Operations Research Essentials You Always Wanted to Know

Nitin Singh

ISBN ...

... Publications

To my Son

Contents

Chapter 1

Concepts in Linear Programming

1.1 Linear Programming: A Brief Overview

Linear Programming (LP) is a mathematical optimization technique used to find the best outcome in a mathematical model whose requirements are represented by linear relationships. It is widely applied in various fields such as economics, business, engineering, and military operations for decision-making and resource allocation. The key components of a linear programming problem include:

- **Decision Variables:** These are the unknowns that we want to solve for. They represent the choices available to the decision-maker.
- **Objective Function:** This is a linear function of the decision variables that we want to maximize or minimize. It represents the goal of the optimization, such as maximizing profit or minimizing cost.
- **Constraints:** These are linear inequalities or equations that restrict the values of the decision variables. They represent the limitations or requirements of the problem, such as resource availability or demand satisfaction.

The standard form of a linear programming problem can be expressed as:

$$\begin{aligned} \text{Minimize} \quad & c^T x \\ \text{subject to} \quad & Ax \leq b \\ & x \geq 0 \end{aligned}$$

where c is the coefficient vector of the objective function, x is the vector of decision variables, A is the matrix of coefficients for the constraints, and b is the right-hand side vector of the constraints.

1.2 Methods to solve Linear Programming Problems

Several algorithms have been developed to solve linear programming problems, with the most notable being:

- **Simplex Method:** Developed by George Dantzig in 1947, the Simplex method is an iterative procedure that moves along the edges of the feasible region defined by the constraints to find the optimal solution. It is efficient for solving small to medium-sized problems.
- **Interior Point Methods:** These methods, such as Karmarkar's algorithm, approach the optimal solution from within the feasible region rather than traversing the edges. They are particularly effective for large-scale linear programming problems.

1.2.1 Details of how Gurobi solves Linear Programming Problems

Gurobi is a state-of-the-art optimization solver that implements both the Simplex method and Interior Point methods to solve linear programming problems efficiently. Gurobi's approach to solving LP problems involves several key steps:

- **Preprocessing:** Gurobi performs various preprocessing techniques to simplify the problem before applying the main optimization algorithms. This includes removing redundant constraints, tightening bounds, and identifying infeasibilities.
- **Algorithm Selection:** Based on the problem characteristics (such as size, sparsity, and structure), Gurobi automatically selects the most appropriate algorithm (Simplex or Interior Point) to solve the LP problem efficiently.
- **Iterative Optimization:** Gurobi iteratively refines the solution by performing pivot operations in the Simplex method or by following a path in the Interior Point method until it converges to the optimal solution.
- **Postprocessing:** After finding an optimal solution, Gurobi may perform additional postprocessing steps to ensure numerical stability and to provide sensitivity analysis results.
- **Parallelization:** Gurobi takes advantage of modern multi-core processors to parallelize certain computations, such as matrix factorizations and presolve operations, to further speed up the solution process.
- **Numerical Stability:** Gurobi incorporates various techniques to maintain numerical stability throughout the optimization process, such as scaling and pivoting strategies, to ensure that the solutions are accurate and reliable.

1.2.2 Details of how CPLEX solves Linear Programming Problems

1.3 Inside the Engine: How CPLEX Solves Linear Programs

When you trigger a solve in CPLEX, the software doesn't just jump into the math. It follows a highly engineered pipeline designed for speed and numerical robustness.

1.3.1 Phase 1: The Presolve (The Hidden Optimizer)

Before any iterations occur, CPLEX performs *Presolve*. This step simplifies the model to reduce its size and improve its "condition number." Key actions include:

- **Variable Fixing:** Identifying variables that must be at their bounds to satisfy the constraints.
- **Redundancy Removal:** Deleting constraints that are logically implied by others.
- **Coefficient Tightening:** Scaling numbers to prevent numerical instability.

1.3.2 Phase 2: Choosing the Algorithm

CPLEX primarily relies on three algorithmic engines to find the optimal vertex:

Simplex Methods (Primal and Dual)

The Simplex algorithm moves along the edges of the feasible region from one vertex to another.

- **Dual Simplex (Default):** Starts with an optimal but infeasible solution and works toward feasibility. It is the gold standard for most LPs and handles "hot starts" (re-solving after small changes) efficiently.
- **Primal Simplex:** Starts with a feasible solution and travels the polytope edges to improve the objective.

The Barrier Method (Interior Point)

For massive, sparse problems (often $> 10^5$ nonzeros), the Barrier method is usually superior. Instead of walking the edges, it tunnels through the *interior* of the feasible region using a logarithmic barrier function:

$$f(x) = c^T x - \mu \sum_{j=1}^{n} \ln(x_j)$$

As the barrier parameter μ approaches zero, the algorithm converges on the optimal solution.

1.3.3 Phase 3: Crossover

Because the Barrier method ends in the interior of the feasible region, CPLEX performs a *Crossover* step. This "pushes" the interior solution to the nearest vertex to provide a **Basic Optimal Solution**, which is necessary for calculating shadow prices and performing sensitivity analysis.

1.3.4 Key CPLEX Solver Settings

To tune the engine, practitioners often adjust specific parameters.

Table 1.1: Critical CPLEX Parameters for LP Performance

Parameter	Function	Usage Case
`LPMETHOD`	Algorithm Choice	Set to 4 (Barrier) for massive, sparse models.
`SCAIND`	Matrix Scaling	Set to 1 if the log shows "Numerical trouble."
`DPRIIND`	Dual Gradient	Change to "Steepest Edge" if Simplex stalls.
`BAREPOT`	Barrier Tolerance	Adjust for faster, approximate solutions.

1.3.5 Expert Insight: Reading the Logs

An OR scientist must be a "log whisperer." If you see the objective value plateauing for thousands of iterations, the model is likely experiencing **Degeneracy**. If the time per iteration is extremely high, consider switching from Simplex to Barrier to exploit the model's sparsity.

1.4 Duality in Linear Programming

Duality is a fundamental concept in linear programming that establishes a relationship between a primal optimization problem and its corresponding dual problem. The dual problem provides

insights into the original problem and can be used to derive bounds on the optimal value of the primal problem. For a given linear programming problem in standard form:

$$\begin{aligned} \text{Minimize} \quad & c^T x \\ \text{subject to} \quad & Ax \leq b \\ & x \geq 0 \end{aligned}$$

The dual problem can be formulated as:

$$\begin{aligned} \text{Maximize} \quad & b^T y \\ \text{subject to} \quad & A^T y \leq c \\ & y \geq 0 \end{aligned}$$

where y is the vector of dual variables corresponding to the constraints in the primal problem. The duality theory states that if the primal problem has an optimal solution, then the dual problem also has an optimal solution, and the optimal values of the objective functions of both problems are equal. This relationship allows for sensitivity analysis and provides economic interpretations of the constraints and objective function in terms of resource allocation and opportunity costs.

1.5 Dual Variables and Shadow Prices

In Linear Programming (LP), **dual variables** (also known as **shadow prices**) are associated with the constraints of the primal problem. They represent the marginal value of relaxing a constraint by one unit. In other words, a dual variable indicates how much the objective function would improve if the right-hand side of a constraint were increased by one unit, while keeping all other parameters constant. For a linear programming problem in standard form:

$$\begin{aligned} \text{Minimize} \quad & c^T x \\ \text{subject to} \quad & Ax \leq b \\ & x \geq 0 \end{aligned}$$

The dual problem can be formulated as:

$$\begin{aligned} \text{Maximize} \quad & b^T y \\ \text{subject to} \quad & A^T y \leq c \\ & y \geq 0 \end{aligned}$$

where y is the vector of dual variables corresponding to the constraints in the primal problem. The value of each dual variable provides insight into the sensitivity of the optimal solution to changes in the constraints, making them a powerful tool for decision-making and economic interpretation in optimization problems.

1.5.1 Shadow Prices

Shadow prices are the values of the dual variables at the optimal solution. They indicate the worth of an additional unit of a resource or the cost of a constraint. For example, if a shadow price is positive, it means that increasing the right-hand side of the corresponding constraint would lead to an improvement in the objective function (e.g., higher profit or lower cost). Conversely, if a shadow price is zero, it indicates that the constraint is not binding and does not affect the optimal solution. Shadow prices are crucial for understanding the trade-offs in resource allocation and for making informed decisions based on the optimization results.

1.6 Nonzeros in Linear Programming Models

In Linear Programming (LP), the complexity of a model is primarily governed by the **Constraint Matrix** A, defined in the standard form:

$$\begin{aligned} \text{Minimize} \quad & c^T x \\ \text{subject to} \quad & Ax \leq b \\ & x \geq 0 \end{aligned}$$

where $c \in \mathbb{R}^n$ is the cost (objective) vector, $x \in \mathbb{R}^n$ is the vector of decision variables, $A \in \mathbb{R}^{m \times n}$ is the constraint matrix, and $b \in \mathbb{R}^m$ is the right-hand side vector.

The term **"Nonzeros"** refers to the number of entries $a_{ij} \in A$ such that $a_{ij} \neq 0$. This metric is a more accurate representation of problem size than the raw count of variables or constraints because:

- **Sparsity:** Real-world OR models are typically sparse. The number of nonzeros represents a direct link where a specific variable affects a specific constraint.
- **Computational Effort:** The execution time of the Simplex and Barrier algorithms is heavily dependent on the cost of performing matrix factorizations (like LU or $Cholesky$ decomposition). The complexity of these operations scales with the number of nonzeros, not the total number of cells $(m \times n)$.

Sparsity in Linear Programming Models

In the context of Linear Programming (LP) and Optimization, **sparsity** is a fundamental property describing the distribution of coefficients within the constraint matrix A. Specifically, it quantifies how many entries in the matrix are zero versus non-zero. Sparsity describes a matrix where the vast majority of the entries are zero.

- **Sparse Matrix:** Contains relatively few non-zero entries (coefficients).
- **Dense Matrix:** Contains many non-zero entries, where most variables appear in most constraints.

Sparsity is mathematically quantified by the **sparsity ratio**, calculated as follows:

$$\text{Sparsity Ratio} = 1 - \left(\frac{\text{Number of Nonzeros}}{\text{Total Number of Entries } (m \times n)} \right)$$

Where:

- m is the number of constraints (rows).
- n is the number of variables (columns).

A high ratio (e.g., 0.99 or 99%) indicates a highly sparse matrix. Most large-scale industrial models in logistics or supply chain optimization exhibit sparsity ratios exceeding 99.9%. Sparsity matters for solvers. It's a primary determinant of solver efficiency. State-of-the-art solvers like Gurobi or CPLEX are specifically engineered to exploit this property.

Table 1.2: Comparison of Sparse vs. Dense Matrices in Optimization

Property	Sparse Matrix (Most Real-World Problems)	Dense Matrix (Rare in OR)
Storage	Solvers only store the **non-zero** entries and their locations. This saves massive amounts of memory.	Solvers must store every single entry, leading to huge memory usage for large problems.
Computation	Solving involves mainly multiplying sparse matrices, which takes far fewer operations than multiplying dense matrices.	Every calculation must process many zero entries, wasting computational power.
Algorithm Performance	Favors algorithms that exploit structure, leading to faster factorization and solution times.	Slows down matrix operations significantly.

1.7 Slack Variables

In Linear Programming (LP), **slack variables** are introduced to convert inequality constraints into equality constraints. Most optimization algorithms, such as the Simplex method, require constraints to be in the form of linear equations. A slack variable, typically denoted by s_i or θ_j, is added to a "less-than-or-equal-to" ($\leq$) constraint to account for the difference between the left and right sides, thus creating an equality.

Consider a standard LP problem with inequality constraints:

$$\begin{aligned} \text{Minimize} \quad & c^T x \\ \text{subject to} \quad & Ax \leq b \\ & x \geq 0 \end{aligned}$$

To convert the inequality constraints $Ax \leq b$ into equalities, we introduce slack variables $s \geq 0$. The modified constraints become:

$$Ax + s = b$$

where $s = (s_1, s_2, \ldots, s_m)^T$ and each $s_i \geq 0$.

Example: The inequality constraint $2x_1 + 5x_2 \leq 10$ is converted to the equality $2x_1 + 5x_2 + s_1 = 10$, where $s_1 \geq 0$ is the slack variable.

In practical applications (like production planning), the value of a slack variable in the final optimal solution represents the amount of an *unused resource* or the "slack" in that constraint:

- **Binding Constraint:** If a slack variable is zero ($s_i = 0$), the constraint is binding (fully utilized), indicating that the corresponding resource is a bottleneck.
- **Non-binding Constraint:** If a slack variable is positive ($s_i > 0$), the constraint is non-binding (not fully utilized), meaning there is excess capacity.

Slack variables do not affect the objective function directly; they are simply auxiliary variables that help in maintaining the feasibility of the solution space and facilitate the application of the Simplex algorithm and other solution methods.

1.8 Big M Method

The **Big M Method** is a technique used in Linear Programming (LP) to handle constraints that involve artificial variables, particularly in problems where the initial basic feasible solution is not readily available. This method is especially useful in the context of the Simplex algorithm when dealing with equality constraints or "greater-than-or-equal-to" ($\geq$) constraints.

Concept

In the Big M Method, artificial variables are introduced to convert constraints into a form suitable for the Simplex algorithm. These artificial variables are penalized in the objective function by assigning them a very large coefficient, denoted as M. The idea is to ensure that these artificial variables are driven out of the solution during the optimization process, as their presence indicates infeasibility in the original problem.

Implementation

Consider a constraint of the form:

$$a_1x_1 + a_2x_2 \geq b$$

To convert this into an equality suitable for the Simplex method, we introduce an artificial variable $A \geq 0$:

$$a_1x_1 + a_2x_2 - s + A = b$$

where s is a surplus variable. The objective function is then modified to include the artificial variable with a large penalty:

$$\text{Minimize} \quad c^T x + M \cdot A$$

where M is a large positive number.

Purpose

The purpose of the Big M Method is to ensure that the artificial variables are minimized in the final solution. If the optimal solution includes any artificial variables with positive values, it indicates that the original problem is infeasible. Conversely, if all artificial variables are zero in the optimal solution, it confirms that a feasible solution to the original problem has been found.

Considerations

When implementing the Big M Method, it is crucial to choose a sufficiently large value for M to effectively penalize the artificial variables without causing numerical instability in the computations. Care must also be taken to interpret the results correctly, as the presence of artificial variables in the final solution can indicate infeasibility in the original LP problem.

1.9 Why Scaling Matters in Linear Programming

Link1 Link2

In Linear Programming (LP), **scaling** refers to the process of adjusting the coefficients of the objective function and constraints to improve the numerical stability and performance of optimization algorithms. Proper scaling can significantly enhance the efficiency of solvers like the Simplex method or Interior Point methods.

Importance of Scaling

Scaling is crucial for several reasons:

- **Numerical Stability:** Poorly scaled problems can lead to numerical instability during computations, resulting in inaccurate solutions or convergence issues.
- **Solver Performance:** Well-scaled problems often lead to faster convergence and reduced computational time, as the solver can more effectively navigate the solution space.
- **Conditioning of the Problem:** Scaling improves the condition number of the constraint matrix, which is a measure of how sensitive the solution is to changes in the input data.

The Reality of Floating-Point Arithmetic

A critical aspect of scaling that practitioners often overlook is that **real numbers aren't "real" in software**. All modern optimization solvers operate on IEEE-754 double-precision floating-point arithmetic, where:

- The expression `1 == 1 + 1e-16` can evaluate to `true`
- Addition is not strictly associative: $(a + b) + c \neq a + (b + c)$ in general
- Small rounding errors accumulate during matrix operations and factorizations

This is the fundamental terrain on which your optimizer operates. Default solver tolerances such as `FeasibilityTol = 1e-6` and `IntFeasTol = 1e-5` work well when coefficients, right-hand sides, and variable ranges exist in "human territory"—typically values between 10^{-3} and 10^{6}. Extremely tiny or huge coefficient ranges amplify floating-point limitations and can make nearly parallel constraint rows numerically toxic.

Techniques for Scaling

Common techniques for scaling LP problems include:

- **Row Scaling:** Adjusting the coefficients of each constraint so that the largest coefficient in each row is of similar magnitude.
- **Column Scaling:** Adjusting the coefficients of each variable so that the largest coefficient in each column is of similar magnitude.
- **Overall Scaling:** Applying a uniform scaling factor to all coefficients in the objective function and constraints.
- **Objective Function Scaling:** Normalizing the objective function coefficients to prevent domination by extremely large or small values.
- **Bound Scaling:** Adjusting variable bounds to maintain numerical consistency with constraint coefficients.

Commercial vs. Open-Source Solver Behavior

An important practical distinction exists between commercial and open-source solvers regarding automatic scaling:

- **Commercial Solvers** (e.g., Gurobi, CPLEX, FICO Xpress) typically include sophisticated automatic scaling and presolve routines that handle poorly scaled models transparently. They apply aggressive normalization techniques without requiring user intervention.
- **Open-Source Solvers** (e.g., HiGHS, GLPK, CLP) generally require more explicit user control of scaling parameters. Users must often manually specify scaling options such as `user_objective_scale` and `user_bound_scale` to achieve optimal performance.

For example, in a recent large-scale energy system optimization case (18.4M constraints, 13.1M variables, 41.9M non-zeros), the open-source HiGHS solver required explicit scaling parameter settings to converge, while commercial solvers handled the same poorly scaled model automatically. Specifically, HiGHS achieved convergence only after setting `user_objective_scale = -7` and `user_bound_scale = -12`, reducing solve time from over 10 hours (no convergence) to approximately 4 hours.[1]

This demonstrates that *a good scaling strategy benefits both open-source and commercial solvers*, but the burden of implementing it differs significantly.

Practical Considerations and Best Practices

When formulating LP models, consider the following guidelines:

- **Choose Sane Scales:** Express variables and constraints in consistent units to avoid large disparities in coefficient magnitudes. Aim for coefficient values in the range $[10^{-3}, 10^{6}]$.
- **Preserve Precision:** Avoid rounding input data unnecessarily. Writing 0.333 instead of a more precise decimal or the exact fraction can change presolve algebra and lead to different (and potentially incorrect) elimination paths. Keep full precision or clear denominators algebraically.
- **Use Symbolic Infinity Correctly:** When specifying unbounded variables, use the solver's built-in infinity constant (e.g., `GRB.INFINITY` $\approx 10^{100}$ in Gurobi). Avoid inventing custom large constants. Most solvers treat values above $\approx 10^{30}$ as infinite for bound purposes.
- **Monitor Coefficient Ranges:** Check the ratio between the largest and smallest non-zero coefficients in your constraint matrix. Ratios exceeding 10^{6} often indicate scaling problems. Many solvers report this information in log output.
- **Test Scaling Parameters:** When using open-source solvers, experiment with scaling flags and presolve options. For problematic instances, try:
  - Enabling/disabling automatic scaling (`ScaleFlag`)
  - Setting manual objective and bound scaling factors
  - Testing different presolve aggressiveness levels
  - Comparing simplex vs. barrier methods with different scaling
- **Reformulate When Possible:** The best scaling is often achieved through model reformulation rather than post-hoc numerical adjustments. Consider:
  - Changing variable units (e.g., measuring energy in MWh instead of Wh)
  - Normalizing constraint right-hand sides
  - Introducing auxiliary variables to reduce coefficient ranges

[1] LinkedIn discussion on solver benchmarking:

Diagnostic and Remedial Actions

When encountering numerical difficulties, systematically diagnose scaling issues:

1. **Review Solver Logs:** Look for warnings about ill-conditioning, large coefficient ranges, or numerical instability.
2. **Compute Matrix Statistics:** Calculate min/max absolute values of non-zero coefficients in A, b, and c. Large ratios (e.g., $> 10^9$) indicate severe scaling problems.
3. **Test Parameter Variations:** If using Gurobi or similar solvers, try:
   - `NumericFocus = 1, 2, or 3` for increased numerical caution
   - `ScaleFlag = 0, 1, 2, or 3` to test different scaling strategies
   - `ObjScale` to manually normalize the objective
   - Switching between barrier and simplex methods
4. **Check Tolerances:** Avoid blindly tightening feasibility or optimality tolerances, as this can worsen numerical problems. Instead, fix the scaling first.

Case Study: Large-Scale Energy System Model

A concrete example illustrates the practical impact of scaling. Consider a multi-year investment and operation EU power-system model coupled with industrial electrification sectors:

- **Problem Size:** 18.4M constraints, 13.1M variables, 41.9M non-zeros
- **Model Characteristics:** Poorly scaled; likely not meeting MIPLIP best-practice standards, but representative of realistic user-generated instances
- **Hardware:** 64 cores @ 3.6 GHz, 393 GB RAM

Results without proper scaling:

- HiGHS standard interior-point method: No convergence after 10 hours

Results with explicit scaling parameters:

- HiGHS v1.12.0 (HiPO solver, `user_objective_scale = -7`, `user_bound_scale = -12`): 4 hours
- FICO Xpress v9.7.0 (automatic scaling): 1 hour 40 minutes

The commercial solver remained faster but the gap narrowed significantly once scaling was addressed. More importantly, *without proper scaling, the open-source solver could not solve the problem at all.* This underscores that scaling is not merely an optimization for speed—it can determine whether a problem is solvable.

Conclusion

Proper scaling is a critical aspect of LP modeling that significantly impacts both the efficiency and reliability of the optimization process. While commercial solvers provide more forgiving automatic scaling, understanding and controlling scaling manually is essential when working with open-source tools or pushing the boundaries of problem size.

Numerics are part of the modeling craft. Respecting the geometry of the problem and the scales of coefficients allows solvers to deliver both speed and reliability. Ignoring these considerations invites silent numerical degradation that can compromise solution quality or prevent convergence entirely.

1.10 Integer Variable Tolerances and Floating-Point Reality

Link3

A common source of confusion for practitioners new to Mixed Integer Programming is observing integer variables returning values like 0.000005 instead of exactly 0, or binary variables showing 0.9999995 instead of exactly 1. This is not a solver bug—it is a fundamental consequence of how MIP solvers operate within floating-point arithmetic.

How Integer Variables Are Handled Internally

MIP solvers do not work in exact arithmetic. Instead:

- **Everything is continuous until proven otherwise:** Internally, all variables—including integer and binary variables—are represented as continuous floating-point numbers during linear relaxations and barrier method computations.
- **Integrality is enforced within tolerances:** Rather than requiring exact integer values (which would be numerically unstable or impossible in floating-point arithmetic), solvers enforce integrality within a small tolerance, typically `IntFeasTol = 1e-5` by default.
- **Near-integer values are acceptable:** If a variable's value is within the integrality tolerance of an integer, the solver treats it as satisfying the integer constraint. Thus, a value of 0.000005 is solver-speak for "this is effectively zero."

This design is *intentional*. Requiring exact equality in floating-point arithmetic would make many real-world models numerically unstable or impossible to solve, particularly when dealing with ill-conditioned matrices or poorly scaled problems.

The Danger of Naive Rounding

A critical mistake practitioners make is blindly rounding solver outputs without understanding tolerances. Consider this scenario:

1. The solver returns a binary variable $y = 0.9999995$ (within tolerance of 1).
2. The user rounds y to exactly 1 and uses this in downstream calculations or a subsequent optimization.
3. This rounding, combined with other constraint coefficients, may inadvertently violate constraints that were satisfied in the solver's original (toleranced) solution.
4. The user then blames the solver for producing an "infeasible" solution, when in fact the infeasibility was introduced by improper post-processing.

The real question is not "why isn't it exactly 0?" but rather "what tolerance is appropriate for my business decision?" If a decision changes because of a few microunits, the problem is not numerical precision—it is problem formulation.

The Trickle Flow Problem: Big M and Integer Tolerances

One of the most pernicious numerical issues in MIP arises from the interaction between **Big M constraints** and **integer tolerances**, commonly known as the "trickle flow" problem.

Consider a Big M formulation where a binary variable $y \in \{0, 1\}$ controls whether a continuous variable x can be positive:

$$x \leq M \cdot y$$

The intent is that when $y = 0$, we force $x = 0$. However, due to integer tolerances:

- The solver may return $y = 0.000005$ (within `IntFeasTol`).
- If M is very large (e.g., $M = 10^6$), then $x \leq 10^6 \times 0.000005 = 5$.
- The constraint allows x to take a small but non-zero value even when y is "effectively" zero.
- This "trickle flow" violates the intended logic of the constraint.

The problem is exacerbated when:

- M is unnecessarily large (poor Big M selection).
- The model is poorly scaled, leading to larger integrality violations.
- Multiple Big M constraints interact, compounding the trickle effects.

Mitigation Strategies for Integer Tolerance Issues

1. Use Indicator Constraints Instead of Big M: Modern solvers like Gurobi and CPLEX support indicator constraints, which enforce logical conditions without relying on large coefficients:

$$y = 0 \Longrightarrow x = 0$$

Indicator constraints avoid the numerical interaction between integrality tolerance and large M values, making them numerically superior to Big M formulations when supported.

2. Tighten Big M Values: Never use arbitrary large constants. Derive the smallest valid M through:

- **Analytical bounds:** Use domain knowledge to establish tight upper bounds on variables.
- **Optimization-based bound tightening:** Solve auxiliary problems like $\max x$ subject to all constraints to determine tight bounds programmatically.

3. Use the IntegralityFocus Parameter (Gurobi): Setting `IntegralityFocus = 1` instructs Gurobi to work harder to find solutions that remain nearly feasible when all integer variables are rounded to exact integers. This reduces trickle flow effects, though it introduces a modest performance penalty.

4. Adjust Integer Feasibility Tolerance (with caution): Reducing `IntFeasTol` (e.g., from 10^{-5} to 10^{-6}) can mitigate trickle flows but often at significant computational cost and with limited success. This should be a last resort after addressing scaling and formulation issues.

5. Post-Processing with Validation: If rounding integer variables for business reporting:

- Round conservatively (e.g., round 0.9999995 to 1 and 0.000005 to 0).
- **Validate the rounded solution:** Re-evaluate all constraints with the rounded values to ensure feasibility is maintained.
- If violations occur, identify whether they stem from poor scaling, formulation issues, or inappropriate rounding strategy.

Some practitioners advocate that solvers should automatically provide rounded integer solutions as part of their output. However, this raises questions:

- Should rounding be toward the nearest integer, or conservative (always rounding down for minimization, up for maximization)?
- Should the solver verify that the rounded solution is feasible, and if not, attempt to repair it?
- What if rounding materially changes the objective value?

These are application-dependent decisions. *In production systems, robustness beats mathematical purity every time.*

Exact Arithmetic Solvers

While most commercial and open-source MIP solvers operate in floating-point arithmetic, some solvers offer exact arithmetic modes:

- **SCIP (Version 10.0+):** Introduced a numerically exact solving mode for MILPs using rational arithmetic. This guarantees exact integer solutions but at significant computational cost, typically suitable only for smaller or specially structured problems.
- **Specialized exact solvers:** Some research-grade solvers use symbolic computation or rational arithmetic libraries, but these are rarely practical for large-scale industrial problems due to performance limitations.

For the vast majority of applications, understanding and managing floating-point tolerances is far more practical than insisting on exact arithmetic.

Key Takeaways

- **Floating-point arithmetic is unavoidable:** Modern MIP solvers operate in IEEE-754 double precision. Near-integer values like 0.000005 are normal and expected.
- **Tolerances are features, not bugs:** Integrality tolerances enable solvers to handle numerical challenges robustly. They reflect the reality of finite-precision computation.
- **Big M and tolerances interact dangerously:** Trickle flow problems arise when large M values multiply small tolerance violations. Prefer indicator constraints and tight M values.
- **Never round blindly:** If post-processing solver outputs, validate that rounded solutions remain feasible and preserve solution intent.
- **Focus on formulation quality:** Well-scaled, tightly formulated models with appropriate Big M values (or indicator constraints) experience fewer numerical issues than poorly formulated models, regardless of solver tolerance settings.

Understanding how solvers think about numbers makes you dramatically better at using them. The question is not whether floating-point arithmetic introduces small errors—it always does. The question is whether your model formulation and post-processing account for this reality appropriately.

1.11 Reading a Gurobi Solver Log in Linear Programming

A Gurobi solver log provides a detailed account of the optimization process, including information about the problem size, presolve reductions, and the performance of the chosen algorithm. When solving a Linear Programming (LP) problem, the log can be interpreted as follows:

- **Problem Size:** The log will indicate the number of constraints, variables, and non-zero coefficients in the model. This gives an initial sense of the problem's complexity.
- **Presolve Information:** Gurobi applies presolve techniques to simplify the model before solving. The log will show how many constraints and variables were removed or reduced during this phase, which can significantly impact solve time.
- **Algorithm Performance:** The log will detail the number of iterations taken by the Simplex method (or other algorithms) to reach optimality, as well as the time taken for each phase of the optimization process.
- **Optimal Solution:** Upon completion, the log will confirm whether an optimal solution was found and provide the objective value, along with any relevant information about the solution's feasibility and optimality.

By carefully analyzing the Gurobi log, practitioners can gain insights into the efficiency of their modeling choices and identify potential areas for improvement in future iterations.

Chapter 2

Concepts in Mixed Integer Programming

Binary Variables and Complexity in MIP

In Mixed Integer Programming (MIP), **binary variables** are decision variables that can take on only two possible values: 0 or 1. These variables are commonly used to model yes/no decisions, on/off states, or the inclusion/exclusion of certain options in optimization problems. The presence of binary variables significantly increases the complexity of MIP problems compared to standard Linear Programming (LP) problems, which involve only continuous variables.

Role of Binary Variables

Binary variables are essential for representing discrete choices in various applications, such as:

- Facility location decisions (e.g., whether to open a warehouse).
- Scheduling problems (e.g., whether to assign a task to a specific time slot).
- Capital budgeting (e.g., whether to invest in a project).

Impact on Complexity

The inclusion of binary variables transforms the problem from a continuous optimization problem into a combinatorial one. This change has several implications:

- **Exponential Growth of Solution Space:** The number of possible combinations of binary variables grows exponentially with the number of binary variables. For example, with k binary variables, there are 2^k possible combinations to evaluate.
- **NP-Hardness:** Many MIP problems are classified as NP-hard, meaning that there is no known polynomial-time algorithm to solve them optimally. This contrasts with LP problems, which can be solved in polynomial time using algorithms like the Simplex method or Interior Point methods.
- **Branch-and-Bound Algorithms:** To solve MIP problems, specialized algorithms such as branch-and-bound or branch-and-cut are employed. These algorithms systematically explore the solution space by branching on binary variables and bounding the objective function to prune suboptimal solutions.

Practical Considerations

When formulating MIP models, it is crucial to carefully consider the number and placement of binary variables to manage complexity. Techniques such as problem decomposition, cutting planes, and heuristics can be employed to improve solution times and handle larger problems effectively. Overall, binary variables are a powerful tool in MIP, enabling the modeling of complex decision-making scenarios while introducing significant computational challenges that require advanced solution techniques.

Big M Method in Mixed Integer Programming

The **Big M Method** is a widely used technique in Mixed Integer Programming (MIP) to handle constraints that involve both continuous and integer variables, particularly when dealing with logical conditions or disjunctive constraints. This method introduces a large constant, denoted as M, to effectively model these conditions within the MIP framework.

Concept

In MIP, the Big M Method is often employed to enforce logical relationships between variables. For example, it can be used to model situations where the activation of a binary variable (0 or 1) determines whether certain constraints are applied to continuous variables.

Implementation

Consider a scenario where we have a binary variable $y \in \{0, 1\}$ and a continuous variable $x \geq 0$. We want to impose the constraint that if $y = 1$, then x must be less than or equal to a certain value b. This can be modeled using the Big M Method as follows:

$$x \leq b + M(1 - y)$$

Here, when $y = 1$, the constraint reduces to $x \leq b$. When $y = 0$, the constraint becomes $x \leq b + M$, effectively removing the upper bound on x if M is sufficiently large.

Purpose

The purpose of the Big M Method in MIP is to create a flexible modeling framework that can accommodate complex logical conditions while maintaining the linear structure required for optimization. By using a large constant M, we can ensure that certain constraints are only active under specific conditions dictated by the binary variables.

Considerations

When implementing the Big M Method in MIP, it is crucial to choose an appropriate value for M. If M is too large, it can lead to numerical instability and poor performance of the solver. Conversely, if M is too small, it may not effectively relax the constraints when needed. This is even more applicable when the data is uncertain or subject to change. Therefore, careful consideration and domain knowledge are essential in selecting an appropriate value for M to ensure the model's effectiveness and solvability.

2.1 Indicator Constraints

Indicator constraints are a powerful feature in Mixed Integer Programming (MIP) that allow for the modeling of conditional relationships between binary and continuous variables. They provide a more intuitive and efficient way to express logical conditions compared to traditional methods like the Big M Method.

Concept

Indicator constraints enable the activation or deactivation of certain constraints based on the value of a binary variable. Specifically, an indicator constraint states that if a binary variable takes on a specific value (usually 0 or 1), then a corresponding constraint on continuous variables must hold.

Implementation

Consider a binary variable $y \in \{0, 1\}$ and a continuous variable $x \geq 0$. An indicator constraint can be formulated as follows:

$$\text{If } y = 1 \text{ then } x \leq b$$

This means that when y is equal to 1, the constraint $x \leq b$ must be satisfied. If y is equal to 0, the constraint on x is not enforced.

Purpose

The purpose of indicator constraints is to provide a clear and direct way to model conditional relationships in MIP problems. They enhance model readability and can lead to improved solver performance by avoiding the potential numerical issues associated with large constants in the Big M Method.

Considerations

When using indicator constraints, it is important to ensure that the MIP solver being used supports this feature, as not all solvers have built-in capabilities for indicator constraints. Additionally, while indicator constraints can simplify model formulation, they may also increase the complexity of the problem, so careful consideration should be given to their use in large-scale MIP models.

When to Use Indicator Constraints vs. Big M Method

Indicator constraints are generally preferred over the Big M Method when the MIP solver supports them, as they provide a more straightforward and numerically stable way to model conditional relationships. However, in cases where the solver does not support indicator constraints, or when the model requires more complex logical conditions that cannot be easily expressed with indicator constraints, the Big M Method may still be a viable alternative.

Cardinality Constraints and Matheuristics: A Practical Case Study

The Problem

Consider a continuous multi-commodity network flow problem with a complicating **cardinality constraint**. A cardinality constraint limits the number of variables that can take non-zero values in the solution. Such constraints frequently arise in portfolio optimization (limiting the number

of assets), network design (limiting the number of active routes), and facility location problems (restricting the number of open facilities).

For an in-depth treatment of linearization techniques for cardinality constraints, see Adam DeJans Jr.'s *The Linearization Handbook for MILP Optimization: Modeling Tricks and Patterns for Practitioners*.

The Challenge: LP to MILP Complexity Explosion

Without the cardinality constraint, the problem is a standard Linear Program (LP) that solves efficiently in seconds, even for large-scale instances. However, introducing the cardinality constraint transforms the problem into a Mixed Integer Linear Program (MILP), typically requiring binary variables to track which variables are active.

This transformation can cause a dramatic increase in computational time. In one real-world case, solve times exploded from **seconds to several hours** on the largest instances—far exceeding the business requirement of a maximum 1-hour solve time, with an ideal target of 30 minutes on average.

When Complex Methods Don't Help

Initial attempts to address the performance issues included:

- **Fine-tuning MILP parameters:** Adjusting solver settings yielded only marginal improvements.
- **Lagrangian Relaxation:** Early experiments were unsuccessful in achieving the desired performance.
- **Benders Decomposition:** This sophisticated decomposition method was on the backlog but wasn't pursued after a simpler approach proved successful.

The core lesson: the MILP formulation itself needed a fundamental rethink, not just parameter adjustments.

The Solution: A Matheuristic Approach

Based on three key observations:

1. The LP relaxation (without cardinality constraints) **scales extremely well**.
2. The optimal MILP solution **heavily overlaps** with the LP solution in terms of which variables are active.
3. **Pareto principle:** Most of the objective value comes from a small subset of high-impact variables.

The developed matheuristic works as follows:

1. **Solve the LP relaxation:** Quickly obtain a continuous solution without cardinality constraints.
2. **Extract high-impact variables:** Identify variables with significant values in the LP solution.
3. **Formulate a restricted MILP:** Build a smaller MILP that includes only the high-impact variables, along with the cardinality constraint.

4. **Solve the restricted MILP:** This smaller problem solves rapidly while maintaining solution quality.

Results

This matheuristic approach achieved:

- **Solve times:** A few seconds in the worst case (down from hours).
- **Solution quality:** High-quality solutions that meet business requirements.
- **Robustness:** Consistent performance across various instance sizes.

Alternative Advanced Methods

While the matheuristic proved sufficient, other sophisticated approaches exist for such problems:

- **Benders Decomposition:** Decomposes the problem into a master problem and subproblems. Modern solvers like CPLEX and SCIP support *automatic Benders decomposition*, which can significantly simplify implementation and experimentation.
- **Lagrangian Relaxation:** Relaxes complicating constraints by moving them into the objective function with Lagrangian multipliers, then solves iteratively.
- **Column Generation:** Particularly effective for problems with a large number of variables but where only a subset are expected to be active in the optimal solution.

Key Takeaway: Simple Can Beat Complex

In practical optimization, especially in business contexts with tight time constraints:

- **Start simple:** Begin with straightforward approaches that leverage problem structure.
- **Exploit LP efficiency:** When possible, use fast LP solves to guide MILP solutions.
- **Focus on business requirements:** A "good enough" solution delivered quickly often beats a perfect solution delivered too late.
- **Advanced methods have their place:** Reserve sophisticated techniques like Benders or Lagrangian methods for when simpler approaches genuinely fail.

The most elegant mathematical technique isn't always the most practical. Understanding *when* to use which approach is a crucial skill in operations research practice.

Chapter 3

Other Notes

3.1 Challenges in Large Scale Problems

Large-scale problems often present challenges that are not apparent in smaller or moderate-sized problems. Key areas to address include:

Data Preprocessing The post-processed data is directly used to generate the model. Efficient preprocessing of large datasets is crucial, as slower processing can significantly impact overall model run times. Understanding which Python libraries to use (e.g., `NumPy`, `Numba`, `Polars`) for optimal data handling is highly beneficial for performance.

Model Generation Time The time taken to generate models can directly influence the total execution time. Algebraic Modeling Languages (AMLs) and solver APIs offer multiple ways to model the same constraint; selecting the most efficient API call is a critical skill. Additionally, performing efficient reformulations—such as using helper variables to reduce model size or making constraint expressions more concise—is essential.

Model Solve Time When models take too long to solve, it is vital to diagnose why the solver is struggling to find an optimal solution. Consider the following strategies:

- Can the model be reformulated into a **tighter formulation**?
- Can the model be **linearized** to some extent if it is currently non-linear?
- Can the **ranges of model coefficients** be scaled down to improve numerical stability?
- Can **variable bounds** be made tighter or hard constraints be relaxed?
- Is the customer comfortable with an **allowable solution gap**?
- Can **solver parameters** be tuned based on the information displayed in the solver log?
- Can the solving methodology be supplemented with **heuristics**, Local Search, or greedy methods?

3.2 Importance of Indexes during Model Formulation

When formulating optimization models, especially in Mixed Integer Programming (MIP), the use of indexes is crucial for several reasons:

- **Clarity and Readability:** Indexes help in clearly defining the relationships between different sets of variables and constraints. They make the model more understandable, especially when dealing with multiple dimensions (e.g., time periods, locations, products).

- **Scalability:** Using indexes allows for the easy expansion of the model. If new elements need to be added (e.g., additional time periods or products), the model can be adjusted without significant restructuring.
- **Efficient Data Management:** Indexes facilitate the organization and retrieval of data. They help in mapping real-world entities to model variables, making it easier to manage large datasets.
- **Reduction of Errors:** Proper indexing reduces the likelihood of errors in model formulation. It ensures that constraints and variables are correctly aligned, preventing misinterpretations that could lead to infeasible or suboptimal solutions.
- **Enhanced Solver Performance:** Well-structured indexes can lead to more efficient model solving. They help the solver understand the problem structure better, potentially improving convergence times.

Overall, the thoughtful use of indexes is a best practice in optimization modeling that enhances both the development process and the performance of the resulting models.

3.3 GPU Acceleration in Linear and Mixed-Integer Programming

Link1

With the rapid advancement of GPU-accelerated computing power in recent years, the optimization community has increasingly explored how this hardware can improve the performance of algorithms used to solve Linear Programming (LP) and Mixed-Integer Programming (MIP) models. This section examines the current state of GPU utilization in optimization, focusing on which algorithmic components benefit from GPU acceleration and what practitioners should consider when evaluating GPU-based solving strategies.

The Fundamental Question: Where Do GPUs Fit?

To understand how GPUs can improve MIP algorithm performance, we must first address a more fundamental question: *How do GPUs improve the performance of LP algorithms?* This question is critical because solving LPs forms the backbone of modern MIP solvers. Every node in the Branch-and-Bound tree for a MIP requires solving an LP relaxation, and the root node LP relaxation often dominates the early phase of the MIP solve.

The performance of any solver on a particular model depends on the interplay between **algorithm capabilities** and **hardware characteristics**. Sophisticated optimization algorithms that can leverage powerful hardware—whether the latest GPUs or high-core-count CPUs—can solve problems that were previously intractable. However, not all algorithmic components are equally suited to GPU architectures.

LP Algorithms and GPU Suitability

Consider the primary algorithms for solving LPs:

- **Simplex Methods (Primal and Dual):** These algorithms traverse the vertices of the feasible polytope, performing pivot operations that involve sparse matrix updates and basis factorizations. Simplex methods are inherently sequential and rely heavily on irregular memory access patterns and conditional branching—characteristics that make them difficult to parallelize even on CPUs, and exponentially more challenging on GPUs.

- **Interior Point (Barrier) Methods:** These algorithms solve a sequence of linear systems derived from Karush-Kuhn-Tucker (KKT) conditions. The computational bottleneck is typically matrix factorization (e.g., Cholesky decomposition) and forward/backward substitution. While certain matrix factorization operations can be implemented on GPUs, the irregular sparsity patterns of real-world LP constraint matrices often limit the efficiency gains.
- **First-Order Methods (e.g., PDHG):** Primal-Dual Hybrid Gradient (PDHG) and related first-order methods are the most amenable to GPU acceleration. These algorithms rely primarily on *sparse matrix-vector multiplication*, which maps naturally to GPU architectures. Unlike simplex and barrier methods, first-order methods avoid complex matrix factorizations and irregular control flow, making them ideal candidates for massively parallel computation.

First-Order Methods on GPUs: Opportunities and Limitations

First-order methods like PDHG have received significant attention for GPU acceleration due to their algorithmic simplicity and suitability for parallel architectures. For certain large-scale LPs, PDHG running on a GPU can achieve runtime improvements compared to traditional simplex or barrier methods on CPUs.

However, practitioners must understand several important caveats:

1. **Crossover Requirement:** PDHG and other first-order methods typically produce *interior point solutions*, not *basic solutions*. For MIP solving, a basic solution is required as a starting point for the Branch-and-Bound search. Obtaining a basic solution requires a **Crossover** step—essentially a simplex-based phase that converts the interior point solution to a vertex of the feasible polytope.

   The time required for Crossover depends on the quality of the solution produced by PDHG. Most PDHG implementations terminate with lower solution quality (larger constraint violations and complementarity gaps) than barrier methods, which means Crossover often requires additional time. In some cases, the Crossover time can dominate the overall runtime, negating the gains from the GPU-accelerated PDHG phase.

2. **Convergence Tolerances:** PDHG requires careful tuning of convergence tolerances. Looser tolerances reduce PDHG runtime but increase Crossover time and may compromise solution quality. Tighter tolerances increase PDHG runtime but reduce Crossover requirements. Benchmarks that report only PDHG time (without Crossover) or use very loose tolerances can be misleading for practitioners evaluating end-to-end performance.

3. **Model Characteristics Matter:** PDHG on GPU is most effective for *very large, well-conditioned LPs* where the algorithm can achieve high-quality solutions efficiently. Models with poor conditioning, extreme coefficient ranges, or complex constraint structures may not benefit significantly, and traditional barrier or simplex methods may remain superior.

The Branch-and-Bound Framework for MIPs

Understanding GPU impact on MIP performance requires examining the Branch-and-Bound framework in detail. A simplified view includes these key components:

1. **Root Node LP Relaxation:** The initial LP solve at the root of the Branch-and-Bound tree. This is often the largest and most time-consuming single LP solve in the entire process.

2. **Presolve and Preprocessing:** Techniques that simplify the model by eliminating variables, tightening bounds, and removing redundant constraints before the main solve begins.

3. **Heuristics:** Algorithms that attempt to find high-quality feasible integer solutions quickly, providing good incumbent solutions that enable more aggressive pruning.

4. **Cut Generation:** The process of adding valid inequalities (cuts) that tighten the LP relaxation, improving bounds and reducing the size of the Branch-and-Bound tree.

5. **Node Selection and Branching:** Strategies for choosing which node to explore next and which variable to branch on, guiding the search through the solution space.

6. **Node Relaxation LPs:** The numerous smaller LPs solved at each node in the Branch-and-Bound tree to evaluate bounds and guide the search.

Where Can GPUs Help in MIP Solving?

Currently, GPU acceleration can contribute to MIP performance in specific areas:

Root Node LP Relaxation: For models where the root LP relaxation is extremely large and challenging, using a GPU-accelerated first-order method (plus Crossover) may reduce the time spent in this initial phase. This is most beneficial when the root LP time dominates the overall MIP solve time.

Example scenario: A MIP model with 10+ million variables and constraints where the root LP alone takes hours to solve. If PDHG on GPU (plus Crossover) can reduce this from 3 hours to 1 hour, and the total MIP time is 5 hours, the overall improvement is substantial (5 hours $\rightarrow$ 3 hours).

Selected Heuristics and Presolve Components: Some heuristics and presolve operations involve computations that are amenable to GPU acceleration, such as parallel feasibility pumps or certain matrix reduction operations. This remains an active area of research, and implementations are gradually becoming available in modern solvers.

However, several critical MIP components are *not currently suitable* for GPU acceleration:

Node Relaxation LPs: The LPs solved at Branch-and-Bound nodes are typically small and benefit enormously from **warm-starting**—reusing the optimal basis from the parent node. The dual simplex method excels at warm-starting and is deeply integrated into modern Branch-and-Bound frameworks. Switching to a first-order method (even on GPU) would sacrifice warm-starting advantages and likely require Crossover at every node, dramatically increasing overall runtime.

Cut Separation: Identifying violated inequalities involves complex logic, dynamic data structures, and irregular memory access—all characteristics poorly suited to GPU architectures.

Node Selection and Branching: These components require sophisticated heuristics, tree search logic, and frequent communication between different solver components. They are inherently sequential and control-flow intensive, making GPU acceleration impractical.

CPU-GPU Coupling: An Essential Consideration

It is crucial to understand that *no solver framework runs purely on GPU hardware.* The CPU orchestrates the overall solve, manages memory, handles control flow, and executes the many algorithmic components that are not GPU-enabled. Only specific computational kernels—primarily sparse matrix-vector multiplication in first-order methods—run on the GPU.

This means that effective GPU utilization requires careful orchestration of data movement between CPU and GPU memory. Excessive data transfer can create bottlenecks that negate computational speedups. Furthermore, if the GPU-accelerated components represent only a small fraction of the total runtime, the overall performance improvement will be modest.

For most MIP models today, the algorithmic components that dominate runtime—node selection, cut generation, heuristics, and node relaxation LPs—all run on the CPU. This is why many MIP models do not yet see significant performance gains from GPU acceleration.

When Does GPU Acceleration Make Sense?

Based on current algorithmic capabilities and hardware characteristics, GPU acceleration is most beneficial for:

- **Very Large LPs:** Models with millions of constraints and variables where traditional simplex or barrier methods struggle, and first-order methods can achieve acceptable solution quality efficiently.
- **MIPs with Challenging Root LP Relaxations:** Models where the root node LP dominates solve time (e.g., $> 50\%$ of total runtime), and the root LP is large enough to benefit from GPU-accelerated PDHG.
- **Non-Basic Solution Applications:** Rare cases where a basic solution is not required, and the interior point solution from PDHG alone is acceptable (avoiding Crossover entirely).

GPU acceleration is currently *not beneficial* for:

- **Most Standard MIPs:** Models where node relaxation LPs, cut generation, and heuristics dominate runtime. These components are not effectively GPU-accelerated with current technology.
- **Small to Medium LPs:** Models that solve quickly (seconds to minutes) with simplex or barrier methods. Overhead from data transfer and Crossover would likely eliminate any GPU gains.
- **Poorly Scaled or Ill-Conditioned Models:** First-order methods are sensitive to problem conditioning. Models with extreme coefficient ranges or numerical issues often perform poorly with PDHG, regardless of hardware.

Evaluating Performance: The Importance of Benchmarking

When evaluating GPU acceleration for your specific models, comprehensive benchmarking is essential. Be aware of the following considerations:

- **Measure End-to-End Time:** Always include Crossover time in benchmarks for MIP applications. Reporting only PDHG time is misleading if you require a basic solution.
- **Use Consistent Tolerances:** Ensure convergence tolerances are appropriate for your application. Comparing GPU results with loose tolerances to CPU results with tight tolerances is not meaningful.
- **Account for Data Transfer:** If your workflow involves loading models from disk or transferring data between CPU and GPU, include this time in your measurements.
- **Test Representative Instances:** Performance can vary dramatically across different model instances. Test on a representative sample of your problem class.

Future Outlook

GPU acceleration for optimization remains an active and exciting area of research. As algorithms continue to evolve and more components of the Branch-and-Bound framework become GPU-enabled, we can expect broader applicability of GPU acceleration for MIP solving.

Promising research directions include:

- **Concurrent Crossover Methods:** Developing parallel Crossover algorithms that can leverage GPU architectures to reduce the time required to obtain basic solutions from interior point methods.
- **GPU-Enabled Heuristics:** Implementing feasibility pumps, local search methods, and other primal heuristics that can exploit GPU parallelism.
- **Hybrid Barrier Methods:** Combining GPU-accelerated matrix operations with CPU-based symbolic factorization and other components to create barrier methods that effectively utilize both hardware types.
- **Dynamic Algorithm Selection:** Developing frameworks that automatically choose between CPU and GPU execution for different solver components based on model characteristics and runtime behavior.

Practical Recommendations

For practitioners considering GPU acceleration:

1. **Understand Your Bottleneck:** Profile your models to identify where solve time is spent. If root LP time dominates, GPU acceleration may help. If node LPs and cuts dominate, focus on algorithmic improvements (stronger formulations, better branching strategies) rather than hardware.
2. **Ensure Good Model Formulation:** No amount of hardware can compensate for a poorly formulated model. Prioritize tight formulations, proper scaling, and appropriate Big M values before investing in specialized hardware.
3. **Benchmark Your Specific Models:** Generic benchmarks may not reflect your problem characteristics. Test GPU acceleration on your actual models with realistic data and requirements.
4. **Consider Total Cost of Ownership:** GPU hardware is expensive and requires specific software support. Compare the cost of GPU infrastructure against the value of reduced solve times for your application.
5. **Stay Informed:** This field is evolving rapidly. Algorithmic advances may change which problems benefit from GPU acceleration. Monitor solver release notes and research literature for new developments.

Conclusion

GPU acceleration represents a valuable tool for solving certain classes of large-scale optimization problems, particularly giant LPs and MIPs with challenging root node relaxations. First-order methods like PDHG are the most mature GPU-accelerated LP algorithms currently available, but practitioners must carefully account for Crossover time and solution quality when evaluating overall performance.

For the majority of MIP models today, traditional CPU-based algorithms remain superior. The many algorithmic components that dominate MIP runtime—node relaxation LPs with warm-starting, cut generation, heuristics, and tree search logic—are not effectively accelerated by current GPU technology.

However, as research progresses and more solver components become GPU-enabled, the applicability of GPU acceleration will broaden. Understanding which problems benefit from GPU acceleration today and how the landscape is evolving will enable practitioners to make informed decisions about hardware investments and algorithm selection.

The interaction between algorithms and hardware is complex and model-dependent. There is no substitute for careful benchmarking on your specific problem instances when evaluating the potential benefits of GPU acceleration for your optimization applications.

Chapter 4

Concepts on Infeasibility Analysis

Irreducible Infeasible Set IIS

In optimization, particularly in linear programming (LP) and mixed-integer programming (MIP), an **Irreducible Infeasible Set (IIS)** is a minimal subset of constraints and variables that cannot be satisfied simultaneously, leading to infeasibility in the overall model. The IIS is "irreducible" because removing any constraint or variable from this set would make the remaining constraints feasible. Identifying an IIS is crucial for diagnosing and resolving infeasibility issues in optimization models. By pinpointing the specific constraints and variables that contribute to infeasibility, modelers can take targeted actions to modify or relax these elements, thereby restoring feasibility to the model. The process of finding an IIS typically involves the following steps:

- **Model Analysis:** The optimization solver analyzes the infeasible model to identify potential sources of infeasibility.
- **Constraint Examination:** The solver systematically examines subsets of constraints to determine which combinations lead to infeasibility.
- **Minimal Set Identification:** The solver identifies the smallest set of constraints and variables that cannot be satisfied together, forming the IIS.

Once an IIS is identified, modelers can use this information to:

- **Modify Constraints:** Adjust or relax the constraints in the IIS to restore feasibility.
- **Improve Model Formulation:** Reevaluate the model structure to ensure that it accurately represents the real-world problem.
- **Communicate Issues:** Provide clear feedback to stakeholders about the sources of infeasibility in the model.

Overall, the concept of an Irreducible Infeasible Set is a powerful tool for diagnosing and addressing infeasibility in optimization models, enabling more effective problem-solving and model refinement.

Feasibility Relaxation

Feasibility relaxation is a technique used in optimization to address infeasible models by allowing certain constraints to be relaxed or violated. The goal is to find a solution that minimizes the degree of infeasibility while still satisfying as many constraints as possible. In feasibility relaxation, the original constraints of the model are modified by introducing slack variables or penalty terms that

quantify the extent of constraint violations. The objective function is then adjusted to minimize these violations, often by assigning weights to the slack variables based on their importance. The process of feasibility relaxation typically involves the following steps:

- **Identify Infeasibility:** Determine which constraints are causing the model to be infeasible.
- **Introduce Slack Variables:** For each infeasible constraint, introduce slack variables that allow for controlled violations.
- **Adjust Objective Function:** Modify the objective function to include terms that penalize the slack variables, encouraging the solver to minimize constraint violations.
- **Solve Relaxed Model:** Use an optimization solver to find a solution to the relaxed model that balances feasibility and optimality.

Feasibility relaxation is particularly useful in scenarios where strict adherence to all constraints is not possible or practical. It allows modelers to explore trade-offs between constraint satisfaction and solution quality, providing insights into how to adjust the model or constraints to achieve a feasible solution. Overall, feasibility relaxation is a valuable technique in optimization that helps address infeasibility issues while maintaining the integrity of the original problem as much as possible.

Chapter 5

Next Work Items

5.1 Future Topics and Next Work

- KKT Conditions
- Reading Solver Logs in MIP
- Duality
- Sensitivity Analysis
- Bender's Decomposition
- Column Generation
- Dantzig-Wolfe Decomposition
- Lagrangian Relaxation
- Cutting Planes
- Integer Programming
- Network Flows
- SOS Constraints
- Stochastic Programming
- Weak and Strong MILP formulations
- LP Relaxation in MIP
- Heuristics in MIP
- Next Work: Continue writing on Big M Method and Indicator Constraints.

Bibliography

[1] Books of Shrii Shrii Anandamurti (Prabhat Ranjan Sarkar):
http://shop.anandamarga.org/

[2] Avtk. Ananda Mitra Ac., *The Spiritual Philosophy of Shrii Shrii Anandamurti: A Commentary on Ananda Sutram*, Ananda Marga Publications (1991)
ISBN: 81-7252-119-7