Integer Linear Programming, Frobenius Instances, and Frobenius Numbers

Daniel Lichtblau

Wolfram Research, Inc.
danl@wolfram.com
January 9, 2008

Abstract

I will show how lattice reduction and branch−and−bound methods may be used in tandem to solve Frobenius instance problems. We apply much the same methods to other aspects of finding Frobenius numbers. Moreover the instance solver can be used to give good (as in tight, with high probability) bounds on the Frobenius number, in many cases where the latter computation is intractable with current methods.

Part 1: Review integer linear programming and Frobenius instance solving

Part 2: Show further applications to Frobenius number problem

Introduction

The Frobenius Instance problem

One is given a set of nonnegative integers $A = \{a_1, \ ..., \ a_n\}$ and a target value k. We wish to know when k can be written as a nonnegative integer combination of the ele−ments of A (if we relax the nonnegativity constraint on the multipliers, then clearly it can be done iff k is a multiple of the GCD of A). This problem goes by other names, among them the postage stamp problem (given several stamp denominations, how to obtain a given total value?) and the change making problem (same idea).

The common feature to this and related problems will be that we take restricted inte−ger combinations of integer vectors to fulfill some task. That is to say, we work with sets of constrained linear diophantine equations.

The Frobenius number problem

SImple to state. Given a set A as above, with stipulation that gcd is 1. Find the largest integer k such that k CANNOT be represented as a nonnegative combination of A.

Integer linear programming (ILP) with lattice reduction

The idea behind the integer linear programming

A blend of classical "branch−and−prune" and more recent lattice reduction methods in a way that surpasses what either alone can typically achieve.

A simple Frobenius instance example

We start with a simple example. We are given

In[37]:=
```
A = {12 223, 12 224, 36 674, 61 119, 85 569};
b = 39 999 425;
```

We wish to find a nonnegative integral combination of elements of A that sum to b.

Integer linear programming (ILP) with lattice reduction

Step 1: Allow negative integers

Our first step is to find an integer combination, waiving the nonnegativity restriction. We also find a basis for the integer null space of A. This is important because our problem will then be recast into finding a combination of null vectors to add to the solution, so that the final result is nonnegative.

The method we use for find ing a solution and null space is from a 1966 article by Blankenship.

In[39]:=
```
{soln, nulls} = systemSolve[{A}, {b}]
```

Out[39]=
```
{{0, 0, -2, 5945, -3778},
 {{0, -1, 1, 1, -1}, {1, 3, 1, 0, -1}, {-3, 1, -1, 1, 0}, {2059, 157, -3336, 2687, -806}}}
```

We see that $\{0, 0, -2, 5945, -3778\}.A == b$ and moreover we have our null space generators. We note that our solution vector has negative elements, hence we will need to do more work.

ILP with lattice reduction

Step 2: Find a "small" solution

Our next step is strictly for efficiency. We change our solution using a lattice reduc−tion known as the embedding method. The object is to get a solution vector with smaller and more evenly balanced entries.

In[40]:=
```
soln2 = smallSolution[soln, nulls]
```

Out[40]=
```
{-336, 10, 723, -298, 417}
```

ILP with lattice reduction

Step 3: Set up a linear programming problem to enforce nonnegativity

We now define a set of variables, one for each null space basis element. We then form a set of linear polynomials.

In[49]:=
```
vars = Array[x, Length[nulls]];
linpolys = soln2 + vars.nulls
```

Out[50]=
```
{-336 + x[2] - 3 x[3] + 2059 x[4], 10 - x[1] + 3 x[2] + x[3] + 157 x[4],
 723 + x[1] + x[2] - x[3] - 3336 x[4], -298 + x[1] + x[3] + 2687 x[4], 417 - x[1] - x[2] - 806 x[4]}
```

We will have a constraint satisfaction problem, to wit, that each element of the above sum to a nonnegative integer. So how do we intend to obtain this? We set up a branching loop that uses ordinary linear programming to enforce nonnegativity, and branching to obtain integrality.

ILP with lattice reduction

Step 3 small print

More specifically, LP is used to solve "relaxed" problems, now over the reals, wherein we allow noninteger values but now enforce inequalities. We choose a noninteger solution vector component x on which to "branch". To this end we set a pair of inequalities forcing the value to be less−or−equal to `Floor`[*x*] or greater−or−equal to `Ceiling`[*x*]. This gives a pair of subproblems (we can put these on a stack, queue, or priority queue). Any time we have a relaxed solution with all integer components it is in fact a solution to the ILP. Eventually subproblem solutions either have all integer components or are empty (when we cannot satisfy the constraints).

ILP with lattice reduction

Step 3 details

"Integer programming with a fixed number of variables" (H. Lenstra, 1983) shows how to solve these problems in polynomial time once dimension is fixed.

Subsequent work, in particular by Aardal, Hurkins, and A. Lenstra (2000) with further refinement found in Aardal and A. Lenstra (2002), use lattice reduction to cast the ideas into algorithmic form.

First improvement

Make the directions as close to orthogonal as possible. This is done in practice by a lattice reduction step (LLL algorithm) on the basis of null vectors.

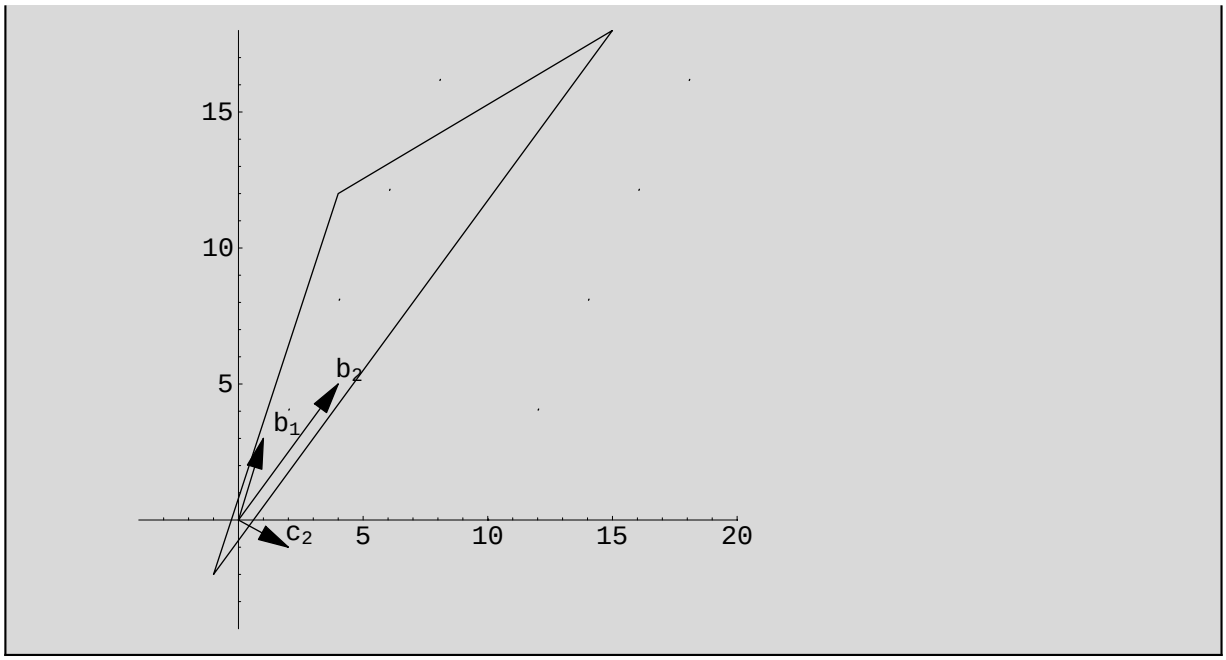
Next improvement: branch−on−largest

Recall form our previous example that the last null vector had components substantially larger than those of the other null vectors (and they came from a reduced basis for the null space lattice). What those Aardal et al papers show is that one can help the search considerably by appropriate choice of the branching variable. In particular we want to choose the direction corresponding to the largest basis element. Why? Because, in some sense, the constrained search space polytope is "thin" in that direction. That is, we expect to encounter fewer hyperplanes with that particular variable set to integer values. Hence we might hope to more quickly exhaust the search space, rather than meandering through it by naive choice of branching variable.

ILP branching strategy

The picture may give some idea of that first improvement. The "bad" directions $\{b_1, b_2\}$ can cause us to wander in the triangle (imagine it to be longer but not wider), whereas we can quickly learn that not many integer multiples of the good direction c_2 will stay inside it.

```
eps = .5;
Show[Graphics[{{Dashing[{.002, .2}], Table[Line[{{x, 0}, {x + 10, 20}}], {x, 0, 23, 10}]},
    Arrow[{0, 0}, {1, 3}], Arrow[{0, 0}, {4, 5}], Arrow[{0, 0}, {2, -1}],
    Line[{{-1, -2}, {15, 18}, {4, 12}, {-1, -2}}],
    Text[Subscript[b, 1], {1 + eps, 3} + eps], Text[Subscript[b, 2], {4 + eps, 5 + eps}],
    Text[Subscript[c, 2], {2 + eps, -1 + eps}]
    (*,Table[Point[{x,y}],{x,0,20,2},{y,0,20,2}]*)}, Axes → True],
  PlotRange → {{-4, 20}, {-4, 18}}, AspectRatio → 1];
```



Frobenius instance code

The code

In[1]:=
```
SetSystemOptions["LatticeReduceOptions" → {"LatticeReduceRatioParameter" → .999}];
```

In[2]:=
```
systemSolve[(mat_)?MatrixQ, (rhs_)?VectorQ] :=
 Module[{newmat, modrows, hnf, j = 1, len = Length[mat], zeros, solvec, nullvecs},
   newmat = Prepend[Transpose[mat], rhs];
   newmat = Transpose[Join[Transpose[newmat], IdentityMatrix[Length[newmat]]]];
   hnf = HermiteDecomposition[newmat][[2]];
   zeros = Table[0, {len}];
   While[j ≤ Length[hnf] && Take[hnf[[j]], len] =!= zeros, j++];
   solvec = Drop[hnf[[j]], len + 1] / (-hnf[[j, len + 1]]);
   nullvecs = (Drop[#1, len + 1] &) /@ Drop[hnf, j];
   nullvecs = LatticeReduce[nullvecs];
   {solvec, nullvecs}] /; Length[rhs] == Length[mat]
```

In[3]:=
```
smallSolution[(sol_)?VectorQ, (nulls_)?MatrixQ] :=
 Module[{max, dim = Length[nulls] + 1, weight, auglat, lat, k, soln}, lat = Prepend[nulls, sol];
  max = Max[Flatten[Abs[lat]]];
  weight = dim * max^2;
  auglat = (Prepend[#1, 0] &) /@ lat;
  auglat[[1, 1]] = weight;
  lat = LatticeReduce[auglat];
  For[k = 1, lat[[k, 1]] == 0, k++];
  soln = lat[[k]];
  Which[soln[[1]] == weight, Drop[soln, 1], soln[[1]] == -weight, -Drop[soln, 1], True, sol]]
```

In[9]:=

```
FrobeniusInstance[lhs_, rhs_] :=
 Module[{soln, nulls, vars, vals, var, x, obj, program, stack, iter = 0, lpresult, badvar, minval, val,
   signs, ineqle, ineqge, ineq, len, sign, origconstraints, constraints, ctoo, oldctoo = {}},
  {soln, nulls} = systemSolve[{lhs}, {rhs}];
  nulls = LatticeReduce[nulls];
  soln = smallSolution[soln, nulls];
  len = Length[nulls];
  vars = Array[x, len];
  vals = soln + vars.nulls;
  soln = {};
  origconstraints = Thread[Floor[rhs / lhs] ≥ vals ≥ 0];
  constraints = origconstraints;
  signs = Table[Equal, {Length[vars]}];
  Do[ctoo = Table[Quiet[lpresult = Minimize[{x[j], constraints}, vars]];
     If[Head[lpresult] == Minimize || ! FreeQ[lpresult, Indeterminate], Return[{{}}, Module]];
     ineqge = Ceiling[lpresult[[1]]];
     Quiet[lpresult = Minimize[{-x[j], constraints}, vars]];
     If[Head[lpresult] == Minimize || ! FreeQ[lpresult, Indeterminate], Return[{{}}, Module]];
     ineqle = Floor[-lpresult[[1]]];
     If[ineqge < ineqle, ineqge ≤ x[j] ≤ ineqle,
      If[ineqge == ineqle, x[j] == ineqge, Return[{{}}, Module]]], {j, len}];
   If[ctoo === oldctoo, Break[], constraints = Join[origconstraints, ctoo]; oldctoo = ctoo];, {4}];
  stack = {{signs, constraints}, {}};
  While[stack =!= {}, iter++;
   program = stack[[1]];
   stack = stack[[2]];
   signs = First[program];
   program = program[[2]];
   sign = 2 * Random[Integer] - 1;
   obj = x[Random[Integer, {1, len}]];
   Quiet[lpresult = Minimize[{sign * obj, program}, vars]];
   If[Head[lpresult] == Minimize || ! FreeQ[vals, Indeterminate], Continue[]];
   minval = First[lpresult];
   If[Abs[minval] === Infinity, Continue[]];
   lpresult = lpresult[[2]];
   badvar = Position[Reverse[vars /. lpresult], (a_ /; ! IntegerQ[a]), {1}, 1, Heads → False];
   If[badvar == {}, Return[{vals /. lpresult}, Module]];
   badvar = badvar[[1, 1]];
   var = Reverse[vars][[badvar]];
   val = var /. lpresult;
   minval = If[sign === 1, Ceiling[minval], Floor[-minval]];
   ineq = If[sign === 1, GreaterEqual, LessEqual];
   ineqle = {var ≤ Floor[val], ineq[obj, minval]};
   ineqge = {var ≥ Ceiling[val], ineq[obj, minval]};
   ineq = signs[[badvar]];
   Which[ineq === Equal || Head[ineq] === LessEqual, signs[[badvar]] = ineqle[[1]];
    stack = {{signs, Join[program, ineqle]}, stack};
    signs[[badvar]] = ineqge[[1]];
    stack = {{signs, Join[program, ineqge]}, stack},
    True (*Head[ineq]==GreaterEqual*), signs[[badvar]] = ineqge[[1]];
    stack = {{signs, Join[program, ineqge]}, stack};
    signs[[badvar]] = ineqle[[1]];
    stack = {{signs, Join[program, ineqle]}, stack}];];
  soln]
```

Frobenius instance examples

We now tackle Frobenius problems that had been out of reach. Here is one such example. We exhaust the search space quickly even when the solution set is empty.

Example 1

In[10]:=
```
Timing[ff = FrobeniusInstance[{10 000 000 000, 35 550 333 797, 42 807 347 507, 55 224 372 861
    67 932 625 959, 75 136 205 917, 79 022 523 667, 80 463 866 750, 7 631 346 246 323}]]
```

Out[10]=
```
{1.39609, {}}
```

Example 2

We handle fairly large examples in what I would consider to be reasonable time.

In[8]:=
```
Timing[ff = FrobeniusInstance[{10 000 000 000, 10 451 674 296, 18 543 816 066, 27 129 592 681,
    27 275 963 647, 29 754 323 979, 31 437 595 145, 34 219 677 075, 36 727 009 883, 43 226 644 83
    47 122 613 303, 57 481 379 652, 73 514 433 751, 74 355 454 078, 78 522 678 316, 86 905 143 02
    89 114 826 334, 91 314 621 669, 92 498 011 383, 93 095 723 941, 1 250 976 029 960}]]
```

Out[8]=
```
{158.738, {{7, 1, 16, 2, 0, 2, 2, 3, 2, 0, 2, 0, 0, 1, 1, 0, 0, 3, 0, 0}}}
```

Finding Frobenius numbers

Use of integer linear programming to compute Frobenius numbers

The gist of the algorithm is to use ILPs to find certain sets containing elbows, then use a method to go from elbows to corners. We finish when we have the furthest corner. While pathological cases (too many corners) arise even at $n = 4$, the average case performance (e.g. random examples) is quite nice. The ILPs in question are similar to those used in solving Frobenius instances, hence the relevance of the methods we have discussed.

Heuristic usage of ILP

The joint paper ELSW discusses a shortcut that frequently gives $g(A)$. The idea is to find a subset of elbows, hence have "corners" that might be too large. The furthest is our candidate for g(A). If it is too large, then using it in Frobenius instance solving will tell us that. If the instance solver fails, then it must be the correct value for $g(A)$.

Estimating Frobenius numbers

An ILP tactic for approximating the Frobenius number

Note that even when the problem is too large to find the Frobenius number in reasonable time, we might still be able to solve sufficiently many relevant ILPs to obtain good lower and upper bounds. We have done this for n as large as 21, with 11 digit numbers, bracketing the Frobenius number by a factor of 100 or so. This is far better than a priori bounds can accomplish (if all elements are O(k) then these bounds are around k and k^2 respectively, whereas typically the Frobenius number is around $k^{\frac{n}{n-1}}$). The idea here is to find the full set of what we call "axial elbows".

Estimating Frobenius numbers

A stronger ILP tactic for approximating the Frobenius number

Proposition: Given $A = \{a_1, \ ..., \ a_n\}$, with $\gcd(A) = 1$, and call the Frobenius number $g(A)$. Let $N(A)$ denote the positive integers less or equal to $g(A)$ for which the Frobenius instance problem cannot be solved.

Proposition: $|N(A)| \geq g(A)/2$.

Proof: Take $a \leq g(A)$. Then either $a \in N(A)$ or $g(A) - a \in N(A)$ (else the sum, $g(A)$, would have a nonnegative representation, contradicting definition of the Frobenius number). Note that both could be in $N(A)$; this is not a mutually exclusive situation. Upshot: By pairing off in this way, at least half the values less than $g(A)$ are in $N(A)$.

Observation

In practice, as we get close to $g(A)$ most instance problems CAN be solved. Thus with probability somewhat greater than 1/2, values less than half $g(A)$ will be in $N(A)$.

How to use this

Find a reasonable but low starting value (use the BHNW heuristic lower bound as a guide). Try to solve the instance problem. If we cannot solve it, increase the target size by some percentage. Else increase target by one. If we continually solve it then we are almost certainly near or above $g(A)$. In practice this approach seems able to get the first two or three bits of $g(A)$, at least in examples for which we can independently assess this.

Tight bounds: iterating the Frobenius solver

Code

Here we show some code and examples, using the ideas above.

In[11]:=
```
bhnwLowBound[aa_List] :=
 Floor[(.5 * Gamma[Length[aa] + 1.] * Apply[Times, aa])^(1 / Length[aa])]
```

In[29]:=
```
estimateFrobenius[aa_List, mult_: 1.1, consec_: 16] := Module[
  {b = Round[.8 * bhnwLowBound[aa]], j, done = False, lastb},
  While[! done,
   done = True;
   For[j = 0, j <= consec, j++,
    soln = FrobeniusSolve[aa, b + j, 1];
    If[soln === {}, j = consec; lastb = b; b = Round[b * mult]; done = False];
   ]];
  lastb
 ]
```

Example1

In[24]:=
```
SeedRandom[1 112 223];
aa1 = RandomInteger[10^10, {8}]
```

Out[25]=
```
{1 505 302 742, 5 572 572 328, 29 510 273, 9 279 374 671,
 1 956 902 370, 1 982 829 533, 995 048 914, 7 902 515 103}
```

In[26]:=
```
bhnwLowBound[aa1]
```

Out[26]=
```
5 871 726 142
```

In[30]:=
```
Timing[ef = estimateFrobenius[aa1]]
```

Out[30]=
```
{16.469, 132 007 850 511}
```

In[22]:=
```
Timing[FrobeniusNumber[aa1]]
```

Out[22]=
```
{51.7432, 164 459 821 067}
```

Not such a great time savings. But we can use this method on sets where a `FrobeniusNumber` computation is not feasible.

Example2

In[31]:=
```
aa2 = RandomInteger[10^10, {18}]
```

Out[31]=
```
{8 580 842 616, 5 707 446 915, 266 469 396, 6 635 059 441, 8 650 981 767, 4 330 092 700,
 2 621 441 038, 1 806 347 720, 7 614 852 913, 7 496 868 713, 6 082 031 401, 1 258 238 091,
 342 253 107, 1 587 216 691, 8 320 388 772, 7 824 471 461, 725 756 237, 4 033 282 577}
```

In[33]:=
```
Timing[ef = estimateFrobenius[aa2]]
```

Out[33]=
```
{3047.15, 46 733 841 844}
```

Summary

- Lattice reduction methods are quite useful for solving linear diophantine problems.
- Branch−and−prune methods remain a viable method for handling constrained lin−ear diophantine systems. The needed adaptations are
  (1) They be used in conjunction with reduced lattice input.
  (2) Branching choices must be based on size considerations a la Aardal & Lenstra.
- These methods tend to work well on Frobenius instance problems and the related ILPs one encounters in computation of Frobenius numbers (in the ELSW implemen−tation).
- Simple iteration of a Frobenius instance solver can give a lower bound estimate of the Frobenius number that is typically about 3/4 of the actual value
  (1) The actual ratio depends to an extent on input parameters that define the itera−tion.
  (2) Simple theory shows that, with VERY high probability, it is not worse than 1/2.

Selected bibliography

K. Aardal, C. A. J. Hurkens, and A. K. Lenstra. Solving a system of linear diophantine equations with lower and upper bounds on the variables. *Mathematics of Operations Research* **25**:427−442, 2000.

K. Aardal and A. K. Lenstra. Hard equality constrained knapsacks. In *Proceedings of the 9th Conference on Integer Programming and Combinatorial Optimization* (IPCO 2002), W. J. Cook and A. S. Schulz, eds. Lecture Notes in Computer Science 2337, 350−366. Springer−Verlag, 2002.

K. Aardal, R. Weismantel, and L. A. Wolsey. Non−standard approaches to integer programming. *Discrete Applied Mathematics* **123**:5−74, 2002.

D. Beihoffer, J. Hendry, A. Nijenhuis, and S. Wagon. Faster algorithms for Frobenius numbers. *Electronic Journal of Combinatorics* **12**, 2005.

W. A. Blankenship. Algorithm 288: Solution of simultaneous linear diophantine equations. *Communications of the ACM* **9**(7):514, 1966.

G. Dantzig. *Linear Programming and Extensions*. Princeton Landmarks in Mathematics and Physics. Princeton University Press, 1963 (Reprinted 1998).

D. Einstein, D. Lichtblau, A. Strzebonski, and S. Wagon. Frobenius numbers by lattice enumeration. *INTEGERS: The Electronic Journal of Combinatorial Number Theory* (2007).
http://www.integers−ejcnt.org/vol7.html

A. Lenstra, H. Lenstra, and L. Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen* **261**:515−534, 1982.

H. Lenstra. Integer programming in a fixed number of variables. Math. Oper. Res.8:538−548, 1983.

D. Lichtblau. Making change and finding repfigits: balancing a knapsack. In: Proceedings of the Second International Congress on Mathematical Software (ICMS 2006), Lecture Notes in Computer Science 4151 182−193. Springer, 2006.

C. P. Schnorr and M. Euchner. Lattice basis reduction: improved practical algorithms and solving subset sum problems. In *Proceedings of the 8th International Conference on Fundamentals of Computation Theory*, 1991. L. Budach, ed. Lecture Notes in Computer Science 529, 68−85. Springer−Verlag, 1991.

A. Schrijver. *Theory of Linear and Integer Programming*. Wiley−Interscience Series in Discrete Mathematics and Optimization, 1986.