Trabalho Prático 1 de Pesquisa Operacional

Nome: Ronald Davi Rodrigues Pereira
Matrícula: 2015004437

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <limits.h>

double **matrixAllocation(int lines, int columns) // Function to allocate
the matrix
{
    int i;
    double **matrix;

    matrix = (double**) calloc(lines,sizeof(double*));
    for(i = 0; i < lines; i++)
        matrix[i] = (double*) calloc(columns, sizeof(double*));

    return matrix;
}

double **matrixDisallocation(double **matrix, int lines) // Function to
disallocate the matrix
{
    int i;

    for(i = 0; i < lines; i++)
        free(matrix[i]);
    free(matrix);

    return NULL;
}

void printLineMatrix(double **matrix, int lines, int columns) // Function to
print the matrix in one line
{
    int i, j;

    printf("{");
    for(i = 0; i < lines; i++)
    {
        for(j = 0; j < columns; j++)
        {
            if(j == 0)
                printf("{");

            if(matrix[i][j] == 0)
                printf("0");
            else if((matrix[i][j] - (int)matrix[i][j]) == 0) // If the cell
is a integer number, do not print it with .3lf
                printf("%.0lf", matrix[i][j]);
            else
                printf("%.3lf", matrix[i][j]);


            if(j < columns-1)
                printf(",");
            else
                printf("}");
        }
        if(i < lines-1)
            printf(",");
    }
    printf("}\n");
}
```

```
bool detectPrimalDual(char input) // Function to detect if, in mode 2, is
primal or dual solving mode
{
    bool mode; // 0 is Primal, 1 is Dual

    if(input == 'P')
        mode = 0;

    else if(input == 'D')
        mode = 1;

    return mode;
}

double **matrixBuilder(char *input, double **matrix) // Function that
converts the input line matrix to a two-dimension matrix
{
    int i = 0, j = 0, k = 0, l = 0;
    bool neg = 0;
    char c, num[50];

    while(1)
    {
        c = input[l];
        l++;

        if(((int)c >= 48 && (int) c <= 57) || c == '.') // C is a number or
a dot
        {
            num[k] = c;
            k++;
            num[k] = '\0';
        }

        else if(c == '-')
            neg = 1;

        else if(c == ',')
        {
            matrix[i][j] = atof(num);
            if(neg)
            {
                matrix[i][j] *= -1;
                neg = 0;
            }
            k = 0;
            j++;
        }

        else if(c == '}')
        {
            matrix[i][j] = atof(num);
            if(neg)
            {
                matrix[i][j] *= -1;
                neg = 0;
            }
            k = 0;
            i++;
            j = 0;

            c = input[l];
            l++;

            if(c == '}') // End of input file
                break;
        }
    }
```

```
    return matrix;
}

double **buildTableau(double **matrix, int *lines, int *columns) // Function
that builds the Tableau matrix
{
    int i, j;
    int origcolumns = *columns;
    double **tableau;

    for(i = 0; i < *columns; i++) // Do the -C^t part
    {
        matrix[0][i] *= -1;
    }

    *columns += (2 * (*lines - 1)); // Adds the operations matrix, the
identity matrix and b vector on the number of columns

    tableau = matrixAllocation(*lines, *columns);

    for(i = 0; i < *lines; i++)
    {
        for(j = 0; j < *columns; j++)
        {
            if(j < (*lines-1) && j == (i - 1)) // We are on the operations
matrix
                tableau[i][j] = 1;

            else if(j >= (*lines-1) && j <= (*columns - (*lines + 1))) // We
are on the A matrix
                tableau[i][j] = matrix[i][j-(*lines-1)];

            else if(j >= ((*lines-1)+(origcolumns-1)) && j < (*columns-1) &&
j + 1 == i + ((*lines-1)+(origcolumns-1))) // We are on the identity matrix
                tableau[i][j] = 1;

            else if(j == (*columns-1)) // We are on the b matrix
                tableau[i][j] = matrix[i][origcolumns-1];
        }
    }

    matrixDisallocation(matrix, *lines);

    return tableau;
}

double **buildAuxiliarToTableau(double **matrix, int *lines, int
*columns) // Function that builds the auxiliar matrix in Tableau
{
    int i, j;
    double **auxiliar;

    *columns += (*lines)-1;

    auxiliar = matrixAllocation(*lines, *columns);

    for(j = 0; j < *columns; j++)
    {
        for(i = 0; i < *lines; i++)
        {
            if(j < (*columns)-(*lines) && i != 0)
                auxiliar[i][j] = matrix[i][j];

            else if(j == (*columns)-1)
                auxiliar[i][j] = matrix[i][j-(*lines)+1];
        }
    }
```

```
    for(i = (*lines)-1, j = (*columns)-2; i > 0; i--, j--)
    {
        auxiliar[i][j] = 1;
        auxiliar[0][j] = 1;
    }

    for(i = 1; i < (*lines); i++)
    {
        for(j = 0; j < (*columns); j++)
            auxiliar[0][j] -= auxiliar[i][j];
    }

    matrixDisallocation(matrix, *lines);

    return auxiliar;
}

void unviableCertificate(double **matrix, int lines) // Function that
outputs the unviable certificate
{
    int i;

    printf("PL inviavel, aqui esta um certificado {{");
    for(i = 0; i < lines-1; i++)
    {
        if(matrix[0][i] == 0)
            printf("0");
        else if((matrix[0][i] - (int)matrix[0][i]) == 0)
            printf("%.0lf", matrix[0][i]);
        else
            printf("%.3lf", matrix[0][i]);

        if(i < lines-2)
            printf(",");
    }

    printf("}}\n");
}

void unlimitedCertificate(double **matrix, int lines, int columns, int base,
int *bases) // Function that outputs the unlimited certificate
{
    int i, j, k;

    printf("PL ilimitada, aqui esta um certificado {{");
    for(j = lines-1; j < columns-lines; j++)
    {
        for(k = 0; k < lines-1; k++)
        {
            if(j == bases[k])
            {
                for(i = 1; i < lines; i++)
                {
                    if(matrix[i][j] == 1)
                    {
                        if(matrix[i][base] == 0)
                            printf("0");
                        else if((matrix[i][base] - (int)matrix[i][base]) ==
0)

                            printf("%.0lf", -1*matrix[i][base]);
                        else
                            printf("%.3lf", -1*matrix[i][base]);
                    }
                }
                break;
            }
        }

        if(j == base)
```

```
            printf("1");

        else if(k == lines-1)
            printf("0");

        if(j < columns-lines-1)
            printf(",");
    }
    printf("}}\n");
}

void viableSolution(double **matrix, int lines, int columns, int *bases) //
Function that outputs the optimum viable solution, the objective value and
the dual solution
{
    int i, j, k;

    printf("Solucao otima x = {{");
    for(j = lines-1; j < columns-lines; j++)
    {
        for(k = 0; k < lines-1; k++)
        {
            if(j == bases[k])
            {
                for(i = 1; i < lines; i++)
                {
                    if(matrix[i][j] == 1)
                    {
                        if((matrix[i][columns-1] - (int)matrix[i][columns-
1]) == 0)
                            printf("%.0lf", matrix[i][columns-1]);
                        else
                            printf("%.3lf", matrix[i][columns-1]);
                    }
                }
                break;
            }
        }
            if(k == lines-1)
                printf("0");

            if(j < columns-lines-1)
                printf(",");
            else
                printf("}");
    }

    printf("}, com valor objetivo ");

    if((matrix[0][columns-1] - (int)matrix[0][columns-1]) == 0)
        printf("%.0lf", matrix[0][columns-1]);
    else
        printf("%.3lf", matrix[0][columns-1]);

    printf(", e solucao dual y = {{");

    for(i = 0; i < lines-1; i++)
    {
        if((matrix[0][i] - (int)matrix[0][i]) == 0)
            printf("%.0lf", matrix[0][i]);
        else
            printf("%.3lf", matrix[0][i]);

        if(i < lines-2)
            printf(",");
        else
            printf("}");
    }
    printf("}\n");
```

```
}

double **primalTableauSolver(double **matrix, int lines, int columns, int
mode) // Function that solves the given LP in the primal Tableau algorithm,
using Bland's Law
{
    int i, j, base, bases[lines-1], pivot, numberofnegatives, ispositive,
unviableflag;
    double minimum, aux, linedivider, multiplier;

    for(i = lines-2, j = 2; i >= 0; i--, j++)
    {
        bases[i] = columns-j;
    }

    while(1)
    {
        if(mode == 2)
            printLineMatrix(matrix, lines, columns);

        base = 0;
        for(i = lines-1; i < columns-1; i++)
        {
            if(matrix[0][i] < 0)
            {
                base = i;
                break;
            }
        }

        if(base == 0) // C^t >= 0
            break;

        else if(base != 0) // Unlimited and Unviable LP test
        {
            numberofnegatives = 0; // Unlimited test
            for(i = 1; i < lines; i++)
            {
                if(matrix[i][base] <= 0)
                    numberofnegatives++;
            }
            if(numberofnegatives == lines-1 && mode == 1) // Unlimited LP
            {
                unlimitedCertificate(matrix, lines, columns, base, bases);
                return matrix;
            }

            unviableflag = 0;
            for(i = 1; i < lines; i++) // Unviable test
            {
                if(matrix[i][columns-1] < 0) // b is negative
                {
                    ispositive = 0;
                    for(j = lines-1; j < columns-1; j++)
                    {
                        if(matrix[i][j] >= 0)
                            ispositive++;
                    }
                    if(ispositive == (columns-lines))
                    {
                        unviableflag = 1;
                        for(j = 0; j < columns; j++)
                            matrix[i][j] *= -1;
                    }
                }
            }
            if(unviableflag && mode == 1)
            {
                matrix = buildAuxiliarToTableau(matrix, &lines, &columns);
```

```
                matrix = primalTableauSolver(matrix, lines, columns, 3);
                unviableCertificate(matrix, lines);
                return matrix;
            }
        }

        minimum = INT_MAX;
        for(i = 1; i < lines; i++)
        {
            if(matrix[i][base] > 0)
            {
                aux = (matrix[i][columns-1] / matrix[i][base]);
                if(aux < minimum)
                {
                    minimum = aux;
                    pivot = i;
                }
            }
        }

        bases[pivot-1] = base;

        linedivider = matrix[pivot][base];

        for(i = 0; i < columns; i++)
            matrix[pivot][i] /= linedivider;

        for(i = 0; i < lines; i++)
        {
            if(matrix[i][base] != 0 && i != pivot)
            {
                multiplier = -1*(matrix[i][base] / matrix[pivot][base]);

                for(j = 0; j < columns; j++)
                    matrix[i][j] += multiplier * matrix[pivot][j];
            }
        }
    }

    if(mode == 1)
        viableSolution(matrix, lines, columns, bases);

    return matrix;
}

double **dualTableauSolver(double **matrix, int lines, int columns) //
Function that solves the given LP in the Dual Tableau algorithm, using
Bland's Law
{
    int i, j, base, pivot;
    double minimum, aux, linedivider, multiplier;

    while(1)
    {
        printLineMatrix(matrix, lines, columns);

        base = 0;
        for(i = 1; i < lines; i++)
        {
            if(matrix[i][columns-1] < 0)
            {
                base = i;
                break;
            }
        }

        if(base == 0)
            break;
```

```
        minimum = INT_MAX;
        for(j = lines-1; j < columns-1; j++)
        {
            if(matrix[base][j] < 0)
            {
                aux = (matrix[0][j] / abs(matrix[base][j]));

                if(aux < minimum)
                {
                    minimum = aux;
                    pivot = j;
                }
            }
        }

        linedivider = matrix[base][pivot];

        for(i = 0; i < columns; i++)
            matrix[base][i] /= linedivider;

        for(i = 0; i < lines; i++)
        {
            if(matrix[i][pivot] != 0 && i != base)
            {
                multiplier = -1*(matrix[i][pivot] / matrix[base][pivot]);

                for(j = 0; j < columns; j++)
                    matrix[i][j] += multiplier * matrix[base][j];
            }
        }
    }
    printLineMatrix(matrix, lines, columns);

    return matrix;
}

int main()
{
    char *input; // Input matrix
    int lines, columns; // Matrix dimensions
    double **matrix; // Two-dimension array to represent the LP
    int mode, primaldual; // Modes of the execution
    char option; // Primal or dual mode of execution

    printf("Welcome to C-Implex (my implementation of Simplex algorithm
using Bland's Law)\n\nAuthor: Ronald    Davi Rodrigues Pereira\nBS student
of Computer Science in Federal University of Minas Gerais\n\nOption
Menu:\n\t1 - Apply the Simplex algorithm (using Bland's Law) on a Linear
Programming and outputs the optimized solution or a certificate of
illimitability or inviability\n\t2 - Given a viable and limited Linear
Programming, it consults the user to use the primal or dual C-Implex
implementation and outputs the solution\n\n");
    printf("Insert a mode:\n> ");
    scanf("modo %d", &mode);
    getc(stdin); // Gets the '\n' token from input

    if(mode == 2)
    {
        printf("Insert the mode (P for primal / D for dual):\n> ");
        scanf("%c", &option);
        getc(stdin); // Gets the '\n' token from input
        primaldual = detectPrimalDual(option);
    }

    printf("Number of lines:\n> ");
    scanf("%d", &lines);
    getc(stdin); // Gets the '\n' token from input
    printf("Number of columns:\n> ");
    scanf("%d", &columns);
```

```
    getc(stdin); // Gets the '\n' token from input
    lines++;
    columns++;

    matrix = matrixAllocation(lines, columns); // Function to allocate the
matrix

    printf("Insert the matrix input:\n> ");
    scanf("%s", input);

    matrixBuilder(input, matrix); // Function to build the matrix from the
input file

    /* First mode implementation */

    if(mode == 1)
    {
        matrix = buildTableau(matrix, &lines, &columns); // Function that
builds the Tableau matrix

        matrix = primalTableauSolver(matrix, lines, columns, mode); //
Primal Tableau Simplex algorithm solver
    }

    /* Second mode implementation */

    else if(mode == 2)
    {
        if(primaldual == 0) // Primal solve mode
        {
            matrix = buildTableau(matrix, &lines, &columns); // Function
that builds the Tableau matrix

            matrix = primalTableauSolver(matrix, lines, columns, mode); //
Primal Tableau Simplex algorithm solver
        }

        else if(primaldual == 1) // Dual solve mode
        {
            matrix = buildTableau(matrix, &lines, &columns); // Function
that builds the Tableau matrix

            matrix = dualTableauSolver(matrix, lines, columns); // Dual
Tableau Simplex algorithm solver
        }
    }

    matrixDisallocation(matrix, lines); // Function to free the allocated
space for the matrix

    return 0;
}
```