ORF 307: Lecture 5

Linear Programming: Chapter 4
Efficiency

Robert Vanderbei

February 16, 2016

Slides last edited on February 28, 2017

Efficiency

Question:

> Given a problem of a certain size, how long will it take to solve it?

Two Kinds of Answers:

- *Average Case.* How long for a *typical* problem.
- *Worst Case.* How long for the *hardest* problem.

Average Case.

- Mathematically difficult.
- Empirical studies.

Worst Case.

- Mathematically tractible.
- Limited value.

Measures

Measures of Size

- Number of constraints m and/or number of variables n.
- Number of data elements, mn.
- Number of nonzero data elements.
- Size, in bytes, of AMPL formulation (model+data).

Measuring Time

Two factors:

- Number of iterations.
- Time per iteration.

Klee–Minty Problem (1972)

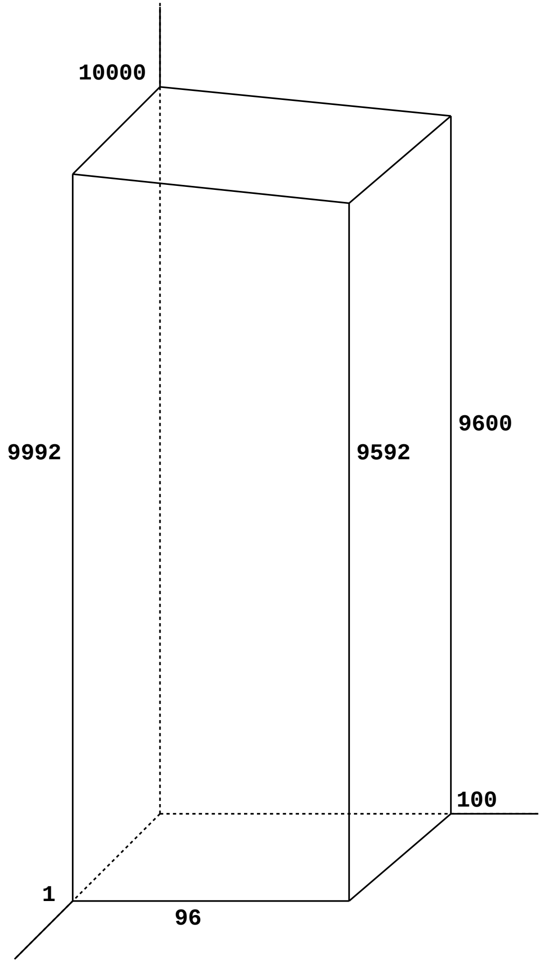
$$
\begin{array}{lll}
\text{maximize} & \displaystyle\sum_{j=1}^{n} 2^{n-j} x_j & \\
\text{subject to} & \displaystyle 2\sum_{j=1}^{i-1} 2^{i-j} x_j + x_i \leq 100^{i-1} & i = 1, 2, \ldots, n \\
& x_j \geq 0 & j = 1, 2, \ldots, n.
\end{array}
$$

Example $n = 3$*:*

$$
\begin{array}{rcccccrcr}
\text{maximize} & 4x_1 & + & 2x_2 & + & x_3 & & & \\
\text{subj. to} & x_1 & & & & & \leq & & 1 \\
& 4x_1 & + & x_2 & & & \leq & & 100 \\
& 8x_1 & + & 4x_2 & + & x_3 & \leq & & 10000 \\
& & & x_1, & x_2, & x_3 & \geq & & 0.
\end{array}
$$

A Distorted Cube

Case $n = 3$:



Constraints represent a "minor" distortion to an n-dimensional hypercube:

$$\begin{aligned} 0 \le \;& x_1 \le 1 \\ 0 \le \;& x_2 \le 100 \\ & \vdots \\ 0 \le \;& x_n \le 100^{n-1}. \end{aligned}$$

Analysis

Replace

$$1, 100, 10000, \ldots,$$

with

$$1 = b_1 \ll b_2 \ll b_3 \ll \ldots.$$

Then, make following replacements to rhs:

$$\begin{array}{lcl}
b_1 & \longrightarrow & b_1 \\
b_2 & \longrightarrow & 2b_1 + b_2 \\
b_3 & \longrightarrow & 4b_1 + 2b_2 + b_3 \\
b_4 & \longrightarrow & 8b_1 + 4b_2 + 2b_3 + b_4 \\
 & \vdots &
\end{array}$$

Hardly a change!

Make a similar constant adjustment to objective function.

Look at the pivot tool version...

Case $n = 3$:

$$
\begin{array}{lclclclclclcl}
\zeta & = & -2 & b_1 + & -1 & b_2 + & 0 & b_3 & + & 4 & x_1 & + & 2 & x_2 & + & 1 & x_3 \\
w_1 & = & 1 & b_1 + & 0 & b_2 + & 0 & b_3 & - & 1 & x_1 & - & 0 & x_2 & - & 0 & x_3 \\
w_2 & = & 2 & b_1 + & 1 & b_2 + & 0 & b_3 & - & 4 & x_1 & - & 1 & x_2 & - & 0 & x_3 \\
w_3 & = & 4 & b_1 + & 2 & b_2 + & 1 & b_3 & - & 8 & x_1 & - & 4 & x_2 & - & 1 & x_3
\end{array}
$$

Now, watch the pivots...

Exponential

Klee–Minty problem shows that:

> Largest-coefficient rule can take $2^n - 1$ pivots to solve a problem in n variables and constraints.

For $n = 70$,

$$2^n = 1.2 \times 10^{21}.$$

At 1000 iterations per second, this problem will take 40 billion years to solve. The age of the universe is estimated to be 13.7 billion years.

Yet, problems with 10,000 to 100,000 variables/constraints are solved routinely every day.

Worst case analysis is just that: worst case.

Complexity

n	n^2	n^3	2^n
1	1	1	2
2	4	8	4
3	9	27	8
4	16	64	16
5	25	125	32
6	36	216	64
7	49	343	128
8	64	512	256
9	81	729	512
10	100	1000	1024
12	144	1728	4096
14	196	2744	16384
16	256	4096	65536
18	324	5832	262144
20	400	8000	1048576
22	484	10648	4194304
24	576	13824	16777216
26	676	17576	67108864
28	784	21952	268435456
30	900	27000	1073741824

Sorting: fast algorithm $= n \log n$,
slow algorithm $= n^2$

Matrix times vector: n^2

Matrix times matrix: n^3

Matrix inversion: n^3

Simplex Method:

- Worst case: $n^2 2^n$ operations.
- Average case: n^3 operations.
- Open question:

  Does there exist a variant of the simplex method whose worst case performance is polynomial?

Linear Programming:

- *Theorem:* There exists an algorithm whose worst case performance is $n^{3.5}$ operations.

Average Case

Matlab Program

Define a random problem:

```
m = ceil(exp(log(400)*rand()));
n = ceil(exp(log(400)*rand()));

A = round(sigma*(randn(m,n)));
b = round(sigma*abs(randn(m,1)));
c = round(sigma*randn(n,1));

A = -A;
```

Initialize a few things:

```
iter = 0;
opt = 0;
```

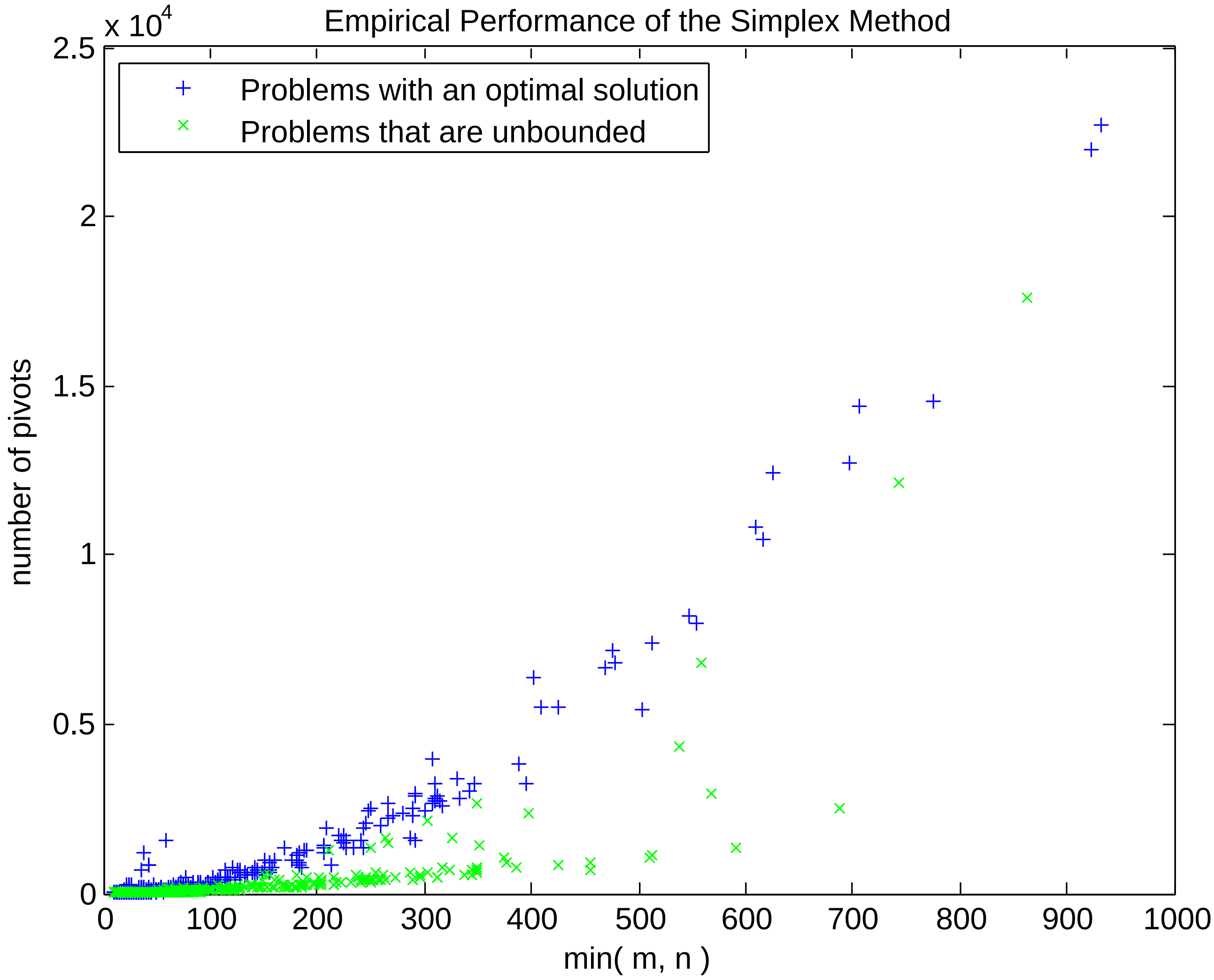
The Main Loop:

```
while max(c) > eps || min(b) < -eps,
   % pick largest coefficient
   [cj, col] = max(c);
   Acol = A(:,col);

   % select leaving variable
   [t, row] = max(-Acol./b);
   if t < eps,
         opt = -1;  % unbounded
         'unbounded'
         break;
   end
   Arow = A(row,:);

   a = A(row,col); % pivot element

   .
   .
   .

   iter = iter+1;
end
```

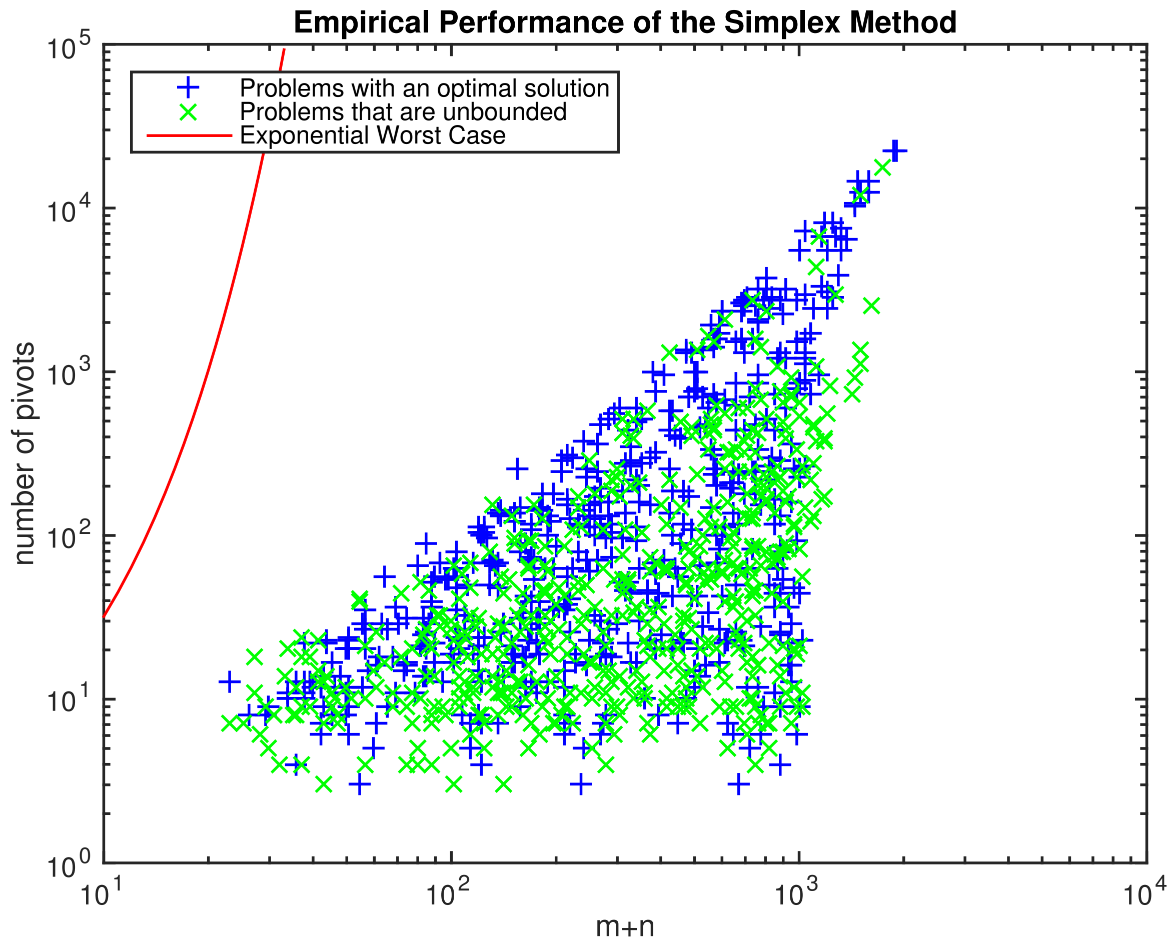
$\Longleftarrow$

The code for a pivot:

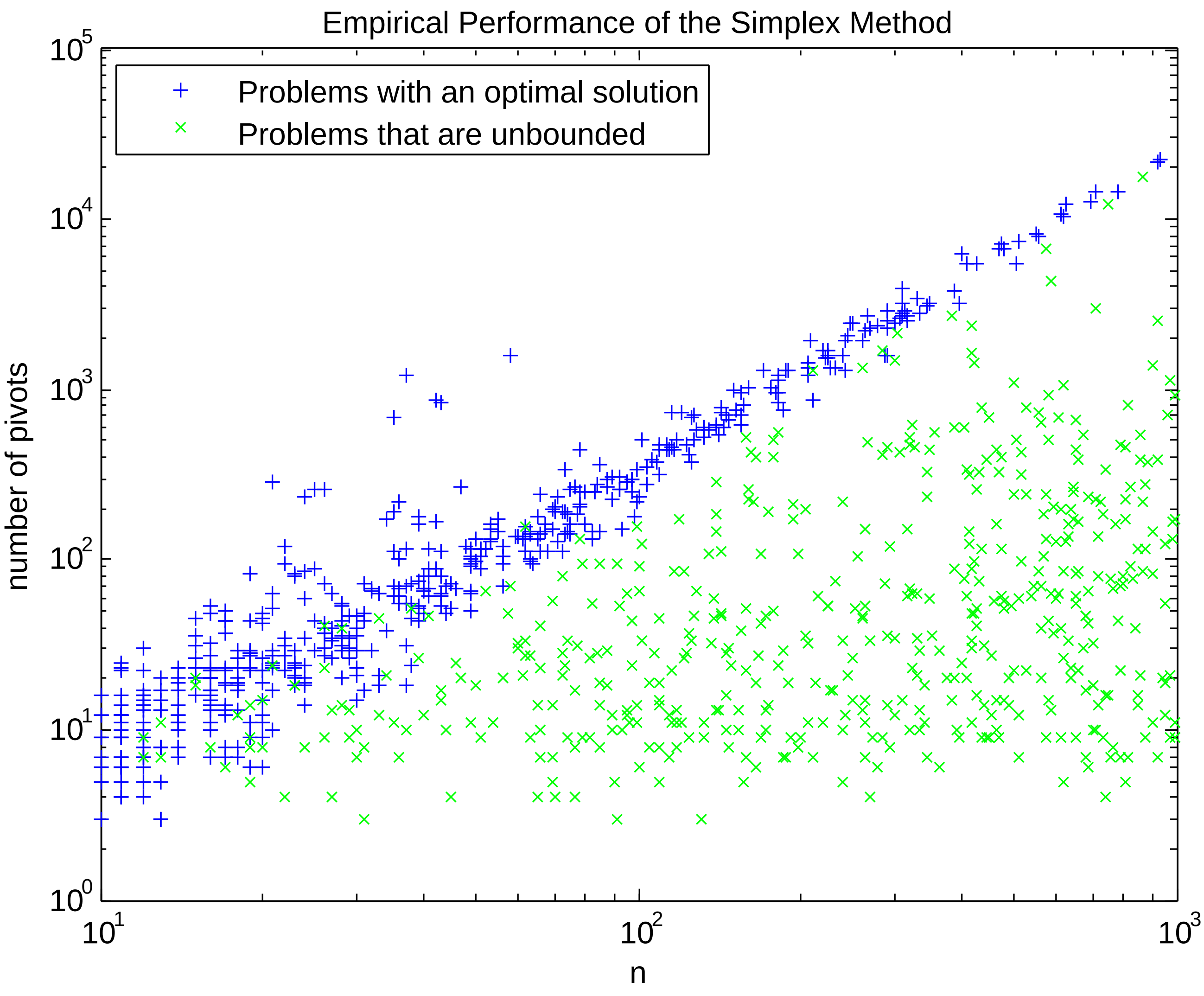
```
A = A - Acol*Arow/a;
A(row,:) = -Arow/a;
A(:,col) = Acol/a;
A(row,col) = 1/a;

brow = b(row);
b = b - Acol*brow/a;
b(row) = -brow/a;

ccol = c(col);
c = c - ccol*Arow/a;
c(col) = ccol/a;
```



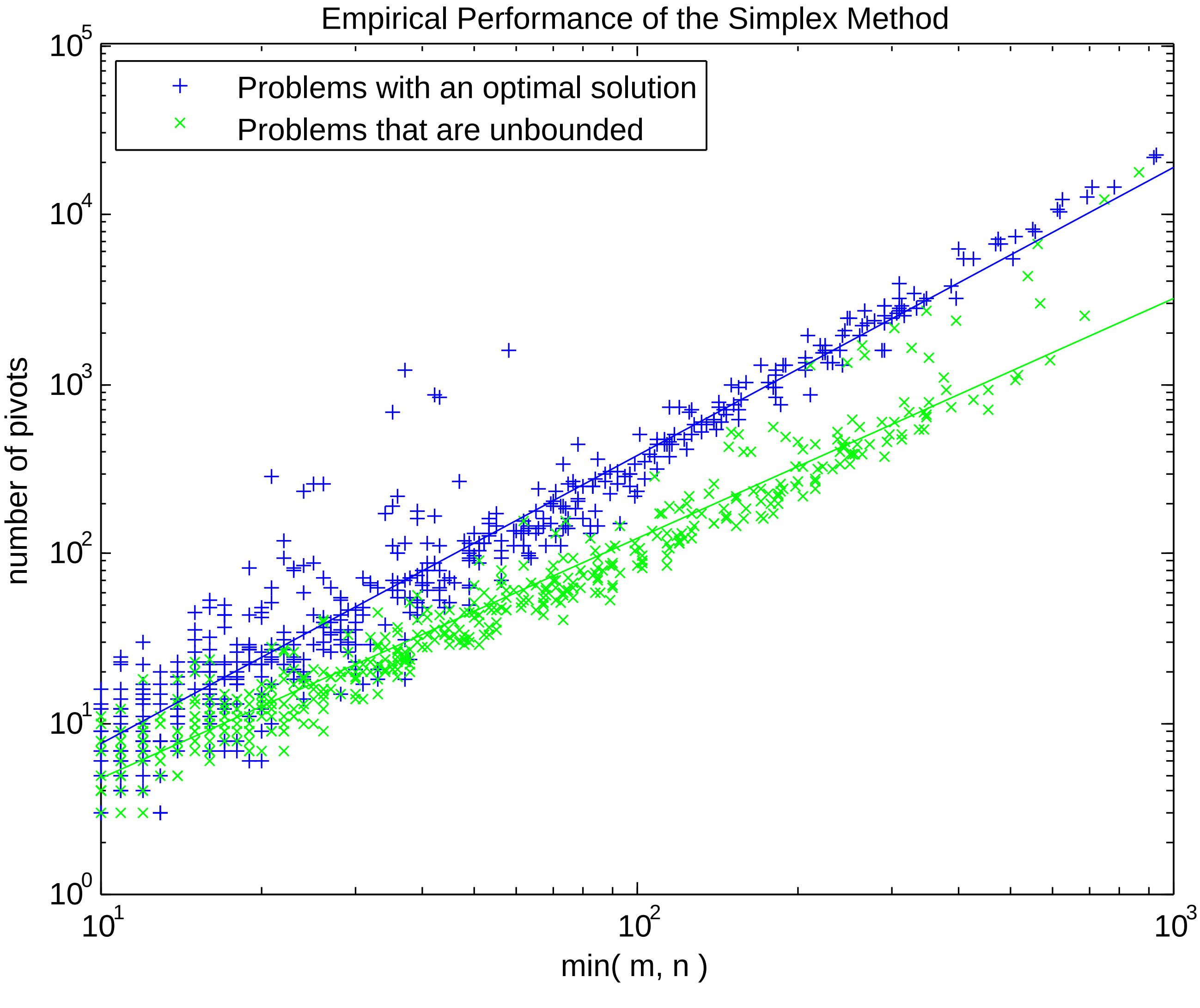
Empirical Performance of the Simplex Method
x 10^4
2.5
2
1.5
1
0.5
0
number of pivots
Problems with an optimal solution
Problems that are unbounded
0
100
200
300
400
500
600
700
800
900
1000
min(m, n)



Empirical Performance of the Simplex Method
Problems with an optimal solution
Problems that are unbounded
Exponential Worst Case
number of pivots
m+n
10^0
10^1
10^2
10^3
10^4
10^5



Empirical Performance of the Simplex Method
Problems with an optimal solution
Problems that are unbounded
number of pivots
n
10^0
10^1
10^2
10^3
10^4
10^5
10^1
10^2
10^3



$$\mathsf{iters} = 0.150\min(\mathsf{m},\mathsf{n})^{1.70} \qquad\qquad \mathsf{iters} = 0.180\min(\mathsf{m},\mathsf{n})^{1.42}$$

Average Case—AMPL Version

Declare parameters:

```
param eps := 1e-9;
param sigma := 30;
param niters := 1000;
param size := 400;

param m;
param n;
param AA {1..size, 1..size};
param bb {1..size};
param cc {1..size};
param A {1..size, 1..size};
param b {1..size};
param c {1..size};
param x {1..size};
param z {1..size};
param y {1..size};
param w {1..size};

param iter;
param opt;
param stop;
param mniters {1..niters, 1..3};
param maxc;
param minbovera;
param col;
param row;
```

```
param Arow {1..size};
param Acol {1..size};
param a;
param brow;
param ccol;
param ii;
param jj;
```

Define a random problem:

```
let m := ceil(exp(log(size)*Uniform01()));
let n := ceil(exp(log(size)*Uniform01()));
let {i in 1..m, j in 1..n} A[i,j] := round(sigma*Normal01());
let {i in 1..m} b[i] := round(sigma*abs(Normal01()));
let {j in 1..n} c[j] := round(sigma*Normal01());
let {i in 1..m, j in 1..n} A[i,j] := -A[i,j];
let {i in 1..m, j in 1..n} AA[i,j] := A[i,j];
let {i in 1..m} bb[i] := b[i];
let {j in 1..n} cc[j] := c[j];
```

The Simplex Method (Phase 2)

```
repeat while (max {j in 1..n} c[j]) > eps {
   let maxc := 0;
   for {j in 1..n} {
       if (c[j] > maxc) then {
          let maxc := c[j];
          let col := j;
       }
   }
   let minbovera := 1/eps;
   for {i in 1..m} {
     if (A[i,col] < -eps) then {
       if (-b[i]/A[i,col] < minbovera) then {
          let minbovera := -b[i]/A[i,col];
          let row := i;
       }
     }
   }
   if minbovera >= 1/eps then {
         let opt := -1; # unbounded
         display "unbounded";
         break;
   }
   .
   .
   .
}
```

$\Longleftarrow$

The code for a pivot:

```
let {j in 1..n} Arow[j]  := A[row,j];
let {i in 1..m} Acol[i]  := A[i,col];
let a := A[row,col];
let {i in 1..m, j in 1..n}
    A[i,j]  := A[i,j] - Acol[i]*Arow[j]/a;
let {j in 1..n} A[row,j]  := -Arow[j]/a;
let {i in 1..m} A[i,col]  :=  Acol[i]/a;
let A[row,col]  := 1/a;

let brow  := b[row];
let {i in 1..m}
    b[i]  := b[i] - brow*Acol[i]/a;
let b[row]  := -brow/a;

let ccol  := c[col];
let {j in 1..n}
    c[j]  := c[j] - ccol*Arow[j]/a;
let c[col]  := ccol/a;
```

The AMPL code can be found here:

http://orfe.princeton.edu/~rvdb/307/lectures/primalsimplex.txt