Name – Ankush Chatterjee Roll – 13MA20008 Date – 07/ 01/ 16

Code contains 3 files

1. *matrix.cpp -> Custom built library for matrix operations*
2. *matrix.h -> Header file for matrix.cpp*
3. *bfs.cpp -> Code to solve using Basic Feasible Solutions method*

1. Code for matrix.cpp

```
#include <stdio.h>
#include <iostream>
#include <math.h>
#include "matrix.h"

using namespace std;


//---------------------------------------------------------------------
//          Constructor for the matrix class
//---------------------------------------------------------------------
Matrix::Matrix(int a=0, int b=0)
{
   rows = a;
   cols = b;
   mat = new double*[rows];
   for(int i = 0; i < rows; ++i)
   {
      mat[i] = new double[cols];
      for(int j = 0; j < cols; j++)
      {
         mat[i][j] = 0;
      }
   }
}

//---------------------------------------------------------------------
//          Function to read a matrix
//---------------------------------------------------------------------
void Matrix::read_matrix()
{
   for(int i = 0; i < rows; i++)
   {
      for(int j = 0; j<cols; j++)
      {
         cin>>mat[i][j];
      }
   }
}

//---------------------------------------------------------------------
//          Function to display a matrix
//---------------------------------------------------------------------
void Matrix::display_matrix()
{
   for(int i = 0; i < rows; i++)
   {
      for(int j = 0; j<cols; j++)
      {
         cout<<mat[i][j]<<"\t";
      }
      cout<<endl;
   }
   cout<<endl;
}

//---------------------------------------------------------------------
//          Function to perform deep copy of matrix
//---------------------------------------------------------------------
Matrix Matrix::copy()
{
   int i,j;
   Matrix clone(rows, cols);
```

```
   for (int i = 0; i < rows; ++i)
   {
      for (int j = 0; j < cols; ++j) clone.mat[i][j] = this->mat[i][j];
   }
   return clone;
}

//---------------------------------------------------------------------
//          Function to find transpose of a matrix
//---------------------------------------------------------------------
Matrix Matrix::transpose()
{
   Matrix result(cols, rows);
   int i, j;
   for(i = 0; i < rows; i++)
   {
      for(j = 0; j < cols; j++)
      {
         result.mat[j][i]  = this->mat[i][j] ;
      }
   }
   return result;
}

//---------------------------------------------------------------------
//          Function to find the first non-zero element
//---------------------------------------------------------------------
void Matrix::update_leading_0s(int *leading_0s, Matrix a)
{
   int i, j;
   for(i = 0; i < a.rows; ++i)
   {
      leading_0s[i] = 0;
      for(j=0; (fabs(a.mat[i][j]) < 0.00001) && (j < a.cols) ;++j) leading_0s[i]++;
   }
}


//---------------------------------------------------------------------
// Function to rearrange arrange A such that pivot positions have 1
//---------------------------------------------------------------------
void Matrix::pivot_rearrange(int *leading_0s, Matrix a)
{
   int l, remrow, i, k, lastrow, large;
   double *rowtemn = new double[a.cols];
   lastrow = rows-1;

   for(l = 0; l < a.rows; ++l)
   {
      large = leading_0s[0];
      for(i = 0; i < a.rows; ++i)
      {
         if( large <= leading_0s[i])
         {
            large=leading_0s[i];
            remrow=i;
         }
      }

   leading_0s[remrow] = leading_0s[lastrow];
   leading_0s[lastrow] = -1;

   for(k = 0; k < a.cols; ++k)   rowtemn[k] = a.mat[lastrow][k];
   for(k = 0; k < a.cols; ++k)   a.mat[lastrow][k] = a.mat[remrow][k];
   for(k = 0; k < a.cols; ++k)   a.mat[remrow][k] = rowtemn[k];

   lastrow--;
   }
}

//---------------------------------------------------------------------------
//          Function definition to scale a A
//---------------------------------------------------------------------------
void Matrix::scale_A(int *leading_0s, Matrix a)
{
   int i,j;
   double divisor;
   for(i = 0; i < a.rows; ++i)
   {
      divisor = a.mat[i][leading_0s[i]];
```

```
      for(j = leading_0s[i]; j < a.cols; ++j)   a.mat[i][j] = a.mat[i][j]/ divisor;
   }
}

//---------------------------------------------------------------------
//          Function to find trank of a matrix
//---------------------------------------------------------------------
int Matrix::rank()
{
   Matrix a = this->copy();
   int i, next_row = 1, grp, p, r, j, *leading_0s, t, m, rank;
   leading_0s = new int[a.rows];

   update_leading_0s(leading_0s, a);
   pivot_rearrange(leading_0s, a);

   if(fabs(a.mat[0][0]) < 0.00001)
   {
      cout << "Not a valid matrix as pivot element is 0" << endl;
      return -1;
   }

   update_leading_0s(leading_0s, a);
   scale_A(leading_0s, a);

   while(next_row == 1)
   {
      grp = 0;
      for( i = 0; i < rows; ++i)
      {
         p = 0;
         while(leading_0s[i+p] == leading_0s[i+p+1] && (i+p+1) < a.rows)
         {
            grp++;
            p++;
         }

         if(grp != 0)
         {
            while(grp != 0)
            {
               for(j = 0; j < a.cols; ++j) a.mat[i+grp][j] = a.mat[i+grp][j] - a.mat[i][j];
               grp--;
            }
            break;
            }
         }

      update_leading_0s(leading_0s, a);
      pivot_rearrange(leading_0s, a);
      update_leading_0s(leading_0s, a);
      scale_A(leading_0s, a);

      next_row=0;

      for(r = 0; r < a.rows ; ++r)
      {
         if(leading_0s[r] == leading_0s[r+1] && r+1 < a.rows)
         {
            if(leading_0s[r] != a.cols) next_row = 1;
         }
      }

   }
   rank = 0;
   for (i = 0; i < a.rows; ++i)
   {
      if (leading_0s[i] != a.cols)  ++rank;
   }
   return rank;
}

//---------------------------------------------------------------------
//          Function to find the determinant
//---------------------------------------------------------------------
double Matrix::determinant()
{

   if(rows != cols) { cout<<" Not a square matrix !!"; return 0;}
```

```
   int j,p,q;
   double det =0;

   if(rows == 2){ return (mat[0][0]*mat[1][1]) - (mat[0][1]*mat[1][0]); }
   Matrix b(rows-1, rows-1);
   for(j = 0; j < cols; j++)
   {
      int r = 0, s = 0;
      for(p = 0; p < rows; p++)
      {
         for(q = 0; q < cols; q++)
         {
            if(p !=0 && q != j)
            {
               b.mat[r][s] = mat[p][q];
               s++;
               if(s > cols-2)
                  {
                     r++;
                     s = 0;
                  }
            }
         }
      }
      det += (j%2 ? -1:1) * mat[0][j] * b.determinant();
   }
   return det;
}

//---------------------------------------------------------------------
//          Function to find the inverse
//---------------------------------------------------------------------
Matrix Matrix::inverse()
{
   double det = this->determinant();

   if( fabs(det) < 0.01)
   {
      cout << "Matrix is not invertible!" << endl;
   }

   int i,j, q, p, sign, r, s;
   double cofdet;
   Matrix inv( rows, cols);
   Matrix cof( rows - 1, cols - 1);

   for(i = 0; i < rows; i++)
   {
      for(j = 0; j < cols; j++)
      {
         sign = ((i+j)%2 ? -1 : 1);
         r = 0, s = 0;
         for(p = 0; p < rows; p++)
         {
            for(q = 0; q < cols; q++)
            {
               if(p != i && q != j)
               {
                  cof.mat[r][s] = mat[p][q];
                  s++;
                  if(s > cols-2)
                     {
                        r++;
                        s = 0;
                     }
               }
            }
         }
         cofdet = cof.determinant();

         inv.mat[i][j] = (fabs(cofdet) < 0.1 ? 0 : sign) * cofdet / det;


      }
   }
   return inv.transpose();
}

//---------------------------------------------------------------------
//          Function to multiply 2 matrices
```

```
//---------------------------------------------------------------------
Matrix Matrix::multiply(Matrix other)
{
   if(cols != other.rows) { cout<< " Invalid dimensions !"; return *this;}

   Matrix product(this-> rows, other.cols);
   int i, j, k;

   for(i = 0; i < this-> rows; i++)
   {
      for(j = 0; j < other.cols; j++)
      {
         for(k = 0; k < this-> cols; k++)
         {
            product.mat[i][j] += this->mat[i][k] * other.mat[k][j];
         }
      }
   }
   return product;
}

//---------------------------------------------------------------------
//          Function to truncate extremely small float values to 0
//---------------------------------------------------------------------
Matrix Matrix::readjust()
{
   for(int i = 0; i < rows; i++)
   {
      for(int j = 0; j<cols; j++)
      {
         if(fabs(mat[i][j]) < 0.00001) mat[i][j] = 0;
      }
   }
   return *this;
}
```

2. matrix.h

```
class Matrix{
public:

   int rows;
   int cols;
   double** mat;

   Matrix(int, int);

   void read_matrix(void);
   void display_matrix(void);
   Matrix copy();
   Matrix transpose();
   int rank(void);
   double determinant();
   Matrix inverse();
   Matrix multiply(Matrix);
   Matrix readjust();
   void scale_A(int *, Matrix);
   void pivot_rearrange(int *, Matrix);
   void update_leading_0s(int *, Matrix);

};
```

3. bfs.cpp

```
#include <iostream>
#include <stdio.h>
#include <math.h>
#include "../Matrix/matrix.h"
#include <limits>

using namespace std;

int count = 0;

//---------------------------------------------------------------------
//          Function to find the factorial of a number
//---------------------------------------------------------------------
```

```
int factorial(int n)
{
		if(n == 0 || n == 1) return 1;
		return n * factorial(n-1);
}


//---------------------------------------------------------------------
//		Function to find al nCr combinations
//---------------------------------------------------------------------
void combination(int *arr, int *data, int start, int end, int index, int r, int **comb)
{
   // Current combination is ready to be printed, print it
   if (index == r)
   {
      for (int j = 0; j < r; j++)
         comb[::count][j] = data[j];
      ::count++;

      return;
   }

   // replace index with all possible elements. The condition
   // "end-i+1 >= r-index" makes sure that including one element
   // at index will make a combination with remaining elements
   // at remaining positions
   for (int i = start; i <= end && end-i+1 >= r-index; i++)
   {
      data[index] = arr[i];
      combination(arr, data, i+1, end, index+1, r, comb);
   }
}

//---------------------------------------------------------------------
//		Function to solve a system of equations of form AX =b
//---------------------------------------------------------------------
Matrix solve(Matrix A, Matrix b, int vars, int eqs, int *zeros)
{
		Matrix a = A.copy();
		int i,j;
		for( i = eqs; i < vars; i++)
			a.mat[i][zeros[i-eqs]-1] = 1;
		if(fabs(a.determinant()) > 0.01)
			return a.inverse().multiply(b);
		else
		{
			Matrix res(vars, 1);
			for( i = 0; i < vars; i++) res.mat[i][0] = -1;
			return res;
		}
}

//---------------------------------------------------------------------
//		Function to find all solutions and then eliminate
//					all non-feasible solutions and to finally obtain
//					Basic Feasible solutions and find optimal solution
//---------------------------------------------------------------------
void find_optimum(Matrix A, Matrix b, int **combinations, int vars, int eqs, int sols, Matrix Z, bool max)
{
		int i, j, flag;

		double min_z = std::numeric_limits<double>::max();
		double max_z = std::numeric_limits<double>::min();
		double z;

		Matrix basic(vars, 1);

		for(i = 0; i < sols; i++)
		{
			flag = 0;

			basic = solve(A, b, vars, eqs, combinations[i]).copy();
			basic.readjust();
			cout << endl << "Solution is " << endl;
			basic.transpose().display_matrix();

			for(j = 0; j < vars; j++)
				if(basic.mat[j][0] < 0) flag = 1;

			if(flag)
```

```
                {
                        cout << "Not a feasible solution" << endl;
                        continue;
                }
                else
                {
                        cout << "Basic feasible solution found" << endl;
                        z = Z.multiply(basic).mat[0][0];
                        if(z > max_z) max_z = z;
                        if(z < min_z) min_z = z;

                        cout << "Value of Z is " << z << endl;
                }


        }
        if(max)
                cout << endl << "Max value of Z is " << max_z << endl;
        else
                cout << endl << "Min value of Z is " << min_z << endl;
}
int main()
{
   int vars, eqs, i, j, flag;
   double max_z, min_z;
   double *optimal_sol;
   cout<<"Enter number of variables"<<endl;
   cin>> vars;
   cout<<"Enter number of equations"<<endl;
   cin>> eqs;

   Matrix A(eqs, vars);
   Matrix Z(1, vars);
   Matrix A_square(vars, vars);
   Matrix b_big(vars, 1);

   cout <<"Enter the data of matrix A"<<endl;
   A.read_matrix();
   cout <<"Matrix A :-"<<endl;
   A.display_matrix();
   flag = 0;

   while(A.rank() != eqs)
   {
        flag = 1;
        cout << "Rank is less than the number of equations" << endl;
        cout << "Check for linear dependency in constraints " << endl;
        eqs = A.rank();
        cout << "You should have " << eqs;
        cout << " independent equations " << endl;

        Matrix A(eqs, vars);
        cout << "Enter matrix A with independent equations" << endl;
        A.read_matrix();
   }

   Matrix b(eqs, 1);
   cout <<"Enter the data of matrix b"<<endl;
   b.read_matrix();

   cout <<"Matrix b :-"<<endl;
   b.display_matrix();

   cout <<"Enter the coefficients of the "<<vars<<" variables in the objective function Z in order"<<endl;
   Z.read_matrix();
   cout<< "Objective function is "<<endl;
   Z.display_matrix();
   cout << "Enter 1 if it is a maximization problem else 0" << endl;
   bool max_or_min;
   cin >> max_or_min;


   for(i = 0; i < eqs; i++)
   {
        for(j = 0; j < vars; j++)       A_square.mat[i][j] = A.mat[i][j];
        b_big.mat[i][0] = b.mat[i][0];
   }

   int rank = A.rank();
   cout << endl << "Beginning Basic Feasible Solution Method . . ." <<endl;
```

```
    cout << "We set " << vars - rank << " variables to be 0 at a time " <<endl;
    optimal_sol = new double[vars];

    int sols = factorial(vars) / (factorial(vars - rank) * factorial(rank));
    cout << "Number of solutions possible is " << sols <<endl;

    int **combinations = new int*[sols];
    for(i = 0; i < sols; i++)
            combinations[i] = new int[vars - rank];

    int *arr = new int[vars];
    for(i = 0; i < vars; i++) arr[i] = i+1;

    int* data = new int[rank];
    combination(arr, data, 0, vars-1, 0, vars - rank, combinations);

    find_optimum(A_square, b_big, combinations, vars, eqs, sols, Z, max_or_min);

    return 0;
}
```

RESULTS -

Problem 2.a (Max Problem)

```
Enter number of variables
5
Enter number of equations
3
Enter the data of matrix A

2 1 -1 0 0
3 4 0 -1 0
1 2 0 0 -1

Matrix A :-
2       1       -1      0       0
3       4       0       -1      0
1       2       0       0       -1

Enter the data of matrix b
20 50 20

Matrix b :-
20
50
20

Enter the coefficients of the 5 variables in the objective function Z in order
4 6 0 0 0

Objective function is
4       6       0       0       0

Enter 1 if it is a maximization problem else 0
1

Beginning Basic Feasible Solution Method . . .
We set 2 variables to be 0 at a time
Number of solutions possible is 10

Solution is
0       0       -20     -50     -20

Not a feasible solution

Solution is
0       20      0       30      20

Basic feasible solution found
Value of Z is 120

Solution is
0       12.5    -7.5    0       5
```

Not a feasible solution

Solution is
0 10 -10 -10 0

Not a feasible solution

Solution is
10 0 0 -20 -10

Not a feasible solution

Solution is
16.6667 0 13.3333 0 -3.33333

Not a feasible solution

Solution is
20 0 20 10 0

Basic feasible solution found
Value of Z is 80

Solution is
6 8 0 0 2

Basic feasible solution found
Value of Z is 72

Solution is
6.66667 6.66667 0 -3.33333 0

Not a feasible solution

Solution is
10 5 5 0 0

Basic feasible solution found
Value of Z is 70

Max value of Z is 120

Problem 3.b (Min Problem)

Enter number of variables
4
Enter number of equations
3

Enter the data of matrix A
1 2 3 1
3 2 2 1
4 4 5 2

Matrix A :-
1 2 3 1
3 2 2 1
4 4 5 2

Rank is less than the number of equations
Check for linear dependency in constraints
You should have 2 independent equations

Enter matrix A with independent equations
1 2 3 1
3 2 2 1

Enter the data of matrix b
90
150

Matrix b :-
90
150

Enter the coefficients of the 4 variables in the objective function Z in order
9 1 1 9

Objective function is
9 1 1 9

Enter 1 if it is a maximization problem else 0
0

Beginning Basic Feasible Solution Method . . .
We set 2 variables to be 0 at a time
Number of solutions possible is 6

Solution is
0 0 -60 270

Not a feasible solution

Solution is
-1 -1 -1 -1

Not a feasible solution

Solution is
0 135 -60 0

Not a feasible solution

Solution is
30 0 0 60

Basic feasible solution found
Value of Z is 810

Solution is
38.5714 0 17.1429 0

Basic feasible solution found
Value of Z is 364.286

Solution is
30 30 0 0

Basic feasible solution found
Value of Z is 300

Min value of Z is 300