CSCI 5454 – Final Project:

Implementation of the Simplex Method

Dan Prendergast

9 Dec 2016

1. Introduction

The Simplex Method is an algorithm for solving linear programming problems. It is perhaps the most widely used optimization algorithm today. Its applications include logistics, production, scheduling, and military planning. As part of this project, a simplex algorithm was implemented. The implementation uses a tableau and pivots according to the Bland Rules.

Theory demonstrates an exponential running time for the simplex algorithm in the worst case; however, practice shows that the algorithm has expected running time much less than this. Tests performed on this implementation demonstrated a polynomial running time as number of variables and number of constraint equations is varied. Running time as a function of matrix sparsity appeared to have a linear relationship.

2. The Algorithm

Let us begin by formalizing the linear programming problem. A linear programming problem is where we try to optimize (i.e., maximize or minimize) a linear objective function, subject to a set of linear constraints. A prototypical example is the candy producer that wants to maximize profits (objective function) by varying the ratios (variables) of different types of candy in prepackaged boxes while remaining within the limits of candy supply, employees, packaging/shipping costs, etc. (constraint functions). In real life many there are many variations of the form of linear programming problems. Objectives may be minimized or maximized, constraints may be upper or lower bounds, and variables may take positive or negative values. The simplex algorithm requires the problem to be stated in canonical form in order to apply the simplex method. The definition of a linear program in canonical form is as follows:

Given a vector of m nonnegative bounds on constraints $= \begin{bmatrix} b_1 \\ \vdots \\ b_m \end{bmatrix}$, a vector of n coefficients in the objective function $\boldsymbol{c} = \begin{bmatrix} c_1 \\ \vdots \\ c_n \end{bmatrix}$, and an $m \times n$ matrix $A = (a_{ij})$, find a vector of n nonnegative variables $\boldsymbol{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$ to solve…

$$\max_{x} \sum_{i=1}^{n} c_i x_i$$

Subject to the constraints…

$$\sum_{j=1}^{n} a_{ij} x_i \leq b_i\,, \qquad \text{for } j = 1, 2, \ldots m$$

Although the form appears restrictive, simple operations can transform almost any linear program into the canonical form. In addition to transforming the problem into canonical form, the implemented algorithm requires the input to be in the form of tableau. This transformation results in an augmented matrix as depicted in the figure below.

x_1	x_2	x_3	x_4	x_5	x_6	M	
2	3	4	1	0	0	0	60
3	1	5	0	1	0	0	46
1	2	1	0	0	1	0	50
−25	−33	−18	0	0	0	1	0

Figure 1: Standard Tableau for Algorithm Input [2]

In the example above the variables x_1 through x_3 are the problem variables from the linear programming definition described above. Variables x_4 through x_6 are slack variables added to transform the constraint inequalities into equalities. The last row of the tableau is the objective function where the first n elements correspond to the c_j coefficients defined above. M is the variable that represents the value of the objective function. The result is a matrix with dimensions $(m + 1) \times (n + m + 2)$.

Now we can present a high-level view of this simplex method implementation. The main block of the implemented algorithm is as follows:

```
def simplex_calc(tableau):
1)   already_opt = 1
2)   neg_c = find_neg_c(tableau)
3)   while neg_c ≠ empty
4)      already_opt = 0
5)      pivot_element, unbound = find_pivot(tableau, neg_c)
6)      pivot(tableau, pivot_element)
7)      neg_c = find_neg_c(tableau)
```

```
8)  if already_opt = 1
9)     return "System already optimal. Variable values and objective
         value are 0."
10) elseif unbound = 1
11)    return "System is unbounded."
12) else
13)    x, opt_value = read(tableau)
14)    return x, opt_value
```

In the algorithm above, the function `find_neg_c` takes the current tableau and scans the last row except for the last element. It returns an array of the indices of the negative elements in the last row of the tableau.

We see that the system is not optimal (i.e., the objective function is not maximized) while negative values of c_j or M are still present in the last row of the tableau. This criteria determines completion of the algorithm. The loop in the algorithm above performs two steps – finding the pivot element and pivoting the tableau. These functions are described below.

a. Finding the Pivot

Much of the success of the simplex method is based on the rules for selecting the pivot element for reducing the tableau. This implementation uses the "Bland Rules" for selecting the pivot element, which guarantee that the system will not cycle [1]. The rule for selecting the pivot column j is to find the most negative element of the last row (excluding the very last element). The Bland Rules guide selection of the pivot row i such that $a_{ij} > 0$ and $\min_i \frac{b_i}{a_{ij}}$. The `find_pivot` is defined as follows:

```
def find_pivot(tableau, neg_c):
1)  pivot_column = neg_c[j] s.t. tableau[last_row,neg_c[j] is most
    negative
2)  prev_ratio = infinity
3)  unbound = 1
4)  for i from 1 to tableau.rows
5)     if unbound = 1 & tableau[i,pivot_column]>0
6)        unbound = 0
7)     ratio = tableau[i,last_column]/tableau[i,pivot_column]
```

```
8)      if tableau[i,pivot_column]>0 & ratio>0 and ratio<prev_ratio
9)         pivot_row = i
10)        prev_ratio = ratio
11)  pivot_element = (pivot_row, pivot_column)
12)  return pivot_element, unbound
```

In addition to finding the pivot element, this function identifies a linear program that is unbounded. Lines 3, 5, and 6 maintain this status. If all elements a_{ij} in the pivot column are less than or equal to zero, then the variable and objective function can be increased infinitely without violating a constraint.

b. Pivot

Next we perform the pivot operation on the problem tableau. The purpose of pivoting on the pivot element a_{ij} is to bring variable x_j into the system solution. Refer back to figure 1. Notice how the columns for the slack variables (x_4 through x_6) have all 0 entries except for a single 1 entry. We call these "basic variables." These variables are in the current solution of the system and their values are given by the b_i value in the same row as the associated 1 entry. By pivoting the tableau on element a_{ij} we are able to bring x_j into the system solution. The code for this operation is given by…

```
def pivot(tableau, pivot_element):
1)   pivot_val = tableau[pivot_row, pivot_column]
2)   for i from 1 to tableau.rows
3)      if i != pivot_row:
4)         factor = tableau[i, pivot_column]/pivot_val
5)         for j from 1 to tableau.columns
6)            tableau[i,j] = tableau[i,j] -
                                  (factor * tableau[pivot_row,j])
7)   for j from 1 to tableau.columns
8)      tableau[pivot_row,j] = tableau[pivot_row,j]/pivot_val
```

This function subtracts multiples of the i^{th} row from the k^{th} row to reduce a_{kj} to 0. Then divides the i^{th} row by a_{ij}, resulting in $a_{ij} = 1$ and all other coefficients $a_{kj} = 0$. By this mechanism, we are able to bring x_j into the solution of our system. As stated above, we continue to pivot until there are no negative elements c_j or M.

c. Reading the Solution

The last step of the process is to read the solution from the final tableau. The process of reading the solution is straightforward. Identify the basic variables (i.e., columns with all "0" entries with the exception of a single "1"), where $x_j = b_i$ such that $a_{ij} = 1$. Keep in mind that only the first n columns correspond the initial problem variables. The other columns represent the slack variables added to turn the constraint inequalities into equalities. The optimal value of the objective function is given by the last element of the last row of the tableau. The function is performed as follows:

```
def read(tableau):
1)   x = []
2)   for j from 1 to n
3)      for i from 1 to tableau.rows
4)         if a_pj = 1 and all other elements a_ij = 0 in column j
5)            push tableau[p,last_column] onto x
6)         else
7)            push 0 onto x
8)   opt_value = tableau[last_row, last_column]
9)   return x, opt_value
```

d. Random Tableau Generator

In order to efficiently test the system, a random tableau generator was developed. Several limits were placed on the generator to prevent invalid responses. For example, if values in matrix A are allowed to be negative, it is possible that the system will be unbounded. For this reason, the generated values of the matrix A are all nonnegative. The random tableau generator asks for user input for the number of variables (not including slack variables), number of constraint equations, magnitude of coefficients, and probability of zeros in matrix A. The function operates as follows:

```
def build_tableau(num_vars, num_eqns, size_coeff, Pr_zeros):
1)   tableau = zeros(num_eqns+1, num_vars+num_eqns+2)
2)   for i from 1 to num_eqns
3)      for j from 1 to num_vars
4)         irf = randomly select 1 with probability = Pr_zeros
5)         if irf = 1
```

```
6)            tableau[i,j] = random integer from 1 to size_coeff
7)   for j from 1 to num_vars
8)      tableau[last_row,j] = random integer from ± size_coeff
9)   for k from 1 to num_eqns+1
10)     tableau[k, num_vars+k] = 1
11)  for d from 1 to num_eqns
12)     tableau[1, last_column] = random integer from 1 to size_coeff
13)  return tableau
```

Lines 2 through 6 create the matrix A values. Lines 7 and 8 create the matrix of c_j values for the objective function. Lines 9 and 10 create the diagonal of 1's for the slack variables, and Lines 11 and 12 create the b_i values for the last column.

3. Analysis

We will now demonstrate the correctness of the simplex algorithm as well as discuss potential problems during execution. We will also analyze the average and worst case running times of the implementation including experimental results.

a. Correctness

Let's begin the proof by stating that every linear programming problem has four possible outcomes when the simplex algorithm is applied:

- The problem is not feasible
- The problem is unbounded
- The problem cycles infinitely
- The problem has an optimal basic feasible solution

Regarding the feasibility of this implementation, the tableau form defined above guarantees that there is a feasible solution to the problem. One requirement of the tableau form is that all values b_i must be nonnegative. Therefore, in the worst-case all values x_j are zero, and the constraints will still be satisfied [2]. In other words, the input to this implementation of the simplex algorithm is limited to feasible problems.

The unbounded condition is identified as the pivot element is being selected. When translating the linear programming problem into the initial tableau, our objective function is rearranged as follows…

$$c_1x_1 + \cdots + c_nx_n = M$$

$$-c_1x_1 - \cdots - c_nx_n + M = 0$$

which becomes the last row of the problem tableau. Thus, our values for c_j become negative in the problem tableau. The algorithm selects the pivot column corresponding to the most negative c_j. The negative coefficient will result in an increase in the objective function, therefore, the most negative coefficient will cause the greatest increase in the objective function when we bring the associated variable x_j into the solution. The `find_pivot` function in this implementation of the simplex method verifies that at least one value a_{ij} in the pivot column are greater than zero. If all values $a_{ij} \leq 0$ then the variable x_j could increase infinitely (i.e., infinitely increasing the objective value) without violating any of the constraint inequalities. If this case occurs, the linear program is unbounded. This implementation identifies that case in the `find_pivot` function on lines 3 and 5.

The problem of cycling is addressed by using the Bland Rules to select the pivot element. According to [1], we can guarantee cycling will not occur by selection of the pivot element a_{ij} by finding $\min_i \frac{b_i}{a_{ij}}$ and $a_{ij} > 0$.

Lastly, we will show that if the algorithm returns a solution, it returns an optimal basic solution. The linear programming problem can be represented as a convex multi-dimensional polyhedron. The basic solutions of the tableau in the simplex algorithm correspond to a vertex of the polyhedron. The process of choosing the pivot column ensures that the objective value is monotonically increasing. As long as a negative value c_j is present in the tableau, we can find another feasible vertex that increases the objective function. The implemented algorithm continues pivoting to the next feasible vertex as long as a negative c_j value exists as indicated by the loop definition in line 3 of the `simplex_calc` function. The algorithm terminates at a vertex representing the basic solution that optimizes the objective function. Because the solution space is convex, we know that this local maximum is also a global maximum.

b. Time

Analysis of the simplex method has proven difficult due to the difference between calculated worst-case performance and practical performance as well as the wide variety of problem configurations that have been tested. For the purposes of this discussion, we will refer to m as the number of constraint equations and n as the number of variables in a linear programming problem (not including slack variables added to transform constrain inequalities into equalities). Theoretically, Dantzig, Bland, and Klee separately proved the worst-case running time to be at least exponential [2]. Their results differed based on assumptions about the feasibility of the initial problem. Dantzig states that the expected complexity to be on the order of $\alpha \cdot m$, where α is between 4 and 6 [4]. As the number of variables is increased, he believes α varies according to

$$\exp(\alpha) < \log_2(2 + \frac{n}{m}) \text{ [4]}$$

However, his estimates assume particular conditions about the problem matrix, such as high sparsity and a triangular form of non-zero coefficients. Lastly, Spielman and Teng perform a smoothed analysis, which is a hybrid of the worst-case and average-case analyses, and demonstrated that the expected running time of the simplex method is a polynomial function of m and n[5].

In this implementation of the simplex method, a random tableau generator was used to test execution time as a function of three factors – number of variables n, number of constraint equations m, and sparsity of matrix A. Five trials were run for each value of n and m, and ten trials were run for each value of the sparsity of A. The running time as a function of number of variables is depicted in Figure 2 below. Figure 3 shows the running time as a function of the number of constraint equations in the problem. The black lines in the figures are a quadratic regression of the mean running times. Note how the running time increases much faster for an increase in constraints compared to an equal increase in variables. This is due to the fact that for every additional constraint equation a slack variable is added to the problem tableau as well.

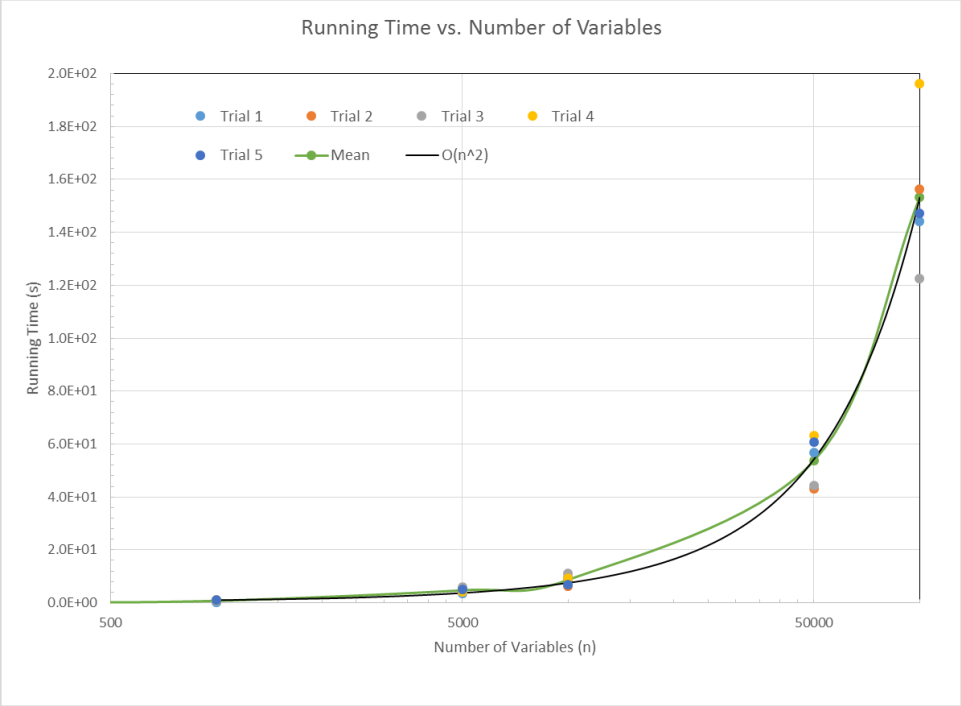

Figure 2: Running Time vs Number of Variables

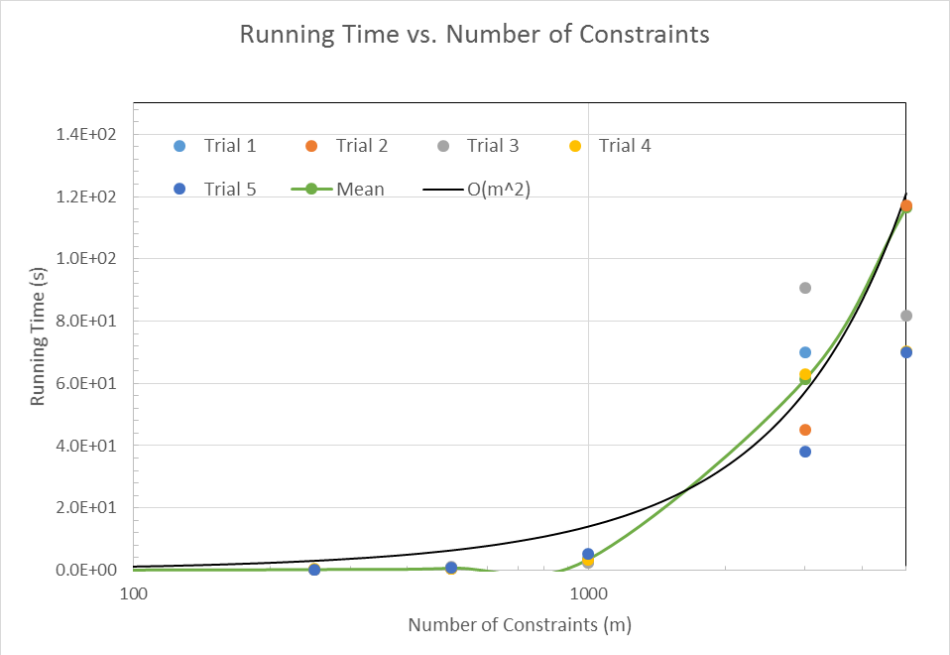

Figure 3: Running Time vs Number of Constraints

Figure 4 shows the running time as a function of sparsity of the matrix A. The relationship appears to be linear; however, no studies were found providing an analysis of running time as a function of sparsity with which to compare these results.

Figure 4: Running Time vs Sparsity of Matrix A

c. Space

Analysis of the space complexity of the simplex algorithm is straightforward. The algorithm stores an $m \times n$ matrix representing the problem tableau. This particular implementation added an array of negative c_j values which is of length n in the worst case (see line 2 of function `simplex_calc`). Therefore the total space complexity is $O(m \times n)$. Experiments were not conducted to verify this.

4. Conclusion

As part of this project, a version of the simplex algorithm was implemented. The implementation utilized a tableau data structure and the Bland Rules for pivoting. Analyses show that the theoretical running time as a function of number of variables and number of constraints is exponential, yet the expected running time is polynomial. Experiments conducted as part of this project seem to confirm the expectations. Future work should include extending this algorithm to take an unstructured linear programming problem and transform it into the initial tableau.

5. References

[1] Bland, Robert G. "New finite pivoting rules for the simplex method." Mathematics of operations Research 2.2 (1977): 103-107.

[2] Lay, David C. *Linear algebra and its applications, 4th ed.*, 2012.

[3] Shamir, Ron. "The efficiency of the simplex method: a survey." Management Science 33.3 (1987): 301-334.

[4] Dantzig, George B. “"Comments on Khachian's Algorithm for Linear Programming," Technical Report SOL 79-22, Department of Operations Research, Stanford University, November 1979.

[5] Spielman, Daniel, and Shang-Hua Teng. "Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time." Proceedings of the thirty-third annual ACM symposium on Theory of computing. ACM, 2001.