Chapter 1

Introduction

In this project, we implement the simplex method, interior point method and the barrier method for the linear programming problem

$$\begin{aligned} \text{minimize} \quad & \mathbf{c}^T\mathbf{x} \\ \text{subject to} \quad & \boldsymbol{A}\mathbf{x} \leq b \\ & \mathbf{x} \geq 0 \end{aligned} \tag{1.1}$$

where $\mathbf{c}$ and $\mathbf{x}$ are vectors in $\mathbb{R}^n$, $\mathbf{b}$ is a vector in $\mathbb{R}^m$ and $\boldsymbol{A}$ is a large sparse matrix in $\mathbb{R}^{m\times n}$. $\boldsymbol{A}$ is full rank, i.e., rank $= m$. Throughout this report, we refer to the linear program in this form as the primal problem.

The goal of the project is to compare the performance of simplex, interior point and barrier methods on different sparse data sets, both in terms of the accuracy of the solution and the running time.

We test our programs on the datasets present in the table. These datasets are available on Tim Davis' sparse matrix repository [2]. The datasets are preprocessed to comply with the project specifications, i.e., $\mathbf{c}$ is a vector of all ones and $\mathbf{x} \geq 0$.

Problem	Size
lpi_itest6	11×17
lpi_chemcom	288×744
lp_agg3	516×758
lp_80bau3b	2262×12061

Table 1.1: Problem sizes

The report is organized as follows. In chapter 1, we introduce the simplex method in the two phase format, discussing interesting implementation details. Chapter 2 describes the predictor corrector algorithm and the different solvers used to solve the linear system in each iteration of predictor corrector. Chapter 3 studies the barrier method for three different schemes: Newton's method, BFGS and limited memory BFGS.

The concluding chapter of this report compares the three different classes of methods in terms of accuracy of the solution and the speed of the algorithms.

We have organized this project as follows:

- Huda Nassar worked on the simplex method
- Varun Vasudevan worked on the interior point method
- Shuvra Nath worked on the barrier method

Note that all the results present in this report are obtained after running our programs on Purdue CS machines.

Chapter 2

Simplex Method

In late 1940s, George Dantzig developed the simplex method and that was the mark of a start of an era in optimization [3]. Economists have reckoned on this method to formalize their models and analyze them. In this chapter, we describe the simplex method analyzing the bottleneck of the algorithm. Theoretically, the algorithm may take an exponential amount of time relative to the number of constraints, but practice has shown that the simplex Method can converge in very few steps in many cases. Aside from aiming at minimizing the number of iterations in the simplex method, Linear Algebra enthusiasts have also made considerable achievements to make each iteration of the simplex method run faster. In this chapter we highlight some of these strategies. The organization of this chapter is as follows

1. Description Of The Simplex Method
2. The Two Phase Approach
3. Solving Three Systems In Each Iteration
4. Choice Of The Entering Index
5. Results And Analysis

2.1 Description Of The Simplex Method

2.1.1 Intuition Behind The Algorithm

The bedrock underlying behind the simplex algorithm is present in the simple idea of checking possible solutions at the boundary of the constraints region. Geometrically speaking, the constraints produce one geometric bounded shape in space (if a feasible solution exists). The Simplex Algorithm is based on examining the vertices of this shape to find the optimal point. Figuratively, the algorithm hops from one vertex to another until certain constraints are satisfied or the optimal solution is found. In fact, the reason that the Simplex

Algorithm may take a long amount of time is due to this reason of hopping from one vertex to another on the boundary.

2.1.2 Algorithm

The Simplex Algorithm is an iterative algorithm that tries to find a better solution at each iteration. Algorithm 1 shown below represents a brief look of the Simplex Algorithm we use in this project. The benchmark we use to compare our method against is MATLAB's existing `linprog` function.

Algorithm 1 The Simplex Algorithm

```
For a given set of indices for $\mathcal{B}, \mathcal{N}$ do
while Not converged do
    Solve $\mathcal{B}\boldsymbol{x}_B = \boldsymbol{b}$
    Solve $\mathcal{B}^T\boldsymbol{l} = \boldsymbol{c}_B$
    Compute $\boldsymbol{s}_N = \boldsymbol{c}_N - \mathcal{N}^T\boldsymbol{l}$
    if $\boldsymbol{s}_N \geq \boldsymbol{0}$ then
        Stop! Solution found
    else Select entering parameter $\boldsymbol{q} \in \mathcal{N}$ such that $\boldsymbol{s}_q < 0$
    end if
    Solve $\mathcal{B}\boldsymbol{y} = \boldsymbol{a}_q$
    if $\boldsymbol{y} \leq 0$ then
        Stop! Problem is unbounded
    end if
Select $\forall i$ such that $\boldsymbol{y}_i > 0, \boldsymbol{x}_q = min\frac{\boldsymbol{x}_B(i)}{\boldsymbol{y}(i)}$
end while
```

The bottleneck of the algorithm happen in three places:

- Choosing the indices of $\mathcal{B}$
- Solving three linear systems at each iteration
- Choosing a suitable entering index

To pick the initial set of indices, we use the two phase approach and then discuss our implemntation details. To solve the three systems, we discuss possible alternatives, namely computing the $\boldsymbol{LU}$ decomposition of $\mathcal{B}$, or updating the $\boldsymbol{LU}$ factors. To choose the entering index, we discuss the tradeoff between imposing more heuristics to decrease the total number of iterations versus the time each iteration consumes.

2.2 The Two Phase Approach

2.2.1 Description

The goal of the two phase approach is to find a non-singular initial basis matrix $\mathcal{B}$. Phase 1 is responsible for finding a feasible basis, whereas phase 2 solves the initial system with the output taken from phase 1. The problem that phase 1 solves is the following

$$\text{Minimize } \zeta' = \boldsymbol{u}^T \boldsymbol{v} \tag{2.1a}$$

subject to

$$\begin{aligned} A\boldsymbol{x} + E\boldsymbol{v} &= \boldsymbol{b}\,, && (2.1b)\\ \boldsymbol{x} &\geq \boldsymbol{0}\,, && (2.1c)\\ \boldsymbol{v} &\geq \boldsymbol{0}\,. && (2.1d) \end{aligned}$$

Where $\boldsymbol{u}$ is the vector of all ones, and $\boldsymbol{E}$ is a diagonal matrix defined as:

$$\boldsymbol{E}(i,i) = \begin{cases} +1 \text{ if b(i)} \geq 0 \\ -1 \text{ if b(i)} < 0 \end{cases}$$

The reason behind the success of the two phase approach goes back to the way it is crafted. We can easily see that the solution $\begin{bmatrix} \mathbf{0} \\ |\mathbf{b}| \end{bmatrix}$ satisfies the set of constraints stated above. Thus, we have a trivial initial basis $\mathcal{B}$. The entries stored in the vector $\boldsymbol{v}$ are called the artificial variables and phase 1 aims at minimizing their influence. The optimal value for this problem gives insight on the feasibility of the original problem. Interestingly, we observe that if ζ' achieves an optimal value of 0, then there is no influence from the artificial variables on the problem leading us to deduce that the the initial problem is feasible.

2.2.2 Implementation Details

Phase 1 of the simplex method plays a crucial role. Aside from providing the phase 2 with a feasible starting point, if set up properly and solved efficiently, it can bring the algorithm to take considerably less iterations. While working on this project, we shall recall that the initial constraints are inequality constraints. Phase 1 takes as input a matrix of standard form with an identity matrix augmented to represent slack variables. Unquestionably, we can see that this might cause some of the columns in the matrix to be duplicated. We avoid duplicating the columns by adding artificial variables whenever the entries in $\boldsymbol{b}$ are non-negatives. To illustrate our scheme, we present the following example

If initially we are given:

$$\boldsymbol{A} = \begin{bmatrix} 5 & 0 & 1 & 4 \\ 5 & 1 & 0 & 4 \\ 0 & 3 & 1 & 4 \end{bmatrix}, \text{ and } \mathbf{b} = \begin{bmatrix} -1 \\ -2 \\ 3 \end{bmatrix}$$

The new augmented system we use in phase 1 is $\boldsymbol{A}_{new}$, where the red colored values represent the basis.

$$\boldsymbol{A}_{new} = \begin{bmatrix} 5 & 0 & 1 & 4 & 1 & 0 & 0 & -1 & 0 \\ 5 & 1 & 0 & 4 & 0 & 1 & 0 & 0 & -1 \\ 0 & 3 & 1 & 4 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

In this example, the number of negative values in $\boldsymbol{b}$ relative to the problem size is not very small. Nevertheless, in the problems we are given in this project, the number of negatives in $\boldsymbol{b}$ is always a very small number, so performing this step saves a lot on time and memory.

2.3 Solving Three Systems At Each Iteration

As part of each iteration in the Simplex Algorithm, we have to solve 3 systems all involving the matrix $\mathcal{B}$. In this section, we describe two ways to solve the systems; both ways involve the $\boldsymbol{LU}$ decomposition of the matrix $\mathcal{B}$. We then end with a comparison between the two approaches. Since in this project we are using MATLAB as our scientific computing package to solve the problems, it is important to mention that MATLAB's backslash operator is trivially one option one could think of. In our case, we know that the matrices we are given are sparse and thus using the $\boldsymbol{LU}$ decomposition is one way to approach the four problems.

2.3.1 LU Decomposition

Since the three systems in each Simplex iteration involve the same matrix $\mathcal{B}$, then computing the $\boldsymbol{LU}$ decomposition of $\mathcal{B}$ once at each iteration appears a natural option. Solving the three systems could then be written using the following commands:

```
x_B = U\(L\(P*b));
l_vec = P'*(L'\(U'\c_B));
d = U\(L\(P*a_q));
```

Compellingly, MATLAB's backslash operator is so powerful that storing the matrix `U\(L\(P)` is not a good idea. This goes back to the way `'\'` is implemented, where its performance on sparse triangular matrices is significantly faster than a matrix-vector product.

In our project, we have four problem of different sizes, so it is vital to get a closer inspection of how the `lu` function behaves. Figure 2.2 represents a semilog plot of the time taken by the `lu` function to execute on problems of different sizes. All problems used in this experiment are taken from [2]. We can easily see that there is a linear relation between the size of the matrix and the time taken by MATLAB's `lu`. But linearity is not all what we see, we can also easily note that the constant for the linear term is pretty small. In fact, spotting figure 2.1 which shows the regular plot, of the same data in figure 2.2 shows that the slope of this line is very small. This experiment includes nine datasets of varying sizes taken from Tim Davis's website. When obtaining the slope of this specific line in this experiment, the value obtained was on the order of 10^{-6}.

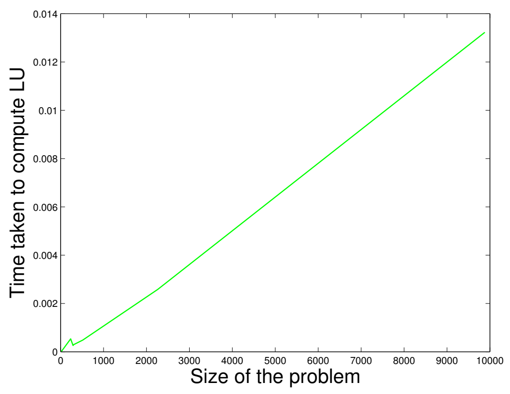

Figure 2.1: Regular plot

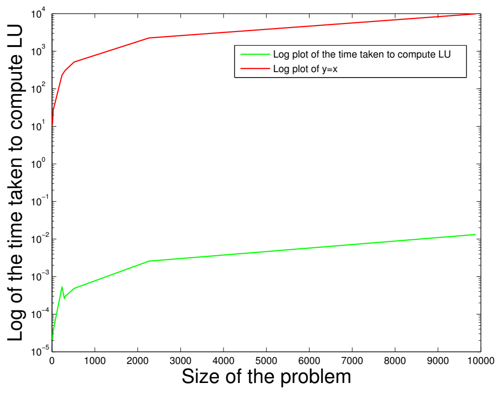

Figure 2.2: Semilog plot

Figure 2.3: Figure 2.1 shows a plot of time taken to perform `lu` versus the problem size. Figure 2.2 shows a semilog plot of time taken to perform `lu` versus the problem size. Red line represents the semilog plot of the line $y = x$.

2.3.2 Updating The LU Factorization

Another observation about the matrix $\mathcal{B}$, is that it only changes one column vector at each iteration. One column leaves the basis and another one enters. This could be expressed as a rank-1 update of the $\boldsymbol{LU}$ factors of $\mathcal{B}$. In this section, we describe the simple rank-1 update, move on to the rank-k update, and then analyze the rank-k update in the context of the simplex algorithm.

2.3.2.1 General LU rank-1 update

The problem we address here is the following: Given the $\boldsymbol{LU}$ decomposition of a matrix $\boldsymbol{A}$, find some sort of factorization of the rank-1 update of $\boldsymbol{A}$. In the following, w.l.o.g. denote $\boxed{1}$ as a rank 1 matrix. Note that if $\boxed{1}$ stays from one step to another, this doesn't necessarily mean that it is still the same matrix, yet it still qualifies as a rank 1 matrix.

$$\begin{gathered}
\text{Let } \boldsymbol{PA} = \boldsymbol{LU} \\
\text{Let } \boldsymbol{A}' = \boldsymbol{A} + \boxed{1}, \text{ then} \\
\boldsymbol{PA}' = \boldsymbol{PA} + \boxed{1} \\
\Rightarrow \boldsymbol{PA}' = \boldsymbol{LU} + \boxed{1} \\
\Rightarrow \boldsymbol{L}^{-1}\boldsymbol{PA}' = \boldsymbol{U} + \boxed{1}
\end{gathered}$$

Interestingly, the righthand side has a special format. Let $\boldsymbol{U}^+ = \boldsymbol{U} + \boxed{1}$. It is an upper triangular matrix with a spike. Applying simple permutations on $\boldsymbol{U}^+$ leads to a matix that is upper triangular except for the last row. Call this matrix $\boldsymbol{G} = \Pi^T \boldsymbol{U}^+ \Pi$. Decomposing $\boldsymbol{G}$ into its $\boldsymbol{LU}$ factors is obvious and we provide an illustration below.

Let $\boldsymbol{U}^+ = \begin{bmatrix} * & * & * & * & * \\ & * & * & * & * \\ & * & * & * & * \\ & * & & * & * \\ & * & & & * \end{bmatrix} \Rightarrow$ for a certain Π, $\boldsymbol{G} = \begin{bmatrix} * & * & * & * & * \\ & * & * & * & * \\ & & * & * & * \\ & & & * & * \\ & * & * & * & * \end{bmatrix}$.

Then by permuting the second column and placing it at the location of the fifth column and then permuting the rows to preserve an upper triangular form, we reach to a matrix $\boldsymbol{G}$ that has the form presented above.

To illustrate the computation of the $\boldsymbol{LU}$ factors of $\boldsymbol{G}$,

Let $\boldsymbol{G} = \begin{bmatrix} g_{1,1} & g_{1,2} & g_{1,3} & g_{1,4} & g_{1,5} \\ & g_{2,2} & g_{2,3} & g_{2,4} & g_{2,5} \\ & & g_{3,3} & g_{3,4} & g_{3,5} \\ & & & g_{4,4} & g_{4,5} \\ & g_{5,2} & g_{5,3} & g_{5,4} & g_{5,5} \end{bmatrix}$, Then the $\boldsymbol{LU}$ factors of $\boldsymbol{G}$ are:

$\boldsymbol{L} = \begin{bmatrix} 1 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & & 1 & \\ & l_{5,2} & l_{5,3} & l_{5,4} & 1 \end{bmatrix}$, and $\boldsymbol{U} = \begin{bmatrix} g_{1,1} & g_{1,2} & g_{1,3} & g_{1,4} & g_{1,5} \\ & g_{2,2} & g_{2,3} & g_{2,4} & g_{2,5} \\ & & g_{3,3} & g_{3,4} & g_{3,5} \\ & & & g_{4,4} & g_{4,5} \\ & & & & u_{5,5} \end{bmatrix}$.

Equating the last row of the system $\boldsymbol{G} = \boldsymbol{LU}$, yields to solutions for the last row of $\boldsymbol{L}$ and the entry $u_{5,5}$ in $\boldsymbol{U}$.

2.3.2.2 General LU rank-k update

The general $\boldsymbol{LU}$ rank-k update follows directly from the rank-1 update. We dervie the rank-1, and rank-2 updates of a general matrix $\boldsymbol{A}$, and then present the the general case:

Let $\boldsymbol{P}_1\boldsymbol{A}_1 = \boldsymbol{L}_1\boldsymbol{U}_1$. Then, if $\boldsymbol{A}_2 = \boldsymbol{A}_1 + \boxed{1}$ Then, $\boldsymbol{P}_1\boldsymbol{A}_2 = \boldsymbol{P}_1\boldsymbol{A}_1 + \boxed{1}$
Thus, $\boldsymbol{P}_1\boldsymbol{A}_2 = \boldsymbol{L}_1\boldsymbol{U}_1 + \boxed{1}$ and performing the steps discussed in 2.3.2.1, we reach:

$$\boldsymbol{P}_1\boldsymbol{A}_2 = \boldsymbol{L}_1\Pi_1\boldsymbol{L}_2\boldsymbol{U}_2\Pi_1^T$$

If $\boldsymbol{A}_3 = \boldsymbol{A}_2 + \boxed{1}$ Then, $\boldsymbol{P}_1\boldsymbol{A}_3 = \boldsymbol{P}_1\boldsymbol{A}_2 + \boxed{1} = \boldsymbol{L}_1\Pi_1\boldsymbol{L}_2\boldsymbol{U}_2\Pi_1^T + \boxed{1}$
Thus, $\boldsymbol{L}_2^{-1}\Pi_1\boldsymbol{L}_1^{-1}\boldsymbol{P}_1\boldsymbol{A}_3\Pi_1 = \boldsymbol{U}_2 + \boxed{1}$ and performing the steps discussed in 2.3.2.1, we reach:

$$\boldsymbol{L}_2^{-1}\Pi_1\boldsymbol{L}_1^{-1}\boldsymbol{P}_1\boldsymbol{A}_3\Pi_1 = \Pi_2\boldsymbol{L}_3\boldsymbol{U}_3\Pi_2^T$$

This leaves us with the following general form:

$$\boldsymbol{P}_1\boldsymbol{A}_i\Pi_1\dots\Pi_{i-1} = \boldsymbol{L}_1\Pi_1\boldsymbol{L}_2\Pi_2\dots\boldsymbol{L}_{i-1}\Pi_{i-1}\boldsymbol{L}_i\boldsymbol{U}_i$$

2.3.2.3 LU rank-k update In The Context Of Simplex

What was presented earlier in this section applies to any general rank-k update of a matrix. It is noteworthy to mention that in the context of Simplex, we know the structure of the matrices $\boldsymbol{L}_2\dots\boldsymbol{L}_i$ and $\Pi_1\dots\Pi_{i-1}$.

- In terms of storage: The lower triangular matrices contain ones on the diagonal, and nonzeros on the last row (the last row is not entirely nonzeros, depending on the leaving index). The permutation matrices are of this form: $\{1 : s-1, s+1 : m, s\}$, where s is the leaving index and m is the size of the matrix. Thus, all of these matrices can obviously be stored compactly in vector forms.

- In terms of computation: At each iteration, we need to solve three systems of linear equations. For the sake of the argument in this item, assume we want to solve: $\boldsymbol{A}_i\mathbf{x} = \mathbf{b}$. Having the factors of $\boldsymbol{A}_i$, will allow us to write the following:

$$\begin{gathered}
[\boldsymbol{P}_1\boldsymbol{A}_i\Pi_1\dots\Pi_{i-1}][\Pi_{i-1}^T\dots\Pi_1^T\mathbf{x}] = \boldsymbol{P}_1\mathbf{b}.\\
\text{Let } \hat{\mathbf{x}} = \Pi_{i-1}^T\dots\Pi_1^T\mathbf{x}, \text{ thus:}\\
\boldsymbol{L}_1\Pi_1\boldsymbol{L}_2\Pi_2\dots\boldsymbol{L}_{i-1}\Pi_{i-1}\boldsymbol{L}_i\boldsymbol{U}_i\hat{\mathbf{x}} = \boldsymbol{P}_1\mathbf{b}\\
\hat{\mathbf{x}} = \boldsymbol{U}_i^{-1}\boldsymbol{L}_i^{-1}\dots\boldsymbol{L}_2^{-1}\Pi_1^T\boldsymbol{L}_1^{-1}\boldsymbol{P}_1^T\mathbf{b}
\end{gathered}$$

  We obviously won't compute $\hat{\mathbf{x}}$ by computing inverses, even better, we won't even need MATLAB's backslash to compute the internal terms of the equation. Since we know the structure of $\boldsymbol{L}_2\dots\boldsymbol{L}_i$ and $\Pi_1\dots\Pi_{i-1}$, this will allow us to compute the factors involving them using a vector-vector product. To see why this is the case, note that the inverses of $\boldsymbol{L}_2,\dots,\boldsymbol{L}_i$ constitute of the same matrix with the entries in the last row in opposite sign. We can easily see that multiplying such a matrix by a vector can be written as a dot product.

  Also, keeping the internal components as vectors is crucial since we want to maintain the vector-vector products and not move to matrix-vector product. To see why we would require this, note that the complexity of a matrix-vector product is $\mathcal{O}(n^2)$, and performing it three times in each iteration yields $\mathcal{O}(3n^2)$ (right, there is some abuse of notation here but bare with us, we're keeping the constant for a reason), whereas a vector-vector product requires $\mathcal{O}(n)$. At iteration i, we will perform $3 \times i \times \mathcal{O}(n)$ work which will always be less work than performing a matrix-vector product as long as i is $< n$. Practically, the number of iterations never got to be bigger than the size of the problem.

In addition to the $3 \times i$ vector-vector products we need at the $i'th$ iteration, we will still need to perform the MATLAB backslash operator on $\boldsymbol{U}_i$ and $\boldsymbol{L}_1$, because we don't know what these factors look like.

Although we do not store the multiplications on the righthand side from one iteration to another, we do the multiplication of the permutation matrices Π_i on the lefthand side. That way, to obtain $\hat{\mathbf{x}}$ we only need to perform one additional vector-vector product and not another i vector-vector products.

2.3.3 Which Approach To Use?

Here comes the trickiest part of this project. In this section we compare ***LU*** updates to ***LU*** factorizations. To make a fair judgement, we note the following:

- In each iteration of the fresh LU decomposition, we pay for the following:
  1. `lu` factors computation
  2. Three equations of the form (or similar to): `x_B = U\(L\(P*b));`

- In each iteration of the LU update, we pay for the following:
  1. Computing the ***LU*** factors of $\boldsymbol{G} = \Pi^T \boldsymbol{U}^+ \Pi$
  2. Three times the iteration number of vector-vector products
  3. Remember, the internal vector-vector products will not give us the final answer. Thus, we would still need three computations of the form (or similar to): `x_B = U\(L\(P*b));`

To compare both of these methods, we will drop the third item in the second list and the second item in the first list. Further, we also won't compare the time taken to compute the ***LU*** factors of ***G*** for reasons present in later sections. Thus, we will only compare the `lu` function in MATLAB to computing three times the iteration number of vector-vector products.
To take a closer look at this analysis we run the following simple experiment:

```
clear all;
ntimes = 100;
originalfiles = {'lpi_itest6.mat';...
    'lpi_chemcom.mat';'lp_agg3.mat';'lp_80bau3b.mat'};
for id = 1:numel(originalfiles)
    datasetname = originalfiles{id};
    load(datasetname);
    A = Problem.A;
    b = Problem.b;
    times1 = zeros(ntimes,1);
    times2 = zeros(ntimes,1);
    for i = 1:ntimes
        t1 = tic;
        v = dot(b,b); % b is of the appropriate
        % size of the vector-vector products
        times1(i) = 3*i*toc(t1);
        B = A(:,1:size(A,1)); % B is of the
        % appropriate size we will perform lu on
        t2 = tic;
        [L U P] = lu(A);
        times2(i) = toc(t2);
    end
    subplot(2,2,id);
    plot(times1,'k','LineWidth',1.15);
    hold on;
    plot(times2,'r','LineWidth',1.15);
    legend('3*i*vec-vec','lu');
    title(datasetname,'Interpreter', 'none');
end
```

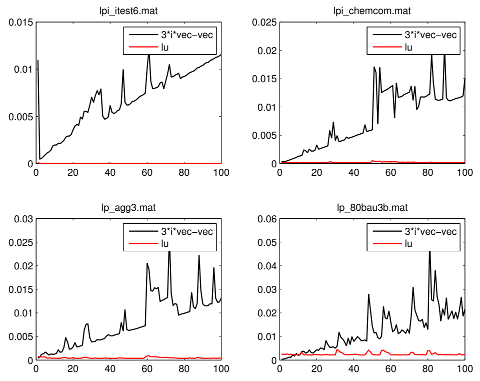

Figure 2.4: The project's 4 datasests

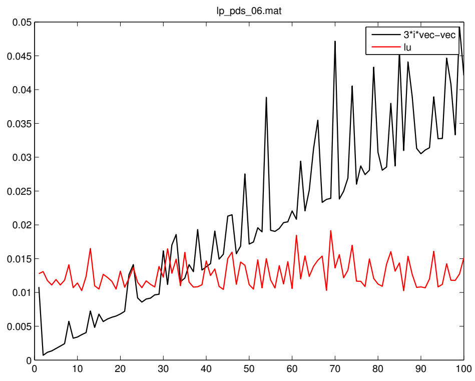

Figure 2.5: lp_pds: $m = 9881$

The above script simply compares computing $3 \times i$ vector-vector products versus calling MATLAB's `lu` as the size of the matrix and the number of iterations increase. The script above generates the plot in figure 2.4 where the black line represents the time taken to perform vector-vector products. If we look closely at the plots, we can notice that except for the fourth dataset, computing the $\boldsymbol{LU}$ factors using `lu` is a better idea. Even for 80_bau, the plot suggests that refactoring every almost 10 iterations making a brand new $\boldsymbol{LU}$ will

produce the best result. Since the time difference was less than a hundredth of a second between the two methods, the tradeoff of wasn't very significant that choosing to perform a new ***LU*** updates suggests itself as a better idea. In fact, all of this is due to the virtue of `lu`, the function by itself is extremely powerful. Note that this is not the case if we were to use our own ***LU*** factorization function or if we were using a different programming language.
Before we end this section, it falls as a natural choice to see what happens as the size of the matrix gets even bigger. Figure 2.5 shows that as the matrix size grows using MATLAB's `lu` is not a very good idea. Figure 2.5 also suggests that refactoring at the 30th iteration will give best performance. Therefore, as the size of the matrix gets bigger, a better option would be to perfom ***LU*** rank-k updates.

2.4 Entering Index

Aside from the selection of the entering indices and solving the 3 systems, there is yet another important component of the algorithm that can have drastic changes in the performance. What we dicsuss in this section is the *pricing step*. During this step, we decide on the entering index to enter the basis matrix $\mathcal{B}$.

2.4.1 Description Of Advantages And Tradeoff

At each iteration in the simplex method, we compute the vector $\mathbf{s}_N$, but this vector may normally contain more than one negative component. In the best case scenario, we would want to choose q in such a way that decreases the number of steps we may need to reach the optimal solution. The tradeoff between spending time finding a suitable such index and the actual time saved is of great concern. In our experiments for example, we have seen that when we check the decrease between the objective function value at the current iterate and the previous iterate has given us a less number of iterations but caused each iteration to take more time. The original simplex method invented by Dantzig suggests that the entering index be picked based on a simple rule: pick the index corresponding the smallest negative value in $\mathbf{s}_N$.

2.4.2 Method We Adopted

Even though the method suggested by Dantzig is the safest to use, it doesn't come free of risk. While the simplex method iterates, we need to guarantee that we maintain a basis matrix $\mathcal{B}$ that is non-singular. To maintain such a matrix, it is compulsory to study the newly generated basis at each iteration. If the matrix is singular, it is important to change the entering index before proceeding with the rest of the iterations. In our case, we have adopted an ***LU*** rank 1 update described in the previous section, and this ***LU*** update gives us the advantage of checking the value of just one element before proceeding. If the value of the element $u_{m,m}$ in the upper triangular matrix described in the previous section is zero, then the matrix is singular and we choose another entering index. In fact, the reason we have dropped the comparison of finding the $\boldsymbol{LU}$ factors of $\boldsymbol{G}$ in section 2.3.2 goes back to this reason.

2.5 Results And Analysis

In this section, we see how our approach performed on the datasets given to us, and then analyze our performance on other datasets. Table 2.1 presents the optimal values we obtained using our method.

Problem	MATLAB IP	MATLAB Simplex	Our Simplex
lpi_itest6	3.682e−15	0	0
lpi_chemcom	8.250e+02	8.250e+02	8.250e+02
lp_agg3	1.471e+06	1.471e+06	1.471e+06
lp_80bau3b	7.000e+02	7.000e+02	7.000e+02

Table 2.1: Simplex results

Figures 2.6 represents the time taken by our method to solve the linear program versus MATLAB's `linprog`. On the initial 4 datasets our results are very comparable to MATLAB, by portions of a second of difference on each problem. On figure 2.6 too, we can see that the time difference with MATLAB becomes almost one second in one of the datasets. Looking at figure 2.7, and trying to analyze this performance, we notice that the number of iterations and the number of singular $\mathcal{B}$ matrices encountered is the major cause of this slow down and consequently improvements could be done on choosing the entering index. More heuristics could be added to this component of the algorithm so that they could be tweaked to a more general case. We end this chapter by the results present in figures 2.6 and 2.7.

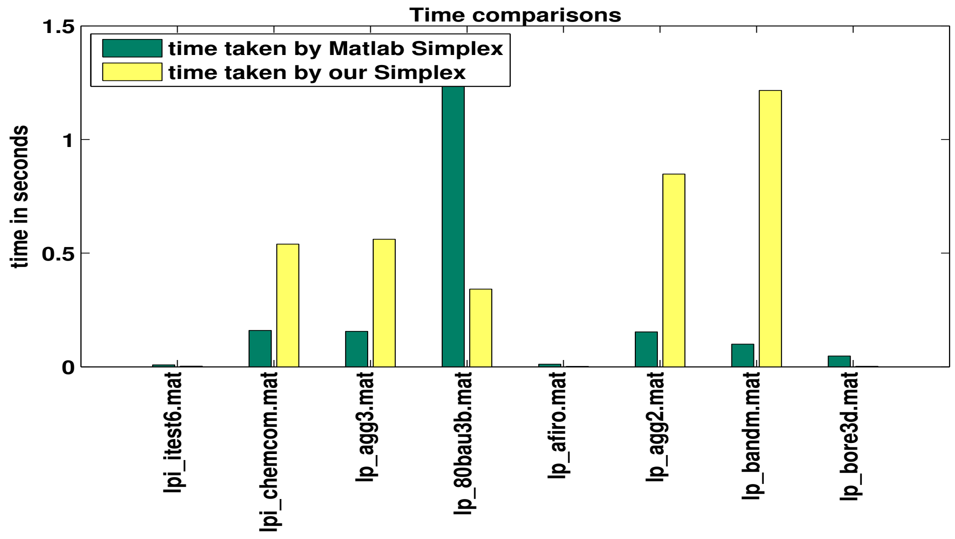

Figure 2.6: Time taken to solve each of these problems

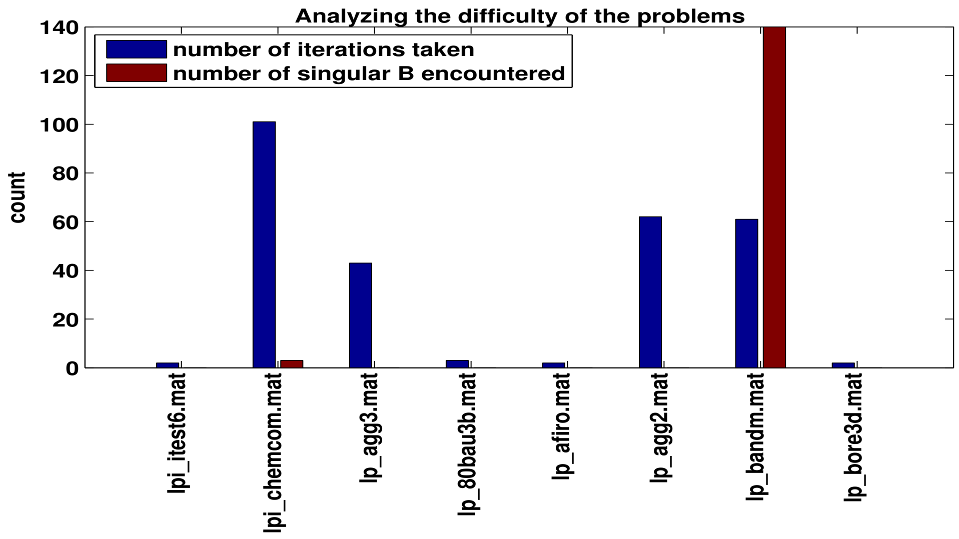

Figure 2.7: Major reasons behind the performance