INSTITUTO TECNOLÓGICO AUTÓNOMO DE MÉXICO

Modelado y Optimización I

Proyecto 1:
Algoritmo Simplex

10 de junio de 2022.

Prof. Luis Antonio Moncayo Martínez

Leonela Simonne Arcos Sotelo 170875

María del Carmen Pliego San Martín 189708

Mauricio Verduzco Chavira 195106

Índice

1. Introducción

a. Algoritmo simplex

La programación lineal es una rama de la investigación de operaciones, que a su vez es una rama de disciplinas como Ingeniería Industrial y Matemáticas, la cual se enfoca en la aplicación de métodos de análisis para tomar mejores decisiones, por ejemplo, optimización de sistemas.

El método simplex es una forma de resolver problemas de optimización por medio de programación lineal usando elementos adicionales. Es decir, un modelo de programación lineal es un método de conseguir el mejor resultado posible dada una ecuación de maximización o minimización con restricciones lineales. La mayoría de los problemas lineales se pueden resolver con aplicaciones como MatLab o Lingo, pero el método Simplex es un algoritmo para calcularlo a mano, que es en realidad lo que está detrás de muchos de los programas mencionados.

Para resolver un modelo de programación lineal usando el algoritmo Simplex se necesita seguir el siguiente método:

- Trabajar con una forma estándar.
- Agregar variables de holgura (cuando la restricción es mayor o igual) y/o de exceso (si la restricción es menor o igual).
- Crear una matriz compuesta por los elementos necesarios para ejecutar las operaciones que hacen funcional al algoritmo.
- Variables pivote.
- Hacer los cambios correspondientes.
- Checar la optimalidad.
- Identificar los valores óptimos.

Todo esto lo tomamos en cuenta para hacer nuestro programa. En este reporte se explica a detalle cómo fue la implementación del modelo Simplex. En la siguiente sección se explica concretamente lo que se debe hacer en este método.

b. Estructura del algoritmo simplex

Para que funcione el modelo se debe comenzar con un problema de programación lineal a optimizar que contenga una función objetivo de maximización o minimización y sus restricciones.

Una vez identificado el problema se pueden determinar las variables de holgura y exceso. El número de variables que se agreguen dependerá del número de restricciones en el problema. Si la restricción es <= se usan variables de holgura que se designan con una “s”. Si la restricción es >= se usan variables de exceso y se designan con “e”. Con los datos que se tienen se genera:

- c: el vector de coeficientes de la función objetivo, incluyendo las variables agregadas.
- b: el vector de recursos que son los valores restrictivos.
- A: matriz de tecnología o de coeficientes de las restricciones con variables agregadas.
- xB: Es el conjunto básico que comienza con los valores de las variables agregadas y se va modificando con cada iteración.
- B: matriz con coeficientes de las restricciones de las variables en la base.
- Bin: B inversa.
- cB: vector con los coeficientes de las variables de la base en la función objetivo.

Con estos datos se puede crear la matriz con la que se trabaja en el algoritmo:

	Nombres de variables	zSol
	zRow = cB*(Binv)*A - c	z* = cB*Binv*b
xB	cij = Binv*A	bj = Binv*b

Una vez que se tiene la matriz, se deben realizar las operaciones necesarias para determinar qué variable entra a la base por medio de la condición de optimalidad y cuál sale con la condición de fisibilidad. Así se realizan varias iteraciones hasta que se llega a la solución óptima. Para saber si ya se llegó, se debe analizar según el caso.

La condición de optimalidad es que la variable que entra, en el caso de maximización, es la variable no básica con el valor más negativo en los coeficientes de zRow. Si todos los valores de zRow son no negativos, se llegó al óptimo.

En el caso de minimización entra la variable más positiva y se llega al óptimo cuando todos los coeficientes de zRow son no positivos.

La condición de factibilidad es igual para maximización y minimización. Es cuando sale la variable básica que obtiene el valor más chico no negativo de la operación bj/cij dado que xi es la variable que entra.

En ambos casos se rompen los empates arbitrariamente.

A continuación se explica a detalle el uso de las condiciones en el caso de primal y en el dual.

- Simplex primal:

Debe cumplir dos condiciones al inicio:

- ❖ Empieza con una solución factible (los valores de bj son positivos).
- ❖ No satisface la condición de optimalidad.

Después se hacen una serie de iteraciones conservando la factibilidad hasta encontrar el óptimo. Para saber qué variable entra cumpliendo con la condición de optimalidad y cuál sale según la de factibilidad, se siguen las siguientes indicaciones:

Función objetivo	**Variable que entra**	**Variable que sale**	**Termina el algoritmo**
Max	La más negativa en zRow.	El valor mínimo en la división del valor de cada variable en bj entre su respectivo valor en la columna de la variable que entra.	Cuando todos los valores de zRow son positivos o cero.
Min	La más positiva en zRow.		Cuando todos los valores de zRow son negativos o cero.

- Simplex dual:

Debe cumplir dos condiciones al inicio:

- ❖ Que comience con una solución no factible (los valores de bj son negativos).
- ❖ Cumple con la condición de optimalidad.

Para saber qué variable entra de acuerdo con la condición de factibilidad y cuál entra según la condición de optimalidad, se siguen las siguientes indicaciones:

Función objetivo	**Variable que sale**	**Variable que entra**	**Termina el algoritmo**
Max o min	La más negativa de bj en la columna de solución. Si no hay	El valor más chico del resultado de la división del valor de	Cuando todos los valores de bj son positivos.

	negativos y no he llegado al óptimo, se pasa al primal.	la variable en zrow entre su valor en la fila de la variable que sale. Solo se dividen los que tienen valor negativo en cij.	

Al terminar las iteraciones, el valor que en la matriz se indicó como z* es el óptimo, los valores de la columna Sol. son los óptimos de cada variable de decisión y zRow son los precios duales. Si una de las variables no está en la base que quedó dentro de la solución, su valor es cero.

2. Documentación

a. Estructura de la aplicación

A nosotros se nos facilitó más trabajar con varias funciones y al final las juntamos todas en una función principal que ejecuta el problema y arroja la solución. A continuación, se explica cada una de las funciones programadas:

- <u>Función auxiliar que maneja el rango de error computacional:</u>

```
simplificarVectoraCeros <- function(input){
  for (i in 1:length(input)){
    if(abs(input[i])<=0.0001){
      input[i]<- 0
    }
  }
  input


}

simplificarMatrizaCeros <- function(input){
  for (i in 1:nrow(input)){
    for (j in 1:ncol(input)){
      if(abs(input[i,j]<=0.0001)){
        input[i,j]<-0
      }
    }
  }
  input
}
```

Lo que hace la primera función (*simplificarVectoraCero*) es analizar los valores del vector de zRow para ver si son menores a 0.0001 y, si lo son, convertirlo en cero.

Esto lo hicimos porque notamos que en ciertos casos el programa no llegaba a la solución óptima correcta porque al preguntar si todos eran positivos, negativos o cero, dependiendo si el caso era de maximización o minimización, los valores muy pequeños los seguía detectando como si no se cumpliera la condición y continuaba iterando. Con esta función se soluciona el problema.

La segunda (*simplificaMatrizaCeros*) es un caso similar. Como en el programa en un punto es necesario dividir los valores para decidir las variables que salen y las que entran, en algunos casos arrojaba un error porque trataba de dividir entre cero, pero no detectaba a ese valor como cero. Para solucionar eso, tuvimos que hacer esta función que convierte los valores menores a 0.0001 de la matriz, en ceros.

- <u>Función que genera la primera base con los datos iniciales:</u>

```
procesar_datos_iniciales <- function(b, c){

  xB <- names(c[(length(c)-length(b)+1):length(c)])
  xB

}
```

Esta es la función que genera el vector xB inicial, es decir, el vector base que contiene los nombres de las variables de holgura y exceso que se obtienen cuando se ingresan los datos por el usuario.

Lo que recibe es el vector "c", que es el vector de la función objetivo dado por el usuario. También recibe el vector "b", que es el de recursos dado también por el usuario, el cual indica cuántas variables de holgura y/o exceso se tienen en el problema. Con estas dos entradas, se genera el vector "xB", que consiste de los nombres de las variables del vector "b".

- <u>Función que genera la segunda sección con la nueva base dada</u>:

```
generar_vectores_NBase <- function(nuevaBase, A, c){

  xB <- nuevaBase
  B <- matrix(A[, (nuevaBase)], nrow = length(nuevaBase), ncol = length(nuevaBase), byrow = FALSE)
  B_inverse <- solve(B)
  cb <- c[nuevaBase]
  names(cb) <- nuevaBase
  res <- list(xB, B, B_inverse, cb)
  res

}
```

Lo que se hace aquí es generar el nuevo vector xB cuando ya se hicieron los cálculos para saber cuál variable entra y cuál sale.

La función recibe como parámetros “nuevaBase”, que es el vector con los nombres de las variables que componen la nueva base; “A” que es la matriz de los coeficientes de las restricciones; “c” que es el vector de la función objetivo.

Con estos parámetros se genera la nueva matriz “B” que contiene los coeficientes de las restricciones de la nueva base y así se obtiene su inversa. También se genera el nuevo vector “cB” que es el de los coeficientes de las variables de la nueva base en la función objetivo.

Como respuesta arroja una lista que contiene el nuevo vector “xB”, la matriz “B”, “B inversa” y el nuevo vector “cB”.

- <u>Función que evalúa la sección tres generada a partir de los datos de la sección dos y su respectiva base:</u>

```
evaluarBase <- function(cB, B, A, B_inverse, b, c){

  zrow <- cB%*%B_inverse%*%A - c
  ci <- B_inverse%*%A
  zsol <- cB%*%B_inverse%*%b
  bj <- B_inverse%*%b
  res3 <- list(zrow, ci, zsol, bj)
  res3

}
```

Esta función lo que hace es analizar los datos que se tienen para generar los componentes de la matriz que se utiliza para hacer el análisis simplex.

Recibe como parámetros el vector “cB”, la matriz “B” y su inversa, la matriz “A”, el vector “b” y el vector “c”. Estos parámetros se obtienen de lo que ingresa el usuario y de lo que generó en otras funciones que se explicaron anteriormente.

Con estos datos se genera la fila “zRow”, la matriz de valores “cij”, el valor “zsol” y la columna “bj”, que son todos los componentes de la matriz con la que se trabaja. La respuesta que arroja es una lista con estos resultados.

- <u>Función auxiliar que nos dice nos pregunta si llegamos al óptimo (MAXIMIZACIÓN):</u>

```
esOptimo_MAX <- function(bj, zrow){
  if(todosPositivosOCero(bj) && todosPositivosOCero(zrow))
    res<-TRUE
  else
    res<-FALSE
  res
}
```

  - Usamos *TodosPositivosOCero*

```
todosPositivosOCero <- function(vectorInput){
  flag = TRUE
  for (i in 1:(length(vectorInput))){
    if(vectorInput[i]<0){
      flag = FALSE
    }
  }
  flag

}
```

La función *todosPositivosOCero* ve si todos los valores de un vector son mayores o iguales a cero. Se inicia un flag en TRUE. Recibe como parámetro un vector y utiliza un for para recorrerlo y ver si sus valores cumplen con la condición. Si no es así, el flag cambia a FALSE. El valor booleano del flag es lo que arroja como respuesta.

Esta función se usa como auxiliar para *esOptimo_MAX*, en la cual se le dan como parámetros el vector "bj" y el "zRow". Dentro de la función, se usan "bj" y "zRow" para darlos como parámetros a la función *todosPositivosOCero*. Si se cumple que para ambos el flag es TRUE, *esOptimo_MAX* regresa TRUE. De lo contrario regresa FALSE.

Esto se usa solo en caso de que se esté trabajando con un caso de maximización.

- <u>Función auxiliar que nos dice nos pregunta si llegamos al óptimo (MINIMIZACIÓN):</u>

```
esOptimo_MIN <- function(bj, zrow){
  if(todosPositivosOCero(bj) && todosNegativosOCero(zrow))
    res<- TRUE
  else
    res<- FALSE
  res
}
```

  - Usamos *todosNegativosOCero*

```
todosNegativosOCero <- function(vectorInput){
  flag = TRUE
  for (i in 1:(length(vectorInput))){
    if(vectorInput[i]>0){
      flag = FALSE
    }
  }
  flag
}
```

Este caso es muy parecido al anterior, solo que es para casos de minimización. La función *todosNegativosOCero* recibe como parámetro un vector, el cual recorre con un for para ver si sus valores son iguales o menores a cero. Se inicializa un flag en TRUE que cambia a FALSE si no cumplen la condición todos los valores del vector. Arroja como respuesta el valor booleano del flag.

Esta función se usa como auxiliar para *esOptimo_MIN*, la cual recibe como parámetros el vector “bj” y el “zRow”. Dentro, se usan estos datos como parámetros para la *todosNegativosOCero* y se usa un if para ver que en ambos casos se cumpla que el resultado arrojado por la función auxiliar sea TRUE. Si es así, la respuesta que da *esOptimo_MIN* es TRUE. De lo contrario es FALSE.

- <u>Simplex Primal Max:</u>

```
#   Simplex Primal Max
simplexPrimalMax <- function(bj, zrow, ci){

  nombresBase <- names(bj)
  vectorEntrada <- names(which(zrow == min(zrow)))[1]
  vectorAuxiliar <- bj/ci[,vectorEntrada ]
  colIndexAux <- which(ci[, vectorEntrada]>0)
  vectorSalida <- names(which(vectorAuxiliar == min(vectorAuxiliar)))
  nombresBase[which(nombresBase==vectorSalida)] = vectorEntrada
  nombresBase

}
```

Esta función es la que ejecuta las operaciones del caso primal en ejercicios de maximización.

Recibe como parámetros el vector “bj”, el “zRow” y la matriz con valores “cij”. Sacamos los nombres de las variables del vector “bj”, el nombre de la variable con el valor mínimo en “zRow”, dividimos los valores del vector “bj” entre los valores de la columna de la variable que entra, que es el “vectorEntrada”, y se le asigna el nombre de “vectorAuxiliar”. Con este se ve cuál es el valor mínimo obtenido de la división de “bj”/”ci” y sacamos el nombre de esa variable. Finalmente se hace el switch del nombre de la variable en la base entre la que sale y la que entra para que se tenga en el vector “xB” la nueva base.

- <u>Simplex Primal Min:</u>

```r
#   Simplex Primal Min
simplexPrimalMin <- function(bj, zrow, ci){

  nombresBase <- names(bj)
  vectorEntrada <- names(which(zrow == max(zrow)))[1]
  vectorAuxiliar <- bj/ci[,vectorEntrada ]
  colIndexAux <- which(ci[, vectorEntrada]>0)
  vectorSalida <- names(which(vectorAuxiliar == min(vectorAuxiliar)))
  nombresBase[which(nombresBase==vectorSalida)] = vectorEntrada
  nombresBase

}
```

Esta es muy similar a la anterior. Es la función que ejecuta las operaciones para encontrar el cambio de variable en el caso primal, pero para los ejercicios de minimización. La única diferencia es que cuando busca el "vectorEntrada", es decir, la variable que entrará a la base en la iteración, toma el valor máximo de "zRow" en lugar del mínimo. El resto funciona igual que en *simplexPrimalMax*.

- <u>Simplex Dual:</u>

```r
simplexDual <- function(bj, zrow, ci){

  nombresBase <- names(bj)
  vectorSalida <- names(which(bj==min(bj)))
  zRowIndexesAux <- which(zrow<0)
  vectorAuxiliar <- abs(zrow[zRowIndexesAux]/ci[which(bj==min(bj)), zRowIndexesAux])
  vectorEntrada <- names(which(vectorAuxiliar == min(vectorAuxiliar)))
  nombresBase[which(nombresBase==vectorSalida)] = vectorEntrada
  nombresBase

}
```

Lo que hace esta función es ejecutar las operaciones para realizar el cambio de base en los casos en lo que se debe usar la solución dual para los problemas. Es parecido al caso anterior, con ciertas modificaciones.

Recibe como parámetro el vector "bj", el "zRow" y la matriz de valores "cij". Primero se guardan los nombres de las variables del vector "bj" en "nombresBase". Después se decide cuál será la variable que sale, que es el valor mínimo del vector "bj", y se guarda el nombre de esta en "vectorSalida". En "zrowIndexAux" se buscan los valores de "zRow" que son menores a cero, porque solamente estos son los que se usan para realizar la siguiente operación que determina la variable que entra. En "vectorAuxiliar" se guardan los valores resultantes de la división de los valores guardados en "zrowIndexAux" entre su valor respectivo en la fila que corresponde al valor mínimo de “bj”. Este “vectorAuxiliar” se usa para determinar la variable que entra buscando cuál es el valor mínimo que contiene.

Finalmente, se hace el cambio entre los nombres de la variable que entra y la que sale. La función regresa los nombres de las variables que componen la nueva base.

- Función auxiliar que se pregunta si utilizaremos Simplex Primal en TRUE y si no, utilizaremos DUAL:

```
esPrimal <- function(bj){
  if(todosPositivosOCero(bj))
    res<-TRUE
  else
    res<- FALSE
  res
}
```

Esta función se utiliza para determinar si se debe ejecutar el caso dual o el primal. Lo que recibe como parámetro es el vector “bj”. Este lo usa como parámetro para la función *todosPositivosOCero*, que se explicó anteriormente, la cual se pone en un if. Si arroja TRUE, entonces la respuesta del if es TRUE, de lo contrario es FALSE. Esto se usará para que en la función principal se tome el camino correspondiente y se pueda ejecutar el programa correctamente.

- Función extra para establecer una matriz estética:

```
funcionRecopiladora<- function(zRow, zSol, ci, bj, xB){

  semisolutionMatrix1 <- matrix(zRow, byrow = TRUE, nrow = 1)
  semisolutionMatrix1 <- cbind(semisolutionMatrix1, zSol)
  semisolutionMatrix2 <- cbind(ci, bj)
  #semisolutionMatrix1
  #semisolutionMatrix2
  solutionMatrix <- rbind(semisolutionMatrix1, semisolutionMatrix2)
  #solutionMatrix
  colNamesAux <- append(names(zRow), "sol", 5)
  colnames(solutionMatrix) <-colNamesAux
  rowNamesAux <- append(xB, "b.v.", 0)
  rownames(solutionMatrix) <- rowNamesAux
  solutionMatrix

}
```

Esta función se usa al final del código para que el usuario pueda ver la solución al problema de una manera estética y comprensible.

Recibe como parámetro todos los componentes de la matriz, que son: el vector “zrow”, el vector “zsol”, la matriz de valores “cij”, el vector “bj” y el vector “xB”. Estos se van acomodando en las filas y columnas correspondientes y finalmente se colocan los nombres de las variables en la fila superior y la columna de la izquierda para que se vea cuáles son los resultados y a qué variable corresponde cada uno.

- <u>Función principal:</u>

```r
algoritmoFinal <- function(c, b, A, max, acuracy){
  bandera = FALSE #Variable auxiliar para conocer optimalidad
  SalidaDeEmergencia = FALSE
  i=1 #contador para evitar loops
  xB <- procesar_datos_iniciales(b, c)#Procesar datos iniciales

  if(max==TRUE){ #CASO MAXIMIZACI?N

    while(bandera == FALSE && i<=1000 && SalidaDeEmergencia==FALSE){#CICLAMOS con la  bandera y el contador
                                                                   #como condiciones

      tryCatch({
        lista2 <- generar_vectores_NBase(xB, A, c) #Primera secci?n de resultados
      }, error = function(e){ SalidaDeEmergencia <<- TRUE})
      if(SalidaDeEmergencia==FALSE){

      lista2 <- generar_vectores_NBase(xB, A, c)#Primera secci?n de resultados
      B <- lista2[2]
      B_inverse <- lista2[3]
      cB <- lista2[4]

      lista3 <- evaluarBase(cB[[1]], B[[1]], A, B_inverse[[1]], b, c) #Segunda secci?n de resultados
      Zrow <- lista3[1]
      Zsol <- lista3[3]
      cij <- lista3[2]
      bj <- lista3[4]
```

```r
      bjAux <- as.vector(bj[[1]]) #Limpieza y transformaci?n de resultados
      names(bjAux) <- xB
      zRowAux <- as.vector(Zrow[[1]])
      names(zRowAux) <- names(c)
      cijAux<- cij[[1]]
      rownames(cijAux) <- xB
      zSolAux <- as.numeric(Zsol[[1]])

      if(acuracy==TRUE){ #Caso de precisi?n computacional
        zRowAux <- simplificarVectoraCeros(zRowAux)
        cijAux <- simplificarMatrizaCeros(cijAux)
      }

      if(esOptimo_MAX(bjAux, zRowAux)==FALSE){ #Condicion de optimalidad

        if(esPrimal(bj[[1]])==TRUE){ #Caso Primal o Dual (Condici?n de factibilidad)
          nuevaBase <- simplexPrimalMax(bjAux, zRowAux, cijAux)
        }else{
          nuevaBase <- simplexDual(bjAux, zRowAux, cijAux)
        }
        xB <- nuevaBase #Guardamos la nueva base y aumentamos el contador
        i=i+1;

      }else{
        bandera = TRUE; #En el caso ?ptimo, convertimos la bandera a TRUE
      }
      }
    }
```

```
  }else{ #CASO MINIMIZACI?N

    while(bandera == FALSE && i<=1000 && SalidaDeEmergencia==FALSE){#CICLAMOS con la  bandera y
                                                                     #el contador como condiciones

      tryCatch({
      lista2 <- generar_vectores_NBase(xB, A, c) #Primera secci?n de resultados
      }, error = function(e){ SalidaDeEmergencia <<- TRUE})
      if(SalidaDeEmergencia==FALSE){

      B <- lista2[2]
      B_inverse <- lista2[3]
      cB <- lista2[4]

      lista3 <- evaluarBase(cB[[1]], B[[1]], A, B_inverse[[1]], b, c) #Segunda secci?n de resultados
      Zrow <- lista3[1]
      Zsol <- lista3[3]
      cij <- lista3[2]
      bj <- lista3[4]

      bjAux <- as.vector(bj[[1]]) #Limpieza y transformaci?n de resultados
      names(bjAux) <- xB
      zRowAux <- as.vector(Zrow[[1]])
      names(zRowAux) <- names(c)
      cijAux<- cij[[1]]
      rownames(cijAux) <- xB
      zSolAux <- as.numeric(Zsol[[1]])

      if(acuracy==TRUE){ #Caso de precisi?n computacional
        zRowAux <- simplificarVectoraCeros(zRowAux)
        cijAux <- simplificarMatrizaCeros(cijAux)
      }
      if(esOptimo_MIN(bjAux, zRowAux)==FALSE){##Condicion de optimalidad

        if(esPrimal(bj[[1]])==TRUE){#Caso Primal o Dual (Condici?n de factibilidad)
          nuevaBase <- simplexPrimalMin(bjAux, zRowAux, cijAux)
        }else{
          nuevaBase <- simplexDual(bjAux, zRowAux, cijAux)
        }
        xB <- nuevaBase #Guardamos la nueva base y aumentamos el contador
        i=i+1;

      }else{
        bandera = TRUE #En el caso ?ptimo, convertimos la bandera a TRUE
      }
    }
    }
  }

  if(SalidaDeEmergencia == TRUE){
    matrixRes <- "Tuvimos alg?n error computacional en las matrices, ?Ya intentaste activar la precisi?n?"
  }else{
    if(!bandera){ #Nos preguntamos si encontramos un ?ptimo
      matrixRes <- "No pudimos encontrar el ?ptimo"
    }else{
      matrixRes <- funcionRecopiladora(zRowAux, zSolAux, cijAux, bjAux, xB) #Intentamos recopilar los resultados
    }
  }

  matrixRes #Regresamos los resultados
}
```

Esta es la función más importante porque es en la que se combinan todas las que se explicaron antes. En ella se presentan diferentes casos, pues algunas veces se ejecuta para problemas de minimización, otras para maximización y también varían entre casos primales y duales. Explicaremos cómo se va ejecutando cada parte:

Primero recibe como parámetros los datos que ingresa el usuario que son: el vector “c” (coeficientes de función objetivo), el vector “b” (recursos), la matriz “A” (coeficientes de variables en restricciones), la variable que indica si es maximización o minimización que entra como booleana llamada “max” y una variable “accuracy” que es una variable booleana dada por el usuario que inicia en FALSE, pero puede cambiarse a TRUE como se explicó en el Manual del Usuario.

Se inicializa una bandera en FALSE que sirve para saber si se llegó al óptimo, momento en el cual se cambia a TRUE, un contador “i” para saber el número de iteraciones que lleva y evitar que se quede en un loop y una variable booleana llamada “SalidaDeEmergencia” en FALSE que más adelante se explica para qué sirve. Después, se declara la variable “xB” que es el vector con los nombres de las variables que componen la primera base. Este se obtiene con la función *procesar_datos_iniciales* a la que se le dan como parámetros “b” y “c”.

Entra un if-else. La condición del if es que la variable booleana max sea TRUE, con lo que entonces se ejecuta el caso de maximización. Si es FALSE se va al else, en el que se ejecuta el caso de minimización.

En el caso de maximización se inicia un while, el cual corre hasta que la bandera cambia a TRUE o se exceden las 1000 iteraciones. Dentro del while se colocó un tryCatch que sirve para verificar que la matriz “B” sea invertible. En caso de serlo se inicializa una variable “lista2” que es el resultado de la función *generar_vectores_NBase* que obtiene como parámetros “xB”, “A” y “c”. Lo que se hace es acomodar todos los datos en el formato adecuado para que se pueda trabajar con ellos. En caso de que no sea invertible, la variable “salidaDeEmergencia” cambia a TRUE y el programa arroja un resultado con la frase “Tuvimos algún error computacional. ¿Ya intentaste activar la precisión?”

Al salir del tryCatch, entra a un if en el que se revisa que la variable “salidaDeEmergencia” sea FALSE. Si no es, sale del algoritmo y se muestra el mensaje mencionado. Si es, a los datos de “lista2” se les asigna el nombre correspondiente para trabajar con el algoritmo como lo aprendimos en clase; así se inicializa “B” que es la matriz de coeficientes de la base en las restricciones, también la inversa de “B” que es “B_inverse” y el vector “cB” que es el de los coeficientes de las variables de la base en la función objetivo.

Después se inicializa “lista3” que se obtiene a partir de la función *evaluarBase* a la que se le dan como parámetros “cB”, “B”, “A”, “B_inverse”, “b” y “c”. Así se obtienen los valores de la matriz con la que se trabaja. A partir de estos datos se asignan los nombres de los datos de la matriz, que son “Zrow”, “Zsol”, “cij” y “bj”. Se convierte “bj” en vector en la variable “bjAux” para poder aplicarle otras funciones. Se le asignan los nombres de “xB” a “bjAux”. Se hace lo mismo con “zRow” en “zRowAux” y se le asignan los nombres de las variables del vector c. Igual es el caso con “cijAux” para asignarle los nombres del vector “xB”. Se inicializa “zSolAux” que contiene “zSol” en valor numérico.

Después se inicia un if que revisa la variable accuracy. Si está en TRUE, se ejecuta la función *simplificarMatrizaCeros* y se reasigna el valor a "cijAux". También se hace con "zRowAux" con la función *simplificaVectoraCeros.* Esto solo se ejecuta en casos específicos en los que el programa no encontró un óptimo y se muestra un mensaje que sugiere que se cambie la variable accuracy a TRUE.

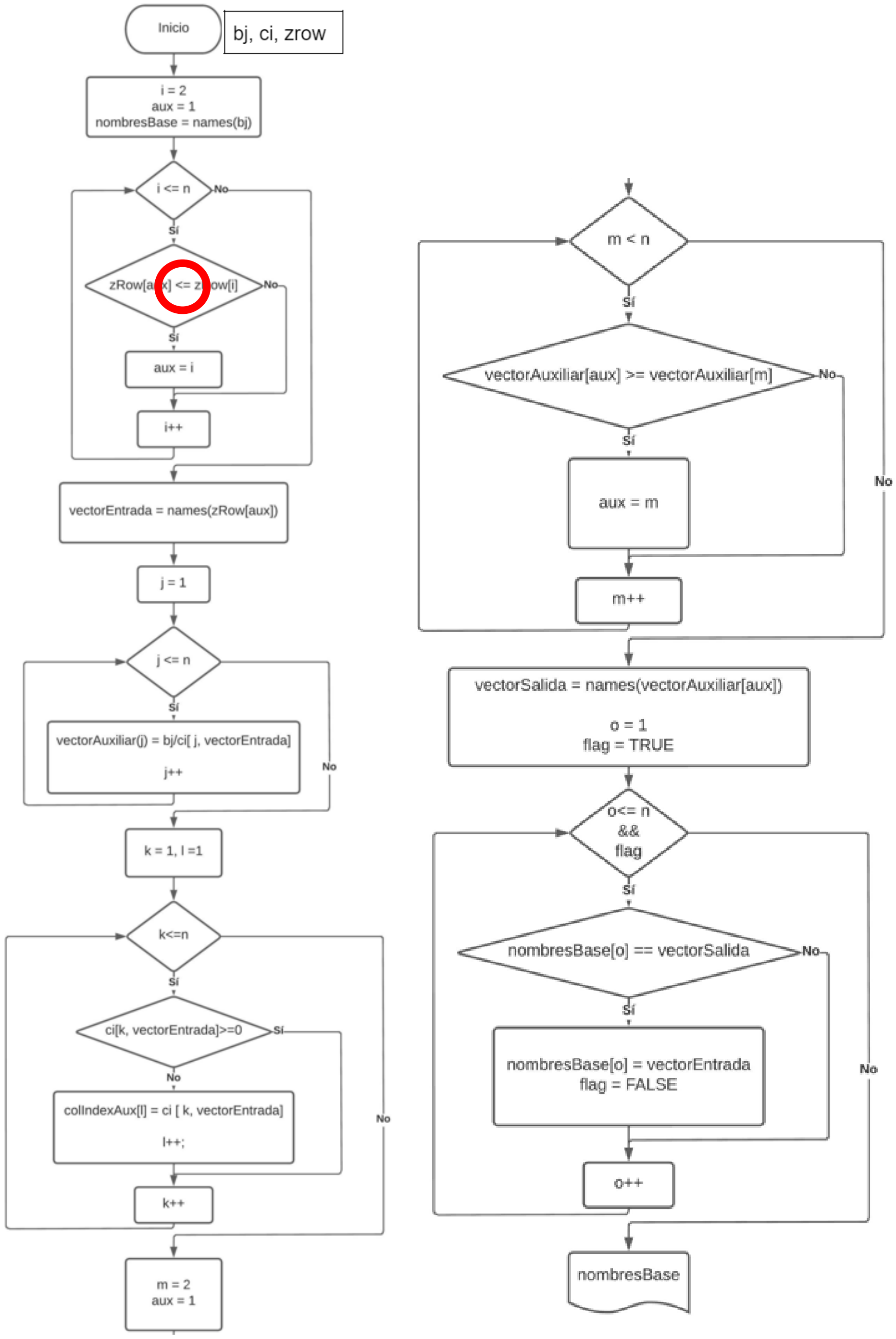
Se declara un if en el que se revisa si la solución ya es óptima con la función *esOptimo_MAX* que recibe como parámetros "bjAux", "zRowAux". Si la respuesta es TRUE, entonces no entra al if y cambia la "bandera" que se inicializó al principio por TRUE para dar fin al programa. Si es FALSE entonces entra al if en el que hay otro if para ver si *esPrimal* con "bj" como parámetro es TRUE. Si es, se genera una nueva base con la función *simplexPrimal* que recibe "bjAux", "zRowAux" y "ciAux" como parámetros. Si no es TRUE, ejecuta la función *simplexDual* con los mismos parámetros. Al salir del if, se asigna a "xB" los datos de la "nuevaBase" generada y se aumenta el valor del contador en uno.

Para el caso de minimización se hace básicamente lo mismo, simplemente cambiando las funciones de maximización por las de minimización que se explicaron anteriormente. Entonces, en la segunda parte, cuando se declara el if para checar si es solución óptima, ahora se usa *esOptimo_MIN*.

Al salir del while de iteraciones hay un if que pregunta si "salidaDeEmergencia" es TRUE. En caso de serlo, se muestra el mensaje que se indicó anteriormente. Si es FALSE, se hace un nuevo if que revisa el valor booleano de la "bandera". Si sigue siendo FALSE, entonces se arroja el mensaje "No pudimos encontrar el óptimo" para que lo vea el usuario. Si es TRUE, se genera una matriz de respuesta con el nombre de "matrixRes", la cual se obtiene por medio de *funcionRecopiladora* que obtiene como parámetros "zRowAux", "zSolAux", "cijAux", "bjAux", y "xB" y con eso se le arroja la solución al usuario.

b. Diagrama de la condición de optimalidad

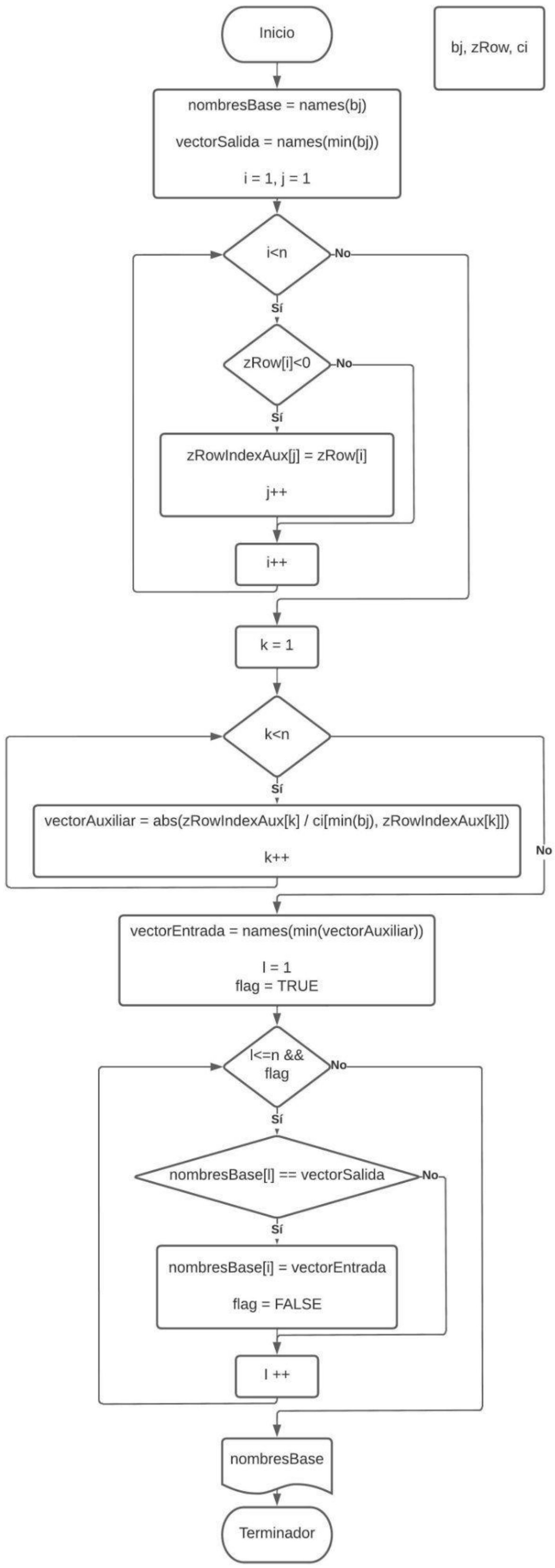
i. Minimización



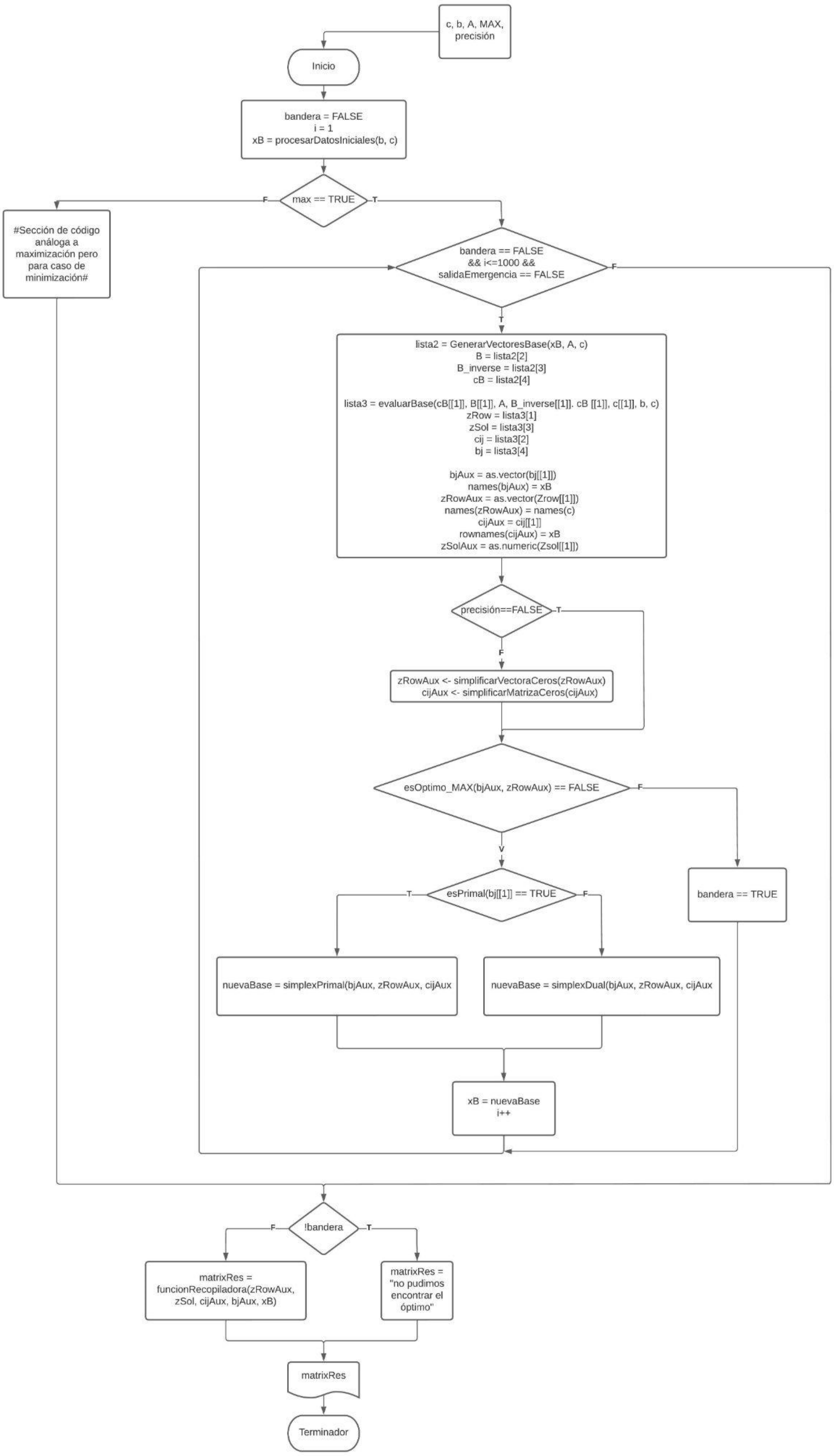
ii. Maximización

El caso de maximización es análogo al de minimización, pero en la sección en la que hay un círculo rojo se debe sustituir el <= por >=. Para evitar hacer más páginas consideramos conveniente especificarlo en el mismo diagrama de minimización.

c. Diagrama de la condición de factibilidad



d. Diagrama de Simplex



e. Manual del usuario

La aplicación es muy amigable con el usuario. Se asume que el usuario tiene conocimiento del método Simplex y del funcionamiento de R pues ahí se desarrolló completamente el algoritmo. De no saber R, igualmente se puede utilizar el programa siguiendo las instrucciones. Como usuario solo se necesita contar con esta aplicación en el ordenador. Una vez que se haya descargado y abierto el archivo, se siguen los siguientes pasos:

1. Se debe correr el programa completo una vez para que se guarden las variables y las funciones en el Environment de R.
2. Una vez hecho esto, el usuario puede ingresar los datos del problema que quiere resolver, los cuales son:
    a. c: el vector de coeficientes de la función objetivo, incluyendo las variables de holgura y exceso.
        i. c ← c(60, 40, 0, 0)
    b. c_names: el vector de nombres de las variables, incluyendo las de holgura y exceso.
        i. c_names ← c(“x1”, “x2”, “e1”, “e2”)
    c. Asignar los nombres de c_names al vector c.
        i. names(c) = c_names
    d. b: el vector con los valores de las restricciones que corresponden a las variables de holgura y exceso. Si son de exceso, se deben poner en negativo.
        i. b ← c(-6, -8)
    e. A: la matriz con los coeficientes de las variables en las restricciones. Se deben poner por fila. Cada fila es una restricción y cada columna una variable. Se debe aclarar el número de filas y de columnas. Si se trata de variables de exceso, los valores se necesitan poner ya multiplicados por -1 para que a la derecha de la matriz quede la matriz identidad.
        i. A ← matrix(c(-5, -4, 1, 0, -10, -4, 0, 1), byrow = TRUE, nrow = 2, ncol = 4)
    f. Asignar los nombres de c_names a las columnas de A.
        i. colnames(A) ← c_names
    g. Indicar si se trata de un caso de maximización o minimización con una variable booleana.

i. Maximización:
    - max = TRUE

ii. Minimización:
    - max = FALSE

h. Inicializar la variable accuracy como FALSE.
    i. miResultado← algoritmoFinal(c, b, A, max, FALSE)

i. Crear una variable en la que se guarde el resultado que se obtiene corriendo la función *algoritmoFinal,* a la que se le dan como parámetros “c”, “b”, “A”, “max”, “accuracy”.
    i. miResultado ← algoritmoFinal(c, b, A, max, FALSE)

j. Imprimir el resultado colocando el nombre de la variable que lo contiene.
    i. miResultado.

k. Correr todo lo que se acaba de ingresar.

Todo junto se ve de esta manera:

Información que da el usuario.

c ← c(60, 40, 0, 0)

c_names ← c("x1", "x2", "e1", "e2")

names(c) = c_names

b ← c(-6, -8)

A ← matrix(c(-5, -4, 1, 0, -10, -4, 0, 1), byrow = TRUE, nrow = 2, ncol = 4)

colnames(A) ← c_names

max = FALSE

Aplicar función principal.

miResultado← algoritmoFinal(c, b, A, max, FALSE)

miResultado

3. Hay tres resultados posibles:
    a. Si se llegó al óptimo, en la consola se mostrará la matriz resultante que contiene hasta la derecha el vector solución.

```
     x1 x2   e1    e2  sol
b.v.  0  0 -8.0 -2.00 64.0
x2    0  1 -0.5  0.25  1.0
x1    1  0  0.2 -0.20  0.4
```

b. Si no se llegó al óptimo hay dos posibilidades:
   i. Se muestra el mensaje: “No pudimos encontrar el óptimo” que significa que no hay solución.
   ii. Se muestra el mensaje: "Tuvimos algún error computacional en las matrices. ¿Ya intentaste activar la precisión?"
      - Si sale este mensaje, entonces la variable de precisión, “accuracy”, que se dio como parámetro en FALSE, se debe cambiar a TRUE. Esto es porque es posible que el error se haya dado por un problema computacional en la matriz que tiene valores muy parecidos a cero y no es invertible.
         a. Nota: no es recomendable activar la variable “accuracy” desde el principio porque, una vez activada, el código puede arrojar *warnings*. Estas no deberían afectar el resultado, pero se recomienda leerlas y tomarlas en cuenta.
      - Si vuelve a aparecer este mensaje, es que el algoritmo no encontró una solución óptima.

3. Ejemplos

Tratamos de abarcar todos los casos posibles, para lo cual designamos un espacio del código para el “Main de Pruebas”. En él hay casos de maximización y minimización, además de un ejemplo dual de minimización. También fuimos haciendo pruebas a lo largo del código para verificar el funcionamiento de cada una de sus partes.
Aquí se incluyen un ejemplo de maximización y uno de minimización:

a. Ejemplo de maximización

```r
#   Informacion que da el usuario:


c1 <- c(3, 5, 0, 0, 0)
c_names1 <- c("x1", "x2", "s1", "s2", "s3")
names(c1) = c_names1
b1 <- c(4, 12, 18)
A1 <- matrix(c(1,0,1,0,0,0,2,0,1,0,3,2,0,0,1), byrow = TRUE, nrow = 3)
colnames(A1)<- c_names1
max1 = TRUE


#   Utilizar el algoritmo para un caso de maximización

miResultado1<-algoritmoFinal(c1, b1, A1, max1)
miResultado1
```

Solución arrojada por el modelo:

```
     x1 x2 s1         s2         s3 sol
b.v.  0  0  0  1.5000000  1.0000000  36
s1    0  0  1  0.3333333 -0.3333333   2
x2    0  1  0  0.5000000  0.0000000   6
x1    1  0  0 -0.3333333  0.3333333   2
```

b. Ejemplo de minimización

```r
#   Informacion que da el usuario:


c2 <- c(60, 40, 0, 0)
c_names2 <- c("x1", "x2", "e1", "e2")
names(c2) = c_names2
b2 <- c(-6, -8)
A2 <- matrix(c(-5, -4, 1, 0, -10, -4, 0, 1), byrow = TRUE, nrow = 2, ncol = 4)
colnames(A2)<- c_names2
max2 = FALSE


#   Utilizar el algoritmo para un caso de minimización

miResultado1<-algoritmoFinal(c2, b2, A2, max2)
miResultado1
```

Solución arrojada por el modelo:

```
     x1 x2   e1    e2  sol
b.v.  0  0 -8.0 -2.00 64.0
x2    0  1 -0.5  0.25  1.0
x1    1  0  0.2 -0.20  0.4
```

4. Conclusión

Con el proyecto terminado, es posible afirmar que se cumplieron los objetivos establecidos al inicio. A lo largo del trabajo, se pudieron desarrollar las habilidades necesarias para hacer uso del lenguaje de programación R, además de que se pudo trabajar correctamente el modelo Simplex y entender a profundidad lo que implica su funcionamiento.

En realidad, ya conocíamos la funcionalidad del algoritmo porque lo trabajamos en la clase de Modelado I, por lo que entenderlo no se nos dificultó mucho. El mayor reto que se presentó en la elaboración fue el descifrar el lenguaje de R, pues teníamos conocimiento de otros lenguajes de programación, pero hay pequeñas variaciones que pueden hacer toda la diferencia entre que funcione o no el programa. Por ejemplo, un caso específico fue cuando aplicamos el tryCatch. No lográbamos que se cambiara el valor de la variable *salidadDeEmergencia* a TRUE y esto solo era porque para que se quedara como valor, debía ponerse doble flecha. Todos estos detalles los fuimos investigando en internet y al final logramos que funcionara correctamente.

Se tuvo que modificar el programa varias veces porque cada vez pensábamos en más escenarios que podrían hacer que tronara. Esto fue lo que más tiempo tomó, pues se hacían las modificaciones y luego hacíamos pruebas para ver que funcionara, pero muchas veces salía un error y tardábamos un tiempo en descubrir lo que fallaba para después ver cómo arreglarlo.

El modelo demostró ser una herramienta compleja y efectiva para la resolución de problemas de optimización que requiere de diversos componentes, los cuales se fueron desarrollando poco a poco para llegar al resultado. Como se mencionó en la introducción, este algoritmo es de suma importancia para ramas administrativas en las que es necesario encontrar la manera más efectiva de trabajar ciertos temas, maximizando o minimizando un valor, tomando en cuenta los recursos que se tienen, que son las restricciones. Estamos seguros de que este tipo de análisis y trabajo lo podremos aplicar en cosas de gran relevancia en nuestros futuros como profesionistas y que tenemos sentadas las bases de lo que debemos hacer en caso de que nos enfrentemos a un problema de esta índole.

5. Código completo en R

```r
#Función axuliar que maneja el rango de error computacional
simplificarVectoraCeros <- function(input){
  for (i in 1:length(input)){
    if(abs(input[i])<=0.0001){
      input[i]<- 0
    }
  }
  input


}

simplificarMatrizaCeros <- function(input){
  for (i in 1:nrow(input)){
    for (j in 1:ncol(input)){
      if(abs(input[i,j]<=0.0001)){
        input[i,j]<-0
      }
    }
  }
  input
}
```

```r
#   Funcion que genera la primera base con los datos iniciales

procesar_datos_iniciales <- function(b, c){

  xB <- names(c[(length(c)-length(b)+1):length(c)])
  xB

}
```

```r
#   Funcion que genera la segunda seccion con la nueva base dada
generar_vectores_NBase <- function(nuevaBase, A, c){

  xB <- nuevaBase
  B <- matrix(A[, (nuevaBase)], nrow = length(nuevaBase), ncol = length(nuevaBase), byrow = FALSE)
  B_inverse <- solve(B)
  cb <- c[nuevaBase]
  names(cb) <- nuevaBase
  res <- list(xB, B, B_inverse, cb)
  res

}
```

```r
#   Funcion que evalua genera la seccion tres generada a partir de los datos de
#   la seccion dos y su respectiva base
evaluarBase <- function(cB, B, A, B_inverse, b, c){

  zrow <- cB%*%B_inverse%*%A - c
  ci <- B_inverse%*%A
  zsol <- cB%*%B_inverse%*%b
  bj <- B_inverse%*%b
  res3 <- list(zrow, ci, zsol, bj)
  res3

}
```

```r
todosPositivosOCero <- function(vectorInput){
  flag = TRUE
  for (i in 1:(length(vectorInput))){
    if(vectorInput[i]<0){
      flag = FALSE
    }
  }
  flag

}
```

```r
todosNegativosOCero <- function(vectorInput){
  flag = TRUE
  for (i in 1:(length(vectorInput))){
    if(vectorInput[i]>0){
      flag = FALSE
    }
  }
  flag
}
```

```r
#   Funcion auxiliar que nos dice nos pregunta si llegamos al Optimo (MAXIMIZACION)
esOptimo_MAX <- function(bj, zrow){
  if(todosPositivosOCero(bj) && todosPositivosOCero(zrow))
    res<-TRUE
  else
    res<-FALSE
  res
}
```

```r
#   Funcion auxiliar que nos dice nos pregunta si llegamos al Optimo (MINIMIZACION)
esOptimo_MIN <- function(bj, zrow){
  if(todosPositivosOCero(bj) && todosNegativosOCero(zrow))
    res<- TRUE
  else
    res<- FALSE
  res
}
```

```r
#   Simplex Primal Max
simplexPrimalMax <- function(bj, zrow, ci){

  nombresBase <- names(bj)
  vectorEntrada <- names(which(zrow == min(zrow)))[1]
  vectorAuxiliar <- bj/ci[,vectorEntrada ]
  colIndexAux <- which(ci[, vectorEntrada]>0)
  vectorSalida <- names(which(vectorAuxiliar == min(vectorAuxiliar)))
  nombresBase[which(nombresBase==vectorSalida)] = vectorEntrada
  nombresBase

}
```

```r
#   Simplex Primal Min
simplexPrimalMin <- function(bj, zrow, ci){

  nombresBase <- names(bj)
  vectorEntrada <- names(which(zrow == max(zrow)))[1]
  vectorAuxiliar <- bj/ci[,vectorEntrada ]
  colIndexAux <- which(ci[, vectorEntrada]>0)
  vectorSalida <- names(which(vectorAuxiliar == min(vectorAuxiliar)))
  nombresBase[which(nombresBase==vectorSalida)] = vectorEntrada
  nombresBase

}
```

```r
#   Simplex Dual
simplexDual <- function(bj, zrow, ci){

  nombresBase <- names(bj)
  vectorSalida <- names(which(bj==min(bj)))
  zRowIndexesAux <- which(zrow<0)
  vectorAuxiliar <- abs(zrow[zRowIndexesAux]/ci[which(bj==min(bj)), zRowIndexesAux])
  vectorEntrada <- names(which(vectorAuxiliar == min(vectorAuxiliar)))
  nombresBase[which(nombresBase==vectorSalida)] = vectorEntrada
  nombresBase

}
```

```r
#   Funcion auxiliar que se pregunta si utilizaremos Simplex Primal en TRUE y si
#   no, utilizaremos DUAL
esPrimal <- function(bj){
  if(todosPositivosOCero(bj))
    res<-TRUE
  else
    res<- FALSE
  res
}
```

```r
#   Función extra para establecer una matriz estetica:

funcionRecopiladora<- function(zRow, zSol, ci, bj, xB){

  semisolutionMatrix1 <- matrix(zRow, byrow = TRUE, nrow = 1)
  semisolutionMatrix1 <- cbind(semisolutionMatrix1, zSol)
  semisolutionMatrix2 <- cbind(ci, bj)
  #semisolutionMatrix1
  #semisolutionMatrix2
  solutionMatrix <- rbind(semisolutionMatrix1, semisolutionMatrix2)
  #solutionMatrix
  colNamesAux <- append(names(zRow), "sol", 5)
  colnames(solutionMatrix) <-colNamesAux
  rowNamesAux <- append(xB, "b.v.", 0)
  rownames(solutionMatrix) <- rowNamesAux
  solutionMatrix

}
```

```r
algoritmoFinal <- function(c, b, A, max, acuracy){
  bandera = FALSE #Variable auxiliar para conocer optimalidad
  SalidaDeEmergencia = FALSE
  i=1 #contador para evitar loops
  xB <- procesar_datos_iniciales(b, c)#Procesar datos iniciales

  if(max==TRUE){ #CASO MAXIMIZACI?N

    while(bandera == FALSE && i<=1000 && SalidaDeEmergencia==FALSE){#CICLAMOS con la  bandera y el contador
                                                                     #como condiciones

      tryCatch({
        lista2 <- generar_vectores_NBase(xB, A, c) #Primera secci?n de resultados
      }, error = function(e){ SalidaDeEmergencia <<- TRUE})
      if(SalidaDeEmergencia==FALSE){

      lista2 <- generar_vectores_NBase(xB, A, c)#Primera secci?n de resultados
      B <- lista2[2]
      B_inverse <- lista2[3]
      cB <- lista2[4]

      lista3 <- evaluarBase(cB[[1]], B[[1]], A, B_inverse[[1]], b, c) #Segunda secci?n de resultados
      Zrow <- lista3[1]
      Zsol <- lista3[3]
      cij <- lista3[2]
      bj <- lista3[4]
```

```
      bjAux <- as.vector(bj[[1]]) #Limpieza y transformaci?n de resultados
      names(bjAux) <- xB
      zRowAux <- as.vector(Zrow[[1]])
      names(zRowAux) <- names(c)
      cijAux<- cij[[1]]
      rownames(cijAux) <- xB
      zSolAux <- as.numeric(Zsol[[1]])

      if(acuracy==TRUE){ #Caso de precisi?n computacional
        zRowAux <- simplificarVectoraCeros(zRowAux)
        cijAux <- simplificarMatrizaCeros(cijAux)
      }

      if(esOptimo_MAX(bjAux, zRowAux)==FALSE){ #Condicion de optimalidad

        if(esPrimal(bj[[1]])==TRUE){ #Caso Primal o Dual (Condici?n de factibilidad)
          nuevaBase <- simplexPrimalMax(bjAux, zRowAux, cijAux)
        }else{
          nuevaBase <- simplexDual(bjAux, zRowAux, cijAux)
        }
        xB <- nuevaBase #Guardamos la nueva base y aumentamos el contador
        i=i+1;

      }else{
        bandera = TRUE; #En el caso ?ptimo, convertimos la bandera a TRUE
      }
      }
    }
```

```
  }else{ #CASO MINIMIZACI?N

    while(bandera == FALSE && i<=1000 && SalidaDeEmergencia==FALSE){#CICLAMOS con la  bandera y
                                                               #el contador como condiciones

      tryCatch({
      lista2 <- generar_vectores_NBase(xB, A, c) #Primera secci?n de resultados
      }, error = function(e){ SalidaDeEmergencia <<- TRUE})
      if(SalidaDeEmergencia==FALSE){

      B <- lista2[2]
      B_inverse <- lista2[3]
      cB <- lista2[4]

      lista3 <- evaluarBase(cB[[1]], B[[1]], A, B_inverse[[1]], b, c) #Segunda secci?n de resultados
      Zrow <- lista3[1]
      Zsol <- lista3[3]
      cij <- lista3[2]
      bj <- lista3[4]

      bjAux <- as.vector(bj[[1]]) #Limpieza y transformaci?n de resultados
      names(bjAux) <- xB
      zRowAux <- as.vector(Zrow[[1]])
      names(zRowAux) <- names(c)
      cijAux<- cij[[1]]
      rownames(cijAux) <- xB
      zSolAux <- as.numeric(Zsol[[1]])

      if(acuracy==TRUE){ #Caso de precisi?n computacional
        zRowAux <- simplificarVectoraCeros(zRowAux)
        cijAux <- simplificarMatrizaCeros(cijAux)
```

```
      }

      if(esOptimo_MIN(bjAux, zRowAux)==FALSE){##Condicion de optimalidad

        if(esPrimal(bj[[1]])==TRUE){#Caso Primal o Dual (Condici?n de factibilidad)
          nuevaBase <- simplexPrimalMin(bjAux, zRowAux, cijAux)
        }else{
          nuevaBase <- simplexDual(bjAux, zRowAux, cijAux)
        }
        xB <- nuevaBase #Guardamos la nueva base y aumentamos el contador
        i=i+1;

      }else{
        bandera = TRUE #En el caso ?ptimo, convertimos la bandera a TRUE
      }
    }
    }
  }

  if(SalidaDeEmergencia == TRUE){
    matrixRes <- "Tuvimos alg?n error computacional en las matrices, ?Ya intentaste activar la precisi?n?"
  }else{
    if(!bandera){ #Nos preguntamos si encontramos un ?ptimo
      matrixRes <- "No pudimos encontrar el ?ptimo"
    }else{
      matrixRes <- funcionRecopiladora(zRowAux, zSolAux, cijAux, bjAux, xB) #Intentamos recopilar los resultados
    }
  }

  matrixRes #Regresamos los resultados
}
```