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

Abstract

Our aim with this project is to implement, investigate and compare different methods and techniques to solve small linear programming problems representing the problem of cardinality estimation. A way to estimate realistic and useful upper bounds of query sizes is through linear programming. Studies have shown that cardinality estimation is the major root of many issues in query optimization, which is why we want a practical estimate to choose the best from data plans to run efficient queries. For this purpouse, the cardinality estimation problem can be represented in the form of a packing linear programming problem with the intention of assuming worst-case join sizes and seeing how large the query can get. The result is hundreds of relatively small LP that we collect in datasets and solve them with different methods and algorithms. We then draw conclusions based on the results of our experiments, benchmarks and the previous work done on similar packing LP problems. This should guide us into constructing a thorough analysis of the particularities of these LP problems, what's unique about their structure and if their solution process follows any patterns. We then discuss and draw hypotheses on the ways this analysis can be exploited to further optimize the solution process: which methods or combination of methods deliver the best time and memory complexity.

Contents

1 Introduction

2 Background and Related work

2.1 Background

In this chapter we will talk about optimization, in particular the field of linear programming. We will 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. We will present the background and related work needed to understand our contribution.

2.2 Linear Programming

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 maxmize profit in a business, given some constraints on your variables like raw material availability, labor hours, etc.

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:

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

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.

To derive the setting for our contribution, we also explore a special instance of LP problems called packing LP.

Packing LP

One LP problem class that we are dealing with is called the packing LP problem. It is a special instance where: $\mathbf{c} = \mathbf{b} = \begin{bmatrix}1 & 1 & \dots & 1\end{bmatrix}$. Our specific problem is then expressed as follows:

$$\begin{aligned} &\text{Maximize} && \sum_{i=1}^{n} x_j \\ &\text{subject to} \\ &&& \mathbf{A}\mathbf{x} \leq \mathbf{1}_m, && (2.2) \\ &&& x_i \geq 0, \qquad i = 1, \dots, n && (2.3) \end{aligned}$$

Where $\mathbf{1}_m = \begin{bmatrix}1\\1\\\vdots\\1\end{bmatrix}$

This specific class of LPs has a simple structure that we can exploit, see Chapter 3, to further optimize our implementation.

2.2.1 Duality

The duality theorem is an interesting result in linear programming, that states that very instance of maximization problem has a corresponding minimization problem called its dual problem. The two problems are linked in an interesting way: if one problem has an optimal solution, then so does the other, and their optimal solutions are equal.

For instance, consider the primal-dual pair LP:

$$\begin{array}{lll} \text{maximize} & \mathbf{c}^T\mathbf{x} & \\ \text{subject to} & \mathbf{A}\mathbf{x} \leq \mathbf{b} & \longrightarrow \\ & \mathbf{x} \geq 0 & \end{array} \qquad \begin{array}{ll} \text{minimize} & \mathbf{b}^T\mathbf{y} \\ \text{subject to} & \mathbf{A}^T\mathbf{y} \geq \mathbf{c} \\ & \mathbf{y} \geq 0 \end{array}$$

2.2.2 Geometric Interpretation

In this part let's 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 2.2.2. The linear programming problem 2.1 can be understood geometrically as follows: Each inequality in the set of constraints is represented by a line. Therefore, the feasible region χ of any LP problem can be described as the intersection delimited by those lines, which is

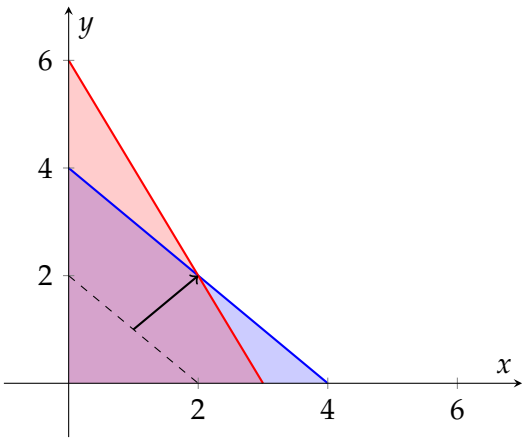


Figure 2.1: Feasible region of the LP problem

called a polyhedron, or in our 2D case, a polytope. The feasible region is the set of all feasible points, or vertices. 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 descibe 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.

2.2.3 Feasibility, unboundedness

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

The LP is said to have unbounded set of solutions if its solution can be made infinitely large without violating any of its constraints in the problem. 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 $\mathbf{x} \in \chi$.

2.2.4 The Standard Simplex Algorithm

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

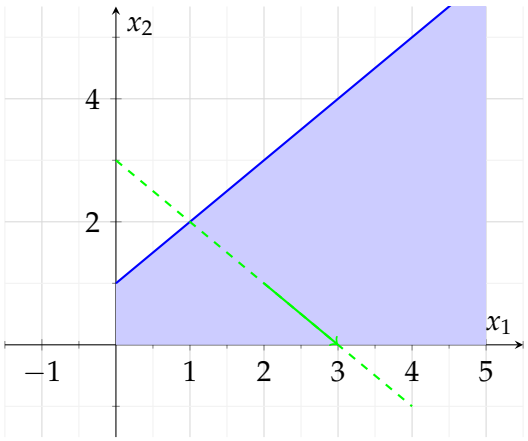


Figure 2.2: Graphical representation of an unbounded LP problem

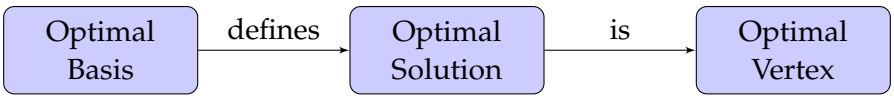


Figure 2.3: Conceptual relationship between an optimal basis, an optimal solution, and an optimal vertex.

have implemented our version of this algorithm in the C++ language and we use it, among others, to solve our dataset.

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 7form 2.4, 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 $s_1, s_2, \ldots, s_m$. After introducing those variables let's look at the problem 2.1, where the constraints are now linear equalities:

$$
\begin{aligned}
\text{Maximize} \quad & z = \mathbf{c}^T \mathbf{x} \\
\text{subject to} \quad & \sum_{j=1}^{n} a_{1j} x_j + s_1 = b_1 \\
& \sum_{j=1}^{n} a_{2j} x_j + s_2 = b_2 \\
& \vdots \\
& \sum_{j=1}^{n} a_{mj} x_j + s_m = b_m \\
& x_1, x_2, \ldots, x_n, s_1, s_2, \ldots, s_m \geq 0
\end{aligned} \tag{2.4}
$$

We then have a LP problem in the appropriate form and 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. 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 feasible and optimal vertex, solution, point interchangeably to refer to the optimum of a LP problem. Now we will explore the concept of feasible and optimal basis. All the concepts are however equivalent, see 2.2.3, 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.

In simpler terms, a basis provides a set of "building blocks" from which all other vectors in the space can be constructed without any redundancy.

If an LP problem has an optimal solution, 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 A associated with it. The set of indices of these columns is called a basis.

A basis can be seen as a basis matrix, consisting of the m linearly independent columns of A associated with it.

A feasible basis contains the indices of basic variables. 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.

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

This constitutes a feasible dictionary (or tableau), formally defined in [Chv83]. The simplex method then contructs a sequence of feasible dictionaries until reaching an optimum. The rest of the steps of the simplex algorithm are broadly presented in the following:

Algorithm 1 Simplex Algorithm

```
1:  procedure SIMPLEX(c, A, b)
2:      Initialize a feasible basic solution
3:      Search for an entering variable, pricing
4:      if no entering variable exists then
5:          return "Optimal solution found"
6:      end if
7:      Search for a leaving variable using the minimum ratio test
8:      if no positive pivot element in the column then
9:          return "Unbounded"
10:     end if
11:     Perform the pivot operation
12:     Update the basic and non-basic variables
13:     return current basic solution
14: 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: The choice of the entering variable: we choose a non-basic variable to enter the basis and thus become basic. This is called Pricing. The choice usually

depends on metrics like the largest increase in the objective function, or the largest coefficient. Bland's rule has been proved to guarantee termination: Choose the entering variable as the non-basic variable with the smallest index that has a negative reduced cost for a maximization problem:

$$j = \min\{j : c_j < 0\}$$

- Ratio test: the choice of the leaving variable: 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:

  $$i = \min\left\{i : \text{ratio}_i = \min_{a_{ij}>0} \frac{b_i}{a_{ij}}\right\}$$

After selecting an entering variable and a leaving variable, we perform Pivoting, or updating the tableau.

Termination

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 number of steps in our implementations. This bound is equal to the maximum value a 32-bit unsigned integer can take, or 4294967295.

2.2.5 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 encountred in practice or even in formal experimental studies of 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 observed that the number of iterations are 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, even though there exists polynomial time implementations for it. [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 $O(m \times n)$.

2.2.6 The interior point method

2.2.7 Revised Simplex Algorithm

While the standard or tableau simplex algorithm maintains and updates the entire tableau in its dense form at each iteration, and the pivotting 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, $mathbbB^{-1}$, thus reducing the amount of writing at each step and overall memory usage. This is explained in the following mathematical proof.

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

Partition x into basic (x_B) and non-basic (x_N) variables. Similarly, partition A into B (columns corresponding to x_B) and N (columns corresponding to x_N).

The constraints can be written as:

$$\begin{aligned} Bx_B + Nx_N &= b \\ x_B &\geq 0 \\ x_N &\geq 0 \end{aligned}$$

From $Bx_B + Nx_N = b$, when $x_N = 0$:

$$x_B = B^{-1}b$$

This is the basic feasible solution if all entries of x_B are non-negative.

Compute the reduced costs:

$$\bar{c}_N^T = c_N^T - c_B^T B^{-1} N$$

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

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

$$d = B^{-1}A_j$$

If all entries of d are non-positive, the problem is unbounded.

Otherwise, compute the step length:

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

Update the solution:

$$\begin{aligned} x_B &= x_B - \theta d \\ x_j &= \theta \end{aligned}$$

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

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

- the indices of basic and non-basic variables
- B^{-1} the inverse of the basis matrix. This is used to solve two types of linear equations during an iteration, see step 1 and 3 in 2.2.7.
- the current values of the basic variables, or the current basic feasible solution $x_B = B^{-1}b$

The algorithm is elaborated in 2.2.7. As we can see, step 1 and 3 represent the solving of two types of systems , Forward Transformation (FTRAN) and Backward Transformation (BTRAN). This can be done using a multiplication with the basis matrix inverse'.

However, having to recompute the inverse of a matrix is 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 employ another tool to efficiently update the inverse of the basis matrix 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 Inverse (PFI), the modified PFI and Forrest-Tomlin update method.

The product form inverse update method

We will discuss the PFI, introduced by George Dantzig [DO54]. The revised simplex method needs a way to represent the inverse of the basis matrix in each step, this is because, when finding the entering and leaving variables in each iteration, we need the inverse B^{-1} to solve the BTRAN and FTRAN systems in Step 1 and 3 in 2.2.7. Having to reinvert the basis matrix in each step is costly, which is why an INVERT operation is applied only once at the beginning, on the initial basis B_0. Inverting a matrix using LU decomposition requires $O(n^3)$. a representaion of the inverse of the basis is kept tr Given a basis matrix B and its inverse B^{-1}, suppose p is the index of the basic variable leaving the basis at this step, and the vector a_q is the entering column, or the solution of the FTRAN system, see 2.2.7 Step 3.

$$\begin{aligned}\hat{B} &= B + (a_q - Be_p)e_p^T \\ &= B(I + (a_q - Be_p)e_p^T) \\ &= BE\end{aligned}$$

where $E = I + (a_q - Be_p)e_p^T$ is an *eta matrix*.

Algorithm 2 Revised Simplex Algorithm

1. **Input:** A feasible basic solution, B, c, A, and b
2. **Output:** Optimal solution or a certificate of unboundedness
3. Initialize B^{-1}, the inverse of the basis matrix B
4. **While True:**

   Step 1: Solve the system $yB = c_B$ (BTRAN)

   Step 2: Choose an entering column. This may be any column a of A_N such that ya is less than the corresponding component of c_N. If there is no such column, then the current solution is optimal. In other words: Choose first j such that $c_j - yA_j > 0$ then $a = A_j$ is the enterig column.

   Step 3: Solve the system $Bd = a$ (FTRAN)

   Step 4: Let $x_B^* = B^{-1}b$ the current basic variables' values. Find the largest t such that $x_B^* - td \geq 0$ if there is no such t, then the problem is unbounded; otherwise, at least one component of $x_B^* - td$ 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 x_B^* of the basic variables by $x_B^* - td$. Replace the leaving column of B by the entering column, and in the basis heading, replace the leaving variable by the entering variable.
5. **Return** Optimal solution $B^{-1}b$

The modified product form inverse

Forrest-Tomlin update form

2.3 Cardinality Estimation

An important use case of linear programming solvers in the field of databases is cardinality estimation. In the context of query optimization, LP solvers can be useful to estimate query plan cardinalities and provide a reliable and good enough estimate to be used in selecting the best Join-order, and hence speeding up query execution time. In the pipeline of query execution, cardinality estimation serves as a cornerstone for the query optimization process. Cardinality, defined as the number of tuples in the output, plays a pivotal role in the selection of an optimal query plan. Modern Databank Management System (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.

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.

To set the stage for our implementation, we focus on the problem of upper-bounding the cardinality of a join query Q.

Scenario

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. have a selectivity equals to 1. So, this is the worst-case upper bound.

Variables

Objective

Constraints

2.3.1 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 query graph (how the relations are connected, if there are any shared attributes) and relation sizes. The dual LP problem of the given minimzation problem, is

$$\max \sum_{i} v_i$$

subject to:

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

The dual theorem 2.2.1 states that the both problems have the same optimal values.

This is how our LP datasets are generated.

We start with the inequality 2.5. Applying the natural logarithm to both sides yields 2.6. We then rename the variables, simplifying the inequality to 2.7. Normalizing by dividing both sides by r', we obtain 2.8. This leads us to the objective function for our packing LP problem.

$$|a| \cdot |b| \leq |R| \tag{2.5}$$

$$\ln |a| + \ln |b| \leq \ln |R| \tag{2.6}$$

$$a' + b' \leq r' \tag{2.7}$$

$$\frac{1}{r'} a' + \frac{1}{r'} b' \leq 1 \tag{2.8}$$

$$\text{maximize } a' + b' + c' + d' \quad \text{s.t.} \quad \frac{1}{r'} a' + \frac{1}{r'} b' \leq 1 \tag{2.9}$$

And in this simple abstracted way we get a sample packing LP from our dataset.

2.4 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, HIGHS... We present HIGHS Scipy, and Cplex, because we are using them as a comparison to our solvers.

2.4.1 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 don't make use of HiGHS directly but of Scipy's linprog, and using as a method highs. It is mainly used to verify the accuracy of the optimal values obtained by our solvers.

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

2.4.2 Cplex

IBM ILOG CPLEX Optimization Studio, commonly known as CPLEX, is a software suite for mathematical optimization. CPLEX consists of a library for linear programming, mixed integer programming, quadratic programming, and other related optimization problems. It provides a range of solvers and supports various interfaces, including C++, Java, and Python. CPLEX is widely used in both academia and industry for solving large-scale optimization problems.

3 Tuning Linear Programming Solvers for Query Optimization

3.1 Proposal or Implementation

Our contribution consists in conducting experiments on small packing LP problems that are generated from real-life queries as mentioned in Section 2.3 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 also 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.

3.1.1 Implementation hierarchy

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.

3.1.2 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. It uses dense matrices and vectors. Although the doPivotting() is costly, the simplicity of this algorithm makes it suitable for smaller problems.

3.1.3 Data structures

Dense Matrix

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

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

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

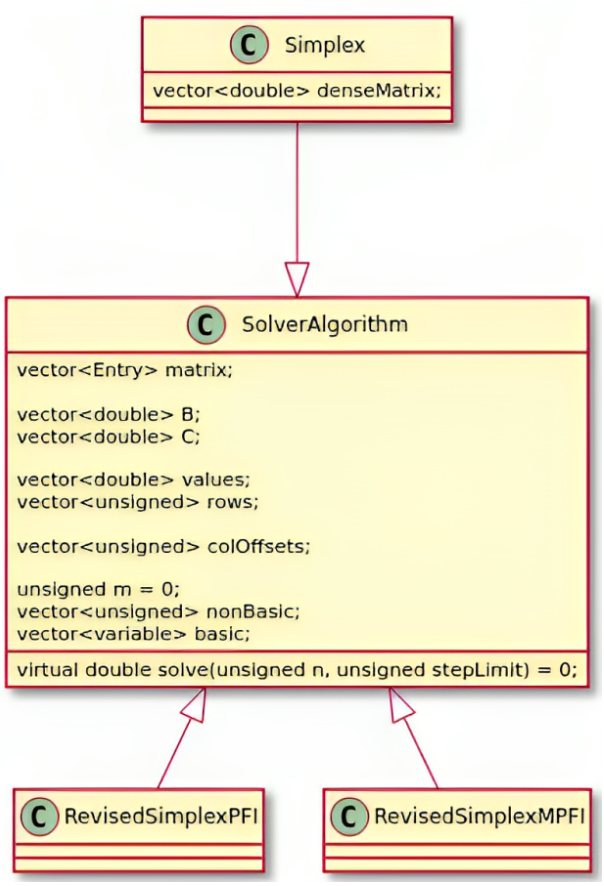


Figure 3.1: An UML Graph explaining the hierarchy of the implementation

Algorithm 3 Tableau Simplex Algorithm

1: **Input:** Packing LP maximisation problem in computational form
2: **Output:** Optimal value z
3: **Step 1:** Pricing: Find pivot column, or entering variable using Bland's rule
4: $enteringVars \leftarrow$ findPivotColumnCandidates()
5: **if** no entering variable found **then**
6: print "Optimal value reached."
7: **return** z
8: **end if**
9: $pivotColumn \leftarrow enteringVars[0]$
10: **Step 2:** Find pivot row, or leaving variable using the ratio test
11: $pivotRow \leftarrow$ findPivotRow($pivotColumn$)
12: **if** no leaving variable **then**
13: print "The given LP is unbounded."
14: **return** ∞
15: **end if**
16: **Step 3:** Update the tableau using pivotting and update the objective function value
17: doPivotting($pivotRow, pivotColumn, z$)
18: **Goto Step 1**

dense). Sparsity of a matrix is a feature that can be exploited to enhance memory complexity of our implementation, as we will discuss next.

Sparse Matrix

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.

```
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`.

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

Comparison of memory complexity

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

- using an array of arrays (two-dimensional array) or `vector<vector<double»`. 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, `vector<vector<double»`. 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`, or dynamoic array, in all our implementation, because it suits our purpouses. We also opt for 1D array as opposed to 2D array for better memory complexity and speed. We explain this choice: The 2D array typically requires slightly more memory than its 1D counterpart. This increased memory usage is attributed 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. However, the 2D dynamic array loses cache locality and consumes more memory because of its non-contiguous memory allocation. While the 2D dynamic array introduces an extra level of indirection, the 1D array has its own overhead stemming from index calculations.

Figure 3.2: Memory Layout of a 1D Dynamic Array

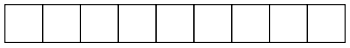
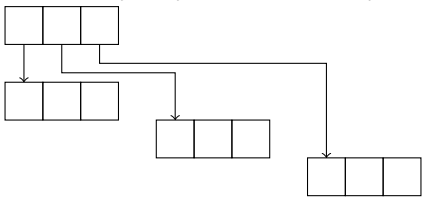


Figure 3.3: Memory Layout of a 2D Dynamic Array



3.1.4 Revised Simplex Solver

3.1.5 Stability

Mention zero tolerances: A zero tolerance epsilon2 saefguards against divisions by extremely small numbers, which tend to produce the most dangerous rounding errors, and may even lead to degeneracy. diagonal entry in eta matrix should be fairly far from otherwise (in our experiment) degeneracy.

3.2 Experiments and 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 correct optimum.

Note on cache locality

Through our experiments, we have noticed that if we call our C++ solvers consecutively, this may affect the recorded exectuin 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 isolating the solvers, and calling each one independently (by commenting out any calls to the others). This is our approach to receive accurate performance measures, and be able to compare the solvers reliably.

3.2.1 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 2.3.

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.

```
lp:
8 2 0 0.0540277 2 0.0540277 ...
```

Table 3.1: Benchmarks and number of queries.

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

The JOB dataset results

In the table 3.2 are some important statistical finds 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 before. We collect the data and perform plotting and data summary using R scripts.

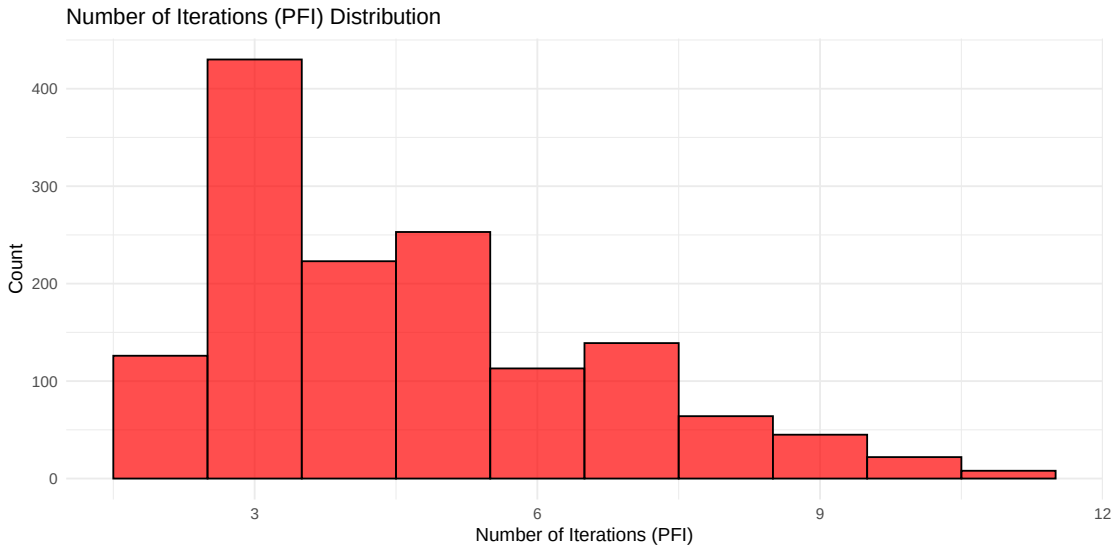


Figure 3.4: Boxplot for number of iterations for PFI for JOB dataset

This is our results:

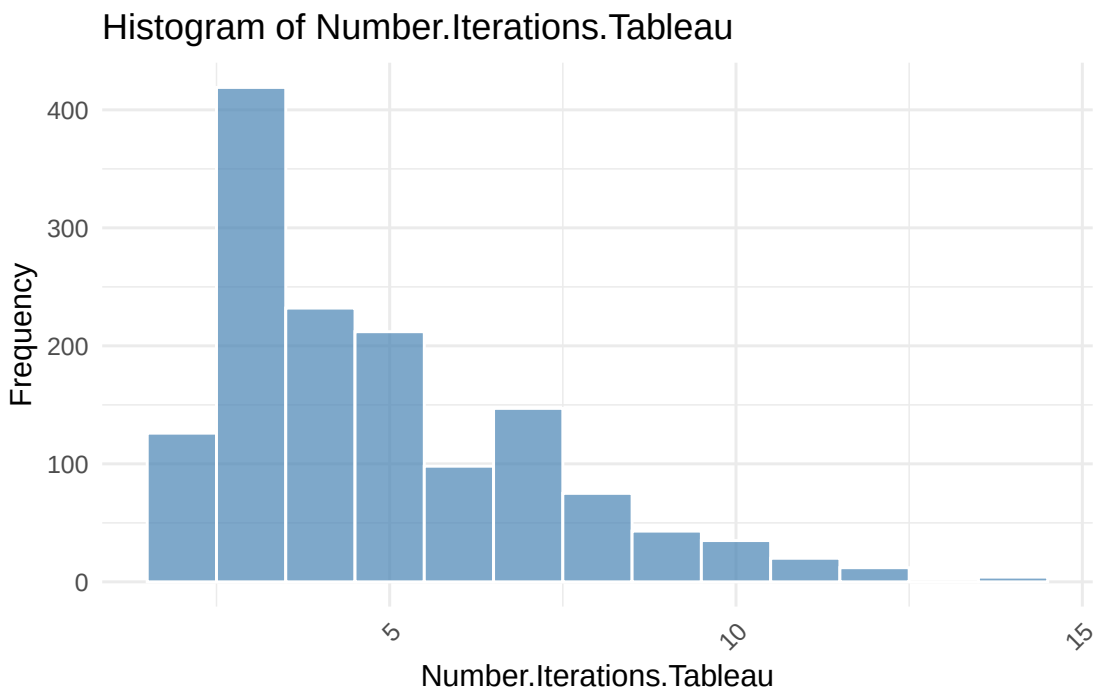


Figure 3.5: Boxplot for number of iterations for Tableau for JOB dataset

TPC-DS results

TPC-H results

3.2.2 Results on randomly generated LPs

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

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. 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 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 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 writes them to a text file. This is how we get our randomly

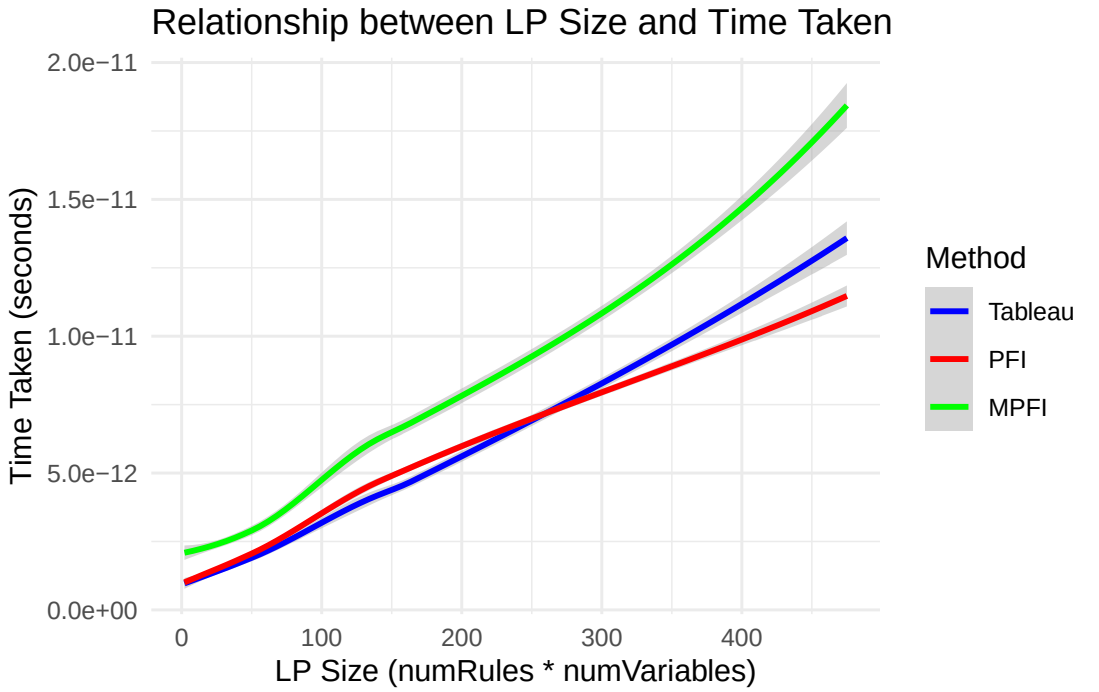


Figure 3.6: Relation between LP size and time for the 3 simplex solvers for JOB dataset.

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 provide us with numerical results in the figures: 3.11, 3.4, 3.12, 3.15, and 3.14.

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 3.11. 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 3.12.

3.3 Analysis

In the following section we will conduct an analysis of our various numerical results and benchmarks. We will discuss the particularites 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 we obtained from running different solvers on these problems. This will provide us with insight regarding the optimization of these

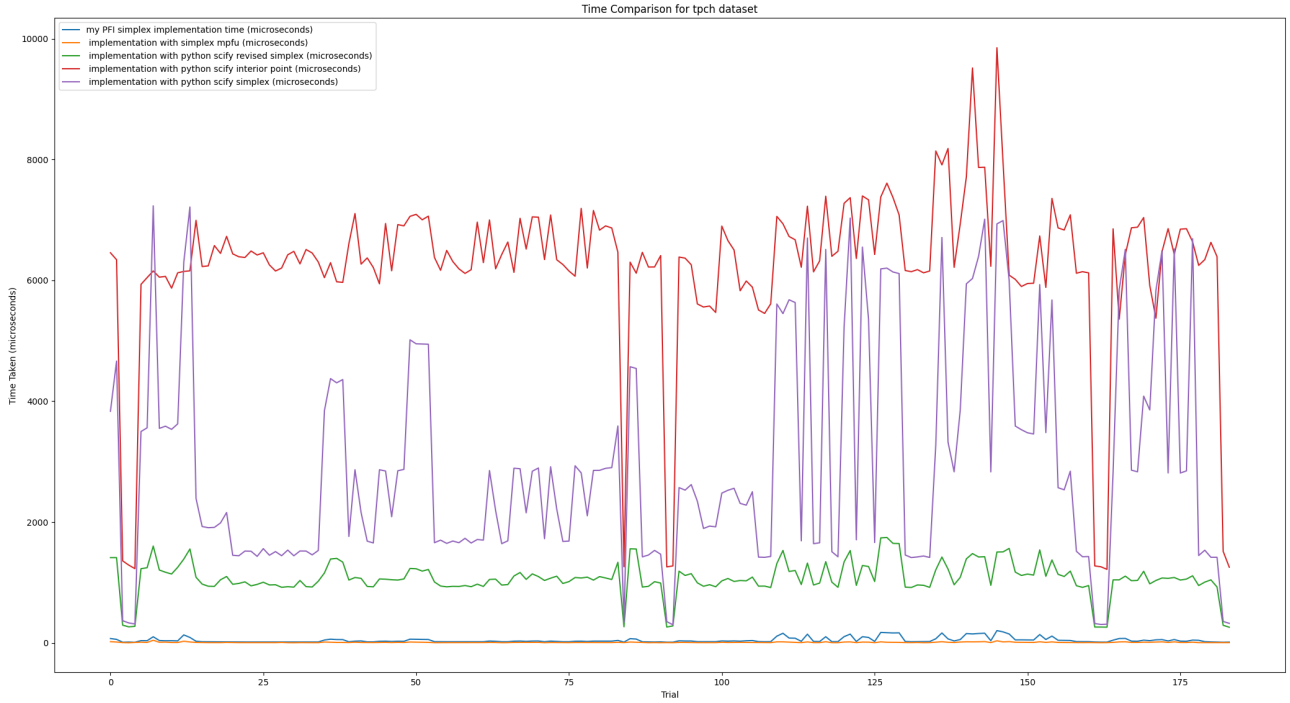


Figure 3.7: A graph comparing the time performance of two of our solvers with scipy solver solving the tpch dataset

Table 3.2: Statistics about JOB dataset

Variable	Min	Median	Mean	Max
LP size	2.00	54.00	94.56	475.00
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 Scipy	650	931	959	1802
Solution Time Cplex	97.04	184.06	190.66	415.56
Solution Time Tableau	2.00	4.00	12.15	4274.00
Solution Time PFI	2.000	6.000	8.316	65.000
Solution Time MPFI	1.000	4.000	5.517	68.000
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

Table 3.3: Number of LPs or Queries Solved by Hour for the JOB dataset

Method	Number of LPs/Queries
Revised Simplex MPFU Umbra	906,607,929
Tableau Simplex	1,400,923,787
Revised Simplex PFI	1,287,140,216
Scipy (method highs)	4,069,108
Cplex	17,849,851

Table 3.4: 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

problems.

We will also analyse and compare the time performance for our implementation solvers, Umbra's revised simplex with the Middle PFI update variant, Cplex and

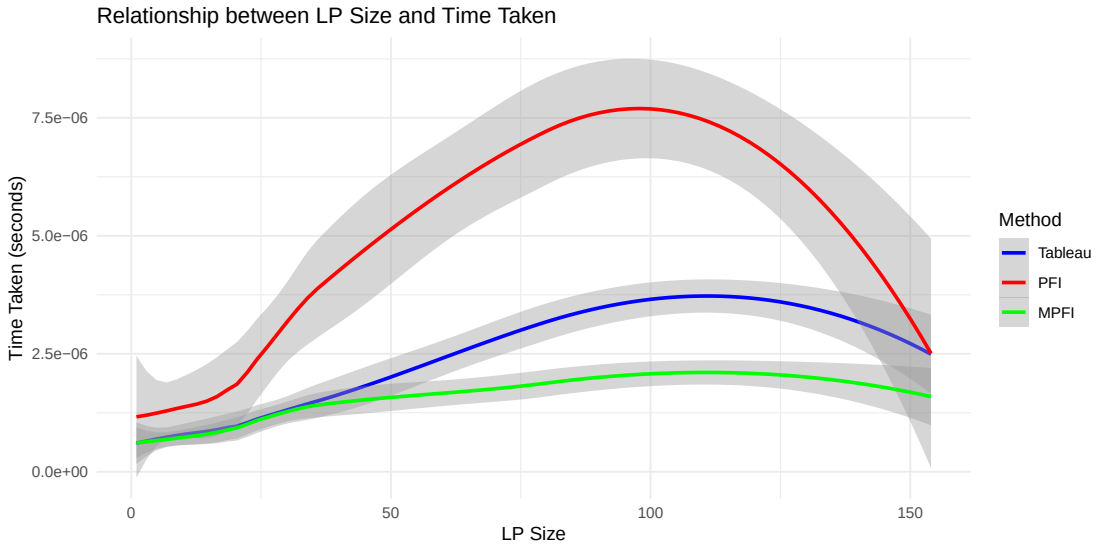


Figure 3.8: A graph comparing the time performance of two of our solvers with scipy solver solving the tpcds dataset

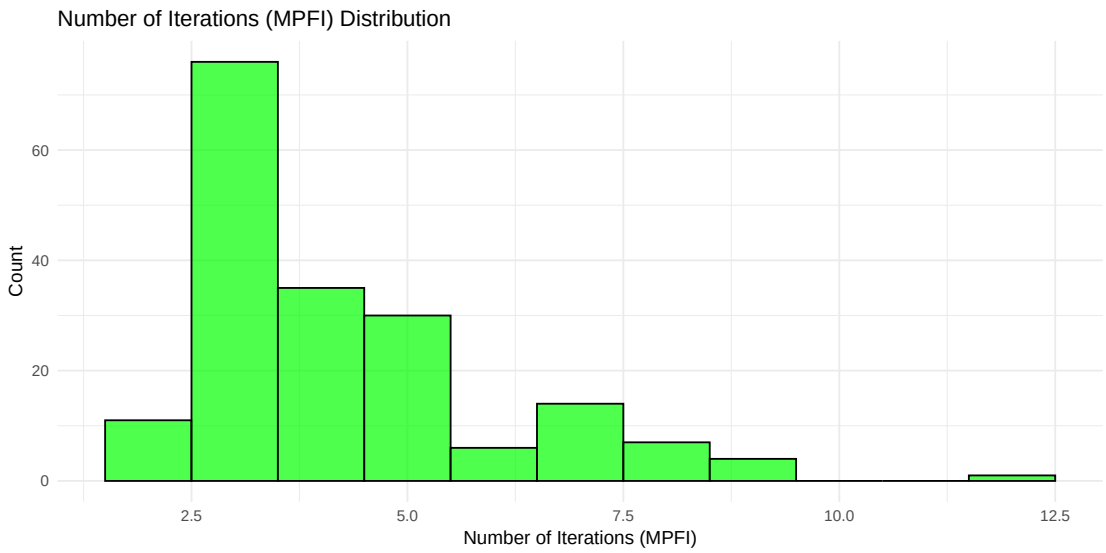


Figure 3.9: Boxplot for number of iterations for MPFI for TPCDS dataset

HiGHS.

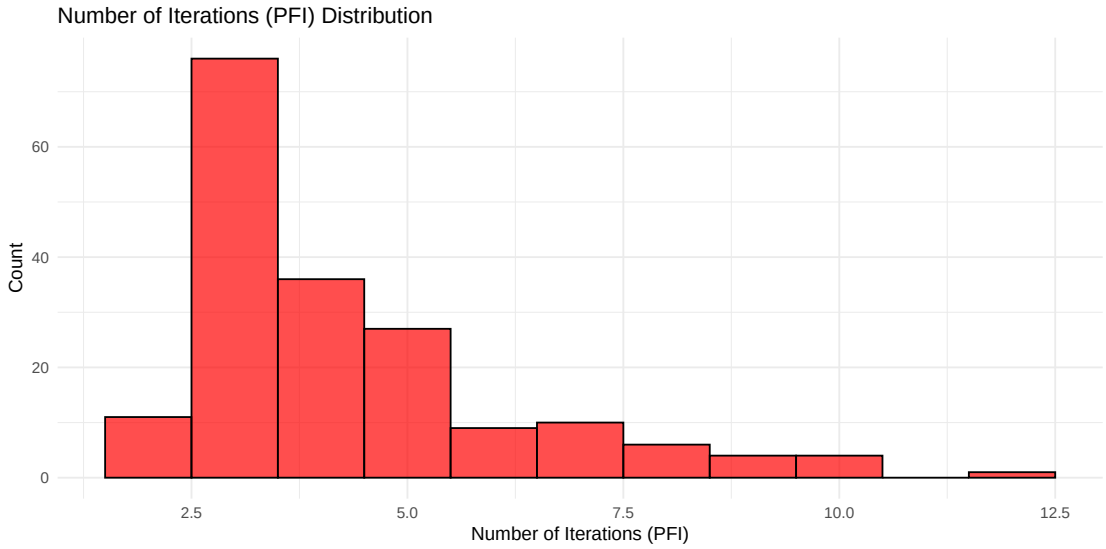


Figure 3.10: Boxplot for number of iterations for PFI for TPCDS dataset

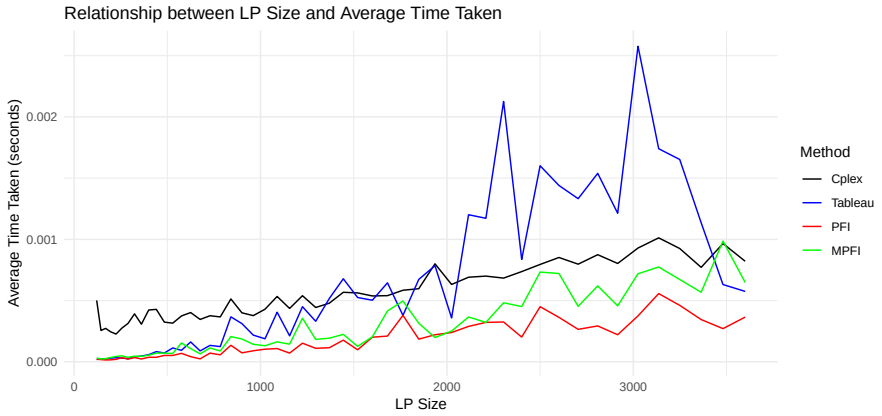


Figure 3.11: Relation between LP size and time for the 3 simplex solvers and Cplex for randomly generated dataset with increasing LP size.

3.3.1 Analysis JOB dataset

Structure and properties

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

A 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

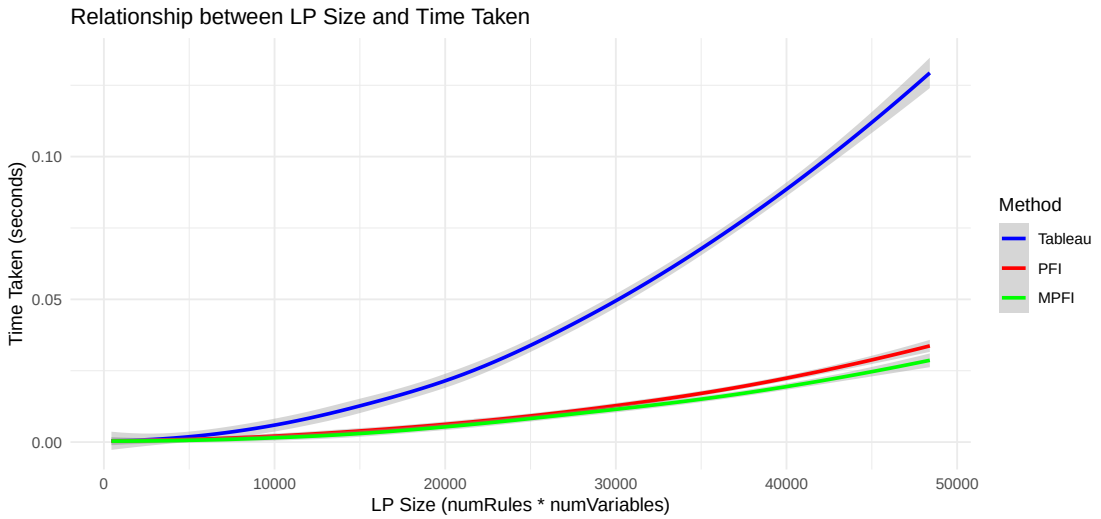


Figure 3.12: Relation between LP size and time for the 3 simplex solvers for the randomly generated dataset with increasing LP size ranging up to 220 rules and 220 variables.

collected about the JOB dataset in the Table 3.2 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 3.2 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 3.3, we can see that the Tableau solver outperforms the two other revised simplex solvers and Cplex. It records the largest Query-Per-Hour metric, around 1.4`billion` queries per hour. The time profiles of the three solvers show that for the queries that have an LP size less than 250 (we define the LP size as the product of the number of variables and the number of rules this LP has $\texttt{lp\_size} = m \times n$) Tableau is the best out of the three. However, from $\texttt{lp\_size} \geq 250$ PFI outperforms the tableau method. Meanwhile, the MPFI delivers the worst execution time out of the three solvers, for this JOB dataset of small queries.

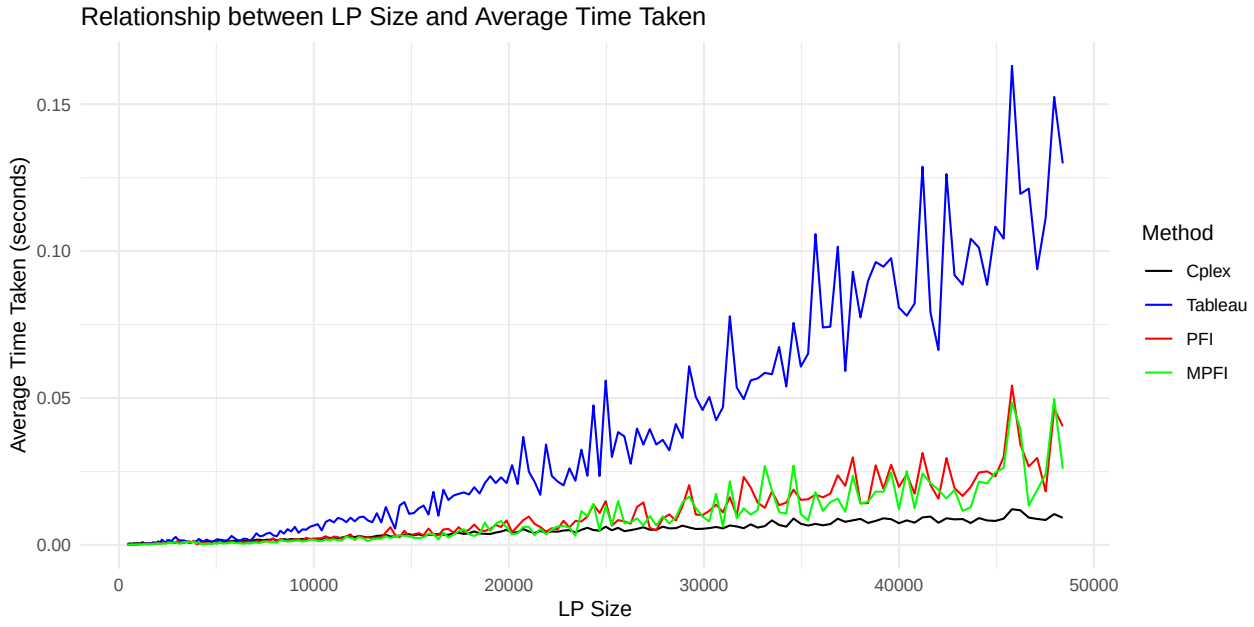


Figure 3.13: Relation between LP size and time for the 3 simplex solvers with CPLEX randomly generated dataset with increasing LP size ranging up to 220 rules and 220 variables.

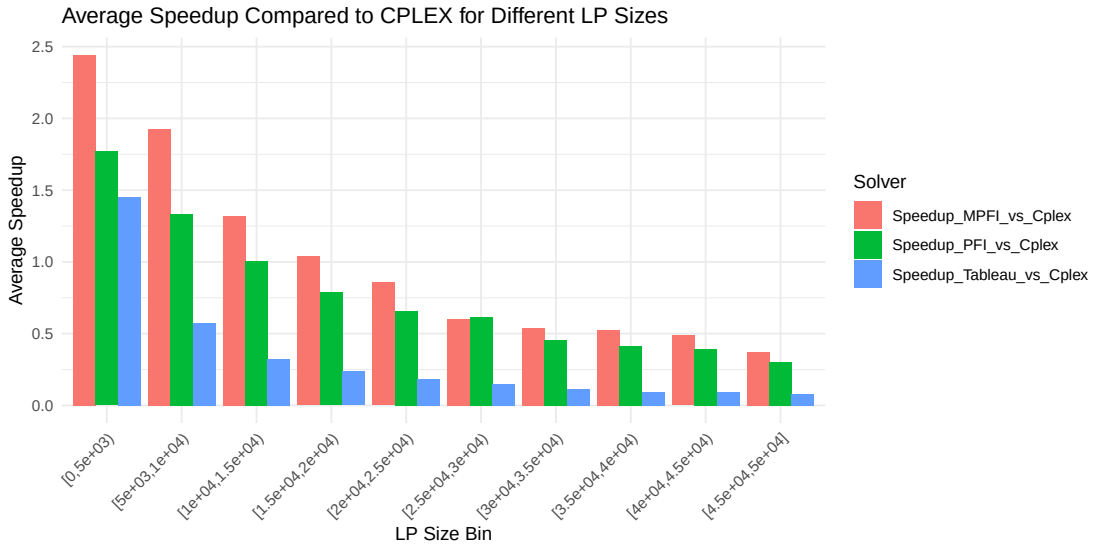


Figure 3.14: Speedup of the three solvers (how many times faster they are) compared to CPLEX for the randomly generated dataset for increasing LP size

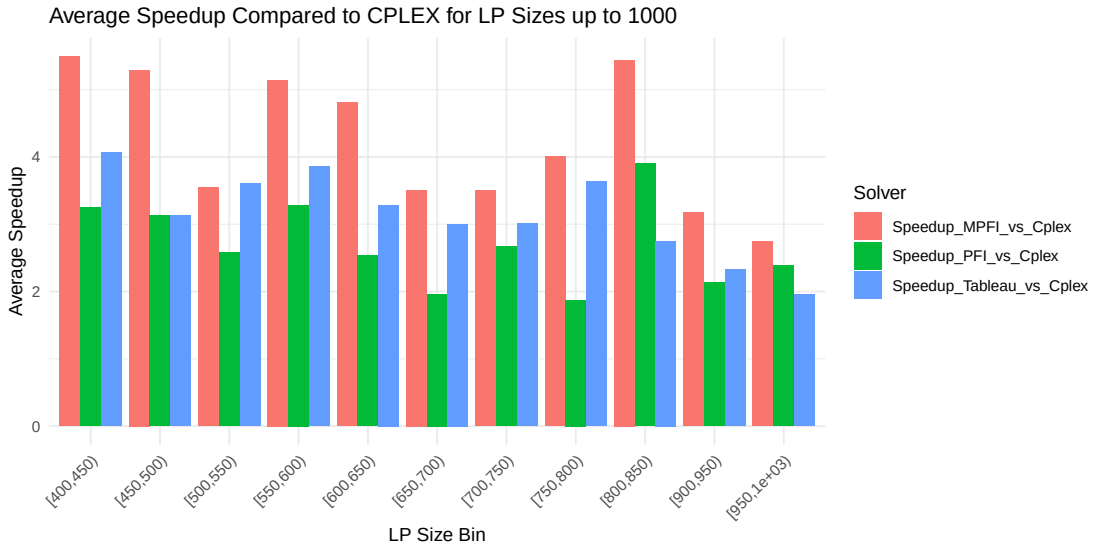


Figure 3.15: Speedup of the three solvers (how many times faster they are) compared to CPLEX for the randomly generated dataset for the LP size up to 1000

Also, Cplex is slower than all other solvers for this dataset.
These results may be due to several reasons. We think 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, being a commercial solver, is designed to handle large-scale LPs. To do this, it incorporates many (in our case unnecessary) advanced techniques like long initialization phases, presolving, where it attempts to simplify the LP, analyze it, 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 very basic, 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 for this reason.
- Overhead of revised simplex methods: in the revised simplex method 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 representaion of the tableau, but in 1D vector, which may be more cache-friendly. The Revised Simplex method updates the matrix by enqueing 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.

It's worth noting that while the Tableau Simplex might be faster for some small LPs, it can be much slower than the Revised Simplex or CPLEX for larger or more complex problems. This is what we will discuss in the next part of the analysis.

3.3.2 Analysis of experiments on randomly generated LPs

For small LP sizes, we observed that a basic implementation like tableau simplex ouperform the revised simplex implementations as well as Cplex. However, for larger LPs we come across other results. From figure 3.12 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 3.13 that Cplex starts outperforming our solvers.

3.3.3 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 Evaluation

4.1 Strengths

4.1.1 Performance

4.1.2 Accuracy

4.1.3 Stability

4.2 Limitations

4.3 Future Work

5 Conclusion

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.

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

[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.

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

[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.

[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.

[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.