Dantzig's Simplex Algorithm

How to Write Fast Numerical Code

by Rico Häuselmann & Donjan Rodic

Swiss Federal Institute of Technology (ETH Zurich)

29.05.2013

Linear Programming

Optimising a Linear Program in standard form:

```
Maximize
2x - 3y + z

Subject To
 x  +  y + z <= 10
4x  - 3y + z <=  3
2x  +  y - z <=  6
```

Restrictions

- all coefficients positive (simplicity)
- all coefficients $\leq 10^6$ (stability)

Steps

- Tableau form $$\begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 10 \\ 4 & -3 & 1 & 0 & 1 & 0 & 0 & 3 \\ 2 & 1 & -1 & 0 & 0 & 1 & 0 & 6 \\ 2 & -3 & 1 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$
- Pivoting, reduced cost (objective function), termination
- Worst runtime $O(e^m)$, but often $O(m)$

Steps

- Tableau form $\begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 10 \\ 4 & -3 & 1 & 0 & 1 & 0 & 0 & 3 \\ 2 & 1 & -1 & 0 & 0 & 1 & 0 & 6 \\ 2 & -3 & 1 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$
- Pivoting, reduced cost (objective function), termination
- Worst runtime $O(e^m)$, but often $O(m)$

Steps

- Tableau form $$\begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 10 \\ 4 & -3 & 1 & 0 & 1 & 0 & 0 & 3 \\ 2 & 1 & -1 & 0 & 0 & 1 & 0 & 6 \\ 2 & -3 & 1 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$
- Pivoting, reduced cost (objective function), termination
- Worst runtime $O(e^m)$, but often $O(m)$

Comparison

- GLPK (GNU Linear Programming Kit), solid standard solver
- CPPLEX, mathematical OO-implementation
- Gurobi (CPLEX), fastest (multithreaded) solver available
- SoPlex, fastest FOSS solver available

Comparison

- GLPK (GNU Linear Programming Kit), solid standard solver
- CPPLEX, mathematical OO-implementation
- Gurobi (CPLEX), fastest (multithreaded) solver available
- SoPlex, fastest FOSS solver available

Comparison

- GLPK (GNU Linear Programming Kit), solid standard solver
- CPPLEX, mathematical OO-implementation
- Gurobi (CPLEX), fastest (multithreaded) solver available
- SoPlex, fastest FOSS solver available

Comparison

- GLPK (GNU Linear Programming Kit), solid standard solver
- CPPLEX, mathematical OO-implementation
- Gurobi (CPLEX), fastest (multithreaded) solver available
- SoPlex, fastest FOSS solver available

Properties

- Tableau: $(m+1) \times (m+n+2)$
  (requires full access each iteration)
  Memory reads: $m(m+n)+2m+n$
  (all capacity misses for bigger problems)
  Flops: $2 * m(m+n)+m$
- Computational intensity $I = \frac{2m^2+2mn+4m+2n}{8(m^2+mn)} \sim \frac{1}{4}$

Properties

- Tableau: $(m+1) \times (m+n+2)$
  (requires full access each iteration)
  Memory reads: $m(m+n) + 2m + n$
  (all capacity misses for bigger problems)
  Flops: $2 * m(m+n) + m$
- Computational intensity $I = \frac{2m^2+2mn+4m+2n}{8(m^2+mn)} \sim \frac{1}{4}$

Implementation

Elementary optimisations:

array: use raw pointers in place of std::vector<std::vector>	**ssa**: increase ILP via static assignment
nta: cache control to inhibit polluting tableau data, prefetch rows	**blockX**: reuse pivot row for X concurrent updates
swap: store pivot row at a fixed location (end of tableau)	**sse/avx**: use alignment & intrinsics to speed up float arithmetic

Framework

- Simplex runtime highly unpredictable
- Randomly generate LPs of increasing size (10-4000 vars)
- Grouped into 4 test sets

preview	(162 LPs, 145 MB)	quick testing
standard	(282 LPs, 260 MB)	plots
heavy	(852 LPs, 785 MB)	statistics
high	(50LPs, 3800 MB)	correctness & profiling

Framework

- Simplex runtime highly unpredictable
- Randomly generate LPs of increasing size (10-4000 vars)
- Grouped into 4 test sets

preview	(162 LPs, 145 MB)	quick testing
standard	(282 LPs, 260 MB)	plots
heavy	(852 LPs, 785 MB)	statistics
high	(50LPs, 3800 MB)	correctness & profiling

Framework

- Simplex runtime highly unpredictable
- Randomly generate LPs of increasing size (10-4000 vars)
- Grouped into 4 test sets

preview	(162 LPs, 145 MB)	quick testing
standard	(282 LPs, 260 MB)	plots
heavy	(852 LPs, 785 MB)	statistics
high	(50LPs, 3800 MB)	correctness & profiling

Framework

- Simplex runtime highly unpredictable
- Randomly generate LPs of increasing size (10-4000 vars)
- Grouped into 4 test sets

preview	(162 LPs, 145 MB)	quick testing
standard	(282 LPs, 260 MB)	plots
heavy	(852 LPs, 785 MB)	statistics
high	(50LPs, 3800 MB)	correctness & profiling

Performance

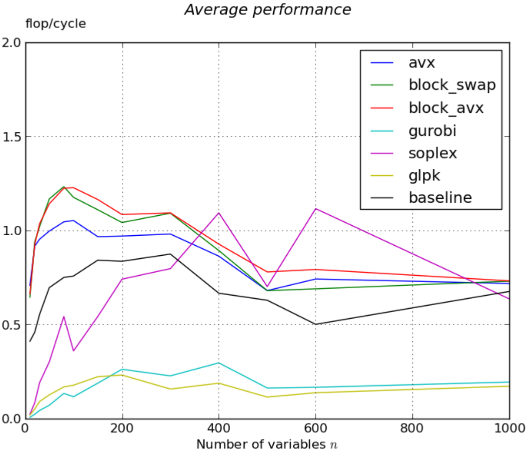

Figure: performance comparison

Wall Time

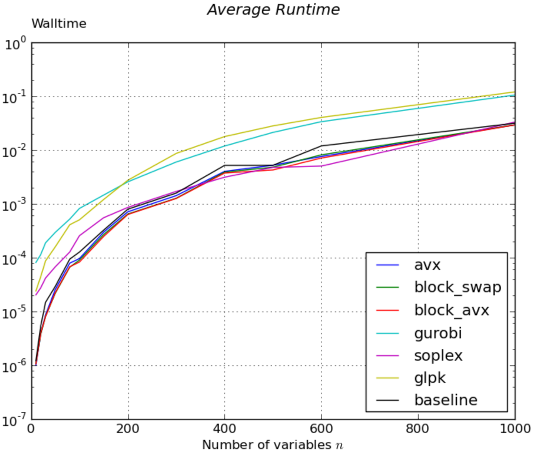

Figure: wall time comparison

Roofline

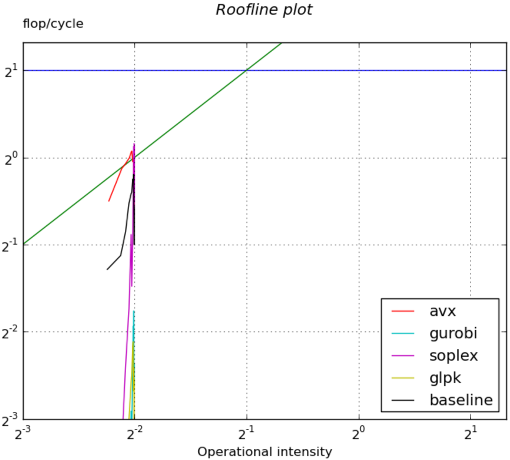

Figure: roofline

Profiling (gprof)

The main work routine has 24 memory access lines (12 shown):

```
  %    sec    line

3.97   0.67   block_swap.hpp:122
4.81   0.81   block_swap.hpp:123
4.05   0.68   block_swap.hpp:124
4.53   0.76   block_swap.hpp:125


3.37   0.57   block_swap.hpp:128
4.00   0.67   block_swap.hpp:129
3.94   0.66   block_swap.hpp:130
4.47   0.75   block_swap.hpp:131


4.41   0.74   block_swap.hpp:139
5.07   0.85   block_swap.hpp:140
3.46   0.58   block_swap.hpp:141
2.98   0.50   block_swap.hpp:142
```

```
src/simplex/block_swap.hpp

122    T r1 = tabp[m*width+j];
123    T r2 = tabp[m*width+j+1];
124    T r3 = tabp[m*width+j+2];
125    T r4 = tabp[m*width+j+3];


128    T la1 = tabp[i*width+j];
129    T la2 = tabp[i*width+j+1];
130    T la3 = tabp[i*width+j+2];
131    T la4 = tabp[i*width+j+3];


139    tabp[i*width+j]   = pa1;
140    tabp[i*width+j+1] = pa2;
141    tabp[i*width+j+2] = pa3;
142    tabp[i*width+j+3] = pa4;
```

Profiling (valgrind)

Performance counters on a 1000 variables run:

	float add&mul	memory access
theoretical estimate	100'900'800	50'450'400
perf counters	100'825'000	50'704'015
perf counters for SSA	100'825'000	126'304'015
cachegrind profile	100'836'841	126'669'074

Profiling (perf)

Annotated perf recording on a 1000 variables run:

```
                               T pa1 = la1 - fac1*r1;
                               T pa2 = la2 - fac1*r2;
 1.95 |      vmulsd %xmm5,%xmm0,%xmm10
 1.46 |      vmovsd (%rax),%xmm12
                               //~ PERFC_MEM += 4;
                               T r1 = tabp[m*width+j];
                               T r2 = tabp[m*width+j+1];
                               T r3 = tabp[m*width+j+2];
                               T r4 = tabp[m*width+j+3];
16.59 |      vmovsd 0x10(%rcx),%xmm3
```