Hart, W.E., Phillips, C.A.: Parallel algorithm design for branch and bound. In: Tutorials on Emerging Methodologies and Applications in Operations Research, Chap. 5. Springer (2005)
5. Baena, D., Castro, J.: Using the analytic center in the feasibility pump. Oper. Res. Lett. **39**(5), 310–317 (2011)
6. Balas, E., Ceria, S., Dawande, M., Margot, F., Pataki, G.: Octane: a new heuristic for pure 0–1 programs. Oper. Res. **49**(2), 207–225 (2001)
7. Balas, E., Schmieta, S., Wallace, C.: Pivot and shift-a mixed integer programming heuristic. Discrete Optim. **1**(1), 3–12 (2004)
8. Bertacco, L., Fischetti, M., Lodi, A.: A feasibility pump heuristic for general mixed-integer problems. Discrete Optim. **4**(1), 63–76 (2007)
9. Berthold, T.: Primal heuristics for mixed integer programs. Diploma Thesis, Technische Universitat Berlin (2006)
10. Berthold, T.: Measuring the impact of primal heuristics. Oper. Res. Lett. **41**(6), 611–614 (2013)
11. Berthold, T.: Rens. Math. Program. Comput. **6**(1), 33–54 (2014)
12. Berthold, T., Hendel, G.: Shift-and-propagate. J. Heuristics **21**(1), 73–106 (2015)
13. Bixby, R., Gu, Z., Rothberg, E.: Gurobi optimization (2010). http://www.gurobi.com/
14. Boland, N.L., Eberhard, A.C., Engineer, F.G., Fischetti, M., Savelsbergh, M.W.P., Tsoukalas, A.: Boosting the feasibility pump. Math. Program. Comput. **6**(3), 255–279 (2014). doi:10.1007/s12532-014-0068-9

15. Carvajal, R., Ahmed, S., Nemhauser, G., Furman, K., Goel, V., Shao, Y.: Using diversification, communication and parallelism to solve mixed-integer linear programs. Oper. Res. Lett. **42**(2), 186–189 (2014)
16. Corporation, I.B.M.: Ibm cplex optimizer (2015). http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/
17. Danna, E.: Performance variability in mixed integer programming. In: Presentation at Workshop on Mixed Integer Programming. http://coral.ie.lehigh.edu/mip-2008/talks/danna.pdf (2008)
18. Danna, E., Rothberg, E., Le Pape, C.: Exploring relaxation induced neighborhoods to improve mip solutions. Math. Program. **102**(1), 71–90 (2005)
19. Fischetti, M., Glover, F., Lodi, A.: The feasibility pump. Math. Program. **104**(1), 91–104 (2005)
20. Fischetti, M., Lodi, A.: Local branching. Math. Program. **98**(1–3), 23–47 (2003)
21. Fischetti, M., Lodi, A.: Repairing mip infeasibility through local branching. Comput. Oper. Res. **35**(5), 1436–1445 (2008). (Part Special Issue: Algorithms and Computational Methods in Feasibility and Infeasibility)
22. Fischetti, M., Lodi, A.: Heuristics in Mixed Integer Programming. Wiley, London (2010)
23. Fischetti, M., Lodi, A., Monaci, M., Salvagnin, D., Tramontani, A.: Improving branch-and-cut performance by random sampling. Math. Program. Comput. **8**(1), 113–132 (2016)
24. Fischetti, M., Monaci, M.: Proximity search for 0–1 mixed-integer convex programming. J. Heuristics **20**(6), 709–731 (2014)
25. Fischetti, M., Salvagnin, D.: Feasibility pump 2.0. Math. Program. Comput. **1**(2–3), 201–222 (2009)
26. Gamrath, G., Berthold, T., Heinz, S., Winkler, M.: Structure-based primal heuristics for mixed integer programming, pp. 37–53. Springer, Japan, Tokyo (2016)
27. Ghosh, S.: Dins, a mip improvement heuristic. Integer Programming and Combinatorial Optimization. Lecture Notes in Computer Science, vol. 4513, pp. 310–323. Springer, Berlin, Heidelberg (2007)
28. Glover, F., Laguna, M.: General purpose heuristics for integer programming–part i. J. Heuristics **2**(4), 343–358 (1997)
29. Glover, F., Laguna, M.: General purpose heuristics for integer programming–part ii. J. Heuristics **3**(2), 161–179 (1997)
30. Glover, F., LøKketangen, A., Woodruff, D.L.: Scatter Search to Generate Diverse MIP Solutions, pp. 299–317. Springer, Boston (2000)
31. Goel, V., Furman, K.C., Song, J.H., El-Bakry, A.S.: Large neighborhood search for lng inventory routing. J. Heuristics **18**(6), 821–848 (2012)
32. Gropp, W., Lusk, E., Doss, N., Skjellum, A.: A high-performance, portable implementation of the MPI message passing interface standard. Parallel Comput. **22**(6), 789–828 (1996)
33. Hansen, P., Mladenović, N., Urošević, D.: Variable neighborhood search and local branching. Comput. Oper. Rese. **33**(10), 3034–3045 (2006). (Part Special Issue: Constraint Programming)
34. Hendel, G.: New rounding and propagation heuristics for mixed integer programming. Bachelor's thesis, TU Berlin (2011)
35. Hewitt, M., Nemhauser, G.L., Savelsbergh, M.W.P.: Combining exact and heuristic approaches for the capacitated fixed-charge network flow problem. INFORMS J. Comput. **22**(2), 314–325 (2010)
36. Koc, U., Mehrotra, S.: Generation of feasible integer solutions on a massively parallel computer (submitted)
37. 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., Steffy, D.E., Wolter, K.: MIPLIB 2010. Mathematical Programming Computation **3**(2), 103–163 (2011)
38. Koch, T., Ralphs, T., Shinano, Y.: Could we use a million cores to solve an integer program? Math. Methods Oper. Res. **76**(1), 67–93 (2012)
39. Mercé, C., Fontan, G.: Mip-based heuristics for capacitated lotsizing problems. Int. J. Prod. Econ. **85**(1), 97–111 (2003). (Planning and Control of Productive Systems)
40. Munguía, L.M., Ahmed, S., Bader, D.A., Nemhauser, G.L., Goel, V., Shao, Y.: A parallel local search framework for the fixed-charge multicommodity network flow problem. Comput. Oper. Res. **77**, 44–57 (2017)
41. Naoum-Sawaya, J.: Recursive central rounding for mixed integer programs. Comput. Oper. Res. **43**, 191–200 (2014)
42. Rothberg, E.: An evolutionary algorithm for polishing mixed integer programming solutions. INFORMS J. Comput. **19**(4), 534–541 (2007)

43. Shao, Y., Furman, K.C., Goel, V., Hoda, S.: A hybrid heuristic strategy for liquefied natural gas inventory routing. Transp. Res. C: Emerg. Technol. **53**, 151–171 (2015)
44. Shinano, Y., Achterberg, T., Berthold, T., Heinz, S., Koch, T.: Parascip: a parallel extension of scip. In: Competence in High Performance Computing, 2010, pp. 135–148. Springer (2012)
45. Thakur, R., Rabenseifner, R., Gropp, W.: Optimization of collective communication operations in mpich. Int. J. High Perform. Comput. Appl. **19**(1), 49–66 (2005)
46. Wallace, C.: Zi round, a mip rounding heuristic. J. Heuristics **16**(5), 715–722 (2010)