Solving knapsack problems

Daniel Lichtblau

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

danl@wolfram.com

IMS 2004, Banff, Canada
August 2004

Introduction

Informal description of a knapsack problem:

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

Classical Example: The "subset sum" problem, from cryptosystems

- ◇ **From a set *S* of numbers, and a given number *k*, find a subset of *S* whose sum is *k*.**

To be honest I am never certain of what specifically comprises a knapsack problem. I wanted to write up what I know about various types of problems that all seem to fall in or near the knapsack umbrella. I don’t know the correct definition of quadratic programs either, but that never stopped me from talking about them.

Acknowledgements (probably should be apologies)

I thank Christian Jacob and Peter Mitic for allowing me to send a paper relatively late in the process. I had for some time wanted to write up whatever I know, or suspect, about knapsack problems. This conference was my incentive to do so. Alas, while I found that I could do many of things I had hoped, some problems proved elusive. I kept working at them after the paper went in, and so I now have an extended version as well. Not surprisingly, I STILL cannot solve all the problems I like to think fall into the knapsack category. So this may grow yet more.

I also thank everyone whose methods and/or code I borrowed, adapted, or stole outright in the course of writing up the paper.

A simple subset sum

We are given a set of rational numbers and a desired value, and seek a subset that sums to the value.

```
vec = {1 / 2, 1 / 3, 1 / 4, 1 / 8, 3 / 10, 12 / 79, 13 / 38};
val = 2509 / 2280;
```

Method: Set up a matrix wherein the last column is precisely the vector augmented at the bottom with the negated value. Augment to the left with an identity matrix (next to the vector), and a row of zeros preceding the negated value. Rescale to make the last column large. Reduce this lattice.

The elements of the identity matrix serve as recorders of the multiples we use in (attempting to) zero the last column, in much the same way that augmenting may be used to form a matrix inverse. If we obtain a row with a zero in the last entry and all ones and zeros preceding, then the columns with ones correspond to set elements we take for our summands.

A simple subset sum

```
lattice1 = Transpose[Append[IdentityMatrix[Length[vec]], 10^10 * vec]]
lattice2 = Append[lattice1, Append[Table[0, {Length[vec]}], -10^10 * val]]
```

```
{{1, 0, 0, 0, 0, 0, 0, 5000000000},
  {0, 1, 0, 0, 0, 0, 0, 10000000000/3},
  {0, 0, 1, 0, 0, 0, 0, 2500000000},
  {0, 0, 0, 1, 0, 0, 0, 1250000000},
  {0, 0, 0, 0, 1, 0, 0, 3000000000},
  {0, 0, 0, 0, 0, 1, 0, 120000000000/79},
  {0, 0, 0, 0, 0, 0, 1, 65000000000/19},
  {0, 0, 0, 0, 0, 0, 0, -(627250000000/57)}}
```

A simple subset sum

```
lr = LatticeReduce[lattice2];
result = Map[Most, Select[lr, Last[#] == 0 && Apply[And, Map[# == 1 || # == 0 &, Most[#]]] &]]
Map[#.vec == val &, result]
```

```
{{0, 1, 0, 1, 1, 0, 1}}

{True}
```

We thus see that the second, fourth, fifth, and seventh elements sum to the desired value.

We discuss handling of harder subset sums in the paper. In addition to lattice reduction tactics we also show code that was presented by a *Mathematica* user in comp.soft−sys.math.mathematica. It uses an implicit back−track scheme (a sort of branch−and−bound) implemented by recursion.

Linear diophantine equations

I always use this example. Given 143267 coins (pennies, nickels, dimes, and quarters) of total value $12563.29, how many coins might be of each type?

We require nonnegative integer solutions to a pair of equations p + 5 n + 10 d + 25 q = 1256329 and p + n + d + q = 143267. We use these as columns of our lattice, again augmenting by an identity matrix.. To assist the process of obtaining a small vector that really gives a solution we will multiply the equations by a large value.

```
vecs = 10^8 * {{1, 5, 10, 25, -1 256 329}, {1, 1, 1, 1, -143 267}};
lattice = Transpose[Join[IdentityMatrix[Length[vecs[[1]]]], vecs]]
```

```
{{1, 0, 0, 0, 0, 100 000 000, 100 000 000}, {0, 1, 0, 0, 0, 500 000 000, 100 000 000},
 {0, 0, 1, 0, 0, 1 000 000 000, 100 000 000}, {0, 0, 0, 1, 0, 2 500 000 000, 100 000 000},
 {0, 0, 0, 0, 1, -125 632 900 000 000, -14 326 700 000 000}}
```

Linear diophantine equations

```
redlat = LatticeReduce[lattice]
```

```
{{0, 3, -4, 1, 0, 0, 0}, {-5, 3, 4, -2, 0, 0, 0}, {41 749, 39 185, 35 978, 26 355, 1, 0, 0},
 {0, -2, 1, 0, 0, 0, -100 000 000}, {1, -2, 1, 0, 0, 100 000 000, 0}}
```

```
{{0, 3, -4, 1, 0, 0, 0}, {-5, 3, 4, -2, 0, 0, 0}, {41 749, 39 185, 35 978, 26 355, 1, 0, 0},
 {0, -2, 1, 0, 0, 0, -100 000 000}, {1, -2, 1, 0, 0, 100 000 000, 0}}
```

The third row gives a solution: 41749 pennies, 39185 nickels, 35978 dimes, and 26355 quarters.

Generally speaking it is not difficult to find "solutions" when we allow negative values. Making this method find nonnegative solutions in the presence of many small null vectors can be difficult. We address this issue in the paper. We also show how to set this up using 0−1 valued variables; in some cases that gives an advantage to the lattice machinery.

A set covering problem

I’ve used this example a few times also. We are given a set of sets, each containing integers between 1 and 64. Their union is the set of all integers in that range, and we want to find a set of 12 subsets that covers that entire range (we are given in advance that that number can be achieved).

```
subsets = {{1, 2, 4, 8, 16, 32, 64}, {2, 1, 3, 7, 15, 31, 63},
   {3, 4, 2, 6, 14, 30, 62}, {4, 3, 1, 5, 13, 29, 61}, {5, 6, 8, 4, 12, 28, 60},
   {6, 5, 7, 3, 11, 27, 59}, {7, 8, 6, 2, 10, 26, 58}, {8, 7, 5, 1, 9, 25, 57},
   {9, 10, 12, 16, 8, 24, 56}, {10, 9, 11, 15, 7, 23, 55}, {11, 12, 10, 14, 6, 22, 54},
   {12, 11, 9, 13, 5, 21, 53}, {13, 14, 16, 12, 4, 20, 52}, {14, 13, 15, 11, 3, 19, 51},
   {15, 16, 14, 10, 2, 18, 50}, {16, 15, 13, 9, 1, 17, 49}, {17, 18, 20, 24, 32, 16, 48},
   {18, 17, 19, 23, 31, 15, 47}, {19, 20, 18, 22, 30, 14, 46},
   {20, 19, 17, 21, 29, 13, 45}, {21, 22, 24, 20, 28, 12, 44}, {22, 21, 23, 19, 27, 11, 43},
   {23, 24, 22, 18, 26, 10, 42}, {24, 23, 21, 17, 25, 9, 41}, {25, 26, 28, 32, 24, 8, 40},
   {26, 25, 27, 31, 23, 7, 39}, {27, 28, 26, 30, 22, 6, 38}, {28, 27, 25, 29, 21, 5, 37},
   {29, 30, 32, 28, 20, 4, 36}, {30, 29, 31, 27, 19, 3, 35}, {31, 32, 30, 26, 18, 2, 34},
   {32, 31, 29, 25, 17, 1, 33}, {33, 34, 36, 40, 48, 64, 32}, {34, 33, 35, 39, 47, 63, 31},
   {35, 36, 34, 38, 46, 62, 30}, {36, 35, 33, 37, 45, 61, 29}, {37, 38, 40, 36, 44, 60, 28},
   {38, 37, 39, 35, 43, 59, 27}, {39, 40, 38, 34, 42, 58, 26}, {40, 39, 37, 33, 41, 57, 25},
   {41, 42, 44, 48, 40, 56, 24}, {42, 41, 43, 47, 39, 55, 23}, {43, 44, 42, 46, 38, 54, 22},
   {44, 43, 41, 45, 37, 53, 21}, {45, 46, 48, 44, 36, 52, 20}, {46, 45, 47, 43, 35, 51, 19},
   {47, 48, 46, 42, 34, 50, 18}, {48, 47, 45, 41, 33, 49, 17}, {49, 50, 52, 56, 64, 48, 16},
   {50, 49, 51, 55, 63, 47, 15}, {51, 52, 50, 54, 62, 46, 14}, {52, 51, 49, 53, 61, 45, 13},
   {53, 54, 56, 52, 60, 44, 12}, {54, 53, 55, 51, 59, 43, 11}, {55, 56, 54, 50, 58, 42, 10},
   {56, 55, 53, 49, 57, 41, 9}, {57, 58, 60, 64, 56, 40, 8}, {58, 57, 59, 63, 55, 39, 7},
   {59, 60, 58, 62, 54, 38, 6}, {60, 59, 57, 61, 53, 37, 5}, {61, 62, 64, 60, 52, 36, 4},
   {62, 61, 63, 59, 51, 35, 3}, {63, 64, 62, 58, 50, 34, 2}, {64, 63, 61, 57, 49, 33, 1}};
```

A set covering problem

To cast this as a standard knapsack problem we first transform our set of subsets into a "bit vector" representa−tion; each subset is represented by a positional list of zeros and ones. We will show the first such bit vector.

```
densevec[spvec_, len_] :=
 Module[{vec = Table[0, {len}]}, Do[vec[[spvec[[j]]]] = 1, {j, Length[spvec]}];
  vec]
mat = Map[densevec[#, 64] &, subsets];
mat[[1]]
```

```
{1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1}
```

A set covering problem

To form a knapsack problem, the idea is to add component−wise as few bitvectors as possible, subject to the constraint that each component sum be greater than zero (indicating we have "covered" that position). We will address this as an optimization problem.

```
spanningSets[set_, iter_, sp_, seed_, cp_: .5] :=
 Module[{vars, rnges, max = Length[set], nmin, vals},
  vars = Array[xx, max]; rnges = Map[(0 ≤ # ≤ 1) &, vars];
  {nmin, vals} = NMinimize[{Apply[Plus, vars],
     Join[rnges, {Element[vars, Integers]}, Thread[vars.set ≥ Table[1, {max}]]]},
    vars, MaxIterations → iter, Method → {DifferentialEvolution,
      CrossProbability → cp, SearchPoints → sp, RandomSeed → seed}];
  vals = vars /. vals;
  {nmin, vals}]
{min, sets} = spanningSets[mat, 2000, 100, 0, .9]

Out[386] = {12.,
  {0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0,
   0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1}}
```

In the extended version of the paper I show how to use the linear programming capabilities of `NMinimize` to do this in a systematic way that is quite efficient.

Large Keith numbers

Around 1987 Michael Keith first wrote about what he called "repfigits", for "replicating Fibonacce 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, ...}. 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.

The property I find most of interest for these numbers is that they are not exactly easy to find. By 1994 all 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 500 hours of computer time. There are 71 such numbers.

As it happens, one can write the Keith number problem as an integer linear program. That is, the digits of a Keith number of a given size will have to satisfy one of a few (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).

Large Keith numbers

In order to find large ones 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. The explanation behind it is in the paper. Which is fortunate for those who might be interested, because I doubt I'll ever again be able to reconstruct it.

```
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]]
```

Quick example:

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

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

```
k6[[2]]
```

```
{-96 160, -4224, 5752, 7144, 7482, 7616}
```

We think of the "equation" as:

$$-96\,160 * d[0] - 4224 * d[1] + 5752 * d[2] + \quad 7144 * d[3] + 7482 * d[4] + 7616 * d[5] == 0$$

Large Keith numbers

These equations each give rise to integer null space computations. That did not take me so long to code, because I cribbed it from another paper (my own, for a change).

```
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]]]]
```

We now find a set of vectors that generates the solution space.

```
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, none of which have 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 (we will see that it does).

Large Keith numbers

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 our 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.

Large Keith numbers

More ambitious:

Go for the record. Some trial and error testing at smaller sizes showed that the middle equations (third, fourth, or fifth) were most likely to yield a valid result. We use the third equation for 20 digit repfigits.

```
k20 = keithEquations[20];
nulls[20, 3] = integerNullSpace[k20[[3]]];
Timing[keithSolution[nulls[20, 3], 200]]
```

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

Ignore the warning message (you have to, because I deleted it, but it’s in the paper). Voila. 27 847 652 577 905 793 413 was the first Keith number found larger than 19 digits.

It turns out that this method got a bit lucky; I only found one other Keith number with it, and that took more like a half hour of cpu time. But it points the way to a more successful approach.

Since this is an ILP, use a branching method coupled to ordinary LP code (`NMinimize` has this part built in). I did this in the extended paper and found all repfigits through 26 digits. I will also note that a variation on the integer solving stage, using lattice reduction, alone sufficed to find several of those larger Keith numbers.

Other topics discussed in the paper

- Two approaches to a puzzle problem, one using simple equation solving and the other using augment−and−prune iterations (this latter in the extended version).
- Use of `NMinimize` on assorted other knapsack and related problems.
- Further discussion of subset sum problems.
- A branch−and−bound code for the set covering is in the extended version.
- More about finding large Keith numbers using lattice reduction tactics.
  In the extended version we also show computations that find all of them through 26 digits. These are now listed on Michael Keith’s web site. The method is based on a simple branch−and−cut scheme implemented through `NMinimize`. Should this talk inspire you to take up the search, let me know if you surpass that mark (and how you did it).
- References to various Usenet posts from which I took a number of examples.
- References to the literature.

```
a
```