Simplex and Affine Scaling Algorithms

Daniel Gemara

The goal for this program is to implement the simplex algorithm we used in class (MIE1620). This method is different from the standard simplex method, as it uses a direction vector to update from the current solution to the next one. The goal of this paper will be to walk through the code and explain its implementation. The code is also attached as a .R file.

Introduction

The goal in the beginning is simple: create a function that will take in matrix A (the coefficients of the constraints), b (RHS) and c (objective function coefficients). We need to figure out m and n to figure out how many basic and non-basic variables there should be. The indices tells us the sequence of variables (from 1 to m+n) that are in play. The combs function is to give us the possible combinations for the nonbasic varialbes (those set to 0).

```
# Good weather
A = rbind(
  c(1   ,   1   ,   1   ,   0   ,   0   ,   0   ,   0   ,   0   ,   0   ,   1   ,   0   ,   0   ,   0
  c(-3  ,   0   ,   0   ,   1   ,   0   ,   0   ,   0   ,   -1  ,   0   ,   0   ,   1   ,   0   ,   0
  c(0   ,   -3.6    ,   0   ,   0   ,   1   ,   0   ,   0   ,   0   ,   -1  ,   0   ,   0   ,   1   ,
  c(0   ,   0   ,   -24 ,   0   ,   0   ,   1   ,   1   ,   0   ,   0   ,   0   ,   0   ,   0   ,   1
  c(0   ,   0   ,   0   ,   0   ,   0   ,   1   ,   0   ,   0   ,   0   ,   0   ,   0   ,   0   ,   0
)
b = c(500, -200, -240,  0, 6000)
cost = c(150,230,260,-170,-150,-36,-10,238,210,0,0,0,0,0)
#cost = c(x1,x2,x3,x4,w1,w2,w3,w4,y1,y1)

m = dim(A)[1]
m
```

```
## [1] 5
```

```
n = dim(A)[2] - m
n
```

```
## [1] 9
```

```
indices = 1:(m + n)
indices
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14
```

```
combs = t(combn(indices, n))
feasible = FALSE
head(combs)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
## [1,]    1    2    3    4    5    6    7    8    9
## [2,]    1    2    3    4    5    6    7    8   10
## [3,]    1    2    3    4    5    6    7    8   11
## [4,]    1    2    3    4    5    6    7    8   12
## [5,]    1    2    3    4    5    6    7    8   13
## [6,]    1    2    3    4    5    6    7    8   14
```

Singular Matrices

Because our examples are so complicated, we run into a problem when trying to solve matrices. The issue is that if the matrices are singular (or non-invertible), then the program will crash with no results. Therefore, we need to filter out these problematic combinations of variables where it will be singular. This is done by creating a new table and then calculating each determinant. If it is 0, then we filter them out so they dont give us problems later on.

```
  a <- 0
  j <- 1
  while(j <= nrow(combs))
  {
    a[j] <- (det(A[,setdiff(indices, combs[j,])]))
    j <- j + 1
  }
  a1 <- as.matrix(a)
  solns <- cbind(combs,a1)
  final1 <- solns[solns[,(n+1)] != 0,]
  final <- final1[,-(n+1)]
  head(final)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
## [1,]    1    2    3    4    5    6    7    8    9
## [2,]    1    2    3    4    5    6    7    8   12
## [3,]    1    2    3    4    5    6    7    9   11
## [4,]    1    2    3    4    5    6    7   11   12
## [5,]    1    2    3    4    5    6    8    9   13
## [6,]    1    2    3    4    5    6    8   12   13
```

Our first loop is coming up. We are trasversing the filtered rows that don't give a determinant of 0. In this case, the matrix is pretty small, so we might not have run into issues, but we would've later. This block of code is showing us which variables it is selecting to be free and which ones it is selecting to be pivots. Then, it solves for the corner point we are looking at, giving us our first x, also known as x0.

Generating a corner point

```
  for (i in 1:nrow(final)) {
    free = final[i,]     # the indices of the free variables (those set to 0)
    pivots = setdiff(indices, free)    # indices of the pivots (not free)
    corner = rep(0,m+n)
    corner[pivots] = solve(A[,pivots], b)
    feasible = TRUE
  }
```

Passing the tests

These 2 boolean checkers are making sure that (1) the proposed x is all positive and (2) that there are m variables that are not 0. If they are not, then the loop will break, and it will select a solution that obeys both. Otherwise, there is no correct solution and the program will terminate.

```
    if (all(corner >= 0)) {
      feasible = TRUE
      break
    }

  if (!feasible) {
    print('no feasible solution--go optimize something else!')
    return(NULL)
  }
```

This part is more important for the reader, as it is explainging what is going on for the nonbasic and basic matrices, and constructing the x vector that we chose previously. Additionally, we need to test if the reduced cost is greater than 0. We will see soon what happens if it is not.

Generating the x vector

```
  N <- A[,free]
  N
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
## [1,]    0    0    0    0    1    0    0    0    0
## [2,]    0    0   -1    0    0    1    0    0    0
## [3,]    1    0    0   -1    0    0    1    0    0
## [4,]    0    1    0    0    0    0    0    1    0
## [5,]    0    0    0    0    0    0    0    0    1
```

```
  B <- A[,pivots]
  B
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1  1.0    1    0    0
## [2,]   -3  0.0    0    1    0
## [3,]    0 -3.6    0    0    0
## [4,]    0  0.0  -24    0    1
## [5,]    0  0.0    0    0    1
```

```
  c_n <- cost[free]
  c_n
```

```
## [1] -150  -10  238  210    0    0    0    0    0
```

```
  c_b <- cost[pivots]
  c_b
```

```
## [1]  150  230  260 -170  -36
```

```
  x_b <- as.matrix(solve(B, b))
  x_b
```

```
##            [,1]
## [1,]  183.33333
## [2,]   66.66667
## [3,]  250.00000
## [4,]  350.00000
## [5,] 6000.00000
```

```
  pivots
```

```
## [1] 1 2 3 4 6
```

```
  x_n <- as.matrix(rep(0,n))
  x_n
```

```
##       [,1]
##  [1,]    0
##  [2,]    0
##  [3,]    0
##  [4,]    0
##  [5,]    0
##  [6,]    0
##  [7,]    0
##  [8,]    0
##  [9,]    0
```

```
  free
```

```
## [1]  5  7  8  9 10 11 12 13 14
```

```
  x <- rbind(x_b,x_n)
  x
```

```
##             [,1]
##  [1,]  183.33333
##  [2,]   66.66667
##  [3,]  250.00000
```

```
##  [4,]  350.00000
##  [5,] 6000.00000
##  [6,]    0.00000
##  [7,]    0.00000
##  [8,]    0.00000
##  [9,]    0.00000
## [10,]    0.00000
## [11,]    0.00000
## [12,]    0.00000
## [13,]    0.00000
## [14,]    0.00000
```

```
  r <- c_n - (c_b %*% solve(B) %*% N)
  r
```

```
##          [,1]     [,2] [,3]     [,4] [,5] [,6]     [,7]     [,8]     [,9]
## [1,] 13.88889 15.83333   68 46.11111  360  170 163.8889 25.83333 10.16667
```

Here, in the case that any of the elements of r < 0, we get a direction vector by figuring out which element should enter. If the direction is >= 0, then we have an unbounded direction vector, and this terminates the program.

Reduced Costs

```
while (any(r < 0)) {
  entering <- which.min(r)
  entering
  B_inv <- solve(B)
  B_inv
  d <- -B_inv %*% N[, entering]
  d
  # keep track of x1,x2,x3,x4 of current soln
  e <- as.matrix(rep(0, n))
  e[entering] <- 1
  direction <- rbind(d, e)
  direction

  if (all(direction >= 0))  {
  print('unbounded!')
  return(NULL)
  }
}
```

The next goal is to figure out the steplength, by dividing the current x by the negative entries of the direction vector. Then we figure out the leaving variable (the one that went from positive to 0, and then we swap the two to get a new x vector. Below is commented out since r>0.

Direction, Entering and Leaving

```
direction.negative <- replace(direction, direction >= 0, NA)
direction.negative
ratio <- x / direction.negative
ratio
minratio <- which.min(-ratio)
minratio
steplength <- -ratio[minratio]
steplength

xcurrent <- x + steplength * direction
xcurrent

z <- x / xcurrent
z
leaving <- which.max(z)
leaving

temp <- pivots[leaving]
pivots[leaving] <- free[entering]
free[entering] <- temp
```

Failsafe

The next part of the code is our failsafe. If the steplength is 0 (and r < 0), then something has gone horribly wrong. This part of the code, however, will alleviate that. By keeping the variable i from the beginning, this part of the code will generate a new (further down on the list of available solutions) to the next possible one, and then starting all over again within this r<0 loop.

```
# if(steplength == 0)  {
#   i <- i + 1
#
#   free = final[i, ]
#   pivots = setdiff(indices, free)
#   corner = rep(0, m + n)
#   corner[pivots] = solve(A[, pivots], b)
#   print(corner)
#   if (all(corner >= 0) & sum(corner != 0) == m) {
#   feasible = TRUE
#   break
#   }
# }
```

Finale

The last part of the code is here, just a repeat of the beginning,but since r>0 (it gets out of that loop), and we generate our corner point.

```
  corner = rep(0,m+n)
  corner[pivots] = solve(A[,pivots], b)
  corner
```

```
##  [1]  183.33333   66.66667  250.00000  350.00000    0.00000 6000.00000
##  [7]    0.00000    0.00000    0.00000    0.00000    0.00000    0.00000
## [13]    0.00000    0.00000
```

I ran it for the all the farming and finanical examples, and it worked, but crashed my computer for the final farming one. I think it came down to the fact that I couldn't generate the combs database, since its 33 choose 22 (1037158320 rows!).

Affine Scaling Algorithm

This one is a lot more simple (in my opinion), since there are very few things to generate.

The beginning is below. We are generating pi, then r, then f, then the steplength, then y and then our new x. We then convert that to a diagonal, and then we test it before the next loop.

```
A = rbind(c(1, 2, 3,-1, 0, 0), c(2, 1,-1, 0,-1, 0), c(0, 0, 1, 0, 0,-1))
b = c(10, 20, 4)
c = c(1, 1, 1, 0, 0, 0)
x <- diag(c(30, 5, 15, 75, 30, 11)) #where Ax = b, and x > 0


  # d <- A %*% x %*% x %*% t(A)
  # d
  # e <- A %*% x %*% x %*% c
  # e
  # pi <- solve(d, e)
  pi <- solve(A %*% x %*% x %*% t(A), A %*% x %*% x %*% c)

  r <- c - t(A) %*% pi
  r
```

```
##             [,1]
## [1,] 0.21599967
## [2,] 0.48080740
## [3,] 0.38300970
## [4,] 0.08479496
## [5,] 0.34960269
## [6,] 0.71220812
```

```
  f <- -x %*% r
  f
```

```
##             [,1]
## [1,]  -6.479990
## [2,]  -2.404037
## [3,]  -5.745145
## [4,]  -6.359622
## [5,] -10.488081
## [6,]  -7.834289
```

```
  step <- -1 / min(f)
  step
```

```
## [1] 0.09534633
```

```
  y = c(rep(1, ncol(A))) + 0.95 * step * f
  y
```

```
##           [,1]
## [1,] 0.4130489
## [2,] 0.7822447
## [3,] 0.4796104
## [4,] 0.4239517
## [5,] 0.0500000
## [6,] 0.2903778
```

```
  xnew <- diag(c(x %*% y))
  xnew
```

```
##          [,1]     [,2]     [,3]     [,4] [,5]     [,6]
## [1,] 12.39147 0.000000 0.000000  0.00000  0.0 0.000000
## [2,]  0.00000 3.911223 0.000000  0.00000  0.0 0.000000
## [3,]  0.00000 0.000000 7.194156  0.00000  0.0 0.000000
## [4,]  0.00000 0.000000 0.000000 31.79638  0.0 0.000000
## [5,]  0.00000 0.000000 0.000000  0.00000  1.5 0.000000
## [6,]  0.00000 0.000000 0.000000  0.00000  0.0 3.194156
```

Similar to the simplex method, we are generating the new variables all over again, getting a new x for every iteration. We then take the norm between the matrices, and if it is less than the selected tolerance level, we terminate the algorithm and obtain the solution. In our case it is (12, 0,4,14,0,0).

```
test <- 100
tol <- 0.001
while (test > tol) {
  d <- A %*% xnew %*% xnew %*% t(A)
  d

  e <- A %*% xnew %*% xnew %*% c
  e

  pi <- solve(d, e, tol = 1e-50)
  pi

  r <- c - t(A) %*% pi
  r

  f <- -xnew %*% r
  f

  step <- -1 / min(f)
  step

  y = c(rep(1, ncol(A))) + 0.95 * step * f
  y
```

```
  x <- diag(c(xnew %*% y))
  x

  temp <- xnew
  xnew <- x
  x <- temp
  test <- norm(x - xnew)
  test
  if (test < tol) {
  break

  }
}
x
```

```
##          [,1]         [,2]     [,3]     [,4]        [,5]         [,6]
## [1,] 12.00001 0.0000000000 0.000000  0.00000 0.000000000 0.000000e+00
## [2,]  0.00000 0.0002560201 0.000000  0.00000 0.000000000 0.000000e+00
## [3,]  0.00000 0.0000000000 4.000023  0.00000 0.000000000 0.000000e+00
## [4,]  0.00000 0.0000000000 0.000000 14.00059 0.000000000 0.000000e+00
## [5,]  0.00000 0.0000000000 0.000000  0.00000 0.000253837 0.000000e+00
## [6,]  0.00000 0.0000000000 0.000000  0.00000 0.000000000 2.331485e-05
```

I ran the affine scaling algorithm on the farming and financial models, but for some reason I came to the problem with the farming problems. If I went line by line in the program, I would get less then the tolerance level (so it should've terminated), but for some reason it gave me the problem of singular matrices (which I couldn't fix like the simplex one). So it should've worked, since I got the proper solutions, but not sure what went wrong.

Sources: A combination between the tableau simplex algorithm from here and Professor Kwon's book "Introduction to Linear Optimization and Extensions with MATLAB".