Coding the Simplex Algorithm

David Baker

I. Introduction

FOR my project, I decided to code both phases of the Simplex method in the Julia programming language. You can find the code in the appendix, as well as in this repository: https://github.com/davvidbaker/Simplex.jl. The project description says, "I need to see you run your code!", so I have posted a video of me running my code here: https://youtu.be/-X7beYLvKGI. This paper is divided into 4 sections (plus the introduction). Section 2 describes the general approach and assumptions. Section 3 describes the implementation of phase one in detail. Section 4 describes the implementation of phase two in detail. Section 5 discusses the computational complexity of the algorithm.

II. General Approach

This writeup includes code snippets throughout which I explain in detail. But you can also find the entire code attached as an appendix. The code in the appendix includes extensive amounts of logging, so you can see bases, duals, reduced costs, permutation matrices, etc. throughout the iterations.

My implementation of the simplex method [1] expects the problem to be formulated as a minimization problem in standard form, like:

$$\begin{aligned} \min \ & cx \\ \text{subject to } & \mathbf{A}x \leq b \\ & x \geq 0 \end{aligned} \tag{1}$$

```
1  function simplex(; A::Matrix, b::Vector, c::Vector)
2      phase1_result = phase1(A=A, b=b, c=c)
3
4      if (phase1_result[:status] == INFEASIBLE)
5          return INFEASIBLE
6      end
7
8      return phase2(
9          A=A,
10         b=b,
11         c=c,
12         basis=phase1_result[:basis],
13         A_B_inv=phase1_result[:A_B_inv],
14     )
15 end
```

Fig. 1. We break down the Simplex method into its two phases, passing information from the first phase to the second.

At a high level the code is very simple, as shown in Figure 1. We have our `simplex` function which calls in turn `phase1` and `phase2` functions.

Our `simplex` function has three inputs:

```
1  function phase1(; A::Matrix, b::Vector, c::Vector)
2      A, basis, artificial_variable_columns = add_artificial_variable_columns_if_necessary(A)
3      if (length(artificial_variable_columns) > 0)
4          # must excise the artificial variables
5
6          c = zeros(size(A, 2))
7          for a in artificial_variable_columns
8              c[a] = 1
9          end
10         # permutation matrix is a square binary matrix that has exactly one entry of 1 in each
   row and each column with all other entries 0
11         A_B_inv = A[:, basis] # for a square "permutation matrix", its inverse is itself
12         result = phase2(A=A, b=b, c=c, basis=basis, A_B_inv=A_B_inv)
13         x, _, basis, A_B_inv = result
14         still_contains_artificial = any(x → x in artificial_variable_columns, basis)
15
16         if (still_contains_artificial)
17             return Dict(:status ⇒ INFEASIBLE)
18         end
19
20         return Dict(
21             :status ⇒ FEASIBLE,
22             :basis ⇒ basis,
23             :A_B_inv ⇒ A_B_inv
24         )
25     end
26     # permutation matrix is a square binary matrix that has exactly one entry of 1 in each row
   and each column with all other entries 0
27     A_B_inv = A[:, basis] # for a square "permutation matrix", its inverse is itself
28     return Dict(
29         :status ⇒ FEASIBLE,
30         :basis ⇒ basis,
31         :A_B_inv ⇒ A_B_inv
32     )
33 end
```

Fig. 2. Code for *Phase I*.

1) matrix `A`
2) vector `b`
3) vector `c`

These three inputs will be passed directly to a `phase1` function, which will determine whether the linear program is feasible, and if it is, it will return a basis and an $\mathbf{A}_B^{-1}$ matrix to use in *Phase II*. The `phase2` function determines if the problem is unbounded. If the problem is bounded, it will return the values for x along with the objective value. The most interesting part of the algorithm is that `phase1` uses `phase2`, albeit with slightly different inputs.

If you look in the repository, you will see a number of unit tests for the various methods in my code. I use a bunch of homework and class examples as my inputs for the unit tests, as well as a few small problems I found online.

III. Phase I

As mentioned above, the `phase1` function checks if a feasible solution exists, and if the problem is feasible, it provides an initial basic feasible solution to be used in `phase2`. If the problem is found to be infeasible, we will execute an early return from the the `simplex` function (see line 5 of Figure 1).

In detail, this is how `phase1` works (Figure 2.). We start by examining the incoming $\mathbf{A}$ matrix to see if it readily contains an identity structure. This work occurs in the `add_artificial_variable_columns_if_nec...` function, which is shown in Figure 3. The code is relatively

straightforward. We find all the columns that contain an identity structure (a single 1 and the rest 0's). Then we check if those "identity columns" contain a 1 for every row. For the rows where they don't have an identity structure, we append horizontally concatenate a corresponding "identity column" to the $\mathbf{A}$ matrix, or perhaps we should now call it $\tilde{\mathbf{A}}$, to indicate that artificial variable columns have been added. From this method, we return the new $\tilde{\mathbf{A}}$ matrix, the basis which is the column indices corresponding to the "identity columns", and a list of which indices in the $\tilde{\mathbf{A}}$ matrix correspond to artificial variables. This way we can keep track of the artificials and determine if they are excised from the basis.

Next we have some conditional logic depending on whether added an artificial variables. If there were no artificial variables added, we skip from line 2 to line 27 (of Figure 2). If we did add artificial variables, we attempt to excise them from the basis. We do this by defining a new minimization problem:

$$\begin{aligned} \min \ & \tilde{c}\tilde{x} \\ \text{subject to } & \tilde{\mathbf{A}}\tilde{x} \leq b \\ & \tilde{x} \geq 0 \end{aligned} \tag{2}$$

Here the tildes represent modified versions of our original inputs. Importantly, the objective function coefficients are modified so removing the artificial variables from the basis would result in the minimum possible objective value:

$$c_i = \begin{cases} 1 & \text{if } i \text{ corresponds with an artificial variable,} \\ 0 & \text{otherwise.} \end{cases} \tag{3}$$

This can be seen in lines 7-9 of Figure 2.

Our $\tilde{\mathbf{A}}_B$ matrix is now identity-like—it contains all the columns of an identity matrix, but not necessarily in the proper order. This kind of matrix is called a **permutation matrix**. (This is slightly different than the permutation matrix we create in *Phase II*.) For a matrix with this structure, it is equal to its inverse, so we trivially have obtained $\tilde{\mathbf{A}}_B^{-1}$.

We can now take $\tilde{\mathbf{A}}$, $\tilde{\mathbf{A}}_B^{-1}$, $\tilde{c}$, and the basis we obtained earlier and plug them into our `phase2` function, which will be described later. This occurs at line 12 of Figure 2. If this run of `phase2` produces finds an optimal solution that still includes one of of the artificial variables, our linear program is infeasible and we are done (Figure 2 line 17). If we *have* managed to excise all the artificial variables, we proceed to *Phase II*.

IV. Phase II

The function `phase2`, as shown in Figure 4, starts with an initial feasible basic solution using the provided basis. It is simple enough to read the code, but I will touch on a couple noteworthy points.

Phase II of the Simplex method has 2 "degrees of freedom".

1) Choosing the incoming variable t when there are multiple negative values in $\bar{c}$.
2) Choosing the outgoing variable with the minimum ratio test, if two variables produce the same minimum ratio.

For that first degree of freedom—choosing the incoming variable—if we choose the most negative value in $\bar{c}$, there

```
1  # find identity structure, else create it
2  function add_artificial_variable_columns_if_necessary(A::Matrix)::Tuple{Matrix,Vector,Vector}
3      num_rows = size(A, 1)
4      artificial_variable_columns = []
5
6      # ith element of this vector has value k, meaning
7      # the ith row has the identity structure with a 1 in column k
8      identity_structure_in_col = zeros(num_rows)
9
10     for (col_idx, col) in enumerate(eachcol(A))
11         # all elements are 0 except one is 1
12         contains_identity_structure = count(x -> x == 1, col) == 1 && all(x -> x in (0, 1),
13 col)     if !contains_identity_structure
14             continue
15         end
16         row_containing_1 = findfirst(v -> v == 1, col)
17         identity_structure_in_col[row_containing_1] = col_idx
18         if all(x -> x ≠ 0, identity_structure_in_col)
19             basis = Int.(identity_structure_in_col)
20             return (A, basis, artificial_variable_columns)
21         end
22     end
23
24     # for every row that doesn't have an identity structure, add one to the end
25     basis = []
26     for (row_idx, col_with_1) in enumerate(identity_structure_in_col)
27         if (col_with_1 == 0)
28             new_col = zeros(num_rows)
29             new_col[row_idx] = 1
30             A = hcat(A, new_col)
31             num_cols = size(A, 2)
32             push!(basis, num_cols)
33             push!(artificial_variable_columns, num_cols)
34         else
35             push!(basis, col_with_1)
36         end
37     end
38
39     return (A, Int.(basis), artificial_variable_columns)
40 end
```

Fig. 3. Creating artificial variable columns if they are needed.

are degenerate cases where this could cause cycling. This can be avoided simply by invoking Bland's rule, where instead of choosing the minimum $\bar{c}_i$, we choose the first negative value in $\bar{c}$ [2]. For the case of a tie in the minimum ratio test, we have to choose arbitrarily. In my code we choose the index of the first tying value.

In lines 28-30, we are looking at the values of the column $\bar{\mathbf{A}}_{\cdot t}$ and to avoid choosing a negative value of this column, we essentially replace all the negative values with a really tiny number, because we are later dividing by that number. This way, when we do

$$\frac{\bar{b}_i}{\bar{\mathbf{A}}_{it}} \tag{4}$$

we divide by a tiny number and get a really large number. This really large number will not be chosen as the minimum. This is a sort of hacky way to avoid having to keep track of indices if we were to just filter out the negative entries.

V. Computational Complexity

The beginning of `phase1`, where we add the artificial variables to construct a feasible starting basis is of polynomial complexity on the order of $O(mn)$ where m is the number of constraints and n is the number of variables. We could improve the `add_artificial_variable_columns_if_nec...` function by doing pivot operations instead of looking exactly for an identity structure, but this would not increase the worst case complexity, and really the overall complexity of the Simplex algorithm comes from *Phase II*.

`phase2` performs iterative updates to the basis, starting with the feasible solution identified in *Phase I*. Each iteration involves computing the basic solution, reduced costs, and dual

```
function phase2(; A::Matrix, b::Vector, c::Vector, basis::Vector,
  A_B_inv::Union{Matrix,Diagonal})
    while (true)
        # compute basic solution
        b_bar = A_B_inv * b

        # Now compute the duals
        c_B = c[basis]
        y_bar = c_B' * A_B_inv
        c_bar = c' - y_bar * A

        # index of incoming variable to basis
        # Bland's rule, when choosing incoming variable, choose one with smallest index
        t = findfirst(v -> v < 0, c_bar') # could also do find last
        if (t == nothing)
            values = printresults(A, basis, b_bar)
            obj_value = c_B' * b_bar
            return (values, obj_value, basis, A_B_inv)
        end

        A_bar_at = A_B_inv * A[:, t]
        # if no positive entries, we are unbounded
        if findfirst(v -> v > 0, A_bar_at) == nothing
            return UNBOUNDED
        end

        # minimum ratio test
        ∞ = 1 / typemax(Int32) # kinda hacky, just want a huge value
        A_bar_at_modified = map(v -> v < 0 ? ∞ : v, A_bar_at)
        args = [b_bar[i] / A_bar_at_modified[i] for i ∈ 1:length(b_bar)]

        # Bland's rule here also, in the case of multiple equal and minimal values,
        # argmin returns the index of the first one
        r = argmin(args) # position to be excised from basis
        basis[r] = t

        # calculate permutation matrix
        P = Matrix{Float64}(LinearAlgebra.I(length(basis)))
        rth_col = ones(length(basis))
        for i in eachindex(rth_col)
            num = (r == i) ? 1 : -A_bar_at[i]
            den = A_bar_at[r]
            rth_col[i] = (num / den)
        end

        P[:, r] = rth_col

        A_B_inv = P * A_B_inv
    end
end
```

Fig. 4. Code for *Phase II*.

variables, and updating the basis. The implementation avoids explicitly inverting the basis matrix $\mathbf{A}_B^{-1}$, which is computationally expensive ($O(m^3)$). Instead, it iteratively updates $\mathbf{A}_B^{-1}$ using permutation matrices, which is a more efficient method that maintains the basis dynamically. This approach reduces the computational burden.

The total runtime depends on the number of iterations, which is typically $O(m)$ in practice, but can become exponential ($O(2^n)$ in the worst-case scenario. (The problem technically exhibits *combinatorial* growth, but this behaves like exponential growth when $m \approx n/2$). The reason for this complexity is that the the number of potential basic feasible solutions is

$$\binom{n}{m} = \frac{n!}{m!(n-m)!} \tag{5}$$

I mentioned earlier that we invoke Bland's rule to avoid cycling under degeneracy. This comes at a potential cost of slower convergence.

REFERENCES

[1] G. Danzig, "Linear programming in problems for the numerical analysis of the future," in *Proceedings of the Symposium on Modern Calculating Machinery and Numerical Methods, UCLA, July*, 1948, pp. 29–31.

[2] R. G. Bland, "New finite pivoting rules for the simplex method," *Mathematics of Operations Research*, vol. 2, no. 2, p. 103–107, May 1977. [Online]. Available: https://pubsonline.informs.org/doi/10.1287/moor.2.2.103

Appendix A.
simplex.jl

```julia
using DataFrames
using Crayons
using LinearAlgebra

LOG_RESULTS = true # turn off if you don't want to see detailed logs

INFEASIBLE = :INFEASIBLE
FEASIBLE = :FEASIBLE
UNBOUNDED = :UNBOUNDED
header_text = Crayon(foreground=:green, bold=true, underline=true) # bold green text
final_answer_text = Crayon(foreground=:red, bold=true) # bold green text

function log(args...)
    if (LOG_RESULTS)
        println(join(args, ""))
        print(Crayon(reset=true))
    end
end

function disp(arg)
    if (LOG_RESULTS)
        display(arg)
    end
end

function simplex(; A::Matrix, b::Vector, c::Vector)
    phase1_result = phase1(A=A, b=b, c=c)
    log("phase I result: ", phase1_result)

    if (phase1_result[:status] == INFEASIBLE)
        return INFEASIBLE
    end

    return phase2(
        A=A,
        b=b,
        c=c,
        basis=phase1_result[:basis],
        A_B_inv=phase1_result[:A_B_inv],
    )
end

function phase1(; A::Matrix, b::Vector, c::Vector)
    log(header_text, "STARTING SIMPLEX PHASE I")

    A, basis, artificial_variable_columns = add_artificial_variable_columns_if_necessary(A)
    if (length(artificial_variable_columns) > 0)
        # must excise the artificial variables

        c = zeros(size(A, 2))
        for a in artificial_variable_columns
            c[a] = 1
        end
        log("artificial_variable_columns: ", artificial_variable_columns)
        log("A: ", A)
        log("basis: ", basis)
        log("c: ", c)
        # a permutation matrix is a square binary matrix that has exactly one entry of 1 in each
    row and each column with all other entries 0
        A_B_inv = A[:, basis] # for a square "permutation matrix", its inverse is itself
        result = phase2(A=A, b=b, c=c, basis=basis, A_B_inv=A_B_inv)
        log("result: ", result)
        x, _, basis, A_B_inv = result
        log("basis: ", basis)
        log("A_B_inv: ", A_B_inv)
        still_contains_artificial = any(x -> x in artificial_variable_columns, basis)

        if (still_contains_artificial)
            log(final_answer_text, "STOP. Problem is infeasible.")
            return Dict(:status => INFEASIBLE)
        end

        return Dict(
            :status => FEASIBLE,
            :basis => basis,
            :A_B_inv => A_B_inv
        )
    end
    # a permutation matrix is a square binary matrix that has exactly one entry of 1 in each row
    and each column with all other entries 0
    A_B_inv = A[:, basis] # for a square "permutation matrix", its inverse is itself
    return Dict(
        :status => FEASIBLE,
        :basis => basis,
        :A_B_inv => A_B_inv
    )
end

# find identity structure, else create it
function add_artificial_variable_columns_if_necessary(A::Matrix)::Tuple{Matrix,Vector,Vector}
    num_rows = size(A, 1)
    artificial_variable_columns = []
    # TODO, do i need to check for things like [ 2 0 ] and divide by
    #
    # ith element of this vector has value k, meaning
    # the ith row has the identity structure with a 1 in column k
    identity_structure_in_col = zeros(num_rows)

    for (col_idx, col) in enumerate(eachcol(A))
        # all elements are 0 except one is 1
        contains_identity_structure = count(x -> x == 1, col) == 1 && all(x -> x in (0, 1), col)
        if !contains_identity_structure
            continue
        end
        row_containing_1 = findfirst(x -> x == 1, col)
        identity_structure_in_col[row_containing_1] = col_idx
        if all(x -> x == 0, identity_structure_in_col)
            basis = Int.(identity_structure_in_col)
            return (A, basis, artificial_variable_columns)
        end
    end

    # for every row that doesn't have an identity structure, add one to the end
    basis = []
    for (row_idx, col_with_1) in enumerate(identity_structure_in_col)
        if (col_with_1 == 0)
            new_col = zeros(num_rows)
            new_col[row_idx] = 1
            A = hcat(A, new_col)
            num_cols = size(A, 2)
            push!(basis, num_cols)
            push!(artificial_variable_columns, num_cols)
        else
            push!(basis, col_with_1)
        end
    end

    return (A, Int.(basis), artificial_variable_columns)
end

function phase2(; A::Matrix, b::Vector, c::Vector, basis::Vector,
A_B_inv::Union{Matrix,Diagonal})
    log(header_text, "STARTING SIMPLEX PHASE II")
    log("starting basis: $(basis)")
    iteration = 1
    while (true)
        log(Crayon(foreground=:yellow), "iteration: $iteration")
        # compute basic solution
        log("A_B_inv_here")
        display(A_B_inv)
        b_bar = A_B_inv * b
        log("b_bar= $b_bar")

        # now compute the duals
        c_B = c[basis]
        log("c_B", c_B)
        y_bar = c_B' * A_B_inv
        log("y_bar", y_bar)
        c_bar = c' - y_bar * A
        log("c_bar", c_bar)



        # index of incoming variable to basis
        # Bland's rule, when choosing incoming variable, choose one with smallest index
        t = findfirst(v -> v < 0, c_bar') # could also do find last
        log("t=", t)
        if (t === nothing)
            log(final_answer_text, "STOP, optimum found. basis is ", basis)
            log("b_bar=$(b_bar)")
            values = printresults(A, basis, b_bar)
            obj_value = c_B' * b_bar
            log("objective value is", obj_value)
            return (values, obj_value, basis, A_B_inv)
        end

        A_bar_at = A_B_inv * A[:, t]
        log("A_bar_at", A_bar_at)
        # if no positive entries, we are unbounded
        if findfirst(v -> v > 0, A_bar_at) === nothing
            log(final_answer_text, "STOP, unbounded.")
            return UNBOUNDED
        end

        # minimum ratio test
        log("MIN-RATIO-TEST")
        ∞ = 1 // typemax(Int32) # kinda hacky, just want a huge value
        A_bar_at_modified = map(v -> v < 0 ? ∞ : v, A_bar_at)
        log("A_bar_at_modified", A_bar_at_modified)
        args = [b_bar[i] / A_bar_at_modified[i] for i ∈ 1:length(b_bar)]

        # Bland's rule here also, in the case of multiple equal and minimal values,
        # argmin returns the index of the first one
        r = argmin(args) # position to be excised from basis
        log("args=", args)
        log("r=$r is the position to be excised from the basis.")
        basis[r] = t
        log("new basis", basis)

        # calculate permutation matrix
        P = Matrix{Float64}(LinearAlgebra.I(length(basis)))
        rth_col = ones(length(basis))
        for i in eachindex(rth_col)
            num = (r == i) ? 1 : -A_bar_at[i]
            den = A_bar_at[r]
            rth_col[i] = (num / den)
        end

        log("rth col= ", rth_col)
        P[:, r] = rth_col
        log("P")
        disp(P)
        log()


        A_B_inv = P * A_B_inv
        log("new A_B_inv")
        disp(A_B_inv)
        log()

        iteration += 1
    end
end


# prints out user's values
function printresults(A, basis, b_bar)
    variables = ["x$col" for col ∈ 1:size(A, 2)]
    values = zeros(size(variables))
    for i ∈ eachindex(variables)
        b_bar_ind = findfirst(v -> v == i, basis)
        if b_bar_ind !== nothing
            values[i] = b_bar[b_bar_ind]
        end
    end
    df = DataFrame(variables=variables, values=values)
    show(df)
    log()
    return values
end
```

Appendix B
tests.jl

```julia
using Test
using LinearAlgebra

include("./simplex.jl")

# HW 8 problem 4 without given basis
@test simplex(
    A=[-1 1 1 0
        1 -1 0 1],
    b=[4, 10],
    c=[-4, -5, 0, 0]) == UNBOUNDED


@testset "add_artificial_variable_columns_if_necessary" begin
    A, basis, artificial_variable_columns = add_artificial_variable_columns_if_necessary([
        0 1 1
    ])
    @test A == [1 0 1; 0 1 1]
    @test basis == [1, 2]
    @test artificial_variable_columns == []

    A, basis, artificial_variable_columns = add_artificial_variable_columns_if_necessary([
        1 1 0
    ])
    @test A == [0 1 1; 1 1 0]
    @test basis == [3, 1]
    @test artificial_variable_columns == []


    A, basis, artificial_variable_columns = add_artificial_variable_columns_if_necessary([
        1 1 1
    ])
    @test A == [1 0 1 1; 1 1 1 0]
    @test basis == [4, 2]
    @test artificial_variable_columns == [4]

    A, basis, artificial_variable_columns = add_artificial_variable_columns_if_necessary([
        1 1
    ])
    @test A == [1 1 1 0; 1 1 0 1]
    @test basis == [3, 4]
    @test artificial_variable_columns == [3, 4]


    A, basis, artificial_variable_columns = add_artificial_variable_columns_if_necessary([
        1 1
        0 0
        0 0
    ])
    @test A == [
        1 1 1 0 0 0
        1 1 0 1 0 0
        0 0 0 0 1 0
        0 0 0 0 0 1
    ]
    @test basis == [3, 4, 5, 6]
    @test artificial_variable_columns == [3, 4, 5, 6]

    A, basis, artificial_variable_columns = add_artificial_variable_columns_if_necessary([
        1 0 1
    ])
    @test A == [1 1 1 0; 1 0 1 1]
    @test basis == [2, 4]
    @test artificial_variable_columns == [4]
end


# Homework 9 problem 4, infeasible
@test phase1(
    A=[-2 1 -1 0
        0 1 0 1
    ],
    b=[2, 1],
    c=[0, 1, 0, 0]
)[:status] == INFEASIBLE


# HW 7 problem 4
x, obj_value = phase2(
    A=[
        1 1 -1 0 0
        -1 1 0 -1 0
        0 1 0 0 1],
    b=[2, 1, 3],
    c=[1, -2, 0, 0, 0],
    basis=[1, 2, 5],
    A_B_inv=[ # from zork
        0.5 -0.5 0
        0.5 0.5 0
        -0.5 -0.5 1]
)
@test x == [0, 3, 1, 2, 0]
@test obj_value == -6

# HW 7 problem 4 without the given basis and A_B_inv
phase1_result = phase1(
    A=[
        1 1 -1 0 0
        -1 1 0 -1 0
        0 1 0 0 1],
    b=[2, 1, 3],
    c=[1, -2, 0, 0, 0]
)
@test phase1_result[:basis] == [1, 2, 5]
@test phase1_result[:A_B_inv] == [
    0.5 -0.5 0
    0.5 0.5 0
    -0.5 -0.5 1]

# HW 8 problem 4
@test phase2(
    A=[-1 1 1 0
        1 -1 0 1],
    b=[4, 10],
    c=[-4, -5, 0, 0],
    basis=[3, 4],
    A_B_inv=LinearAlgebra.I(2)
) == UNBOUNDED

# HW 8 problem 4 without given basis
@test simplex(
    A=[-1 1 1 0
        1 -1 0 1],
    b=[4, 10],
    c=[-4, -5, 0, 0]) == UNBOUNDED


# 3D Klee Minty Problem
x, obj_value = simplex(
    A=[
        1 0 0 1 0 0
        2 1 0 0 1 0
        4 2 1 0 0 1
    ],
    b=[1, 2, 4],
    c=[-1, -2, -4, 0, 0, 0]
)
@test x[3] == 4
```