UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

UNIVERSITÀ DEGLI STUDI DI PADOVA

SCHOOL OF ENGINEERING DEPARTMENT OF INFORMATION ENGINEERING

MASTER DEGREE IN COMPUTER ENGINEERING

Enhancing the ACS Heuristic: Parameter Tuning and Computational Experiments for Solving MIP Problems

Supervisor:
PROF. DOMENICO SALVAGNIN

Candidate:
FRANCESCO BISCACCIA CARRARA
2120934

Academic Year 2024/2025
Date TBD

Abstract

Contents

Chapter 1

Introduction

1.1 Mixed Integer Programming (MIP)

Mixed Integer Programming (MIP) is a powerful optimization technique used to model complex real-world problems where discrete decisions are essential—such as resource allocation, scheduling, and logistics. MIP models aim to find the best solution according to a given objective function, which is either maximized or minimized. A generic mixed-integer program (MIP) will be defined as

$$\begin{cases} \min & c^T x \\ \text{s.t.} & Ax = b \\ & x_i \in \mathbb{Z},\ \forall i \in \mathcal{I} \\ & l \leq x \leq u \end{cases} \tag{1.1}$$

where $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$ and $\mathcal{I} \subseteq \{1 \dots n\}$ is the subset of integer variables indices. The solution vector x is bounded by $l \in \bar{\mathbb{R}}^n$ and $u \in \bar{\mathbb{R}}^n$ where $\bar{\mathbb{R}} = \mathbb{R} \cup \{-\infty, \infty\}$. Here, some decision variables are constrained to take integer values, which increases the model's complexity.

1.1.1 Heuristic in MIP solving

The presence of integer variables significantly increases the computational complexity of MIP problems, especially as the number of variables and constraints grows. Solvers such as IBM ILOG CPLEX[1] and GUROBI[2] incorporate a variety of built-in strategies to improve performance. Among these strategies are heuristics, which are specialized algorithms designed to quickly find feasible (but not necessarily optimal) solutions. Although heuristics do not guarantee optimality, they are computationally efficient and can significantly accelerate the solution process by providing good initial solutions or guiding the search within the solution space.

1.2 Alternating Criteria Search (ACS)

Finding high-quality feasible solutions is a key aspect of the discrete optimization process, and constructing them has been one of the main focuses of research in the last few decades. Primal heuristics differ in whether they require a starting feasible solution or not. In the first case, the heuristics are called starting heuristics, and state-of-the-art techniques rely on Large Neighborhood Search (LNS)[3] to find high-quality feasible solutions. The effectiveness of such strategies relies on the power of the MIP solver to solve the subproblems generated during the LNS heuristic. Improvement heuristics, in contrast, require a feasible starting solution and aim to improve its quality with respect to the objective. The simplest improvements are the 1-opt and 2-opt[4] methods, but—as with starting heuristics—there are many improvement heuristics based on LNS ideas. The neighborhoods explored are defined based on branch-and-bound information—such as the best incumbent and the LP relaxation. This is essentially a limitation in terms of diversification: the strategies may reach good solutions, but it is difficult to find them in the early stages of the search. Therefore, to address this issue, the Alternating Criteria Search (ACS)[5]—or Parallel ACS, its straightforward parallel implementation—can be utilized. The search neighborhoods are defined based on randomization instead of branch-and-bound information, allowing for a wider exploration of the search space. The aim of this strategy is to find high-quality feasible solutions at the beginning of the search, increasing the heuristic's effectiveness.

1.2.1 Parallelization of ACS

Parallelism can be exploited in MIP optimization to accelerate computation and enhance solver performance, in terms of execution time and scalability. The exploration of the branch-and-bound tree in parallel is now incorporated in most state-of-the-art MIP solvers—such as CPLEX or GUROBI—and it essentially consists of solving multiple subproblems of the original MIP simultaneously. However, this strategy may not scale well to a large number of cores[6]. To address this limitation, the proposed heuristic—Parallel ACS (PACS)—is a parallel algorithm that integrates elements of both starting solution and improvement heuristics, offering the capability of generating starting solutions and improving them with respect to the original objective. To leverage parallelism, Parallel ACS performs a large number of LNS runs simultaneously over a diversified set of neighborhoods, with the aim of increasing the chances of finding higher-quality solutions. After each LNS exploration, the strategy consolidates the local improvements through a recombination phase. PACS combines parallelism and diversified LNS in order to address large instances of MIPs arising from various application domains.

1.2.2 Limitations of the PACS Approach

Although the experiments about the PACS strategy bring evidence about the effectiveness on hard MIP instances, there are several considerations that must be addressed. First of all, the PACS algorithm was executed on an 8-node computing cluster, each equipped with two Intel Xeon X5650 6-core processors and 24 GB of memory, totaling 96 cores and 192 GB of RAM. Such computing resources are not typically available in consumer-grade or general-purpose environments. Furthermore, as discussed in the original study, the PACS strategy could assist the MIP solver in accelerating convergence. This suggests that integrating PACS heuristics within a MIP solver could provide a meaningful contribution to solver performance. Second, before the PACS execution, an initial calibration phase is required to tune the parameters prior to execution. This undermines the objective of designing a self-contained, efficient heuristic. With these considerations in mind, the goal of this thesis is to refine the PACS strategy to make it more compatible with MIP solvers. In particular, the aim is to establish a fixed, robust parameter configuration that remains effective regardless of the input instance.

Chapter 2

The ACS Framework

The Alternating Criteria Search (ACS) heuristic is designed to pursue a twofold objective: to identify a feasible solution and to subsequently enhance its quality with respect to the objective function $c^T x$. To this end, ACS employs a Large Neighborhood Search (LNS) strategy, which iteratively solves two auxiliary mixed-integer programming (MIP) subproblems in order to address both objectives. The heuristic requires an initial vector, which is not mandated to be a feasible solution for the original MIP. This vector is progressively refined by solving sub-MIPs in which a subset of variables is fixed to the values of the initial input.

2.1 Feasibility-MIP (FMIP)

In linear programming theory, it is well established that the feasibility problem can be addressed using the two-phase Simplex method, in which an auxiliary optimization problem is solved to obtain a feasible starting basis, and hence a feasible solution. In a similar manner, the following auxiliary MIP problem, denoted as the Feasibility-MIP (FMIP), is formulated to identify a feasible starting solution:

$$\begin{cases} \min & \sum_{i=0}^{m} \Delta_i^+ + \Delta_i^- \\ \text{s.t.} & Ax + I_m\Delta^+ - I_m\Delta^- = b \\ & x_i = \hat{x}_i,\ \forall i \in F \\ & l \leq x \leq u \\ & x_i \in \mathbb{Z},\ \forall i \in \mathcal{I} \\ & \Delta^+ \geq 0, \Delta^- \geq 0 \end{cases} \tag{2.1}$$

Here, I_m denotes an $m \times m$ identity matrix, while Δ^+ and Δ^- are vectors in R^m corresponding to the m constraints. These are introduced as slack variables, and the objective is to minimize their sum. Analogous to the two-phase Simplex method, a vector x is feasible for the original MIP if and only if it can be extended to a solution of value 0 for

the associated FMIP. Since solving an FMIP is as computationally demanding as solving the original problem, the neighborhoods are restricted by fixing a given subset F of the integer variables to the values of an input vector $[\hat{x}, \Delta^+, \Delta^-]$. Because of the introduction of slack variables, the vector $\hat{x}$ is not required to be a feasible solution itself, but it must be integral and within the variable bounds to preserve the feasibility of the model. In this way, the FMIP guarantees that feasibility is preserved under any arbitrary variable-fixing scheme. Once a feasible solution is obtained, any LNS-based improving heuristic—such as RINS[7], DINS[8], or local branching[9]—can then be applied to refine the solution.

2.2 Optimality-MIP (OMIP)

Rather than executing the FMIP until convergence to a feasible solution vector, the following auxiliary MIP problem, denoted as Optimality-MIP (OMIP), is designed to improve a partially feasible solution $[\hat{x}, \hat{\Delta}^+, \hat{\Delta}^-]$, which satisfies $\sum_{i=1}^{m}(\hat{\Delta}_i^+ + \hat{\Delta}_i^-) \neq 0$, with respect to the original objective $c^T x$:

$$\begin{cases} \min & c^T x \\ \text{s.t.} & Ax + I_m\Delta^+ - I_m\Delta^- = b \\ & \sum_{i=0}^{m} \Delta_i^+ + \Delta_i^- \leq \sum_{i=0}^{m} \hat{\Delta}_i^+ + \hat{\Delta}_i^- \\ & x_i = \hat{x}_i, \ \forall i \in F \\ & l \leq x \leq u \\ & x_i \in \mathbb{Z} \quad \forall i \in \mathcal{I} \\ & \Delta^+ \geq 0, \ \Delta^- \geq 0 \end{cases} \tag{2.2}$$

Analogous to the FMIP, the OMIP represents a reformulation of the original MIP model in which auxiliary slack variables are introduced for each constraint. This formulation enables the OMIP to enhance a solution even if it is not feasible for the original MIP. Moreover, to ensure that the optimal solution of the OMIP does not exceed the infeasibility of the input solution $\hat{x}$, an additional budget constraint is imposed, limiting the total slack to $\sum_{i=0}^{m} \hat{\Delta}_i^+ + \hat{\Delta}_i^-$.

By iteratively solving subproblems of both auxiliary MIPs, the ACS heuristic is designed to converge—although convergence is not formally guaranteed—to a high-quality feasible solution. By construction, infeasibility decreases monotonically after each iteration. However, the quality of the solution with respect to the original objective function may fluctuate.

2.3 Parallelization of ACS

The parallelization of ACS exploits parallelism by generating a diversified set of large neighborhood searches, which are solved simultaneously. Exploring multiple search neighborhoods in parallel is expected to increase the likelihood of identifying high-quality solutions. Following this parallelization step, the improvements obtained in parallel must be combined efficiently. To this end, an additional search subproblem is generated in which variables with identical values across different solutions are fixed. Consequently, the recombination phase constitutes a crucial step in PACS: by merging the improvements achieved during the parallel phase, the subsequent phase can explore a newly diversified set of large neighborhood searches based on the recombined solution, thereby enhancing the probability of further improvements. The pseudocode[1] illustrates the overall workflow. Each processor generates a set of randomized variable fixings and solves the associated

Algorithm 1 Parallel Alternating Criteria Search

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

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

Ensure: Feasible solution $\hat{x}$ if found
Initialize $[\hat{x}, \Delta^+, \Delta^-]$ as an integer solution
$T := numThreads()$
while $timeElapsed() \leq$ timeLimit **do**
  if $\sum_i \Delta_i^+ + \Delta_i^- > 0$ **then**
    for all threads $t_i \in \{0, T-1\}$ **in parallel do**
      F_{t_i} := randomized variable index subset, $F_{t_i} \subseteq \mathcal{I}$
      $[x^{t_i}, \Delta^{+t_i}, \Delta^{-t_i}]$:= FMIP_LNS($F_{t_i}, \hat{x}$)
    end for
    $U := \{j \in \mathcal{I} | x_j^{t_i} = x_j^{t_k},\ 0 \leq i < k < T\}$
    $[\hat{x}, \Delta^+, \Delta^-]$:= FMIP_LNS(U, x^{t_0})
  end if
  $\Delta^{UB} := \sum_i \Delta_i^+ + \Delta_i^-$
  for all threads $t_i \in \{0, T-1\}$ **in parallel do**
    F_{t_i} := randomized variable index subset, $F_{t_i} \subseteq \mathcal{I}$
    $[x^{t_i}, \Delta^{+t_i}, \Delta^{-t_i}]$:= OMIP_LNS($F_{t_i}, \hat{x}, \Delta^{UB}$)
  end for
  $U := \{j \in \mathcal{I} | x_j^{t_i} = x_j^{t_k},\ 0 \leq i < k < T\}$
  $[\hat{x}, \Delta^+, \Delta^-]$:= OMIP_LNS(U, x^{t_0}, Δ^{UB})
end while
return $[\hat{x}, \Delta^+, \Delta^-]$

sub-MIP—either FMIP or OMIP—until the allotted time limit is reached. Subsequently,

the solutions are exchanged, and the set U, containing the indices of variables common across solutions, is constructed. The recombination MIP then consists of a subproblem, again associated with either FMIP or OMIP, in which the variables in U are fixed. The best solution obtained will be either the most feasible or the most optimal, depending on whether a recombination FMIP or OMIP is employed. Moreover, every solution used as input can be incorporated as a MIP start, as it remains feasible under any variable-fixing strategy. Processor synchronization and memory communication are handled via the Message Passing Interface (MPI)[10], owing to its efficient all-to-all collective communication primitives in distributed large-scale architectures.

2.4 Initialization of ACS

As introduced earlier in this chapter, ACS only requires a starting vector that is integer feasible and within the variable bounds. However, a stronger starting point is a solution that is as feasible as possible with respect to the objective function of the FMIP. The proposed algorithm provides a lightweight heuristic that seeks to minimize the infeasibility of the initial solution. This algorithm can be described by the pseudocode[2]. The algorithm first sorts the list of integer variables in order of increasing bound range. It

Algorithm 2 Starting vector heuristic

Require: Percentage of variables to fix θ, $0 < \theta \leq 100$, Fixed bound constant c_b
Ensure: Starting integer-feasible vector $\hat{x}$
$V :=$ list of integer variables sorted by increasing bound range $u - l$
$F := \emptyset$
while $\hat{x}$ is not integer feasible **AND** $F \neq \mathcal{I}$ **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
    $F := F \cup \mathcal{K}$
    $[x, \Delta^+, \Delta^-] := \min\{\sum_i \Delta_i^+ + \Delta_i^- \mid Ax + I_m\Delta^+ - I_m\Delta^- = b,\ x_j = \hat{x}_j \ \ \forall j \in F\}$
    $Q :=$ index set of integer variables of x with integer value
    $\hat{x}_q = x_q, \ \ \forall q \in Q$
    $F := F \cup Q$
end while
return $\hat{x}$

then fixes the top $\theta\%$ of variables to random integer values within their respective bounds. The input parameter θ controls the trade-off between the difficulty of the LP relaxation and the quality of the resulting starting solution. In cases where the bounds are infinite, a constant value of 10^6 is used to clamp the bounds. The rationale behind the sorting step is to prioritize binary variables first, followed by the remaining integer variables. Until all integer variables are fixed, the LP relaxation of the FMIP is solved to optimize the unfixed variables. Any variables that attain integer values in this process are then fixed.

Since at least $\theta\%$ of the variables are fixed at each iteration, the algorithm is guaranteed to terminate after at most $\lceil 100/\theta \rceil$ iterations.

2.5 Variable Fixing Strategy

Selecting an appropriate variable fixing scheme is a challenging task: an overly restrictive strategy may fail to yield improvements, whereas an excessively loose strategy can lead to a search space that is too large to explore efficiently within a reasonable timespan. The proposed algorithm constitutes a simple yet intuitive variable fixing method. It incorporates randomness to promote diversification and allows for controlling the number of variables to be fixed through an adjustable parameter. The algorithm is described in the pseudocode[3]. The fixings are determined by selecting a random integer variable

Algorithm 3 Variable Fixing Selection Algorithm

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

x' and fixing a consecutive sequence of integer variables starting from x' up to a cap determined by ρ, an input parameter that specifies the number of variables to be fixed. The fixing is performed in a circular fashion: if the end of the set $\mathcal{I}$ is reached before the required number of variables are fixed, the algorithm continues from the beginning of $\mathcal{I}$. The effectiveness of this strategy relies on the fact that, for many problems—such as network flow and routing—the variables are arranged consecutively, often defining a cohesive substructure within the problem.

Chapter 3

The PACS Tuning

Although the baseline PACS algorithm performs well on many MIP instances, its focus on integer variables and reliance on fixed parameters can limit diversification and lead to premature convergence.

This section presents enhancements aimed at overcoming these limitations:

1. Generalizing starting vector construction and variable fixing to include both integer and continuous variables.
2. Implementing architecture-agnostic parallelization for consistent behavior across hardware.
3. Removing manual parameter tuning via adaptive sub-MIP time limits and dynamic adjustment of ρ.
4. Introducing a parameter-free initial solution strategy.
5. Adding optimization procedures, including slack upper bound enforcement and WalkMIP-based variable fixing, to improve feasibility and solution quality.

These improvements increase PACS's robustness, efficiency, and applicability while preserving its strengths in diversification and parallelism.

3.1 PACS with Generalized Fixing

The baseline PACS algorithm enforces both the starting vector construction and the fixing scheme to operate exclusively on the set $\mathcal{I}$ of integer variables. While this restriction may appear efficient and straightforward, it can, in fact, be limiting. In particular, if the MIP instance contains only a small fraction of integer variables relative to the total, focusing solely on them may reduce diversification and cause the algorithm to converge prematurely to a local minimum, which is undesirable in an optimization process. To overcome this issue, Algorithm 2 and Algorithm 3 are generalized into Algorithm 4 and

Algorithm 5, respectively, thereby allowing both integer and continuous variables to be considered.

Algorithm 4 Generalized Starting vector heuristic

Require: Percentage of variables to fix θ, $0 < \theta \leq 100$, Fixed bound constant c_b

Ensure: Starting integer-feasible vector $\hat{x}$

$V :=$ list of ~~integer~~ variables sorted by increasing bound range $u - l$
$F := \emptyset$
while $\hat{x}$ is not integer feasible **AND** $F \neq V$ **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
    $F := F \cup \mathcal{K}$
    $[x, \Delta^+, \Delta^-] := \min\{\sum_i \Delta_i^+ + \Delta_i^- \mid Ax + I_m\Delta^+ - I_m\Delta^- = b,\ x_j = \hat{x}_j \ \ \forall j \in F\}$
    $Q :=$ index set of ~~integer~~ variables of x with integer value
    $\hat{x}_q = x_q, \ \ \forall q \in Q$
    $F := F \cup Q$
end while
return $\hat{x}$

Algorithm 5 Generalized Variable Fixing Selection Algorithm

Require: Fraction of variables to fix ρ, $0 < \rho < 1$

Ensure: Set of integer indices F

function RandomFixings(ρ)
    $i :=$ random element in $\{1 \dots n\}$ ▷ n: number of variables in the original MIP
    $F :=$ first $\rho \cdot n$ consecutive ~~integer~~ variable indices starting from i in a circular fashion
    return F
end function

Since these algorithms are executed on the auxiliary MIP problems -FMIP or OMIP-, the variables subject to fixing correspond exactly to those defined in the original MIP formulation. By generalizing the fixing strategy to include both integer and continuous variables, diversification is enhanced, thereby reducing the likelihood of stagnation in local minima and potentially improving the exploration of the solution space.

3.2 Architecture-Agnostic Parallelization

In the original study, the Message Passing Interface (MPI) was employed to synchronize processors at each recombination phase. While this approach is well suited to

high-performance computing environments, it may be unnecessarily complex in general-purpose scenarios, where a simpler multi-threading implementation is often preferable. In this thesis, communication is instead managed through a set of logical threads, which may differ from the number of available hardware threads. This abstraction ensures that, even on machines with fewer physical cores, the algorithm can reproduce the same behavior across different architectures, provided sufficient computational time is allowed. More specifically, during the coordination phase, either the most feasible or the most optimal solution—depending on whether a recombination FMIP or OMIP is performed—is shared among the logical processors. Each processor then continues working independently on its own copy, with updates to the incumbent solution handled exclusively through a thread-safe update function, formally described in Algorithm 6.

Algorithm 6 Parallel ACS Incumbent Update Procedure

Require: Candidate solution x with slack sum $S(x)$ and objective value $C(x)$; Incumbent $\tilde{x}$; Zero-tolerance ϵ

Ensure: Updated incumbent $\tilde{x}$ in a thread-safe manner

1: **function** UPDATEINCUMBENT(x)
2: acquire lock
3: **if** $(|S(x)| < |S(\tilde{x})|) \ \vee \ (|S(x)| < \epsilon \ \wedge \ C(x) < C(\tilde{x}))$ **then**
4: $\tilde{x} \leftarrow x$
5: **end if**
6: release lock
7: **end function**

This mechanism is crucial to ensure that the algorithm consistently improves and converges within the given time limit. The incumbent solution is updated whenever a better solution is identified: either one with a smaller slack sum, indicating improved feasibility, or one with a lower objective value $c^T x$ provided that the slack sum is less than the tolerance parameter ϵ, the zero-feasibility threshold.

3.3 Eliminating Calibration in PACS Parameter Selection

After adapting the PACS algorithm to a more general-purpose environment, another important challenge arises: parameter selection. Since PACS must be applicable to a wide range of hard MIP instances, it is necessary to identify parameter settings that generally perform well, both in terms of computational efficiency and heuristic solution quality. The goal is to remove the need for an explicit calibration phase, while still ensuring reliable performance. The tuning process specifically concerns the following parameters:

1. The time span assigned to each sub-MIP, either FMIP_LNS or OMIP_LNS

2. The parameter ρ governing the fixing strategy

3. The parameter θ governing the starting vector

3.3.1 Adaptive Determination of Sub-MIP Time Span

To guarantee determinism in the implementation, each sub-MIP is assigned both a time limit equal to the remaining computation time and a deterministic time limit. The latter is defined as the maximum number of instructions that the solver may execute before termination. The deterministic time limit is computed according to the following formula:

$$TL_{DET} = \max\left(x, \min\left(\frac{nz}{y}, X\right)\right) \tag{3.1}$$

which provides a dynamic mechanism for adapting the computational effort of each sub-MIP. In this formulation, x and X denote the minimum and maximum allowable deterministic time limits, respectively, nz represents the number of nonzeros in the constraint matrix A of the MIP problem, and y acts as a scaling factor. The chosen parameter values are:

1. Minimum deterministic time limit: $x = 10^3$

2. Scaling factor: $y = 10^2$

3. Maximum deterministic time limit: $X = 10^7$

This dynamic adjustment removes the need to explicitly set a fixed time limit for each subproblem, thereby eliminating the necessity of parameter tuning in this respect.

3.3.2 Adaptive Variable Fixing through ρ Adjustment

Fixed ρ Initialization

As discussed in Section 2.5, selecting an appropriate variable fixing scheme is a non-trivial task. In particular, the random fixings scheme presents additional challenges: if the fraction ρ of variables to be fixed is set too high, the procedure may fail to yield meaningful improvements; conversely, if ρ is set too low, the resulting search space may become excessively large, making it computationally intractable within a reasonable time frame.
For this reason, in the first instance, the parameter ρ is selected from a set of predetermined candidate values. Anticipating the results presented in Section **??**, it can be observed that certain values of ρ are more effective in terms of solution quality, as they grant the solver greater flexibility during the optimization process.

Dynamic Adjustment of Parameter ρ

Since the LNS heuristics in the PACS algorithm restrict the search space by fixing a number of variables according to the value of ρ, it is natural to design a mechanism for dynamically adapting this parameter in order to increase the likelihood of discovering high-quality solutions.

Algorithm 7 Parallel ACS Rho Update (Parallel Phases)

Require: Status code MIP_{code} returned by the solver; Adjustment step Δ_ρ for ρ; The variable fixing parameter ρ; Number of parallel sub-MIPs num_{MIP}

Ensure: Updated value of ρ, synchronized across parallel sub-MIPs

function DynRhoUpdate(MIP_{code}, Δ_ρ, ρ, num_{MIP})
  $\hat{\Delta}_\rho \leftarrow \frac{\Delta_\rho}{num_{MIP}}$
  acquire lock
  if $MIP_{code} = OPT \ \vee \ MIP_{code} = OPT_{TOL}$ **then**
    $\rho \leftarrow \rho - \hat{\Delta}_\rho$
  end if
  if $MIP_{code} = FEAS_{TL} \ \vee \ MIP_{code} = FEAS_{DET\_TL}$ **then**
    $\rho \leftarrow \rho + \hat{\Delta}_\rho$
  end if
  Clap ρ within $[0.01, 0.99]$
  if *Tie Case detected* **then**
    $\rho \leftarrow \rho - \Delta_\rho$
  end if
  release lock
end function

The procedure, described in Algorithm 7, is applied after each sub-MIP optimization phase in the parallel step. Based on the status returned by the solver, the value of ρ is updated as follows:

- If the solver hits the time limit—either the deterministic limit or the global remaining time—while still producing a feasible solution, ρ is increased by $\frac{\Delta_\rho}{num_{MIP}}$. This adjustment suggests fixing more variables in subsequent phases, thereby simplifying the subproblem to be solved.

- Conversely, if the solver converges to an optimal solution within the tolerance, this indicates that the corresponding region of the search space has already been sufficiently explored. In this case, ρ is decreased by $\frac{\Delta_\rho}{num_{MIP}}$, enlarging the search space and granting the solver greater freedom in the following optimization steps.

Since each sub-MIP independently attempts to modify the value of ρ, synchronization through locking is required to prevent inconsistencies. The final update is determined as

the average of the adjustments proposed by the parallel optimization phases. In case of a tie, a deterministic rule is applied: ρ is decreased by Δ_ρ.

Algorithm 8 Parallel ACS Rho Update (Recombination Phases)

Require: Status code MIP_{code} returned by the solver; Adjustment step Δ_ρ for ρ; The variable fixing parameter ρ; Number of parallel sub-MIPs num_{MIP}

Ensure: Updated value of ρ after recombination adjustment

function DynRhoUpdate(MIP_{code}, Δ_ρ, ρ, num_{MIP})
  $\hat{\Delta}_\rho \leftarrow \frac{2\Delta_\rho}{num_{MIP}}$
  if $MIP_{code} = OPT \ \vee \ MIP_{code} = OPT_{TOL}$ **then**
    $\rho \leftarrow \rho - \hat{\Delta}_\rho$
  end if
  if $MIP_{code} = FEAS_{TL} \ \vee \ MIP_{code} = FEAS_{DET\_TL}$ **then**
    $\rho \leftarrow \rho + \hat{\Delta}_\rho$
  end if
  Clap ρ within $[0.01, 0.99]$
end function

For the recombination phase, the procedure is slightly modified into Algorithm 8, where the adjustment step is doubled, i.e. $2\Delta_\rho/num_{MIP}$, in order to resolve potential tie cases.

Finally, although an initial value of ρ must be provided to start the fixing process, experimental results in Section **??** show that performance is only marginally affected by this initialization. Consequently, the effectiveness of the method does not critically depend on the initial choice of ρ.

3.3.3 Parameter-Free Strategy for Initial Solution Construction

Another non-trivial challenge concerns the selection of the initial solution vector. The parameter θ, which regulates the trade-off between solution quality and execution time, must be tuned to suit the most general cases. Similar to the fixing strategy, the analysis considers a set of predetermined values for θ, while also introducing a novel initialization strategy for the starting solution vector. The motivation is that using a small value of θ—which corresponds to generating a high-quality initial solution—may waste computation time, since the true strength of PACS lies in its diversification and parallelization capabilities.

Therefore, Algorithm 9 is designed with the following goal: to provide a heuristic starting solution that is deterministic and computationally lightweight, while remaining reasonably feasible for the original MIP problem.

Algorithm 9 Heuristic Initialization of Starting Solution

Require: MIP instance with n variables, objective coefficients c_i, bounds $[l_i, u_i]$; zero-tolerance ϵ

Ensure: Starting solution vector $x \in \bar{\mathbb{R}}^n$

function StartSolMaxFeas(MIP)
  $x \leftarrow (+\infty, \ldots, +\infty) \in \bar{\mathbb{R}}^n$
  for $i = 1 \rightarrow n$ **do**
    $(l_i, u_i) \leftarrow$ variable bounds of variable i
    if $l_i = -\infty \ \wedge \ u_i = +\infty$ **then**
      $x_i \leftarrow 0$
    else if $l_i > -\infty \ \wedge \ u_i = +\infty$ **then**
      $x_i \leftarrow l_i$
    else if $l_i = -\infty \ \wedge \ u_i < +\infty$ **then**
      $x_i \leftarrow u_i$
    else
      if $c_i \leq -\epsilon$ **then**
        $x_i \leftarrow l_i$
      else if $c_i \geq \epsilon$ **then**
        $x_i \leftarrow u_i$
      else
        $x_i \leftarrow$ RandomInteger(l_i, u_i)
      end if
    end if
  end for
  return x
end function

The initialization procedure in Algorithm 9 operates as follows:

- If a variable has no finite bounds, it is set to 0.
- If a variable has only one finite bound, it is set to that bound.
- If both bounds are finite:
  - set the variable to the lower bound if $c_i \leq -\epsilon$,i.e. the coefficient is negative,
  - set it to the upper bound if $c_i \geq \epsilon$, i.e.the coefficient is positive,
  - choose a random integer within $[l_i, u_i]$ if $-\epsilon \leq c_i \leq \epsilon$, i.e.the coefficient is close to zero.

In this way, the solution initialization is almost entirely deterministic and computationally lightweight, although feasibility with respect to the original MIP problem may be partially

compromised. Furthermore, the proposed strategy is parameter-free, which makes it particularly well-suited for the purposes of this thesis.

3.4 Additional PACS Optimizations

While the previous modifications focused on tuning or slightly adjusting specific components of the PACS workflow, the following procedures—analyzed in Section **??**—aim to further enhance the quality of the final solution.

3.4.1 Slack Upper Bound Enforcement and Budget Constraint Removal

The first idea is to exploit the values of the slack variables Δ^+ and Δ^- obtained in a previous iteration to enforce these values as upper bounds. In the PACS algorithm, this mechanism is already embedded in the budget constraint $\sum_{i=0}^{m} \Delta_i^+ + \Delta_i^- \leq \sum_{i=0}^{m} \hat{\Delta}_i^+ + \hat{\Delta}_i^-$, which provides only a global upper bound. The proposed strategy, instead, aims at fixing the upper bounds of individual slack variables, thereby assisting subsequent FMIP optimizations and improving feasibility. The procedure is implemented as shown in Algorithm 10.

Algorithm 10 Fixing Slack Variables to Upper Bound

Require: MIP model with n variables; Solution x of length $n + 2m$; Zero-tolerance ϵ
Ensure: Updated model with adjusted upper bounds for slack variables

```
1: function SlackUpperBoundFixing(MIP, x)
2:     S_UB ← ∅                          ▷ Set of variables-upper bound to be updated
3:     for i = n → n + 2m do
4:         u_i ← upper bound of variable i
5:         if u_i − x_i > ε then
6:             S_UB ← S_UB ∪ (i, x_i)
7:         end if
8:     end for
9:     return S_UB
10: end function
```

Here, the solution vector x corresponds to the current PACS incumbent—valid in both the parallel and recombination phases—and each slack variable is compared with its respective upper bound. If the variable value is smaller than the existing upper bound, the variable index together with its solution value is added to S_{UB} as a pair (i, x_i). The resulting set S_{UB} is subsequently employed to update the upper bounds of the slack variables, thereby modifying the auxiliary FMIP and OMIP formulations as follows.

FMIP Reformulation

$$\begin{cases} \min & \sum_{i=0}^{m} \Delta_i^+ + \Delta_i^- \\ \text{s.t.} & Ax + I_m\Delta^+ - I_m\Delta^- = b \\ & x_i = \hat{x}_i,\ \forall i \in F \\ & \Delta_i^+ \leq \hat{u}_i,\ \forall (i, \hat{u}_i) \in S_{UB}^+ \\ & \Delta_i^- \leq \hat{u}_i,\ \forall (i, \hat{u}_i) \in S_{UB}^- \\ & l \leq x \leq u \\ & x_i \in \mathbb{Z},\ \forall i \in \mathcal{I} \\ & \Delta^+ \geq 0,\ \Delta^- \geq 0 \end{cases} \tag{3.2}$$

OMIP Reformulation

$$\begin{cases} \min & c^T x \\ \text{s.t.} & Ax + I_m\Delta^+ - I_m\Delta^- = b \\ & \cancel{\sum_{i=0}^{m} \Delta_i^+ + \Delta_i^- \leq \sum_{i=0}^{m} \hat{\Delta}_i^+ + \hat{\Delta}_i^-} \\ & x_i = \hat{x}_i,\ \forall i \in F \\ & \Delta_i^+ \leq \hat{u}_i,\ \forall (i, \hat{u}_i) \in S_{UB}^+ \\ & \Delta_i^- \leq \hat{u}_i,\ \forall (i, \hat{u}_i) \in S_{UB}^- \\ & l \leq x \leq u \\ & x_i \in \mathbb{Z},\ \forall i \in \mathcal{I} \\ & \Delta^+ \geq 0,\ \Delta^- \geq 0 \end{cases} \tag{3.3}$$

In these formulations, the set S_{UB} is partitioned into S_{UB}^+ and S_{UB}^- to remain consistent with the notation used for FMIP and OMIP. It is worth noting that the budget constraint in OMIP can now be removed, since the slack upper bound fixing provides a stronger restriction, as will be further confirmed by the results in Section **??**.

3.4.2 WalkMIP: Violated-Constraints-Based Variable Fixing

It is worthwhile to investigate a more sophisticated and potentially efficient variable fixing strategy, aiming to enrich diversification and consequently increase the likelihood of finding high-quality solutions within a relatively short time span. The WalkMIP strategy adapts ideas from the WalkSAT[11] algorithm, targeting specifically the variables that cause infeasibility in a partially feasible solution.

Algorithm 11 presents the WalkMIP procedure adapted for MIP constraint satisfaction problems.

Algorithm 11 Walk-based Repair Heuristic for MIP

Require: MIP model with $n + 2m$ variables (FMIP or OMIP); Initial solution x; Fixing parameter ρ; Random walk probability $p \in (0, 1)$; Solution kick probability p_{HUGE_KICK}

Ensure: Updated model with improved feasible solution

1: **function** WalkMIP(MIP, x, ρ, p,p_{HUGE_KICK})
2: $violConstr \leftarrow$ Set of indices of violated constraints, under x
3: **if** *First Execution* **then** $numInitialConstr \leftarrow |violConstr|$
4: **end if**
5: **if** $violConstr = \emptyset$ **then return**
6: **end if**
7: **if** with probability p_{HUGE_KICK} **then**
8: RandomFixings(ρ)
9: **end if**
10: $k \leftarrow$NumConstrToFix($numInitialConstr$,$|violConstr|$)
11: **repeat**
12: $c \leftarrow$ Random violated constraint, $c \in violConstr$
13: $(i_{best}, val_{best}, minDMG) \leftarrow$ MinDamageMove(c, x)
14: **if** $minDMG \leq -\epsilon$ **then**
15: $\hat{x} \leftarrow$ UpdateSol(x, i_{best}, val_{best}) ▷ Apply best move to x
16: **else if** with probability $1 - p \wedge i_{best} \neq -1$ **then**
17: $\hat{x} \leftarrow$ UpdateSol(x, i_{best}, val_{best}) ▷ Apply min-damage move to x
18: **else**
19: $i_{rnd} \leftarrow$ RandomInteger($0, n - 1$)
20: $\hat{x} \leftarrow$ UpdateSol($x, i_{rnd}, x_{i_{rnd}}$) ▷ Randomly fixing a var in x
21: **end if**
22: $violConstr \leftarrow$ Set of indices of violated constraints, under $\hat{x}$
23: **if** $violConstr = \emptyset$ **then break**
24: **end if**
25: **until** k times
26: **if** $k < \rho \cdot n$ **then**
27: $\hat{x} \leftarrow$Fix remaining variables (up to $\rho \cdot n$) from original x
28: **end ifreturn** $\hat{x}$
29: **end function**

The WalkMIP algorithm is executed independently by each processor, replacing the standard random fixing procedure. The strategy can be decomposed into three main phases:

1. **Initialization and Huge-Kick Phase (lines 1-9 in Algorithm 11)**: If this is the first execution of the routine, the algorithm records the number of violated

constraints under the current solution x, which is later used to determine the number of iterations for the main loop. To enhance diversification and escape local minima, a probability p_{HUGE_KICK} is introduced to randomly fix variables according to ρ. In our implementation, this parameter is set to 1/16.

2. **Constraint Selection and Optimization Loop (lines 10-25 in Algorithm 11)**: The number of constraints to target is determined by the function:

$$k(v, v_0) = \begin{cases} \lceil k_{\min} + (k_{\max} - k_{\min}) \cdot \hat{r}^{\,\beta} \rceil & \text{if } v > k_{\min} \\ v & \text{otherwise} \end{cases}$$

where

$$\hat{r} = \min\left(1, \max\left(0, \frac{v}{\max(1, v_0)}\right)\right), \quad k_{\min} = 32,\ k_{\max} = 1024,\ \beta = \frac{\sqrt{5}-1}{2}.$$

This ensures that the number of touched constraints is bounded by $k_{\max}$ while accounting for constraints that have become feasible since the algorithm started. Only one variable per constraint is modified, imitating the WalkSAT strategy. The main loop iterates over k randomly chosen violated constraints—updated at each iteration—, selecting the "minimum damage move" for each constraint. The function MinDamageMove, the Algorithm 1, evaluates potential moves:

- If a variable fixing reduces infeasibility, the best move is applied.
- Otherwise, with probability $1 - p$, the minimal-damage move is applied.
- Otherwise, with probability p, a random variable is fixed using its current solution value $x_{i_{rnd}}$.

3. **Fixing Remaining Variables (lines 26-29 in Algorithm 1)**: To maintain consistency with the original variable fixing strategy, the total number of fixed variables is set to $\rho \cdot n$. If the previous phases fixed fewer variables, the remaining variables are fixed according to the standard strategy.

Algorithm 12 Minimum Damage Move Selection for a Violated Constraint in WalkMIP

Require: Violated constraint c, current solution vector x

Ensure: Tuple $(i_{best}, val_{best}, minDMG)$, where i_{best} is the index of the variable to modify, val_{best} is the new value for that variable, and $minDMG$ is the resulting minimum change in constraint violation

```
function MinDamageMove(c, x)
    minDMG ← +∞, i_best ← −1, val_best ← +∞
    for each variable i in c do
        if i.type = BIN then x̂_i ← ¬x_i
        else if i.type = INT then x̂_i ← x_i + RandomInteger(−1, 1)
        else if i.type = DOUBLE then x̂_i ← x_i + RandomDouble(−0.5, 0.5)
        end if
        Clamp x̂_i within its bounds [l_i, u_i]
        δ ← ComputeViolation(x, x̂_i)
        if δ < minDMG then
            i_best ← i, minDMG ← δ, val_best ← x̂_i
        end if
    end for
    return (i_best, val_best, minDMG)
end function
```

Notice that any modifications of ρ via the dynamic adjustment routine are incorporated in subsequent WalkMIP iterations. It is important to emphasize that WalkMIP can only be applied when the slack variables are nonzero. Consequently, the original random fixing strategy, described in Section 3.3.2, remains necessary during OMIP optimization phases where the slack sum is below the zero-tolerance threshold ϵ. Although the results in Section **??** indicate that this strategy is less effective than the simpler, more straightforward variable fixing approach, it offers valuable insights and suggests a potential direction for future research.

Chapter 4

Experimental Results

4.1 Experimental Setup

The experiments are designed to compare the PACS framework with the standalone IBM ILOG CPLEX Optimization Studio (version 22.1). The PACS framework is implemented in C++ for performance reasons, interfacing with the solver through the C API. The source code is available under a non-commercial MIP license at [**repository link**]. All experiments were conducted on a cluster within the UniPD DEI Blade infrastructure, running Rocky Linux 8.10. Each node is equipped with an Intel(R) Xeon(R) E5-2623 v3 processor (4 cores, 3.00 GHz) and 16 GB of RAM. Although the infrastructure is cluster-based, the computational power of a single node is comparable to that of a general-purpose laptop, or even lower. For fairness in comparison, PACS executions use 4 logical threads, implemented as C++ standard threads, each running a CPLEX instance restricted to a single core. The baseline counterpart is a standalone CPLEX execution restricted to 4 cores.

4.1.1 Dataset

As anticipated in the previous sections, the benchmark for this comparison must consist of hard MIP instances. To this end, the set of hard instances from MIPLIB2017[12] has been used as the test bed.[1] While CPLEX directly processes each MIP instance, the PACS framework additionally requires a random seed in order to reproduce the same randomized choices across different runs. For this purpose, PACS was evaluated on each instance using the seeds $\{38472910, 56473829, 27384910, 91827364, 8374659\}$. This setup increases statistical reliability and can be considered representative of the algorithm's standard behavior. Consequently, for each instance, five independent PACS runs and one CPLEX run were executed.

[1] The instance *tpl-tub-ss16* was excluded, as its execution was terminated due to insufficient computational resources.

4.1.2 Metrics

Both PACS and plain CPLEX terminate as soon as an incumbent solution (i.e., a feasible solution) is identified, subject to a global time limit of five minutes. Since the PACS framework was executed five times for each instance, the reported solution quality and computation time are given as the mean across all runs. Moreover, if the majority of PACS executions for a given instance—at least three out of five seeds—fail to produce a solution, the instance is recorded as unsolved. Instead of directly comparing the objective values of the solutions returned by the two approaches, the MIP gap metric[13] has been adopted. The MIP gap is defined as follows: given a solution $\hat{x}$ for a MIP with an optimal solution x, the primal gap $\gamma(\hat{x}) \in [0, 1]$ is

$$\gamma(\hat{x}) = \begin{cases} 0 & \text{if } |c^T x| = |c^T \hat{x}| = 0, \\ 1 & \text{if } c^T x \cdot c^T \hat{x} < 0, \\ \frac{|c^T x - c^T \hat{x}|}{\max\{|c^T \hat{x}|, |c^T x|\}} & \text{otherwise.} \end{cases} \tag{4.1}$$

This metric captures the relative gap between an incumbent and the best-known solution, normalized to the interval $[0, 1]$. Since the algorithms operate under a strict time limit, the metric must be extended accordingly. Given a time limit $t_{\max}$, the primal function $p : [0, t_{\max}] \to [0, 1]$ is defined as:

$$p(\hat{x}) = \begin{cases} 1 & \text{if no solution is found up to time } t, \\ \gamma(\hat{x}(t)) & \text{if } \hat{x}(t) \text{ is the incumbent at time } t. \end{cases} \tag{4.2}$$

For each test, two types of plots are produced to summarize and compare performance:

1. **Success Rate vs. Computation Time:** The x-axis represents elapsed time, and the y-axis reports the cumulative number of instances for which an incumbent was found. The closer a curve is to the top-left corner, the more efficient the algorithm.

2. **Success Rate vs. MIP Gap:** The x-axis represents the MIP gap, and the y-axis reports the cumulative number of instances solved with a gap less than or equal to the given value. Consistently, curves closer to the top-left corner indicate better performance

4.1.3 Tolerance Parameters

Because both algorithms involve floating-point computations and comparisons, PACS employs a set of tolerance parameters. In these experiments, the parameters were selected to be reasonable relative to the defaults used by CPLEX:

- **Zero-tolerance parameter** ϵ: values smaller than ϵ are treated as zero. In all tests, ϵ was set to 10^{-5}.

- **Absolute maximum constraint violation**: the maximum permissible violation of any constraint under a candidate solution $\hat{x}$. In these tests, it was set equal to ϵ, i.e., 10^{-5}.

- **Absolute maximum integrality violation**: the maximum deviation allowed for variables constrained to take integer values. This tolerance was likewise set to $\epsilon = 10^{-5}$.

- **Relative objective error**: the maximum admissible relative error between the recomputed objective value $c^T\hat{x}$ and the value stored internally by PACS. This threshold was again set to $\epsilon = 10^{-5}$.

In the latter three cases, if the corresponding tolerance is exceeded, the algorithm terminates with an error and the run is recorded as unsolved.

4.2 Results

Bibliography

[1] IBM Corporation. *IBM ILOG CPLEX Optimization Studio*. Version 22.1. 2024. URL: https://www.ibm.com/products/ilog-cplex-optimization-studio.

[2] Gurobi Optimization, LLC. *Gurobi Optimizer Reference Manual*. Version 11.0. 2024. URL: https://www.gurobi.com.

[3] David Pisinger and Stefan Ropke. "Large Neighborhood Search". In: *Handbook of Metaheuristics*. Ed. by Michel Gendreau and Jean-Yves Potvin. Boston, MA: Springer US, 2010, pp. 399–419. ISBN: 978-1-4419-1665-5. DOI: 10.1007/978-1-4419-1665-5_13. URL: https://doi.org/10.1007/978-1-4419-1665-5_13.

[4] Tobias Achterberg, Timo Berthold, and Gregor Hendel. "Rounding and Propagation Heuristics for Mixed Integer Programming". In: Jan. 2012.

[5] Lluís-Miquel Munguía et al. "Alternating criteria search: a parallel large neighborhood search algorithm for mixed integer programs". In: *Computational Optimization and Applications* 69.1 (Jan. 2018), pp. 1–24. ISSN: 1573-2894. DOI: 10.1007/s10589-017-9934-5. URL: https://doi.org/10.1007/s10589-017-9934-5.

[6] Thorsten Koch, Ted Ralphs, and Yuji Shinano. "Could we use a million cores to solve an integer program?" In: *Mathematical Methods of Operations Research* 76.1 (Aug. 2012), pp. 67–93. ISSN: 1432-5217. DOI: 10.1007/s00186-012-0390-9. URL: https://doi.org/10.1007/s00186-012-0390-9.

[7] Emilie Danna, Edward Rothberg, and Claude Le Pape. "Exploring relaxation induced neighborhoods to improve MIP solutions". In: *Mathematical Programming* 102.1 (Jan. 2005), pp. 71–90. ISSN: 1436-4646. DOI: 10.1007/s10107-004-0518-7. URL: https://doi.org/10.1007/s10107-004-0518-7.

[8] Shubhashis Ghosh. "DINS, a MIP Improvement Heuristic". In: June 2007, pp. 310–323. ISBN: 978-3-540-72791-0. DOI: 10.1007/978-3-540-72792-7_24.

[9] Matteo Fischetti and Andrea Lodi. "Local branching". In: *Mathematical Programming* 98.1 (Sept. 2003), pp. 23–47. ISSN: 1436-4646. DOI: 10.1007/s10107-003-0395-5. URL: https://doi.org/10.1007/s10107-003-0395-5.

[10] William Gropp et al. "A high-performance, portable implementation of the MPI message passing interface standard". English (US). In: *Parallel Computing* 22.6 (Sept. 1996), pp. 789–828. ISSN: 0167-8191. DOI: 10.1016/0167-8191(96)00024-5.

[11] Bart Selman, Henry A. Kautz, and Brain Cohen. "Noise strategies for improving local search". In: *Proceedings of the Twelfth AAAI National Conference on Artificial Intelligence*. AAAI'94. Seattle, Washington: AAAI Press, 1994, pp. 337–343.

[12] Ambros Gleixner et al. "MIPLIB 2017: Data-Driven Compilation of the 6th Mixed-Integer Programming Library". In: *Mathematical Programming Computation* (2021). DOI: 10.1007/s12532-020-00194-3. URL: https://doi.org/10.1007/s12532-020-00194-3.

[13] Timo Berthold. "Measuring the impact of primal heuristics". In: *Operations Research Letters* 41.6 (2013), pp. 611–614. ISSN: 0167-6377. DOI: https://doi.org/10.1016/j.orl.2013.08.007. URL: https://www.sciencedirect.com/science/article/pii/S0167637713001181.