Branch and Bound: Theory, Algorithms, and Applications

A Integer Programming: Motivation and Challenges

Integer programming (IP) is a branch of mathematical optimization where some or all decision variables are required to take integer values. IP models are essential for representing real-world problems involving discrete choices, such as scheduling, assignment, facility location, and network design.

A.1 Motivation

Many practical optimization problems cannot be solved meaningfully with fractional solutions. For example:

- Assigning workers to shifts (each worker is either assigned or not).
- Selecting projects under a budget (each project is either chosen or not).
- Routing vehicles or goods (a route is either used or not).

In such cases, integer programming provides a natural and necessary modeling framework.

A.2 Challenges of Integer Programming

While linear programming (LP) problems can be solved efficiently using algorithms like the Simplex Method, integer programming is fundamentally more difficult:

- **Combinatorial Explosion:** The number of possible integer solutions grows exponentially with the number of variables.
- **NP-Hardness:** Many IPs are NP-hard, meaning no polynomial-time algorithm is known for solving all instances.
- **Non-Convexity:** The feasible region of an IP is generally non-convex, so LP techniques cannot be directly applied.
- **Integrality Gap:** The optimal value of the LP relaxation can differ significantly from the true integer optimum.

A.3 Why Not Just Use LP?

Relaxing the integrality constraints and solving the LP relaxation often yields fractional solutions that are not feasible for the original problem. Rounding these solutions can lead to infeasibility or suboptimality. Specialized algorithms, such as Branch and Bound, are required to systematically search for optimal integer solutions.

A.4 Overview of Solution Approaches

Several approaches exist for solving IPs:

- **Exact Methods:** Branch and Bound, Cutting Planes, Branch-and-Cut, Branch-and-Price.
- **Heuristics:** Greedy algorithms, local search, metaheuristics (e.g., genetic algorithms, simulated annealing).
- **Relaxation and Rounding:** Solve the LP relaxation and round the solution, possibly with additional feasibility checks.

This chapter focuses on Branch and Bound, the foundational exact method for integer programming.

B Branch and Bound

Branch and Bound is a fundamental algorithmic framework for solving integer programming (IP) and combinatorial optimization problems. It was introduced by A. H. Land and A. G. Doig in 1960 as a general framework for solving integer programming and combinatorial optimization problems, and has since become the backbone of exact methods for discrete optimization. Branch and Bound systematically explores the solution space by partitioning it into smaller subproblems (branching) and using bounds to eliminate subproblems that cannot contain an optimal solution (bounding).

Note: In this chapter, we focus on minimization problems of the form $\min\{c^T x : Ax \geq b,\ x \in \mathbb{Z}_+^n\}$. Maximization problems can always be transformed into minimization problems by negating the objective function (i.e., $\max c^T x$ is equivalent to $\min -c^T x$), so all results and algorithms apply to both cases.

B.1 Mathematical Formulation

Consider the integer program:

$$\begin{aligned} \text{minimize} \quad & c^T x \\ \text{subject to} \quad & Ax \geq b \\ & x \in \mathbb{Z}_+^n \end{aligned}$$

where $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$.

B.2 Basic Definitions

- **Relaxation:** The linear program obtained by removing the integrality constraints ($x \in \mathbb{R}_+^n$).
- **Node:** A subproblem defined by additional constraints (e.g., fixing some variables).
- **Branching:** Splitting a node into two or more subproblems by imposing new constraints on a variable.
- **Bounding:** Computing an upper (or lower) bound on the optimal value of a node's subproblem.

B.3 The Branch and Bound Algorithm

1. Start with the root node (the original problem).
2. Solve the LP relaxation. If the solution is integer, update incumbent (best known integer solution).
3. If the solution is fractional, select a variable x_j with a fractional value and branch:
   - Create two subproblems: $x_j \leq \lfloor x_j^* \rfloor$ and $x_j \geq \lceil x_j^* \rceil$.
4. For each subproblem:
   - If infeasible or its bound is worse (greater) than the incumbent, prune it.
   - Otherwise, repeat the process recursively.
5. Terminate when all nodes are pruned or solved.

B.4 Branching Strategies

The choice of variable and value to branch on can significantly affect the size of the search tree and the efficiency of the algorithm.

- **Most Fractional Branching:** Select the variable whose LP relaxation value is furthest from integrality. This often leads to faster progress toward integrality, but may not always reduce the tree size.
- **Largest Coefficient Branching:** Select the variable with the largest objective coefficient among those with fractional values. This can quickly improve the objective but may not always yield the smallest tree.
- **Pseudo-cost Branching:** Use historical information about the impact of branching on each variable. This adaptive strategy often leads to smaller trees and faster convergence in practice.

In general, branching on variables that are most fractional or have the greatest impact on the objective can reduce the number of nodes explored. However, the best choice may depend on the problem structure. Modern solvers often use hybrid or adaptive strategies.

B.5 Exploration Strategies

The order in which nodes are explored (the search strategy) also impacts performance.

- **Depth-First Search (DFS):** Explores one branch to completion before backtracking. Uses less memory and can quickly find feasible solutions, but may miss better solutions in other branches until later.
- **Breadth-First Search (BFS):** Explores all nodes at the current depth before moving deeper. Guarantees finding the optimal solution with the smallest number of branches, but can require much more memory.
- **Best-First Search:** Always explores the node with the best (lowest) bound. This can lead to faster discovery of the optimal solution, but may require maintaining a large list of open nodes.

DFS is memory-efficient and good for quickly finding feasible solutions, while best-first is often preferred for quickly proving optimality. The choice should balance memory usage, speed to first feasible solution, and speed to optimality. Many solvers use a combination of these strategies.

C Pruning Strategies

Pruning is the process of eliminating nodes (subproblems) from the Branch and Bound search tree that cannot lead to a better integer solution. Effective pruning is essential for reducing the computational effort required to solve integer programming problems.

Definition of Pruning

A node is pruned if it is determined that no feasible or better integer solution can be found in its subproblem. Pruning prevents unnecessary exploration of the search tree and focuses computational resources on promising regions.

Common Pruning Strategies

- **Infeasibility Pruning:** If the LP relaxation at a node is infeasible, then the subproblem has no feasible solutions and can be pruned.
- **Bound Pruning:** If the lower bound (for minimization) at a node is greater than or equal to the best known integer solution (incumbent), the node can be pruned because it cannot yield a better solution.

- **Integrality Pruning:** If the LP relaxation at a node yields an integer solution, update the incumbent and prune the node, as further branching is unnecessary.
- **Dominance Pruning:** If a node's feasible region is a subset of another node already explored with a better or equal bound, it can be pruned (rare in practice, but possible in highly structured problems).

Guidelines: Pruning is critical for the efficiency of Branch and Bound. The more aggressively and accurately nodes can be pruned, the smaller the search tree and the faster the algorithm.

C.1 Mathematical Properties and Proofs

Correctness. The algorithm is correct because it exhaustively explores all feasible integer solutions, but prunes subproblems that provably cannot improve the incumbent.

Proof. At each node, the feasible region is partitioned so that every integer solution is contained in exactly one subproblem. Pruning is only performed when a node is infeasible or its bound is worse than the incumbent, so no optimal integer solution is lost. Thus, the best integer solution found is optimal. □

Finite Termination. The algorithm terminates in finitely many steps because each branch adds a constraint, and there are only finitely many integer vectors in any bounded region.

C.2 A Bit of History

The Simplex Method was invented by George Dantzig in 1947 and quickly became the standard algorithm for solving linear programming problems due to its practical efficiency. Branch and Bound was introduced by A. H. Land and A. G. Doig in 1960 as a general framework for solving integer programming and combinatorial optimization problems. Both methods have had a profound impact on the field of optimization and operations research.

C.3 Proof of Correctness for Branch and Bound

The Branch and Bound algorithm is guaranteed to find an optimal integer solution (if one exists) or prove infeasibility. Here is a formal proof of correctness:

Theorem 1. *Branch and Bound always returns an optimal solution to the integer program, or correctly determines that none exists.*

Proof. Let S be the set of all feasible integer solutions. The algorithm starts with the original problem and recursively partitions the feasible region into subproblems (nodes), each defined by additional constraints. At each node, the LP relaxation provides an upper bound (for minimization). If the relaxation is infeasible or its bound is worse than the best known integer solution (incumbent), the node is pruned.
Every feasible integer solution is contained in at least one node's feasible region. The algorithm only prunes nodes that cannot contain a better integer solution than the incumbent. When a node's LP relaxation is integer and better than the incumbent, the incumbent is updated. Since there are finitely many integer solutions in any bounded region, and the algorithm explores all possibilities unless pruned, the best integer solution found is optimal. If no integer solution is found, the problem is infeasible. □

C.4 Limitations of Branch and Bound

- **Exponential Complexity:** In the worst case, the number of nodes explored is exponential in the number of variables.
- **Memory Usage:** Best-first and breadth-first strategies can require storing many nodes in memory.

- **Weak Bounds:** If the LP relaxation provides poor bounds, many nodes must be explored.
- **Symmetry:** Problems with many symmetric solutions can lead to redundant exploration.

D Numerical Example

Consider the following integer program:

$$\begin{aligned} \text{minimize} \quad & z = 100x_1 + 120x_2 \\ \text{subject to} \quad & 7x_1 + 2x_2 \leq 28 \\ & 2x_1 + 12x_2 \leq 24 \\ & x_1, x_2 \geq 0,\ x_1, x_2 \in \mathbb{Z} \end{aligned}$$

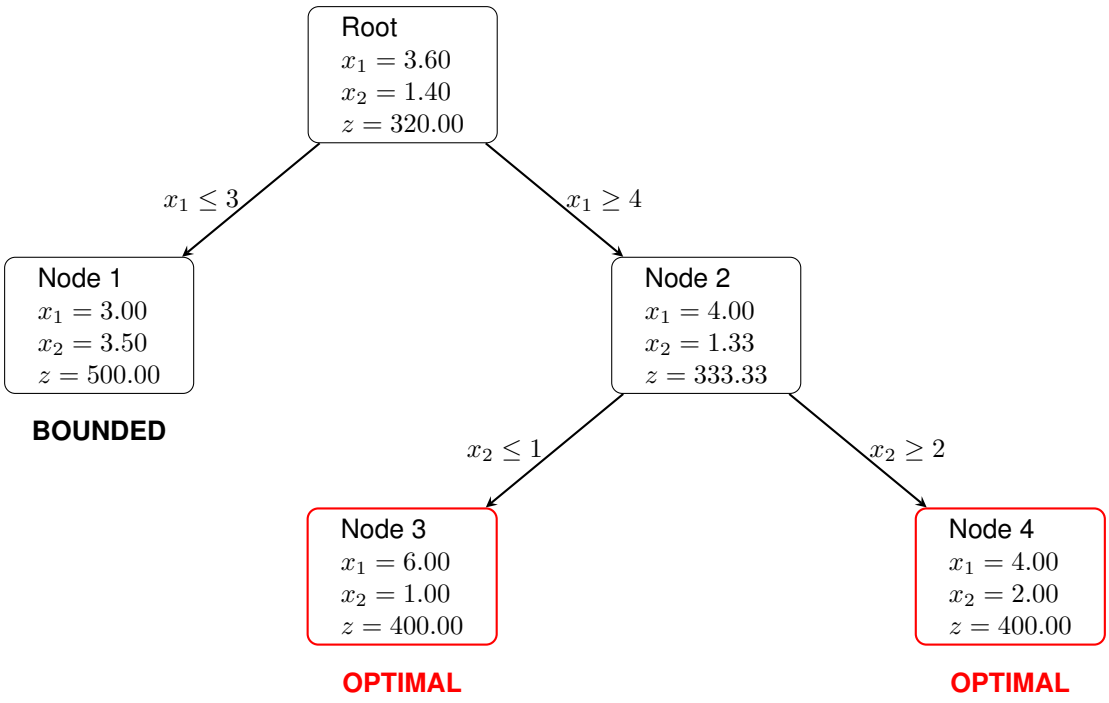
Step 1: Solve the LP relaxation (ignore integrality): the solution is $x_1^* = 3.6$, $x_2^* = 1.4$, $z^* = 320$.
Step 2: Branch on x_1:

- Node 1: $x_1 \leq 3$ (LP relaxation: $x_1 = 3$, $x_2 = 3.5$, $z^* = 500$; bounded/pruned if z^* worse than incumbent)
- Node 2: $x_1 \geq 4$ (LP relaxation: $x_1 = 4$, $x_2 = 1.33$, $z^* = 333.33$)

Step 3: Branch on x_2 in Node 2:

- Node 3: $x_2 \leq 1$ (LP relaxation: $x_1 = 6$, $x_2 = 1$, $z^* = 400$; integer, optimal)
- Node 4: $x_2 \geq 2$ (LP relaxation: $x_1 = 4$, $x_2 = 2$, $z^* = 400$; integer, optimal)

Bounding: At each node, if the LP relaxation value z^* is less than the best known integer solution, the node is pruned. For example, if an incumbent $z = 400$ is found, any node with $z^* < 400$ is pruned.
Exploration Tree:



This example illustrates how the Branch and Bound algorithm uses bounding to eliminate subproblems and efficiently find the optimal integer solution.
The exploration tree in Branch and Bound visually and conceptually represents the sequence of subproblems (nodes) generated and explored during the algorithm. The order in which nodes are explored can significantly affect computational effort and the speed at which good feasible solutions are found.
For example, exploring Node 1 before Node 2 may result in more computation (more nodes generated), but it would not prevent the eventual exploration of Node 2, especially if Node 2 has a lower z value. Even if an integer solution is found early, we cannot know in advance whether exploring another node (like Node 2) will yield a better feasible solution for pruning or will converge more quickly to an integer solution. Thus, the structure and traversal of the exploration tree are crucial for both efficiency and effectiveness, but optimal choices are not always apparent a priori.

E Variable Fixing Strategies

Variable fixing refers to the process of setting certain variables to specific values (often 0 or 1 in binary/integer programs) in order to reduce the size of the search space and accelerate the solution process. Effective variable fixing can be based on problem structure, reduced costs, or logical inference.
Common Variable Fixing Strategies include:

- **Reduced Cost Fixing:** In the LP relaxation, if the reduced cost of a non-basic variable indicates that setting it to a particular value cannot improve the objective beyond the current best integer solution (incumbent), the variable can be fixed at that value. For example, in a minimization problem, if increasing x_j from 0 would increase the objective above the incumbent, set $x_j = 0$.
- **Logical Inference:** Use problem-specific logic or constraints to deduce that certain variables must take specific values. For example, in assignment problems, if all but one assignment in a row or column are fixed, the remaining variable can be fixed by feasibility.
- **Preprocessing Bounds:** Tighten variable bounds using constraint propagation or presolve techniques before or during Branch and Bound. If a variable's lower and upper bounds coincide, it can be fixed.
- **Dominance and Symmetry Breaking:** Fix variables to break symmetries or eliminate dominated solutions, reducing redundant exploration.

Variable fixing is most effective when combined with strong bounds and problem-specific knowledge. It can dramatically reduce the number of nodes explored in Branch and Bound, especially in large-scale or highly structured problems.

F A Note on Parallelization

Branch and Bound can be parallelized to improve performance on modern multi-core and distributed systems. The key idea is to explore multiple nodes simultaneously, leveraging the independent nature of subproblems:

- **Node Pooling:** Maintain a pool of nodes to be explored, allowing multiple threads or processes to pick nodes independently. This can lead to better load balancing and reduced idle time.
- **Bounding and Pruning:** Each thread can independently compute bounds and prune nodes, allowing for faster convergence to the optimal solution.

- **Dynamic Load Balancing:** As threads finish exploring nodes, they can dynamically take on new nodes from the pool, ensuring that all computational resources are utilized effectively.
- **Task Granularity:** Adjust the granularity of tasks assigned to threads to optimize performance. Finer granularity can improve load balancing, while coarser granularity can reduce overhead.
- **Synchronization:** Minimize synchronization overhead by allowing threads to work independently as much as possible. Use locks or atomic operations only when necessary (e.g., updating the incumbent solution).

Parallelization can significantly speed up Branch and Bound, especially for large and complex integer programming problems. However, care must be taken to manage shared resources (like the incumbent solution) and to ensure that the parallelization does not introduce inconsistencies in the search process.