Tanner Crane

Math 464

Project Report

1. Revised Simplex Function (RSF)

This function takes as input the constraint matrix A, the right hand-side vector b, the cost vector c, as well as the rules for choosing the entering and leaving variables. The entering\_rule can be set to 'most\_negative' or 'first\_negative', while the leaving\_rule can be set to 'break\_ties\_arbitrarily' or 'smallest\_index'.

The function first checks if the problem is feasible, then initializes the variables and pivot counter. It then enters a loop that continues until optimality is reached or unboundedness is detected. At each iteration, it selects the entering variable based on the specified rule, calculates the leaving variable ratios and selects the leaving variable based on the specified rule. It then updates the basis and solution, increments the pivot counter, and continues to the next iteration. Once optimality is reached, the function outputs the optimal solution, optimal value, and number of pivots, and elapsed time using the tic() and toc() functions. If unboundedness is detected, an error is thrown.

1. Simplex Tableau Method:

This code implements the simplex algorithm to solve linear programming problems in standard form:

maximize c'x

subject to:

Ax ≤ b

x ≥ 0.

The simplex algorithm is an iterative method that starts with a feasible basic solution and moves along the edges of the feasible region in search of the optimal solution.

The input parameters to the function are:

* A: the matrix of coefficients of the constraints
* b: the vector of constants in the constraints
* c: the vector of coefficients of the objective function
* enter\_rule: the rule to determine the entering variable in each iteration ('most\_negative' or 'smallest\_index')
* leave\_rule: the rule to determine the leaving variable in each iteration ('break\_ties' or 'lexicographic')

The output variables of the function are:

* x: the optimal solution (or an empty vector if the LP is infeasible or unbounded)
* optimal\_cost: the optimal value of the objective function (or an empty vector if the LP is infeasible or unbounded)
* pivots: the number of simplex pivots performed (or NaN if the maximum number of iterations is reached)

The code starts by checking if the LP is feasible by looking for any negative values in the vector b. If any negative values are found, the LP is infeasible, and the function returns empty vectors for x and optimal\_cost and sets pivots to 0.

If the LP is feasible, the code adds slack variables to the constraints to convert the LP to standard form. It also initializes the tableau by appending the vector c to the bottom of the matrix [A b] and appending a 0 to the end of the vector b.

The code then enters a loop that runs until an optimal solution is found or the LP is determined to be unbounded or the maximum number of iterations is reached. I put the maximum number of iterations reached because the loop seemed to run indefinitely. I wasn’t sure how to fix this problem.

The loop consists of the following steps:

1. Check if the maximum number of iterations has been reached. If so, the function returns empty vectors for x and optimal\_cost and sets pivots to NaN.
2. Determine the entering variable by computing the reduced costs for each variable and selecting the one with the most negative reduced cost if enter\_rule is 'most\_negative' or the one with the smallest index if enter\_rule is 'smallest\_index'.
3. If no entering variable is found, the solution is optimal, and the function computes the optimal solution and optimal\_cost from the tableau and breaks out of the loop.
4. Determine the leaving variable by computing the ratios of the constants to the coefficients of the entering variable for each constraint and selecting the one with the smallest ratio if leave\_rule is 'break\_ties' or the one with the smallest lexicographic index if leave\_rule is 'lexicographic'.
5. If no leaving variable is found (i.e., all the ratios are non-positive), the LP is unbounded, and the function returns empty vectors for x and optimal\_cost and sets pivots to NaN.
6. Update the tableau by performing a pivot operation on the entering and leaving variables. This involves dividing the leaving row by the pivot element, subtracting multiples of the leaving row from the other rows to eliminate the entering variable, and setting the pivot element to 1.
7. Increment the pivot count

At the end of the loop, the function returns the optimal solution x, the optimal value of the objective function optimal\_cost, and the number of simplex pivots (pivots).

Jimbo LP formulation:

The objective function coefficients are defined in the first block of code using a nested loop over **k** and **i**. **f** is initially an empty matrix, and for each **k** and **i**, the matrix **c(i,k)** is multiplied by a row vector of ones with length **m**, resulting in a row vector of length **m**. The resulting row vector is then appended to the end of the **f** matrix.

The inequality constraints matrix **A** and vector **b** are defined in the second block of code. **A** is initially created as a sparse matrix with dimensions **m\*n\*p** by **m\*n+2\*p\*n**, which contains all zeros. The **sparse** function is used to create an empty sparse matrix. The vector **b** is initially an empty matrix.

The nested loops over **k** and **j** create the rows of **A** and **b**. For each **k** and **j**, a row vector **row** of length **m\*n+2\*p\*n** is created, initially containing all zeros. For each **i**, the element **(j-1)\*n+i** of the row vector is set to **t(i,j)**. Then, the elements **m\*n+(k-1)\*n+j** and **m\*n+p\*n+(k-1)\*m+j** of the row vector are set to 1. The resulting row vector **row** is then assigned to the appropriate row of **A** using the formula **(k-1)\*m+j**. Finally, the element **h(k,j)** is appended to the end of the **b** vector.

The equality constraints matrix **Aeq** and vector **beq** are defined in the third block of code. **Aeq** is initially created as a matrix of zeros with dimensions **n\*p** by **n\*m+2\*p\*n**, and **beq** is initially a vector of zeros with length **n\*p**. The nested loops over **k** and **i** create the rows of **Aeq** and **beq**. For each **k** and **i**, two row vectors **row1** and **row2** of length **m\*n+2\*p\*n** are created, initially containing all zeros. For each **j**, the element **(j-1)\*n+i** of **row1** is set to 1, and the element **m\*n+(k-1)\*n+i** of **row1** is set to 1. Similarly, for each **j**, the element **m\*n+p\*n+(k-1)\*n+i** of **row2** is set to 1. The resulting row vectors **row1** and **row2** are then assigned to the appropriate rows of **Aeq** using the formulas **(k-1)\*n+i** and **n\*p+(k-1)\*n+i**, respectively. Finally, the element **d(i,k)** is assigned to the appropriate element of **beq**.

The lower bounds and upper bounds for the decision variables are defined in the fourth block of code. **lb** is initially a row vector of zeros with length **m\*n+2\*p\*n**, and **ub** is an empty matrix.

Finally, the **linprog** function is called with the input arguments **-f**, **A**, **b**, **Aeq**, **beq**, **lb**, and **ub**, which solves the LP problem and returns the optimal solution vector **x**. However, when running this code with the randomly generated data, I encountered the following error:

Unable to perform assignment because the indices on the left side are not compatible with the size of the right side.

Error in Jimbo (line 21)

A((k-1)\*m+j,:) = row;

I wasn’t able to determine a solution.

5. To modify the Jimbo Enterprises problem, one possible approach is to change the constraints such that they become infeasible. For instance, we could increase the right-hand side of one of the inequality constraints such that no feasible solution exists. We can set one of the h(k,j) values to be larger than the sum of t(i,j) values for all i=1,...,n.

Alternatively, we could set one of the d(i,k) values to be larger than the sum of c(i,k)\*t(i,j) values for all j=1,...,m.

To generate instances, we can use the modified problem formulation and randomly generate values for the input parameters such as c(i,k), t(i,j), h(k,j), and d(i,k). We can then set one of the h(k,j) or d(i,k) values as described above to create an infeasible instance.

For each instance, we can first use the linprog function to check if the LP is feasible or not. If the LP is infeasible, then we can use the revised simplex method and tableau simplex method implementations to detect infeasibility. We can measure the running time and number of iterations required for each method to detect infeasibility. We can repeat this process for several instances and compare the performance of the methods. Unfortunately, we weren’t able to do any meaningful comparisons because the code didn’t run.

Comparison Algorithms Code:   
The code describes the steps to modify the Jimbo Enterprises problem to create infeasible instances, generate random input parameters for these instances, and compare the performance of the revised simplex method, tableau simplex method, and linprog function for detecting infeasibility.

To create infeasible instances, one can increase the right-hand side of one of the inequality constraints such that no feasible solution exists. Another approach is to set one of the d(i,k) values to be larger than the sum of c(i,k)\*t(i,j) values for all j=1,...,m or set one of the h(k,j) values to be larger than the sum of t(i,j) values for all i=1,...,n.

To generate instances, the modified problem formulation can be used, and random values can be assigned to input parameters such as c(i,k), t(i,j), h(k,j), and d(i,k). One of the h(k,j) or d(i,k) values can be set as described above to create an infeasible instance.

The revised simplex method and tableau simplex method implementations can be used to detect infeasibility. The linprog function in MATLAB can also be used with default options for choosing entering/leaving variables. For each instance, the linprog function can be used first to check if the LP is feasible or not. If the LP is infeasible, the revised simplex method and tableau simplex method implementations can be used to detect infeasibility. The running time and number of iterations required for each method to detect infeasibility can be measured and compared for several instances.