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

Contents

1 Introduction

Our aim with this project is to investigate and compare different methods and techniques to solve small linear programming problems representing among others 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 [Ngo22], which is why we want a practical estimate to choose the best from data plans to run efficient queries. For this purpouse, we will introduce a formal description of the cardinality estimation problem, represent it in the form of a packing linear programming problem with the intention of maximizing the size of the query under some constraints. 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.

2 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.

2.1.1 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:

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

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.

2.1.2 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 LP and its dual problem on the right:

$$\begin{array}{rlcrl} \text{maximize} & \mathbf{c}^T\mathbf{x} & & \text{minimize} & \mathbf{b}^T\mathbf{y} \\ \text{subject to} & \mathbf{A}\mathbf{x} \le \mathbf{b} & \longrightarrow & \text{subject to} & \mathbf{A}^T\mathbf{y} \ge \mathbf{c} \\ & \mathbf{x} \ge 0 & & & \mathbf{y} \ge 0 \end{array}$$

Geometric Interpretation

The linear programming problem can be understood geometrically as follows: The linear constraints constitute the vertices (corners) of a polytope defined by the feasible region the problem. The simplex algorithm starts at an initial vertex and moves along the edges of the polytope to vertices with better objective values until the optimal solution is reached.

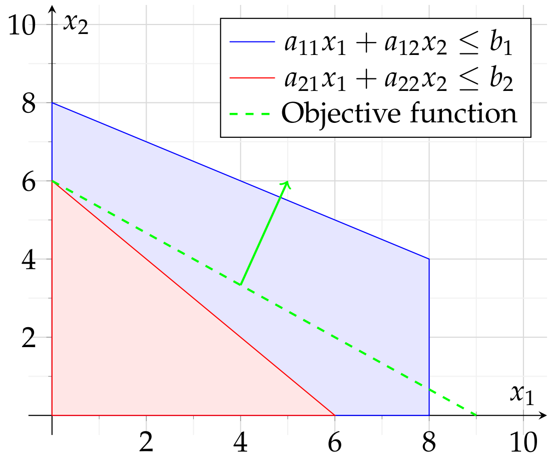


Figure 2.1: Graphical representation of the LP problem with directional arrows.

2.1.3 Feasibility, unboundedness

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

The algorithm

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. Conversely, a minimization problem is optimized when the slack variables are "stuffed," minimizing the true variables' effect on the objective function.

We already defined the concepts of feasibility and unboundedness. We then will define what a basis is. A feasible basis is a set containing the basic variables and their values. Basic variables are set to 0 in the linear constraints. For example an initial feasible basis for the problem, is the set of the decision variables. Meaning we set all the decision variables in the linear constraints to zero and calculate the nonbasis variables' values

The simplex method is first initialized by an initial feasible solution $\bar{\mathbf{x}}$, which is a vector of nonnegative numbers. 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. This is how a simplex algorithm broadly looks like:

We call this a feasible dictionary[Chv83], or a feasible tableau. This is apparent, since all values in the RHS are positive, and the constraint matrix A also has positive coefficients.

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.

- 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

Algorithm 1 Simplex Algorithm

```
procedure SIMPLEX(c, A, b)
    Initialize a feasible basic solution
    if no entering variable with positive reduced cost exists then
        return "Optimal solution found"
    end if
    if no positive pivot element in the column then
        return "Unbounded"
    end if
    Choose a leaving variable using the minimum ratio test
    Perform a pivot operation
    Update the basic and non-basic variables
    return current basic solution
end procedure
```

metrics like the largest increase in the objective function, or the largest coefficient. Bland'S rule has been proved to guarantee termination.

- The choice of the leaving variable: we choose a basic variable to leave the basis: we do this by performing a ratio test.

Termination

The runtime complexity

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. 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. 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].

2.1.5 Time complextity analysis of one iteration of the tableau simplex algorithm

2.1.6 The interior point method

2.1.7 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 costlywe 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, and elucidate the equivalency between Tableau and Revised simplex, and introduce the speedup the latter brings. Given a linear programming problem in standard form:

$$\begin{aligned} \text{maximize} \quad & {}^{T} \\ \text{subject to} \quad & \mathbb{A} = \\ & \geq \end{aligned}$$

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:

$$x_B = x_B - \theta d$$

$$x_j = \theta$$

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

Repeat the optimality test, and if necessary, the pivot operation, until an optimal solution is found or the problem is determined to be unbounded.

This proof elucidates that at every stage of the simplex method, we only have to track the following, and with this only we are able to "recreate" exactly the tableau at each step, without performing the costly pivotting operation:

- 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 2.1.7.
- the current values of the basic variables, or the current basic feasible solution $x_B = B^{-1}b$

In practical terms, we only update the basis matrix every iteration, and we will be able to use it to perform all the steps needed to update our problem data and go on to the next iteration. The algorithm is elaborated in 2.1.7. 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.

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, making it computationally more efficient for large-scale problems. Such tools are called update methods, and we will later describe the Product Form Inverse (PFI), the modified PFI and Forrest-Tomlin update method.

Finally, in our implementation we also use the Compact Column Representation (CCR) format to store matrices, see 3.1.3 of the, which greatly benefits the performance of the implementation.

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 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 2.1.7. 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 2.1.7 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*.

The modified product form inverse

Forrest-Tomlin update form

2.1.8 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.

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 2.1.2 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.

Variables

Objective

Constraints

2.1.9 Other use cases and techniques

We will focus on one study [HH15].

2.1.10 Other techniques

The primal simplex method starts from a trial point that is primal feasible and iterates until dual feasibility. The dual simplex method starts from a trial point that is dual feasible and iterates until primal feasibility. ALGLIB implements a three-phase dual simplex method with additional degeneracy-breaking perturbation:

- Forrest-Tomlin updates for faster LU refactorizations
- A bound flipping ratio test (also known as long dual step) for longer steps
- Dual steepest edge pricing for better selection of the leaving variable
- Shifting (dynamic perturbations applied to cost vector) for better stability

2.2 State-of-the-art LP solvers

Here we will discuss alternative approaches that are used today to solve LPs.

2.2.1 HIGHS Scipy

2.2.2 Cplex

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 2.1.8 as well as randomly generated LPs with varying sizes. We use different LP solvers, and different update methods to solves 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

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

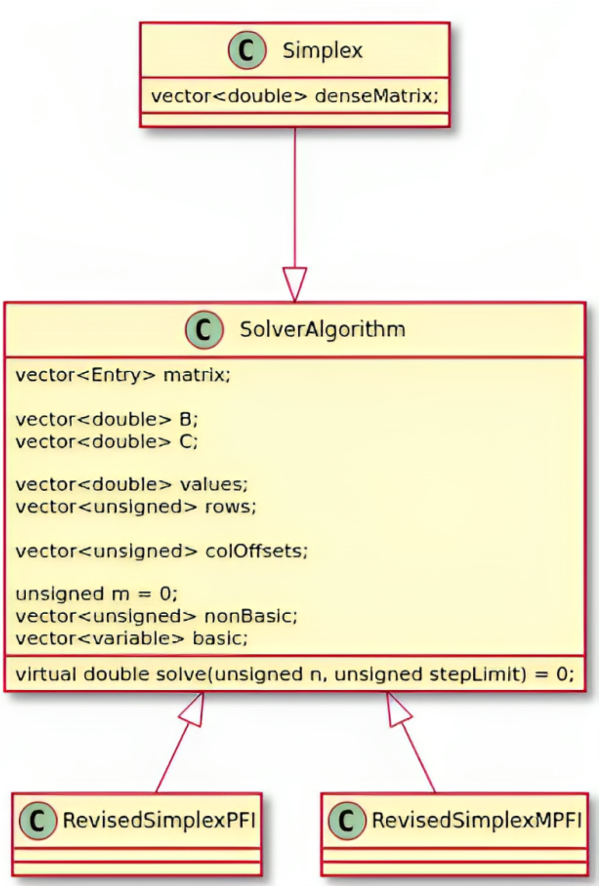


Figure 3.1: An UML Graph explaining the hierarchy of the implementation

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 ← findPivotColumnCandidates()
    if no entering variable found then
        print "Optimal value reached."
        return z
    end if
    pivotColumn ← enteringVars[0]
Step 2: Find pivot row, or leaving variable using the ratio test
    pivotRow ← findPivotRow(pivotColumn)
    if no leaving variable then
        print "The given LP is unbounded."
        return ∞
    end if
Step 3: Update the tableau using pivotting and update the objective function value
    doPivotting(pivotRow, pivotColumn, z)
Goto Step 1
```

Sparse Matrix

In our dataset, we deal with sparse matrices. We use the 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

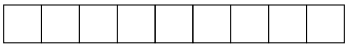
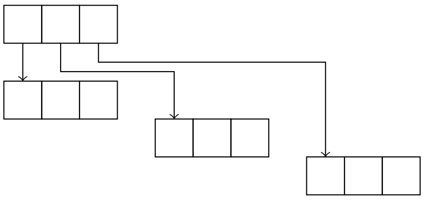


Figure 3.3: Memory Layout of a 2D Dynamic Array



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 personal computer with AMD 4000 series RYZEN, 16GB RAM running Ubuntu. Using the following settings:

- Presolve techniques are not used

- The computed optimal solutions have been validated using the scipy python library.

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 2.1.8. There are two main formats for these problems: `lpp.txt` and `lp.txt`.

The `lpp.txt` file provides a more human-readable representation of the packing LPs, detailing each rule in a clear mathematical format. For instance, it might describe a problem with 8 rules, where each rule is represented as a linear inequality of variables (like v_0, v_1, etc.) with their respective coefficients:

```
lpp:
LP with 8 rules:
v0*0.0540277 + v2*0.0540277 <= 1
v0*0.0540277 + v3*0.0540277 <= 1
...
```

On the other hand, 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 ...
```

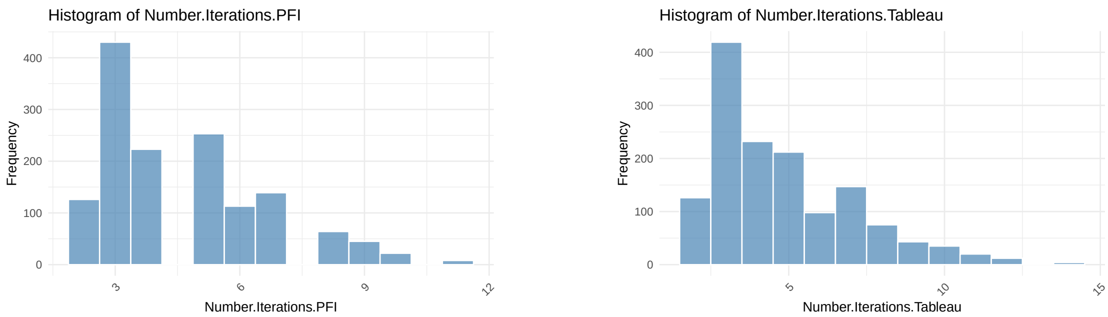
Table 3.1: Benchmarks and workloads.

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

The JOB dataset results

In the following table are some important statistical finds.

This is our results:



(a) Boxplot for number of iterations for PFI for JOB dataset

(b) Boxplot for number of iterations for Tableau for JOB dataset

Figure 3.4: Statistics about the number of iterations needed to solve JOB dataset

TPC-DS results

3.2.2 Results on randomly generated LPs

3.3 Analysis

3.3.1 Dataset Structure

Our dataset stucture: as opposed to what the linear programming research has dealt with, which is very large problems, we are dealing with hundreds of small problems. These are represented in the revised simplex algorithm by sparse matrices but not as sparse as it would have been if the problem was large, small matrices that are not small enough to be dense. (they still have quite a number of non-zeroes).

3.3.2 Analysis of dataset properties

In this subsection we will conduct an analysis of our dataset properties. What are the particularites of the structure of these LP problems, is their any patterns in their solution process. This anaylsis is based on observing the statistical results we obtained from running different solvers on these problems. This will later provide us with insight regarding optimization of these problems.

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, and there are a few reasons for this:

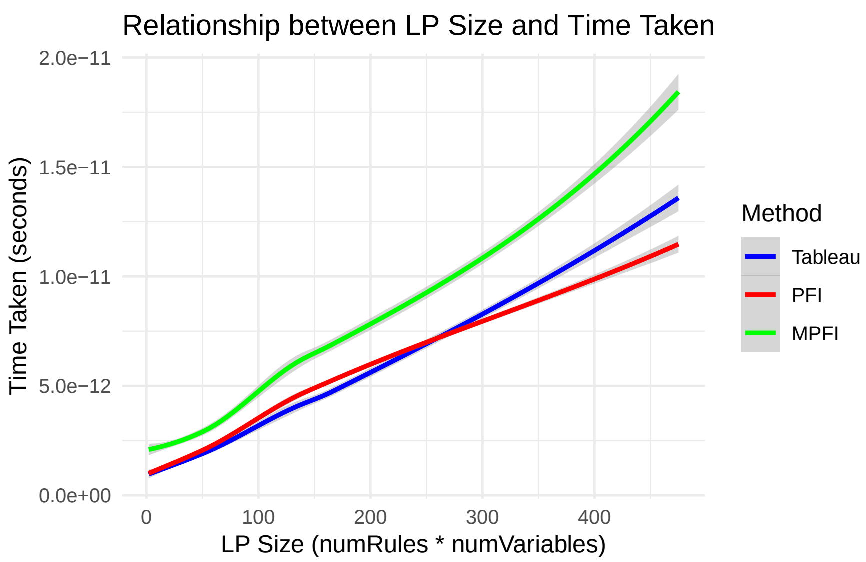


Figure 3.5: Relation between LP size and time for the 3 simplex solvers for JOB dataset.

1. **Overhead from Python and SciPy**: When you use `linprog` from SciPy, there's an overhead associated with Python's interpretation and SciPy's function calls. This overhead might not be significant for small problems, but for larger LPs or when solving multiple LPs, it can add up.

2. **Data Conversion**: `linprog` has to convert the problem data into a format that HiGHS can understand. This conversion can introduce additional computational overhead.

3. **Additional Features**: `linprog` provides a unified interface for multiple solvers, and it might perform some additional checks or operations that are not strictly necessary when you know you're going to use HiGHS.

4. **Version Differences**: Depending on how you installed SciPy and HiGHS, there might be version differences between the HiGHS in SciPy and the standalone HiGHS. Newer versions of solvers often come with performance improvements, so if SciPy's version is older, it might be slower.

5. **Default Parameters**: The default parameters set by `linprog` for HiGHS might not be the most optimal for your specific problem. When using the HiGHS interface directly, you have more control over these parameters.

For these reasons, if performance is critical and you're solving large-scale LPs or

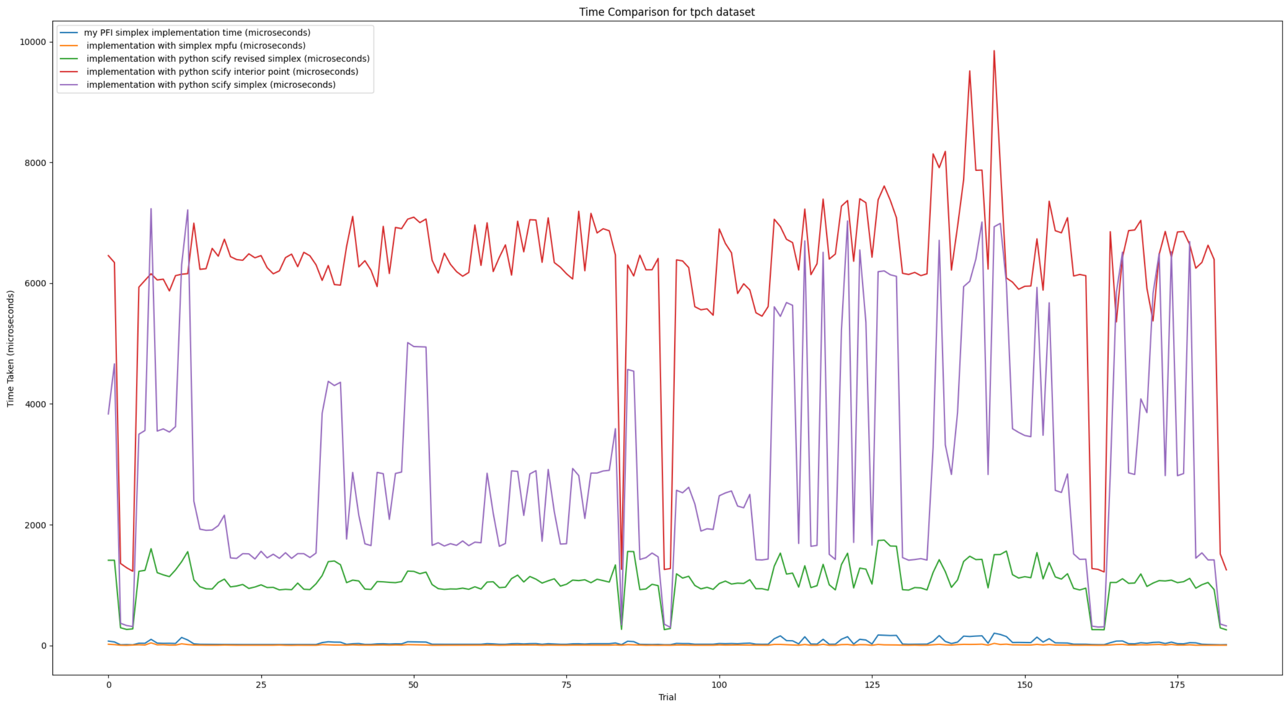


Figure 3.6: A graph comparing the time performance of two of our solvers with scipy solver solving the tpch dataset

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 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

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

solving many LPs, it might be beneficial to use the HiGHS interface directly. However, for many users, the convenience of using `linprog` and its unified interface might outweigh the performance benefits of using HiGHS directly.

4 Evaluation

4.1 Setup

4.1.1 Evaluation metrics

4.1.2 Evaluation baselines

4.2 Results

4.3 Discussion

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.

[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.

[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`.

[Ngo22] H. Q. Ngo. “On an Information Theoretic Approach to Cardinality Estimation (Invited Talk).” In: *25th International Conference on Database Theory (ICDT 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2022.

[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.