Lattice Reduction, Integer Programming, and Knapsacks

Daniel Lichtblau
danl@wolfram.com

Wolfram Research, Inc.
100 Trade Centre Dr.
Champaign IL USA, 61820

ICMS, Castro Urdiales, Spain
September 1−3, 2006

Abstract

We will discuss knapsack problems that arise in certain computa−tional number theory settings. A common theme is that the search space for the standard real relaxation is large; in a sense this trans−lates to a poor choice of variables. Lattice reduction methods have been developed in the past few years to improve handling of such problems. We show explicitly how they may be applied to computa−tion of Frobenius instances, Keith numbers (also called "repfigits"), and as a first step in computation of Frobenius numbers.

Introduction

Informal description of a knapsack problem

- **One has a set of items.**
- **One must select from it a subset that fulfills specified criteria.**

Introduction

Informal description of a knapsack problem

- ◇ **One has a set of items.**
- ◇ **One must select from it a subset that fulfills specified criteria.**

Other problems under consideration in this talk

- ◇ **Find Keith numbers (first we must define them...)**
- ◇ **Solve the "Frobenius instance" problem (aka "postage stamp" or "change−making" problem).**
- ◇ **Compute the Frobenius number for a given set of nonnegative integers whose GCD is unity.**

The common feature to these problems will be that we take restricted integer combinations of integer vectors to fulfill some task. That is to say, we work with sets of constrained linear diophantine equations. This is really just a mild generalization of the classical subset sum problem.

Introduction

Informal description of a knapsack problem

- ◇ **One has a set of items.**
- ◇ **One must select from it a subset that fulfills specified criteria.**

Other problems under consideration in this talk

- ◇ **Find Keith numbers (first we must define them...)**
- ◇ **Solve the "Frobenius instance" problem (aka "postage stamp" or "change−making" problem).**
- ◇ **Compute the Frobenius number for a given set of nonnegative integers whose GCD is unity.**

The common feature to these problems will be that we take restricted integer combinations of integer vectors to fulfill some task. That is to say, we work with sets of constrained linear diophantine equations. This is really just a mild generalization of the classical subset sum problem.

The idea

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

Frobenius instances and Frobenius numbers

We mention in brief Frobenius instances and numbers.

Frobenius instances and Frobenius numbers

We mention in brief Frobenius instances and numbers.

Statement of the Frobenius instance problem

Given a set of n positive integers in increasing order $A = \{a_1, \ldots, a_n\}$, we wish to know when a given positive integer k can be written as a nonnegative integer combination of the elements 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?), the chicken McNugget problem (how to get k McNuggets from orders of size 6, 9, or 20?), and the change mak−ing problem (for equally obvious reasons).

Frobenius instances and Frobenius numbers

We mention in brief Frobenius instances and numbers.

Statement of the Frobenius instance problem

Given a set of n positive integers in increasing order $A = \{a_1, ..., a_n\}$, we wish to know when a given positive integer k can be written as a nonnegative integer combination of the elements 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?), the chicken McNugget problem (how to get k McNuggets from orders of size 6, 9, or 20?), and the change making problem (for equally obvious reasons).

Frobenius numbers

Given a set A as above, with the added restriction that it have GCD of unity, it is not hard to show that we can always represent sufficiently large n as a nonnegative integer combination of A. The largest integer that CANNOT be so represented is called the Frobenius number of A.

Keith numbers

Now we describe a different sort of problem. Well, it’s sort of different. That is to say, it looks different. Sort of.

Around 1987 Michael Keith first wrote about what he called **repfigits** (for "replicating Fibonacci digits"). These are sequences of numbers that, when one progressively addsthem to form new elements in the style of the Fibonacci sequence, one obtains the number whose digits began the sequence. For example we take the number 197. We thus start with $\{1, 9, 7\}$. Adding these to get the next element gives $\{1, 9, 7, 17\}$. Adding the preceding thee digits to get the next one gives $\{1, 9, 7, 17, 33\}$. If we continue this we get $\{1, 9, 7, 17, 33, 57, 107, 197, \ldots\}$. Because our number, 197, appears in the sequence originating with its digits, it is a repfigit. Others later began referring to these as Keith numbers in honor of the first to discuss them.

Question: Why do we care about these numbers?

Keith numbers

Answer: Beyond the moderate interest of their definition, we want to find them because the solution techniques are applicable to other problems. Like Frobenius instances/numbers, as we'll see presently.

A property I find of interest for repfigits is that they are not exactly easy to find. By 1994 all and only the ones up to 15 digits were known. In 1997 Michael Keith found all of them through 19 digits, using a clever search algorithm and about 500 hours of computer time. There are 71 such numbers. No larger ones were known (an unfortunate situation that persisted until 2004).

As it happens, one can write Keith number problems as integer linear programs. The digits of a Keith number of a given size will have to satisfy one of a few readily computable linear diophantine equations, subject to the obvious inequalities that force them to actually be digits (that is, they must be between 0 and 9 inclusive, with the first being at least 1).

Keith numbers

In order to find such numbers we need to describe the possible equations they might satisfy. The code to do this is concise but in fact took me over two hours to write. I have an explanation for how it works written down somewhere but I doubt I will ever again understand it myself.

```
keithEquations[len_Integer /; len > 0] := Module[
  {matrow, n, list, res, vecs}, res = list[];
  Do[matrow[j] = Table[KroneckerDelta[k, j + 1],
     {k, len}], {j, len - 1}];
  matrow[len] = Table[1, {len}];
  n = len;
  While[9 * Apply[Plus, matrow[n]] <
    10 ^ (len - 1), n++;
   matrow[n] = Sum[matrow[k],
     {k, n - len, n - 1}];];
  While[First[matrow[n]] ≤ 10 ^ (len - 1),
   res = list[res, matrow[n]];
   n++;
   matrow[n] =
    Sum[matrow[k], {k, n - len, n - 1}];];
  vecs = Apply[List, Flatten[
     res, Infinity, list]];
  Map[(# - 10 ^ Range[len - 1, 0, -1]) &, vecs]]
```

Simple example finding a Keith number

Quick example:

```
k6 = keithEquations[6];
```

We'll work with the second possible equation for 6 digit Keith numbers.

```
k6[[2]]
{-96160, -4224, 5752, 7144, 7482, 7616}
```

The equation this represents is:

$$-96160\, d_0 - 4224\, d_1 + 5752\, d_2 + 7144\, d_3 + 7482\, d_4 + 7616\, d_5 == 0$$

Simple example finding a Keith number

These equations each give rise to integer null space computations. This is easy to handle e.g. with the Hermite normal form of a matrix.

```
integerNullSpace[vec : {_Integer ..}] :=
 Module[{mat, hnf}, mat = Transpose[Join[
     {vec}, IdentityMatrix[Length[vec]]]];
  hnf = Last[Developer‘HermiteNormalForm[
     mat]];
  LatticeReduce[Map[Drop[#, 1] &,
    Drop[hnf, 1]]]]
```

Simple example finding a Keith number

These equations each give rise to integer null space computations. This is easy to handle e.g. with the Hermite normal form of a matrix.

```
integerNullSpace[vec : {_Integer ..}] :=
 Module[{mat, hnf}, mat = Transpose[Join[
     {vec}, IdentityMatrix[Length[vec]]]];
  hnf = Last[Developer`HermiteNormalForm[
     mat]];
  LatticeReduce[Map[Drop[#, 1] &,
    Drop[hnf, 1]]]]
```

With this we now find a set of vectors that generates the solution space for our example.

```
nulls[6, 2] = integerNullSpace[k6[[2]]]
```

```
{{0, 3, -3, 0, 4, 0},
 {-1, 1, -3, -2, -4, -4}, {0, -6, -3, 1, 0, -2},
 {0, -1, 1, 5, 0, -6}, {0, 4, -12, 15, -12, 9}}
```

We arrive at a set of small null vectors. This is referred to as solving a relaxation over the integers (that is, we insist of integer solutions but do not constrain the values by inequalities e.g. nonnegativity). We observe that none of these vectors has entirely nonnegative or entirely nonpositive values. Hence none gives a Keith number. We need to take integer combinations of these in order to get a Keith number, assuming the equation we began with will yield any (it does).

Simple example finding a Keith number

The idea: For each null vector we create an integer−valued variable. Allow arbitrary linear combinations of these vectors subject to the constraints that all resulting components be nonnegative, and the first be positive. This is simply a constraint satisfaction problem. We use a handy optimizer and we are off to the races.

```
keithSolution[nulls_, iters_: Automatic] :=
 Module[{len = Length[nulls], vars, x,
   vec, constraints, program, min, vals},
  vars = Array[x, len];
  vec = vars.nulls;
  constraints = Join[{Element[vars, Integers],
     1 ≤ First[vec] ≤ 9},
    Map[0 ≤ # ≤ 9 &, Rest[vec]]];
  program = {1, constraints};
  {min, vals} = NMinimize[program,
    vars, MaxIterations → iters];
  vec /. vals]
```

```
keithSolution[nulls[6, 2]]
```

```
{1, 4, 7, 6, 4, 0}
```

One can check that 147 640 is in fact a repfigit.

This method of finding such numbers using a black−box optimizer is a bit lame. But it illustrates the basic idea, which was to form a con− straint satisfaction problem using integer combinations of a lattice− reduced set of generators. So what remains is to do this in a system− atic and efficient, manner.

Big Keith numbers

This is an integer linear program (ILP) and more specifically, an inte−ger constraint satisfaction problem (that is, we have no objective function to optimize). We will use a branching method coupled to ordinary LP code. Here is the idea. We solve "relaxed" problems, now over the reals, wherein we allow noninteger values but now enforce inequalities. We choose a noninteger solution vector compo−nent 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. To get all solutions we spawn further problems that force at least one component to have a different value from the solution we found. Eventually subproblem solutions either have all integer com−ponents or are empty (when we cannot satisfy the constraints).

There are numerous technical details for purposes of efficiency. I do not plan to discuss most of them as they are variants of classical ideas such as cutting planes. One, however, is not (yet) standard and is quite relevant to this talk.

Big Keith numbers

First we will see the code. Actually this is quite unlikely, because I have the font size reduced, but I want to show that it can fit comfort-ably on two pages or less.

```
keithSolutions[onulls_] :=
 Module[{nulls, vars, x, len = Length[onulls], vecs, constraints, program,
   stack, soln, solns = {}, badvar, varvals, val, counter = 1, var, extra,
   maxs, mins, ctmp = {}, bad = False, vnum, vval, eps = 10^(-5), rndvar},
  nulls = Reverse[onulls[[Ordering[Map[Norm, N[onulls]]]]]];
  vars = Array[x, len]; vecs = vars.nulls;
  constraints = Join[{1 ≤ First[vecs] ≤ 9}, Map[0 ≤ # ≤ 9 &, Rest[vecs]]];
  Do[mins = Table[
      Internal‘DeactivateMessages[val = NMinimize[{vars[[j]], Join[constraints, ctmp]}, vars];
       If[Head[val] === NMinimize || ! FreeQ[val, Indeterminate], bad = True; Break[]];
       val = First[val], NMinimize::"nsol"], {j, len}];
   maxs = Table[
      Internal‘DeactivateMessages[val = NMaximize[{vars[[j]], Join[constraints, ctmp]}, vars];
       If[Head[val] === NMaximize || ! FreeQ[val, Indeterminate], bad = True; Break[]];
       val = First[val], NMaximize::"nsol"], {j, len}];
   ctmp = Join[Thread[LessEqual[vars, Floor[maxs + eps]]],
      Thread[GreaterEqual[vars, Ceiling[mins - eps]]]];
   , {4}];
  If[bad, Return[{counter, {}}]];
  constraints = Join[constraints, ctmp];
  program = constraints; stack = {program, {}};
  While[stack =!= {}, counter++; program = stack[[1]]; stack = stack[[2]];
   rndvar = vars[[Random[Integer, {1, len}]]]; program = {rndvar, program};
   Internal‘DeactivateMessages[vals = NMinimize[program, vars], NMinimize::"nsol"];
   If[Head[vals] == NMinimize, Continue[]];
   vval = Ceiling[First[vals] - eps]; vals = Chop[vals[[2]]]; soln = Chop[vecs /. vals];
   If[! FreeQ[soln, Indeterminate], Continue[]];
   constraints = program[[2]]; varvals = vars /. vals;
   badvar = Position[varvals, (a_ /; Chop[a - Round[a]] =!= 0), {1}, 1, Heads → False];
   If[badvar == {}, soln = Round[soln]; solns = {soln, solns};
    Do[extra = Table[vecs[[k]] == soln[[k]], {k, j - 1}];
     stack = {Join[constraints, Append[extra, vecs[[j]] ≤ soln[[j]] - 1]], stack};
     stack = {Join[constraints, Append[extra, vecs[[j]] ≥ soln[[j]] + 1]], stack};,
     {j, Length[soln]}];
    Continue[]];
   badvar = badvar[[1, 1]]; var = vars[[badvar]]; val = var /. vals;
   stack = {Join[constraints, {rndvar ≥ vval, var ≤ Floor[val]}], stack};
   stack = {Join[constraints, {rndvar ≥ vval, var ≥ Ceiling[val]}], stack};];
  {counter, Partition[Flatten[solns], Length[First[nulls]]]}]
```

Big Keith numbers

Running the code above on all possible equation sets, I found all Keith numbers between 20 and 29 digits. Here are the ones in the range 24−28 digits.

```
{2, 2, 9, 1, 4, 6, 4, 1, 3, 1, 3,
 6, 5, 8, 5, 5, 5, 8, 4, 6, 1, 2, 2, 7}
{9, 8, 3, 8, 6, 7, 8, 6, 8, 7, 9, 1, 5,
 1, 9, 8, 5, 9, 9, 2, 0, 0, 6, 0, 4}
{1, 8, 3, 5, 4, 9, 7, 2, 5, 8, 5, 2, 2,
 5, 3, 5, 8, 0, 6, 7, 7, 1, 8, 2, 6, 6}
{1, 9, 8, 7, 6, 2, 3, 4, 9, 2, 6, 4, 5,
 7, 2, 8, 8, 5, 1, 1, 9, 4, 7, 9, 4, 5}
{9, 8, 9, 3, 8, 1, 9, 1, 2, 1, 4, 2, 2,
 0, 7, 1, 8, 0, 5, 0, 3, 0, 1, 3, 1, 2}
{1, 5, 3, 6, 6, 9, 3, 5, 4, 4, 5, 5, 4, 8,
 2, 5, 6, 0, 9, 8, 7, 1, 7, 8, 3, 4, 2}
{1, 5, 4, 6, 7, 7, 8, 8, 1, 4, 0, 1, 0, 0,
 7, 7, 9, 9, 9, 7, 4, 5, 6, 4, 3, 3, 6}
{1, 3, 3, 1, 1, 8, 4, 1, 1, 1, 7, 4, 0, 5,
 9, 6, 8, 8, 3, 9, 1, 0, 4, 5, 9, 5, 5}
{1, 5, 4, 1, 4, 0, 2, 7, 5, 4, 2, 8, 3, 3,
 9, 9, 4, 9, 8, 9, 9, 9, 2, 2, 6, 5, 0}
{2, 9, 5, 7, 6, 8, 2, 3, 7, 3, 6, 1, 2, 9,
 1, 7, 0, 8, 6, 4, 5, 2, 2, 7, 4, 7, 4}
{9, 5, 6, 6, 3, 3, 7, 2, 0, 4, 6, 4, 1, 1,
 4, 5, 1, 5, 8, 9, 0, 3, 1, 8, 4, 1, 0}
{9, 8, 8, 2, 4, 2, 3, 1, 0, 3, 9, 3, 8, 6,
 0, 3, 9, 0, 0, 6, 6, 9, 1, 1, 4, 1, 4}
{9, 4, 9, 3, 9, 7, 6, 8, 4, 0, 3, 9, 0, 2,
 6, 5, 8, 6, 8, 5, 2, 2, 0, 6, 7, 2, 0, 0}
```

Big Keith numbers

One will observe that all of these begin with digits in {1,2,9} and end with digits in {0,2,4,5,6,7}. If these were in some way "random" and uniform we would believe the probability of this limited set of first digits occuring exclusively to be $\left(\frac{1}{3}\right)^{13}$, or about $\frac{1}{10^6}$. The probabil−ity of the last digits being so restricted is a tame one−in−a−thousand or so. This raises an obvious conjecture and a perhaps unusual question: Might there be something intrinsically "interesting" to Keith numbers that would keep them from being "random"?

Big Keith numbers

One will observe that all of these begin with digits in {1,2,9} and end with digits in {0,2,4,5,6,7}. If these were in some way "random" and uniform we would believe the probability of this limited set of first digits occuring exclusively to be $\left(\frac{1}{3}\right)^{13}$, or about $\frac{1}{10^6}$. The probabil−ity of the last digits being so restricted is a tame one−in−a−thousand or so. This raises an obvious conjecture and a perhaps unusual question: Might there be something intrinsically "interesting" to Keith numbers that would keep them from being "random"?

So I checked further. After a few computer−days I found the only two 29 digit Keith numbers.

```
{4, 1, 7, 9, 6, 2, 0, 5, 7, 6, 5, 1, 4, 7, 4,
 2, 6, 9, 7, 4, 7, 0, 4, 7, 9, 1, 5, 2, 8}
{7, 0, 2, 6, 7, 3, 7, 5, 5, 1, 0, 2, 0, 7, 8,
 8, 5, 2, 4, 2, 2, 1, 8, 8, 3, 7, 4, 0, 4}
```

Both violate the seemingly strong trend of first digits, and one vio−lates the last digit trend as well. So this becomes a much weaker conjecture that there might be a mathematical justification for "most" large Keith numbers to fit the trend patterns.

Frobenius instances

We return to the Frobenius problems.

Solving Frobenius instances

One quickly sees that this is the same sort of linear diophantine problem as was encountered in finding Keith numbers. So we would like to believe we can attack it similarly and have similar success. But wait...

Frobenius instances can be tricky

Beginning in 1983 with "Integer programming with a fixed number of variables" by H. Lenstra various papers have appeared that indicate how lattice bases might be utilized to improve ILP performance. Subsequent work, in particular by Aardal, Hurkins, and A. Lenstra (2000) with further refinement found in Aardal and A. Lenstra (2002), cast the ideas into algorithmic form.

What I did with Keith numbers is certainly in this vein, and in many respects quite similar, but with one shortcoming. During a visit Stan Wagon made to WRI in October 2004, he had me try out a pathological benchmark example from the Aardal & Lenstra paper. My code did a beautiful impression of an empty coat, which is to say, it hung. What I had was tolerable for the "average case", but not up to the task for bad cases.

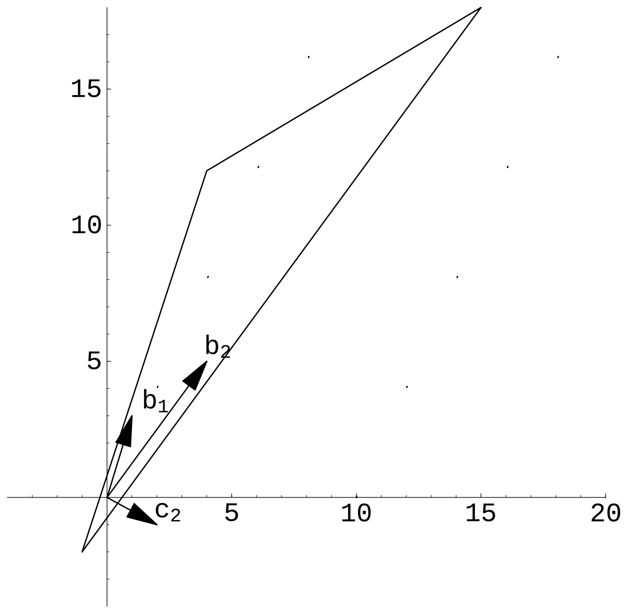
Frobenius instances can be tricky

What was missing, and what those papers clarified, 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 choosing a naive branching method. In the pathological examples one direction was much larger than the rest, and failing to heed their choice strategy consigned the algorithm to wander endlessly.

Why did this not appear when I worked with Keith numbers? It turns out that (1) the equations that yield them are not "pathological" and (2) it really did show up but in such a mild way that I did not notice. In making the appropriate algorithm change I did get a reasonable speed improvement on average (factor of 2 or so).

Frobenius instances can be tricky

The picture may give some idea. 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 multi− ples of the good direction c_2 will stay inside it.



Frobenius instances can be tricky

With this appropriate branching strategy, and the relevant lattice basis ordered by size, 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.

```
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]]
```

```
{0.99 Second, {}}
```

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

```
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, 43 226 644 830, 47 122 613 303
     57 481 379 652, 73 514 433 751,
     74 355 454 078, 78 522 678 316, 86 905 143 028
     89 114 826 334, 91 314 621 669, 92 498 011 383
     93 095 723 941}, 1 250 976 029 960]]
```

```
{161.66 Second, {2, 8, 3, 0, 9, 12, 0,
  4, 0, 2, 2, 0, 0, 0, 1, 0, 0, 0, 0, 1}}
```

The literature quite clearly shows that naive branching, even with an industry standard program such as CPLEX, simply cannot handle such problems. On the flip side, lattice methods alone do not suffice. What we require is a bona fide marriage of the two approaches.

Frobenius numbers

Computing Frobenius numbers

Sylvester gave an explicit value for $n = 2$ as $a_1 a_2 - a_1 - a_2$. A method was described by Brauer and Shockley in 1963 to handle sets of three elements. An efficient continued fraction method due independently to Greenberg and Davison appeared in 1989 and 1994 respectively. These have good complexity with regards to the size of the numbers.

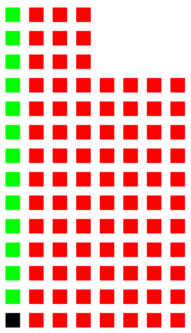
Several algorithms have arisen in the past 25 years that handle larger sets. For example there is a method based on a tree construction from circulant graph, due to Beihoffer, Hendry, Nijenhuis, and Wagon. Theirs can handle sets of, say, 100 elements. The drawback is that the smallest element a_1 must be no larger than 10 million or so (a residue table is computed comprising all values modulo a_1).

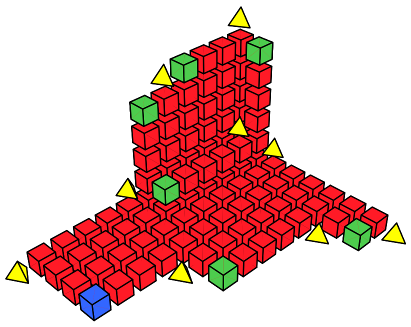
Stan Wagon and David Einstein then set about to develop a method that was not restricted to having a small element in the set, even if it meant limiting the cardinality of the set to, say, 10 or so. When Stan visited WRI in 2004 he discussed this with some of us. At the time they had made good progress on sets of 4 elements but were hitting walls after that. Well, they could attack n as large as 5 or 6, but with considerable programming resources.

Finding Frobenius numbers

For Frobenius number computations Wagon and Einstein defined a "fundamental domain". It has turning points we refer to as elbows. It has extremal points called corners. This region is one dimension smaller than the input set (everything is done modulo a_1), hence can be illustrated for the cases $n = 3$ and $n = 4$. With respect to these domains, the Frobenius number corresponds to the farthest corner from the origin, with distance an l_1 metric weighted by ele− ment sizes.

A reviewer kindly pointed out that these domains are closely related to "multiple loop networks", which also (typically) begin with a set of integers. There appears to be but small crossover in the literature (that is to say, there is similarity of methods but I've not found much overlap of authors). I suspect this is because one asks different ques- tions in these two realms.





Finding 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 or finding repfigits, hence the relevance of the methods we have discussed.

Here we indicate an example with 11 digit numbers from sets of 7 and 8 elements respectively. To the best of my knowledge no other existing algorithm can compute these Frobenius numbers.

```
Timing[
 fnum = FrobeniusFEfficient[{10 000 000 000,
    18 543 816 066, 27 129 592 681, 27 275 963 647,
    43 226 644 830, 73 514 433 751, 74 355 454 078}]]
```

```
{28.04 Second, 7 688 309 842 406}
```

```
Timing[fnum = FrobeniusFEfficient[
   {10 000 000 000, 18 543 816 066, 27 129 592 681
    27 275 963 647, 43 226 644 830, 73 514 433 751
    74 355 454 078, 78 522 678 316}]]
```

```
{541.71 Second, 5 215 972 252 834}
```

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 $k^{\frac{n}{n-1}}$).

Summary

- Lattice methods are, as ever, quite useful for solving linear diophan−tine problems. This is true even in the presence of possibly unpleas−ant constraints.

Summary

- Lattice methods are, as ever, quite useful for solving linear diophan−tine problems. This is true even in the presence of possibly unpleas−ant constraints.
- Branch−and−prune methods remain a viable method for handling constrained linear 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.

Summary

- Lattice methods are, as ever, quite useful for solving linear diophantine problems. This is true even in the presence of possibly unpleasant constraints.
- Branch−and−prune methods remain a viable method for handling constrained linear 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.
- There are several possible refinements we did not discuss.
  (1) There are tactics for improving the underlying LP performance (e.g. use of dual simplex method which, I gather, is amenable to a "hot restart" when adding a new constraint to move away from a particular solution).
  (2) There are different lattice embeddings one might use in order to cull yet smaller solutions to the unconstrained solutions prior to the branch−and−prune phase.
  (3) Use of objective functions allows us to cheaply emulate cutting planes, thus tightening the constraints as we branch.
  (4) My use of cutting planes is best described as "hopelessly naive". Certainly improvements there might lead to substantially faster ILP solving overall.