Objective Domain Reduction for Enhancing Solver Performance on Challenging Integer Programs

Aadya Bhattarai[1], Lagnajita Basu[1], Tapas K. Das[1], Kimia Keshanian[2], Hadi Charkhgard[1]

[1]Department of Industrial and Management Systems Engineering, University of South Florida, Tampa, Florida, USA

[2]Information and Technology Management Department, University of Tampa, Tampa, Florida, USA

Abstract

In this study, we explore how the domain of objective function values for challenging integer programs can be reduced and whether such a reduction can improve the solution process. Our work is motivated by binary search, a technique that efficiently narrows a search space by repeatedly halving it through feasibility checks. Building on this idea, we first propose a method that combines binary search with branch-and-bound techniques by encoding objective values in terms of bits, which we call the Bitwise Branch-and-Bound method. The method attempts to determine the value of each bit, from the most significant bit to the least significant one, progressively tightening the objective value bounds. When these bounds converge to a single value, the optimal solution is identified directly. Otherwise, the method returns a set of disjoint ranges guaranteed to contain the optimal value, effectively reducing the objective-value domain. To exploit this reduction, we introduce a reformulation, referred to as the Domain-Reduced Reformulation, which passes the disjoint ranges to an integer programming solver. We apply the proposed framework, including both the Bitwise Branch-and-Bound method and the Domain-Reduced Reformulation, to two classes of challenging integer programs: the Traveling Salesman Problem and the Capacitated Vehicle Routing Problem. The proposed approach is evaluated on standard benchmark instances and compared against the direct use of an integer programming solver with lazy constraint callbacks. While the standard solver performs better on easy-to-medium instances, the proposed method provides significant improvements on more challenging instances in terms of both computational time and optimality gap.

Keywords: Integer Program, Objective Domain Reduction, Binary Search, Branch and Bound, Domain-Reduced Reformulation

1 Introduction

Integer Programming (IP) restricts a subset of decision variables to discrete values, enabling the modeling of a wide range of complex real-world applications. However, unlike linear programs,

which can be solved in polynomial time, integer programs are generally computationally challenging. Such problems can be addressed using several exact approaches (Clautiaux and Ljubić 2025, Belenguer and Benavent 2003, Contardo et al. 2023, Achuthan et al. 2003, Applegate et al. 2011). Among these, branch-and-bound remains the most widely used framework. It operates by solving Linear Programming (LP) relaxations to obtain bounds and recursively partitioning the feasible region through branching decisions (Morrison et al. 2016). Branch-and-bound forms the foundation of modern generic IP solvers such as *Gurobi*, *CPLEX*, and *SCIP* (Clautiaux and Ljubić 2025). Although these solvers are highly effective for many practically sized integer programs, they still struggle on particularly challenging instances. In such cases, it is often necessary to sacrifice generality and instead develop customized solution methodologies that exploit problem-specific structures. One common strategy is to embed IP solvers within decomposition frameworks, e.g., Benders decomposition or column generation, to construct custom-built exact methods (Fukasawa et al. 2006, Lübbecke and Desrosiers 2005, Rahmaniani et al. 2017, Barnhart et al. 1998, Pecin et al. 2017, Errami et al. 2024).

It is worth noting that IP solvers are not the only tools available for solving integer programs. Constraint Programming (CP) solvers provide an alternative optimization paradigm for certain classes of discrete optimization problems. Rather than relying primarily on branch-and-bound, CP solvers employ constraint propagation techniques to systematically eliminate values from variable domains that cannot participate in any feasible solution (Rossi et al. 2006, Stuckey 2010). This capability makes CP solvers particularly effective for detecting infeasibility early in the search process. Consequently, recent research has increasingly focused on hybrid optimization frameworks that integrate CP and IP methodologies to develop faster and more robust approaches for solving challenging combinatorial optimization problems (Hooker and Van Hoeve 2018).

Despite these advances, improving the performance of generic IP solvers on challenging integer programs remains an active area of research, and the present study contributes to this direction. Specifically, we focus on minimization problems with integer-valued objective functions and investigate whether reducing the domain of objective values can improve the performance of IP solvers on challenging instances. Our study is motivated by binary search, a classical technique that repeatedly halves a search interval to locate a target value. In the context of optimization, binary search progressively reduces the objective-value domain by partitioning the range of possible objective values into two subsets and eliminating one subset at each iteration. Although binary search is well known from a computational complexity perspective, its practical potential as a solution methodology for difficult combinatorial optimization problems remains largely unexplored. The primary reason is that each iteration requires solving a feasibility problem, which is theoretically as difficult as the original optimization problem itself. Consequently, the direct application of binary search can become computationally expensive for challenging integer programs.

To address this limitation, we propose a solution framework aimed at improving the performance of IP solvers through objective-domain reduction. To this end, we first introduce a method termed the *Bitwise Branch-and-Bound* method, which combines binary search with branch-and-bound by

encoding objective values in binary form. The proposed method determines the value of each bit sequentially, starting from the most significant bit and proceeding toward the least significant bit. At each node of the branch-and-bound tree, the algorithm fixes the current bit to a candidate value and performs a feasibility check to determine whether a feasible solution exists with an objective value at or below the resulting threshold, conceptually similar to the binary search process.

Performing these feasibility checks requires a feasibility-check solver, which can be any suitable optimization solver. However, because CP solvers are often highly effective at detecting infeasibility, we employ Google's CP-SAT solver (Perron and Didier 2025) for this purpose in our implementation. To address the computational limitations of binary search discussed earlier, we impose a time limit on each feasibility query. If the solver resolves the feasibility problem within the allotted time, the corresponding bit is fixed, the objective-value interval is reduced, and the algorithm proceeds to the next bit. Otherwise, if the solver fails to resolve the query within the specified time limit, the current bit remains undetermined, and the node is divided into two subproblems corresponding to the two possible values of that bit, each associated with tighter objective bounds.

The proposed Bitwise Branch-and-Bound method can directly identify the optimal objective value because it progressively reduces the objective-value domain, and once this domain collapses to a single value, optimality is established immediately. However, if the procedure terminates prematurely due to imposed time limits, it can still serve as an effective preprocessing tool for challenging integer programs. In such cases, the method returns a collection of disjoint objective ranges guaranteed to contain the optimal value, together with a feasible solution corresponding to the best known global upper bound. Any objective value outside these ranges has already been proven infeasible or suboptimal through the branching and pruning process. Each disjoint range is associated with an open node corresponding to a partial bit assignment that restricts the objective value to a specific interval. To exploit the information generated by the Bitwise Branch-and-Bound method when it terminates prematurely, we introduce a reformulation referred to as the *Domain-Reduced Reformulation*. This reformulation uses the disjoint objective ranges to construct a reduced model that is subsequently passed to an IP solver. As a result, the solver operates over a substantially smaller search space, thereby improving the overall efficiency of the solution process.

We apply the proposed framework, including both the Bitwise Branch-and-Bound method and the Domain-Reduced Reformulation, to two well-studied classes of challenging integer programming problems: the Traveling Salesman Problem (TSP) and the Capacitated Vehicle Routing Problem (CVRP). Both problems possess widely used benchmark libraries, such as TSPLIB and CVRPLIB, making them suitable testbeds for evaluating new optimization methodologies. For both the TSP and CVRP, state-of-the-art IP solvers achieve significant performance gains when enhanced with lazy constraint callbacks to efficiently handle their constraint structures; accordingly, we adopt this implementation strategy in our experiments.

Computational experiments on standard benchmark instances demonstrate that the Bitwise Branch-and-Bound method alone can solve many instances to proven optimality. However, for smaller instances, the proposed framework does not offer an advantage over state-of-the-art IP

solvers equipped with lazy constraint callbacks. In contrast, for larger and more difficult instances where the Bitwise Branch-and-Bound method cannot completely determine the objective value within the allotted time, the resulting disjoint ranges provide structured guidance to the IP solver through the proposed Domain-Reduced Reformulation. By restricting the search to narrowed subranges of the objective space, the IP solver is able to reach optimality more rapidly than when solving the original formulation directly. Even when neither phase alone completely closes the optimality gap, the framework consistently produces feasible solutions together with tighter bounds, thereby providing practitioners with valuable information that may not be obtainable from a standalone solver.

The remainder of this paper is organized as follows. Section 2 reviews the relevant literature. Section 3 formally defines the class of integer programs considered in this study and introduces the binary-encoded reformulation of the objective function. Section 4 presents a high-level overview of the proposed framework. Section 5 describes the framework components in detail, including the Bitwise Branch-and-Bound procedure, the fathoming rules, and the construction of the Domain-Reduced Reformulation. Section 6 reports computational experiments on benchmark instances of the TSP and CVRP. Finally, in Section 7, we provide some concluding remarks.

2 Literature Review

Since binary search and binary encoding are two of the main building blocks of our proposed solution framework, in this section we briefly review a subset of the literature relevant to these domains.

2.1 Binary Search

Binary search is a classical technique that finds a target value by repeatedly halving a search range. Suppose the optimal objective value lies somewhere between a lower bound L and an upper bound U. The algorithm picks the midpoint $k = (L + U)/2$ and asks whether a feasible solution with cost $\leq k$ exists. If such a solution exists, the optimum must be at most k, so the search continues in $[L, k]$. If no such solution exists, the optimum must be greater than k, so the search continues in $(k, U]$. By repeating this process, the algorithm narrows down to the optimal value in very few steps. Zhang et al. (2013) applied this idea to a rectangle packing problem where the task is to pack a set of rectangles into a strip of fixed width using the least possible height. Their algorithm performs a binary search over the height. At each step, it checks whether a feasible packing can be found within a height k using a placement heuristic. Yao et al. (2024) used a similar strategy for an irregular bin packing problem from the steel industry, where leftover steel sheets of different shapes and sizes serve as bins. Since there are many possible combinations of bins, they sort these combinations by total area and binary search over them to find the smallest total area that still fits all the pieces. Khatami and Salehipour (2020) proposed a binary search heuristic for a single machine coupled task scheduling problem, where the goal is to minimize the makespan. Given a

lower and upper bound on the makespan, they pick a value between the two bounds and ask the solver whether a schedule with a makespan at most this value exists. The answer is used to tighten either the lower or the upper bound, and the process repeats until the bounds converge. They showed that their binary search heuristic outperforms the standard solver Gurobi on benchmark instances.

2.2 Binary Encoding

Binary encoding has been used in reformulations of integer programming models in various studies. Early work applied binary encoding to represent a bounded integer variable using binary variables corresponding to its base-2 digits. In particular, if an integer variable x has a finite upper bound u, then it can be written as $x = \sum_{j=0}^{k} 2^j y_j$ where $y_j \in \{0, 1\}$ for all j, and k is the smallest integer such that $u \leq 2^{k+1} - 1$. For example, if $x \in \{0, \ldots, 7\}$, then x can be represented as $x = 4y_2 + 2y_1 + y_0$ where $y_0, y_1, y_2 \in \{0, 1\}$.

This representation was used by Watters (1967) in reducing integer polynomial programming problems to binary linear programming problems. Later, Owen and Mehrotra (2002) examined whether replacing general integer variables by their binary expansions improves the solution of mixed-integer linear programs. They found that such reformulations do not generally help standard branch-and-bound, because nearly all of the introduced binary variables still need to be explored during branching. Adams and Henry (2012) used the base-2 expansion for linearizing products of functions of discrete variables. Such products frequently appear in the objective or constraints of the optimization problem which makes the problem nonlinear and difficult to solve. By expressing each discrete variable through its binary expansion, they rewrote these products as linear expressions. Their approach requires only about $\lceil \log_2 n \rceil$ binary variables for a variable with n possible values, compared to the n binary variables needed by standard methods. This leads to significantly smaller formulations. Muldoon et al. (2013) then studied the mathematical quality of these base-2 reformulations. They proved that the linear programming relaxation produced by a base-2 expansion is as tight as possible and does not introduce unnecessary fractional solutions. This finding suggests that while pure branching on bit variables may not help, as Owen and Mehrotra (2002) showed, the polyhedral quality of the binary expansion is strong. Such expansions can therefore be effective when combined with cutting planes or other tightening techniques.

Binary encoding also plays an important role in decomposition methods for integer programs. For example, Benders decomposition partitions a problem into a master problem and at least one subproblem, and iteratively generates cuts in the master problem using information obtained from the subproblem (Benders 1962). Classical Benders cuts are derived from the dual of the subproblem and therefore require the subproblem to be formulated as a linear program. When such a derivation is not available, for example when the subproblem is nonlinear or when big-M coefficients lead to weak classical cuts, a common alternative is the no-good cut (Codato and Fischetti 2006). A no-good cut excludes the current master solution by enforcing that at least one binary variable must change its value. Since no-good cuts are naturally expressed in terms of binary variables, they apply

directly when the master problem is binary. However, when the master problem contains general integer variables, these variables are often converted into binary form so that standard no-good cuts can be applied. Forbes et al. (2024) employ this idea within a logic-based Benders decomposition framework for a staffing problem. Specifically, each integer staffing variable is replaced with a set of binary indicator variables, where each indicator corresponds to one possible value of the original integer variable. An indicator variable takes value one if the integer variable assumes the corresponding value and zero otherwise, ensuring that exactly one indicator is active at any time. Whenever a candidate solution is found to be suboptimal, a no-good cut is generated using these binary indicators.

More recently, Ghasemi Saghand et al. (2023) utilized binary encoding to represent the product operation and used it as the foundation of a new family of solution methods for an important class of integer optimization problems known as multiplicative programs, where the objective function consists of the product of several integer variables. Their method represents the outcome of the product function in terms of binary bits and determines the optimal objective value one bit at a time, starting from the most significant bit. At each iteration, a single integer linear program is solved to determine whether the current bit should take the value 0 or 1, after which the algorithm proceeds to the next bit and repeats the process. This bitwise strategy avoids the numerical instabilities that arise when multiplying many variables together, since the full product is never computed explicitly. Computational experiments demonstrated that, among the tested approaches, their method was the only one that remained numerically stable when the problem parameters became large.

3 Problem Description

We consider integer programs with non-negative integer-valued objective functions of the form:

$$z^* = \min\{f(x) : x \in \mathcal{X}\}, \tag{1}$$

where $\mathcal{X} \subseteq \mathbb{Z}^n$ denotes the feasible region and $f : \mathcal{X} \to \mathbb{Z}_{\geq 0}$ is the objective function. We assume that both the objective function and the constraints are linear, so that the problem corresponds to a standard linear integer programming formulation with an integer-valued cost function. Note that the non-negativity assumption is made without loss of generality, since any integer program whose objective is bounded below can be transformed into an equivalent problem with a non-negative objective by adding a sufficiently large constant to $f(x)$. This transformation shifts all objective values uniformly without affecting the set of optimal solutions. The integrality of the objective function, however, is fundamental to our framework, as it enables objective values to be represented using a finite number of bits. This requirement is satisfied by a broad class of combinatorial optimization problems, including those with integer edge weights or costs (e.g., TSP, VRP, CVRP, shortest path problems), unit-cost or counting objectives (e.g., set cover, vertex cover), and problems in which distances are derived from integer coordinates.

3.1 Binary Encoded Reformulation

We reformulate Problem (1) by expressing the objective function using its binary representation. Let IUB denote a known valid initial upper bound on the optimal objective value of Problem (1). Using this bound, we define

$$K = \lceil \log_2(\text{IUB} + 1) \rceil \,,$$

which represents the number of bits required to encode any integer value in the interval $[0, \text{IUB}]$. If such an upper bound is unavailable, one may instead choose a sufficiently large value for IUB. In practice, most integer programming applications require only a moderate number of bits to represent objective values. For example, requiring more than 30 bits would correspond to objective values on the order of billions, which is uncommon in many practical combinatorial optimization settings. With this in mind, any objective value z in this range can be written as

$$z = \sum_{k=0}^{K-1} 2^{K-1-k} \cdot b_k, \tag{2}$$

where b_0 is the most significant bit (MSB), b_{K-1} is the least significant bit (LSB), and each $b_k \in \{0, 1\}$. Introducing these binary variables, the binary encoded reformulation (BER) of Problem (1) is:

$$\begin{aligned} \min \quad & \sum_{k=0}^{K-1} 2^{K-1-k} \cdot b_k \\ \text{s.t.} \quad & f(x) = \sum_{k=0}^{K-1} 2^{K-1-k} \cdot b_k \\ & x \in \mathcal{X} \\ & b_k \in \{0, 1\} \quad \forall\, k \in \{0, 1, \ldots, K-1\}. \end{aligned} \tag{3}$$

This reformulation is equivalent to the original problem, meaning that any optimal solution (x^*, b^*) of Problem (3) yields an optimal solution x^* of Problem (1), with $z^* = f(x^*)$, and vice versa. In Problem (3), the key observation is that minimizing $f(x)$ is equivalent to finding the lexicographically smallest feasible bit vector. Specifically, a binary vector $(b_0, \ldots, b_{K-1})$ is lexicographically smaller than $(b'_0, \ldots, b'_{K-1})$ if, at the first index j where they differ, $b_j < b'_j$. For instance, consider two objective values $z = 5$ and $z' = 6$ with $K = 4$ bits:

$$5 = (0, 1, 0, 1), \quad 6 = (0, 1, 1, 0).$$

The first index where they differ is $j = 2$, where $b_2 = 0 < b'_2 = 1$. Hence $(0, 1, 0, 1) <_{\text{lex}} (0, 1, 1, 0)$, confirming $5 < 6$.

For any two objective values z, z' with binary representations $(b_0, \ldots, b_{K-1})$ and $(b'_0, \ldots, b'_{K-1})$,

$$z < z' \iff (b_0, b_1, \ldots, b_{K-1}) <_{\text{lex}} (b'_0, b'_1, \ldots, b'_{K-1}).$$

Since the positional weight 2^{K-1-k} decreases with k, setting $b_k = 0$ is always preferable to $b_k = 1$ for minimization. This leads to a natural greedy strategy: for each bit position k from the MSB to the LSB, if there exists a feasible solution with $b_k = 0$ given the current bit assignments, then the optimal solution must satisfy $b_k = 0$. If no such solution exists, $b_k = 1$ is forced. The optimal bit values can therefore be determined sequentially by solving a series of feasibility subproblems.

3.2 Bitwise Objective Domain Reduction Inspired by Binary Search

The binary representation in Equation (2) induces a natural mechanism for progressively narrowing the objective domain. Each bit in the binary encoding of the objective value corresponds to a specific sub-interval of the objective domain, and fixing that bit eliminates all values outside the corresponding sub-interval. Since bits are determined sequentially from the most significant to the least significant, the feasible objective range is reduced by approximately half at each step, yielding a nested sequence of increasingly tighter intervals. This process essentially mimics the classical binary search procedure at the level of the objective space. However, instead of operating directly on numerical intervals, it is implemented through binary decision variables induced by the encoding structure.

Partitioning via the MSB Let $\mathcal{Z} = \{0, 1, \ldots, 2^K - 1\}$. Consider the first decision: fixing the MSB b_0. Setting $b_0 = 0$ restricts the objective to the lower half of the domain, while setting $b_0 = 1$ restricts it to the upper half:

$$\mathcal{Z}^{(0)} := \{z \in \mathcal{Z} : b_0 = 0\} = [0,\ 2^{K-1} - 1] \cap \mathbb{Z},$$
$$\mathcal{Z}^{(1)} := \{z \in \mathcal{Z} : b_0 = 1\} = [2^{K-1},\ 2^K - 1] \cap \mathbb{Z}.$$

These two sets are disjoint and their union covers the entire domain. In a single step, half of all possible objective values are eliminated.

General bounds after m fixed bits This halving effect compounds with each subsequent bit. Suppose the first m bits have been fixed to values $v_0, v_1, \ldots, v_{m-1}$, collected in the set:

$$\mathcal{C} = \{(0, v_0), (1, v_1), \ldots, (m-1, v_{m-1})\}.$$

The contribution of these fixed bits to the objective value is the partial sum $\sum_{k=0}^{m-1} 2^{K-1-k}\, v_k$. The remaining $K - m$ unfixed bits can still take any combination of values in $\{0, 1\}$, so any objective value z consistent with $\mathcal{C}$ satisfies

$$z = \sum_{k=0}^{m-1} 2^{K-1-k}\, v_k + \sum_{k=m}^{K-1} 2^{K-1-k}\, b_k.$$

Consequently, the smallest such value is obtained by setting all remaining bits to zero, and the

largest by setting them all to one, giving

$$\mathrm{LB}(\mathcal{C}) = \sum_{k=0}^{m-1} 2^{K-1-k}\, v_k, \tag{4}$$

$$\mathrm{UB}(\mathcal{C}) = \sum_{k=0}^{m-1} 2^{K-1-k}\, v_k + 2^{K-m} - 1. \tag{5}$$

The interval $[\mathrm{LB}(\mathcal{C}), \mathrm{UB}(\mathcal{C})]$ contains exactly 2^{K-m} integer values. Compared to the original domain of 2^K values, this represents a cumulative reduction factor of 2^m: each additional bit fixed halves the remaining search space.

4 High-Level Methodology

In this section, we present a high-level description of the proposed framework for solving Problem (3), which is summarized in Figure 1. The process begins with a problem instance together with an Initial Lower Bound (ILB) and an Initial Upper Bound (IUB). The ILB can be obtained by solving a relaxation of the problem (e.g., the Assignment Problem relaxation for the TSP and the LP relaxation of the degree-constrained formulation for the CVRP), while the IUB can be obtained by generating a feasible solution using a heuristic (e.g., Nearest Neighbor or Guided Local Search for both the TSP and the CVRP). If such initial bounds are unavailable, the ILB can be set to $-\infty$ and the IUB can be set to $+\infty$. These inputs, together with a user-specified time limit T are passed to the proposed Bitwise Branch-and-Bound method in the solution framework. The framework operates on the binary-encoded reformulation (3) and invokes the proposed Bitwise Branch-and-Bound method to sequentially determine the bit values of z^*. The proposed method relies on a feasibility-check solver, which can be any suitable optimization solver, to determine the value of each bit. Once the proposed method terminates, either naturally or due to the imposed time limit, the framework produces one of two possible outcomes, depending on whether all bits are successfully resolved within the allotted time.

Outcome 1: Optimality If the framework successfully determines all bits $b_0, b_1, \ldots, b_{K-1}$, then the objective value $z^* = \sum_{k=0}^{K-1} 2^{K-1-k} b_k$ is fully specified, and an optimal solution (x^*, b^*) is obtained. In this case, no further computation is required.

Outcome 2: Partial progress with open subproblems If the time limit T is reached before all bits are resolved, the Bitwise Branch-and-Bound method terminates and returns (i) the best feasible solution $(\hat{x}, \hat{b})$, which provides a global upper bound $\mathrm{GUB} = f(\hat{x})$, and (ii) a set of open subproblems $\mathcal{O} = \{\mathcal{P}^1, \ldots, \mathcal{P}^M\}$. Each open subproblem $\mathcal{P}_j$ corresponds to an unexplored branch of the bit-fixing process, defined by a set of fixed bit assignments

$$\mathcal{C}^j = \{(k, v_k) : b_k = v_k \text{ for some but not all } k\}.$$

The objective subspace associated with $\mathcal{P}_j$ is given by

$$\mathcal{Z}^j = \left\{ z \in \mathbb{Z}_{\geq 0} : z = \sum_{k=0}^{K-1} 2^{K-1-k} b_k,\ b_k = v_k \ \forall\, (k, v_k) \in \mathcal{C}^j,\ b_k \in \{0,1\} \text{ otherwise} \right\}.$$

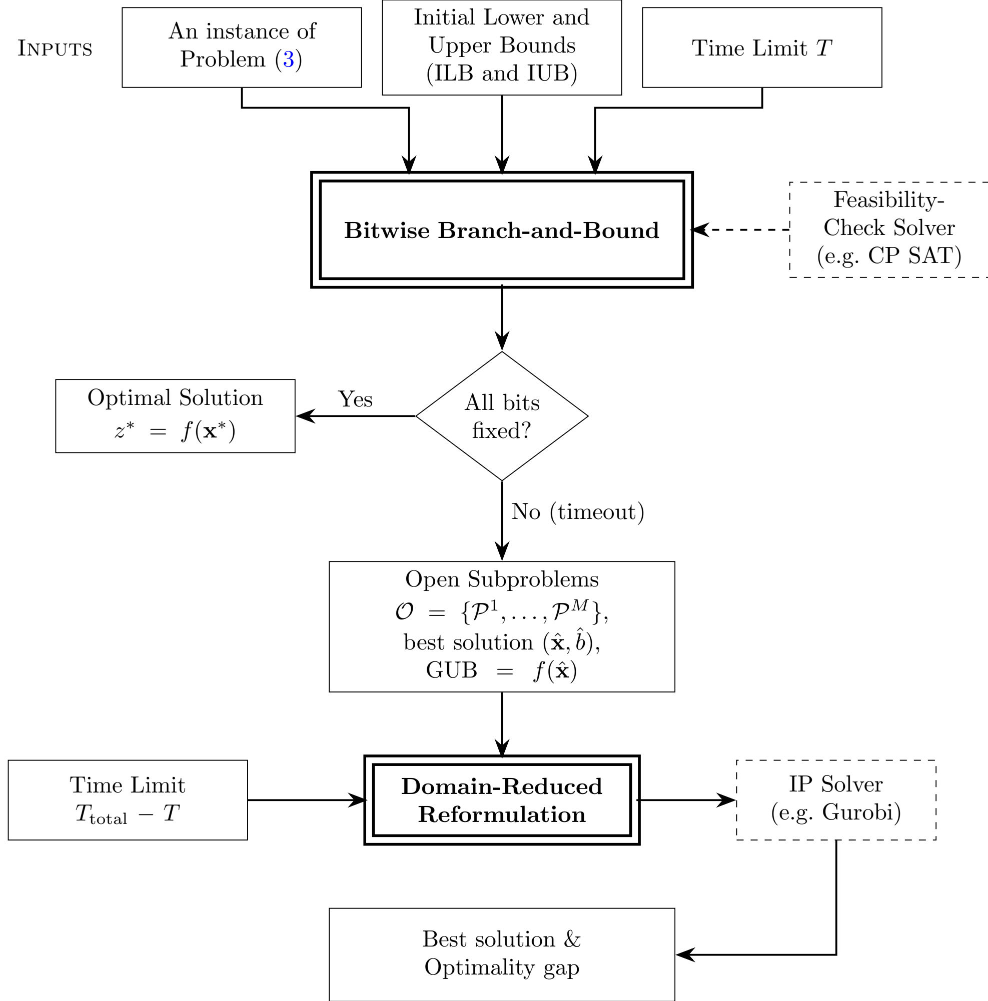

Figure 1: High-level overview of the proposed solution framework

Throughout this paper, we use a symmetric TSP instance with 52 nodes as a running example. Suppose an initial upper bound IUB = 8980 is obtained from a heuristic solution and a an initial lower bound ILB = 7390 from a relaxation, giving $K = \lceil \log_2(8981) \rceil = 14$ bits to encode the objective value. Suppose that when the time limit is reached, the Bitwise Branch-and-Bound method has explored part of the branch and bound tree up to depth six, and two subproblems remain open. The open subproblems arise because the solver was unable to resolve the MSB in one branch within the allotted time, leaving both values of that bit to be explored further. The details of how bit assignments are determined are described in Section 5. At termination, the two open

subproblems are defined by:

$$\mathcal{C}^1 = \{(0,0),(1,1),(2,1),(3,1),(4,0),(5,1)\}, \quad \mathcal{C}^2 = \{(0,1),(1,0),(2,0),(3,0),(4,0),(5,1)\},$$

with corresponding objective subspaces (derived from Equations (4)–(5)):

$$\mathcal{Z}^1 = [7424,\ 7679] \cap \mathbb{Z}, \quad \mathcal{Z}^2 = [8448,\ 8703] \cap \mathbb{Z}.$$

Any improving solution with $z^* < 8980$ must lie in $\mathcal{Z}^1 \cup \mathcal{Z}^2$, and no feasible region is lost.

The framework derives its value from two structural properties of these open subproblems. First, if an improving solution exists (i.e., $z^* <$ GUB), it is guaranteed to lie in exactly one of the objective subspaces $\mathcal{Z}^1, \ldots, \mathcal{Z}^M$. No region of the search space that could contain an improving solution is lost, regardless of how early the time limit interrupted the process. Second, they are mutually independent. Each subproblem $\mathcal{P}_j$ defines a self-contained optimization problem over a restricted objective interval $[\mathrm{LB}(\mathcal{C}^j),\ \mathrm{UB}(\mathcal{C}^j)]$, derived from Equations (4)–(5). In the remainder of this study, for convenience and whenever appropriate, we represent an open subproblem $\mathcal{P}_j$ using the tuple

$$\mathcal{P}^j := \left(\mathcal{C}^j,\ \mathrm{LB}(\mathcal{C}^j),\ \mathrm{UB}(\mathcal{C}^j)\right).$$

It is worth noting that the remaining objective intervals can be handled in two different ways because they are disjoint. First, they may be treated collectively within a unified formulation that leverages the embedded parallelization capabilities of modern IP solvers. Alternatively, each interval can be treated as an independent subproblem and solved separately in parallel. Investigating such distributed or parallel strategies is an interesting direction for future research. In this study, however, we adopt the unified approach and use the open subproblems to construct a single reformulation, referred to as the Domain-Reduced Reformulation, which is then passed to the IP solver for further exploration.

In summary, the two highlighted components in Figure 1 constitute the core building blocks of the proposed methodology and are described in detail in the following sections. In our implementation, Google's CP-SAT solver is employed as the feasibility-check engine within the Bitwise Branch-and-Bound method (Section 5.1). When the Bitwise Branch-and-Bound method does not generate an optimal solution within the prescribed time limit T, the remaining open subproblems are used to generate the proposed Domain-Reduced Reformulation, which is then passed to an IP solver (Gurobi in our implementation) together with a warm-start solution (Section 5.2). Users may additionally specify an overall time limit T_{total} for the complete framework, where $T_{\text{total}} > T$. In this setting, the first T units of time are allocated to the Bitwise Branch-and-Bound phase, while the remaining time budget, namely $T_{\text{total}} - T$, is automatically allocated to the IP solver for solving the Domain-Reduced Reformulation and reporting the best feasible solution together with its associated optimality gap.

5 Detailed Description

In this section, we provide a detailed description of the two main components of the proposed solution framework: the Bitwise Branch-and-Bound method and the Domain-Reduced Reformulation. An illustrative example based on the running TSP instance is provided in Appendix 8 for readers seeking additional insight into the mechanics of these components.

5.1 Bitwise Branch-and-Bound Method

The proposed Bitwise Branch-and-Bound method is shown in Algorithm 1. The proposed method maintains a priority queue of nodes, denoted by Q, where each node represents a subproblem consisting of a partial bit assignment together with its corresponding lower and upper bounds, as described in Section 4. Specifically, each node is represented as $\mathcal{P} = (\mathcal{C},\ \mathrm{LB}(\mathcal{C}),\ \mathrm{UB}(\mathcal{C}))$, where $\mathcal{C}$ denotes the set of fixed bit assignments. The nodes are maintained in nondecreasing order of their corresponding lower bounds, meaning that the proposed method follows a best-bound search strategy. At the root node (Line 5), $\mathcal{C} = \emptyset$. Consequently, $\mathrm{LB}(\mathcal{C}) = 0$ and $\mathrm{UB}(\mathcal{C}) = 2^K - 1$ following Equations (4)–(5).

The proposed method operates under two time limits: a per-bit time limit T_b, which governs individual feasibility queries, and an overall time limit T, which bounds the entire computation. The value of T_b is typically chosen to be relatively small (e.g., a few seconds) and can be tuned for each problem class under consideration (Line 3). In practice, users may calibrate this parameter by experimenting on a small number of representative instances. In the computational study, we report the selected values of T_b for both the TSP and the CVRP instances. The search proceeds by sequentially fixing bit variables from the MSB to the LSB. Thus, when a node is extracted from the priority queue, the algorithm first identifies the most significant unexplored bit, denoted by k, and attempts to determine its value.

Since we consider minimization problems, the preferred assignment is always $b_k = 0$, as established in Section 3.1. Therefore, the proposed method first tests whether fixing $b_k = 0$ yields a feasible solution. Depending on the outcome, the algorithm determines whether the complementary assignment $b_k = 1$ must also be explored. The outcome of this exploration may lead to one of three cases: (i) no child nodes are generated due to infeasibility or pruning, (ii) exactly one child node is generated when the value of bit k can be determined uniquely, or (iii) two child nodes are generated corresponding to both possible assignments of bit k. For each node in the branch-and-bound tree, feasibility is checked using a feasibility-check solver (Google's CP-SAT solver in our implementation), which searches for a feasible solution to Problem (3) subject to additional constraints enforcing the current partial bit assignment.

As discussed in Section 4, an ILB and an IUB are provided to the proposed Bitwise Branch-and-Bound method to strengthen pruning throughout the search process. In particular, the IUB is used to initialize the Global Upper Bound (GUB), corresponding to the best known feasible solution (Line 2). Two straightforward pruning mechanisms (among others) are employed. First, whenever

the upper bound of a node satisfies $\text{UB}(\mathcal{C}) < \text{ILB}$, the node can be fathomed immediately, since its entire objective range lies below the known valid lower bound and is therefore infeasible (Lines 9-11). Second, whenever $\text{GUB} \leq \text{LB}(\mathcal{C})$, all feasible solutions associated with that node must have objective values no better than the current best known solution. In this case, the node is said to be *dominated* by the GUB and can also be fathomed without further exploration (Lines 24-25, 29-30, 40-41, 45-46). During the search process, the GUB is updated whenever a better feasible solution is identified (Lines 18-19, 34-35). At termination (Lines 50-52), the proposed method reports the best known solution $(\hat{x}, \hat{b})$, the corresponding objective value $\text{GUB} = f(\hat{x})$, and the set of remaining non-dominated open nodes, if any. Next, we describe the node exploration procedure, branching strategy, and additional pruning mechanisms in detail.

When exploring a node, the algorithm first calls the operation $\textsc{CheckFeasibility}(\mathcal{C}, k, 0, T_b)$, which tests whether the current node remains feasible after fixing bit $b_k = 0$, subject to the per-bit time limit T_b (Line 16). The outcome of this operation is represented by the tuple $(s^0, (x^0, b^0), \text{opt}^0)$, where s^0 denotes the solver status, which can be *feasible*, *infeasible*, or *timeout*. A timeout status indicates that the feasibility-check solver was unable to resolve the query within the allotted time limit. If the status is feasible, then (x^0, b^0) is returned by the solver and may improve the current best known feasible solution. Additionally, if the solver proves optimality for the considered assignment, then opt^0 is set to TRUE. In this case, the node can be fathomed immediately, since no further search is required within that branch (Lines 20-22). If the assignment is feasible but opt^0 is FALSE, then a single child node corresponding to fixing $b_k = 0$ is generated (Lines 23-25):

$$\mathcal{P}^0 = \left(\mathcal{C}^0,\ \text{LB}(\mathcal{C}^0),\ \text{UB}(\mathcal{C}^0)\right),$$

where

$$\mathcal{C}^0 = \mathcal{C} \cup \{(k, 0)\}.$$

If s^0 is infeasible, then the branch corresponding to $b_k = 0$ can be discarded, and the algorithm proceeds to test the complementary assignment $b_k = 1$ (Line 32) by calling $\textsc{CheckFeasibility}(\mathcal{C}, k, 1, T_b)$. The outcome of this call is represented by the tuple $(s^1, (x^1, b^1), \text{opt}^1)$. If s^1 is feasible, then (x^1, b^1) is returned by the solver and may improve the best known feasible solution. Furthermore, if s^1 is feasible and opt^1 is TRUE, then no additional child node needs to be created, since the corresponding branch has been solved to optimality (Lines 36-38). Similarly, if s^1 is infeasible, then no child node is generated because both assignments $b_k = 0$ and $b_k = 1$ have been proven infeasible, implying that the entire node is infeasible. Finally, if s^1 results in either a timeout or a feasible solution with $\text{opt}^1 = \text{FALSE}$ (Lines 39-41, 44-46), then the child node corresponding to fixing $b_k = 1$ is generated:

$$\mathcal{P}^1 = \left(\mathcal{C}^1,\ \text{LB}(\mathcal{C}^1),\ \text{UB}(\mathcal{C}^1)\right),$$

where

$$\mathcal{C}^1 = \mathcal{C} \cup \{(k, 1)\}.$$

The remaining scenario occurs when s^0 is timeout. In this case, the child node corresponding to $\mathcal{C}^0$ must be created because feasibility could not be resolved within the allotted time (Lines 27-31). The complementary assignment $b_k = 1$ is then tested (Line 32) through CHECKFEASIBILITY$(\mathcal{C}, k, 1, T_b)$, and the child node corresponding to $\mathcal{C}^1$ is generated only if s^1 results in either a timeout or a feasible solution with opt^1 = FALSE (Lines 39-41, 44-46).

Algorithm 1 Bitwise Branch-and-Bound

Require: Problem instance, time limit T, ILB, and IUB
Ensure: Best solution $(\hat{x}, \hat{b})$, global upper bound GUB $= f(\hat{x})$, open nodes $\mathcal{O}$

1: GUB $\leftarrow$ IUB ▷ *Initialize the global upper bound*
2: $(\hat{x}, \hat{b}) \leftarrow$ Solution corresponding to IUB ▷ *Initialize the best known solution*
3: $T_b \leftarrow$ User-defined value ▷ *Time limit for checking each feasibility subproblem*
4: $K \leftarrow \lceil \log_2(\text{IUB} + 1) \rceil$ ▷ *Number of bits needed to encode objective range $[0, IUB]$*
5: PUSH$\Big(\mathcal{Q}, \big(C = \emptyset, \text{LB}(C), \text{UB}(C)\big)\Big)$ ▷ *Initialize the priority queue of open nodes, ordered by nodal LB*
6: **while** $\mathcal{Q} \neq \emptyset$ **and** elapsed time $\leq T$ **do**
7: $(\mathcal{C}, \text{LB}, \text{UB}) \leftarrow$ POPMIN$(\mathcal{Q})$ ▷ *Pop out the node with the smallest LB*
8: $k \leftarrow |C|$ ▷ *The most significant unexplored bit index*
9: **if** UB $<$ ILB **then**
10: **Go to next iteration** ▷ *Fathom: Infeasible range/domain*
11: **end if**
12: **if** $k = K$ **then** ▷ *All bits are fixed but still an undecided node due to T_b*
13: $\mathcal{O} \leftarrow \mathcal{O} \cup \{(\mathcal{C}, \text{LB}(\mathcal{C}), \text{UB}(\mathcal{C}))\}$ ▷ *Add it to the list of open nodes*
14: **Go to next iteration** ▷ *Fathom: no further exploration can be done*
15: **end if**
16: $(s^0, (x^0, b^0), \text{opt}^0) \leftarrow$ CHECKFEASIBILITY$(\mathcal{C}, k, 0, T_b)$ ▷ *Check if bit k can be set to 0*
17: **if** s^0 = FEASIBLE **then** ▷ *If the status is feasible, no need to check for $b_k = 1$*
18: **if** $f(x^0) <$ GUB **then** $(\text{GUB}, (\hat{x}, \hat{b})) \leftarrow (f(x^0), (x^0, b^0))$ ▷ *Update GUB and the best known solution*
19: **end if**
20: **if** opt^0= True **then** ▷ *If (x^0, b^0) is proved optimal when $b_k = 0$*
21: **Go to next iteration** ▷ *Fathom: No further exploration needed*
22: **end if**
23: $\mathcal{C}^0 \leftarrow \mathcal{C} \cup \{(k, 0)\}$ ▷ *Fix $b_k = 0$ to create a child node*
24: **if** $\text{LB}(\mathcal{C}^0) < GUB$ **then** PUSH$(\mathcal{Q}, (\mathcal{C}^0, \text{LB}(\mathcal{C}^0), \text{UB}(\mathcal{C}^0)))$ ▷ *Add the child node if not dominated*
25: **end if**
26: **else** ▷ *$s^0 \in \{INFEASIBLE, TIMEOUT\}$, explore for $b_k = 1$*
27: **if** s_0 = TIMEOUT **then** ▷ *$b_k = 0$ is undecided; may require further exploration*
28: $\mathcal{C}^0 \leftarrow \mathcal{C} \cup \{(k, 0)\}$ ▷ *Fix $b_k = 0$ to create a child node*
29: **if** $\text{LB}(\mathcal{C}^0) <$ GUB **then** PUSH$(\mathcal{Q}, (\mathcal{C}^0, LB(\mathcal{C}^0), \text{UB}(\mathcal{C}^0)))$ ▷ *Add the child node if not dominated*
30: **end if**
31: **end if**
32: $(s^1, (x^1, b^1), \text{opt}^1) \leftarrow$ CHECKFEASIBILITY$(\mathcal{C}, k, 1, T_b)$ ▷ *Check if bit k can be set to 0*
33: **if** s_1 = FEASIBLE **then**
34: **if** $f(x^1) <$ GUB **then** $(\text{GUB}, (\hat{x}, \hat{b})) \leftarrow (f(x^1), (x^1, b^1))$ ▷ *Update GUB and the best known solution*
35: **end if**
36: **if** opt_1 = True **then** ▷ *If (x^1, b^1) is proved optimal when $b_k = 1$*
37: **Go to next iteration** ▷ *Fathom: No further exploration needed*
38: **else**
39: $\mathcal{C}^1 \leftarrow \mathcal{C} \cup \{(k, 1)\}$ ▷ *Fix $b_k = 1$ to create a child node*
40: **if** $\text{LB}(\mathcal{C}_1) <$ GUB **then** PUSH$(\mathcal{Q}, (\mathcal{C}^1, \text{LB}(\mathcal{C}^1), \text{UB}(\mathcal{C}^1)))$ ▷ *Add the child node if not dominated*
41: **end if**
42: **end if**
43: **else if** s_1 = TIMEOUT **then** ▷ *$b_k = 1$ is undecided; may require further exploration*
44: $\mathcal{C}^1 \leftarrow \mathcal{C} \cup \{(k, 1)\}$ ▷ *Fix $b_k = 1$ to create a child node*
45: **if** $\text{LB}(\mathcal{C}_1) <$ GUB **then** PUSH$(\mathcal{Q}, (\mathcal{C}^1, \text{LB}(\mathcal{C}^1), \text{UB}(\mathcal{C}^1)))$ ▷ *Add the child node if not dominated*
46: **end if**
47: **end if**
48: **end if**
49: **end while**
50: $\mathcal{O} \leftarrow \mathcal{O} \cup \{(\mathcal{C}, \text{LB}(\mathcal{C}), \text{UB}(\mathcal{C})) \in \mathcal{Q}\}$ ▷ *Collect all remaining open nodes*
51: $\mathcal{O} \leftarrow \mathcal{O} \setminus \{(\mathcal{C}, \text{LB}(\mathcal{C}), \text{UB}(\mathcal{C}) \in \mathcal{O} : \text{LB}(\mathcal{C}) \geq \text{GUB}\}$ ▷ *Remove dominated open nodes*
52: **return** $((\hat{x}, \hat{b}), \text{GUB}, \mathcal{O})$

As a final note, we highlight that if an open node is encountered in the queue for which all bits have already been fixed, then by construction the node must correspond to an unresolved feasibility query (i.e., the feasibility-check solver was unable to determine feasibility or infeasibility within the prescribed time limit). In this case, the node is not explored further. Instead, it is directly added to the list of remaining open nodes (Lines 12-15) that will be reported when the algorithm terminates. Additionally, a simple enhancement to the proposed algorithm is to skip certain feasibility checks whenever a previously identified feasible solution is already known to satisfy the current partial bit assignment under consideration.

5.2 Domain-Reduced Reformulation

When the proposed Bitwise Branch-and-Bound method terminates, if $\mathcal{Q} = \emptyset$, then the best found solution $(\hat{x}, \hat{b})$ is optimal. Otherwise, the remaining open subproblems, together with the best incumbent solution, are used to construct a reformulation of the original problem that restricts the search to the disjoint objective ranges remaining after the branch-and-bound process. Let $\{\mathcal{P}^1, \ldots, \mathcal{P}^M\}$ be the set of open subproblems at termination. Since we also wish to provide the best known feasible solution, i.e., $(\hat{x}, \hat{b})$, to the IP solver as a warm-start solution, we treat its corresponding bit assignment as an additional open subproblem in order to guarantee that the incumbent solution remains feasible in the proposed Domain-Reduced Reformulation. Specifically, we define

$$\mathcal{P}^{M+1} := \left(\mathcal{C}^{M+1}, \mathrm{LB}(\mathcal{C}^{M+1}), \mathrm{UB}(\mathcal{C}^{M+1})\right),$$

where

$$\mathcal{C}^{M+1} = \{(k, \hat{b}_k) : k = 0, \ldots, K-1\}.$$

To construct the proposed reformulation, for each subproblem $\mathcal{P}^j$, where $j \in \{1, \ldots, M+1\}$, we define lower and upper bounds for each bit variable. If a bit is fixed in $\mathcal{C}^j$, then both bounds are equal to the fixed value. Otherwise, the lower bound is set to 0 and the upper bound is set to 1. Specifically, for each subproblem $\mathcal{P}^j$, we define

$$l_k^j = \begin{cases} v & \text{if } (k, v) \in \mathcal{C}^j, \\ 0 & \text{if bit } k \text{ is not fixed in } \mathcal{P}^j, \end{cases} \qquad u_k^j = \begin{cases} v & \text{if } (k, v) \in \mathcal{C}^j, \\ 1 & \text{if bit } k \text{ is not fixed in } \mathcal{P}^j. \end{cases} \tag{6}$$

Next, we introduce a binary selection variable y_j for each $j = 1, \ldots, M+1$ to activate exactly one subproblem. Using this notation, the proposed Domain-Reduced Reformulation is given by

$$
\begin{aligned}
\min \quad & \sum_{k=0}^{K-1} 2^{K-1-k}\, b_k \\
\text{s.t.} \quad & f(x) = \sum_{k=0}^{K-1} 2^{K-1-k}\, b_k \\
& x \in \mathcal{X} \\
& f(x) \geq \text{ILB} \\
& \sum_{j=1}^{M+1} y_j = 1 \\
& b_k \geq \sum_{j=1}^{M+1} l_k^j\, y_j \qquad \forall\, k \in \{0, \ldots, K-1\} \\
& b_k \leq \sum_{j=1}^{M+1} u_k^j\, y_j \qquad \forall\, k \in \{0, \ldots, K-1\} \\
& b_k \in \{0,1\}, \quad y_j \in \{0,1\} \qquad \forall\, k \in \{0, \ldots, K-1\},\ \forall\, j \in \{1, \ldots, M+1\}.
\end{aligned}
\tag{7}
$$

In the proposed reformulation, the additional constraints ensure that exactly one subproblem is activated. When a subproblem $\mathcal{P}^j$ is selected (i.e., $y_j = 1$), the bit variables are forced to satisfy the corresponding partial bit assignment $\mathcal{C}^j$. Consequently, the feasible search space is automatically restricted to the objective interval $\big(\text{LB}(\mathcal{C}^j), \text{UB}(\mathcal{C}^j)\big)$. As discussed in Section 4, a time limit of $T_{\text{total}} - T$ is imposed on the IP solver when solving the Domain-Reduced Reformulation. At the end of this time limit, the solver either returns the warm-start solution provided to it, potentially with an improved dual bound and a smaller optimality gap, or identifies a better feasible solution. As a final note, we highlight that, from an implementation perspective, the variables

$$y_1, \ldots, y_{M+1}$$

may also be declared as a Special Ordered Set of Type I (SOS-I), allowing the IP solver to potentially exploit additional branching and presolve mechanisms.

6 Computational Experiments

In this section, we evaluate the performance of the proposed framework on two challenging problem classes: TSP and CVRP. These problems align well with the main premise of this work, namely improving solver performance on challenging integer programs via objective-domain reduction. All experiments were conducted on Ubuntu 24.04.1 LTS running on a workstation equipped with two Intel Xeon Gold 6230R CPUs @ 2.10 GHz (52 physical cores / 104 threads), 187 GB RAM, and an NVIDIA RTX A6000 GPU with 48 GB VRAM. Google OR-Tools CP-SAT was used as the feasibility-check solver at each node of the Bitwise Branch-and-Bound method, while Gurobi

Optimizer 12.0.3 (build v12.0.3rc0, linux64) was employed as the IP solver.

It is important to note that both TSP and CVRP are challenging problems for IP solvers due to the large number of subtour elimination constraints required in their formulations. Therefore, in all experiments involving the IP solver, we implemented lazy constraint callbacks to dynamically handle subtours (details are provided in the subsequent subsections), allowing the solution of large-scale instances. Otherwise, only relatively small instances could be solved effectively, which would not lead to meaningful computational results. Throughout this section, we use the following acronyms to refer to the three solution approaches under comparison:

- **BBB**: the Bitwise Branch-and-Bound method, i.e., Algorithm 1, applied as a standalone procedure. For relatively small-sized instances, BBB can solve the problem to proven optimality independently.
- **BBB+DRR**: the complete proposed framework, which first applies the BBB method. If optimality is not achieved within the allotted time and open nodes remain, the corresponding Domain-Reduced Reformulation (DRR) is generated and passed to the IP solver (with lazy constraint callbacks) for further optimization.
- **IP**: the direct application of the IP solver (with lazy constraint callbacks) to the original formulation of the problem, without employing the proposed framework.

Observe that the only difference between the IP and BBB+DRR settings is the formulation provided to the IP solver. In the IP setting, the solver is applied directly to the original formulation, whereas in BBB+DRR, the same IP solver, with the same lazy constraint callback strategy, is applied to the proposed Domain-Reduced Reformulation. Thus, the underlying IP solution strategy remains identical across both settings, which is essential for ensuring meaningful and fair comparisons. Extending the proposed framework to incorporate other advanced optimization techniques, particularly decomposition methods that are effective for certain problem classes, requires further investigation and is left for future research.

Another important aspect of ensuring a fair comparison concerns the use of the initial bounds. In the proposed framework, we employ ILB and IUB. To maintain consistency in the IP baseline, the same information is also provided to the IP solver: the constraint $f(x) \geq \text{ILB}$ is added to the model, and the solution corresponding to IUB is used as a warm start. This ensures that all approaches benefit from identical initial information, thereby avoiding any bias in favor of the proposed framework.

The total time limit T_{total} is set to 3 hours per instance for TSP, 6 hours for CVRP A-instances, and 24 hours for the larger CVRP M-instances, reflecting the increasing difficulty of these problem classes. This overall time limit is applied consistently across all three approaches: BBB, BBB+DRR, and IP. In practice, however, BBB alone often solves the smaller instances much faster than the imposed time limit. As mentioned earlier (see Figure 1), the proposed framework uses two main time parameters. The first is T, which bounds the wall-clock time spent in the

BBB phase, while the second is the remaining budget, $T_{\text{total}} - T$, which bounds the time allocated to solving the DRR formulation. Additionally, during the feasibility checks performed within the BBB phase, a per-query time limit T_b is imposed. Both T and T_b must be tuned for each problem class, since the performance of the proposed framework is sensitive to these parameter choices. This tuning can be performed by evaluating several candidate values on a subset of representative instances and selecting the configuration that yields the best overall performance. In the subsequent subsections, we report the values selected for the TSP and CVRP experiments.

6.1 Traveling Salesman Problem

We evaluate our approach on 83 symmetric instances from the TSPLIB benchmark library (Reinelt 1991) with fewer than 1,500 nodes. For each instance, the initial upper bound is estimated using the nearest neighbor heuristic and the OR-Tools Routing Library with Guided Local Search. The lower bound is obtained from the linear assignment problem relaxation, solved using the Hungarian method. At each node of the BBB method, we solve a feasibility check using the CP-SAT solver from OR-Tools, formulating the problem with the `AddCircuit` global constraint. This constraint enforces a single Hamiltonian cycle over the full node set without requiring explicit subtour elimination constraints. Warm-start hints from the best-known tour are passed to CP-SAT at every feasibility check to accelerate the search. The bit-level time limit (T_b) for each feasibility check is set to $\ln(n)$ seconds, and the overall BBB time limit (T) for each instance is set to $100 \ln(n)$ seconds, where n is the number of nodes in the TSP instance. The TSP formulation uses degree constraints requiring each node to have exactly two incident edges, and subtours are eliminated dynamically through lazy constraint callbacks. Whenever the solver finds a candidate solution that contains multiple disconnected cycles instead of a single tour, a constraint of the form $\sum_{\{i,j\} \subseteq S} x_{ij} \leq |S| - 1$ is added for each disconnected cycle S, which forces the solver to reject that solution and continue searching. Based on problem difficulty, we organize the results into three classes:

Class I: Instances solved to optimality by BBB alone, without requiring the DRR phase. These are relatively easy instances, as they can also be solved by the IP baseline within a fraction of a second. Therefore, they are not particularly informative for performance comparisons. However, they demonstrate that BBB alone is capable of solving such instances to optimality, with solutions matching the known optima from TSPLIB.

Class II: Medium-difficulty instances for which BBB alone is not sufficient, while the IP baseline typically requires between a few seconds and up to 1,000 seconds to reach optimality. These instances are not primarily used to compare the full BBB+DRR framework against the baseline; instead, they are used to study the impact of the initial bounds. In particular, different strategies for computing the IUB can significantly affect performance. To show the impact of instance difficulty within this category, we further divide Class II into Class II.I (instances for which the IP baseline solves in under 20 seconds) and Class II.II (instances for which the IP baseline requires between 20 and 1,000 seconds).

Table 1: Validation of BBB on Class I TSP instances

Instance	Time (s)	Objective	Optimal
burma14	0.04	3,323	✓
ulysses16	0.05	6,859	✓
gr21	0.07	2,707	✓
ulysses22	0.10	7,013	✓
gr17	0.12	2,085	✓
gr24	0.13	1,272	✓
fri26	0.18	937	✓
bays29	0.21	2,020	✓
bayg29	0.36	1,610	✓
dantzig42	0.56	699	✓
berlin52	0.86	7,542	✓
swiss42	1.40	1,273	✓
hk48	1.61	11,461	✓
att48	2.56	10,628	✓
brazil58	2.92	25,395	✓
gr48	3.77	5,046	✓
st70	13.46	675	✓
lin105	58.36	14,379	✓
rat99	77.76	1,211	✓
rd100	84.04	7,910	✓
eil101	85.18	629	✓

Class III: The most challenging instances, for which the IP baseline requires more than 1,000 seconds, including cases not solved to optimality within the 3-hour time limit. These instances best reflect the main motivation of this work, namely improving solver performance via objective-domain reduction, and are therefore used for the primary comparison against the IP baseline.

Table 1 summarizes the performance of BBB on 21 instances identified in Class I. We observe that in all cases, BBB successfully identifies the optimal solution, with computation times ranging from 0.04 seconds for the smallest instance (burma14) to under 90 seconds for instances with approximately 100 nodes. These results validate the correctness of the algorithm, as it consistently recovers the known optimal solutions. In Table 2, we focus on 51 instances classified as Class II and analyze the impact of the IUB on the performance of BBB+DRR on such instances. We compare two widely used and straightforward strategies for computing IUB: a Nearest Neighbor (NN) heuristic and Google OR-Tools. The latter typically produces tighter upper bounds but requires additional computational effort. We impose a time limit of 1 second for Google OR-Tools on Class II.I instances and 30 seconds on Class II.II instances. The results confirm that tighter initial bounds lead to faster BBB+DRR solution times. For Class II.I instances, replacing the NN heuristic with OR-Tools yields an average time improvement of 6.56%, with individual instances such as `br180` achieving a reduction of 88.93%. For Class II.II instances, the effect is more pronounced, with an average improvement of 21.76% and reductions of 75% and 68% on `gr431` and `fl417`, respectively. This sensitivity to the quality of the initial bound is expected, as tighter bounds reduce the number of bits that must be resolved, resulting in fewer feasibility checks and a shallower search tree.

In Table 3, we focus on 21 instances classified as Class III and compare the performance of

Table 2: Impact of initial upper bound on BBB+DRR performance for TSP instances

Class II.I: IP < 20s						**Class II.II: IP 20s–1000s**					
Instance	Initial Upper Bound		Time (s)		Improv.	Instance	Initial Upper Bound		Time (s)		Improv.
	NN	OR	NN	OR	(%)		NN	OR	NN	OR	(%)
brg180	12,360	1,960	521.10	**57.67**	88.93	rat195	2,752	2,368	687.40	**543.75**	20.89
pr107	46,680	44,991	480.83	**475.24**	1.16	kroA200	35,859	29,590	544.83	**541.47**	0.62
kroC100	26,227	21,817	466.11	**464.58**	0.33	lin318	54,019	43,312	707.97	**649.01**	8.33
kroD100	26,947	22,439	472.69	**228.38**	51.68	tsp225	5,030	4,024	612.81	**568.72**	7.19
kroA100	27,807	21,389	476.22	**467.45**	1.84	gr229	162,430	139,411	595.38	**564.86**	5.12
gr96	70,916	56,880	469.61	**462.81**	1.45	pr264	58,023	51,495	595.89	**593.27**	0.44
kroB100	29,158	22,199	480.35	**468.99**	2.37	pcb442	61,979	51,727	**878.28**	991.67	−12.91
kroE100	27,460	22,465	469.72	**469.58**	0.03	fl417	15,013	12,141	3,132.02	**1,024.46**	67.30
ch130	7,579	6,308	**491.75**	491.79	−0.01	pr299	59,890	50,304	**670.35**	865.85	−29.17
pr136	120,769	102,225	506.87	**501.01**	1.16	rd400	19,183	15,654	958.13	**772.09**	19.42
gr120	9,351	7,075	487.35	**479.06**	1.70	ali535	253,127	216,341	1,160.28	**1,070.73**	7.72
gr137	93,912	71,067	502.93	**495.77**	1.42	att532	35,516	28,655	2,884.24	**1,946.08**	32.54
ch150	8,191	6,681	514.11	**507.43**	1.30	gr431	210,069	177,440	3,748.69	**941.83**	74.88
u159	54,675	43,515	**508.82**	510.69	−0.37	si535	50,144	48,724	3,069.37	**1,078.79**	64.85
bier127	135,737	121,304	495.13	**487.82**	1.48	u574	50,459	38,908	2,454.99	**1,000.70**	59.24
pr76	153,462	109,996	500.30	**488.62**	2.33						
kroB150	34,499	26,671	526.17	**507.17**	3.61						
pr124	69,297	60,413	**492.19**	493.69	−0.30						
pr144	61,652	58,869	**502.93**	504.63	−0.34						
kroA150	33,633	27,061	517.47	**508.40**	1.75						
kroB200	36,980	30,773	545.80	**533.68**	2.22						
pr152	85,699	76,497	516.88	**507.77**	1.76						
si175	22,263	21,537	543.60	**528.30**	2.82						
a280	3,157	2,666	578.26	**574.93**	0.58						
gr202	49,336	42,563	**537.34**	540.37	−0.56						
d198	18,240	16,007	563.35	**545.62**	3.15						
				Avg. improvement:	**6.56**					*Avg. improvement:*	**21.76**

Notes. NN: Nearest Neighbor heuristic. OR: Google OR-Tools (1s solve time for Class II.I, 30s for Class II.II). Bold indicates faster solve time. Improvement: (NN − OR)/NN × 100; positive values favor OR-Tools.

Table 3: Performance comparison on Class III TSP instances (IP time > 1000s)

	Instance	BBB+DRR Time (s)	IP Time (s)	BBB+DRR Gap (%)	IP Gap (%)	Time Improv. (%)	Gap Improv. (%)
Solved by IP	pr226	**582.72**	1,159.84	0.00	0.00	**49.77**	—
	rat783	**960.66**	1,301.37	0.00	0.00	**26.18**	—
	d493	**1,602.07**	1,707.12	0.00	0.00	**6.15**	—
	gr666	**2,120.98**	2,133.61	0.00	0.00	**0.59**	—
	rat575	**1,891.14**	2,627.76	0.00	0.00	**28.04**	—
	pa561	3,502.91	**2,744.51**	0.00	0.00	−27.61	—
	d657	**2,260.72**	10,282.64	0.00	0.01	**78.01**	—
	u724	**6,098.85**	10,341.85	0.00	0.01	**41.03**	—
	pr1002	**2,975.70**	3,012.81	0.00	0.00	**1.23**	—
					Avg.:	**22.60**	—
IP timeout	p654	**7,699.29**	timeout	**0.00**	1.04	—	**100.00**
	ts225	timeout	timeout	1.56	**1.55**	—	−0.65
	u1060	timeout	timeout	**6.74**	11.00	—	**38.73**
	dsj1000	timeout	timeout	**8.33**	8.81	—	**5.45**
	rl1304	timeout	timeout	11.03	**9.88**	—	−11.64
	fl1400	timeout	timeout	**9.79**	24.68	—	**60.33**
	vm1084	timeout	timeout	**0.08**	8.08	—	**99.01**
	rl1323	timeout	timeout	**7.64**	12.32	—	**37.99**
	u1432	timeout	timeout	**9.05**	15.29	—	**40.81**
	pcb1173	timeout	timeout	**0.06**	9.71	—	**99.38**
	d1291	timeout	timeout	**9.79**	13.33	—	**26.56**
	nrw1379	timeout	timeout	**0.02**	8.80	—	**99.77**
					Avg.:	—	**49.64**

Notes. Bold indicates better performance by BBB+DRR. Time improvement: (IP − BBB+DRR)/IP × 100. Gap improvement: (IP gap − BBB+DRR gap)/IP gap × 100. Positive values favor BBB+DRR.

BBB+DRR against the IP baseline on these instances. The results are divided into two groups. In the first group, both methods reach optimality, but BBB+DRR is faster on 8 out of 9 instances, with an average time improvement of 22.60%. The largest gains are observed on instances `d657` (78% faster) and `pr226` (50% faster), where the domain reduction obtained from the BBB phase substantially narrows the search space explored by the IP solver. The second group consists of 12 instances for which the IP baseline exceeds the 3-hour time limit. In this regime, the advantage of the proposed framework becomes more pronounced. On instances such as `vm1084`, `pcb1173`, and `nrw1379`, BBB+DRR achieves final optimality gaps below 0.1%, whereas the IP baseline reports gaps in the range of 8–10%. The average gap reduction across these timeout instances is 49.64%. In addition, on instance `p654`, our approach reaches optimality within the time limit, while the IP baseline does not. Overall, these results highlight the effectiveness of the objective-domain reduction strategy, which produces tighter bounds and leads to substantially improved convergence behavior.

6.2 Capacitated Vehicle Routing Problem

In this subsection, we evaluate the proposed framework on CVRP instances from the CVRPLIB benchmark library. Specifically, we consider two sets of benchmark instances: Set A (Augerat et al. 1995) and Set M (Christofides et al. 1979), the latter being regarded as one of the most challenging collections in the library. Similar to the TSP experiments, we compute the IUB using Google

OR-Tools, specifically the Routing Library with Guided Local Search, while the ILB is obtained from the linear programming relaxation of the arc-based CVRP formulation. To enable feasibility checks using the CP-SAT solver within the proposed framework, we formulate the CVRP using the `AddMultipleCircuit` global constraint, which enforces that each vehicle route forms a valid circuit starting and ending at the depot. For the CVRP, the parameter tuning process for T_b and T resulted in a strategy different from that used for the TSP. Preliminary experiments revealed that the middle bits are the most influential in determining the overall computational time. Although the most significant bits define the largest regions of the objective-value domain, allocating excessive time to them was not beneficial. Instead, traversing these bits quickly and reallocating computational effort to the middle bits enabled earlier pruning as infeasibility information propagated through the search tree. Consequently, the remaining bits could often be fixed more efficiently.

Table 4: Performance comparison on Class A CVRP instances

	Instance	BBB+DRR Time (s)	IP Time (s)	BBB+DRR Gap (%)	IP Gap (%)	Improv. (%)
IP < 500s	A-n32-k5	81.75*	56.31	0.00	0.00	−45.18
	A-n33-k5	70.88*	23.58	0.00	0.00	−200.59
	A-n33-k6	196.52*	132.88	0.00	0.00	−47.89
	A-n34-k5	**145.45***	253.55	0.00	0.00	**42.63**
	A-n37-k5	**97.88***	258.84	0.00	0.00	**62.18**
	A-n38-k5	**66.10***	358.18	0.00	0.00	**81.54**
					Avg. time improvement:	−17.89
IP > 500s	A-n36-k5	**112.86***	1,195.14	0.00	0.00	90.56
	A-n39-k6	**726.80***	1,048.91	0.00	0.00	30.71
	A-n45-k6	**1,582.10***	4,085.35	0.00	0.00	61.28
	A-n46-k7	**390.79***	6,316.89	0.00	0.00	93.81
	A-n53-k7	20,368	12,884.51	0.00	0.00	-58.06
					Avg. time improvement:	**43.66**
IP timeout	A-n37-k6	timeout	timeout	**0.31**	0.74	**58.11**
	A-n39-k5	**2,992.12***	timeout	**0.00**	**0.97**	—
	A-n44-k6	**1,838.19***	timeout	**0.00**	**0.75**	—
	A-n45-k7	timeout	timeout	**2.09**	8.20	**74.51**
	A-n48-k7	timeout	timeout	**1.39**	2.33	**40.34**
	A-n54-k7	timeout	timeout	**3.67**	9.07	**59.54**
	A-n55-k9	timeout	timeout	**0.84**	1.86	**54.84**
	A-n60-k9	timeout	timeout	**5.46**	8.98	**39.20**
	A-n61-k9	timeout	timeout	**4.15**	6.11	**32.08**
	A-n62-k8	timeout	timeout	**8.29**	10.18	18.57
	A-n63-k9	timeout	timeout	**4.44**	7.24	**38.67**
	A-n63-k10	timeout	timeout	**5.52**	7.19	**23.23**
	A-n64-k9	timeout	timeout	**6.47**	8.24	**21.48**
	A-n65-k9	timeout	timeout	**2.70**	3.80	**28.95**
	A-n69-k9	timeout	timeout	**4.78**	6.80	**29.71**
	A-n80-k10	timeout	timeout	**8.43**	9.56	**11.82**
					Avg. gap improvement:	**37.93**

Notes. * indicates instances solved optimally by BBB alone. Bold values indicate better performance by BBB+DRR. Improvement is measured by runtime if both methods solve the instance, or by optimality gap if both time out.

Based on these observations, we partition the bits into three segments and allocate additional time to the middle segment. Specifically, the middle segment receives about twice the computational budget assigned to the other segments. For the middle bits, we set $T_b = 80\ln(n)$ for Set A instances

Table 5: Performance comparison on Class M CVRP instances

Instance	BBB+DRR Time (s)	IP Time (s)	BBB+DRR Gap (%)	IP Gap (%)	Improv. (%)
M-n101-k10	**1,261.37***	18,772.08	0.00	0.00	**93.28** (Time)
M-n121-k7	timeout	timeout	**8.52**	14.90	**42.82**
M-n151-k12	timeout	timeout	**7.42**	9.26	**19.87**
M-n200-k16	timeout	timeout	**14.20**	15.53	**8.56**
M-n200-k17	timeout	timeout	**12.53**	13.67	**8.34**
				Avg. gap improvement:	**19.90**

Notes. * indicates instances solved optimally by BBB alone. Bold values indicate better performance by BBB+DRR. Improvement is measured by runtime if both methods solve the instance, or by optimality gap if both time out.

and $T_b = 640 \ln(n)$ for Set M instances, where n denotes the number of nodes in the CVRP instance. Furthermore, the overall search time parameter T is set to $1200 \ln(n)$ and $10000 \ln(n)$ for Set A and Set M, respectively. We also emphasize that tuning the parameters T_b and T at the problem-class level and designing a general strategy significantly improve solution performance. However, the impact can be substantially greater if these parameters are tuned adaptively for each individual instance based on its characteristics. For example, for the Set A instance A-n53-k7, our general tuning strategy solves the problem in approximately 20,368 seconds using the proposed tuned values. In contrast, when the parameters are specifically tuned for that instance, the solution time decreases to below 8,000 seconds, representing nearly a threefold improvement. Developing such an adaptive instance-based tuning strategy remains an important direction for future research, where machine learning models may prove particularly useful.

Table 4 presents results for 27 Set A instances. Instances solved by BBB alone, without requiring DRR, are marked with '*'. Six instances were identified as the first group, for which the IP baseline solved the problem within 500 seconds. All instances in this group were also solved by BBB alone; notably, the three largest instances were solved significantly faster than the IP baseline. The second group consists of five instances that were solved to optimality by the IP baseline within the imposed time limit but required more than 500 seconds. For these instances, the proposed framework achieved an average time improvement of approximately 43.66%, with most instances again solved by BBB alone. The third group contains 16 instances that the IP baseline could not solve to optimality within the imposed time limit. Among these, two instances were solved to optimality by BBB alone in less than 50 minutes. The remaining 14 instances timed out for both BBB+DRR and the IP baseline. However, the proposed framework achieved a substantially better optimality gap, improving the average gap by approximately 38%. Table 5 presents results for the Set M instances. Five instances were considered, each containing between 101 and 200 customers. Interestingly, for instance M-n101-k10, which includes 101 customers and 10 vehicles, BBB alone proved optimality in under 1,300 seconds, whereas the IP baseline required more than 15 times longer. The remaining four instances timed out under both approaches. Nevertheless, the proposed framework again produced significantly better optimality gaps, with an average improvement of approximately 19.9%.

7 Conclusion

This paper addressed a central question in solving challenging integer programs: whether the domain of objective function values can be meaningfully narrowed, and whether such a reduction can translate into improved solver performance. To answer this question, we developed a solution framework that combines ideas from binary search, branch-and-bound, and binary-encoded reformulations. The proposed framework progressively reduces the objective domain to either a single value, corresponding to the optimal objective value, or to a collection of disjoint ranges guaranteed to contain the optimum. In the former case, the optimal solution is obtained directly. In the latter case, the remaining objective ranges are exploited through the proposed Domain-Reduced Reformulation to further guide the search process. We applied the framework to two well-studied and challenging classes of combinatorial optimization problems, namely TSP and CVRP, using standard benchmark instances from TSPLIB and CVRPLIB. Computational results confirm that, on challenging instances, narrowing the objective domain through the proposed framework can significantly improve solver performance, either by reducing computation time or by producing substantially tighter optimality gaps compared to a standard IP solver equipped with lazy constraint callbacks.

Several directions for future research emerge from this work. First, the disjoint open subproblems produced by the Bitwise Branch-and-Bound method are independent and can therefore be explored in parallel, which may substantially reduce computational time for large-scale instances. Second, since the binary-encoded reformulation is solver-agnostic, the proposed framework could potentially be integrated with other advanced optimization techniques. In particular, combining it with decomposition approaches such as Benders decomposition and column generation appears to be a promising direction. Third, adaptive time-allocation strategies within the Bitwise Branch-and-Bound framework could have a significant impact on overall performance. Consequently, developing machine learning or artificial intelligence models to dynamically estimate suitable time limits based on instance characteristics represents another interesting avenue for future research.

References

Achuthan NR, Caccetta L, Hill SP (2003) An improved branch-and-cut algorithm for the capacitated vehicle routing problem. *Transportation Science* 37(2):153–169.

Adams WP, Henry SM (2012) Base-2 expansions for linearizing products of functions of discrete variables. *Operations Research* 60(6):1477–1490.

Applegate DL, Bixby RE, Chvátal V, Cook WJ (2011) *The Traveling Salesman Problem: A Computational Study* (Princeton University Press).

Augerat P, Belenguer J, Benavent E, Corberán A, Naddef D, Rinaldi G (1995) Computational results with a branch and cut code for the capacitated vehicle routing problem. Technical Report 949-M, Université Joseph Fourier, Grenoble, France.

Barnhart C, Johnson EL, Nemhauser GL, Savelsbergh MW, Vance PH (1998) Branch-and-price: Column generation for solving huge integer programs. *Operations research* 46(3):316–329.

Belenguer JM, Benavent E (2003) A cutting plane algorithm for the capacitated arc routing problem. *Computers & Operations Research* 30(5):705–728.

Benders J (1962) Partitioning procedures for solving mixed-variables programming problems. *Numerische Mathematik* 4(1):238–252.

Christofides N, Mingozzi A, Toth P (1979) The vehicle routing problem. Christofides N, Mingozzi A, Toth P, Sandi C, eds., *Combinatorial Optimization*, 315–338 (Chichester: Wiley).

Clautiaux F, Ljubić I (2025) Last fifty years of integer linear programming: A focus on recent practical advances. *European Journal of Operational Research* 324(3):707–731.

Codato G, Fischetti M (2006) Combinatorial benders' cuts for mixed-integer linear programming. *Operations Research* 54(4):756–766.

Contardo C, Lodi A, Tramontani A (2023) Cutting planes from the branch-and-bound tree: Challenges and opportunities. *INFORMS Journal on Computing* 35(1):2–4.

Errami N, Queiroga E, Sadykov R, Uchoa E (2024) Vrpsolvereasy: a python library for the exact solution of a rich vehicle routing problem. *INFORMS Journal on Computing* 36(4):956–965.

Forbes MA, Harris MG, Jansen H, Van Der Schoot FA, Taimre T (2024) Combining optimisation and simulation using logic-based benders decomposition. *European Journal of Operational Research* 312(3):840–854.

Fukasawa R, Longo H, Lysgaard J, Aragão MPd, Reis M, Uchoa E, Werneck RF (2006) Robust branch-and-cut-and-price for the capacitated vehicle routing problem. *Mathematical programming* 106(3):491–511.

Ghasemi Saghand P, Rigterink F, Mahmoodian V, Charkhgard H (2023) Solving multiplicative programs by binary-encoding the multiplication operation. *Computers & Operations Research* 159:106340.

Hooker JN, Van Hoeve WJ (2018) Constraint programming and operations research. *Constraints* 23(2):172–195.

Khatami M, Salehipour A (2020) A binary search algorithm for the general coupled task scheduling problem. *4OR* 19:249–272.

Lübbecke ME, Desrosiers J (2005) Selected topics in column generation. *Operations research* 53(6):1007–1023.

Morrison DR, Jacobson SH, Sauppe JJ, Sewell EC (2016) Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning. *Discrete optimization* 19:79–102.

Muldoon FM, Adams WP, Sherali HD (2013) Ideal representations of lexicographic orderings and base-2 expansions of integer variables. *Operations Research Letters* 41(1):32–39.

Owen JH, Mehrotra S (2002) On the value of binary expansions for general mixed-integer linear programs. *Operations Research* 50(5):810–819.

Pecin D, Pessoa A, Poggi M, Uchoa E (2017) Improved branch-cut-and-price for capacitated vehicle routing. *Mathematical Programming Computation* 9(1):61–100.

Perron L, Didier F (2025) CP-SAT. URL https://developers.google.com/optimization/cp/cp_solver/, v9.12, Google.

Rahmaniani R, Crainic TG, Gendreau M, Rei W (2017) The benders decomposition algorithm: A literature review. *European Journal of Operational Research* 259(3):801–817.

Reinelt G (1991) Tsplib—a traveling salesman problem library. *ORSA journal on computing* 3(4):376–384.

Rossi F, Van Beek P, Walsh T (2006) *Handbook of constraint programming* (Elsevier).

Stuckey PJ (2010) Lazy clause generation: Combining the power of sat and cp (and mip?) solving. *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, 5–9 (Springer).

Watters LJ (1967) Reduction of integer polynomial programming problems to zero-one linear programming problems. *Operations Research* 15(6):1171–1174.

Yao S, Tang C, Zhang H, Wu S, Wei L, Liu Q (2024) An iteratively doubling binary search for the two-dimensional irregular multiple-size bin packing problem raised in the steel industry. *Computers & Operations Research* 162:106481.

Zhang D, Wei L, Leung SCH, Chen Q (2013) A binary search heuristic algorithm based on randomized local search for the rectangular strip-packing problem. *INFORMS Journal on Computing* 25(2):332–345.

8 Example

In this section, we demonstrate the main components of the proposed Bitwise Branch-and-Bound method and the Domain-Reduced Reformulation using our running TSP example. Figure H-1 provides a graphical illustration of the progress of the Bitwise Branch-and-Bound method after exploring the first seven bits. For illustrative purposes, we employ a breadth-first search strategy in this example instead of the default best-bound strategy, as it better highlights the different branching and pruning mechanisms of the proposed method. Recall from Section 4 that IUB = 8980, ILB = 7390, and $K = 14$. These values are provided to the proposed method at the root node. Starting from the root, the algorithm first tests whether setting $b_0 = 0$ is feasible. Since feasibility is confirmed, the GUB is improved from the IUB to 8102.

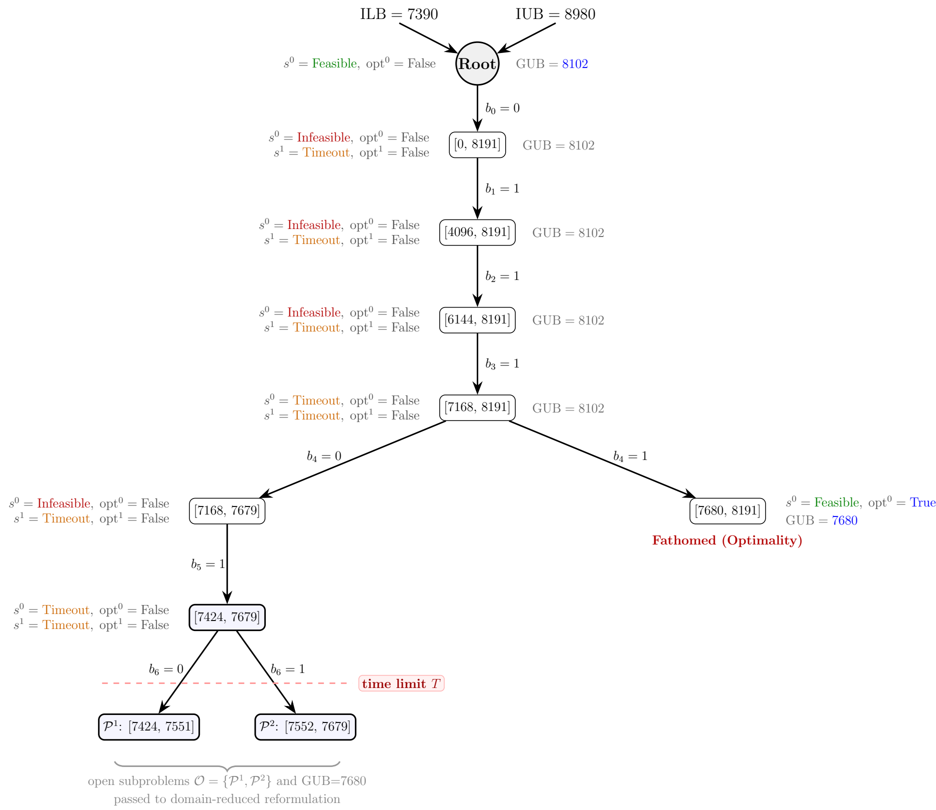

Figure H-1: Illustration of the Bitwise Branch-and-Bound method on the `berlin52` instance using breadth-first search, terminated after exploring seven bits.

For the next three bits, the method successfully determines that $b_1 = b_2 = b_3 = 1$. When exploring bit b_4, the method is unable to determine a unique value within the prescribed feasibility-check time limit, and therefore branching occurs, generating two child nodes. Among these two nodes, one is subsequently fathomed because the feasibility-check solver not only identifies a feasible

solution but also proves it optimal for that branch. As a result, the GUB is improved further to 7680. The remaining non-fathomed node is then explored, and the method successfully determines that $b_5 = 1$ for that branch. However, the method branches again because the value of b_6 cannot be determined conclusively within the allotted feasibility-check time limit. Suppose that the overall time limit is reached at this stage. The proposed method therefore terminates and returns the current GUB value of 7680, together with the following two remaining open objective intervals to be passed to the Domain-Reduced Reformulation:

$$[7424, 7551], \quad \text{and} \quad [7552, 7679].$$

Note that GUB = 7680 corresponds to the binary representation $(01111000000000)_2$ with $K = 14$ bits. With this in mind, to construct the Domain-Reduced Reformulation, we generate $M + 1$ subproblems (i.e., 3 in this example) as follows:

$$\begin{aligned}
&\mathcal{C}^1 = \{(0,0),(1,1),(2,1),(3,1),(4,0),(5,1),(6,0)\}, && \mathrm{LB}(\mathcal{C}^1) = 7424, && \mathrm{UB}(\mathcal{C}^1) = 7551,\\
&\mathcal{C}^2 = \{(0,0),(1,1),(2,1),(3,1),(4,0),(5,1),(6,1)\}, && \mathrm{LB}(\mathcal{C}^2) = 7552, && \mathrm{UB}(\mathcal{C}^2) = 7679,\\
&\mathcal{C}^3 = \{(0,0),(1,1),(2,1),(3,1),(4,1),(5,0),(6,0), && &&\\
&\qquad (7,0),(8,0),(9,0),(10,0),(11,0),(12,0),(13,0)\}. && \mathrm{LB}(\mathcal{C}^5) = 7680, && \mathrm{UB}(\mathcal{C}^5) = 7680.
\end{aligned}$$

Using these subproblems, the bitwise lower and upper bounds (l_k^j, u_k^j) required to construct the proposed Domain-Reduced Reformulation are summarized in Table H-1 for subproblems $\mathcal{P}^1$–$\mathcal{P}^3$. In the reformulation, whenever a subproblem $\mathcal{P}^j$ is selected (i.e., $y_j = 1$), the bit variables are forced to satisfy the corresponding partial bit assignment $\mathcal{C}^j$. For example, if $y_1 = 1$, then subproblem $\mathcal{P}^1$ becomes active, and the feasible search space is automatically restricted to the integer objective interval $[7424, 7551]$.

Table H-1: Bitwise lower and upper bounds (l_k^j, u_k^j) for subproblems $\mathcal{P}^1$–$\mathcal{P}^3$

	b_0	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	b_9	b_{10}	b_{11}	b_{12}	b_{13}
$\mathcal{P}^1$	(0,0)	(1,1)	(1,1)	(1,1)	(0,0)	(1,1)	(0,0)	(0,1)	(0,1)	(0,1)	(0,1)	(0,1)	(0,1)	(0,1)
$\mathcal{P}^2$	(0,0)	(1,1)	(1,1)	(1,1)	(0,0)	(1,1)	(1,1)	(0,1)	(0,1)	(0,1)	(0,1)	(0,1)	(0,1)	(0,1)
$\mathcal{P}^3$	(0,0)	(1,1)	(1,1)	(1,1)	(1,1)	(0,0)	(0,0)	(0,0)	(0,0)	(0,0)	(0,0)	(0,0)	(0,0)	(0,0)