Implementation of the Simplex Algorithm

Kurt Mehlhorn

May 18, 2010

1 Overview

There are many excellent public-domain implementations of the simplex algorithm; we list some of them in Section 2. All of them are variants of the revised simplex algorithm which we explain in Sections 4 to 6. The simplex algorithm is a numerical algorithm; it computes with real numbers. Almost all implementations of the simplex algorithm use floating point arithmetic; floating point arithmetic incurs round-off error and hence none of the implementations mentioned in Section 2 are guaranteed to find optimal solutions. They may also declare a feasible problem infeasible or vice versa. The papers [DFK$^+$03, ACDE07] give examples where CPLEX – a very popular commercial solver – and SoPlex – a very popular public-domain solver – fail to find optimal solutions. However, for most instances the inexact solvers return optimal or near optimal solutions. These solutions can be taken as a starting point for an exact solver as we discuss in Section 8. Large linear programs have sparse constraint matrices, i.e., the number of nonzero entries of the constraint matrix is small compared to the total number of entries. For the sake of efficiency, it is important to exploit sparsity.

2 Resources

There are many excellent implementations of the simplex algorithm available. KM has used the public-domain solver SoPlex (`http://soplex.zib.de/`) and the commercial solver CPLEX.

H. Mittelmann maintains a page (`http://plato.asu.edu/ftp/lpfree.html`) where he compares the following LP-solvers. I quote from his web-page.

This benchmark was run on a 2.667GHz Intel Core 2 processor under Linux. The MPS-datafiles for all testcases are in one of (see column "s") `miplib.zib.de/` [1], `plato.asu.edu/ftp/lptestset/` [2] `www.sztaki.hu/~meszaros/public_ftp/lptestset/` (NETLIB[3], MISC[4], PROBLEMATIC[5], STOCHLP[6], KENNINGTON[7], INFEAS[8]) NOTE: files in [2-8] need to be expanded with emps in same directory!

The following codes were tested:

- BPMPD-2.21 www-neos.mcs.anl.gov/neos/solvers/lp:bpmpd/MPS.html (run locally)

- CLP-1.11.0 projects.coin-or.org/Clp PCx-1.1 www-fp.mcs.anl.gov/OTC/Tools/PCx/
- QSOPT-1.0 www.isye.gatech.edu/ wcook/qsopt/
- SOPLEX-1.4.1 soplex.zib.de/
- GLPK-4.36 www.gnu.org/software/glpk/

Times are user times in secs including input.

s	problem	BPMPD	CLP	PCX	QSOPT	SOPLEX	GLPK
2	cont1	15	4011	22	13241	4127	
2	cont11	26		106		51341	
2	cont4	20	2062	44	9682	3355	

Problem sizes

problem	rows	columns	nonzeros
cont1	160793	40398	399991
cont11	160793	80396	439989
cont4	160793	40398	398399

3 Linear Programming in Practice

As you can see from the examples above, LPs can involve a larger number of constraints (160 thousand in the examples above) and variables (40 to 80 thousand in the examples above). However, the constraint matrix tends to be sparse, i.e., the average number of nonzero entries per row or column tend to be small (in the examples about, about 3 nonzero entries per row and 10 nonzero entries per column). Dense constraint matrices also arise in practice. However, then the number of constraints and variables are much smaller. Large LPs have up to 10 million nonzero entries in the constraint matrix.

The main concerns for a good implementation of the simplex method are:

- exploit sparsity of the constraint matrix so as to make iterations fast and keep the space requirement low and
- numerical stability.

I will address the first issue first and turn to the second issue in Section 7.

4 The Standard Simplex Algorithm

We consider LPs in standard form: minimize $c^T x$ subject to $Ax = b$ and $x \geq 0$. The constraint matrix A has m rows and n columns where $m < n$; A is assumed to have full row rank. Let B be the basic variables and N the non-basic variables. A_B is the $m \times m$ submatrix of A selected by the basis. The standard simplex algorithm maintains:

- the basic solution $x_B = A_B^{-1} b$ with $x_B \geq 0$.
- the dictionary $A_B^{-1} A$,
- the objective value $z = c_B^T x_B = c_B^T A_B^{-1} b$, and
- the vector of reduced costs $c^T - c_B^T A_B^{-1} A$. The entries corresponding to the basic variables are zero.

In each iteration, we do the following:

1. find a negative reduced cost, say in column j. If there is none, we terminate. The current solution is optimal.
2. determine the basic variable b_i to leave the basis. For this we inspect the j-th column of the dictionary. If all entries are nonnegative, the problem is unbounded. Let $d_B = -A_B^{-1} A_j$, where A_j is the j-th column of A. Among the $b_i \in B$ with $d_{b_i} < 0$, we find the one that minimizes
$$\frac{x_{b_i}}{|d_{b_i}|}.$$
3. We move to the basis $B \setminus b_i \cup j$. For this purpose, we perform row operations on the tableau. We subtract suitable multiples of the row corresponding to the leaving variable from all other rows to as to generate a unit vector in the j-th row. This will also update cost vector, objective value, and the basic solution.

The cost for the update is $O(nm)$. For $n > m = 10^6$, this is prohibitive. Even more prohibitive is that the matrix $A_B^{-1} A_N$, where A_N is the part of A corresponding to non-basic variables is dense. Thus the space requirement of the tableau implementation is $O(n(n-m))$.

5 The Revised Simplex Method

The revised simplex method reduces the cost of an update to $O(m^2 + n)$ and the space requirement to $O(m^2 + n + nz(A))$, where $nz(A)$ is the number of nonzero entries of the constraint matrix.

How does one store a sparse matrix? One could store it as $n + m$ linear lists, one for each row and column. The linear list corresponding to a row contains the nonzero entries of the row (together with the indices of these entries). With this representation, the cost of a matrix-vector product Ax is $O(nz(A))$. For each row of A, we do the following.

assume that row r has nonzero entries $(r, j_1, a_{r,j_1}), \ldots (r, j_k, a_{r,j_k})$.
$s := 0$, $\ell := 1$
while $(\ell \leq k)$ **do**
  $s := s + a_{r,j_\ell} x_\ell$, $\ell = \ell + 1$
end while

We no longer maintain the dictionary $A_B^{-1}A$. We now only maintain the inverse A_B^{-1} of the basis matrix. We will refine this further in later sections. An iteration becomes:

1. compute $x_B = A_B^{-1}b$. This is a matrix-vector product and takes time $O(m^2)$.
2. compute the vector $c^T - c_B^T A_B^{-1} A$ of reduced costs by first computing $y^T = c_B^T A_B^{-1}$ and then $y^T A$. So we are computing two matrix-vector products; the costs are $O(m^2)$ and $O(nz(A))$, respectively.
3. select the variable j that should enter the basis.
4. compute the relevant column of the dictionary as $d_B = -A_B^{-1} A_j$; this takes time $O(m^2)$.
5. determine the variable b_i that should leave the basis; this takes time $O(m^2)$.
6. update A_B^{-1} by row operations. We are now performing row operations on a matrix of size $m \times m+1$ and hence this step takes time $O(m^2)$.

The space requirement is $O(m^2)$ for the inverse of the basis matrix plus $O(nz(A))$ for the constraint matrix plus $O(n)$ for the vector of reduced costs.

6 Sparse Revised Simplex Method

The inverse of sparse matrix tends to be dense. Can we avoid to store the inverse of A_B? Is there are better way to maintain the inverse of A_B?

An LU-decomposition of a matrix A_B is a pair (L,U) of matrices such that $A_B = LU$, L is a lower diagonal matrix (all entries above the diagonal are zero) and U is an upper diagonal matrix. The LU-decomposition is unique, if we require, in addition, that L has ones on the diagonal. Here are some useful facts about LU-decompositions.

- LU-decompositions can be computed by Gaussian elimination.
- Sparse matrices frequently have sparse LU-decompositions. This might require to permute the rows and columns of A_B. Observe that we may permute the rows and columns of A_B. Permuting columns corresponds to renumbering variables and permuting rows corresponds to renumbering constraints.

  Consider the selection of the first pivot. By interchanging columns and rows, we may move any element of the matrix to the left upper corner. There are two considerations in choosing the element.

- the element should not be too small (certainly nonzero) for the sake of numerical stability. Recall that one is dividing by the pivot element in Gaussian elimination.
- We subtract a multiple of the row containing the pivot from any row where the pivot column contains a nonzero element. Thus the number of nonzero elements created by the step is the product of the number of nonzero elements in the row and column containing the pivot. This number should be small; this rule is know as Markowitz criterion.

- Given an LU-decomposition of a matrix $A_B = LU$, it is easy to solve a linear system $A_B x = b$ in time $O(nz(L) + nz(U))$. We have $A_B x = (LU)x = L(Ux) = b$. We therefore first solve $Ly = b$ and then $Ux = y$. Linear systems with a lower or upper diagonal matrix are easily solved by backward or forward substitution. For example, in order to solve $Ly = b$, we first compute the first entry of y using the first equation, substitute this value into the second equation and solve for the second variable, and so on. The time for this is proportional to the number of nonzero entries of L.

Exercise 1 *Show that the inverse of an upper diagonal matrix is an upper diagonal matrix and can be computed in time proportional to the product of the dimension of the matrix times the number of nonzero entries of the matrix. Also, show that the product of two upper diagonal matrices is upper diagonal.*

The idea is now to use the LU-decomposition of A_B wherever we used A_B^{-1} in the preceding section, i.e., instead of computing $A_B^{-1}b$, we solve $LUx_B = b$ by first computing y with $Ly = b$ and then x_B with $Ux_B = y$. Please convince yourself that steps 1) to 5) are readily performed. We still need to discuss what we do in step 6). Step 6) asks us to update the inverse of the basis matrix. We now need to update the LU-decomposition.

The new basis matrix $\tilde{A}_B$ is obtained from A_B by replacing column b_i of A_B by the j-th column of A. Observe that $A_B e_{b_i}$, where e_{b_i} is the b_i-th unit vector yields the b_i-th column of A_B and hence $A_B e_{b_i} e_{b_i}^T$ is a $m \times m$ matrix whose b_i-th column is equal to $A_B e_{b_i}$ and which has zeros everywhere else. Similarly $A_j e_{b_i}^T$ is a matrix with A_j in column b_i and zeros everywhere else. Thus

$$\tilde{A}_B = A_B - A_B e_{b_i} e_{b_i}^T + A_j e_{b_i}^T = A_B + (A_j - A_B e_{b_i}) e_{b_i}^T.$$

Multiplying by L^{-1} from the left yields

$$L^{-1}\tilde{A}_B = U + (L^{-1}A_j - U e_{b_i}) e_{b_i}^T =: R.$$

The right hand side R is the matrix U with its b_i-th column replaced by the vector $L^{-1}A_j$; in general, this is a sparse vector, because L^{-1} is sparse and A_j is sparse.

We next compute an LU-factorization of R, say $R = \tilde{L}\tilde{U}$. This is not a general LU-factorization since R differs from an upper triangular matrix in only one column. We refer the reader to [SS93] and its references for a detailed discussion on how to compute the LU-decomposition of a nearly upper diagonal matrix sufficiently.

We can now write

$$L^{-1}\tilde{A}_B = R = \tilde{L}\tilde{U}$$

and hence have

$$\tilde{A}_B = L\tilde{L}\tilde{U}.$$

Since the product of two lower triangular matrices is a lower triangular matrix, we now have an LU-decomposition of $\tilde{A}_B$.

Actually, it is better to keep the factorization $L\tilde{L}\tilde{U}$ and to use $A_B^{-1} = U^{-1}\tilde{L}^{-1}L^{-1}$. In this way, the factorization of the basis matrix grows by one factor in every iteration. Accordingly, the costs of the steps 1) to 5) grow in each iteration. When the costs of these steps become too large, one refactors A_B from scratch.

7 Numerical Stability

LP-solvers use floating point arithmetic and hence the arithmetic incurs round-off error. In a row operation, one first scales one of the rows (by dividing by the pivot element) and then subtracts a multiple of this row from another row. Divisions by small elements are numerically instable and hence to be avoided.

An Anecdote: KM taught this course in the year 2000. As part of the course, he organized a competition. He asked the students to compute the optimal solution for an NP-complete cutting-stock problem.

He also took part of the competition. He used a heuristic to produce feasible solutions and used linear programming to compute lower bounds. He found a solution of objective value 213 and the LP proved a lower bound of 212.000001 (KM does not recall the true numbers). Since the solution had to be integral, he knew that he had found the optimum and stopped his heuristic. However, in class a student produced an integral solution of value 212. When KM looked deeper, he found that value 212.000001 was larger than 212 due to round-off errors.

This experience motivated the research reported on in the next section.

8 Exact Solvers

How can one avoid the pitfalls of approximate arithmetic. The answer is easy. Use exact arithmetic. Under the assumption that all coefficients of the problem are integral (or more generally rational), rational arithmetic suffices. The drawback of this approach is that rational arithmetic is much slower than floating point arithmetic and hence only small problems can be solved.

The paper [DFK$^+$03] pioneered a more clever approach. The authors argued that inexact solvers usually find an optimal or a near-optimal basis. So they took the basis returned by the inexact solver as the starting basis for an exact solver implemented in rational arithmetic. For a collection of medium size LPs, they found that CPLEX found the optimal basis in all but two

cases. In the cases, where it did not find the optimum a small number of pivots sufficed to reach the optimum.

[ACDE07] took this a step further. They used floating point arithmetic even more aggressively. Instead of switching to rational arithmetic they switched to higher precision floating arithmetic for computing the LU-decomposition of the constraint matrix. Then they rounded the floating point numbers computed in this way to rational numbers (with small denominators) and tried to verify that the decomposition is correct for the rational numbers obtained in this way. This approach gave them a tremendous speed-up over [DFK$^+$03].

Converting Reals to Rationals: The standard method for finding a rational number with small denominator close to a real number is to compute the continued fraction expansion of the real numbers; see the lectures notes of Computational Geometry and Geometric Computing (winter term 09/10). For example,

$$\begin{aligned}
0.6705 &= 0+\frac{1}{1/0.6705} = 0+\frac{1}{10000/6705} \\
&= 0+\frac{1}{1+3295/6705} = 0+\frac{1}{1+\frac{1}{6705/3295}} \\
&= 0+\frac{1}{1+\frac{1}{2+115/3295}} = 0+\frac{1}{1+\frac{1}{2+\frac{1}{3295/115}}} \\
&= 0+\frac{1}{1+\frac{1}{2+\frac{1}{28+75/115}}} = 0+\frac{1}{1+\frac{1}{2+\frac{1}{28+\frac{1}{115/75}}}}
\end{aligned}$$

In this way, we obtain the following rational approximations with small denominator:

$$0 \qquad 0+\frac{1}{1}=1 \qquad 0+\frac{1}{1+\frac{1}{2}}=\frac{2}{3} \qquad 0+\frac{1}{1+\frac{1}{2+\frac{1}{28}}}=\frac{57}{85}.$$

References

[ACDE07] D. Applegate, W. Cook, S. Dash, and D. Espinoza. Exact solution to linear programming problems. *Operations Research Letters*, 35:693–699, 2007.

[DFK$^+$03] M. Dhiflaoui, S. Funke, C. Kwappik, K. Mehlhorn, M. Seel, E. Schömer, R. Schulte, and D. Weber. Certifying and Repairing Solutions to Large LPs, How Good are LP-solvers?. In *SODA*, pages 255–256, 2003.

[SS93] L. Suhl and U. Suhl. A fast LU update for linear programming. *Annals of Operations Research*, 43:33 – 47, 1993.