Solving Small LP Problems Quickly and Reliably

Max Guy, Josep Garcia-Reyero Sais

February 2023

Abstract

Individual linear programming solve methods are usually either very fast or relatively reliable. Using a combination of these methods in an efficient manner can lead to a combined solver capable of both. In particular, presolve rules are very fast and the simplex algorithm with Bland's rule is very robust. Normally, these methods are not used in conjunction with each other because the presolve rules are usually applied to large scale optimization problems on which Bland's simplex is prohibitively expensive. However due to a relatively new avenue of exploration in compiler optimization, we need a solver which can solve a lot of small problems. Therefore, a blend of these methods is now a feasible solution to give fast and reliable results [1].

Contents

1 Introduction

In recent years, advances in compiler optimisation research have given rise to research on adapting classic linear programming techniques to compiler specific sets of linear programming problems. This project was motivated in particular by novel research [2], where a simplex solver was implemented for use in a compiler. They looked to solve a set of problems as continuous linear programming problems (LPs), with the ultimate aim of deducing the integer feasibility of the problems. However, the only way they had of determining integer feasibility was by checking if they had found an integer solution to the continuous problem.

In this paper we look to expand on the work of [2], exploring solvers which may be more efficient mathematically, with the aim of solving all the problems in a given set fast and reliably as LPs. We also make steps towards investigating the integer feasibility of the problems. The problem set is generated directly by a compiler [2].

We start by developing both a dual and a primal simplex solver, adapted and optimised to solve compiler specific problems, and with anti-cycling modifications such as Bland's rule. We then compare them and evaluate which is better adapted to solve the problems at hand. Although the simplex solver is very robust, it is not very computationally efficient. To this extent, we exploit an abundant presence of redundancy in the set of problems by implementing a presolve and postsolve procedure. The presolve and postsolve procedure can be used to solve problems very fast, but it will most likely only solve a subset of the problems.

Finally, we combine the presolve and postsolve procedure with the simplex solver in order to produce a fast and yet robust solver, optimised to solve compiler generated problems in the set, and evaluate its performance on the full problem set. Additionally, in our presolve and postsolve implementation we make headway in determining integer feasibility.

2 Background

2.1 Context

Systems of linear constraints can be used to model large scale computer programs, in particular how and where they access data from memory [2]. The feasibility of these problems can lead to optimizations being made in large scale programs, in particular switching the number of reads from a hard drive (a slow operation) to reads from RAM (a significantly faster operation). These optimizations are made at compile time on programs, necessitating a lightweight solver which can read and inform the program as to which optimizations are feasible. Given that this solver will be run at compile time, the solver will need to be both fast and reliable. The feasibility constraints problems themselves are relatively small (less than 50 variables, typically around 6-8 variables per problem) but due to the method in which they are generated, are produced in very large quantities (100K+).

2.2 The problems

Every problem in our set is a feasibility problem where a series of linear constraints are proposed without an objective function. Our problem set consists of $150,218$ problems, contained in a standard text file. Each problem has the form:

- <number of variables>
- <number of inequality constraints>
- <matrix of inequality constraints>
- <number of equality constraints>
- <matrix of equality constraints>
- a row of tildes ($\sim$) to separate problems.

For example, the first problem input is:

```
8
2
0 1 0 0 0 0 0 0 0
−1 −1 0 0 0 0 0 1 0
1
0 −1 0 0 1 0 0 0 0
~~~~~
```

Corresponding to the following system of equations:

$$\begin{aligned} x_1 &\geq 0 \\ -x_1 + x_7 &\geq 1 \\ -x_1 + x_4 &= 0 \\ x_i &\in \mathbb{Z}, \end{aligned} \tag{1}$$

where all the variables are free. In this particular problem can clearly see this example is feasible ($x_7 = 1$, all other $x_i = 0$).

2.3 Note on results

For the sake of consistency, all runtimes stated were gathered on the same laptop with consistent compiler options. The laptop itself is a standard Windows laptop, running the implementations in a Windows Subsystem for Linux terminal. While exact runtimes may vary across different devices, the times relative to each other should remain relatively consistent.

2.4 The primal simplex algorithm

2.4.1 Introduction

The original primal simplex algorithm was designed by George Dantzig to solve logistics problems for the US Air Force in 1947 [3]. It began when it was realised that such logistics problems could often be modelled as a series of linear inequalities and could be used to find the optimal value to an objective function. It is still used inside large-scale optimization software (such as HiGHS [4]) however present day research into optimization largely focuses on solving large scale problems with sparse matrices, often using the simplex method alongside an ensemble of other methods.

2.4.2 Standard form and notation

For the sake of consistency, we will use the standard form of the LP problem as given in [5] and equivalent to the form originally proposed by Dantzig [3]:

$$\begin{aligned} \text{maximize } \mathbf{c^T x} \\ \text{subject to } \mathbf{Ax} \leq \mathbf{b} \\ \text{and } \mathbf{x} \geq \mathbf{0} \end{aligned} \tag{2}$$

.

Where :

- $\mathbf{x}$ represents our n variables
- $\mathbf{c}$ represents our costs vector of length n
- $\mathbf{A}$ is an $m \times n$ matrix
- $\mathbf{b}$ is our vector of bounds (length m).

Therefore we have a clearly setup problem: find the value of $\mathbf{x}$ that maximizes the value of $\mathbf{c^T x}$.

In order to use the simplex method to solve this, we need to change the inequalities into equalities that we can work with. To do this, we add “slack” variables. This means that we add a new variable to the end of each inequality which we say accounts for the “slack” between the expression and the bound. From there, our problem in standard form becomes:

$$\begin{aligned} \text{maximize } \mathbf{\bar{c}^T x} \\ \text{subject to } \mathbf{\bar{A}x} = \mathbf{b} \\ \text{and } \mathbf{x} \geq \mathbf{0}, \end{aligned} \tag{3}$$

.

where $\bar{\mathbf{c}} = \left[\begin{smallmatrix}\mathbf{c}\\ \mathbf{0}\end{smallmatrix}\right]^T$ and $\bar{A} = [A \ \ I]$

For example the problem

$$\begin{aligned}
&\text{maximize} \\
&\qquad f = 3x_1 + 2x_2 \\
&\text{subject to} \\
&\qquad x_1 + x_2 \leq 3 \\
&\qquad 2x_1 + 3x_2 \leq 6 \\
&\qquad x_1 \geq 0 \text{ and } x_2 \geq 0
\end{aligned} \tag{4}$$

becomes

$$\begin{aligned}
&\text{maximize} \\
&\qquad f = 3x_1 + 2x_2 \\
&\text{subject to} \\
&\qquad x_1 + x_2 + x_3 = 3 \\
&\qquad 2x_1 + 3x_2 + x_4 = 6 \\
&\qquad x_i \geq 0
\end{aligned} \tag{5}$$

with x_3, x_4 being the slack variables.

2.4.3 The algorithm

To begin with, we find a basic feasible solution to the problem (essentially a solution which may not be optimal but which still satisfies the problem constraints). In practice, an "all slack" basis can be used however this only holds if the bounds vector $\underline{b} \geq 0$. For example in (5), the initial solution with an all-slack basis we extract would be $x_1 = x_2 = 0$, $x_3 = 3$, $x_4 = 6$. The condition that all x_i remain positive is therefore the reason why we must maintain that $\underline{b} > 0$; to give us an initial feasible solution.

For our description of the simplex algorithm, we will both give an algebraic explanation and explain how it relates to the corresponding tableau. The tableau method is a way in which we can express this problem in tabular form and perform row operations on it which are equivalent to iterations of the simplex algorithm. In our implementations, the tabular forms were used due to their suitability for use in code.

The algebraic expression of the primal simplex method is as such:

1. Assuming our bounds vector $\underline{b} > 0$, we initialise with an "all-slack" basis.
2. We then calculate the vector of reduced costs, $\hat{\underline{c}}_N = \underline{c}_N - N^T B^{-T} \underline{c}_B$ where:
    - $\underline{c}_N$ is the cost vector of the non-basic set (vector of costs associated with all variables not in the basis)
    - $\underline{c}_B$ is the cost vector of the basic set
    - N is the matrix of coefficients for the non-basic variables
    - B is the matrix of coefficients for the basic variables

    and stop if all $\hat{\underline{c}}_N > 0$ since we would then have the optimal solution.
3. Find $q' \in N$, the most positive reduced cost (this will be the "entering variable").
4. Let $\hat{a}_q = B^{-1} a_q$ where a_q is column q of N. If $a_q \leq 0$ then stop, the LP is unbounded and hence does not have a finite optimal solution.
5. Conduct ratio tests to find $\underset{i=1\ldots m}{argmin} \left\{ \frac{\hat{b}_i}{\hat{a}_{iq}} : \hat{a}_{iq} > 0 \right\}$. The variable associated with this index will be the "leaving variable", p'.

6. Exchange indices p' and q' between B and N to give a new basic feasible solution.
7. Return to step 2 and repeat until one of the two break conditions are met.

The primal simplex tableau takes the below form:

c_N	
N I	b

And has the equivalent following steps

1. We already have positive bounds and implemented an all-slack basis when constructing the tableau
2. We find the most positive value in c_N. The column this value is in will be our “pivot column”, $\underline{p}$.
3. If all values in $c_N > 0$, stop. We have reached an optimal solution.
4. If all values in the pivot column are greater than 0, stop. The LP is unbounded and hence does not have a finite optimal solution.
5. We conduct a ratio test where we divide each element in the bounds vector b_i with the corresponding element in the pivot column, p_i when $p_i > 0$. The row with the smallest ratio will be the “pivot row”.
6. We now take the ‘pivot value” to be the value occurring at the intersection of the pivot row and the pivot column. We now “zero out” the pivot column by performing row operations to reduce every other value in the column to zero with the exception of the pivot row which will be divided by the pivot value to give an entry of one in the pivot column.
7. Return to step 2 and repeat until one of the two break conditions are met.

Note that in practice we don’t use c_N or N, we use the whole objective function f and the full constraints matrix A while keeping track of the basis separately, ensuring pivot columns are in the basis before selecting them. Also due to the way in which the row operations were constructed, it was simpler to use the negative of the objective function however all operations that need to be inverted due to this have been.

2.4.4 Example

Let’s take the following optimization problem:

$$\begin{aligned} \text{maximize:}& \\ & f = 2x_1 - 5x_2 \\ \text{subject to:}& \\ & x_1 + 2x_2 \leq 20 \\ & x_2 \leq 1 \\ & x_i \geq 0. \end{aligned} \tag{6}$$

The first operation we must perform is adding slack variables to the problem. This gives us:

$$\begin{aligned}
&\text{maximize:}\\
&\qquad f = 2x_1 - 5x_2\\
&\text{subject to:}\\
&\qquad x_1 + 2x_2 + x_3 = 20\\
&\qquad x_2 + x_4 = 1\\
&\qquad x_i \geq 0.
\end{aligned} \tag{7}$$

We will first solve this with the algebraic simplex method and then with the tabular method. Following the algebraic steps, we get:

1. We initialise our basis as all slack variables (i.e. the initial basis is $\{x_3, x_4\}$)
2. $\hat{\underline{c}}_N = \underline{c}_N - N^T B^{-T} \underline{c}_B = \underline{c}_N = [2, -5]^T$ since $\underline{c}_B = \underline{0}$. Note that $\hat{\underline{c}}_N$ is not all negative yet so we are not yet at an optimal solution.
3. We now get that $q' = 1$ since the first element of our reduced costs vector $\hat{\underline{c}}_N$ is the most positive.
4. Since $B = I$, $\hat{a}_q = a_q = [1, 0]^T$. Since there is at least one strictly positive element in a_q, the problem has not yet been shown to be unbounded and so we continue.
5. We now conduct ratio tests to find $\underset{i=1\ldots m}{argmin} \left\{\frac{20}{1}\right\}$. We do not have any more ratios because the second element of $\hat{a}_q = 0$ and we do not compute any ratios where $\hat{a}_{iq} \leq 0$. Therefore we get that $p' = 3$.
6. Now we can replace p' in the basis with q', giving us a new basis $b = \{x_1, x_4\}$
7. We now return to step 2 and continue with the next iteration of the algorithm

2. $\hat{\underline{c}}_N = \begin{bmatrix} 0 \\ 5 \end{bmatrix} - \begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 0 \end{bmatrix} = \begin{bmatrix} -2 \\ -9 \end{bmatrix}$. We now have that $\hat{\underline{c}}_N < 0$ so we are at an optimal solution. Our basis is currently $B = \{1, 4\}$ and $\hat{\underline{b}} = [20, 1]^T$, giving us an optimal solution of $x_1 = 20$, $x_2 = x_3 = 0$, $x_4 = 1$.

Now we solve the equivalent problem using the tableau method. The first step is to format as a tableau and set the initial basis:

	2	-5	0	0	
x_3	1	2	1	0	20
x_4	0	1	0	1	1

Now we can select our pivot column as the most positive cost in our objective row:

	2	-5	0	0	
x_3	1	2	1	0	20
x_4	0	1	0	1	1

and conduct the same ratio tests that we did in the algebraic method, giving us the pivot row (and hence the pivot element):

	2	-5	0	0	
x_3	1	2	1	0	20
x_4	0	1	0	1	1

From here we complete the row operations to remove all nonzero values from the pivot column. We also switch the exiting variable (x_3) for the entering variable (x_1) in the basis. This gives us our new table and commences the second iteration of the algorithm:

	0	-9	-2	0	
x_1	1	2	1	0	20
x_4	0	1	0	1	1

From here we can see that the objective row contains only negative values so the algorithm will terminate. We can also see that we have the same basis, costs and bounds as the algebraic method at this point, giving us the same solution of $x_1 = 20$, $x_2 = x_3 = 0$, $x_4 = 1$.

While both methods are equivalent and will yield the same solution, the repetitive and formulaic nature of the tableau method lends itself to computational applications more easily since all that is required to keep track of is the basis and the table itself.

3 Adapted primal method

The adapted primal method is the first of the reliable methods trialled on this problem set. It involves performing a series of operations on the input problem to convert it to a problem which can be solved using primal simplex methods.

3.1 Modification for solving dual problems

Unfortunately, the problems we have do not necessarily meet the requirements that $\underline{b} > 0$ and so the standard all-slack basis does not necessarily produce a basic feasible solution. While it is possible to make a solver which works for general variable and constraint bounds (i.e. no longer requiring that $x_i > 0$ and $\underline{b} > 0$), making such modifications unnecessary is a level of complexity beyond the scope of this project. Therefore we must reformat the input problems to convert them into a form which the standard primal simplex algorithm will be able to solve.

3.1.1 Making variables positive

To assert that all variables in our problem are strictly positive, we must make a modification to the original variables. To do this, we construct two new variables for every variable in our problem such that $x_i = x_i^+ - x_i^-$ with $x_i^+ \geq 0$ and $x_i^- \geq 0$. Doing this, our standard form (2) is changed to allow for negative variables (8)[5].

$$\begin{aligned} &\text{maximize} \\ f = &\begin{bmatrix} \mathbf{c} \\ -\mathbf{c} \end{bmatrix}^T \begin{bmatrix} \mathbf{x}^+ \\ \mathbf{x}^- \end{bmatrix} \\ \text{Subject to } [\mathbf{A}| - \mathbf{A}] &\begin{bmatrix} \mathbf{x}^+ \\ \mathbf{x}^- \end{bmatrix} \leq \mathbf{b} \\ &\text{and } \mathbf{x} \geq \mathbf{0} \end{aligned} \tag{8}$$

.

3.1.2 Equality rows

Next, since we want to start with an all slack basis, we have to make sure all given constraints are inequalities with the expression being less than the bound. Unfortunately, some of our input rows are equality rows so we have to change them to inequality rows while maintaining the equality in our constraints.

To do this, we turn the equality into a row of the desired form and then add a row below it with the negative of the expression in place (but without reversing the inequality). For example if we had the constraint $2x_1 + 3x_2 = 5$, we would create the rows $2x_1 + 3x_2 \leq 5$ and $-2x_1 + -3x_2 \leq -5$. Therefore, we still implicitly have the equality but also have our constraints in the desired form.

3.1.3 Taking the dual

Since the purpose of this solver is to find a feasible solution to the primal problem, we need to start off with a basic feasible solution. In the dual problem, the all slack basis is initially feasible, allowing us to work towards an optimal solution from there. If the dual problem terminates, we are able to work back to a primal solution.

The dual problem is formed by altering the primal problem. The dual has some useful relationships to the primal problem. In particular, we have the strong duality property which states that if we have an optimal solution for the dual of the problem, then there is a complimentary optimal solution for the primal problem (and hence it must be feasible). There is a method for extracting this solution however we will discuss that method in more detail in 3.1.5.

When converting a primal problem into its dual:

- The coefficients in the objective function in the primal problem become the bounds vector of the dual problem.
- The bounds vector of the primal problem become the coefficients in the objective function in the dual problem.
- The matrix of bound coefficients stays the same but instead of being multiplied by a column vector of primal variables on the right, it is multiplied by a row vector of dual variables on the left.
- The direction of the inequality changes and the problem becomes a minimization instead of a maximization.

Taking the dual of a problem is better explained using an example. Take the primal problem:

$$\begin{aligned}
&\text{Maximize} \\
&\qquad f = 2x_1 + 7x_2 \\
&\text{Subject to} \\
&\qquad x_1 \phantom{{}+8x_2} \leq 3 \\
&\qquad \phantom{6x_1+{}} 4x_2 \leq 9 \\
&\qquad 6x_1 + 8x_2 \leq 5 \\
&\text{and } x_i \geq 0
\end{aligned} \tag{9}$$

if we take the dual of this, the problem becomes:

$$\begin{aligned}
&\text{Minimize} \\
&\qquad g = 3y_1 + 9y_2 + 5y_3 \\
&\text{Subject to} \\
&\qquad y_1 \phantom{{}+4y_2} + 6y_3 \geq 2 \\
&\qquad \phantom{y_1+{}} 4y_2 + 8y_3 \geq 7 \\
&\text{and } y_i \geq 0
\end{aligned} \tag{10}$$

However we are not quite finished yet because the simplex algorithm is a maximization algorithm so we also need to ensure that we are dealing with such a problem before we begin. Fortunately, the two below forms are equivalent [5] :

$$\begin{aligned}
&\text{Minimize} \\
&\qquad f \\
&\text{Subject to} \\
&\qquad \sum_{j=1}^{n} a_{ij}x_j \geq b_i \\
&\qquad \text{and } x_i \geq 0
\end{aligned} \tag{11}$$

$$\begin{aligned}
&\text{Maximize} \\
&\qquad -f \\
&\text{Subject to} \\
&\qquad -\sum_{j=1}^{n} a_{ij}x_j \leq -b_i \\
&\qquad \text{and } x_i \geq 0
\end{aligned} \tag{12}$$

And so we can take the dual of a problem and negate it to give us the dual maximization problem. This now gives us a problem which we can solve and whose solution we can relate back to the initial feasibility problem.

3.1.4 Putting the modifications together

In order to put all these together, we will introduce a distinction between the inequality input rows and the equality input rows. For this, any constraint that is related to an equality will have a subscript "E" and inequality rows a subscript "I". So the original problem becomes

$$\begin{aligned} &\text{maximize} \\ &\qquad \mathbf{c^T x} \\ &\text{Subject to} \\ &\qquad \mathbf{A_I x} \le \mathbf{b_I} \\ &\qquad \mathbf{A_E x} = \mathbf{b_E} \\ &\text{and } \mathbf{x} \ge \mathbf{0}. \end{aligned} \tag{13}$$

Now, we deal with the equality rows and ensure all variables are ≥ 0:

$$\begin{gathered} \text{maximize} \\ f = \begin{bmatrix} \mathbf{c} \\ -\mathbf{c} \end{bmatrix}^T \begin{bmatrix} \mathbf{x}^+ \\ \mathbf{x}^- \end{bmatrix} \\ \text{Subject to} \\ \begin{bmatrix} A_I & -A_I \\ A_E & -A_E \\ -A_E & A_E \end{bmatrix} \begin{bmatrix} \mathbf{x}^+ \\ \mathbf{x}^- \end{bmatrix} \le \begin{bmatrix} b_I \\ b_E \\ -b_E \end{bmatrix} \\ \text{and } \mathbf{x}^+, \mathbf{x}^- \ge 0. \end{gathered} \tag{14}$$

Finally, we take the dual of the problem:

$$\begin{gathered} \text{maximize} \\ f = \begin{bmatrix} b_I \\ b_E \\ -b_E \end{bmatrix}^T \begin{bmatrix} y_I \\ y_E^+ \\ y_E^- \end{bmatrix} \\ \text{Subject to} \\ \begin{bmatrix} A_I^T & A_E & -A_E^T \\ -A_I^T & -A_E & A_E^T \end{bmatrix} \begin{bmatrix} y_T \\ y_E^+ \\ y_E^- \end{bmatrix} \le \begin{bmatrix} 0 \\ 0 \end{bmatrix} \\ \text{and } \mathbf{x}^+, \mathbf{x}^- \ge 0. \end{gathered} \tag{15}$$

.

Finally, we add slack onto this formulation to give us an initial basic feasible solution. While this is useful and allows us to coerce the problem into a solveable format, it has significant drawbacks. The predominant issue is the increased size of these new, solveable problems. For example, the first problem in our set (1) can be easily solved by manual observation but once the problem has been reformatted, its becomes a 17×22 problem matrix, drastically increasing the time it would take a person to solve. While this is still relatively small in computer terms and does not run the risk of causing hardware related issues (such as making problems too large for standard data structures to handle), it is still important to note that this approach leads to producing a significantly more complex problem.

3.1.5 Extracting the results

The solved values for $\mathbf{x}_+$ and $\mathbf{x}_-$ correspond to the values of the slack variables in the costs row of our tableau [5]. Therefore, to extract our solutions, we simply take $\mathbf{x} = \mathbf{x}_+$ - $\mathbf{x}_-$ and verify the value of $\mathbf{x}$ against the constraints.

3.2 Optimizations in implementation

Fortunately while implementing a tableau simplex, there are certain optimizations that can be made to the program to allow for more efficient problem solving. This essentially boils down to removing unnecessary operations wherever possible and ensuring that calculations are not being repeated.

3.2.1 Max iterations

Although rare, problems which "cycle" are possible [6] (and when you have $\approx$ 150k problems, significantly more likely). These are problems which return to their original state after a set number of simplex algorithm iterations and hence cannot be solved by the simplex algorithm (unless algorithmic prevention methods such as Bland's rule have been used).

Therefore, a simple check to ensure that we have not computed an inconceivably large number of simplex iterations makes sense and allows the program to keep moving in the case of such a problem. In our simplex implementation, an error is logged and the solver continues on to the next problem in this case.

3.2.2 Checking necessity of row operations

When performing the row operations on the tableau in order to update it, it is important to check that the row you are about to perform row operation on does not have a 0 entry in the pivot column. In this case, you would be multiplying the pivot row by 0 (hence making all entries 0) and subtracting this from the row with the original 0 entry. While this does not cause any issues with the algorithm, it also does not change anything when updating the tableau hence making it a pointless set of operations.

If we take the computational cost of a single arithmetic operation to be equal to the computational cost of a comparison of pre-computed values (unit cost), we can compare the cost of the two approaches on a row with n elements:

- If we don't implement this check, the computational cost is $3n$ (n multiplications on each row, n subtractions).
- If we do implement this check, the cost is conditional on if the row has a zero entry in the pivot column. If it does then the cost is 1 (cost of a single comparison. If the value is 0 then no further action needs to be taken). If the pivot column does not equal zero, the cost becomes $3n + 1$ (the $3n$ from the row operations and the unit cost of a comparison).

Therefore, we can see that the efficacy of this check is dependant on the number of zeros present in your problem set.

In our problem set, the average number of constraints in a row is approximately 6.5 and approximately 80% of all coefficients in our problem set are 0. Therefore, comparing the expected computational costs of these approaches:

- Without the check, we expect to make an average of 17.5 operations for every row operation.
- With the check, we expect to make an average of $0.8(1) + 0.2(3 * 6.5) = 4.7$ operations.

Therefore, this single check reduces the average number of row operations by more than 70% on our problem set.

3.3 Results

This solver runs in approximately 2.65 seconds on average with the following results:

- $29,473$ empty problems
- $120,474$ feasible problems
- 25 infeasible problems
- 28 problems where a program error occurred

This suggests that the modified primal simplex method is quite a robust method however it is slow in comparison to presolve methods (6.8).

4 Dual method

The dual simplex algorithm is essentially the primal simplex algorithm applied to the dual problem. This can be achieved by either taking the dual and running the primal algorithm on it explicitly or running the dual algorithm on the primal problem. As previously seen, the dual problem is closely related to the primal problem and so performing the dual simplex algorithm can either solve our primal problem or at least show us that it is infeasible (since the dual problem being unbounded implies the primal problem is infeasible[5]).

4.1 The dual simplex algorithm

Instead of the primal simplex being applied to the dual algorithm as done previously, we now discuss the direct dual algorithm. We will discuss this in the tableau form since this is how we format the problem for our solver.

Before using the direct dual method, there is still a small amount of reformatting to be done to the problem. In particular, we still need to modify the problem to allow all variables to be positive as seen in (8). Also, since we have "$\geq$" inequalities in the primal but making pivots in the dual space, we need to add negative slack variables instead of positive ones.

Since we are working in the dual space, our primal bounds vector becomes our dual costs vector and vise versa. Therefore the dual tableau is:

$$\begin{array}{|cc|c|}\hline \multicolumn{2}{|c|}{b} & \\ \hline N & -I & c_N \\ \hline\end{array}$$

Where everything in the above table is in terms of dual properties (i.e. b is the dual bounds, not primal bounds).

For the dual tableau simplex algorithm, the method is very similar to the primal tableau simplex algorithm with a small number of changes:

- Instead of first finding the pivot column, we find the pivot row
- Instead of finding the pivot row by conducting ratio tests along the pivot column, we now find the pivot column by conducting ratio tests along the pivot row.
- Optimality is now deduced by the lack of negative values in the primal bounds column, not the primal costs vector.

4.2 Example

Take the system

$$\begin{aligned} -x_1 + x_2 + x_3 &\leq 0 \\ -x_1 - 2x_3 &\leq -1 \\ x_2 &\leq 0 \\ x_i &\geq 0. \end{aligned} \tag{16}$$

We first have to add slack variables to our problem (these slack variables are positive since the constraints are "$\leq$"):

$$\begin{aligned} -x_1 + x_2 + x_3 + x_4 &= 0 \\ -x_1 - 2x_3 + x_5 &= -1 \\ x_2 + x_6 &= 0 \\ x_i &\geq 0. \end{aligned} \tag{17}$$

Now we solve using our dual tableau (for this, we will also use Bland's rule (5)):

	0	0	0	0	0	0	
x_4	-1	1	1	1	0	0	0
x_5	-1	0	-2	0	1	0	-1
x_6	0	1	0	0	0	1	0

Now instead of choosing the first negative value in the objective function to find the pivot column as we would do in primal tableau, we choose the first negative value in the bounds column to give us our pivot row:

	0	0	0	0	0	0	
x_4	-1	1	1	1	0	0	0
x_5	-1	0	-2	0	1	0	-1
x_6	0	1	0	0	0	1	0

And then choose the first negative nonbasic value along this row to decide our pivot row and hence our pivot value:

	0	0	0	0	0	0	
x_4	-1	1	1	1	0	0	0
x_5	-1	0	-2	0	1	0	-1
x_6	0	1	0	0	0	1	0

Finally, now that we have our pivot value of -1, we can complete our row operations (and switch our basic variables):

	0	0	0	0	0	0	
x_4	0	1	3	1	1	0	1
x_1	1	0	2	0	-1	0	1
x_6	0	1	0	0	0	1	0

Therefore, we can see that the primal solution we have is $x_1 = x_4 = 1$ and all other $x_i = 0$. Checking this against our initial problem, we see that our solution for $\underline{x}$ satisfies the constraints.

4.3 Results

This solver runs in approximately 2.8 seconds on average with the following results:

- $29,537$ empty problems
- $110,314$ feasible problems
- 64 infeasible problems
- $10,303$ problems where a program error occurred

The errors occur when an optimal solution is found but it violates one or more of the problem constraints. This could be due to floating point errors (so an increase in tolerance may be required) or with the verification process. While this solver still finds a solution for almost 95% of problems, it is both less robust and marginally slower than the primal solver in our implementation so we will not include it in the combined solver. Note that this dual implementation has all of the same optimizations as the primal solver.

5 Bland's rule for termination

5.1 Description of method

Previously, the method we have used for deciding the entering and exiting variable of the basis in the simplex method is based on finding the "best" swap. However, as can be seen in the Hall-McKinnon problems, [7] and elsewhere in literature [6] the simplex algorithm does not prevent cycling. Therefore we introduce Bland's anti-cycling rule.

Bland's rule is a lexicographic method for deciding on the entering and leaving variables in the simplex algorithm. Instead of deciding the pivot row by choosing the most negative reduced cost, we now simply select the first negative nonbasic reduced cost in our objective function. In our selected pivot column, the pivot row (and hence leaving variable) is selected as the first strictly positive element in the column. By using these rules, the simplex algorithm will not cycle however since it is only choosing the first available pivot instead of the best available pivot, it may require more simplex iterations to converge [8].

5.2 Termination

In 1977, Robert Bland produced a full proof of the termination of the primal simplex algorithm when using Bland's rule [8]. However we now make the observation of this rule holding for the dual simplex as well.

We know that Bland's rule terminates on our first problem since we apply the primal simplex algorithm to the dual problem (and we know that Bland's rule guarantees termination in the primal algorithm). Since the dual of the dual problem is the primal, it holds that Bland's rule also applies to the dual algorithm being applied to the primal problem since the problems are equivalent.

6 Presolve and Postsolve

6.1 Introduction

Upon manual inspection of the problems at hand, it was noted that many of them had redundant information in their constraints and variables. For instance, we found linearly dependant constraints and empty columns, amongst others. This redundancy can be exploited, when it comes to solving the problems, by implementing a presolve and postsolve procedure.

Presolve is a way of systematically removing redundancy from linear programming problems [9] by iteratively applying a set of rules. A problem may be reduced to empty after removing all redundancy by applying presolve, that is, all variables and constraints are removed from the problem. After applying presolve, postsolve can then be used to attempt to find feasible values for variables that have been removed from the problem on presolve, starting from the last variable that was removed and working back from there. If the problem had reduced to empty, postsolve will find feasible values for all variables in the problem. Note that even if presolve does not reduce a problem to empty, removing redundancy can still make the problem smaller and faster to solve with a simplex method.

In this section we implement a presolve and postsolve procedure for LPs and apply it to the full problem set. Presolve and postsolve can be very computationally efficient [4]. Our aim is therefore to reduce to empty and work back a feasible solution for as many problems as possible. All such problems can then be deemed LP feasible, and thus the primal simplex method described in the previous sections need not be applied, which might improve the efficiency of the combined solver. Problems that are not reduced to empty might still be reduced in size, meaning that the primal simplex method could be applied to the smaller, reduced problem, further decreasing computational time (however this simplex solver would have to work with general bounds). Additionally, presolve might already identify infeasible constraints.

Finally, we will alter the implementation by treating the problems as integer programming problems, or IPs, by adding an integrality restriction to the feasible solutions. We will then compare how our results change, which is relevant to the ultimate goal of the original research in [2] of determining integer feasibility.

6.1.1 Presolve and postsolve worked example

Taking an example problem, which is an altered sub-problem extracted from a full problem in the test problem set, we can informally apply presolve and postsolve manually to the problem to illustrate the procedure. The example problem is given by

$$\begin{aligned} x_0 &\geq 0 \\ -x_0 + x_1 &\geq 1 \\ -x_1 + x_3 &\geq 1 \\ x_1 + x_2 &= 1 \\ x_4 &= 2, \end{aligned} \tag{18}$$

where we have 5 constraints and 5 variables and, as with all the test problems, we have that the objective function is empty and all the variables are free. We start by applying presolve:

- Going through the problem the first time we notice that x_3 only appears in the third constraint. Therefore, we can remove x_3 and the constraint from the problem, although we need to keep track of the dependency $x_3 \geq 1 + x_1$. Similarly, x_2 only appears in the fourth constraint, so we remove x_2 and the constraint from the problem and keep track of the dependency $x_2 \geq 1 - x_1$. Finally, x_4 is the only variable in the fifth constraint, which is an equality, and it does not come up in any other constraints. We thus already have a feasible value for x_4, $x_4 = 2$, and we remove both x_4 and the fifth constraint from the problem. We are left with:

$$\begin{aligned} x_0 &\geq 0 \\ -x_0 + x_1 &\geq 1. \end{aligned} \tag{19}$$

- As with x_3 and x_2, x_1 now only appears in the second constraint. We thus remove x_1 and the constraint, keeping track of $x_1 \geq 1 + x_0$, and we are left with

$$x_0 \geq 0. \tag{20}$$

- We now only have the first constraint left, where the only variable is x_0, so we remove both x_0 and the constraint from the problem, keeping track of $x_0 \geq 0$, and the problem has reduced to empty.

We now apply postsolve, reversing the presolve procedure and working out feasible values for all the variables.

- We start from the last variable that was removed, which was $x_0 \geq 0$. Therefore, we set $x_0 = 0$.
- We now go to the next variable that was removed, $x_1 \geq 1 + x_0$. However, we now have $x_0 = 0$, thus we have that $x_1 \geq 1$, and we can set $x_1 = 1$.
- Similarly, x_3 and x_2 only depended on x_1. Therefore, we have $x_3 \geq 1 + x_1 = 2$ so we set $x_3 = 2$; and $x_2 = 1 - x_1 = 0$, so we have $x_2 = 0$.
- Finally, we have $x_4 = 2$, thus we get the feasible solution

$$\begin{aligned} x_0 &= 0 \\ x_1 &= 1 \\ x_2 &= 0 \\ x_3 &= 2 \\ x_4 &= 2. \end{aligned} \tag{21}$$

We can manually check that all the constraints in (18) are met.

6.2 Presolve and postsolve problem formulation

Our presolve and postsolve implementation is specific to the the problems in the test problem set as described in Section 1.2. Therefore, we are dealing with problems without an objective function, with m constraints and n variables, where all the variables are free. Additionally, we have both equality and inequality constraints. We will let k denote the number of inequality constraints and h denote the number of equality constraints, where $k + h = m$, $k, h \in \mathbb{Z}^+$. In our presolve and postsolve implementation, we will then formulate the problems as

$$\begin{aligned} \mathbf{b_L} \leq \mathbf{Ax} &\leq \mathbf{b_U}, \\ \mathbf{l} \leq \mathbf{x} &\leq \mathbf{u}, \end{aligned} \tag{22}$$

where the coefficient matrix $\mathbf{A} \in \mathbb{Z}^{m \times n}$; $\mathbf{l}, \mathbf{x}, \mathbf{u} \in \mathbb{R}^n$; and the last h rows of $\mathbf{A}$ represent equalities, that is, $\mathbf{b_L}_i = \mathbf{b_U}_i \ \forall \ i \geq h$, so the lower and upper bounds of the constraints are equal. Note that for inequalities we always start off with $\mathbf{b_U}_i = \infty$, $i \leq k$.

Moreover, even though $\mathbf{l}$ and $\mathbf{u}$ are the starting bounds on the free variables, so we have $\mathbf{l}_i = -\infty$, $\mathbf{u}_i = \infty \ \forall \ i$ to start with, we will refer to them as implied bounds, since when applying presolve rules we might end up with implied bounds on variables which will alter $\mathbf{l}$ and $\mathbf{u}$.

6.3 Presolve

As previously mentioned, and as illustrated above, in presolve we iteratively apply a set of rules to the problem with the aim of removing redundancy. Generally, presolve rules can be categorised into row rules and column rules, where row rules will remove constraints from the problem and column rules remove variables (columns) from the problem, although some rules remove both a constraint and a variable.

In each iteration of presolve we go through our complete set of row and column rules, and we apply the ones that are relevant given the current state of the problem. If we have been able to apply one or more rules, the state of the problem will have changed as it is now reduced in size from having removed either constraints, variables or both. Then, in the next iteration we repeat the process on the reduced problem. Since the problem is now different, it may be the case the rules that were not relevant in the previous iteration have now become relevant.

We continue the iterative process until the state of the problem does not change from iteration to iteration, meaning that we have removed from the problem all redundancy that was detectable by the given set of rules that we are using. It may be the case that the problem has reduced to empty, implying that all constraints and variables were redundant.

6.4 Presolve rules

We now formally define a set of presolve rules that are commonly used in LP presolve. We will focus on primal rules, since in this project we will only implement primal presolve, however in future extensions of the project dual rules could be implemented. In later sections we will carry out a further investigation in order to determine the exact set of primal rules that we will use in our implementation.

The fact that we have both inequality and equality constraints means that in some row rules we will need to carry out a different procedure depending on the constraint type. We will use i to denote row index and j to denote column index.

6.4.1 Primal row rules

We start by considering row rules that are applied to the primal problem [9].

1. **Free row**: This is an inequality specific rule which we apply when a constraint has free bounds, that is
$$\exists\, i \leq h : \quad \mathbf{b_L}_i = -\infty,\ \mathbf{b_U}_i = \infty. \tag{23}$$
Such constraint will always be met, so we can remove it from the problem.

2. **Singleton row**: We apply singleton row when we have only one nonzero variable, that is
$$\exists\, i, k : \quad a_{ik} \neq 0,\ a_{ij} = 0\ \forall j \neq k. \tag{24}$$
In singleton row we carry out a different procedure for inequalities and equalities.

   - *Singleton row equality*: If row i is an equality then we remove the row from the problem since we have that the feasible value for x_k is given by $x_k = \mathbf{b_L}_i/a_{ik} = \mathbf{b_U}_i/a_{ik}$. This now changes the implied bounds on x_k as it leads to them being equal and given by $\mathbf{l}_k = \mathbf{b_L}_i/a_{ik} = \mathbf{u}_k$.

     The fact that the implied bounds are now equal will lead to column k being turned off when we apply the column rules.

   - *Singleton row inequality*: If row i is an inequality we only remove the row from the problem if a_{ik} is the only nonzero coefficient in column k. In this case we also remove column k from the problem since we will now be able to determine a feasible solution for x_k in postsolve, as we will explain in the postsolve section.

If the condition above is not met, we get a single implied bound on x_k. If $a_{ik} \geq 0$ the implied bound is given by $x_k \geq \mathbf{b_L}_i/a_{ik}$, so we have $\mathbf{l}_k = \mathbf{b_L}_i/a_{ik}$. On the other hand, if $a_{ik} < 0$, we get an implied upper bound $x_k \leq \mathbf{b_L}_i/a_{ik}$, and hence $\mathbf{u}_k = \mathbf{b_L}_i/a_{ik}$.

3. **Parallel rows**: We apply parallel rows when two rows are linearly dependent, that is

$$\exists\, i_1, i_2 : \quad a_{i_1 j} = r a_{i_2 j}\ \forall j,\ r \in \mathbb{R}. \tag{25}$$

Since one of the rows is redundant, we remove from the problem the row with the largest coefficients in absolute value, that is, row i_2.

However, in parallel rows we need to ensure that the system of parallel rows is feasible. The process for checking feasibility of parallel rows is different for equalities and inequalities.

- *Feasibility check equalities*: Checking whether two parallel equalities are feasible is rather straight forward, since they will only be feasible if the bounds are also parallel by the same ratio, so

$$\mathbf{b_L}_{i_1} = r\mathbf{b_L}_{i_2} \iff \mathbf{b_U}_{i_1} = r\mathbf{b_U}_{i_2}. \tag{26}$$

- *Feasibility check inequalities*: If we have parallel inequalities and we have that $\mathbf{b_U}_{i_1} = \mathbf{b_U}_{i_2} = \infty$, if $r > 0$, dividing row i_2 by r we get:

$$\begin{aligned} \sum_{j=1}^{n} a_{i_1 j} &= \sum_{j=1}^{n} a_{i_2 j}, \\ \sum_{j=1}^{n} a_{i_1 j} &\geq \mathbf{b_L}_{i_1}, \\ \sum_{j=1}^{n} a_{i_2 j} &\geq \frac{\mathbf{b_L}_{i_2}}{r}. \end{aligned} \tag{27}$$

System (27) is feasible, however when we remove row i_2 from the problem we need to update the lower bound on row i_1 by

$$\mathbf{b_L}_{i_1} = \min\left\{\mathbf{b_L}_{i_1}, \frac{\mathbf{b_L}_{i_2}}{r}\right\}, \tag{28}$$

which ensures that both inequalities are fully satisfied. However, if $r < 0$, then dividing row i_2 by r gives:

$$\begin{aligned} \sum_{j=1}^{n} a_{i_1 j} &= \sum_{j=1}^{n} a_{i_2 j}, \\ \sum_{j=1}^{n} a_{i_1 j} &\geq \mathbf{b_L}_{i_1}, \\ \sum_{j=1}^{n} a_{i_2 j} &\leq \frac{\mathbf{b_L}_{i_2}}{r}. \end{aligned} \tag{29}$$

Therefore, if $\mathbf{b_L}_{i_1} > \mathbf{b_{L i_2}}/r$, then (29) is never satisfied and the parallel rows are infeasible. Thus, in order to ensure feasibility if $r < 0$ we need

$$\mathbf{b_L}_{i_1} \leq \frac{\mathbf{b_{L i_2}}}{r}, \tag{30}$$

and if they are feasible, after removing row i_2 we update the upper bound of row i_1 to $\mathbf{b_U}_{i_1} = \mathbf{b_{L i_2}}/r$.

4. **Redundant row**: We apply redundant row whenever we have a row where all the coefficients are 0, that is
$$\exists\, i : \quad a_{ij} = 0 \;\forall j. \tag{31}$$
Row i can just be removed from the problem.

5. **Dependent equations**: In dependent equations we remove all linear dependency between rows. This rows removes the same redundancy that parallel rows would remove after eventually detecting all systems of parallel rows, but it does so in a single application of the rule.

6.4.2 Primal column rules

We now consider column rules that are applied to the primal problem [9].

1. **Empty column**: This is the simplest column rule, where we have a column of zeros, so
$$\exists\, j : \quad a_{ij} = 0 \;\forall i. \tag{32}$$
Column j can just be removed from the problem.

2. **Fixed column**: We apply fixed column when we have a variable for which the implied lower bound is equal to the implied upper bound, that is
$$\exists\, j : \mathbf{l}_j = \mathbf{u}_j. \tag{33}$$
When j is a fixed column, we immediately have a feasible value for x_j, $x_j = \mathbf{l}_j = \mathbf{u}_j$, and can remove the column from the problem. However, when we remove the column we need to update the problem by subtracting the j^{th} column of $\mathbf{A}$ times x_j from the constraints bounds $\mathbf{b_L}$ and $\mathbf{b_U}$. Fixed column is always, but not only, applied after a singleton row equality is applied.

3. **Free column substitution**: We apply free column substitution when we have a free column where only one coefficient is nonzero, that is
$$\exists\, k, j : \quad \mathbf{l}_j = -\infty, \mathbf{u}_j = \infty \quad \text{and} \quad a_{kj} \neq 0,\; a_{ij} = 0 \;\forall i \neq k. \tag{34}$$
Since the variable x_j is free, if we assume that k is an equality, we can write x_j in terms of the other variables in row k as
$$x_j = \frac{\mathbf{b_L}_k - \displaystyle\sum_{z=1, z\neq j}^{n} a_{kz} x_z}{a_{kj}}. \tag{35}$$
Then we remove both row k and column j from the problem. This is a very efficient rule because we get rid of both a row and a column without modifying the problem.

   If k is an inequality we apply the rule in the same way, but now we get
$$x_j \geq \frac{\mathbf{b_L}_k - \displaystyle\sum_{z=1, z\neq j}^{n} a_{kz} x_z}{a_{kj}}. \tag{36}$$
We still remove both row k and column j from the problem, since x_j was free before applying the rule, the inequality in (42) will always be satisfied.

4. **Doubleton equation with column singleton**: Doubleton equation with column singleton can be applied whenever we have a row doubleton, that is we have two nonzero variables, and one of the variables is a column singleton, so we have
$$\exists\, i, j, k : \quad a_{ij} x_j + a_{ik} x_k = \mathbf{b}_{Li} \;\; j \neq k,\;\; a_{ij} \neq 0,\;\; a_{ik} \neq 0. \tag{37}$$
We if let x_k be the column singleton, then x_k leads to implied bounds on x_j, and we can remove x_k from the problem.

6.5 Selection of presolve rules

Before implementing a presolve and postsolve procedure we first needed to know which presolve rules were going to be relevant in attempting to reduce our specific set of test problems to empty. Additionally, since every rule that we apply in the presolve procedure adds to the computational time, in order to have a fast but effective solver we want to minimise the number of rules that we use while maximising the number of problems that we reduce to empty.

Therefore, it was decided to carry out a preliminary investigation using HiGHS, a high performance serial and parallel software for solving large-scale sparse linear programming (LP), mixed-integer programming (MIP) and quadratic programming (QP) models [4].

6.5.1 Presolve investigation with HiGHS

HiGHS has built in presolve functionality for both LPs and MIPs, and it has options to both report the presolve rules that it uses and to turn off presolve rules. Our methodology was then to use C++ (see test-presolve in [1]) to feed all of the 150,218 test problems into HiGHS, as LPs, and apply presolve with all of the available presolve rules initially turned on. We then checked the report on which rules were used, how many problems were reduced to empty, and the time taken.

We then repeated the process several times, each time turning off either a single presolve rule or a combination of rules. This allowed us to see if turning rows and combinations of rows off had an effect on the number of problems reduced to empty and computational time, so we could then approximate the optimal minimal set of rules needed to reduce the maximum number of problems to empty, while not increasing computational time.

After applying the HiGHS presolve to all the problems, with all the presolve rules turned on, the initial results obtained were:

- **Problems reduced to empty**: 83,512.
- **Time taken**: 17 seconds.
- **Presolve rules used**: Singleton row, Empty column, Free column substitution, Aggregator, Parallel rows and columns, Dominated column, Redundant row, Doubleton equation, Fixed column.

After repeating the process but turning off rules, it was found that the only combination of rules that did not have an effect on the number of problems reduced to empty were Aggregator and Doubleton Equation, and if they were turned off the time taken also stayed the same.

6.5.2 Final set of presolve rules

We based our final set of presolve rules on the HiGHS optimal set of rules, which is the complete set listed above minus Aggregator and Doubleton Equation. However, one of the rules that HiGHS used was Dominated Column, which is a dual rule, and consequently this rule was also discarded because we only focus on primal rules.

Additionally, we checked that we don't have any zero rows to start with, so it has been decided not to implement redundant row. Moreover, in our case, we will now treat singleton row for equalities and singleton row for inequalities as different rules. Finally, we also added Free row to our set of rules, as it is simple to implement, cheap in terms of computational time, and it was noted that there was a significant amount of problems in the problem set that had free rows in them.

Therefore, the final set of rules that were included in our presolve and postsolve procedure was: Free row, Singleton row equality, Singleton row inequality, Parallel rows, Empty column, Free column substitution and Fixed column.

6.6 Postsolve

In postsolve we work backwards, starting from the last variable that was removed in presolve, reversing the presolve procedure. In this process we attempt to find feasible values for variables that had been removed from the problem, and we reconstruct the problem back to its original state. In postsolve, we will use the lower bounds of constraints when calculating feasible values, unless for some reason the lower bound has become infinity.

If we had started off with a problem with coefficient matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$; and presolve had removed $c \leq m$ constraints and $v \leq n$ variables, thus leading to a reduced problem with coefficient matrix $\mathbf{A}_R \in \mathbb{R}^{(m-c)\times(n-v)}$; after having finished the postsolve procedure we will have added the c constraints and v variables back into the problem, which goes back to its original state, and we will have feasible values for subsets of the c constraints and v variables, of size $c_F \leq c$ and $v_F \leq v$ respectively.

It is important to note that c_F and v_F need not equal c and v because postsolve might not be able to find the feasible solution of some variables that have been removed using certain presolve rules. We will refer to the problem that constitutes the constraints and variables we have not found values for after postsolve as the final reduced problem, which will have $m - c_F$ constraints, $n - v_F$ variables, and coefficient matrix $\mathbf{A}_{RF} \in \mathbb{R}^{(m-c_F)\times(n-v_F)}$.

6.6.1 Rule-specific postsolve

We now describe the exact postsolve procedure to be carried out for each rule in our final set of rules.

1. **Free row**: In free row postsolve of a row i we just add row i back into the problem.
2. **Singleton row equality**: In singleton row equality postsolve of a row i we once again just add row i back into the problem.
3. **Singleton row inequality**: If we have a row singleton inequality i, and the nonzero column in the row is column k, the postsolve procedure will depend on whether or not a_{ik} is the only nonzero coefficient column k. If a_{ik} is the only non-zero coefficient in column k, the presolve will have removed both row i and column k from the problem, and we can obtain a feasible value for x_k as
$$x_k = \frac{\mathbf{b}_{\mathbf{L}i}}{a_{ik}}. \tag{38}$$
On the other hand, if a_{ik} was not the only non-zero coefficient in column k then presolve only led to implied bounds on x_k and did not remove row i, meaning that we do not do anything in postsolve.
4. **Parallel rows**: If postsolve identified rows i_1 and i_2 as parallel, where
$$a_{i_1 j} = r a_{i_2 j} \ \forall j, \ r \in \mathbb{R}. \tag{39}$$
and thus removed i_2, we add row i_2 back into the problem. Note however that we will only know the value of row i_2 if we have been able to find feasible values for all nonzero variables that were present in row i_1 when row i_2 was removed. Therefore, if there exists any variable x_k that we have not found feasible values for, row i_2 will not be in the set of c_F constraints that were removed; and the final reduced problem will contain row i_2 with all variables we have not found a feasible value for.
5. **Empty column**: If we have an empty column j in postsolve we just add column j back into the problem and we can arbitrarily set $x_j = 0$.
6. **Fixed column**: When we get a fixed column j in postsolve we know that the implied bounds on x_j are equal, and thus we can immediately set $x_j = \mathbf{l}_j = \mathbf{u}_j$ and add column j back into the problem. However, when we removed the column in presolve, we updated the problem by subtracting the j^{th} column of $\mathbf{A}$ times x_j from the constraints bounds $\mathbf{b_L}$ and $\mathbf{b_U}$, so in order for to revert the problem to it original state we need to reverse this operation.
7. **Free column substitution**: If we have a free column substitution in postsolve, where the free column is j and and the corresponding row is k, if we have found feasible values for all

variables x_z in k, $z \neq k$, then we can find a feasible value for x_j as

$$x_j = \frac{\mathbf{b_L}_k - \sum\limits_{z=1, z\neq j}^{n} a_{kz} x_z}{a_{kj}}, \tag{40}$$

regardless of whether or not row k is an inequality. However, if there are n variables in row k for which we have not found a feasible value, then we can not find a feasible value for x_j, and the final reduced problem will contain both row k with the n variables we do now have a feasible value for and column j.

6.7 Presolve and postsolve procedure implementation

We now give an outline of how we have implemented the code for the presolve and postsolve procedure implementation. In the actual implementation it was decided to add a new rule called row and column singleton, which we will use every time we have a row singleton that is also a column singleton, regardless of whether it is an inequality or an equality. The presolve and postsolve will be the same as in inequality row singletons that are also column singletons. We do this because in such cases the presolve and postsolve works regardless of the constraint type, and in the case that it is an equality we turn off the row and the column straight away rather than just the row, which may increase efficiency.

6.7.1 Problem data structures

In order to represent a given problem (as formulated in the presolve and postsolve procedure) in the code, we made use of C++ vectors. The coefficient matrix $\mathbf{A} \in \mathbb{Z}^{m\times n}$ of the problem was represented by a 2D vector, allowing us to store the whole matrix in a single variable. For constraints bounds and implied bounds, we used one one-dimensional vector to store lower bounds and one one-dimensional vector to store upper bounds.

We then used two vectors of booleans, one for rows and one for columns of, length m and n respectively, in order to keep track of which rows and columns of $\mathbf{A}$ were still active in the problem, that is, had not yet been removed in presolve or had been added back in postsolve. If we take the example problem (18) from Section 6.1.1, the data structure of the problem before applying any presolve or postsolve can be visualised in the following table:

$\mathbf{b_L}$		**T**	**T**	**T**	**T**	**T**	$\mathbf{b_U}$
0	**T**	1	0	0	0	0	∞
1	**T**	-1	1	0	0	0	∞
1	**T**	0	-1	0	-1	0	∞
1	**T**	0	1	1	0	0	∞
2	**T**	0	0	0	0	1	2

Therefore, we have that all the rows and columns of $\mathbf{A}$ are active in the problem. Then, after applying the first presolve iteration (see (19)), where we now know we applied free column substitution twice and singleton row equality once, we turned off rows 2, 3 and 4 and columns 2, 3 and 4, thus the problem is now represented by

$\mathbf{b_L}$		**T**	**T**	**F**	**F**	**F**	$\mathbf{b_U}$
0	**T**	1	0	0	0	0	∞
1	**T**	-1	1	0	0	0	∞
1	**F**	0	-1	0	-1	0	∞
1	**F**	0	1	1	0	0	∞
2	**F**	0	0	0	0	1	2

If the problem reduces to empty, all columns and rows will be set to false, and then postsolve will reverse the process turning all of them back on.

However, in order to be able to access the final reduced problem, we make use of an additional two postsolve vectors of booleans, again one of length m for rows and one of length n for columns, where we set all rows and columns to false to start with. Then, when we apply postsolve, we not only build back A turning on the original vectors of booleans, but if postsolve has found a feasible value for a variable or a constraint we turn that variable or constraint on in the corresponding postsolve vector of booleans. Therefore, once we have finished postsolve, the final reduced problem is given by all the rows and columns that are set to false in the postsolve vectors of booleans.

6.7.2 Presolve

In each presolve iteration we carry out two iterations. In the first iteration we iterate through the problem's active rows, and in each row we iterate through the presolve row rules, check if we can apply the rule to the row and if so, we apply it. In the second iteration we do the same, but iterating through the problem's active columns and iterating through and trying to apply the column rules. Every time that we detect parallel rows we check whether the system of rows is feasible.

In order to be able to apply postsolve later on, every time we apply a rule we need to keep track of the rule that we have applied, and the row and or column that has been removed from the problem. To this end, in our implementation we have assigned a unique rule ID to each presolve rule (see const.hpp in [1]). Then, we define a class member which is a presolve log stack for logging rules used. Every time we apply a rule we add a struct to the end of the stack, containing the index of affected row, the index of the affected column and the rule ID. If a given row rule has only affected a row and not a column we add a -1 for the column index or row index, and if a given column rule has only affected a column and not a row we add a -1 for the row index.

For each presolve rule, we have a function called `updateState<RuleName>` which is called every time the rule is applied. This function takes care of updating the state of the problem as according to the rule, setting to inactive the corresponding row and or columns and logging the rule to the log stack.

6.7.3 Postsolve

After having applied presolve, we then use the presolve log stack to apply postsolve. We start from the last rule that was logged to the stack, and apply postsolve as according to the rule id. For each rule we have a function `apply<RuleName>PostSolve` which carries out the rule-specific postsolve for that rule, given the relevant row and column indices. After having applied the rule-specific postsolve we remove the rule log from the top stack, and carry out the same procedure for rule log that is now at the top of the stack. We repeat this until the stack is empty, at which point we will have reversed the whole presolve procedure.

Every time we find a feasible value for a variable in postsolve we check that its implied bounds are met; and every time we have found feasible values for all the nonzero variables in a constraint we check that the constraint bounds are met.

6.8 Results and performance

Applying the finalised presolve and postsolve procedure for LPs to the full problem set we obtained the following results:

- 29,473 empty problems.
- 61,204 non-empty problems reduced to empty.
- 61,204 feasible solutions found.
- 0 infeasible problems found.
- 0 problems with unsatisfied constraints.
- 0.8 seconds taken on average.

We can see that presolve reduced to empty 61,024 problems, and postsolve successfully found feasible solutions to all the problems that were reduced to empty. Consequently, no infeasible problems were found and no unsatisfied constraints were found. The only way that we may detect infeasible problems when working with LPs, which implies that $x_i \in \mathbb{R}$, is by infeasible parallel rows. Therefore, it can be concluded that there are no infeasible parallel rows in the problem set.

However, we see that we reduced to empty 22,308 non-empty problems less than the HiGHS presolve. This could be because we are using a smaller set of rules, as we have not implemented the dual rule dominated column. Additionally, the order in which rules are applied may have an effect on the number of problems that are reduced to empty. Nevertheless, we have obtained a very fast and computationally efficient way to solve 40% of the problems in the problem set in only 0.8 seconds. On top of this, many of the problems that did not remove to empty were reduced in size after removing all the redundancy detectable by our procedure.

6.9 Integer feasibility

We will now consider an alternative implementation of our presolve and postsolve procedure, where we imposed the condition that all our variables need to be integer valued, that is, we treated the problems as integer programming problems (IPs). We do so in order to investigate the integer feasibility of the problems in the problem set.

6.9.1 Feasibility checks and operations

As a starting step, we first checked that out of the 61,204 problems that we solved with the LP presolve and postsolve procedure, 49,832 problems had an integer feasible solution. However, in an IP implementation, we want to ensure that if we find a feasible value for a variable in postsolve then either the feasible value is integer valued or the IP is infeasible. While IP presolve and postsolve is beyond the scope of this project, we identified several operations that we could apply in order to try to achieve an integrality restriction. The first operation that we noticed is that if we find an equation singleton row in presolve, and we see that

$$x_j = \frac{\mathbf{b}_{L_i}}{a_{ij}} \notin \mathbb{Z}, \tag{41}$$

then we can already set the problem to infeasible in presolve.

All other operations are relevant when attempting to find the feasible value of a variable in postsolve, and we find that the default value that we would assign to the variable in LP postsolve is not integer valued. First, if we have that

$$a_{ij}x_j \geq \mathbf{b_L}_k - \sum_{z=1, z\neq j}^{n} a_{kz}x_z, \tag{42}$$

we already have feasible values for all $x_z,\ z \neq j$, in row i; and

$$\frac{\mathbf{b_L}_k - \sum\limits_{z=1,z\neq j}^{n} a_{kz}x_z}{a_{ij}} \notin \mathbb{Z}. \tag{43}$$

Then, we can try to find a feasible value for x_j by rounding up (44) to the nearest integer if a_{ij} is positive, and rounding down if a_{ij} is negative, but we then need to check that the implied bounds on x_j are still satisfied, and that the constraint is still satisfied.

On the other hand, if we have the same situation as in 42 and 44 but for an equality, we can not change the value x_j due to the equality. In this case therefore, we will try to change the feasible values of the other variables in the row such that we obtain

$$\frac{\mathbf{b_L}_k - \sum\limits_{z=1,z\neq j}^{n} a_{kz}x_z}{a_{ij}} = 1. \tag{44}$$

Then, if the new values for other variables in the row have not led to any unsatisfied constraints, and 1 satisfied the implied bounds of x_j, we set $x_j = 1$ and update the feasible values of other variables. However, in our implementation we will limit this operation to rows with only two variables.

6.9.2 Integer feasibility results

After implementing the operations described above, we obtained the following results:

- 61,204 non-empty problems reduced to empty.
- 60,740 integer feasible solutions found.
- 0 infeasible problems found.
- 464 problems led to unsatisfied constraints errors.
- 0.8 seconds taken on average.

Therefore, our IP procedure identified 11,372 integer feasible problems more that our LP procedure in the same time. However, we can see that 464 problems led to unsatisfied constraints errors, that is, we found feasible values in postsolve that did not satisfy constraints.

It was decided to manually investigate several of the problems with unsatisfied constraint errors. All the problems we checked had failed because presolve had fixed the value for a variable, and when that variable was then used to calculate the value of another variable where rounding operations were needed, the constraint could not be made to satisfy the rounding operations. For instance, in problem 29,473 we had the constraint

$$0 \leq -32x_1 + x_4 \leq 30, \tag{45}$$

where parallel rows in presolve had led to the upper bound of the constraint being 30, and the constraint had been removed by free column substitution in presolve. Then, postsolve fixed $x_4 = -1$ early in the postsolve procedure by inequality singleton row, where $-1 \leq x_4 \leq \infty$. Therefore, when it came to find a feasible value for x_1 we calculated $x_1 = -1/32$ and rounded down to $x_4 = -1$. Substituting these feasible values back into 46, we see that it is not satisfied. Furthermore, the inequality

$$0 \leq -32x_1 - 1 \leq 30 \tag{46}$$

does not have any integer solutions. However, since we had $-1 \leq x_4 \leq$, x_4 could be set to another feasible value, for instance $x_4 = 0$, which would lead to $x_1 = 0$ and the constraint would be satisfied. The whole problem was, in fact, integer feasible. In a proper IP presolve and postsolve procedure, all such relationships that are leading to errors in our integer restricted implementation would already be captured in presolve, and we would be guaranteed that if the problem reduces to empty, then it has an integer feasible solution. However, since our presolve is not optimised for IPs, we do not claim that all the problems that we reduce to empty are integer feasible.

Nevertheless, by adding the integer feasibility operations and checks to our LP-specific implementation, we have achieved a procedure that finds an integer feasible solution to almost all the problems that it reduces to empty, as it only failed to produce an integer feasible solution in 0.75% of the problems. Therefore, even though our integer restricted presolve and postsolve procedure has some limitations, it can be used to determine integer feasibility of 40% of the problems in the problem set in a very short amount of time.

7 Combined solver

In our final solver we combine our presolve and postsolve implementation and our primal simplex solver together. Note that we use the primal simplex as opposed to the dual simplex for both improved runtime and robustness (since this is out "slow" solver, we need to ensure it is as robust as possible).

Our aim in combining both methods is to solve as many problems as possible with the presolve and postsolve procedure, which is very fast but only solves a subset of the problems, and then solve the rest of the problems with the primal simplex method. The method we use for combining both solvers is very simple: try applying the presolve and postsolve procedure and if the problem doesn't reduce to empty, use the simplex method.

7.1 Trivial problems

Before we use any of the advanced solvers to check the feasibility of a problem, we conduct some basic checks on the problem to ensure that it is nontrivial. Firstly, we check that the problem is non-empty. Then we check that we do not have a single basic constraint with appropriate bounds (i.e. either all zero or at least one variable if a bound exists).

7.2 Implementation

The high level implementation follows the following simple logic for each problem in our set:

- Read in new problem.
- Conduct basic checks to ensure problem is nontrivial. If it is, move on to the next problem.
- Reformat the problem for presolve methods.
- Try solving the problem with presolve methods. If this yields a (verified) feasible solution, move on to the next problem. If it does not, move to try the simplex solver.
- Reformat the problem for the primal simplex solver.
- Try solving this problem with the primal simplex solver. If there is no verified feasible solution, record the error and move on.

7.3 Results

This solver runs in approximately 2.5 seconds on average with the following results:

- $29,537$ empty problems.
- $120,646$ feasible problems.
- 25 infeasible problems.
- 10 problems where a program error occurred.

Therefore we can see that as expected, it runs faster than primal simplex alone but slower than the presolve and postsolve procedure. However, combining both methods has led to a more robust solver, since the presolve and postsolve procedure took care of some of the problems that were yielding errors in the simplex method, and out of the whole problem set of 150,218 problems only 10 led to errors.

7.4 Possible improvements

While the solver we have built is still effective for solving these problems, there are still efficiencies that could be made to improve the solver. These include:

- Modifying the simplex solver to interpret the reduced problem from presolve methods. This could significantly reduce the size of the problem the simplex solver has to deal with hence increasing the speed of the program.
- Through extensively checking which type of problems we are being asked to solve, we could change the order of operations we perform to reduce the expected number of operations required per problem. For example if we want to introduce a check for if we already have the answer from a function. We would need to know the expected proportion of cases where the condition is met and hence decide if the cost of the check itself is less than the cost of leaving the function to exit normally on average.
- Since some of the problems are repeated in close succession, we could keep a "history" of the last n problems and check if the problem has been recently solved. When considering this, how we efficiently encode the solved problems and the lookup speed of finding a problem in a data structure would have to be considered.
- Even-though our presolve and postsolve implementation is fairly efficient and runs very fast, the small computational time is also due to the fact that most problems are rather small in size, and we are applying rules that are fairly inexpensive. The implementation, however, could be made more efficient to run even faster in future research. A good way in which this could be done is to convert the problem coefficient matrix to a dense matrix rather than a sparse matrix before applying the procedure. This would reduce the number of iterations and comparisons that we need to do within each iteration of presolve, since we would not have to check whether the coefficients in the matrix are zero. We do this to some extent by finding all the rows and columns that are active and nonzero before each application of presolve rules, however, the code could be modified to work specifically with dense matrices.
- In future research, a proper working IP presolve and postsolve procedure could be implemented. This would ensure that all problems that are reduced to empty are indeed integer feasible and it would work back an integer feasible solution for all problems. Therefore, not only it would be a very computationally efficient way of determining integer feasibility for around 40% of the problems, which our modified LP specific implementation already does with some limitations, but the IP presolve would be suited to establish integer infeasibility of problems.

8 Conclusion

We have produced a solver combining a presolve and postsolve procedure and a primal simplex solver, in combination with Bland's rule for termination. Applying the solver to the full problem set of compiler generated problems provided by [2], we have seen that our solver solves all the problems in the problem set apart from 10, in the very short amount of time of 2.5 seconds. As could be expected, the combined solver runs faster than the primal simplex method by itself and solves a lot more LPs than the presolve and postsolve procedure on its own, as it takes advantage of the benefits of each method. Therefore, we have produced a lightweight, reliable and efficient LP solver with improved mathematical efficiency which is heavily optimised for use in compilers.

Additionally, we made progress in assessing the integer feasibility of the problems in the problem set. Our combined solver found that 25 problems are LP infeasible, that is, the continuous problem is infeasible. This therefore implies that at least 25 problems in the full problem set are integer infeasible, which is relevant to the original research in [2]. Moreover, by modifying our LP presolve and postsolve procedure by adding integer feasibility checks and operations, we obtained a procedure that is able to guarantee integer feasibility for the large majority of the problems it reduces to empty. Therefore, it can determine the integer feasibility of a rather large subset of the test problems in a very short amount of time. Although our modified procedure has some limitations, it is already relevant to [2], and we feel that further research could greatly benefit from a fully working integer programming presolve and postsolve procedure, which could also determine integer infeasibility.

References

1. Guy M, Sais JGR. Honours Project Code Base (hons-project) - https://github.com/maxguy2001/hons-project. 2023.

2. Grosser T, Et al . Fast linear programming through transprecision computing on small and sparse data. *Proceedings of the ACM on Programming Languages.* 2020;4 (OOPSLA):1-28.

3. Dantzig GB. *Linear Programming and Extensions.* Princeton University Press 1963.

4. Huangfu Q, Hall JAJ. Parallelizing the dual revised simplex method. *Mathematical Programming Computation.* 2018;10 (1):119-142.

5. Hillier FS, Lieberman GJ. *Introduction to Operations Research, Ninth Edition (Chapter 6).* McGraw-Hill 2010.

6. Gass S, Vinjamui S. Cycling in linear programming problems. *Computers Operations Research.* 2004;31 (2):303-311.

7. Hall J, McKinnon K. The simplest examples where the simplex method cycles and conditions where EXPAND fails to prevent cycling. *Mathematical Programming.* 2000;100 (1).

8. Bland R. New finite rules for the simplex method. *Mathematics of Operations Research.* 1977;2 (2):103-107.

9. Andersen E, Andersen K. Presolving in linear programming. *Mathematical Programming.* 1995;71:221-245.