Electronic Research Archive

https://www.aimspress.com/journal/era

ERA, 33(6): 3950–3967.
DOI: 10.3934/era.2025175
Received: 22 March 2025
Revised: 30 May 2025
Accepted: 09 June 2025
Published: 24 June 2025

Research article

A divide-and-conquer strategy for integer programming problems

Chao Yang[1], Xifeng Ning[2,3], Dejun Yu[2,3], Hailu Sun[2,3], Guohui Kang[2,3] and Hongyan Wang[4,*]

[1] College of Petroleum Engineering, China University of Petroleum, Beijing 100100, China

[2] Petro China Planning & Engineering Institute, Beijing 100083, China

[3] Key Laboratory of Oil Gas Business Chain Optimization, CNPC, Beijing 100083, China

[4] School of Vehicle and Mobility, Tsinghua University, Beijing 100084, China

* **Correspondence:** Email: hongyan-wang@mail.tsinghua.edu.cn.

Abstract: Integer programming (IP) is a fundamental combinatorial optimization problem, traditionally solved using branch-and-bound methods. However, these methods often struggle with large-scale instances as they attempt to solve the entire problem at once. Existing heuristics, such as a large neighborhood search (LNS), improve efficiency but lack a structured decomposition strategy. We have proposed a divide-and-conquer strategy for IP, which partitions decision variables based on constraint relationships, optimizes them in smaller subproblems, and progressively merges solutions while repairing constraints. This structured approach significantly enhances both solution quality and computational efficiency. Experiments on four standard benchmarks—minimum vertex cover (MVC), maximum independent set (MIS), set cover (SC), and combinatorial auction (CA)—demonstrated that our method significantly outperforms both state-of-the-art solvers (e.g., Gurobi, SCIP) and advanced LNS-based heuristics. When using Gurobi and SCIP as sub-solvers, our approach achieved up to 30× improvement in objective value on certain tasks (e.g., MIS), and reduced solution cost by over 44% compared to exact solvers in minimization settings (e.g., MVC). In addition, our method showed consistently faster convergence and better final solution quality across all tested problems. These results highlight the robustness and scalability of our divide-and-conquer strategy, establishing it as a powerful framework for solving large-scale integer programming problems beyond the capabilities of traditional and heuristic-based solvers.

Keywords: integer programming; combinatorial optimization; divide-and-conquer; optimization; large-scale

1. Introduction

Integer programming (IP) is a fundamental combinatorial optimization problem with broad applications in logistics [1, 2], scheduling [3, 4], network design [5–7], and resource distribution [8, 9]. Given its NP-hard nature [10], solving large-scale IP instances efficiently remains a significant challenge. Traditional solvers rely on branch-and-bound (B&B) methods [11, 12], which systematically explore the solution space by branching on decision variables and pruning suboptimal regions using bounds. While B&B guarantees optimality, its worst-case complexity grows exponentially [13], making it computationally intractable for large-scale problems.

To mitigate this, heuristic and metaheuristic approaches have been developed to accelerate solution finding [14–17]. Among them, the large neighborhood search (LNS) has proven to be an effective strategy for improving computational efficiency [18, 19]. The LNS iteratively refines solutions by fixing a subset of variables and re-optimizing a selected neighborhood. However, the LNS and its variants heavily rely on domain-specific heuristics to define neighborhoods, often lacking a principled mechanism for structured decomposition. As a result, these methods may struggle to generalize across problem instances and may not fully exploit the structural dependencies inherent in IP formulations.

We address the limitations of solving large-scale integer programming (IP) problems by proposing a divide-and-conquer strategy. Instead of optimizing the entire problem at once, our method follows a structured process: we first obtain an initial feasible solution, then systematically partition the decision variables based on constraint relationships, forming smaller, more manageable subproblems. These subproblems are optimized independently along different variable dimensions, yielding directional optimal solutions. The solutions are then merged in a conquer phase, ensuring feasibility while progressively improving solution quality. Finally, we apply a fixed-radius search-based refinement step, which further enhances feasibility and optimality by searching for improved integer solutions within a local neighborhood. This structured approach balances computational efficiency with solution quality, enabling iterative refinement of the global solution.

To evaluate the effectiveness of our approach, we conduct extensive experiments on four standard benchmarks: the maximum independent set (MIS) [20], minimum vertex cover (MVC) [21], set cover (SC) [22], and combinatorial auction (CA) [23]. Our results demonstrate that our method not only outperforms state-of-the-art solvers such as Gurobi [24] and SCIP [25] but also surpasses advanced large neighborhood search (LNS) variants [18, 26–28]. Specifically, our strategy achieves lower optimality gaps in shorter computational times, highlighting its potential for solving large-scale IP instances efficiently. Our code is open-source at https://github.com/happywhy/Divide-and-Conquer-Strategy.

Our contributions can be summarized as follows:

- We introduce the first divide-and-conquer strategy tailored for integer programming, systematically decomposing, optimizing, merging, and refining solutions to enhance computational efficiency.
- We propose a novel partitioning-merging-refinement strategy for IPs, ensuring that variable dependencies are preserved while progressively improving solution quality.
- We conduct extensive experiments on four benchmark problems, demonstrating that our method outperforms state-of-the-art solvers and LNS variants in both solution quality and efficiency.

The remainder of this paper is organized as follows. Section 2 reviews related work, including existing solvers and heuristic methods for IP. Section 3 describes our divide-and-conquer strategy in

detail. Section 4 presents experimental results and comparisons with baseline methods. Finally, Section 5 concludes the paper and discusses future research directions.

2. Related work

2.1. Integer programming

Integer programming (IP) is a fundamental optimization framework widely used in combinatorial optimization, operations research, logistics, and artificial intelligence [29]. It involves optimizing a given objective function while satisfying a set of constraints, with the additional requirement that some or all decision variables must take integer values. A standard formulation of an IP problem is given by:

$$\min_{\mathbf{x}} \quad c^\top \mathbf{x} \tag{2.1}$$

$$\text{s.t.} \quad A\mathbf{x} \leq \mathbf{b}, \tag{2.2}$$

$$x_j \in \mathbb{Z}, \quad \forall j \in \mathcal{I}, \tag{2.3}$$

$$l_j \leq x_j \leq u_j, \quad \forall j, \tag{2.4}$$

where $\mathbf{x} \in \mathbb{R}^n$ represents the decision variables, and the objective function is given by $f(\mathbf{x}) = c^\top \mathbf{x}$, where $c \in \mathbb{R}^n$ is a coefficient vector. The constraints are defined by a linear system $A\mathbf{x} \leq \mathbf{b}$, where $A \in \mathbb{R}^{m\times n}$ is the constraint matrix, and $\mathbf{b} \in \mathbb{R}^m$ is the right-hand-side vector. The subset $\mathcal{I} \subseteq \{1, \ldots, n\}$ specifies the indices of variables that are required to be integers. The bounds l_j and u_j define lower and upper limits on the values of x_j.

2.2. Methods for solving integer programs

Solving integer programming (IP) problems is computationally challenging, as they belong to the class of NP-hard problems [10]. Existing approaches can be broadly categorized into exact algorithms and heuristic algorithms.

2.2.1. Exact algorithms

Exact algorithms aim to find the globally optimal solution while guaranteeing optimality. The most widely used methods for solving mixed-integer programming (MIP) problems include branch-and-bound (B&B) and branch-and-cut (B&C).

Branch-and-Bound (B&B) Branch-and-bound [11, 12] is a tree-based search algorithm that systematically explores the solution space by recursively branching on integer variables and using bounding techniques to prune suboptimal branches. The process consists of:

- Relaxation: Solve the linear programming (LP) relaxation of the problem (i.e., ignore integer constraints).
- Branching: If the LP solution violates integer constraints, split the problem into subproblems by fixing an integer variable to different values.
- Bounding: Compute bounds on the objective function to eliminate regions that cannot contain the optimal solution.

- Pruning: Discard subproblems if their bounds indicate they cannot improve upon the best known solution.

Branch-and-Cut (B&C) Branch-and-cut [30] extends B&B by incorporating cutting planes, which iteratively refine the LP relaxation by adding valid inequalities (cuts) to remove fractional solutions while preserving feasibility. Common types of cutting planes include Gomory cuts, cover inequalities, and Chvátal-Gomory closures.

MIP solvers: a unified framework State-of-the-art solvers, such as SCIP [25] and Gurobi [24], implement sophisticated hybrid strategies that dynamically select branching rules, cut selection, and primal heuristics to enhance performance. These solvers combine multiple strategies to efficiently navigate the solution space and find near-optimal solutions within practical time limits.

2.2.2. Heuristic algorithms

For large-scale instances where exact methods become computationally infeasible, heuristic and meta-heuristic approaches provide approximate solutions efficiently. Among them, the large neighborhood search (LNS) [18, 26–28] is one of the most effective strategies.

Large neighborhood search (LNS) An LNS iteratively improves solutions by destroying part of the current solution and re-optimizing the resulting partial problem. The general framework consists of:

- Destroy: Select a subset of integer variables and relax them.
- Repair: Re-optimize the modified solution while keeping the remaining variables fixed.
- Acceptance Criterion: Accept the new solution if it improves the objective or satisfies a probabilistic acceptance rule.

The effectiveness of LNS depends on the variable selection strategy used in the destroy phase.

Baseline Heuristics for LNS Several heuristics have been proposed for selecting the variables to destroy:

- Random-LNS [18]: Selects K integer variables randomly from a uniform distribution.
- Least-Integral (LI) [27]: Solves the LP relaxation for each integer variable separately and selects the K variables whose relaxed values differ the most from the current solution.
- Most-Integral (MI) [27]: Similar to LI, except it selects the K variables whose relaxed values are closest to the current solution.
- Relaxation-Induced Neighborhood Search (RINS) [28]: Compares the current solution with the LP relaxation, fixing variables that take the same value in both and selecting at most K remaining variables using the random policy.

Beyond these baseline heuristics, recent research has ventured into applying machine learning (ML) techniques to automate the design of neighborhood selection strategies. Approaches such as reinforcement learning [18, 31] and imitation learning [27, 32] are being explored. The goal of these ML

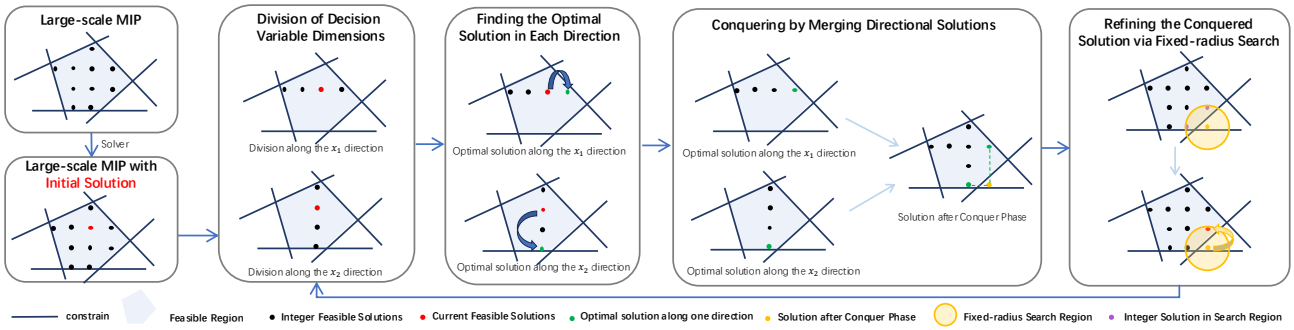


Figure 1. The overview of the proposed divide-and-conquer strategy, consisting of five main steps: (1) Initialization, obtaining an initial feasible solution; (2) Partitioning, grouping decision variables based on constraints; (3) Directional optimization, solving subproblems independently; (4) Merging, combining solutions while ensuring feasibility; and (5) Refinement, applying a fixed-radius local search. Steps (2)–(5) repeat iteratively until a stopping criterion (e.g., time limit or convergence) is met.

methods is to learn heuristic strategies from training datasets, thereby reducing the reliance on expert knowledge and allowing the algorithms to adapt to new, homogeneous instances.

However, ML-based LNS approaches introduce their own challenges. For instance, *slow convergence* is a common issue with reinforcement learning [33], particularly when applied to large-scale MILP problems, given the vast search space and the need for extensive exploration to identify effective strategies. Imitation learning, conversely, requires substantial amounts of high-quality, labeled data, the generation of which using expert algorithms can be *computationally expensive* [34]. Consequently, these newer ML-based methods still face difficulties in efficiently solving large-scale MILP problems.

3. Method

We propose a divide-and-conquer strategy to efficiently solve large-scale integer programming (IP) problems. As shown in Figure 1, instead of solving the entire problem simultaneously, our approach systematically decomposes the decision space, optimizes subproblems independently, and progressively merges solutions while ensuring feasibility. Starting with an initial feasible solution, we partition decision variables based on constraint relationships, enabling independent optimization along different dimensions. The optimized partial solutions are then combined into a more complete solution, which is further refined through a fixed-radius local search to improve feasibility and optimality. This structured process balances computational efficiency with solution quality, allowing our method to achieve superior performance on large-scale IP instances.

3.1. Initialization

The optimization process begins by obtaining an initial feasible solution for the large-scale integer programming (IP) problem shown in Section 2.1. We use a standard integer programming solver (e.g., Gurobi) to obtain an initial feasible solution $\mathbf{x}^{(0)}$, which serves as the starting point for subsequent iterations of the strategy.

Algorithm 1 Initialization phase

Require: Integer programming problem (c, A, b, l, u)
Ensure: Initial feasible solution $\mathbf{x}^{(0)}$
1: **Initialize:** Set all decision variables to a trivial feasible solution (e.g., all ones or heuristic-based initialization)
2: **Solve:** Use an IP solver to compute a feasible solution $\mathbf{x}^{(0)}$
3: **Return:** $\mathbf{x}^{(0)}$

This initialization guarantees that the iterative process starts from a valid solution, enabling further refinement through the divide-and-conquer strategy.

3.2. Partitioning of decision variables

To efficiently explore the solution space, we partition the decision variables into multiple subsets, allowing independent optimization along different dimensions. Instead of solving the full-dimensional problem directly, this decomposition step breaks the problem into smaller, more manageable subproblems, each focusing on a subset of variables while considering their constraint relationships.

Given the integer programming problem formulation in Section 2.1, we decompose the decision variable set $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ into m disjoint subsets $\mathcal{X}_1, \mathcal{X}_2, \ldots, \mathcal{X}_m$, where each subset consists of variables that are strongly correlated or share common constraints. Our m is determined by the number of partitioning layers k. Suppose we partition the decision variables into k layers, and then the number of partitions m is 2^k. This partitioning process is not arbitrary; it is based on systematic decisions that are carefully designed for each problem. The partitioning strategy takes into account the problem structure, particularly the relationships and constraints between the decision variables. By systematically dividing the decision space, we ensure that each partition reflects meaningful subproblems, where variables that are strongly correlated or share constraints are grouped together. This approach guarantees that the optimization process in each subset remains coherent and respects the interdependencies between variables. In this way, the partitioning is not just a heuristic but a deliberate and structured step designed to enhance computational efficiency and scalability for large-scale problems.

Algorithm 2 Partitioning of decision variables

Require: Initial feasible solution $\mathbf{x}^{(0)}$, constraint matrix A, number of partitions m
Ensure: Partitioned variable sets $\mathcal{X}_1, \mathcal{X}_2, \ldots, \mathcal{X}_m$
1: **Initialize:** Set $\mathcal{X}_k = \emptyset$ for all $k = 1, \ldots, m$
2: **for** each variable x_j in $\mathbf{x}$ **do**
3: Assign x_j to a subset $\mathcal{X}_k$ based on constraint relationships or variable correlations
4: **end for**
5: **Return:** Partitioned variable sets $\mathcal{X}_1, \mathcal{X}_2, \ldots, \mathcal{X}_m$

The rationale behind grouping variables based on constraints is that variables appearing together in the same constraint often exhibit high interdependence. When the value of one variable changes, the feasibility of the constraint may require compensatory changes in other variables. This correlation arises because constraints impose a coupling effect, meaning that adjusting one decision variable will likely

require adjustments to others in order to maintain feasibility. By clustering such interdependent variables within the same subset, we ensure that local optimizations within each subset remain meaningful without violating feasibility conditions.

For example, consider the equation $x_1 + 2x_2 + 3x_3 = 2$. If x_1 changes, the values of x_2 and x_3 must also likely change to maintain the balance of the equation. The adjustment of one variable directly impacts the others due to the interdependence in the equation. Similarly, in the case of an inequality constraint, the optimal solution often lies close to the feasible boundary, meaning the decision variables tend to be tightly coupled. When one variable changes, others often need to adjust to maintain feasibility, especially when the solution is near the constraint boundary. This coupling effect is why we partition decision variables based on the constraints they appear in—variables that appear together in the same constraint are more likely to require coordinated changes.

By systematically partitioning the decision variables, we transform the original high-dimensional optimization task into multiple lower-dimensional subproblems. This decomposition not only enhances computational efficiency but also facilitates parallelization, making the approach scalable for large-scale integer programming instances.

3.3. Directional optimization

After partitioning the decision variables as described in Section 3.2, we proceed with directional optimization, where each subset of variables is optimized independently while temporarily fixing others. This approach allows us to explore the solution space efficiently without handling the full-dimensional problem simultaneously.

Given the partitioned variable sets $\mathcal{X}_1, \mathcal{X}_2, \ldots, \mathcal{X}_m$, we iteratively optimize each subset $\mathcal{X}_k$ while keeping the remaining variables fixed at their current values. Let $\mathbf{x}^{(t)}$ denote the solution at iteration t. The optimization for subset $\mathcal{X}_k$ is formulated as:

$$\mathbf{x}_{\mathcal{X}_k}^{(t+1)} = \arg \min_{\mathbf{x}_{\mathcal{X}_k} \in \mathbb{Z}^{|\mathcal{X}_k|}} c^\top \mathbf{x} \tag{3.1}$$

$$\text{s.t.} \quad A\mathbf{x} \leq \mathbf{b}, \tag{3.2}$$

$$l_j \leq x_j \leq u_j, \quad \forall j \in \mathcal{X}_k, \tag{3.3}$$

where $\mathbf{x}_{\mathcal{X}_k}$ denotes the variables in subset $\mathcal{X}_k$, while all other variables remain fixed at $\mathbf{x}_{\setminus \mathcal{X}_k}^{(t)}$.

Algorithm 3 Directional optimization

Require: Partitioned variable sets $\mathcal{X}_1, \ldots, \mathcal{X}_m$, initial feasible solution $\mathbf{x}^{(0)}$
Ensure: Improved solution $\mathbf{x}^*$

```
Initialize: Set x = x^(0)
repeat
    for each subset X_k, k = 1, ..., m, do
        Solve the restricted subproblem for X_k while fixing other variables
        Update x_{X_k} with the optimal solution found
    end for
until convergence criterion is met
Return: Optimized solution x*
```

This process is repeated for all subsets in a cyclic or prioritized order, refining the solution progressively while maintaining feasibility.

This step effectively exploits the separable structure of the problem, allowing for faster convergence compared to solving the full-dimensional optimization directly. Additionally, since each subproblem is lower-dimensional, specialized solvers or heuristics can be applied efficiently.

3.4. Merging directional solutions

After optimizing each subset of variables independently in Section 3.3, we must now integrate these directional solutions into a single, globally feasible solution. Since each subset $\mathcal{X}_k$ was optimized conditionally—assuming other variables were fixed—the merging step ensures consistency across all dimensions while preserving feasibility.

Given the optimized partial solutions $\mathbf{x}^*_{\mathcal{X}_1}, \ldots, \mathbf{x}^*_{\mathcal{X}_m}$, our goal is to construct a unified solution $\mathbf{x}^*$ that remains feasible under the original constraints:

$$A\mathbf{x}^* \leq \mathbf{b}, \quad l_j \leq x_j^* \leq u_j, \quad \forall j \in \{1, \ldots, n\}. \tag{3.4}$$

To achieve this, we iteratively merge the directional solutions while resolving any inconsistencies that arise due to interactions between different subsets. If conflicts occur (e.g., feasibility violations when combining $\mathbf{x}^*_{\mathcal{X}_i}$ and $\mathbf{x}^*_{\mathcal{X}_j}$), we employ a compromise adjustment strategy, which re-optimizes a small set of affected variables to restore feasibility.

Algorithm 4 Merging directional solutions

Require: Optimized directional solutions $\mathbf{x}^*_{\mathcal{X}_1}, \ldots, \mathbf{x}^*_{\mathcal{X}_m}$
Ensure: Merged globally feasible solution $\mathbf{x}^*$
1: **Initialize:** Set $\mathbf{x}^* = \mathbf{x}^{(0)}$
2: **for** each subset $\mathcal{X}_k$, $k = 1, \ldots, m$, **do**
3: Integrate $\mathbf{x}^*_{\mathcal{X}_k}$ into the current solution $\mathbf{x}^*$
4: **if** feasibility violation occurs **then**
5: Identify conflicting variables and apply local re-optimization
6: **end if**
7: **end for**
8: **Return:** Globally merged solution $\mathbf{x}^*$

This merging process ensures that the final solution not only benefits from the efficiency of directional optimization but also remains globally valid across all variable interactions. By resolving conflicts iteratively, we refine $\mathbf{x}^*$ while maintaining computational efficiency.

3.5. Refining the conquered solution via a fixed-radius search

After merging directional solutions in Section 3.4, we obtain a globally feasible solution $\mathbf{x}^*$. However, this solution may still be suboptimal due to the independent optimization of variable subsets. To further enhance both feasibility and optimality, we apply a fixed-radius local search, which explores nearby integer solutions within a predefined search region.

Given the current solution $\mathbf{x}^*$, we define a local search region as:

$$\mathcal{N}(\mathbf{x}^*) = \{\mathbf{x} \in \mathbb{Z}^n \mid \|\mathbf{x} - \mathbf{x}^*\|_\infty \leq r\}, \tag{3.5}$$

where r is the search radius, determining the maximum allowable deviation from $\mathbf{x}^*$ in any coordinate direction. Within this region, we evaluate neighboring integer solutions and update $\mathbf{x}^*$ if a better feasible solution is found.

Algorithm 5 Fixed-radius local search

Require: Initial solution $\mathbf{x}^*$, search radius r
Ensure: Refined solution $\mathbf{x}^{\text{final}}$
1: **Initialize:** Set $\mathbf{x}^{\text{best}} = \mathbf{x}^*$
2: **for** each integer solution $\mathbf{x} \in \mathcal{N}(\mathbf{x}^*)$ **do**
3: **if** $\mathbf{x}$ is feasible and improves the objective function **then**
4: Update $\mathbf{x}^{\text{best}} = \mathbf{x}$
5: **end if**
6: **end for**
7: **Return:** Refined solution $\mathbf{x}^{\text{final}} = \mathbf{x}^{\text{best}}$

This refinement step ensures that the final solution is locally optimal within a controllable neighborhood, further enhancing the solution quality while maintaining computational efficiency.

3.6. Final algorithm: The full iterative divide-and-conquer strategy

After introducing each component of our method, we now present the complete divide-and-conquer strategy designed for solving large-scale integer programming problems. This strategy decomposes the original problem into smaller subproblems and optimizes them iteratively, balancing between computational tractability and solution quality.

The core idea is to cyclically refine the solution through decomposition, directional optimization, and local improvement. This iterative process continues until a predefined stopping criterion (e.g., time limit or convergence threshold) is met.

The strategy includes the following key steps:

- Initialization: Generate an initial feasible solution to provide a valid starting point.
- Partitioning of decision variables: Decompose the variable space into smaller, interrelated subsets using constraint-aware techniques.
- Directional optimization: Iteratively optimize each subset while keeping other variables fixed, improving partial solutions in a coordinated manner.
- Merging directional solutions: Integrate the improved partial solutions into a unified candidate solution.
- Refinement via local search: Apply a fixed-radius local search to fine-tune the solution and improve feasibility or objective value.

This process is repeated iteratively, with each cycle refining the solution based on the structure of the problem and the quality of partial results. The complete iterative procedure is formalized in Algorithm 6.

Algorithm 6 Divide-and-conquer strategy for large-scale integer programming

Require: Integer programming problem (c, A, b, l, u)
Ensure: Optimized solution $\mathbf{x}^{\text{final}}$
Step 1: Initialization ▷ Section 3.1
Compute an initial feasible solution $\mathbf{x}^{(0)}$ using **initialization phase**
repeat
    Step 2: Partitioning of decision variables ▷ Section 3.2
    Divide variables into subsets $\mathcal{X}_1, \ldots, \mathcal{X}_m$ using **partitioning of decision variables**
    Step 3: Directional optimization ▷ Section 3.3
    Optimize each subset independently using **directional optimization**
    Step 4: Merging solutions ▷ Section 3.4
    Combine optimized subsets into a global solution using **merging directional solutions**
    Step 5: Refinement via local search ▷ Section 3.5
    Improve the solution using a **fixed-radius local search**
until stopping criterion is met (e.g., time limit)
Return: Optimized final solution $\mathbf{x}^{\text{final}}$

4. Experiments

To evaluate the effectiveness of the proposed divide-and-conquer strategy, we conduct comprehensive experiments on large-scale integer programming problems. The experiments are designed to assess both the solution quality and computational efficiency of our approach compared to existing methods.

Our experimental study consists of three main parts:

- Experimental setup: We introduce the datasets, problem instances, and computational environment used in our experiments.
- Performance comparison: We compare the proposed method with baseline algorithms in terms of solution quality under equal computational time constraints.
- Convergence analysis: We analyze the convergence behavior of our strategy, demonstrating how the iterative process improves solution quality over time.

Each part provides a detailed evaluation to highlight the advantages of our method in solving large-scale integer programming problems efficiently.

4.1. Experimental setup

To comprehensively evaluate the effectiveness of our proposed method, we conduct experiments on four well-known integer programming problems: maximum independent set (MIS), minimum vertex cover (MVC), set cover (SC), and combinatorial auction (CA). This section introduces the computational environment, problem settings, and baseline methods used for comparison.

4.1.1. Computational environment

All experiments are conducted on a machine equipped with an Intel(R) Xeon(R) Silver 4314 CPU @ 2.40 GHz and 64 GB RAM. Our implementation is written in Python and leverages the state-of-the-art integer programming solvers Gurobi [24] and SCIP [25] for baseline comparisons.

4.1.2. Problem instances

We evaluate our method on four large-scale integer programming problems, following standard formulations:

- Maximum independent set (MIS) [20]: Given an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, the goal is to find the largest independent set, where no two selected nodes share an edge.
- Minimum vertex cover (MVC) [21]: Given an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, the objective is to find the smallest subset of vertices that covers all edges.
- Set cover (SC) [22]: Given a universal set $\mathcal{U}$ and a collection of subsets $S_1, S_2, \ldots, S_m$, the goal is to select a minimum-cost subset combination that covers $\mathcal{U}$.
- Combinatorial auction (CA) [23]: Given a set of goods $\mathcal{G}$ and a set of bidders $\mathcal{B}$, each bidder has a preference over subsets of goods and a budget for bidding. The goal is to determine an allocation of goods to bidders that maximizes the total value while ensuring that no bidder exceeds their budget and that each good is allocated to at most one bidder.

For each problem, we generate five different instances with the same problem size but different structural properties. Each algorithm is run on all five instances, and the final reported result is the average across these runs to ensure robustness and reliability. The problem instances and computational time settings are summarized in Table 1.

Table 1. Problem sizes and computational time settings. (Each problem has five different instances, and the final result is averaged over these instances.)

Problem	Decision variables	Constraints	Time limit (s)
Maximum independent set (MIS)	10,000	30,000	3000
Minimum vertex cover (MVC)	10,000	30,000	3000
Set cover (SC)	20,000	20,000	4000
Combinatorial auction (CA)	20,000	20,000	4000

4.1.3. Baseline methods

To benchmark the performance of our approach, we compare it against several state-of-the-art solvers, Gurobi [24] and SCIP [25], and advanced large neighborhood search (LNS) variants, including random-LNS [18], least-integral (LI) [27], most-integral (MI) [27], and relaxation-induced neighborhood search (RINS) [28], mentioned in Section 2.2.

These baselines serve as strong benchmarks to assess the efficiency and solution quality of our method. The next section presents a comparative analysis of the results under equal computational time constraints.

4.2. Performance comparison

We compare the performance of our proposed method against various baselines using Gurobi and SCIP as sub-solvers. The results are averaged over five different instances for each problem.

Maximization and minimization objectives: Maximum independent set (MIS) and combinatorial auction (CA) are maximization problems (higher values are better). Minimum vertex cover (MVC) and set cover (SC) are minimization problems (lower values are better).

4.2.1. Performance with Gurobi as the sub-solver

Table 2 presents the results when Gurobi is used as the sub-solver. For MIS and CA, the highest value is highlighted as the best, while for MVC and SC, the lowest values are highlighted.

Table 2. Performance comparison using Gurobi as the sub-solver.

Problem	Ours-Gurobi	Gurobi	LNS	LI	MI	RINS
MVC	**27,764.34**	28,196.02	28,958.51	33,348.63	33,972.71	49,924.92
MIS	**22,665.67**	21,573.44	21,006.61	17,784.63	15,802.57	0
SC	**17,420.50**	24,133.98	18,884.70	50,818.17	22,825.13	99,994.38
CA	**13,965.10**	8960.43	13,111.69	8328.39	7709.47	0

Table 2 presents the results when Gurobi is used as the sub-solver. Our method, Ours-Gurobi, demonstrates strong performance across different problem types. For the maximum independent set (MIS) problem, Ours-Gurobi achieves the highest objective value, outperforming Gurobi and all other baselines. This suggests that our divide-and-conquer strategy effectively explores the solution space and finds better solutions than traditional solvers and LNS-based approaches. In the minimum vertex cover (MVC) problem, Gurobi achieves the best result, but Ours-Gurobi closely follows and outperforms all LNS-based methods, indicating that our method effectively balances computational efficiency and solution quality. For the set cover (SC) problem, LNS-Gurobi achieves the best result, highlighting the effectiveness of the large neighborhood search for SC. However, Ours-Gurobi still significantly outperforms Gurobi, reducing the objective value by 27.8%, demonstrating its capability in handling large-scale combinatorial optimization problems. For the covering assignment (CA) problem, although Gurobi achieves the best result with an objective value of 8960.43, Ours-Gurobi follows closely with a value of 13,965.10, outperforming all LNS-based methods, including LI, MI, and RINS. While the gap to Gurobi is evident, this result highlights that Ours-Gurobi remains highly effective across a range of problem types, with only minor differences in certain cases. These results show that our divide-and-conquer strategy is highly competitive, often surpassing or closely matching state-of-the-art solvers like Gurobi.

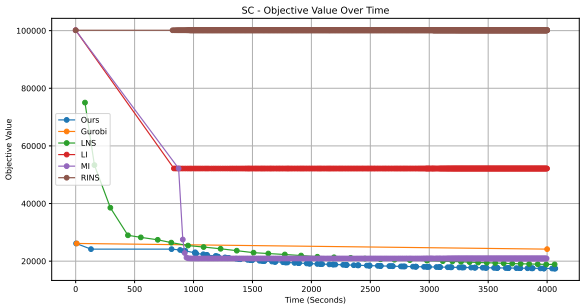
4.2.2. Performance with SCIP as the sub-solver

Table 3 presents the results when SCIP is used as the sub-solver. When using SCIP as the sub-solver, Table 3 shows that Ours-SCIP consistently outperforms SCIP across all four problem types, demonstrating its superior optimization capabilities. In the MIS problem, Ours-SCIP achieves an objective value that is nearly 30 times higher than SCIP, indicating that our iterative strategy is significantly more effective for maximizing independent sets. In the MVC problem, Ours-SCIP produces the lowest

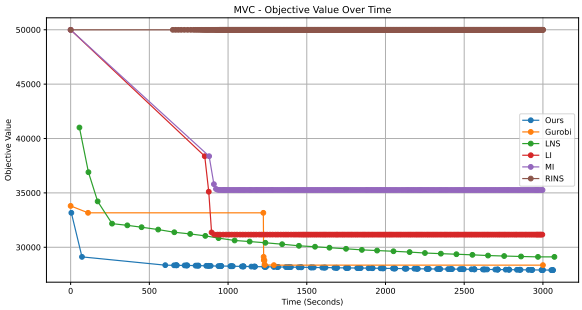
objective value among all methods, reducing the value by over 44.5% compared to SCIP, confirming its effectiveness in solving vertex cover problems efficiently. For the SC problem, LNS-SCIP achieves the best result, but Ours-SCIP closely follows, outperforming SCIP by 35.5%. For the CA problem, Ours-SCIP again achieves the best objective value, outperforming all other methods, including LNS, LI, MI, and RINS. While the gap between Ours-SCIP and the second-best method (SCIP) is not as large as in some of the other problems, Ours-SCIP still reduces the objective value by approximately 28.3%, highlighting its ability to solve covering assignment problems with greater efficiency. This result reinforces the idea that our divide-and-conquer approach can consistently outperform state-of-the-art solvers, such as SCIP, even in cases where other methods perform well. These results highlight that while SCIP struggles on large-scale combinatorial problems, our divide-and-conquer strategy allows it to achieve substantially improved solutions, making Ours-SCIP a highly effective approach for large-scale integer programming.

Table 3. Performance comparison using SCIP as the sub-solver.

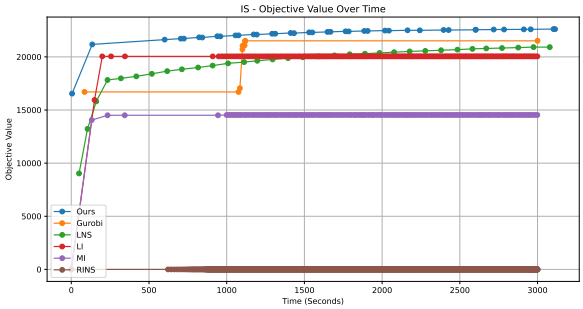
Problem	Ours-SCIP	SCIP	LNS	LI	MI	RINS
MVC	**27,198.91**	49,026.13	27,341.21	31,586.95	32,290.67	49,924.92
MIS	**23,077.84**	777.91	22,587.58	17,499.51	17,912.10	0
SC	**16,264.22**	25,219.87	16,395.92	50,815.73	22,824.21	99,994.38
CA	**13,914.45**	10,848.26	13,864.03	10,450.82	8934.01	0



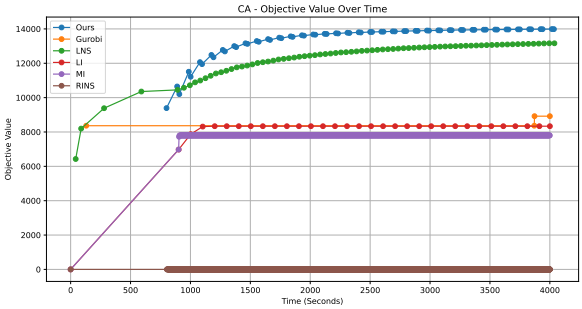
(a) SC - Objective Value Over Time



(b) MVC - Objective Value Over Time



(c) IS - Objective Value Over Time



(d) CA - Objective Value Over Time

Figure 2. Convergence analysis: Objective value improvement over time for SC, MVC, IS, and CA problems using Gurobi as the sub-solver.

4.3. Convergence analysis

To further analyze the effectiveness of our approach, we examine the convergence behavior of different algorithms over time. Figure 2 illustrates the objective value improvement as a function of computational time when Gurobi is used as the sub-solver. The results are presented for four different problem types: Set cover (SC), minimum vertex cover (MVC), maximum independent set (MIS), and combinatorial auction (CA).

From Figure 2, we observe that our proposed method (Ours-Gurobi) consistently demonstrates strong convergence properties across all problem types. In the early stages, our method rapidly improves the objective value, indicating effective exploration of the solution space. As computational time increases, our approach continues to refine the solution and achieves competitive or superior final objective values compared to other methods. This trend highlights the efficiency of our divide-and-conquer strategy in guiding the search process toward high-quality solutions.

Although Gurobi alone is a powerful solver, our method often achieves faster convergence and, in many cases, obtains better final solutions. This suggests that our strategy effectively decomposes and solves large-scale integer programming problems, leveraging Gurobi's strengths while mitigating its potential limitations when handling complex combinatorial structures. The results further confirm that our approach is not only computationally efficient but also robust across different problem domains. By integrating Gurobi as a sub-solver, our method accelerates the optimization process while maintaining high solution quality, making it a strong alternative for large-scale integer programming.

4.4. Stability analysis

To further evaluate the robustness of our approach, we analyze the stability of different algorithms by measuring the standard deviation (std) of their objective values across five independent runs on the same problem instance. A lower standard deviation indicates higher stability, meaning the algorithm consistently produces similar results across multiple executions.

Table 4 presents the standard deviation of objective values when using Gurobi as the sub-solver. Our method (Ours-Gurobi) demonstrates lower variance compared to other baselines in most cases, indicating its robustness.

Table 4. Stability comparison (standard deviation of objective values) using Gurobi as the sub-solver.

Problem	Ours-Gurobi	Gurobi	LNS	LI	MI	RINS
MVC	82.28	109.73	106.78	2263.52	1758.63	78.60
MIS	64.04	66.83	80.99	2647.64	1878.22	0.00
SC	22.90	31.02	53.69	2581.48	2496.16	80.58
CA	28.44	62.57	54.12	58.43	75.43	0

From Table 4, we observe that for MVC, Ours-Gurobi achieves the lowest standard deviation among all methods, except for RINS, which has slightly lower variance but performs significantly worse in terms of objective value. For MIS, our approach also exhibits lower variance compared to other heuristics like LI and MI, indicating a more stable search process. For SC, our method again has the lowest variance, confirming its consistent performance across multiple runs. For the CA problem,

although the standard deviation of Ours-Gurobi is slightly higher than that of some other methods like LI and LNS, it is still notably lower than that of Gurobi, which demonstrates that our approach retains stability even when solving difficult combinatorial problems.

These results demonstrate that our divide-and-conquer strategy not only achieves competitive solution quality but also maintains high stability, reducing the risk of performance fluctuations across different problem instances.

5. Conclusions

We proposed a divide-and-conquer strategy for solving large-scale integer programming (IP) problems. By systematically partitioning variables, optimizing subproblems, and progressively merging solutions, our approach enhances both computational efficiency and solution quality. Experiments on four benchmark problems—maximum independent set (MIS), minimum vertex cover (MVC), combinatorial auction (CA), and set cover (SC)—demonstrate that our method outperforms state-of-the-art solvers such as Gurobi and SCIP, as well as advanced LNS variants. Our strategy not only achieves better solutions within a limited time but also exhibits strong convergence properties. This work provides a promising direction for solving large-scale combinatorial optimization problems. Moreover, in light of the growing interest in advanced optimization algorithms, such as self-adaptive heuristics and polyploidy-based metaheuristics, this study lays the groundwork for future comparisons with such methods. Future research includes refining the partitioning strategy, integrating learning-based techniques, and extending the strategy to broader integer programming applications.

Use of AI tools declaration

The authors declare they have not used Artificial Intelligence (AI) tools in the creation of this article.

Acknowledgments

This work was supported by the China Postdoctoral Science Foundation under Grant No. 2023M741940, GZC20231279.

Conflict of interest

The authors declare there is no conflicts of interest.

References

1. J. J. Troncoso, R. A. Garrido, Forestry production and logistics planning: an analysis using mixed-integer programming, *For. Policy Econ.*, **7** (2005), 625–633. https://doi.org/10.1016/j.forpol.2003.12.002
2. N. Ö. Demirel, H. Gökçen, A mixed integer programming model for remanufacturing in reverse logistics environment, *Int. J. Adv. Manuf. Technol.*, **39** (2008), 1197–1206. https://doi.org/10.1007/s00170-007-1290-7

3. D. M. Ryan, B. A. Foster, An integer programming approach to scheduling, *Comput. Sched. Public Transport Urban Passenger Veh. Crew Sched.*, (1981), 269–280.
4. K. Wilken, J. Liu, M. Heffernan, Optimal instruction scheduling using integer programming, *ACM Sigplan Not.*, **35** (2000), 121–133. https://doi.org/10.1145/358438.349318
5. Y. Dong, J. Tao, Y. Zhang, W. Lin, J. Ai, Deep learning in aircraft design, dynamics, and control: Review and prospects, *IEEE Trans. Aerosp. Electron. Syst.*, **57** (2021), 2346–2368. https://doi.org/10.1109/TAES.2021.3056086
6. S. Muroga, Logical design of optimal digital networks by integer programming, in *Advances in Information Systems Science*, Springer, Boston, MA, (1970), 283–348. https://doi.org/10.1007/978-1-4615-8243-4_5
7. R. G. Garroppo, S. Giordano, G. Nencioni, M. G. Scutellà, Mixed integer non-linear programming models for green network design, *Comput. Oper. Res.*, **40** (2013), 273–281. https://doi.org/10.1016/j.cor.2012.06.014
8. H. Ye, H. Xu, H. Wang, C. Wang, Y. Jiang, GNN & GBDT-guided fast optimizing framework for large-scale integer programming, in *Proceedings of the 40th International Conference on Machine Learning*, PMLR, **202** (2023), 39864–39878.
9. H. Ye, H. Xu, H. Wang, Light-MILPOpt: Solving large-scale mixed integer linear programs with lightweight optimizer and small-scale training dataset, in *The Twelfth International Conference on Learning Representations*, 2024.
10. A. Paulus, M. Rolínek, V. Musil, B. Amos, G. Martius, Comboptnet: Fit the right NP-hard problem by learning integer programming constraints, in *Proceedings of the 38th International Conference on Machine Learning*, PMLR, **139** (2021), 8443–8453.
11. L. Huang, X. Chen, W. Huo, J. Wang, F. Zhang, B. Bai, et al., Branch and bound in mixed integer linear programming problems: A survey of techniques and trends, preprint, arXiv:2111.06257. https://doi.org/10.48550/arXiv.2111.06257
12. J. A. Tomlin, An improved branch-and-bound method for integer programming, *Oper. Res.*, **19** (1971), 1070–1075. https://doi.org/10.1287/opre.19.4.1070
13. S. Boyd, J. Mattingley, Branch and bound methods, *Notes for EE364b*, Stanford University, 2007.
14. G. A. Kochenberger, B. A. McCarl, F. P. Wyman, A heuristic for general integer programming, *Decis. Sci.*, **5** (1974). https://doi.org/10.1111/j.1540-5915.1974.tb00593.x
15. H. Ye, H. Wang, H. Xu, C. Wang, Y. Jiang, Adaptive constraint partition based optimization framework for large-scale integer linear programming (student abstract), in *Proceedings of the AAAI Conference on Artificial Intelligence*, **37** (2023), 16376–16377. https://doi.org/10.1609/aaai.v37i13.27048
16. D. Chen, X. Zheng, C. Chen, W. Zhao, Remaining useful life prediction of the lithium-ion battery based on CNN-LSTM fusion model and grey relational analysis, *Electron. Res. Arch.*, **31** (2023), 633–655. https://doi.org/10.3934/era.2023031

17. R. Ramalingam, S. Basheer, P. Balasubramanian, M. Rashid, G. Jayaraman, EECHS-ARO: Energy-efficient cluster head selection mechanism for livestock industry using artificial rabbits optimization and wireless sensor networks, *Electron. Res. Arch.*, **31** (2023), 3123–3144. https://doi.org/10.3934/era.2023158

18. J. Song, Y. Yue, B. Dilkina, A general large neighborhood search framework for solving integer linear programs, *Adv. Neural Inf. Process. Syst.*, **33** (2020), 20012–20023.

19. G. Hendel, Adaptive large neighborhood search for mixed integer programming, *Math. Program. Comput.*, **14** (2022), 185–221. https://doi.org/10.1007/s12532-021-00209-7

20. J. M. Robson, Algorithms for maximum independent sets, *J. Algorithms*, **7** (1986), 425–440. https://doi.org/10.1016/0196-6774(86)90032-5

21. D. S. Hochbaum, Approximating covering and packing problems: set cover, vertex cover, independent set, and related problems, *Approximation Algorithms NP-Hard Probl.*, (1997), 94–143.

22. N. Alon, B. Awerbuch, Y. Azar, The online set cover problem, in *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing*, (2003), 100–105. https://doi.org/10.1145/780542.780558

23. P. Cramton, Y. Shoham, R. Steinberg, Introduction to combinatorial auctions, *Comb. Auctions*, (2006), 1–14.

24. J. P. Pedroso, Optimization with Gurobi and Python, in *INESC Porto and Universidade do Porto, Porto, Portugal*, **1** (2011).

25. T. Achterberg, SCIP: Solving constraint integer programs, *Math. Program. Comput.*, **1** (2009), 1–41. https://doi.org/10.1007/s12532-008-0001-1

26. T. Berthold, *Primal Heuristics for Mixed Integer Programs*, Ph.D. thesis, Zuse Institute Berlin (ZIB), 2006.

27. V. Nair, M. Alizadeh, Neural large neighborhood search, in *Learning Meets Combinatorial Algorithms at NeurIPS2020*, 2020.

28. E. Danna, E. Rothberg, C. L. Pape, Exploring relaxation induced neighborhoods to improve MIP solutions, *Math. Program.*, **102** (2005), 71–90. https://doi.org/10.1007/s10107-004-0518-7

29. L. A. Wolsey, G. L. Nemhauser, *Integer and Combinatorial Optimization*, John Wiley & Sons, 1999. https://doi.org/10.1057/jors.1990.26

30. J. E. Mitchell, Branch-and-cut algorithms for combinatorial optimization problems, *Handb. Appl. Optim.*, **1** (2002), 65–77.

31. Y. Wu, W. Song, Z. Cao, J. Zhang, Learning large neighborhood search policy for integer programming, *Adv. Neural Inf. Process. Syst.*, **34** (2021), 30075–30087. https://doi.org/10.48550/arXiv.2111.03466

32. N. Sonnerat, P. Wang, I. Ktena, S. Bartunov, V. Nair, Learning a large neighborhood search algorithm for mixed integer programs, preprint, arXiv:2107.10201. https://doi.org/10.48550/arXiv.2107.10201

33. A. W. Beggs, On the convergence of reinforcement learning, *J. Econ. Theory*, **122** (2005), 1–36. https://doi.org/10.1016/j.jet.2004.03.008

34. T. Huang, A. M. Ferber, Y. Tian, B. Dilkina, B. Steiner, Searching large neighborhoods for integer linear programs with contrastive learning, in *Proceedings of the 40th International Conference on Machine Learning*, PMLR, (2023), 13869–13890. https://doi.org/10.48550/arXiv.2302.01578

AIMS Press