SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

# Tuning Linear Programming Solvers for Query Optimization

Sarra Ben Mohamed

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

# Tuning Linear Programming Solvers for Query Optimization

# Anpassung von Linear Programming Solvern für Anfrageoptimierung

| | |
|---|---|
| Author: | Sarra Ben Mohamed |
| Supervisor: | Prof. Dr. Thomas Neumann |
| Advisor: | Altan Birler |
| Submission Date: | 15/10/2023 |

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15/10/2023 Sarra Ben Mohamed

# Acknowledgments

For S and S.

# Abstract

The quality of cardinality estimates is critical for query optimization. Research has shown that cardinality estimation is the major root of many issues in query optimization, which is why pessimistic cardinality estimators, that anticipate worst-case outcomes, can be very useful. A way to estimate reliable upper bounds of query sizes is through linear programming (LP). Recognizing this, our project aims to implement and compare different methods suitable for solving small LP problems. We investigate packing LP problems taken from cardinality estimation benchmarks. This should guide us into constructing a thorough analysis of the particularities of these LP problems, what is unique about their structure and if their solution process follows any patterns. In analysing our findings, we aspire to present insights that could inform recommendations on the most efficient solution methods, optimizing for time performance.

# Contents

# 1 Introduction

In this chapter we motivate the purpose of our research and present an overview of its structure, highlighting the importance of reliable cardinality estimation in the field of query optimization. Our contribution's relevance lies in selecting the best method or combination of methods to solve hundreds of small LP problems efficiently, optimizing execution time and memory usage. The cardinality estimation problem can be solved using linear programming, so our research opens doors for a faster cardinality estimation, and, in turn, efficient query execution.

## 1.1 Motivation

The quest for optimizing query execution in databases is a critical challenge. The query optimizer attempts to determine the most efficient way to execute a given query by considering the possible query plans [Lei+18]. These plans are all equivalent in their final output but vary in their cost, that is, the amount of time that they need to run.

Cost-based query optimizers select query plans that have the lowest estimated cost to execute. One parameter that plays an important role in this selection process is the cardinality estimates of these plans. Therefore, improving cardinality estimation leads to better estimated costs and, in turn, faster execution plans [GM93].

Linear Programming (LP) is a mathematical modeling technique to calculate the optimum of a linear function given a set of linear constraints [Chv83]. Cardinality estimation can be represented through linear programming [AGM13].

Our aim with this project is to implement, investigate, and compare different methods suitable for solving hundreds of small LP problems. These methods include the standard simplex algorithm and the revised simplex algorithm.

We test our methods alongside Umbra's LP solver for cardinality estimation [NF20]. By running various benchmarks, we conduct an analysis of the structure of these LP problems, compare the time profiles for the different methods tested and finally make a recommendation on the methods or combination of methods that deliver the best time and memory complexity.

## 1.2 Chapter Overview

This thesis is structured as follows:

- **Chapter 2: Background and Related Work** - Provides a foundational understanding of linear programming, duality, and an overview of the different LP solvers. We also explore cardinality estimation and state-of-the-art LP solvers.
- **Chapter 3: Tuning Linear Programming Solvers for Query Optimization** - Introduces our proposal and implementation, including the hierarchy, tableau simplex solver, data structures, revised simplex solver, and stability considerations.
- **Chapter 4: Evaluation** - Presents our results as well as an in-depth analysis of our results, including the analysis of the query dataset, and the randomly generated LP problems.
- **Chapter 5: Conclusion** - Summarizes our findings and contributions. We also discuss the implications of our research. and outline potential future work. and discusses

Throughout this thesis, we aim to provide a comprehensive examination of the cardinality estimation problem and the methods to solve it using linear programming. Our ultimate goal is to contribute valuable insights that can enhance the query optimization process.

# 2 Background and Related work

In this chapter, we present the background and related work that forms the foundation for our research on solving small linear programming problems related to cardinality estimation. We will talk about optimization, in particular the field of linear programming. We elaborate on the most widely used algorithms and techniques to tackle this problem, and we present some use cases and benchmarks. A major use case of linear programming solvers is cardinality estimation, which is a crucial step in the pipeline of query optimization.

## 2.1 Linear Programming Fundamentals

In this section, we will present the background of linear programming.

### 2.1.1 Linear Programming Problem

Informally, linear programming (LP) is a method to calculate the best possible outcome from a given set of requirements. A concrete real-world application of such a method is for instance aiming to maximize profit in a business, given some constraints on variables like raw material availability, labor hours...

Formally, LP is a mathematical modeling technique in which a linear function (called the objective function) $z : \mathbb{R}^n \to \mathbb{R}$ is maximized or minimized when subject to a set of linear constraints or inequalities.

A maximization LP problem is then defined as:

$$\begin{aligned} \text{Maximize} \quad & z = \mathbf{c}^T\mathbf{x} \\ \text{subject to} \quad & \mathbf{A}\mathbf{x} \le \mathbf{b} \\ & \mathbf{x} \ge \mathbf{0} \end{aligned} \tag{2.1}$$

Where $n$ is the number of decision variables and $m$ is the number of constraints: $\mathbf{x} \in \mathbb{R}^n$ is the column vector of decision variables. $\mathbf{c} \in \mathbb{R}^n$ is the column vector of coefficients in the objective function. $\mathbf{A} \in \mathbb{R}^{m \times n}$ is the coefficient matrix in the constraints. $\mathbf{b} \in \mathbb{R}^m$ is the column vector of the right-hand sides of the constraints. In the following sections, we focus on LP problems that are maximization problems and we primarily use the matrix representation of the problem.

### 2.1.2 Packing LP

A packing LP problem is a type of LP problem where the decision variables are subject to non-negative constraints, and the objective is to maximize a linear function, meaning $A$ contains positive coefficients.

We also present a special instance where: $\mathbf{c} = \mathbf{b} = \begin{bmatrix} 1 & 1 & \dots & 1 \end{bmatrix}$. Our specific problem is then expressed as follows:

$$\begin{aligned} &\text{Maximize} \quad \sum_{i=1}^{n} x_j \\ &\text{subject to} \\ &\qquad \mathbf{Ax} \leq \mathbf{1}_m, \\ &\qquad x_i \geq 0, \qquad\qquad i = 1, \dots, n \end{aligned}$$

Where $\mathbf{1}_m = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}$ This type of LP problem is relevant to our experiments.

### 2.1.3 Duality

The duality theorem is an interesting result in linear programming, that states that every instance of maximization problem has a corresponding minimization problem called its dual problem [Chv83]. It is also a measure of how close we are to arriving at the optimum. The two problems are linked in an interesting way: if one problem has an optimal solution, then so does the other, and their optimal objective function values are equal.

For instance, consider the primal-dual LP pair:

$$\begin{array}{llcll} \text{maximize} & z = \mathbf{c}^T\mathbf{x} & & \text{minimize} & z' = \mathbf{b}^T\mathbf{y} \\ \text{subject to} & \mathbf{Ax} \leq \mathbf{b} & \longrightarrow & \text{subject to} & \mathbf{A}^T\mathbf{y} \geq \mathbf{c} \\ & \mathbf{x} \geq 0 & & & \mathbf{y} \geq 0 \end{array}$$

### 2.1.4 Geometric Interpretation

In this part, we assume for simplicity that we have two decision variables in our LP problem, i.e. $\mathbf{x} \in \mathbb{R}^2$ is a two-dimensional vector. This assumption will allow us to plot our problem on a 2D plane. An example is shown in Figure 2.1.4. The LP problem 2.1 can be understood geometrically as follows: each inequality in the set of constraints

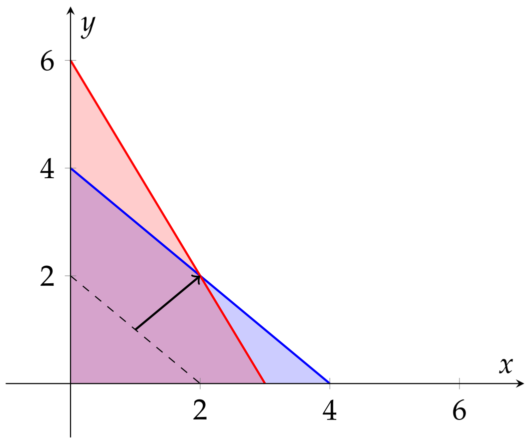


Figure 2.1: Feasible region of the LP problem, the area (purple) delimited by the constraint lines, blue and red. The dashed black line represents the objective function.

is represented by a line. Therefore, the feasible region $\chi$ of any LP problem can be described as the intersection delimited by those lines, which is called a polyhedron, or in our 2D case, a polytope.

A feasible vertex is a point at which, when substituted into the constraints, they are satisfied. The corners of the polytope are called vertices. They lie in the intersection of at most $n$ constraints. If a vertex lies in the intersection of more than $n$ lines, it is called degenerate.

The simplex algorithm that we will describe later starts at an initial feasible vertex and moves along the edges of the polytope to vertices with better objective values until the optimal vertex is reached.

### 2.1.5 Feasibility, Unboundedness

We described how in geometric terms, an LP problem is feasible if the polytope defining its feasible region $\chi$ is not empty. In other words, the LP is infeasible if no feasible solution exists, or $\chi = \emptyset$.

The LP is said to have an unbounded set of solutions if its solution can be made infinitely large without violating any of its constraints. Meaning its feasible region is infinite, or unbounded. An example is shown in Figure 2.2.

The LP has an optimal solution, if there is a vertex $\mathbf{x}^*$, called the optimal vertex, such that $\mathbf{c}^T\mathbf{x} \leq \mathbf{c}^T\mathbf{x}^*$ for all $\mathbf{x} \in \chi$.

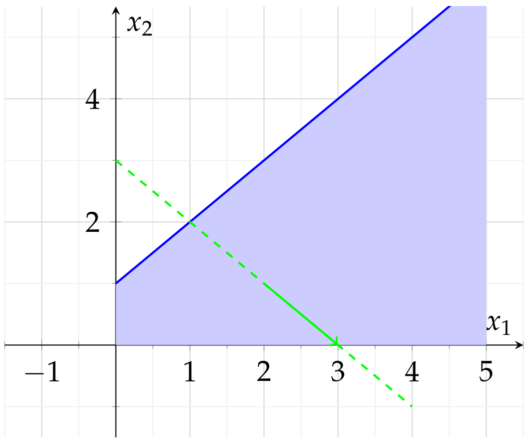


Figure 2.2: Graphical representation of an unbounded LP problem.

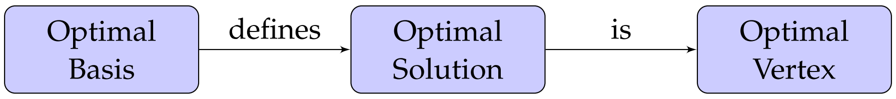


Figure 2.3: Conceptual equivalence between an optimal basis, an optimal solution, and an optimal vertex.

### 2.1.6 The Standard Simplex Algorithm

In the following, we will present the most widely used algorithm for solving LP problems, the simplex algorithm, as introduced by George Dantzig in 1947 [Dan90].

#### Computational form of LP

To be approachable by the simplex algorithm, the LP problem 2.1 needs to be cast in a computational or standard form 2.2, that fulfills the requirement of the constraint matrix having to have full row rank and only equality constraints are allowed. To convert the inequalities to equations, we introduce slack variables $s_1, s_2, \dots, s_m$. After introducing those variables, we look at the problem 2.1, where the constraints are now linear equalities:

$$\begin{aligned}
\text{Maximize} \quad & z = \mathbf{c}^T\mathbf{x} \\
\text{subject to} \quad & \sum_{j=1}^{n} a_{1j}x_j + s_1 = b_1 \\
& \sum_{j=1}^{n} a_{2j}x_j + s_2 = b_2 \\
& \vdots \\
& \sum_{j=1}^{n} a_{mj}x_j + s_m = b_m \\
& x_1, x_2, \ldots, x_n, s_1, s_2, \ldots, s_m \geq 0
\end{aligned} \tag{2.2}$$

We then have an LP problem in the appropriate form which can be used as input for the simplex algorithm. To develop an intuition for how this algorithm works, it is helpful to view the strategy of the simplex algorithm as that of successive improvements until reaching an optimum [Chv83]. For instance, a maximization problem is optimized when the slack variables are "squeezed out," maximizing the true variables' effect on the objective function.

### Defining a Basis

So far we have used the terms optimal vertex and solution interchangeably to refer to the point that delivers the optimum of an LP problem if plugged in the objective function. Now we will explore the concept of feasible and optimal basis. All the concepts are however equivalent, see 2.3. Yet the simplex algorithm uses the basis terminology.

In linear algebra, a basis of a vector space is a set of vectors that:

- Spans the space: this means that every vector in the vector space can be expressed as a linear combination of the basis vectors.
- Is linearly independent: this means that no vector in the basis can be expressed as a linear combination of the other basis vectors.

If an LP problem has an optimal solution, then it is enough to search the finite number of vertices of the feasible region. Every vertex, or corner point of the feasible region has at least one set of $m$ linearly independent columns of $\mathbf{A}$ associated with it. The set of indices of these columns is called a basis.

Basic variables are the indices contained in a basis. A feasible basis is a basis for which all the basic variables have non-negative values when the other variables (non-basic variables) are set to zero in the constraints.

### The algorithm

As we have seen, slack variables are introduced to convert inequalities into equations, and these slack variables can form an initial feasible basis. In this case, the decision variables are set to zero, and the slack variables are set to the values on the right-hand side of the constraints. This provides a starting point for the simplex method.

If the right-hand side is a vector of nonnegative numbers, i.e. $\mathbf{b} \geq 0$, then the problem is said to be initially feasible.

**Algorithm 1** Simplex Algorithm

```
procedure SIMPLEX($\mathbf{c}, \mathbf{A}, \mathbf{b}$)
    Initialize a feasible basic solution
    while true do
        Search for an entering variable (pricing)
        if no entering variable exists then
            return "Optimal solution found"
        end if
        Search for a leaving variable using the minimum ratio test
        if no positive pivot element in the column then
            return "Unbounded"
        end if
        Perform the pivot operation
        Update the basic and non-basic variables
    end while
end procedure
```

Let's present the methods used to perform the exchange in each step, i.e. the choice of the entering variable and the choice of the leaving variable.

- Pricing: we choose a non-basic variable to enter the basis and thus become basic. The choice usually depends on metrics like the largest increase in the objective function, or the largest coefficient. Bland's rule has been proven to guarantee termination [Chv83], and it states as follows: Choose the entering variable as the non-basic variable with the smallest index that has a negative reduced cost for a maximization problem:
$$j = \min\{j : c_j < 0\}$$

If there are no candidates for entering variable, we have reached the optimum.

- Ratio test: we choose a basic variable to leave the basis. we do this by performing a ratio test. For the chosen entering variable, compute the ratios for all positive values in its column:
$$\text{ratio}_i = \frac{b_i}{a_{ij}}$$
Choose the leaving variable as the basic variable has the smallest non-negative ratio. If there are no positive values in entering column, the problem is unbounded.

After selecting an entering variable and a leaving variable, we perform pivoting or updating the Tableau. This is done through row operations [Chv83].

**Termination**

The simplex algorithm can terminate in multiple ways.

1. *Optimality:* The algorithm terminates when there are no more negative coefficients in the objective function row for a maximization problem (or no more candidates for entering variables).
2. *Unboundedness:* If all the entries in the column of the entering variable are non-positive during the pivot operations, the problem is unbounded.
3. *Cycling:* The algorithm might enter a cycle, revisiting the same basic feasible solutions. Bland's rule can prevent this.
4. *Maximum Iterations:* A set maximum number of iterations can be used to prevent indefinite running due to numerical issues or other unforeseen circumstances. We set a bound on the number of steps in our implementations. This bound is equal to the maximum value a 32-bit unsigned integer can take, or 4294967295.

**The runtime complexity of the simplex algorithm**

Since the idea of the simplex algorithm is to search the finite number of bases until a basis is found that belongs to the optimal vertex, and there are $\binom{m}{n}$ bases, this might take an exponential number of steps. In fact, Klee and Minty (1972) [KM72] constructed a worst-case example where $2^m - 1$ iterations may be required, making the simplex' worst-case time complexity exponential, which we denote by $O(2^m)$.

It can be however argued that this is only one worst-case example. Indeed, the number of iterations usually encountered in practice is much lower. With $m < 50$ and $m + n < 200$, where $m$ and $n$ are the number of constraints and variables in the LP problem respectively, Dantzig [Dan90] observed that the number of iterations is usually less than $3m/2$ and only rarely going to $3m$.

However, there is no proof that for every problem the simplex algorithm for linear programming has a number of iterations or pivots that is majorized by a polynomial function.

For packing linear programs, the worst-case time complexity of the simplex algorithm remains exponential, though there exist polynomial implementations in practice [Sti10].

#### Time complexity analysis of one step

Given a linear program with $m$ constraints and $n$ variables, the Tableau for the simplex algorithm will be of size $(m+1) \times (n+m+1)$. The time complexity of one iteration of the Tableau simplex algorithm can be broken down as follows:

- Identifying the entering variable (choosing from $m$ non-basic variables): $O(m)$
- Identifying the leaving variable: $O(n)$
- Pivoting operation: $O(m \times n)$

Thus, the overall time complexity of one iteration is $O(m \times n)$. This is considered large, especially knowing that there may be an exponential number of iterations. This is why we proceed to define other solution methods that succeeded in achieving a better time performance.

### 2.1.7 The Interior Point Method

The interior point method is another mathematical technique used to solve LP problems. Practically, this method is based on the idea of iteratively moving through the interior of the feasible region of the LP problem [Ali95]. Mathematically, it can be described as follows:

Consider an LP problem in standard form:

$$\begin{aligned} \text{minimize:} \quad z =& \mathbf{c}^T\mathbf{x} \\ \text{Subject to:} \quad & \mathbf{A}\mathbf{x} = \mathbf{b} \\ & \mathbf{x} \geq 0 \end{aligned}$$

The interior point method introduces a penalty function to the objective function, which discourages solutions near the boundary of the feasible region. This barrier function is typically defined as $-\sum \log(x_i)$, where $x_i$ represent the elements of the vector $x$. The optimization problem then becomes:

$$\text{Minimize:} \quad c^T x - \mu \sum \log(x_i)$$

The key mathematical insight of the interior point method is that as $\mu$ approaches zero, the solution converges toward the optimal solution of the original LP problem. The method iteratively updates $\mu$ and the solution vector $\mathbf{x}$ until a suitable solution within the interior of the feasible region is found.

**Time Complexity Analysis**

The time complexity of the interior point method for linear programming is determined by the number of iterations required for convergence. In practice, the number of iterations is often polynomial in the problem size, which includes the number of decision variables $n$ and constraints $m$. Each iteration involves solving a system of linear equations, for example, $Hd = -(c + A^T\lambda)$. A mathematical derivation of this equation can be found in literature [Ali95].

Where:

- $H$ is a symmetric positive definite matrix.
- $d$ is the Newton direction.
- $c$ is the vector of coefficients representing the objective function.
- $A$ is the constraint matrix.
- $\lambda$ is the vector of dual variables.

Solving this system can be done efficiently, especially if $H$ and $A$ are sparse (e.g. using Cholesky, LU decomposition [GV13]). We can express the overall time complexity of the interior point method as follows:

$$O(k \cdot (f(n,m) + g(n)))$$

Where:

- $m$ and $n$ represent the size of the LP
- $k$ is the number of iterations.
- $f(n,m)$ represents the cost per iteration, which is polynomial in $n$ and $m$.
- $g(n)$ represents any additional preprocessing or initialization costs, which are typically polynomial in $n$.

Therefore, the overall time complexity of the interior point method is polynomial $n$ and $m$, the number of variables and constraints in the LP, respectively. This is better than the (worst-case) exponential time complexity of the simplex algorithm.

### 2.1.8 Revised Simplex Algorithm

The standard or Tableau simplex algorithm maintains and updates the entire Tableau in its dense form at each iteration, and the pivoting step of this algorithm is highly costly as we have to update the entire matrix using row operations. The revised simplex method transforms only the inverse of the basis matrix, $\mathbf{B}^{-1}$, thus reducing the amount of writing at each step and overall memory usage.

#### The algorithm

Let's derive the mathematical proof of this algorithm: Given a linear programming problem in standard form:

$$\begin{aligned} \text{maximize} \quad & \mathbf{c}^T\mathbf{x} \\ \text{subject to} \quad & \mathbf{A}\mathbf{x} = \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{aligned}$$

where $\mathbf{A}$ is an $m \times n$ matrix, $\mathbf{b}$ is an $m \times 1$ vector, and $\mathbf{c}$ is an $n \times 1$ vector.

Partition $\mathbf{x}$ into basic ($\mathbf{x}_B$) and non-basic ($\mathbf{x}_N$) variables. As mentioned before, basic variables are the indices contained in a basis. A feasible basis is a basis for which all the basic variables have non-negative values when the other variables (non-basic variables) are set to zero in the constraints. Similarly, partition $\mathbf{A}$ into $\mathbf{B}$ (columns corresponding to $\mathbf{x}_B$) and $\mathbf{N}$ (columns corresponding to $\mathbf{x}_N$).

The constraints can be written as:

$$\begin{aligned} \mathbf{B}\mathbf{x}_B + \mathbf{N}\mathbf{x}_N &= \mathbf{b} \\ \mathbf{x}_B &\geq 0 \\ \mathbf{x}_N &\geq 0 \end{aligned}$$

From $\mathbf{B}\mathbf{x}_B + \mathbf{N}\mathbf{x}_N = \mathbf{b}$, when $\mathbf{x}_N = 0$:

$$\mathbf{x}_B = \mathbf{B}^{-1}\mathbf{b}$$

This is the basic feasible solution if all entries of $\mathbf{x}_B$ are non-negative.

Compute the reduced costs:

$$\bar{\mathbf{c}}_N^T = \mathbf{c}_N^T - \mathbf{c}_B^T\mathbf{B}^{-1}\mathbf{N}$$

If all entries of $\bar{\mathbf{c}}_N^T$ are non-negative, then the current basic feasible solution is optimal.

If some entries of $\bar{\mathbf{c}}_N^T$ are negative, choose $j$ such that $\bar{c}_j < 0$. Compute:

$$\mathbf{d} = \mathbf{B}^{-1}\mathbf{A}_j$$

If all entries of $\mathbf{d}$ are non-positive, the problem is unbounded.

Otherwise, compute the step length:

$$\theta = \min \left\{ \frac{\mathbf{x}_B[i]}{\mathbf{d}[i]} : \mathbf{d}[i] > 0 \right\}$$

Update the solution:

$$\mathbf{x}_B = \mathbf{x}_B - \theta\mathbf{d}$$

$$x_j = \theta$$

and update the sets of basic and non-basic variables.

This proof elucidates that the only variables required to "recreate" exactly the Tableau at each step, without performing the costly pivoting operation are:

- the indices of basic and non-basic variables.
- $\mathbf{B}^{-1}$ the inverse of the basis matrix. This is used to solve two types of linear equations during an iteration, see steps 1 and 3 in Algorithm 2. In the actual implementation, there is no need to compute or store the $\mathbf{B}^{-1}$. We just use the basis matrix to solve these linear systems of equations. Therefore in practice, we only store $\mathbf{B}$.
- the current values of the basic variables, or the current basic feasible solution. $\mathbf{x}_B = \mathbf{B}^{-1}\mathbf{b}$

The algorithm is elaborated in Algorithm 2. As we can see, step 1 and 3 represent the solving of two types of systems, Forward Transformation (FTRAN) and Backward Transformation (BTRAN).

However, having to recompute the inverse of a matrix, or in our actual implementation, solve a linear equation system may be costly.

### Inverting a square matrix

For a square matrix of size $n \times n$, the time complexity of LU decomposition [GV13], which is one of the most prominent methods to invert a matrix is $O(n^3)$.

This is why it is desirable to use another tool to efficiently update the inverse of the basis matrix $\mathbf{B}^{-1}$ without having to recompute it from scratch each time. Such tools are called update methods, and we will later describe the Product Form of the Inverse (PFI), the Middle Product Form of the Inverse (MPFI) and we also describe the Forrest-Tomlin update method in the future work section.

**Algorithm 2** Revised Simplex Algorithm [Chv83]

```
1. Input: A feasible basic solution, $\mathbf{B}$, $\mathbf{c}$, $\mathbf{A}$, and $\mathbf{b}$
2. Output: Optimal solution or a certificate of unboundedness
3. Initialize $\mathbf{B}^{-1}$, the inverse of the basis matrix $\mathbf{B}$
4. While True:
        Step 1: Solve the system $y\mathbf{B} = \mathbf{c}_B$ (BTRAN)
        Step 2: Choose an entering column. This may be any column $\mathbf{a}$ of $\mathbf{A}_N$ such
         that $\mathbf{ya}$ is less than the corresponding component of $\mathbf{c}_N$. If there is no such
         column, then the current solution is optimal, Break.
        Step 3: Solve the system $\mathbf{Bd} = \mathbf{a}$ (FTRAN)
        Step 4: Let $\mathbf{x}_B^* = \mathbf{B}^{-1}\mathbf{b}$ the current basic variables' values. Find the largest $t$
         such that $\mathbf{x}_B^* - t\mathbf{d} \geq 0$ if there is no such $t$, then the problem is unbounded,
         Break; otherwise, at least one component of $\mathbf{x}_B^* - t\mathbf{d}$ equals zero and the
         corresponding variable is leaving the basis.
        Step 5: Set the value of the entering variable at $t$ and replace the values $\mathbf{x}_B^*$
         of the basic variables by $\mathbf{x}_B^* - t\mathbf{d}$. Replace the leaving column of $\mathbf{B}$ by the
         entering column, and in the basis heading, replace the leaving variable by
         the entering variable.
5. Return Optimal solution $\mathbf{B}^{-1}\mathbf{b}$
```

**The product form inverse update method**

We will discuss the Product Form of the Inverse (PFI), introduced by George Dantzig [DO54]. In the revised simplex method, there is a need to represent the inverse of the basis matrix and efficiently solve the BTRAN and FTRAN systems in Steps 1 and 3 of Algorithm 2 in each iteration to find the entering and leaving variables.

To avoid the costly reinversion of the basis matrix in each step, we apply an INVERT operation only once at the beginning, on the initial basis matrix $\mathbf{B}_0$. Inverting a matrix using LU decomposition requires $O(n^3)$ operations.

If $\mathbf{B}_k$ denotes the basis matrix after $k$ iterations of the simplex method, and each $\mathbf{B}_k$ differs from the preceding $\mathbf{B}_{k-1}$ in only one column, with index $p$, the entering index, we have the following update equation:

$$\mathbf{B}_k = \mathbf{B}_{k-1}\mathbf{E}_k$$

with $\mathbf{E}_k$ representing the identity matrix whose $p$th column is replaced by $\mathbf{d}$, and is referred to as an *eta matrix*. Consequently, when the initial basis consists of slack variables, we have $\mathbf{B}_0 = \mathbf{I}$, and successive applications of Equation 2.1.8 yield:

$$\mathbf{B}_k = \mathbf{E}_1\mathbf{E}_2 \cdot \mathbf{E}_k$$

This eta factorization of $\mathbf{B}$ provides an optimized way to solve the two systems.

Solving the system $y\mathbf{B}_k = \mathbf{c}_B$ comes down to solving:

$$(((y\mathbf{E}_1)\mathbf{E}_2)\dots)\mathbf{E}_k = \mathbf{c}_B$$

Solving the system $\mathbf{B}_k d = \mathbf{a}$ comes down to solving:

$$\mathbf{E}_1(\mathbf{E}_2(\dots(\mathbf{E}_k\mathbf{d}))) = \mathbf{a}$$

**Time complexity of direct solution:** Solving the system $y\mathbf{B}_k = \mathbf{c}_B$ or $\mathbf{B}_k\mathbf{d} = \mathbf{a}$ directly using Gaussian elimination or LU decomposition has a time complexity of $O(m^3)$, where $m$ is the dimension of $\mathbf{B}_k$, i.e. the number of constraints in the LP.

**Time complexity using eta factorization:** Solving $\mathbf{C}_i x = y$ or $\mathbf{x}\mathbf{E}_i = y$ involves simple vector operations and matrix multiplication, which has a time complexity of $O(n^2)$. Therefore, applying $\mathbf{B}_k = \mathbf{E}_1\mathbf{E}_2\dots\mathbf{E}_k$ takes a time complexity of $O(kn^2)$, where $k$ is the number of eta vectors, or the current iteration count.

**Comparing the Two Methods:**

- Direct Solution: $O(m^3)$ for both systems.
- Using Eta Factorization: $O(km^2)$ for both systems, where $k$ is the number of iterations.

In summary, when using the eta factorization method, the time complexity for solving both systems becomes $O(km^2)$, which depends on the number of iterations ($k$) and the number of constraints ($m$).

**The Middle Product Form of the Inverse**

The Middle Product Form of the Inverse (MPFI) update is a variant of the PFI that integrates updates in product form into the middle of factors L and U. This update is described mathematically in literature [HH15], and has been proven empirically to have a better performance than PFI and other PFI variants. This is also the update method used in Umbra's cardinality estimator [**Umbra**].

## 2.2 Cardinality Estimation

An important use case of linear programming solvers in the field of database systems is cardinality estimation. In the context of query optimization, LP solvers can be useful in estimating query plan cardinalities and providing a reliable and good enough estimate to be used in selecting the best join order, and hence speeding up query execution time [Lei+18].

Modern DBMS often rely on cost-based query optimizers to make this selection. For example, the SQL Server Query Optimizer [Mic23] employs a cost-based approach, aiming to minimize the estimated processing cost of executing a query.

Cardinality is defined as the number of tuples in the output. Enhanced cardinality estimation can lead to more accurate cost models, which in turn results in more efficient query execution plans. Consequently, accurate and reliable cardinality estimates are crucial in achieving faster query execution times. The objective is to develop a LP solver designed specifically for cardinality estimation. This solver aims to maximize a cost function that represents the upper bound of the output size, optimizing for both time and memory complexity. We assume the worst-case join sizes and want to see how large our variable domains can be given this worst-case scenario.

To set the stage for our implementation, we introduce the AGM bound.

### 2.2.1 AGM bound

The AGM bound [AGM13] proves using entropy that

$$\min_{w} \left( \sum_{i=1}^{k} w_i \log |R_i| \right)$$

is a tight upper bound for join size, given the query graph (how the relations are connected if there are any shared attributes) and relation sizes. The AGM bound is tight; in other words, we can always find a database instance linked to this join size upper bound. The dual LP problem of the given minimization problem is

$$\max \sum_i v_i$$

subject to:

$$A^T \mathbf{v} \leq \log |R|$$

The dual theorem 2.1.3 states that both problems have the same optimal values.

This is how our query LP datasets are generated. Umbra cardinality estimation utilizes pessimistic estimates, that guarantee that the cardinalities will not exceed that bound.

To elucidate the core concepts, suppose we have two relation $R$ and $S$ with attributes

$$Q(a,b,c) = R(a,b) \bowtie S(b,c)$$

where we denote the sizes of the relations as $|R|$ and $|S|$ respectively. It is easy to see that the largest possible output is $|R| \cdot |S|$, which occurs when the join behaves like a cartesian product, i.e. it has a selectivity equal to 1. So, this is the worst-case upper bound.

Deriving a simple example for an LP problem related to cardinality estimation leads us to the inequality 2.3. Applying the natural logarithm to both sides yields 2.4. We then rename the variables, simplifying the inequality to 2.5. Normalizing by dividing both sides by $r'$, we obtain 2.6. This leads us to the objective function for our packing LP problem.

$$|a| \cdot |b| \leq |R| \tag{2.3}$$

$$\ln |a| + \ln |b| \leq \ln |R| \tag{2.4}$$

$$a' + b' \leq r' \tag{2.5}$$

$$\frac{1}{r'}a' + \frac{1}{r'}b' \leq 1 \tag{2.6}$$

$$\text{maximize } a' + b' + c' + d' \quad \text{s.t.} \quad \frac{1}{r'}a' + \frac{1}{r'}b' \leq 1 \tag{2.7}$$

In this simple abstracted way, we get a sample packing LP from our dataset, where the decision variables would be the logarithms of the domains sizes of our variables, Our objective is to maximize the sum of our variables, and the constraints are the ones imposed by the AGM bound.

## 2.3 State-of-the-art LP solvers

Here we will discuss alternative approaches that are used today to solve LPs. Nowadays, the best existing open-source and commercial LP systems are based on implementations that make use of various algorithms and techniques to speed up the solution of large-scale LP problems. There exist various LP solvers today, like CPLEX (IBM), CBC, Gurobi, and HiGHS... We present HiGHS Scipy, and Cplex, because we are using them as a comparison to our solvers.

### 2.3.1 HiGHS Scipy

HiGHS, or High Performance Optimization Software is a software used to define, modify and solve large-scale sparse linear optimization models.

For LPs, HiGHS has implementations of both the revised simplex and interior point methods [HH18]. HiGHS has primal and dual revised simplex solvers, originally written by Qi Huangfu and further developed by Julian Hall. In our comparison, we do not make use of HiGHS directly but through Scipy's linprog, and using the solver as a method. It is mainly used to verify the accuracy of the optimal values obtained by our solvers.

```
linprog(method='highs')
```

### 2.3.2 Cplex

IBM ILOG CPLEX Optimization Studio, commonly known as CPLEX, is a software suite for mathematical optimization [Cpl09]. CPLEX consists of a library for linear programming, mixed integer programming, quadratic programming, and other optimization problems.

CPLEX utilizes a variety of algorithms to tackle linear programming (LP) problems. Although the majority of LP instances are efficiently solved using CPLEX's cutting-edge dual simplex algorithm, there are cases where other algorithms might be more suitable. For certain problem types, the primal simplex algorithm and the interior point method.

Furthermore, CPLEX provides a concurrent option, which runs multiple algorithms in parallel, returning the solution from the first algorithm to complete. It also supports various interfaces, including C++, Java, and Python. In our experiments we use the Python interface. CPLEX is widely used in both academia and industry for solving large-scale optimization problems.

# 3 Tuning Linear Programming Solvers for Query Optimization

Our contribution consists in conducting experiments on small packing LP problems that are generated from cardinality estimation benchmarks as mentioned in Section 2.2 as well as randomly generated LPs with varying sizes. We use different LP solvers, and different update methods to solve these LPs. We then proceed to compare results based on time and memory performance. We later build an analysis of our datasets' properties. Finally, we aim to give a recommendation on how to build the best performing LP solver based on the particularities of the LP problems.

## 3.1 Implementation overview

The final code repository contains 3 different solvers as shown in the UML graph 3.1 and a `compareSolvers.cpp`, in which we can conduct our benchmarks.

## 3.2 Tableau simplex solver

This solver is the simplest of the three solvers, it follows the steps of the standard simplex algorithm in its tabular form, see Algorithm 3. It uses dense matrices and vectors. Although the `doPivotting()` operation is costly, ( with a time complexity of $O(m \times n)$ where $m$ is the number of rules and $n$ the number of variables) the simplicity of this algorithm can make it suitable for smaller problems.

## 3.3 Data structures

In the following, we will discuss the data structures used in our implementations, to store the constraint matrix **A**.

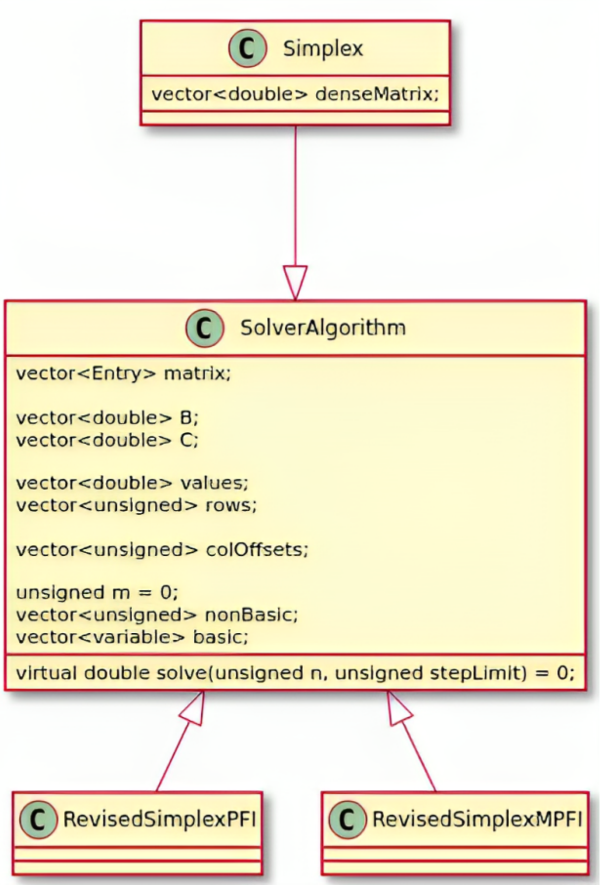


Figure 3.1: An UML Graph explaining the hierarchy of the implementation

**Algorithm 3** Tableau Simplex Algorithm

```
Input: Packing LP maximisation problem in computational form
Output: Optimal value $z$
Step 1: Pricing: Find pivot column, or entering variable using Bland's rule
    $enteringVars \leftarrow$ findPivotColumnCandidates()
    if no entering variable found then
        print "Optimal value reached."
        return $z$
    end if
    $pivotColumn \leftarrow enteringVars[0]$
Step 2: Find pivot row, or leaving variable using the ratio test
    $pivotRow \leftarrow$ findPivotRow($pivotColumn$)
    if no leaving variable then
        print "The given LP is unbounded."
        return $\infty$
    end if
Step 3: Update the tableau using pivotting and update the objective function value
    doPivotting($pivotRow, pivotColumn, z$)
Goto Step 1
```

### 3.3.1 Dense Matrix

Given a matrix $\mathbf{A}$ of dimensions $m \times n$, the density $D$ of the matrix is defined as:

$$D(\mathbf{A}) = \frac{\text{Number of non-zero elements in } \mathbf{A}}{m \times n}$$

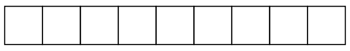
$D$ is a measure between 0 and 1, where 0 indicates a matrix with all zero elements (completely sparse) and 1 indicates a matrix with all non-zero elements (completely dense).

Storing a dense matrix variable $\mathbf{A}$ of dimensions $m \times n$ in C++, we have two alternatives.

- using a two-dimensional array. This array would contain $m$ arrays, representing the rows, each contains $n$ doubles, representing the matrix entries in each row.
- using a one-dimensional array with rows stacked next to each other. This array contains $m \times n$ entries. With the $a_{row,col}$ entry located at `A[row * (m + n) + col]`

Note that even though there is a difference between `array`, `vector` and `list`, we choose `std::vector`, in all our implementation, because it suits our purpouses. We also opt for 1D array as opposed to 2D array for better memory usage and speed. We explain this choice: The 2D array typically requires slightly more memory, due to the pointers in the 2D array that point to the set of allocated 1D arrays. While this difference might seem negligible for large arrays, it becomes relatively significant for smaller arrays. In terms of speed, the 1D array often outperforms the 2D array due to its contiguous memory allocation, which reduces cache misses. While the 2D dynamic array introduces an extra level of indirection, the 1D array has its own overhead stemming from index calculations. However, index calculations are considered generally cheaper than indirection. Overall, 1D vector is a better choice.

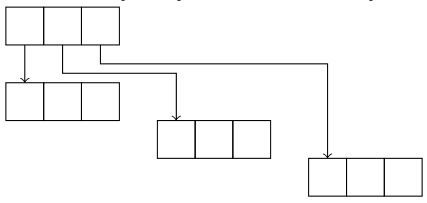
Figure 3.2: Memory Layout of a 1D Dynamic Array



### 3.3.2 Sparse Matrix

Sparsity of a matrix is a feature that can be exploited to enhance memory complexity of our implementation, as we will discuss next. In our dataset, we deal with sparse matrices. We use the Compact Column Representation (CCR) format to store sparse matrices in C++. They are represented using this structure.

Figure 3.3: Memory Layout of a 2D Dynamic Array



```
struct CCRMatrix {
    float *values;  // Non-zero values in the matrix
    int *rowIdx;  // Row indices corresponding to the non-zero values
    int *colPtr;  // Points to the index in ‘values‘ where each column starts
};
```

For example, consider the matrix $A$:

$$A = \begin{bmatrix} 5 & 0 & 0 \\ 0 & 8 & 0 \\ 0 & 0 & 3 \\ 0 & 6 & 0 \end{bmatrix}$$

In CCR format, the matrix is represented using three arrays: `values`, `row_indices`, and `column_pointers`.

$$\begin{aligned} \texttt{values} &= [5, 8, 6, 3] \\ \texttt{row\_indices} &= [0, 1, 3, 2] \\ \texttt{column\_pointers} &= [0, 1, 3, 4] \end{aligned}$$

### 3.3.3 Comparison of memory complexity

Let’s compare the memory complexity for storing a dense matrix using a 1D vector and a sparse matrix using the Compressed Column Representation.

**Dense Matrix stored as 1D vector:**

For a matrix of size $m \times n$: Memory complexity: $O(m \times n)$

**Sparse Matrix stored with CCR format:**

Memory requirements are:

- $nnz$ for the non-zero values. Where $nnz$ is the number of non-zero values in the matrix

- *nnz* for the row indices.
- $n + 1$ for column pointers.

Memory complexity: $O(2nnz + n)$
**Comparison:**
The best choice depends on the matrix sparsity. If the matrix is truly dense (i.e., most of its values are non-zero), then *nnz* approaches $m \times n$ and the dense representation might be more appropriate. However, if the matrix is sparse, then *nnz* is much smaller than $m \times n$ and the CCR format will be much more memory efficient.

## 3.4 Revised Simplex Solver with PFI

In Algorithm 4 we reiterate the revised simplex algorithm, specifically our implementation with PFI.

**Algorithm 4** Simplex Algorithm Using Eta Matrices

```
Initialization
Main Iteration:
Find the Entering Variable through solving: $\mathbf{yB} = \mathbf{c_b}$
Initialize $\mathbf{y}$ to $\mathbf{c_b}$.
Solve for $\mathbf{y}$ using a succession of BTRAN operations using eta matrices 1 to $k$:
Choose the first $j$ such that $\mathbf{c_j} - \mathbf{yA_j}$ is positive.
if no such $j$ exists then
    return "Optimal solution found."
end if
Find the Leaving Variable through solving: $\mathbf{Bd} = \mathbf{a}$
Initialize $\mathbf{d}$ to be the entering column $\mathbf{a}$.
Solve for $\mathbf{d}$ using a succession of FTRAN operations with eta matrices 1 to $k$:
Find the largest $t$ such that $\mathbf{x_B^*} - t\mathbf{d} \geq 0$ for positive components of $\mathbf{d}$. Choose the
corresponding component as the leaving variable.
if no positive entries in $\mathbf{d}$ then
    return "The problem is unbounded."
end if
Update the Current Basic Feasible Solution:
Set the entering variable at $t$.
Update other basic variables: $\mathbf{x_B^*} - t\mathbf{d}$.
Push $\mathbf{d}$ as a new eta matrix onto the vector.
GOTO Main Iteration.
```

## 3.5 Stability

In numerical algorithms, small rounding errors can have a substantial impact. Such inaccuracies can arise from divisions by exceedingly small numbers, resulting in misleading outcomes. Specifically, a zero tolerance, denoted as $\epsilon_2$, safeguards against divisions by extremely small numbers. These divisions tend to produce dangerous rounding errors and might even lead to degeneracy. An essential check in our method is to ensure the diagonal entry in the eta matrix is sufficiently distant from zero; otherwise, based on our experiments, the algorithm may not terminate.

For the chosen leaving and entering variable, we check if the diagonal element in the eta column is smaller than $\epsilon_2$, if so, we reject the current choice of the entering variable and proceed with the next candidate.

Typical values for computations with 15 decimal digit precision are $\epsilon_2 = 10^{-8}$ according to B.A. Murtagh [Mur81].

# 4 Evaluation

## 4.1 Analysis

In the following section, we will conduct an analysis of our various numerical results and benchmarks. We will discuss the particularities of the structure of these LP problems, and if there are any patterns in their solution process. This analysis is based on observing the statistical results, presented later, that we obtained from running different solvers on these problems. This will provide us with insight regarding the optimization of these problems.

We will also analyze and compare the time performance of our implementation solvers, Umbra's revised simplex with the MPFI update variant, Cplex, and HiGHS.

### 4.1.1 Analysis JOB dataset

#### Structure and properties

In this subsection, we will conduct an analysis of the JOB dataset properties.

As opposed to what the linear programming research has dealt with, which is very large problems, we are dealing with hundreds of small problems. The statistics collected about the JOB dataset in Table 4.2 prove this, since the LP size range is small, (number of rules ranging from 1 to 19 and number of variables between 1 and 6).

Given the size of these LPs, one might think the constraint matrices involved are practically dense, however, our results show differently. If we look at the densities in the Table 4.2 we can see that the constraint matrix average density for the JOB queries is around 0.6703.

These matrices are represented in the revised simplex algorithm by sparse matrices but not as sparse as it would have been if the problem was large, i.e. small matrices that are not small enough to be dense.

#### Time efficiency

From the table 4.3, we can see that MPFI solver outperforms the two other revised simplex solvers and Cplex. It records the largest Query-Per-Hour metric, around 3.8`billion` queries per hour. The time profiles of the three solvers show that for the for an LP size less than 30 (we define the LP size as the product of the number of variables and the number of rules this LP has. `lp_size` $= m \times n$) Tableau's performance is almost as good as MPFI. However, from `lp_size` $\geq 30$ MPFI outperforms the Tableau method. Meanwhile, the PFI delivers the worst execution time out of the three solvers, for this JOB dataset of small queries.

Also, Cplex is slower than all other solvers for this dataset.

These results may be due to several reasons. We think the most likely reason, according to theoretical analysis of the functionalities of these solvers, and the documentation of Cplex, is based on the small size of these LP problems.

- Overhead of Cplex: Cplex is a commercial solver designed to handle large-scale LPs. To do this, it incorporates many (in our case unnecessary) advanced techniques like long initialization phases, and presolving, where it attempts to simplify the LP, analyze it, and identify special structures... While this can be highly beneficial for large and more complex problems, for small ones it introduces unnecessary overhead.

- Simplicity of Tableau solver: our solver is fundamental, and skips unnecessary preprocessing or initialization. The bottleneck of the Tableau simplex solver is the Pivoting step, which includes row operations on dense matrices. For small problems, the data might fit entirely within the CPU cache, making certain operations faster. The 1D dense matrix data structure may be a good choice.

- Overhead of revised simplex methods: in the revised simplex method with PFI we start by preparing the sparse matrix format. This data structure offers a beneficial speedup for larger sparser problems, but for small problems, it represents an unnecessary overhead, compared to the Tableau simplex, even though the latter uses a dense representation of the tableau, but in 1D vector, which may be more cache-friendly. The Revised Simplex with PFI method updates the matrix by enqueuing an eta file to the pivots, but it still applies two basis multiplications BTRAN and FTRAN , which can be slower than the Pivoting operation of Tableau, especially for small problems.

- Additionally, in our implementation of PFI we utilize a zero tolerance, see in 3.5, to reject the choice of the entering variable if it may lead to diagonal entries very clos to zero. This may be slowing down our PFI implementation by making the steps take longer than they should.

It's worth noting that the time profiles of Revised Simplex and Cplex for larger or more complex problems may look very different than for smaller LPs. This is what we will discuss in the next part of the analysis.

### 4.1.2 Analysis of experiments on randomly generated LPs

For small LP sizes, we observed that MPFI outperforms other variants of Simplex, but they all outperform Cplex. However, for larger LPs, we come across different results. From figure 4.12 we see that Tableau does not scale well, compared to revised simplex with PFI and MPFI update methods.

Also, starting from `lp_size` $\geq 15,000$ approximately, we observe in Figure 4.13 that Cplex starts outperforming our solvers.

#### Why is HiGHS so slow?

Using `linprog` from SciPy with HiGHS as a method can be slower than using the HiGHS interface directly, mainly due to the overhead, additional features and encapsulation.

## 4.2 Results

All the following results have been obtained on a system with the following settings:

- OS: Ubuntu 22.04.1 LTS x86_64
- Host: 82A2 Yoga Slim 7 14ARE05
- CPU: AMD Ryzen 7 4800U with Radeon Graphics (16) @ 1.800GHz
- GPU: AMD ATI 03:00.0 Renoir
- RAM: 10409MiB / 15363MiB

Presolve techniques are not used, the solvers assume an input of the form explained in the UML graph 3.1 and in the format input as described in the following section.

The computed optimal solutions have been validated using the SciPy Python library, our solvers terminate and deliver the correct optimum. In the following results, we refer to Umbra's code as MPFI.

#### Note on cache locality

Through our experiments, we have noticed that if we call our C++ solvers consecutively, this may affect the recorded execution times and thus make our benchmarks unreliable or inaccurate. This may be due to cache locality: If the second solver is working on data that was recently processed by the first solver, it might benefit from cache locality, as some of the required data might still be in the cache. This can lead to faster execution times for the second solver. However, if the first solver used a significant amount of memory and displaced data relevant to the second solver from the cache, then the second solver might experience cache misses. Cache misses can slow down the execution as the CPU has to fetch the required data from the main memory. We detect varying results when we switch the order in which we call our solvers, and so this effect is likely significant. Our solution to avoid this is by running the solvers multiple times and then dividing the duration by the number of repetitions. This is our approach to receive accurate performance measures and be able to compare the solvers reliably.

### 4.2.1 Results on Query datasets

The input files `TPCH`, `TPCDS`, and `JOB` contain packing LP problems. We have already established the mathematical derivation of how these query-related packing LP problems are generated in 2.2.

The `lp.txt` file is structured for machine readability. In this format, each line represents a single LP. The line starts with the number of rules in that problem. For each rule, the number of entries in the coefficient matrix is specified first, followed by pairs of values: the column number and the coefficient. This is convenient to parse the entries and then populate our sparse matrix representation quite efficiently.

An example:

```
lp:
3 1 0 0.0512386 2 0 0.0510433 1 0.0510433 2 0 0.0401758 1 0.0803516
```

Table 4.1: Benchmarks and number of LPs.

| Benchmark | Number of LPs |
|---|---|
| JOB [Lei+15] | 2230 |
| TPC-H [TPC23b] | 16 |
| TPC-DS [TPC23a] | 148 |

#### The JOB dataset results

In the Table 4.2 are some important statistical findings following our experiments with the JOB dataset. This is the largest dataset of all three query datasets. It contains duplicated queries so we perform a removal of the duplicated LPs, this leads to a drop from 2230 to 1423 LPs. We collect the data and perform plotting and data summary using R scripts. Our results regarding this dataset can be viewed in Figures 4.2, 4.3, 4.1 and the Table 4.2.

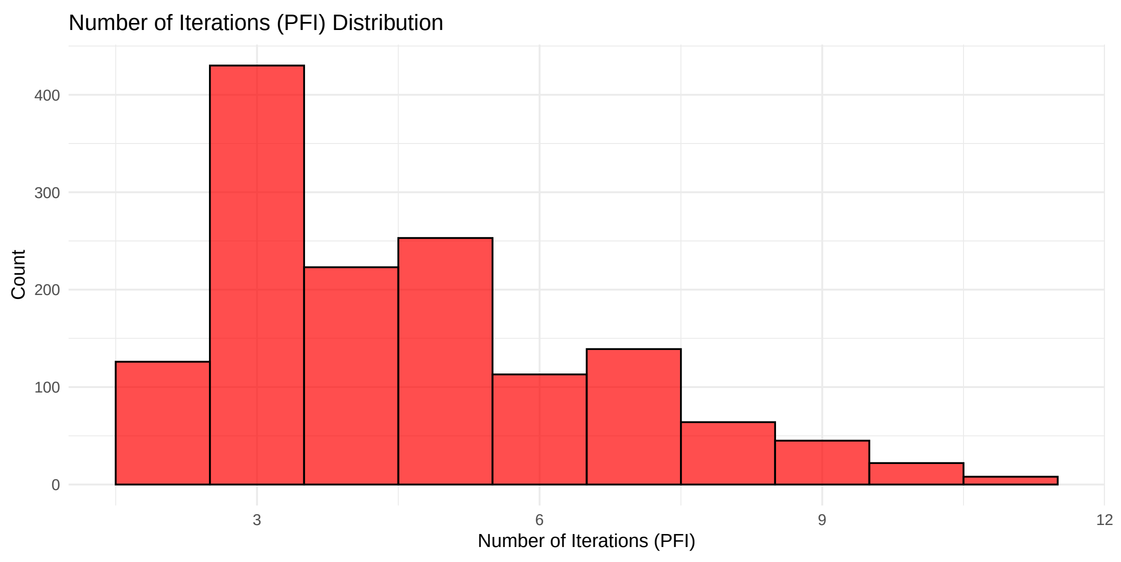


Figure 4.1: Boxplot for number of iterations for PFI for JOB dataset. This shows that PFI takes a small number of iterations for our dataset, so we do not need to refactorize the basis matrix.

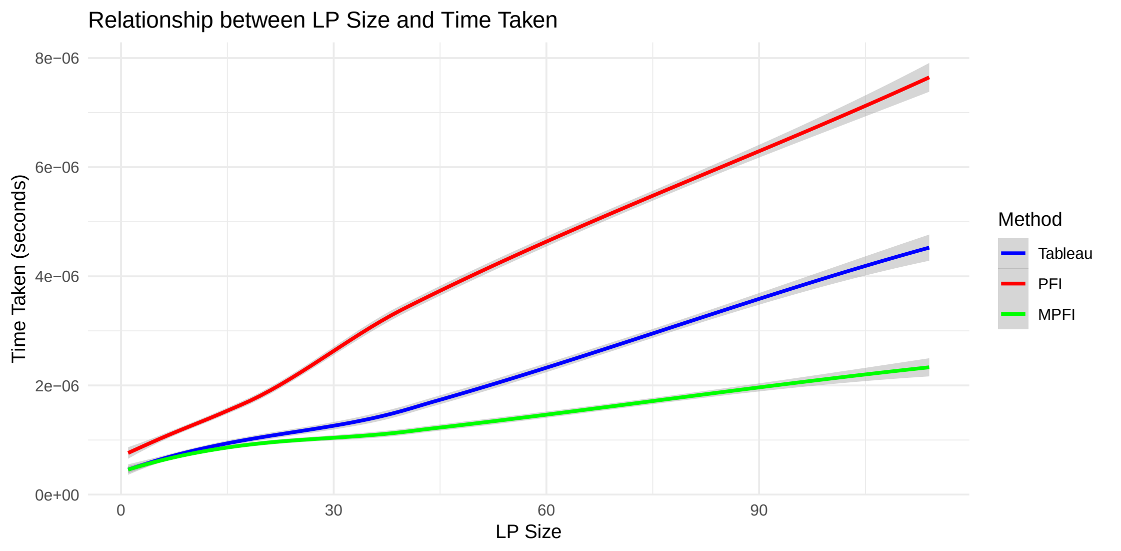


Figure 4.2: Relation between LP size and time for the 3 simplex solvers for JOB dataset.

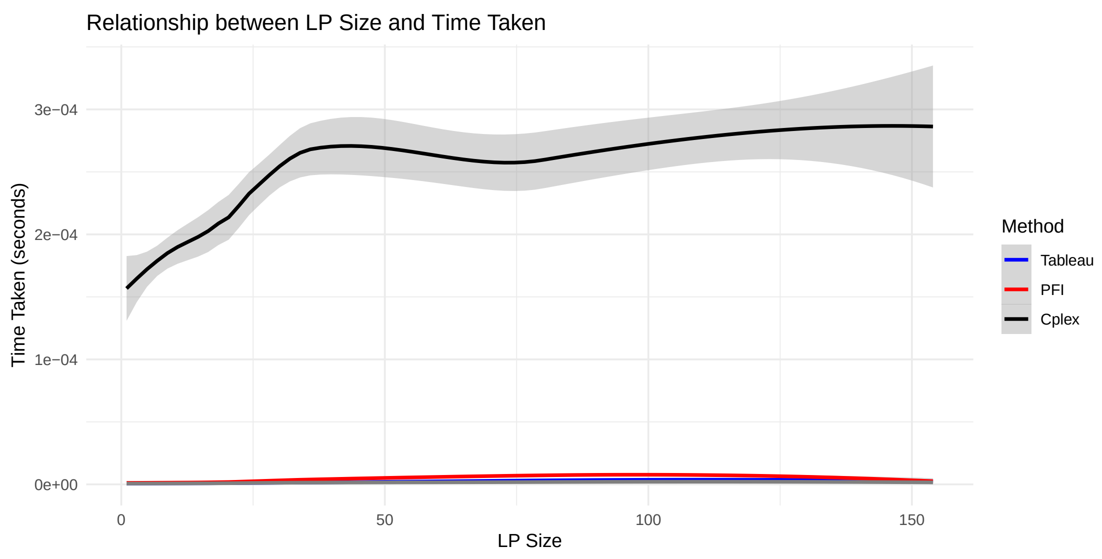


Figure 4.3: Relation between LP size and time for the 3 simplex solvers for JOB dataset and Cplex.

Table 4.2: Statistics about JOB dataset, solution time is in microseconds.

| Variable | Min | Median | Mean | Max |
|---|---|---|---|---|
| LP size | 1.00 | 18.00 | 26.46 | 114.00 |
| Number of Rules | 1.000 | 6.000 | 7.037 | 19.000 |
| Number of Variables | 1.00 | 3.00 | 3.07 | 6.00 |
| Constraint Matrix Density | 0.3684 | 0.6667 | 0.6703 | 1.0000 |
| Solution Time Cplex | 96.08 | 193.60 | 204.18 | 651.84 |
| Solution Time Tableau | 0.160 | 0.970 | 1.301 | 11.060 |
| Solution Time PFI | 0.660 | 1.770 | 2.334 | 10.260 |
| Solution time MPFI | 0.1200 | 0.8900 | 0.9866 | 7.1800 |
| Number Iterations Tableau | 2.000 | 4.000 | 4.829 | 14.000 |
| Number Iterations PFI | 2.000 | 4.000 | 4.621 | 11.000 |
| Number Iterations MPFI | 2.000 | 4.000 | 4.671 | 13.000 |
| Optimal Value | 0.3188 | 20.6702 | 21.8575 | 42.1804 |

**TPC-DS results**

In the Table 4.4 are some important statistical findings following our experiments with the TPC-DS dataset. We collect the data and perform plotting and data summary using

Table 4.3: Number of LPs or Queries Solved by Hour for the JOB dataset

| Method | Number of LPs/Hour |
|---|---:|
| Revised Simplex MPFU Umbra | 3,809,424,659 |
| Tableau Simplex | 2,870,945,324 |
| Revised Simplex PFI | 1,521,766,898 |
| SciPy (method highs) | 4,069,108 |
| Cplex | 18,180,632 |

R scripts. The results can be viewed in the Figures 4.6, 4.5, 4.4 and 4.7.

Table 4.4: Statistics about the TPC-DS dataset, the solution time is expressed in microseconds.

| Variable | Min | Median | Mean | Max |
|---|---|---|---|---|
| LP size | 1.00 | 20.00 | 33.51 | 154.00 |
| Number of Rules | 1.00 | 6.00 | 7.88 | 22.00 |
| Number of Variables | 1.000 | 3.000 | 3.234 | 7.000 |
| Constraint Matrix Density | 0.4365 | 0.8000 | 0.7415 | 1.0000 |
| Solution Time Cplex | 94.65 | 209.81 | 219.54 | 543.36 |
| Solution Time Tableau | 0.190 | 0.990 | 1.459 | 11.370 |
| Solution Time PFI | 0.730 | 1.980 | 2.965 | 32.540 |
| Solution time MPFI | 0.2700 | 0.9800 | 1.1139 | 7.7000 |
| Number Iterations Tableau | 2.000 | 4.000 | 4.315 | 11.000 |
| Number Iterations PFI | 2.000 | 4.000 | 4.288 | 12.000 |
| Number Iterations MPFI | 2.000 | 4.000 | 4.228 | 12.000 |
| Optimal Value | 4.82 | 14.48 | 15.95 | 34.60 |

**TPC-H results**

In the Tables 4.6 and 4.5 are some important statistical finds following our experiments with the TPC-H dataset. We collect the data and perform plotting and data summary using R scripts. The results can be viewed in the Figures 4.9, 4.8 and 4.10.

### 4.2.2 Results on randomly generated LPs

We also test our solvers alongside Cplex, on packing LPs generated randomly.

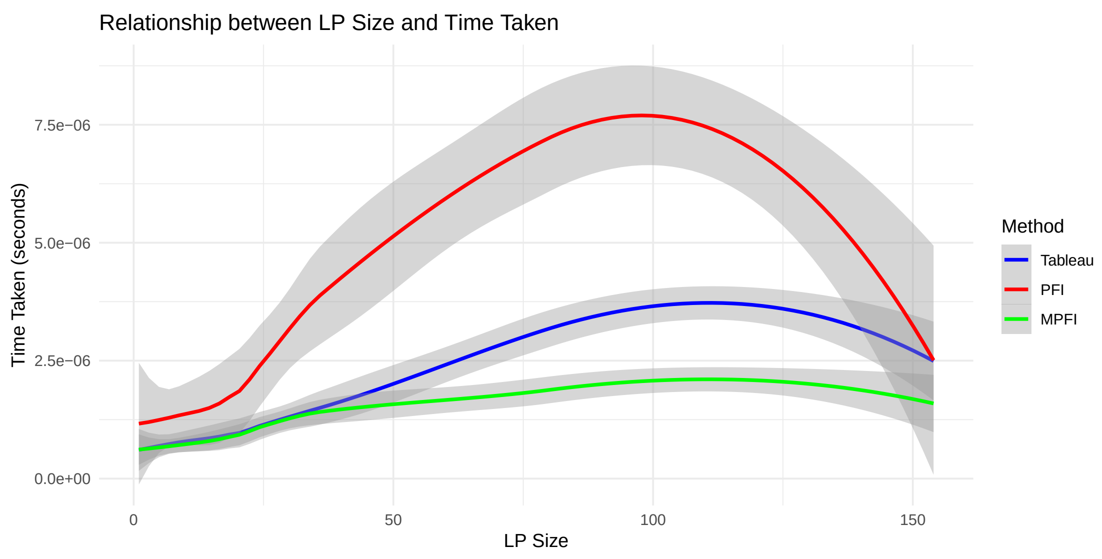


Figure 4.4: Time profile comparison (Relationship between time and LP-size) of our 3 solvers solving the TPC-DS dataset

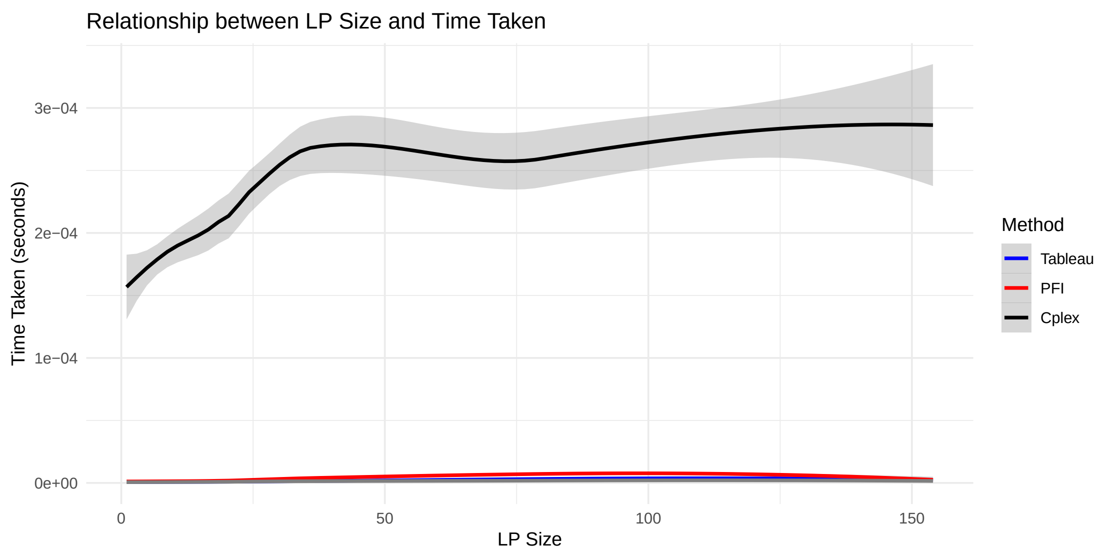


Figure 4.5: Time profile comparison (Relationship between time and LP-size) of our 3 solvers and Cplex solving the TPC-DS dataset

We use a Python script as a generator for packing linear programming (LP) problems. It creates LP problem instances based on the specified number of rules and variables.

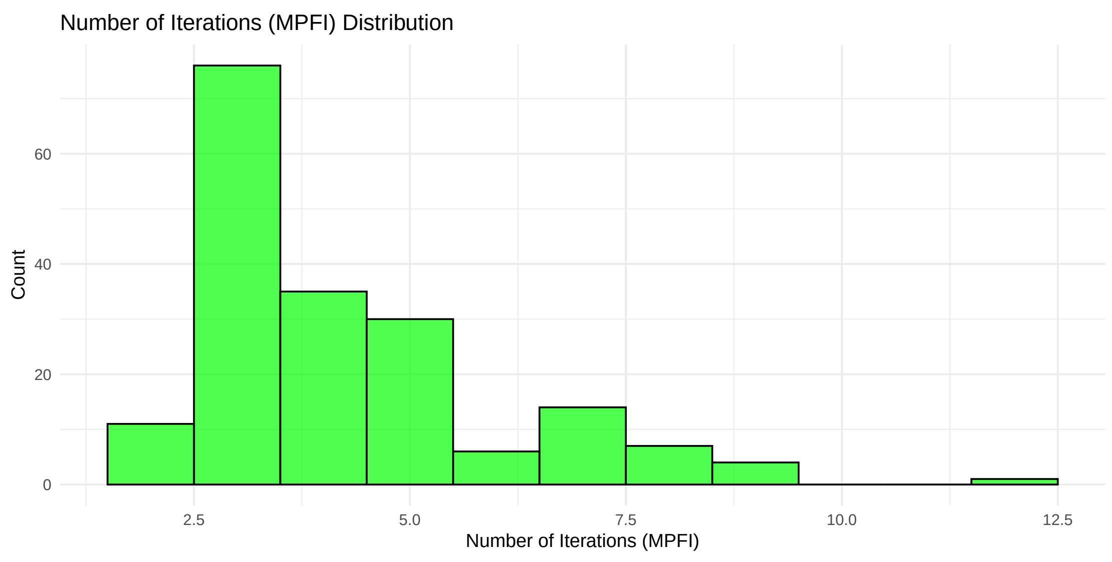


Figure 4.6: Histogram for number of iterations for MPFI for TPC-DS dataset

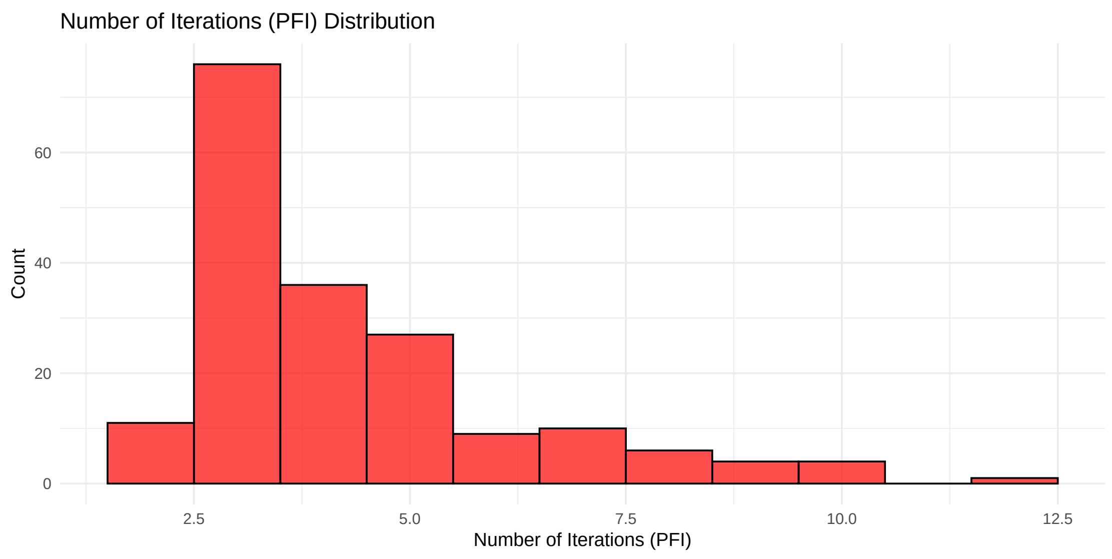


Figure 4.7: Histogram for number of iterations for PFI for TPC-DS dataset

For each rule, a random number of entries, ranging from 1 to the total number of variables, is determined. For every entry in a rule, a unique column number is randomly selected from the available variables, ensuring that the same column is not used more than once for the same rule. A random coefficient, between 0.1 and 1.0, is then assigned

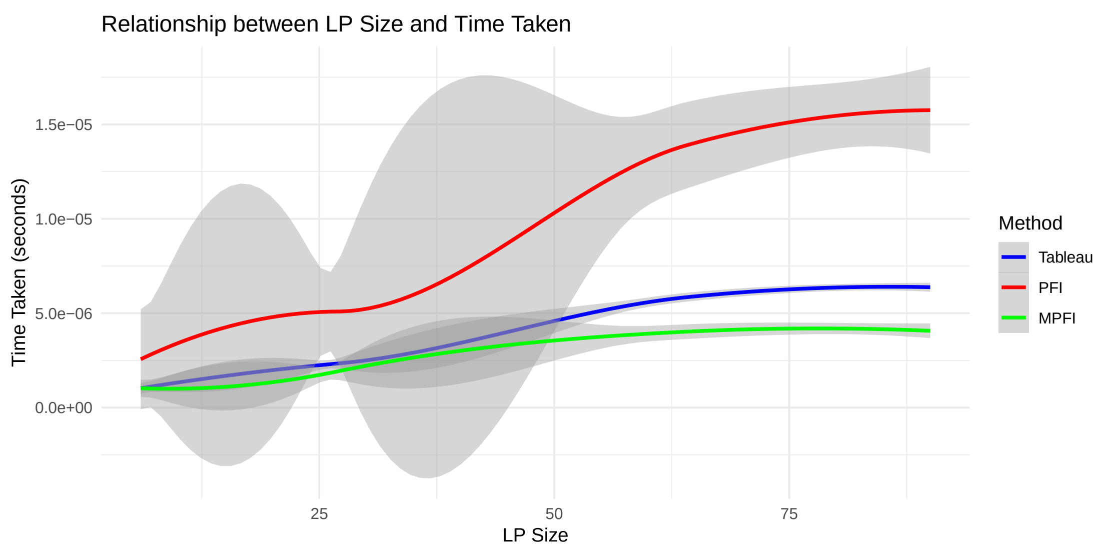


Figure 4.8: Time profile comparison (Relationship between time and LP-size) of our 3 solvers solving the TPC-H dataset

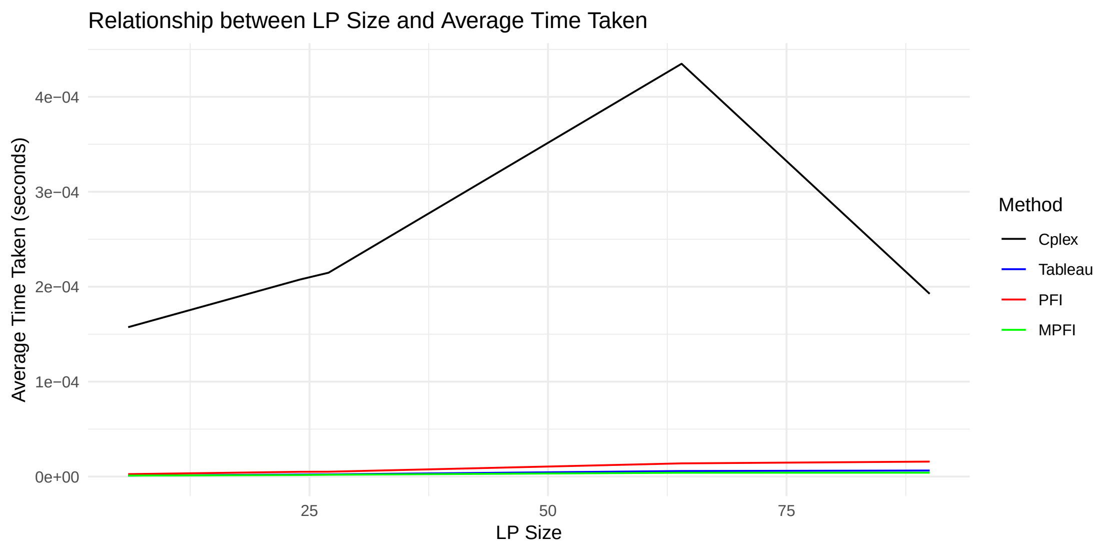


Figure 4.9: Time profile comparison (Relationship between time and LP-size) of our 3 solvers and Cplex solving the TPC-H dataset

to this column. The entire LP problem is represented as a space-separated string, where the first entry denotes the number of rules, followed by the number of entries for each

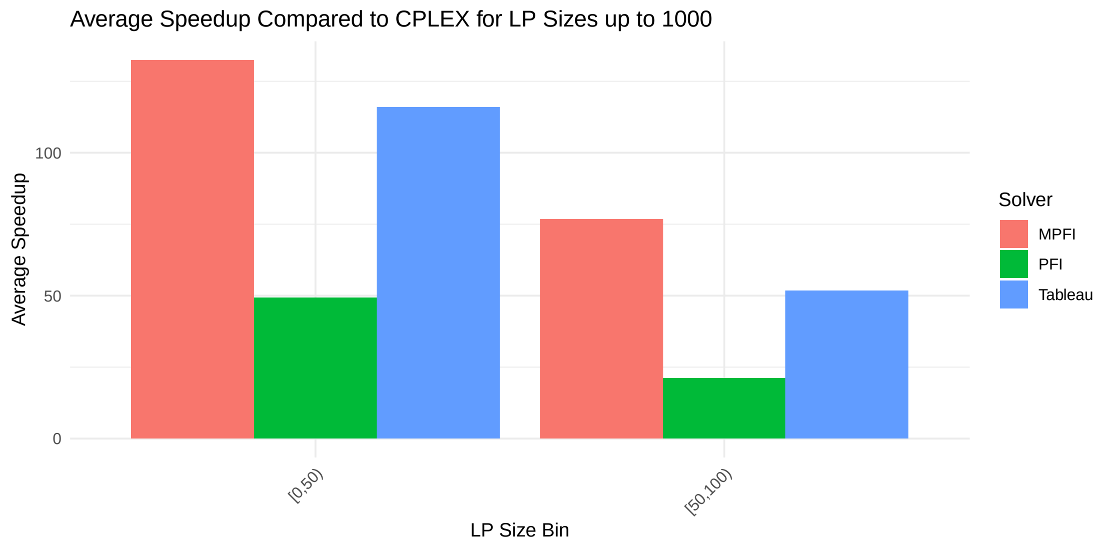


Figure 4.10: Speedup of our 3 solvers solving the TPC-H dataset compared to Cplex.

Table 4.5: Number of LPs Solved by Hour for the TPC-H dataset

| Method | Number of LPs/Hour |
|---|---:|
| Revised Simplex MPFU Umbra | 1,295,255,228 |
| Tableau Simplex | 909,090,909 |
| Revised Simplex PFI | 379,997,361 |
| Cplex | 8,042,340 |

rule, and then the column-coefficient pairs. The main part of the script generates a series of such LP problems, incrementally increasing the number of rules and variables for each problem, and write them in a text file. This is how we get our randomly generated packing LPs with increasing sizes in the same format as the query datasets.

We run our solvers, and also Cplex on this dataset, which provides us with numerical results in the figures: 4.11, 4.7, 4.12, 4.15, and 4.14.

We want to explore those results to investigate how these solvers scale, and if the speedup they provide is still substantial at larger LPs.

We get the time profile in Figure 4.11. We want to see the further evolution of this time profile, so we increase the number of generated LPs and explore larger LPs, see Figure 4.12.

Table 4.6: Statistics about the TPC-H dataset, the solution time is expressed in microseconds.

| | Min. | Median | Mean | Max. |
|---|---|---|---|---|
| **Number of Rules** | 3.00 | 12.50 | 11.75 | 18.00 |
| **Number of Variables** | 2.000 | 3.500 | 3.562 | 5.000 |
| **Constraint Matrix Density** | 0.4111 | 0.6146 | 0.6132 | 0.8333 |
| **Solution Time Cplex** | 149.2 | 200.5 | 252.6 | 1059.3 |
| **Solution Time Tableau** | 1.010 | 4.230 | 3.960 | 6.510 |
| **Solution Time PFI** | 2.540 | 7.970 | 9.474 | 19.050 |
| **Solution time MPFI** | 1.010 | 2.740 | 2.779 | 4.700 |
| **Number Iterations Tableau** | 3.0 | 5.5 | 5.0 | 7.0 |
| **Number Iterations PFI** | 3.00 | 5.00 | 5.50 | 8.00 |
| **Optimal Value** | 14.44 | 20.73 | 19.69 | 22.77 |

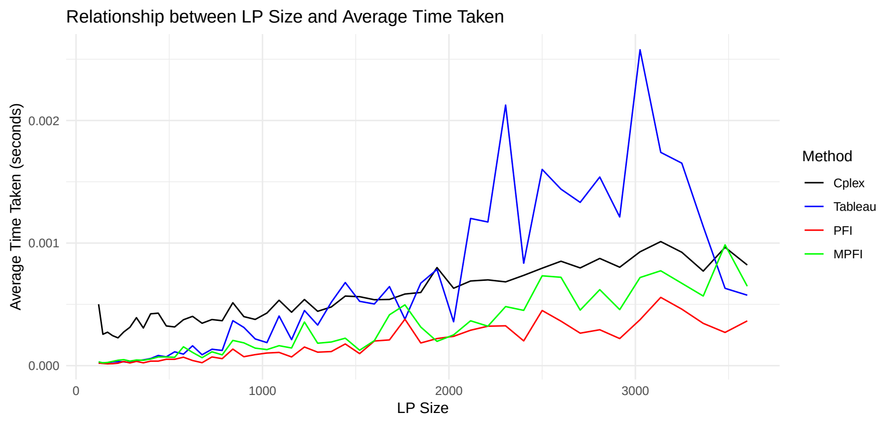


Figure 4.11: Relation between LP size and time for the 3 simplex solvers and Cplex for randomly generated dataset with increasing LP size.

Table 4.7: Number of LPs Solved by Hour for the randomly generated dataset

| Method | Number of LPs |
|---|---|
| Revised Simplex MPFU Umbra | 13,520,349 |
| Tableau Simplex | 6,963,530 |
| Revised Simplex PFI | 24,352,331 |
| Cplex | 6,666,083 |

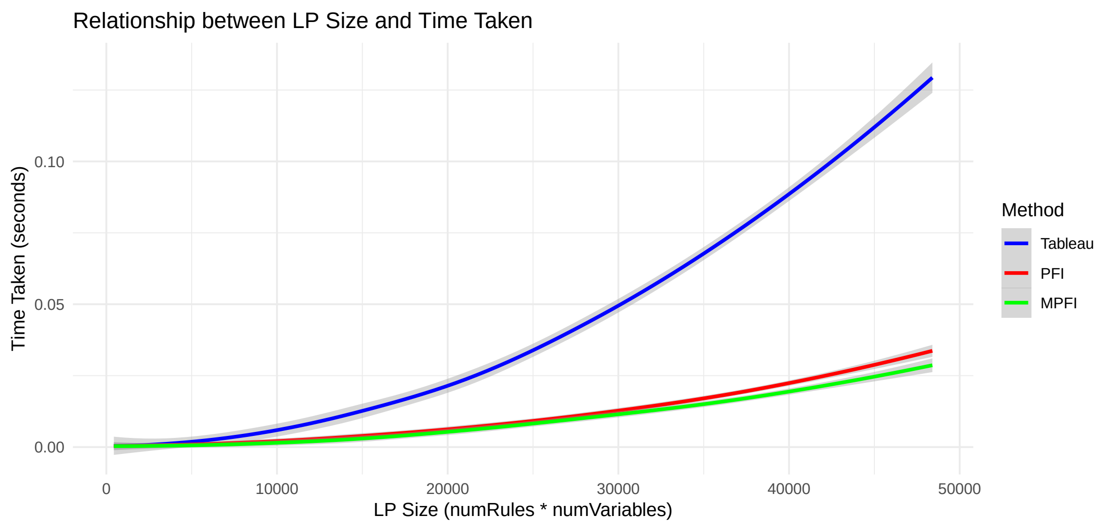


Figure 4.12: Relation between LP size and time for the 3 simplex solvers for the randomly generated dataset with increasing LP size ranging up to 220 rules and 220 variables.

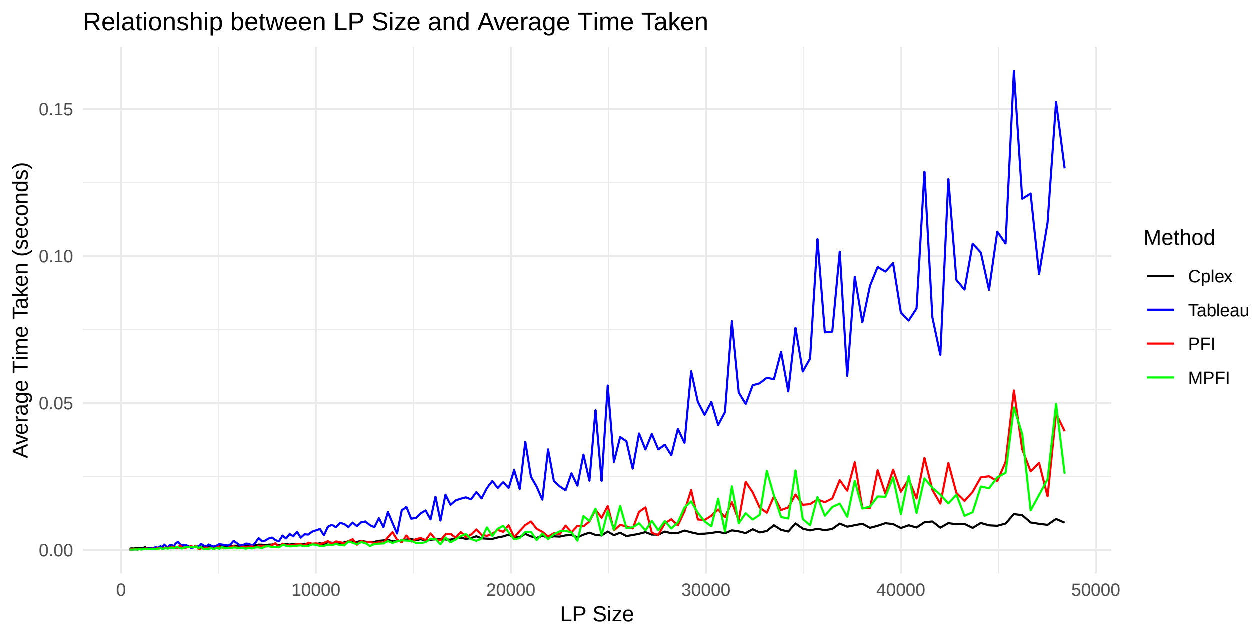


Figure 4.13: Relation between LP size and time for the 3 simplex solvers with CPLEX randomly generated dataset with increasing LP size ranging up to 220 rules and 220 variables.

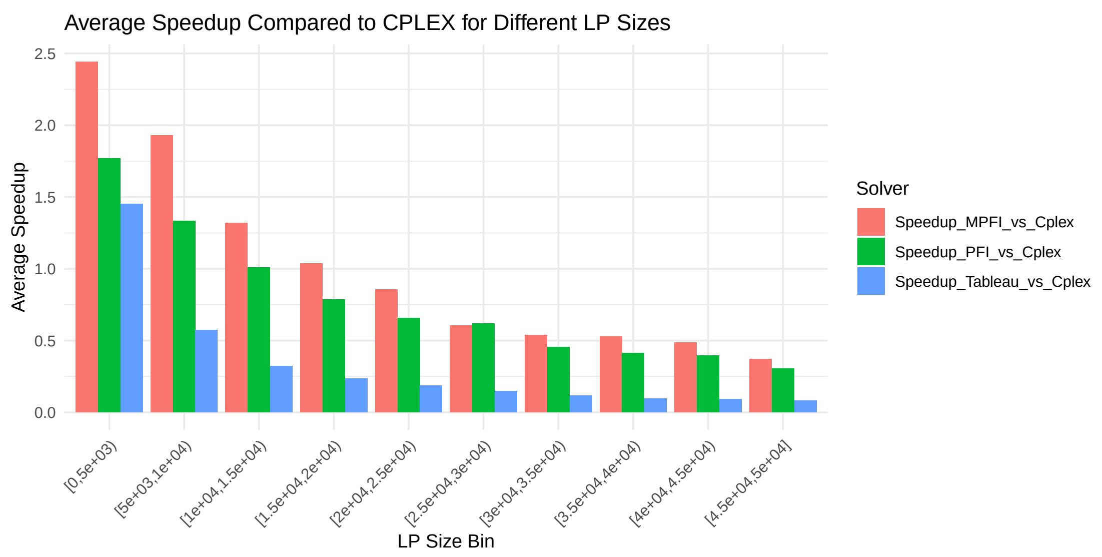


Figure 4.14: Speedup of the three solvers (how many times faster they are) compared to CPLEX for the randomly generated dataset for increasing LP size

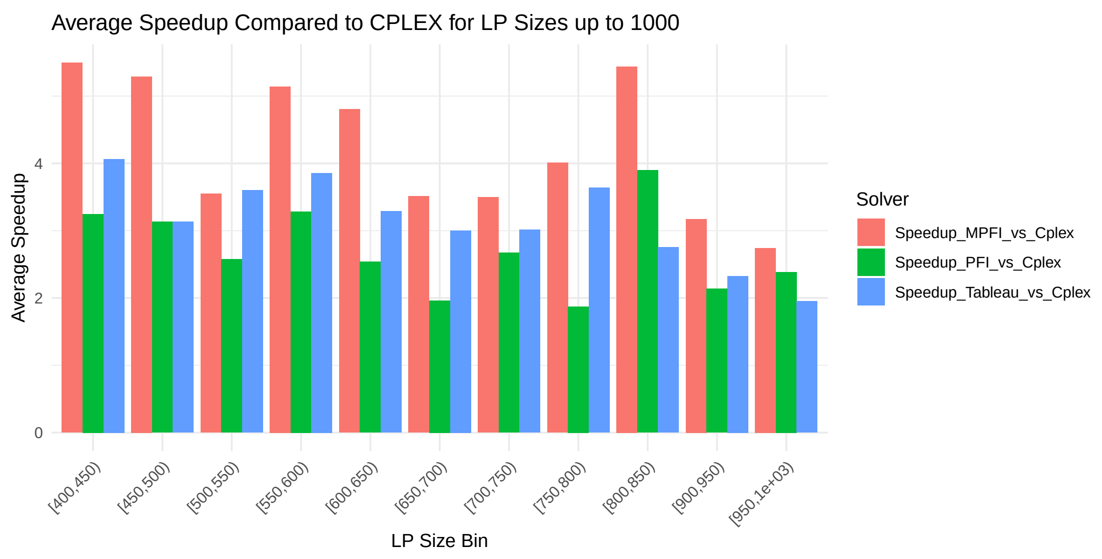


Figure 4.15: Speedup of the three solvers (how many times faster they are) compared to CPLEX for the randomly generated dataset for the LP size up to 1000

# 5 Conclusion

## 5.1 Summary of the results

Our empirical results and analysis revealed some interesting insights about LP solvers' performance as well as the structure of the LPs related to cardinality estimation problems. Unlike typical large LP problems, the datasets contain smaller LP problems. Interestingly, despite their small size, the constraint matrices appear to be predominantly sparse.

The MPFI solver emerged as the top performer, owing to the optimization techniques used in Umbra's code and the assumptions made about the input's structure. However, Cplex, designed for large LP problems, is hampered by its overheads in initialization and presolving for smaller LPs. This results in a reduced performance on the JOB dataset. Our solvers outperform Cplex for smaller LPs, but this advantage diminishes for larger-scale problems.

Over time, Tableau's performance scales the worst, validating our claims regarding its potentially exponential time complexity and underscoring the necessity for more efficient methods when dealing with sizable LPs. Moreover, the hypothesis that MPFI is the fastest PFI variant was confirmed.

To summarize, it's imperative to align the LP problem's size and nature with the appropriate solver. The JOB dataset's analysis elucidates the downsides of using an overly sophisticated solver for more straightforward tasks.

## 5.2 Future Work

### 5.2.1 Forrest-Tomlin update form

The Forrest-Tomlin (FT) update enhances efficiency in the revised simplex method by modifying the LU decomposition of the basis matrix B. This update efficiently handles the BTRAN and FTRAN systems. It may be interesting to compare this variant with the other variants of update methods, since it has been said to be the fastest in literature [HH15].

# List of Figures

# List of Tables

# Bibliography

[AGM13] A. Atserias, M. Grohe, and D. Marx. "Size bounds and query plans for relational joins." In: *SIAM Journal on Computing* 42.4 (2013), pp. 1737–1767.

[Ali95] F. Alizadeh. "Interior point methods in semidefinite programming with applications to combinatorial optimization." In: *SIAM journal on Optimization* 5.1 (1995), pp. 13–51.

[Chv83] V. Chvátal. *Linear programming*. Macmillan, 1983.

[Cpl09] I. I. Cplex. "V12. 1: User's Manual for CPLEX." In: *International Business Machines Corporation* 46.53 (2009), p. 157.

[Dan90] G. B. Dantzig. "Origins of the simplex method." In: *A history of scientific computing*. 1990, pp. 141–151.

[DO54] G. B. Dantzig and W. Orchard-Hays. "The product form for the inverse in the simplex method." In: *Mathematical Tables and Other Aids to Computation* (1954), pp. 64–67.

[GM93] G. Graefe and W. J. McKenna. "The volcano optimizer generator: Extensibility and efficient search." In: *Proceedings of IEEE 9th international conference on data engineering*. IEEE. 1993, pp. 209–218.

[GV13] G. H. Golub and C. F. Van Loan. *Matrix computations*. JHU press, 2013.

[HH15] Q. Huangfu and J. J. Hall. "Novel update techniques for the revised simplex method." In: *Computational Optimization and Applications* 60 (2015), pp. 587–608.

[HH18] Q. Huangfu and J. A. J. Hall. "Parallelizing the dual revised simplex method." In: *Mathematical Programming Computation* 10.1 (2018), pp. 119–142. ISSN: 1867-2957. DOI: 10.1007/s12532-017-0130-5.

[KM72] V. Klee and G. J. Minty. "How good is the simplex algorithm." In: *Inequalities* 3.3 (1972), pp. 159–175.

[Lei+15] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. "How Good Are Query Optimizers, Really?" In: *Proc. VLDB Endow.* 9.3 (Nov. 2015), pp. 204–215. ISSN: 2150-8097. DOI: 10.14778/2850583.2850594.

[Lei+18] V. Leis, B. Radke, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. “Query optimization through the looking glass, and what we found running the join order benchmark.” In: *The VLDB Journal* 27 (2018), pp. 643–668.

[Mic23] Microsoft. *Cardinality Estimation (SQL Server)*. Accessed: Sep 19th. 2023. URL: `https://learn.microsoft.com/en-us/sql/relational-databases/performance/cardinality-estimation-sql-server?view=sql-server-ver16`.

[Mur81] B. A. Murtagh. “Advanced linear programming: computation and practice.” In: *(No Title)* (1981).

[NF20] T. Neumann and M. J. Freitag. “Umbra: A Disk-Based System with In-Memory Performance.” In: *CIDR*. Vol. 20. 2020, p. 29.

[Sti10] W. Stille. “Solution techniques for specific bin packing problems with applications to assembly line optimization.” PhD thesis. Technische Universität, 2010.

[TPC23a] TPC-DS Benchmark. *TPC-DS Benchmark*. Last Accessed: 2023/10/9. 2023.

[TPC23b] TPC-H Benchmark. *TPC-H Benchmark*. Last Accessed: 2023/10/9. 2023.