Instituto Politécnico Nacional

Escuela Superior de Cómputo

Simplex Dual, Genéticos y Aleatorio

Métodos cuantitativos para la toma de decisiones

Grupos: 3CM8 y 3CV8

Estudiantes:
Naranjo Ferrara Guillermo
Nicolás Sayago Abigail
Parra Garcilazo Cinthya Dolores
Sánchez Valencia Sergio Gabriel

Profesor:
Ariel Lopez Rojas

09 de Abril 2019

Contents

1 Código

1.1 Simplex Dual

```
INF = 1 << 30;
epsilon = 0.00001

function printProcess(restrictions, zj, ssol) {
        console.log('---------------')
        console.log('Matrix:');
        console.log(restrictions);
        console.log('zj:', zj);
        console.log('ssol:', ssol);
        console.log('---------------')
}

function makeZeroIfCloseToZero(n) {
        return Math.abs(n) < epsilon ? 0 : n;
}

function getKeyColumn(keyRow, zj, cj) {
        let minRatio = INF;
        let column = -1;
        for (let i = 1; i < keyRow.length; i++) {
                if (keyRow[i] < 0) {
                        ratio = (zj[i] - cj[i]) / keyRow[i];
                        if (ratio < minRatio)
                                minRatio = ratio, column = i;
                }
        }
        return column;
}

function getKeyRow(mat) {
        let mostNegative = INF;
        let row = -1;
        for (let i = 1; i < mat.length; i++)
                if (mat[i][0] < mostNegative)
                        mostNegative = mat[i][0], row = i;
        return row;
}

function dualSimplex(mat, zj, ssol) {
        keyRow = getKeyRow(mat);
        console.log('keyRow', keyRow)
        keyCol = getKeyColumn(mat[keyRow], zj, mat[0]);
        console.log('keyCol', keyCol)
        if (keyCol == -1) {
                console.log('not feasible')
                return false;
        }
        keyPivot = mat[keyRow][keyCol];
        console.log('keyPivot', keyPivot)
        ssol[keyRow - 1] = mat[0][keyCol]; // cj.size * 2 = ssol.size Check
         ↪  exception

        for (let j = 0; j < mat[keyRow].length; j++) {
                mat[keyRow][j] /= keyPivot;
                zj[j] = ssol[keyRow - 1] * mat[keyRow][j];
        }

        let isFinalIteration = true;
        for (let i = 1; i < mat.length; i++) {
                if (i != keyRow) {
                        let keyColValInRowI = mat[i][keyCol];
                        for (let j = 0; j < mat[i].length; j++) {
                                mat[i][j] = makeZeroIfCloseToZero(mat[i][j]
                                 ↪  - mat[keyRow][j] * keyColValInRowI);
                                zj[j] += ssol[i - 1] * mat[i][j];
                        }
                }
                if (mat[i][0] < 0)
                        isFinalIteration = false;
        }

        printProcess(mat, zj, ssol);

        if (!isFinalIteration)
                return dualSimplex(mat, zj, ssol)
        let ans = 0;
        let limit = ((mat[0].length - 1) >> 1) + 1;
        for (let i = 1; i < limit; i++) {
                ans += mat[0][i] * mat[i][0];
        }
        console.log('Ans', ans);
        return ans;
}
```

```
function modifiedTranspose(mat) {
        let transposed = Array.from({
                length: mat[0].length
        }, () => (Array(mat.length).fill(0)));
        for (let i = 0; i < mat.length; i++)
                for (let j = 0; j < mat[i].length; j++)
                        transposed[j][i] = j ? -1 * mat[i][j] : mat[i][j];
        return transposed;
}

function fixInput(data) {
        console.log(data);
        for (const restriction of data.restrictions)
                if (restriction.pop() === 1)
                        for (let j = 0; j < restriction.length; j++)
                                restriction[j] = -1 * restriction[j];

        if (data.isMaximitation) data.restrictions =
         ↪  modifiedTranspose(data.restrictions);

        data.zj = Array(data.restrictions[0].length - 1).fill(0);
        data.ssol = Array(data.restrictions.length - 1).fill(0);

        for (let i = 1; i < data.restrictions.length; i++) {
                for (let j = 1; j < data.restrictions.length; j++)
                        data.restrictions[i].push(i ^ j ? 0 : 1);
                data.restrictions[0].push(0);
                data.zj.push(0);
        }

        console.log(data);
}

function Input() {
        this.restrictions = [
                [0, 1, 2, 3, 0], // Cj
                [-4, -2, 1, -1, -1], // R0
                [8, 1, 1, 2, -1], // R1
                [-2, 0, -1, 1, -1] // R2
        ];

        this.isMaximitation = false;

        this.restrictions = [
                [0, 5, 4, 5, 0],
                [350, 6, 2, 3, -1],
                [150, 5, 3, 0, -1],
                [20, 0, 0, 1, 1]
        ];

        this.restrictions = [
                [0, 0.2, 0.5, 0],
                [2000, 0.1, 0.6, -1],
                [6000, 1, 1, -1],
                [4000, 1, 0, -1]
        ];

        this.isMaximitation = true;

        this.zj = Array(this.restrictions[0].length - 1).fill(0);
        this.ssol = Array(this.restrictions.length - 1).fill(0); // slack
         ↪  variables solution
};
let data = new Input();
fixInput(data);
console.log(data)
dualSimplex(data.restrictions, data.zj, data.ssol)
```

1.2 Genético

```
// Genetic Algorithm

/* INPUT
 * Function Z
 * n restrictions
 * bits of precision
 *
 * */

/************************************/
/**   GENERATE VECTORS      **/
/**********************************/

// Calculate limits based in the restrictions (aj, bj) OK

// Calculate Mj OK
// Generate vectors OK

// Check that Xj and Yj satisfy the restrictions

// Verify that the time to generate valid vectors is less than 2 minutes

// Calculate the table for each round (table)


/*************************************/
/**   TABLE FOR EACH ROUND     **/
/***********************************/

// Calculate Z for each solution vector

// Calculate %Z with the n variables

// Calculate the commulative sum of %Z

// Choose a random number between [0, 1]


/*************************************************/
/**   GENERATE NEW VECTORS TO NEXT ROUND     **/
/***********************************************/

// FOR EACH ROUND:

//   First we try to find a solution vector that repeats the most

//   If we find it then we apply mutation over him to pass it to the next
 ↪  iteration

//   Else if there are multiple options

//   we apply cross with the ones that repeated the most, if there are more
 ↪  than two options we apply a random to select two of them

//   Else we just apply random two select two solution vectors

/***********************************************/
/**         FINAL RESULT       **/
/*********************************************/
```

```
// It happens when the number of rounds is over

// We choose the best Z depending if it is Max or Min

const TIME_LIMIT = 12000;

function printRound(round) {
  for (let i = 1; i < round.length; i++) {
    let row = "";
    for (let j = 1; j < round[i].length; j++) {
      row += '\t' + round[i][j].toFixed(8);
    }
    console.log(row, '\n');
  }
}

/*****************************************/
/*                PRINTS                */
/***************************************/
// 0 <= x <= 3
// 2 <= y <= 5
// gets transformed into:

//  2x + 5y
// x >= 0
// x <= 3
// y >= 0
// y >= 2
// y <= 5
function Input() {
  this.restrictions = [
    [0, 1, 1, -2, 1, 0],
    [50, 1, 0, 1, 0, -1], // Cj
    [75, 0, 1, 0, 1, -1], // R0
    [10, 1, 0, 0, 0, 1], // R1
    [100, 0, 1, 0, 1, -1],
    [30, 0, 0, 2, 1, 1],
    [0, 1, 0, 0, 0, 1],
    [0, 0, 1, 0, 0, 1],
    [0, 0, 0, 1, 0, 1],
    [0, 0, 0, 0, 1, 1]
  ];
  // this.restrictions = [
  //   [0, 1, 1, 0],
  //   [20, 2, 1, -1],
  //   [10, 1, 1, 1],
  //   [0, 1, 0, 1],
  //   [0, 0, 1, 1]
  // ];
  this.isMaximization = false;
  this.variablesCount = 4;
  this.restrictionsCount = 3;
  this.iterationsCount = 10;
  this.populationSize = 50;
  this.precisionBits = 1;
}

let INF = 1 << 30;

function myRandom(min, max) {
  return Math.floor(Math.random() * (max - min)) + min;
}

function getLimits(restrictions) {
  let minMax = []; // limits[i][0] = lower, limits[i][1] = upper
  for (let i = 1; i < restrictions[0].length - 1; i++)
    minMax[i] = [INF, -INF];
  for (let i = 1; i < restrictions.length; i++) {
    for (let j = 1; j < restrictions[i].length - 1; j++) {
      if (restrictions[i][j] !== 0) {
        let aux = restrictions[i][0] / restrictions[i][j];
        if (aux > minMax[j][1])
          minMax[j][1] = aux;
        else if (aux < minMax[j][0])
          minMax[j][0] = aux;
      }
    }
  }
  return minMax;
}

function calculateConst(lowerLimit, upperLimit, mj) {
  return (upperLimit - lowerLimit) / ((1 << mj) - 1);
}

function calculateXj(lowerLimit, subVector, constant) {
  return lowerLimit + (subVector * constant);
}
```

```
function verifyVector(restrictions, xj) {
  let  flag = 1;
  for (let i = 1; i < restrictions.length; i++) {
    let sum = 0; // Result of replace the value for a restriction
    for (let j = 1; j < restrictions[i].length - 1; j++)
      sum += restrictions[i][j] * xj[j];

    if(restrictions[i][restrictions[i].length - 1] === 1) {  // We have an
     ↪  >=
      if(sum < restrictions[i][0])
        flag = 0;
    } else if(sum > restrictions[i][0])
        flag = 0;
  }
  return flag;
}

function genXjvector(limits, randoms, constants) {
  let xj = [];
  for (let i = 1; i < randoms.length; i++) {
    xj[i] = calculateXj(limits[i][0], randoms[i], constants[i]);
  }
  return xj;
}
//  ath.floor(Math.random() * 11);      // returns a random integer from 0
 ↪  to 10
function generateVector(restrictions, mj, limits, constants) {
  let varCount = restrictions[0].length - 2;
  let randoms = [];
  let xj = [];
  let i, j;
  let counter = 0;
  // CONSTANTS is an array that contain (bj - aj) / (x^{mj} - 1)
  // We generate a constants for each variable

  while (true) {
    if (counter === TIME_LIMIT) {
      console.log("Vector not found");
      break;
    }

    // We generate a random value that have a length of mj[i] for each
     ↪  variable
    // Is like a Vector_i
    for (let i = 1; i < varCount + 1; i++)
      randoms[i] = myRandom(0, 1 << mj[i]);

    // console.log(randoms);
    // We calculate the Xj value for each variable that is in random[i]
    xj = genXjvector(limits, randoms, constants);
    if (flag = verifyVector(restrictions, xj)) {
      break;
    }
    counter++;
  }
  return [flag, randoms, xj];
}

function findClosest(acum, i) {
  // if (i <= acum[0]) return 0;
  // if (i >= acum[acum.length - 1]) return acum.length - 1;
  // let l = 1, r = acum.length - 1;
  // while (l <= r) {
  //   let mid = (l + r) >> 1;
  //   if (i < acum[mid]) r = mid - 1;
  //   else if (i > acum[mid]) l = mid + 1;
  //   else return mid;
  // }
  // return (acum[l] - i) < (i - acum[r]) ? l : r;

  let closest = -1;
  let closestDist = INF;
  for (let j = 1; j < acum.length; j++)
    if (Math.abs(acum[j] - i) < closestDist) {
      closest = j;
      closestDist = Math.abs(acum[j] - i);
    }

  return closest;
}

function nlength(n) {
  let length = 0;
  while (n) {
    n >>= 1;
    length++;
  }
  return length;
}

function mutate(n) {
  let m = arrayCopy(n);
  // 011 101 10
  // [3, 5, 2]
  let random1 = myRandom(1, n.length)
  let random = myRandom(1, nlength(n[random1]));
  m[random1] ^= (1 << random);
  return m;
}

function cross(a, b) {
  // 011 101 10    011 101 10
  // [3, 5, 2]     [3, 5, 2]
  let aux = arrayCopy(a);
  let random = myRandom(1, a.length);
  aux[random] = b[b.length - random];
  return aux;
}

function getStrongest(newVectors) {
  let strongest = -1;
  let mostCommon = 0;
  let helper = {};
  for (let i = 1; i < newVectors.length; i++)
    helper[newVectors[i]] = 0;
  for (let i = 1; i < newVectors.length; i++) {
    if (++helper[newVectors[i]] > mostCommon) {
      mostCommon = helper[newVectors[i]];
      strongest = newVectors[i];
    }
  }
  return [strongest, mostCommon];
}

function calculateRound(vectors, restrictions) {
  let round = [];
  for(let i = 1; i < vectors.length; i++)
    round[i] = [];

  for(let i = 1; i < vectors.length; i++) {
    for(let j = 1; j < vectors[i].length; j++)
```

```
      round[i][j] = vectors[i][j];
  }

  let sum = 0;
  // Calculate Z
  for (let i = 1; i < vectors.length; i++) {
      let z = 0; // Result of replace the value for a restriction
      for (let j = 1; j < vectors[i].length; j++)
        z += restrictions[0][j] * vectors[i][j];
      round[i][round[i].length] = z;
    sum += z;
  }
  // Calculate %Z and Zacum
  let Zacum = 0;
  let acumColumn = [];  // Get %Z acum column
  for (let i = 1; i < vectors.length; i++) {
      let z = 0; // Result of replace the value for a restriction
      z = (round[i][round[i].length - 1] / sum) * 100;
      round[i][round[i].length] = z;
      Zacum += z;
      round[i][round[i].length] = Zacum;
      acumColumn[i] = Zacum;
  }
  // It always have to be 100 because is the max
  round[vectors.length - 1][round[vectors.length - 1].length - 1] = 100;
  acumColumn[acumColumn.length - 1] = 100;

  // Calculate #Random[0 1]
  for (let i = 1; i < vectors.length; i++) {
      let random = 0; // Result of replace the value for a restriction
      random = Math.floor(Math.random() * 11) * 10;
      round[i][round[i].length] = random;
  }
  for(let i = 1; i < vectors.length; i++)
    round[i][round[i].length] = findClosest(acumColumn,
     ↪  round[i][round[i].length - 1]);
  return round;
}

function matrixCopy(matrix) {
  let copy = [];
  for(let i = 0; i < matrix.length; i++)
    copy[i] = arrayCopy(matrix[i]);
  return copy;
```

```
}
function arrayCopy(array) {
  let copy = [];
  for(let i = 0; i < array.length; i++)
    copy[i] = array [i];
  return copy;
}

function getColumn(matrix, index) {
  let col = [];
  for (let i = 1; i < matrix.length; i++)
    col[i] = matrix[i][index];
  return col;
}
function bestZ(column, isMaximization) {
  let index = -1;
  let best = isMaximization ? -INF : INF;

  for(let i = 1; i < column.length; i++) {
    if(isMaximization) {
      if(column[i] > best) {
        best = column[i];
        index = i;
      }
    }
    else {
      if(column[i] < best) {
        best = column[i];
        index = i;
      }
    }
  }
  return index;
}

function geneticAlgorithm(input) {
  let limits = getLimits(input.restrictions);
  console.log(limits);
  let mj = [];
  // MJ is an array that contain the number of bits for each variable
  // We generate MJ with the precision of Bits that the user says
  for (let i = 1; i < limits.length; i++)
    mj[i] = Math.ceil(Math.log2((limits[i][1] - limits[i][0]) * Math.pow(10,
     ↪ input.precisionBits)));
```

```
let varCount = input.restrictions[0].length - 2;
let constants = [];
for(i = 1; i <= varCount; i++)
  constants[i] = calculateConst(limits[i][0], limits[i][1], mj[i])
let vectors = [];
let randoms = [];
for(let i = 1; i <= input.populationSize; i++) {
  let v = generateVector(input.restrictions, mj, limits, constants);
  console.log('vvvvvvvvvv', v);
  if(v[0] === 1) {
    // We have to save valid vector
    vectors[i] = v[2];
    randoms[i] = v[1];
  } else {
    console.log("We don't have vectors that satisfy the conditions");
    return;
  }
}
// CALCULATE ROUNDS
let rounds = [];
let zs = [];
for (let i = 1; i <= input.iterationsCount; i++) {
  // console.log(vectors)
  rounds[i] = calculateRound(arrayCopy(vectors),
   ↪  matrixCopy(input.restrictions));
  let newVectors = [];

  let strongest = getStrongest(getColumn(rounds[i], rounds[i][1].length -
   ↪  1));
  console.log(strongest);
  let z = bestZ(getColumn(rounds[i], rounds[i][1].length - 5),
   ↪  input.isMaximization);
  // console.log(z);
  zs[i] = rounds[i][z][rounds[i][z].length - 5]; // AUIIIIIIIIIIIIIIIII NO
   ↪  MOVI NADAAAAAAAAAAAAAAAAA ATTT SERGIOOOOOOOOOOOOOO
  if(strongest[1] > 1) {
    newVectors[1] = arrayCopy(vectors[strongest[0]]);
    // console.log('preveNewVector', newVectors);
    for(let j = 1; j < vectors.length; j++) {
      let counter = 0;
      while (true) {
        if (counter === TIME_LIMIT) {
          console.log("Vector not found for method 1");
```

```
            return;
          }
          let mutation = mutate(randoms[strongest[0]]);
          let crossed = cross(mutation, randoms[j]);
          newVectors[j] = genXjvector(limits, crossed, constants);
          if (verifyVector(input.restrictions, newVectors[j]))
            break;
          counter++;
        }
      }
    }
    else {
      for (let j = 1; j < vectors.length; j++) {
        let counter = 0;
        while (true) {
          if (counter === TIME_LIMIT) {
            console.log("Vector not found for method 2");
            return;
          }
          newVectors[j] = genXjvector(limits, cross(randoms[strongest[0]],
           ↪  randoms[z]), constants);
          if (verifyVector(input.restrictions, newVectors[j]))
            break;
          counter++;
        }
      }
    }
    // console.log('newVectors', newVectors);
    vectors = newVectors;
    console.log("ROUND:" , i);
    console.log("");
    printRound(rounds[i]);
  }
  let best = bestZ(zs, input.isMaximization);

  console.log("ANSWER:", zs[best]);
  console.log("");
}
let data = new Input();
geneticAlgorithm(data);
```

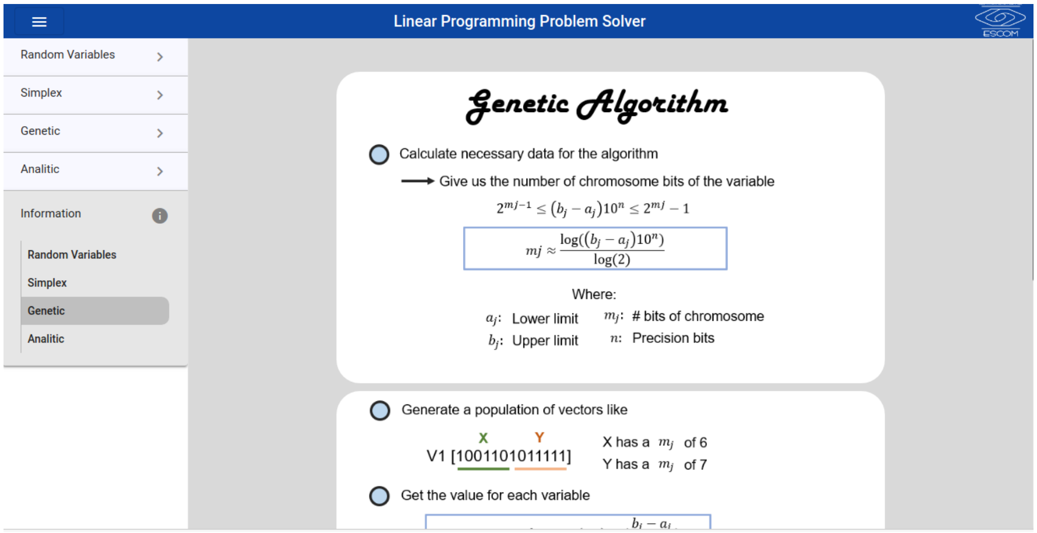
1.3 Aleatorio

```
INF = 1 << 30;
function arrayValores(restricciones, ef) {
    var i = 0;
    this.valores = [];
    for(i; i < restricciones.length; i++)
        this.valores[i] =
         ↪  numAleatorio(restricciones[i][0],restricciones[i][1], ef);
    return this.valores;
}

function numAleatorio(min, max, ef) {

    this.random = Math.random()*(max - min ) + min;
    return (ef == 0) ? Math.floor(this.random): this.random;
}

function minimo(mat) {
        if (mat === undefined)
            return -INF;
        return mat[0];
}

function maximo(mat) {
        if (mat === undefined)
                return INF;
        return mat[mat.length - 1];
}

function cumpleCondiciones(mat, valores){
    var i = 0;
    this.resultado = 1;
    for(i = 1; i < mat.length; i++){
        this.resultado *= cumpleCondicion(mat[i], valores);
    }
    return this.resultado;
}

function cumpleCondicion(condicion, valores) {
    var i = 0;
    this.res = 0;
    this.restriccion = condicion[condicion.length - 1];
    this.valor = condicion[0];
    for (i = 1; i < condicion.length - 1; i ++) {
        this.res += condicion[i]*this.valores[i-1];
    }
    switch(this.restriccion){
        case -1:
            return (this.res <= this.valor)? 1 : 0;
        case 1:
            return (this.res >= this.valor)? 1 : 0;
        case 0:
            return (this.res == this.valor) ? 0 : 0;
        default:
            return 0;
    }
}

function poblacion(matriz, nPoblacion, rango, ef){
    var i = 0;
    this.arrayPoblacion = [];
    for(i = 0; i < nPoblacion; i++){
        this.valores = arrayValores(rango, ef);
        if(cumpleCondiciones(matriz, valores) == 1){
            this.arrayPoblacion.push(getZ(matriz[0], valores));
        }
    }
    if(this.arrayPoblacion.length !== 0)
        return this.arrayPoblacion.sort(function(a,b){return a[0] - b[0]});
    else
        return "E";
}

function getZ(z, valores) {
    var i = 0;
    this.zRes = []
    this.res = 0;
    for(i = 1; i < z.length - 1; i++){
        this.res += z[i] * valores[i - 1];
        this.zRes[i] = valores[i - 1];
    }
    this.zRes[0] = this.res;
    this.zRes[z.length - 1] = 0;
    return this.zRes;
}
```

```
function metodoAleatorio(matriz, nMuestra, nPoblacion, rango, ef) {
    var i = 0;
    this.muestraMin = [];
    this.muestraMax = [];
    this.muestra = [];
    this.metodoAl = [];
    this.conf ='';
    for (i = 0; i < nMuestra; i++){
        this.conf = poblacion(matriz, nPoblacion, rango, ef);
        if (this.conf !== "E")
            this.muestra.push(conf);
    }
    if (!muestra.length) {
        console.log("No se logro obtener muestras con poblaciones validas")
        return false;
    }

    for(i = 0; i < muestra.length; i++){
        this.muestraMin[i] = minimo(muestra[i]);
        this.muestraMax[i] = maximo(muestra[i]);
    }
    this.muestraMin.sort(function(a,b){return a[0]-b[0]});
    this.muestraMax.sort(function(a,b){return a[0]-b[0]});
    this.muestraMin = minimo(muestraMin);
    this.muestraMax = maximo(muestraMax);

    this.metodoAl[0] = this.muestraMin;
    this.metodoAl[1] = this.muestraMax;

    for(i = 0; i < muestra.length; i++)
        metodoAl.push(muestra[i]);

    return this.metodoAl;
}

const mat = [
                [0, 1, 2, 3, 0],
                [-4, -2, 1, -1, -1],
                [8, 1, 1, 2, -1],
                [-2, 0, -1, 1, -1],
                [0, 1, 0, 0, 1],
                [0, 0, 1, 0, 1],
                [0, 0, 0, 1, 1]
            ];
```

```
const val = [[0, 8],[0, 8], [-2, 4]];

console.log(metodoAleatorio(mat,5,1000,val,1));
```

2 Capturas de pantalla



Genetic Algorithm

○ Calculate necessary data for the algorithm

⟶ Give us the number of chromosome bits of the variable

$$2^{m_j-1} \leq (b_j - a_j)10^n \leq 2^{m_j} - 1$$

$$\boxed{mj \approx \frac{\log((b_j - a_j)10^n)}{\log(2)}}$$

Where:

a_j: Lower limit m_j: # bits of chromosome

b_j: Upper limit n: Precision bits

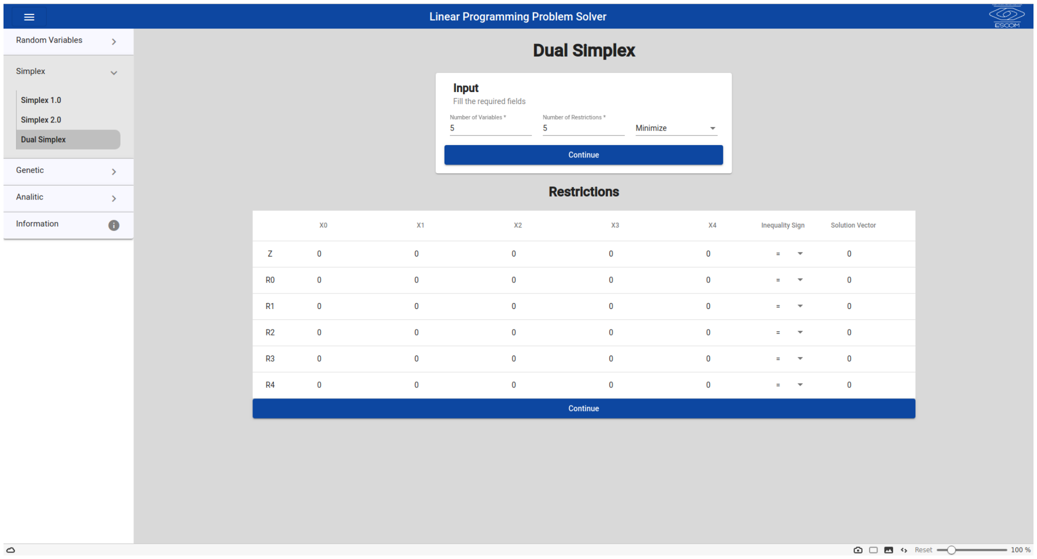
○ Generate a population of vectors like

V1 [10011010111111] X has a m_j of 6

Y has a m_j of 7

○ Get the value for each variable

$b_i - a_i$



Dual Simplex

Input

Fill the required fields

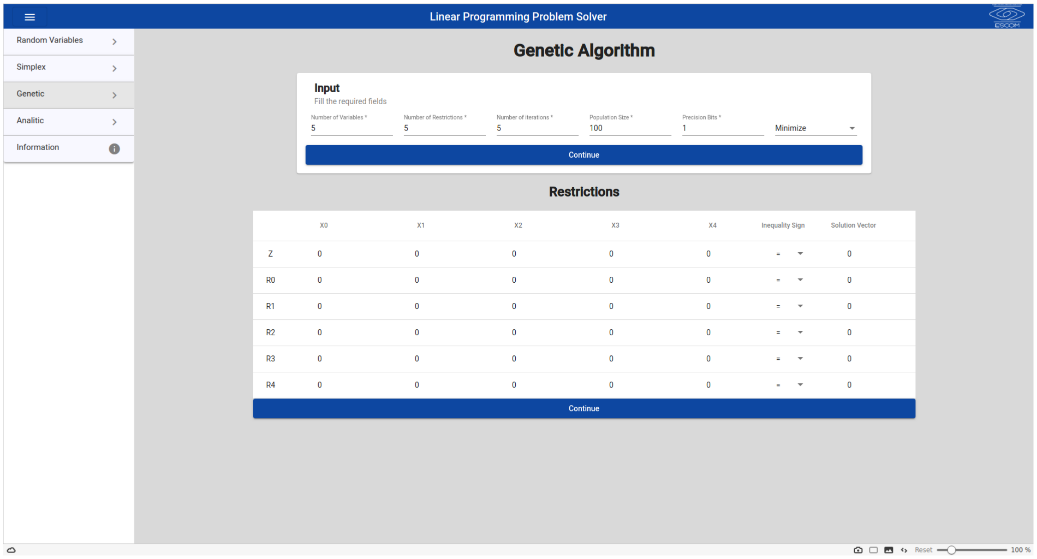
Number of Variables * 5

Number of Restrictions * 5

Minimize

Restrictions

	X0	X1	X2	X3	X4	Inequality Sign	Solution Vector
Z	0	0	0	0	0	=	0
R0	0	0	0	0	0	=	0
R1	0	0	0	0	0	=	0
R2	0	0	0	0	0	=	0
R3	0	0	0	0	0	=	0
R4	0	0	0	0	0	=	0



Genetic Algorithm

Input

Fill the required fields

Number of Variables * 5

Number of Restrictions * 5

Number of iterations * 5

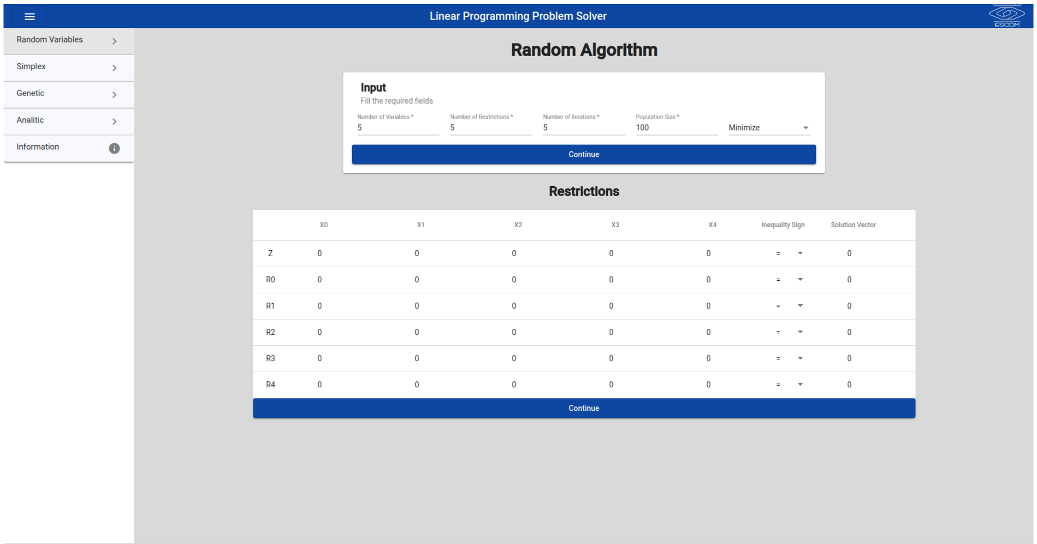
Population Size * 100

Precision Bits * 1

Minimize

Restrictions

	X0	X1	X2	X3	X4	Inequality Sign	Solution Vector
Z	0	0	0	0	0	=	0
R0	0	0	0	0	0	=	0
R1	0	0	0	0	0	=	0
R2	0	0	0	0	0	=	0
R3	0	0	0	0	0	=	0
R4	0	0	0	0	0	=	0



Random Algorithm

Input

Fill the required fields

Number of Variables * 5

Number of Restrictions * 5

Number of iterations * 5

Population Size * 100

Minimize

Restrictions

	X0	X1	X2	X3	X4	Inequality Sign	Solution Vector
Z	0	0	0	0	0	=	0
R0	0	0	0	0	0	=	0
R1	0	0	0	0	0	=	0
R2	0	0	0	0	0	=	0
R3	0	0	0	0	0	=	0
R4	0	0	0	0	0	=	0