Solving knapsack and related problems

Daniel Lichtblau

Wolfram Research, Inc.
100 Trade Centre Dr.
Champaign IL USA, 61820
danl@wolfram.com

Abstract. Knapsack problems and variants thereof arise in several different fields from operations research to cryptography to really, really serious problems for hard−core puzzle enthusiasts. We discuss some of these and show ways in which one might formulate and solve them using *Mathematica*.

1. Introduction

A knapsack problem is described informally as follows. One has a set of items. One must select from it a subset that fulfills specified criteria. A classical example, from cryptosystems, is what is called the "subset sum" problem. From a set S of numbers, and a given number k, find a subset of S whose sum is k. A variant is to find a subset whose sum is as close as possible to k. Another variant is to allow integer multiples of the summands, provided they are small. That is, we are to find a componentwise "small" vector v such that $v.S \approx k$ (where we regard S as being an ordered set, that is, a vector). More general knapsack problems may allow values other than zero and one (typically selected from a small range), inequality constraints, and other variations on the above themes..

Of note is that the general integer linear programming problem (ILP) can be cast as a knapsack problem provided the search space is bounded. Each variable is decomposed into new variables, one for each "bit"; they are referred to as $0-1$ variables because these are the values they may take. One multiplies these new variables by appropriate powers of two in reformulating the problem in terms of the new variables. We will use this tactic in an example below. An advantage (as we will see) is that often one need not strictly enforce the $0-1$ requirement.

Applications of knapsack problems are manifold. The approximate knapsack with small multipliers variant is used, for example, to find a minimal polynomial given an approximation to a root [Lenstra 1984]. The knapsack approximation problem is also used in a more efficient algorithm for univariate factorization from [van Hoeij 2002]. Applications to cryptosystems are discussed in [von zur Gathern and Gerhard 1999, chapter 17] and [Nguyen 1999].

Among the tools one might use for knapsack problems are brute force search, smart combinatorial search techniques, integer programming optimization methods, algebraic solvers with appropriate constraints, and lattice methods. We will illustrate several of these tactics in the examples below, using *Mathematica* [Wolfram 2003]. The methods we discuss are not new and most have previously appeared in various venues as cited. The object of this paper is to gather together several convenient examples, references, applications, and useful *Mathematica* code under the unifying theme of knapsack solvers.

Timings, where indicated, are performed on a 1.4 Ghz machine using the development kernel for *Mathemat−ica*. A part of this work appeared in [Lichtblau 2004]. I thank Frank Kampas for useful remarks about the problems and certain solving methods, and Adam Strzebonski for expalining how the *Mathematica* function `FindInstance` handles constrained diophantine equations.

2. A simple subset sum

The example below arose in the Usenet news group comp.soft−sys.math.mathematica. A response indicating the method we illustrate appears in [Lichtblau 2002a]. It is the very classical subset sum problem: 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;
```

In a way this is a linear diophantine equation but we seek a particular type of solution, wherein all compo−nents are zero or one. As such solutions are small in Euclidean norm, a common method by which to attempt such problems involves lattice reduction [Lenstra, Lenstra, and Lovász 1982]. The idea is to set up a matrix wherein the last column is precisely the vector augmented at the bottom with the negated value, and we augment to the left with an identity matrix (next to the vector), and a row of zeros preceding the negated value. We then reduce this lattice. The elements of the identiy 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. We will rescale to make the last column large though this is not always necessary. We will show the lattice below so one might see why such a reduction can give something useful.

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

$$\left\{\{1,0,0,0,0,0,0,5000000000\},\left\{0,1,0,0,0,0,0,\frac{10000000000}{3}\right\},\{0,0,1,0,0,0,0,2500000000\},\{0,0,0,1,0,0,0,1250000000\},\{0,0,0,0,1,0,0,3000000000\},\left\{0,0,0,0,0,1,0,\frac{120000000000}{79}\right\},\left\{0,0,0,0,0,0,1,\frac{65000000000}{19}\right\},\left\{0,0,0,0,0,0,0,-\frac{627250000000}{57}\right\}\right\}$$

We will reduce and then select those rows with last element zero and all preceding elements either zero or one. We check that such candidate solutions actually work.

```
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 should note that the method illustrated above it is by no means guaranteed to give a solution if one exists. The useful fact is that, for a large class of problems, it does (see [Lagarias and Odlyzko 1985]). This method and a variant thereof are discussed in [Schnorr and Euchner 1991]. The variant uses an encoding that frequently works better when there might be small vectors in the lattice other than the solution vector.

Variations on this lattice technique have applications to polynomial factorization methods already noted in the introduction, finding small solutions to linear diophantine equations (the "closest vector problem" in lattices; see e.g. [Nguyen 1999], [Matthews 2001], or [Lichtblau 2003b]), and simultaneous diophantine approximation ([Lenstra, Lenstra, and Lovász 1982], [von zur Gathen and Gerhard 1999]; this is in essence the method behind the function `AffineRationalize` in the *Mathematica* standard add−on package `NumberTheory‘Rationalize‘`).

3. Numeric solvers and knapsacks

Another method for the previous example, workable for problems of small size, is to use a numeric solver with equations in place to insure that all variables take on only values 0 or 1. This method was first brought to my attention by [Boneh 1997] (though not indicated in [Lichtblau 2002a], I used it on this example in private follow−up correspondence).

```
vars = Array[x, Length[vec]];
polys = Append[vars * (vars - 1), vars.vec - val];
NSolve[polys]
```

```
{{x[1] → 0., x[2] → 1., x[3] → 0.,
  x[4] → 1., x[5] → 1., x[6] → 0., x[7] → 1.}}
```

It must be mentioned that this method is, for practical purposes, generally no better than brute search, and sometimes considerably worse. The reason, roughly, is that we have an overdetermined nonlinear system wherein all equations but one are quadratic. Such systems are easily overwhelmed by computational complexity when one applies the methods of `NSolve` [Lichtblau 2000]. The nice feature of the method lies in the simplicity of code. More importantly, and the reason we showed it above, is that it may be used to advantage in larger knapsack−style problems when there are more linear equations, as these have an effect of reducing the complexity (for one they reduce the a priori bound on the size of the solution set; indeed even nonlinear equations may help to reduce complexity since the system is overdetermined). In [Rasmusson 2003] we see a nice application to solving a type of classic puzzle. The particular example is known as "Johnny's Ideal Woman" and the problem statement is as follows (see the URL to the Georgia College & State University BITS & BYTES electronic journal given in the Rasmusson reference below).

Johnny's ideal woman is brunette, blue eyed, slender, and tall. He knows four women: Adele, Betty, Carol, and Doris. Only one of the four women has all four characteristics that Johnny requires.
1. Only three of the women are both brunette and slender.
2. Only two of the women are both brunette and tall.
3. Only two of the women are both slender and tall.
4. Only one of the women is both brunette and blue eyed.
5. Adele and Betty have the same color eyes.
6. Betty and Carol have the same color hair.
7. Carol and Doris have different builds.
8. Doris and Adele are the same height.

Which one of the four women satisfies all of Johnny's requirements? Rasmusson solved this is tackled as follows (I have made modest alterations to the code but the essentials remain unchanged).

```
gals = {Adele, Betty, Carol, Doris};
features = {blueeye, slender, brunette, tall};
solvvars = Map[perfectgirl, Range[4]];
elimvars = Flatten[Outer[Operate[##, 0] &, features, Range[4]]];
galnumbers = Thread[{Adele, Betty, Carol, Doris} → {1, 2, 3, 4}];
constraints = Map[#^2 == # &, Flatten[Join[elimvars, solvvars]]];
problem = Join[constraints, {Sum[blueeye[i] * slender[i], {i, 4}] == 3,
      Sum[brunette[i] * tall[i], {i, 4}] == 2,
      Sum[slender[i] tall[i], {i, 4}] == 2,
      Sum[blueeye[i] * brunette[i], {i, 4}] == 1, blueeye[Adele] ==
       blueeye[Betty], brunette[Betty] == brunette[Carol],
      slender[Carol] == 1 - slender[Doris], tall[Adele] == tall[Doris]},
     Table[brunette[i] blueeye[i] slender[i] tall[i] == perfectgirl[i],
      {i, 4}]] /. galnumbers;
```

We solve the system, eliminating all variables other than the `perfectgirl` set. We then do some replace−ment postprocessing to make it clear who is Johnny's favorite (at least this should make it clear to Johnny!).

```
sol = (NSolve[problem, solvvars, elimvars] /.
     feature_[j_Integer] :→ feature[gals[[j]]]) /.
   {1. → True, 0. → False, Rule → Equal}
Cases[sol, perfectgirl[gal_] == True :→ gal, {2}]
```

```
{{perfectgirl[Adele] == True, perfectgirl[Betty] == False,
  perfectgirl[Carol] == False, perfectgirl[Doris] == False}}
```

```
{Adele}
```

In this example it turns out that elimination of extraneous variables has the desirable effect of removing multiplicity from the solution set; the problem specification does not uniquely determine heights of all four women.

As problems of that particular type often contain several equations, some of them linear, the computational complexity tends to be more modest than is the case for a subset sum problem of comparably many vari−ables. Hence one can have more variables and still hope for a result in reasonable time.

At this point I will mention another approach to puzzle problems of this type, as utilized heavily in [Trott 1999]. The idea is to enlarge, via enumeration of possibilities, and then prune the search space iteratively as new variables are taken into account. The enumeration steps consist of adding all possible values of a new variable to each partial solution. These are then pruned using available constraints in an effort to keep the size of the partial solution set reasonable. We will illustrate the method on this example. We begin by showing the power set of eye combinations that can go with the four women (we will not show this for the remaining features).

```
eyes = Flatten[
  Outer[List, Sequence @@ Table[{blueeye, Not[blueeye]}, {4}]], 3]
```

```
{{blueeye, blueeye, blueeye, blueeye}, {blueeye, blueeye, blueeye, ! blueeye},
 {blueeye, blueeye, ! blueeye, blueeye}, {blueeye, blueeye, ! blueeye, ! blueeye},
 {blueeye, ! blueeye, blueeye, blueeye}, {blueeye, ! blueeye, blueeye, ! blueeye},
 {blueeye, ! blueeye, ! blueeye, blueeye},
 {blueeye, ! blueeye, ! blueeye, ! blueeye},
 {! blueeye, blueeye, blueeye, blueeye}, {! blueeye, blueeye, blueeye, ! blueeye},
 {! blueeye, blueeye, ! blueeye, blueeye},
 {! blueeye, blueeye, ! blueeye, ! blueeye},
 {! blueeye, ! blueeye, blueeye, blueeye},
 {! blueeye, ! blueeye, blueeye, ! blueeye},
 {! blueeye, ! blueeye, ! blueeye, blueeye},
 {! blueeye, ! blueeye, ! blueeye, ! blueeye}}
```

We form the lists of possibilities combining women with eye color.

```
galsAndEyes = Flatten[Outer[Thread[And[##]] &, {gals}, eyes, 1], 1]
```

```
{{Adele && blueeye, Betty && blueeye, Carol && blueeye, Doris && blueeye},
 {Adele && blueeye, Betty && blueeye, Carol && blueeye, Doris && ! blueeye},
 {Adele && blueeye, Betty && blueeye, Carol && ! blueeye, Doris && blueeye},
 {Adele && blueeye, Betty && blueeye, Carol && ! blueeye, Doris && ! blueeye},
 {Adele && blueeye, Betty && ! blueeye, Carol && blueeye, Doris && blueeye},
 {Adele && blueeye, Betty && ! blueeye, Carol && blueeye, Doris && ! blueeye},
 {Adele && blueeye, Betty && ! blueeye, Carol && ! blueeye, Doris && blueeye},
 {Adele && blueeye, Betty && ! blueeye, Carol && ! blueeye, Doris && ! blueeye},
 {Adele && ! blueeye, Betty && blueeye, Carol && blueeye, Doris && blueeye},
 {Adele && ! blueeye, Betty && blueeye, Carol && blueeye, Doris && ! blueeye},
 {Adele && ! blueeye, Betty && blueeye, Carol && ! blueeye, Doris && blueeye},
 {Adele && ! blueeye, Betty && blueeye, Carol && ! blueeye, Doris && ! blueeye},
 {Adele && ! blueeye, Betty && ! blueeye, Carol && blueeye, Doris && blueeye},
 {Adele && ! blueeye, Betty && ! blueeye, Carol && blueeye, Doris && ! blueeye},
 {Adele && ! blueeye, Betty && ! blueeye, Carol && ! blueeye, Doris && blueeye},
 {Adele && ! blueeye, Betty && ! blueeye, Carol && ! blueeye, Doris && ! blueeye}}
```

Now we use the relevant constraint to prune this list.

```
galsAndEyes = Cases[galsAndEyes, {AorB_ && a_, ___, BorA_ && a_, ___} /;
   MatchQ[AorB, Adele | Betty] && MatchQ[BorA, Adele | Betty]]
```

```
{{Adele && blueeye, Betty && blueeye, Carol && blueeye, Doris && blueeye},
 {Adele && blueeye, Betty && blueeye, Carol && blueeye, Doris && ! blueeye},
 {Adele && blueeye, Betty && blueeye, Carol && ! blueeye, Doris && blueeye},
 {Adele && blueeye, Betty && blueeye, Carol && ! blueeye,
  Doris && ! blueeye}, {Adele && ! blueeye,
  Betty && ! blueeye, Carol && blueeye, Doris && blueeye},
 {Adele && ! blueeye, Betty && ! blueeye, Carol && blueeye,
  Doris && ! blueeye}, {Adele && ! blueeye, Betty && ! blueeye,
  Carol && ! blueeye, Doris && blueeye}, {Adele && ! blueeye,
  Betty && ! blueeye, Carol && ! blueeye, Doris && ! blueeye}}
```

We add an attribute for girth.

```
widths = Flatten[
   Outer[List, Sequence @@ Table[{slender, Not[slender]}, {4}]], 3];
galsAndEyesAndWidth = Flatten[
   Outer[Thread[And[##]] &, galsAndEyes, widths, 1], 1];
```

This time we can prune with more than one constraint.

```
galsAndEyesAndWidth = Cases[galsAndEyesAndWidth,
   {___, And[CorD_, _, a_], ___, And[DorC_, _, b_], ___} /; MatchQ[CorD,
      Carol | Doris] && MatchQ[DorC, Carol | Doris] && b == Not[a]];
galsAndEyesAndWidth = Select[galsAndEyesAndWidth, Length[
     Select[#, Function[x, MatchQ[x, _ && blueeye && slender]]]] == 3 &]
```

```
{{Adele && blueeye && slender, Betty && blueeye && slender,
  Carol && blueeye && slender, Doris && blueeye && ! slender},
 {Adele && blueeye && slender, Betty && blueeye && slender,
  Carol && blueeye && ! slender, Doris && blueeye && slender},
 {Adele && blueeye && slender, Betty && blueeye && slender,
  Carol && blueeye && slender, Doris && ! blueeye && ! slender},
 {Adele && blueeye && slender, Betty && blueeye && slender,
  Carol && ! blueeye && ! slender, Doris && blueeye && slender}}
```

Now add hair coloring.

```
hairs = Flatten[
   Outer[List, Sequence @@ Table[{brunette, Not[brunette]}, {4}]], 3];
galsAndEyesAndWidthAndHair = Flatten[
   Outer[Thread[And[##]] &, galsAndEyesAndWidth, hairs, 1], 1];
```

Again we prune.

```
galsAndEyesAndWidthAndHair = Cases[galsAndEyesAndWidthAndHair,
   {___, And[BorC_, _, _, a_], ___, And[CorB_, _, _, a_], ___} /;
    MatchQ[BorC, Betty | Carol] && MatchQ[CorB, Betty | Carol]];
galsAndEyesAndWidthAndHair = Select[galsAndEyesAndWidthAndHair,
  Length[Select[#,
      Function[x, MatchQ[x, _ && blueeye && _ && brunette]]]] == 1 &]
```

```
{{Adele && blueeye && slender && brunette, Betty && blueeye && slender && ! brunette,
  Carol && blueeye && slender && ! brunette,
  Doris && blueeye && ! slender && ! brunette},
 {Adele && blueeye && slender && ! brunette,
  Betty && blueeye && slender && ! brunette, Carol && blueeye &&
   slender && ! brunette, Doris && blueeye && ! slender && brunette},
 {Adele && blueeye && slender && brunette, Betty && blueeye && slender && ! brunette,
  Carol && blueeye && ! slender && ! brunette,
  Doris && blueeye && slender && ! brunette},
 {Adele && blueeye && slender && ! brunette,
  Betty && blueeye && slender && ! brunette, Carol && blueeye &&
   ! slender && ! brunette, Doris && blueeye && slender && brunette},
 {Adele && blueeye && slender && brunette, Betty && blueeye && slender && ! brunette,
  Carol && blueeye && slender && ! brunette,
  Doris && ! blueeye && ! slender && brunette},
 {Adele && blueeye && slender && brunette, Betty && blueeye && slender && ! brunette,
  Carol && blueeye && slender && ! brunette,
  Doris && ! blueeye && ! slender && ! brunette},
 {Adele && blueeye && slender && brunette, Betty && blueeye && slender && ! brunette,
  Carol && ! blueeye && ! slender && ! brunette,
  Doris && blueeye && slender && ! brunette},
 {Adele && blueeye && slender && ! brunette, Betty && blueeye && slender && brunette,
  Carol && ! blueeye && ! slender && brunette,
  Doris && blueeye && slender && ! brunette},
 {Adele && blueeye && slender && ! brunette,
  Betty && blueeye && slender && ! brunette, Carol && ! blueeye &&
   ! slender && ! brunette, Doris && blueeye && slender && brunette}}
```

Finally add height (we want her to have dimension).

```
heights =
  Flatten[Outer[List, Sequence@@Table[{tall, Not[tall]}, {4}]], 3];
galsAndEyesAndWidthAndHairAndHeight = Flatten[Outer[
    Thread[And[##]] &, galsAndEyesAndWidthAndHair , heights , 1], 1];
galsAndEyesAndWidthAndHairAndHeight =
  Cases[galsAndEyesAndWidthAndHairAndHeight ,
   {___, And[AorD_, _, _, a_], ___, And[DorA_, _, _, a_], ___} /;
    MatchQ[AorD, Adele | Doris] && MatchQ[DorA, Adele | Doris]];
```

This time we prune with three constraints. When finished, we will have only one ideal woman (who, as it happens, repeats herself. Possibly Johnny was not aware of this trait.)

```
galsAndEyesAndWidthAndHairAndHeight = Flatten[Outer[
    Thread[And[##]] &, galsAndEyesAndWidthAndHair , heights , 1], 1];
galsAndEyesAndWidthAndHairAndHeight =
  Cases[galsAndEyesAndWidthAndHairAndHeight ,
   {___, And[AorD_, _, _, a_], ___, And[DorA_, _, _, a_], ___} /;
    MatchQ[AorD, Adele | Doris] && MatchQ[DorA, Adele | Doris]];
galsAndEyesAndWidthAndHairAndHeight =
  Select[galsAndEyesAndWidthAndHairAndHeight , Length[Select[#,
       Function[x, MatchQ[x, _ && blueeye && _ && _ && tall]]]] == 2 &];
galsAndEyesAndWidthAndHairAndHeight = Select[
  galsAndEyesAndWidthAndHairAndHeight ,
  Length[Select[#, Function[x, MatchQ[x, __ && brunette && tall]]]] == 2 &]
```

```
{{Adele && blueeye && slender && brunette && tall,
  Betty && blueeye && slender && ! brunette && tall,
  Carol && blueeye && slender && ! brunette && ! tall,
  Doris && ! blueeye && ! slender && brunette && tall},
 {Adele && blueeye && slender && brunette && tall,
  Betty && blueeye && slender && ! brunette && ! tall,
  Carol && blueeye && slender && ! brunette && tall,
  Doris && ! blueeye && ! slender && brunette && tall}}
```

We see again that only Adele has the requisite features.

4. Linear diophantine equations

We now show a simple example from [Lichtblau 2002a]. Given 143267 coins (pennies, nickels, dimes, and quarters) of total value \$12563.29, how many coins might be of each type? In general one might expect many different solutions; we will be happy to obtain any one of them. In the reference we show an easy way to obtain results using an optimization method (this amounts to a constraint satisfaction ILP). Here we will attempt a lattice−based method quite similar to what was shown above for the subset sum . We require nonnegative integer solutions to a pair of equations $p + 5n + 10d + 25q = 1\,256\,329$ and $p + n + d + q = 143\,267$. 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]];
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}}
```

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 work in the presence of many small null vectors becomes problematic precisely because it then becomes all the more likely that small solutions will have components of both signs.

It is of some interest to pursue this as a 0 −1 problem. From each variable we make new variables corresponding to each bit. Note that, as above, we will work with vectors of numbers and not form explicit variables.

```
base = 2;
sizes1 = Ceiling[Log[base, 1 256 329/ {1, 5, 10, 25}]];
size2 = Ceiling[Log[base, 143 267]];
sizes = Map[Min[#, size2] &, sizes1];
expandedvec = Map[base ^Range[0, # - 1] &, sizes];
mult = 10 ^3;
vec1 = mult * Append[Flatten[expandedvec * {1, 5, 10, 25}], -1 256 329];
vec2 = mult * Append[Flatten[expandedvec * {1, 1, 1, 1}], -143 267];
lattice = Transpose[Join[IdentityMatrix[Length[vec1]], {vec1, vec2}]];
```

What we seek is a solution vector containing all zeros and ones, with one in the third to last position (denoting that we utilized the last row but not with a multiplier), and zeros in the last two slots (indicating that we satisfied both linear relations).

```
redlat = LatticeReduce[lattice];
solvec = First[Cases[redlat, {a___, 1, 0, 0} :> {a}]]

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

This, in the words of television’s Maxwell Smart, is "not quite what I had in mind." No matter; we can use it anyway. We simply reform our solution values as coefficients times powers of two. For this we must first break apart the solution vector into components that correspond to each of the original variables (pennies, nickels, dimes, and quarters). So long as the largest nonzero component in each is positive we will be able to get a valid solution.

```
start = 1;
solcomponents =
 Table[res = Take[solvec, {start, start + sizes[[j]] - 1}];
  start += sizes[[j]];
  res, {j, Length[sizes]}]

{{1, 0, 0, 0, 1, 0, -1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0},
 {-1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0},
 {0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1},
 {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}}

values = Map[#.(2^Range[0, Length[#] - 1]) &, solcomponents]

{10 449, 16 463, 116 354, 1}

{10 449, 16 463, 116 354, 1}
```

We check that this satisfies the two equations.

```
values .{1, 1, 1, 1} == 143 267 && values .{1, 5, 10, 25} == 1 256 329

True
```

5. A more challenging subset sum

The next example is from [Reinholtz 2003]. We are given the first forty reciprocals of squares of integers larger than one. The goal is to find a subset whose sum is $\frac{1}{2}$. We show Reinholtz’ method below. We begin with the vector itself, clear denominators, and compute the target value in terms of the new vector.

```
vec = Table[1 / n^2, {n, 2, 40}];
len = Length[vec];
lcm = Apply[LCM, Denominator[vec]];
ivec = lcm * vec;
s = lcm / 2;
```

Now find partial sums of all smallest elements, as we decrease the number of such elements. A placeholder table is defined along with a pair of predicates that in effect allow us to tell when to call recursively, and what to test. The bounding feasibility test, used recursively, acts as a sort of backtracking mechanism (one might also view it as a branch−and−bound tactic).

```
tottbl = Apply[Plus, ivec] - FoldList[#1 + #2 &, 0, ivec];
b = Table[0, {len}];
feasibleQ[i_, tot_] := tot ≤ s ≤ tot + tottbl[[i]];
solutionQ[i_, tot_] := tot == s;

try[i_?(#1 ≤ len &), tot_] := With[{rtot = tot + ivec[[i]]}, b[[i]] = 1;
   solutionQ[i, rtot] && Throw[b];
   If[feasibleQ[i + 1, rtot], try[i + 1, rtot]];
   b[[i]] = 0; try[i + 1, tot];];
Timing[result = Catch[try[1, 0]];]
result .vec == 1 / 2
Select[result * vec, # ≠ 0 &]
```

```
{20.55 Second, Null}

True
```

$$\left\{\frac{1}{4}, \frac{1}{9}, \frac{1}{16}, \frac{1}{25}, \frac{1}{49}, \frac{1}{144}, \frac{1}{225}, \frac{1}{400}, \frac{1}{784}, \frac{1}{1225}\right\}$$

With some work this can also be attacked as a lattice problem. Specifically we can set this up as a linear diophaantine problem. There is a hitch: we typically have many null vectors and these tend to give solutions that do not have all ones. We can attempt to alleviate this by adding extra linear relations. For example, suppose we know or at least suspect that a solution using exactly 10 values exists (perhaps we peeked at Reinholtz' solution, but only long enough to guess how big it was). We can set up a lattice as follows.

```
size = 40;
squares = Table[n^2, {n, 2, size}];
lcm = Apply[LCM, squares];
vec = lcm / squares;
vec = Append[vec, -lcm / 2];
vec2 = Table[1, {size}];
vec2[[size]] = -10;
lattice = Transpose[Join[IdentityMatrix[Length[vec]], {vec, vec2}]];
```

The last columns contain the two linear relations we need to enforce; the sum of square reciprocals must equal $\frac{1}{2}$ and the number of values in the sum must be 10. As in earlier examples, the augmented identity matrix records the multiples used in satisfying these relations.

We still face the same difficulty; if we reduce this lattice we can readily make zeros in the last two columns, using the last row multiplied by one, but the multiples of the other rows are not all likely to be one. This in turn is because there are still many small "null" vectors that give rise to small solutions, from which lattice reduction may not find the sort we require (using all ones). In order to further coerce the lattice reduction to yield a good vector we will alter the default behvior. Specifically we change what is often called the δ parameter, typically set to $\frac{3}{4}$ in the liteature (see e.g. [Lenstra, Lenstra, Lovász 1982]), to something much closer to 1.

```
Developer'SetSystemOptions[
  "LatticeReduceOptions "→ {"LatticeReduceRatioParameter "→ .99}];
Timing[redlat = LatticeReduce[lattice];]
```

```
{0.36 Second, Null}
```

We now see if we have any solutions. These will have final three components of a one (signifying that the last row is multiplied by one) followed by two zeros (indicating that we successfully satisfied the two linear relations). Actually these might all be negated but we ignore that possibility for now. Moreover even if the last value is not zero (signifying that the number of elements used was not 10) that would not be cause for concern.

```
solns = Cases[redlat, {a___, 1, 0, _} /; Max[a] == 1 && Min[a] == 0]

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

Our solution is given by this vector with the last three elements removed. We see that it agrees with the previous one.

```
soln = Drop[First[solns], -3];
soln.(1 / squares) == 1 / 2
Select[soln * (1 / squares), # ≠ 0 &]

True
```

$$\left\{\frac{1}{4}, \frac{1}{9}, \frac{1}{16}, \frac{1}{25}, \frac{1}{49}, \frac{1}{144}, \frac{1}{225}, \frac{1}{400}, \frac{1}{784}, \frac{1}{1225}\right\}$$

Obviously we made use of the fact that we expected the solution to use 10 values. In reality we would loop over the possible lengths of a solution subset. It remains to ponder what to do in cases where we do not get a solution vector in the reduced lattice. One method I have used with modest success involves adding one to three more columns of random small integers (perhaps all multiplied by some constant larger than unity). The last element in each column will be the negative of a possible sum of a subset of the values above it. This forces solutions to satisfy more relations, thus often removing the small null vectors. We now must iterate over all possible values of sums of subsets of the new columns. If the random values are drawn from a small pool then typically there are not many such values for each column. This is important because we want to arrange matters so that we do no more than some low degree polynomial number of iterations in the number of rows of the lattice.

An alternative is to put those problematic null vectors to work for us (in modern parlance one might regard this as "embracing our null vectors"). There is direct functionality built into *Mathematica* to do something of this sort. The method used behind the scenes is a type of recursive enumeration over combinations of an unrestricted integer solution plus null vectors ([Strzebonski 2004]). The code below is from [Kampas 2004].

```
vars = Table[x[i], {i, 2, 40}];
cons = Table[0 <= x[i] <= 1, {i, 2, 40}];
```

$$\text{Timing}\left[\text{FindInstance}\left[\text{Join}\left[\left\{\sum_{i=2}^{40} \frac{x[i]}{i^2} == \frac{1}{2}\right\}, \text{cons}\right], \text{vars}, \text{Integers}\right]\right]$$

```
{74.71 Second,
 {{x[2] → 1, x[3] → 1, x[4] → 1, x[5] → 1, x[6] → 0, x[7] → 1, x[8] → 0,
   x[9] → 0, x[10] → 0, x[11] → 0, x[12] → 1, x[13] → 0, x[14] → 0,
   x[15] → 1, x[16] → 0, x[17] → 0, x[18] → 0, x[19] → 0,
   x[20] → 1, x[21] → 0, x[22] → 0, x[23] → 0, x[24] → 0,
   x[25] → 0, x[26] → 0, x[27] → 0, x[28] → 1, x[29] → 0,
   x[30] → 0, x[31] → 0, x[32] → 0, x[33] → 0, x[34] → 0, x[35] → 1,
   x[36] → 0, x[37] → 0, x[38] → 0, x[39] → 0, x[40] → 0}}}
```

6. Random determinants

In this section we illustrate some heuristic methods on certain extremal matrix problems of modest size. It is sometimes important to understand extremal behavior of random polynomials or matrices comprised of elements from a given set. Below we apply knapsack−style optimization to study determinants of $7 x 7$ matrices of integers taken from the set $\{-1, 0, 1\}$, with diagonal elements all set to 1. We arrived at the option settings utilized below after some trial and error experimentation.

```
n = 7;
mat = Array[x, {n, n}];
func[a : {{_?NumberQ ..} ..}] /;
  Length[a] == Length[First[a]] := Det[a]
vars = Flatten[mat];
problemlist = {func[mat], Flatten[{Element[vars, Integers],
     Map[-1 ≤ # ≤ 1 &, vars], Table[x[j, j] == 1, {j, n}]}]};
```

```
Timing[{min, vals} = NMinimize[problemlist , vars,
    MaxIterations → 50, Method → {"DifferentialEvolution ",
      CrossProbability → 1 / 50, SearchPoints → 50}];]
```

```
{37.34 Second, Null}
```

```
{min, mat /. vals}
```

```
{-576., {{1, -1, -1, 1, 1, -1, -1}, {1, 1, 1, -1, 1, 1, -1},
  {-1, -1, 1, 1, 1, 1, -1}, {1, 1, -1, 1, -1, 1, -1}, {1, 1, 1, 1, 1, 1, 1},
  {1, -1, -1, -1, -1, 1, 1}, {-1, 1, -1, -1, 1, 1, 1}}}
```

Note that the Hadamard bound claims the minimum must be no smaller than $-7^{\frac{7}{2}}$, or -907. A random search that took approximately twice as long as the code above found nothing smaller than -288. We'll now try with dimension increased by one.

```
Timing[{min, vals} = NMinimize[problemlist , vars,
    MaxIterations → 200, Method → {"DifferentialEvolution ",
      CrossProbability → 1 / 50, SearchPoints → 100}];]
```

```
{464.97 Second, Null}
```

```
{min, mat /. vals}
```

```
{-4096., {{1, 1, 1, -1, 1, -1, -1, -1}, {1, 1, -1, 1, 1, 1, 1, -1},
  {1, 1, 1, 1, -1, 1, -1, 1}, {-1, 1, 1, 1, -1, -1, 1, -1},
  {1, -1, 1, 1, 1, -1, 1, 1}, {-1, 1, 1, -1, 1, 1, 1, 1},
  {1, -1, 1, -1, -1, 1, 1, -1}, {1, 1, -1, -1, -1, -1, 1, 1}}}
```

In this case we actually attain the Hadamard bound.

We now show an example that, while not really a knapsack problem, is a cousin to the one above. A matrix is called doubly stochastic is all entries are nonnegative and all rows and columns sum to one. A famous theorem due to Birkhoff shows that any such matrix may be written as a convex sum of permutation matrices (these are thus the vertices of the linear space of such matrices). It is not hard to show that the permutation matrices are moreover the doubly stochastic matrices of extremal determinant. Below we find one with determinant equal to -1.

```
n = 7;
mat = Array[x, {n, n}];
func[a : {{_?NumberQ ..} ..}] /;
  Length[a] == Length[First[a]] := Det[a]
vars = Flatten[mat];
problemlist = {func[mat],
   Flatten[{Map[# ≥ 0 &, vars], Table[Sum[x[j, k], {j, n}] == 1, {k, n}],
     Table[Sum[x[j, k], {k, n}] == 1, {j, n}]}]};
```

```
{min, vals} = NMinimize[problemlist , vars, MaxIterations→ 200];
```

```
{min, Chop[mat /. vals, 10^-7]}
```

```
{-1., {{0, 0, 1., 0, 0, 0, 0}, {0, 0, 0, 0, 0, 1., 0},
  {0, 1., 0, 0, 0, 0, 0}, {0, 0, 0, 0, 1., 0, 0}, {0, 0, 0, 0, 0, 0, 1.},
  {1., 0, 0, 0, 0, 0, 0}, {0, 0, 0, 1., 0, 0, 0}}}
```

The actual decomposition of a doubly stochastic matrix into a convex sum of permutation matrices is itself a type of knapsack problem, albeit one that can be handled by an efficient greedy algorithm. Simple *Mathematica* code for this is given in [Lichtblau 1996]. As a rule of thumb, when a greedy algorithm will work to solve a knapsack problem, nothing will beat it.

7. Covering a set by subsets

Subset covering is an important task that appears, for example, in the Quine−McCluskey algorithm for finding an optimal disjunctive normal form for a boolean expression [McCluskey 1956]. We give an example that arose in the Usenet news group comp.soft−sys.math.mathematica. The approach we use appeared previously in [Lichtblau 2002b]. 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}};
```

```
Union[Flatten[subsets]] == Range[64]
```

```
True
```

To cast this as a standard knapsack problem we first transform our set of subsets into a "bit vector" represen−tation; 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}
```

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 remark that this example is in a sense harder than might otherwise be the case due to the presence of inequali−ties. This also provides a clue that it might be wise to employ discrete optimization methods. We thus code it as such and use the *Mathematica* function `NMinimize` in order to solve it.

```
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]

{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}}
```

Information regarding `NMinimize` and in particular the selection and setting of its various options, may be found in advanced documentation for *Mathematica*, or in [Champion 2002]. The fascinating engine behind the optimizer utilized in the example above is described in [Price and Storn 1997]. While it is primarily intended for continuous optimization, one lesson we learn here is that applications of evolutionary methods can themselves evolve. Several related discrete optimization examples are attacked using this functionality in [Lichtblau 2002b] where further mention is made of option settings for `NMinimize` appropriate for such problems. In that article we also show a very different way to approach this particular example using `NMinimize`.

8. In search of those elusive Keith numbers

Keith numbers are defined as follows. Suppose we are given a number s of n digits (we work in base 10, but these can be defined with respect to arbitrary bases). Form a sequence in Fibonacci style as follows. The first n elements are the digits themselves. The $(n+1)^{\text{th}}$ element is the sum of the first n digits. Subsequent elements are the sums of the preceding n elements. Then s is called a Keith number (for Mike Keith, who first discussed these), if it appears in this sequence. For example, the sequence for 197 is $\{1, 9, 7, 17, 33, 57, 107, 197, \ldots\}$ and so 197 is a Keith number. Keith originally referred to these as repfigits, for "replicating Fibonacci digits".

Keith numbers tend to be quite rare (there are only 71 of them below 10^{19}), and known methods for finding them, while flawless (in the sense that they find all of them), are limited in range due to algorithmic complexity and memory requirements. At the time the present work was begun the state of the art, from [Keith 1998], was that all such numbers up to 19 digits had been found but no larger ones were known. We will remedy that situation.

We begin with some background remarks on the nature of two methods on which we have relied thus far. One, lattice reduction, can be used to find small integer solutions to diophantine linear problems. It is particularly useful for finding small null vectors to a given homogeneous integer equation. The other tool, differential evolution, can frequently enforce "reasonable" linear inequality constraints if provided with input that is not too far from satisfying the constraints, especially when that input contains small components and the constraints are directly influenced by (integer) perturbations in the optimization variables. Both methods in a sense form new vectors from old by their respective means of recombination. The fact that one tries to find small things, and the other can more readily impose constraints on sets comprised of small values (as well as the fact that we discuss them together in this section), suggests that it might be profitable to use the two methods in tandem. That is exactly what we will do.

To begin we must find equations to describe these things. If the digits are $\{d_0, d_1, \ldots, d_{n-1}\}$ then the number is $\sum_{j=0}^{n-1} d_j\, 10^{n-1-j}$. Meanwhile we form the sequence using a Fibonacci matrix of dimension n. This is simply a matrix that, when operating on a vector, replaces each element up to the last by its successor, and replaces the last by the sum of the elements. For $n = 2$ this is simply $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$. For, say, $n = 4$, it is $\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$.

If we multiply this matrix by itself $k-1$ times then the dot product of the bottom row with the digit sequence will give the $(n+k)^{\text{th}}$ term in the sequence. Some simple inequality considerations will give fairly tight bounds on how many such multiples can possibly work for a given number of digits n. We will use each possibility to form a homogeneous linear diophantine equation (that is, the sum will be zero). For efficiency, in the actual code we take advantage of the structure of the matrix to avoid forming explicit matrix products.

```
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]]
```

Next we need to solve such systems. This is really just an integer null space computation. For convenience we strip down code from [Lichtblau 2003b].

```
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 demonstrate with a short example. We start by obtaining the set of candidate equation vectors for 5 digit examples.

```
k5 = keithEquations[5];
```

We find the small null vectors for one of these candidates.

```
nulls[5, 2] = integerNullSpace[k5[[4]]]

{{-3, -1, -3, -3, -1}, {-2, -4, -3, 3, -3},
 {1, 6, -5, 6, -2}, {7, -3, -15, 5, 26}}
```

Notice that for any solution vector, its negative is also a solution vector. Thus we see that 31 331 is a Keith number of five digits. That was not too difficult. We now look at examples with 6 digits.

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

We find the small null vectors for one of these candidates.

```
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 (with the first being nonzero, in order that they give a legitimate six digit number). It turns out that this will be the case for all possible equations given by k6. We now need a way to recombine these so that the first component is positive and the rest are nonnegative. This job can be tackled by differential evolution with integer variables. The idea is quite simple. For each null vector we create an integer−valued variable. We allow arbitrary linear combinations of these vectors subject to the constraints that all resulting compo− nents be nonnegative, and the first be positive. Since this is a constraint satisfaction problem we can either use a constant objective function or else use some function that would have the effect of imposing a penalty on combinations that violate the constraints. As the `NMinimize` constraint handler already do this, we will opt for the former (actually, since the zero vector is close to satisfying the constraints, one should perhaps add emphasis to the constraint that the first value is nonzero). The code for this is below.

```
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}
```

We note from [Keith 1998] that 147 640 is in fact in the list. Now we will try for something more ambitious. As there are no known Keith numbers of 20 digits, we will attempt to find one.

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

NMinimize::incst : NMinimize was unable to generate
   any initial points satisfying the inequality constraints
   {-2 Round[x$1808[2]] + Round[x$1808[3]] + 3 Round[x$1808[4]] + 2 Round[x$1808[
         5]] + <<4>> + Round[x$1808[10]] - 2 Round[x$1808[11]] - 4 Round[x$1808[
         12]] + <<5>> ≤ 0, <<10>>}. The initial region specified
   may not contain any feasible points. Changing the initial region
   or specifying explicit initial points may provide a better solution.

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

We have found the first known example of a 20 digit Keith number (hooray for us!). It is, moreover, the first pandigital example (that is, containing all 10 digits). The warning message tells us, not surprisingly, that none of the initial combinations satisfied the constraints. Letting differential evolution work its magic over the course of 200 generations sufficed to overcome that defect.

We check that this is in fact a Keith number. The code below will bracket the original value with the last value in the sequence that is strictly less, and its successor. We have a Keith number if and only if that successor is the original value. We code this to take a list of digits as input.

```
f[list_] := Append[Rest[list], Apply[Plus, list]]
knumsums[list_] := With[{val = FromDigits[list]},
  Take[NestWhile[f, list, Last[#] < val &], -2]]
knum[list_] := Last[knumsums[list]] === FromDigits[list]
knum[{2, 7, 8, 4, 7, 6, 5, 2, 5, 7, 7, 9, 0, 5, 7, 9, 3, 4, 1, 3}]

True
```

Utilizing a clever search algorithm that relies on large tables, Mike Keith found all examples up to 19 digits using about 500 hours of computation time with hardware of mid−to−late 1990's vintage. To be fair in a comparison, the method we show above is by no means guaranteed to work. It just happens to do a nice job, and in the example above required no tuning beyond setting the `MaxIterations` (though possibly other option tuning would make it more effective). But clearly it is much faster than the direct search and by no means requires much memory.

A tentative conclusion is that for some types of knapsack problem, the tandem of lattice and optimization tools can be quite powerful. Actually the scenario is not quite so nice. It turns out that the method above got fairly lucky with the example we did. With substantial work one can get another such set of 20 digits: {1, 2, 7, 6, 3, 3, 1, 4, 4, 7, 9, 4, 6, 1, 3, 8, 4, 2, 7, 9}. This uses the second rather than third of the candidate equations. So we can regard this approach as something that will sometimes work, perhaps after substantial tuning, and more importantly it provides evidence to the effect that lattice reduction in tandem with integer programming methods can be a powerful combination for attacking knapsack problems.

Even alone, lattice methods can find sporadic large Keith numbers. For example, in the code below we borrow a method from [Schnorr and Euchner 1991] to improve our chances of getting a valid result from the lattice reduction step. The idea is to augment each null vector with a zero, and augment the lattice with a row consisting of some nonzero value (typically one) in the new column of zeros, and $\frac{9}{2}$ everywhere else. Thus if there is a valid solution then in this augmented lattice contains the vector consisting of that nonzero value (or its negative) and the remaining entries in the range $\{-\frac{9}{2}, \frac{9}{2}\}$. As this would be a fairly "small" vector, one can hope that it will appear in the reduced basis (this is essentially the idea used by Schnorr and Euchner, in a binary setting, to raise the density at which one can hope to solve subset sum problems). In practice we get a few Keith numbers this way, as well as a larger number of near misses.

```
integerNullSpace2[origvec : {_Integer ..}] :=
 Module[{vec, mat, hnf, red, vecs, m}, vec = origvec;
  mat = Transpose[Join[{vec}, IdentityMatrix[Length[vec]]]];
  hnf = Drop[Last[Developer`HermiteNormalForm[mat]], 1];
  vec = Table[-9 / 2, {Length[vec] + 1}];
  vec[[1]] = 1;
  hnf = LatticeReduce[hnf];
  hnf = Prepend[hnf, vec];
  red = LatticeReduce[hnf];
  vecs = Cases[red, {1 | -1, ___}];
  vecs = Map[Rest[# / Sign[First[#]]] &, vecs];
  vecs + 9 / 2
 ]
```

Using this code we found the following digit lists for Keith numbers.

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

of 22, 23, 24, and 26 digits respectively. The last is again pandigital.

As a side remark, one might well wonder what is the probability that a number of n digits is pandigital. the code below will compute it via recursion.

```
p[n_, 1] = (1 / 10) ^ (n - 1);
p[10, 10] = 10 ! / 10 ^ 10;
p[n_, j_] /; j > n = 0;
p[n_, j_] /; j ≤ 10 :=
 p[n, j] = (j / 10) * p[n - 1, j] + (10 - j + 1) / 10 * p[n - 1, j - 1]
```

It turns out that for 20 or 26 digits the probabilities are about 21 % and 48 % respectively.

```
N[{p[20, 10], p[26, 10]}]
```

```
{0.214737, 0.478985}
```

9. Set covering via branch−and−bound

The method we showed above for handling the set cover uses a heuristic sort of integer programming based on the `DifferentialEvolution` method of `NMinimize`. While this is useful, it would be nice to have the capability to search for solutions in a way that is more exhaustive and guaranteed to find at least one if it exists. This can be attained by the "branch−and−bound" method of implicitly enumerating through a con−strained linear problem. The method is discussed in [Schrijver 1986]. We give a brief synopsis below.

The idea is to solve what are termed relaxations of the problem wherein integrality is not enforced but inequality constraints are in use. For any solution one looks for noninteger parts. If all are integer the solution is fine. Otherwise one takes, say, the first coordinate that is not an integer and spawns a pair of new prob−lems, one constraining the corresponding variable to be less or equal to the floor of the value in the solution, and the other constraining it to be greater or equal to the ceiling of that value. This is the "branching". The "bounding" part comes once we obtain solutions that are integer valued. We evaluate the objective function and now can ignore all spawned relaxed problems for which the evaluation of the objective function is greater, as these can only get worse when integrality is enforced. Note that this approach is also useful for what are referred to as mixed problems, wherein some but not all variables may be constrained to take on integer values. One simply branches only on those so constrained.

The code below is tailored for the set partition problem previously discussed. We keep a counter of the number of times we actually solve a linear programming problem. For simplicity we use `NMinimize` but one might avoid preprocessing and thus obtain greater speed by setting up a direct call to `LinearProgramming`. Also we provide for the possibility that one might know in advance roughly the minimum value, and hence more quickly discard subproblems (and candidate solutions) that will eventually turn out to be suboptimal. We also set up the code in such a way as to stop once we have found some set number of solutions beneath the given bound. Even without this it is quite possible that the code above may miss some solutions (though it is not too hand to alter in such a way as to find all possible solutions). As is, it will find many, and perhaps most.

```
setCover[vecs_, startmin_: 0, maxsols_: Infinity] :=
 Module[{len = Length[vecs], x, vars, c1, c2, program, min, vals, stack
   counter = 0, tmp, solns = {}, mylist, obj, constraints, numsols = 0},
  min = If[startmin == 0, Length[vecs], startmin];
  vars = Array[x, len];
  c1 = Join[Map[GreaterEqual[#, 0] &, vars],
    Map[LessEqual[#, 1] &, vars]];
  c2 = Map[GreaterEqual[#, 1] &, Apply[Plus, vars * vecs]];
  obj = Apply[Plus, vars];
  program = {obj, Join[c1, c2], min};
  stack = {program, {}};
  While[stack =!= {},
   program = First[stack]; stack = Last[stack];
   If[Last[program] > min, Continue[]];
   counter++; program = Drop[program, -1];
   Internal`DeactivateMessages[
    soln = NMinimize[program, vars], NMinimize::"nsol"];
   If[! FreeQ[soln, Indeterminate], Continue[]];
   {tmp, soln} = soln; If[tmp > min, Continue[]];
   soln = Chop[vars /. soln];
   badpos = Position[soln,
     (a_ /; Chop[a - Round[a]] =!= 0), {1}, 1, Heads -> False];
   If[badpos == {},
    If[tmp < min, solns = {}; min = tmp; numsols = 0];
    numsols++; solns = {Apply[mylist, soln], solns};
    If[startmin ≠ 0 && numsols == maxsols, Break[]];
    ,
    badpos = badpos[[1, 1]]; constraints = Last[program];
    stack =
     {{obj, Append[constraints, vars[[badpos]] == 0], tmp}, stack};
    stack = {{obj, Append[constraints, vars[[badpos]] == 1], tmp},
      stack};];];
  {numsols, counter, Flatten[solns, Infinity] /. mylist -> List}]
```

We use this to find several solution to the set covering problem in question. We will use the knowledge that solutions have length 12 in order to gain a speed advantage.

```
Timing[Round[setCover[mat, 12.1, 10]]]

{156.11 Second,
 {10, 166, {{1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1,
     0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0,
     0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0},
    {1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
     0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0,
     0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
    {1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
     1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0,
     0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0},
    {1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
     0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0,
     0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0},
    {1, 1, 0, 0, 0, 0, 0, 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, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0,
     0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0},
    {1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0,
     0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0,
     0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0},
    {1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
     0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0,
     0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0},
    {1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
     1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0,
     0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
     0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
     0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0},
    {1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
     0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0,
     0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0}}}}
```

One will observe that the speed for this problem is quite reasonable. This is due in large part to the power of bounding relaxed problems, so we can avoid many of them. It takes about eight hours on the same machine to find all solutions (there are 1875 in total), and over 30 000 linear subproblems are solved in the effort.

Note that our use of a stack is naive in the case where we do not know in advance a good upper bound on the objective function. One can attempt to make faster initial progress with a priority queue (also known as an ordered heap) [Aho, Hopcroft, and Ullman 1974]. The idea is to inspect subproblem relaxations ordered by objective function values, hence looking earlier at more promising candidates. A simple *Mathematica* implementation of this data structure, along with some applications, is discussed in [Lichtblau 2003a]. Another reason to proceed by objective function values is to rule out more quickly any possibility of solu− tions of low values. For example, our problem above has a lower bound for the relaxation of something under 11. This means the problem with integer constraints might have a solution of size 11. Early exploration of relaxations with lower objective function values will tend to show relatively soon that the actual solutions must have size 12 or larger. If we do not do this then generally we must try more possibilities before we can rule out smaller solution sets.

We should point out that a variety of related problems of modest size can be handled in an essentially similar manner. For example, the travelling salesman problem on n cities, and variants thereof, may be set up as a $0-1$ problem with variables $x_{j,k}$ for each pair $\{j, k\}$ of cities. We interpret $x_{j,k} = 1$to mean there is a direct path in the tour from city j to city k. One has the obvious restrictions that

$$x_{j,j} = 0 \text{ for all } 1 \le j \le n$$
$$0 \le x_{j,k} \le 1 \text{ for all } 1 \le \{j,\ k\} \le n$$
$$\sum_{j=1}^{n} x_{j,k} = 1 \text{ for all } 1 \le j \le n$$
$$\sum_{k=1}^{n} x_{j,k} = 1 \text{ for all } 1 \le k \le n$$

These, coupled with the $0-1$ integrality condition, force any solution to be a permutation of the cities (e.g. one path leads into each city and another leads out). They do not impose that it be a tour, so such a solution might in fact have nontrivial cycles. The tour condition may be enforced in various ways. One is to make it a branching condition. That is, for any integer solution we check for cycles and, if present, add new conditions and spawn subproblems accordingly. A simpler approach is given in [Dantzig 1963, 26−3] and is there attributed to A. W. Tucker. For $2 \le j \le n$ we add variables u_j and constraints $u_j - u_k + n\, x_{j,k} \le n-1$ for all pairs $1 \le \{j,\ k\} \le n$. It can be shown that this suffices to enforce that all solutions be tours, and moreover that no actual tours are excluded from consideration.

Again, one can improve the method we indicate using heuristics to speed the process such as starting with a known value near the minimum (to make pruning faster). That said, for combinatorial problems involving n entities we have $O(n^2)$ variables and a comparable number of constraints. Beyond modest size one is gener−ally better off applying heuristic (or, in some cases, deterministic) methods dedicated to combinatorial optimization.

10. A logic puzzle with $0-1$ inequality constraints

This next example illustrates a type of popular logic puzzle often found in supermarket and airport publica−tions. This one is called "Dinner on the run" [Hannagan 2004]; I cribbed it from my son’s puzzle magazine. A description of the problem is as follows. Five men order five sandwiches from Max’s Deli, such that each has a distinct fillings, toppings, spreads, and breads. Further clues are as below, and we are to deduce the components of each person’s sandwich. We follow customary rules of interpretation of such puzzles, e.g. "Neither the sandwich with cheese nor the one with mayo was on white bread" means, among other things, that the cheese and mayo sandwiches are themselves distinct.

(1) One filling is salami. One topping is tomatoes. One spread is butter. One bread is pumpernickel. Frank placed the order.
(2) Regardless of order, Max refuses to put mustard or ketchup on tuna salad, or onions on turkey.
(3) Tim does not like onions.
(4) The roast beef was not on rye, and did not contain pickles.
(5) The tuna salad was not on seven−grain bread.
(6) Neither tuna salad nor sandwhich with lettuce was spread with mayo.
(7) The turkey, which was on whole wheat, did not have mustard.
(8) The sandwich with lettuce did not have mustard.
(9) Neither Nick nor the person requesting ketchup and onions used white bread.
(10) The ham sandwich had neither pickes nor onions.
(11) The sandwich with pickles did not have mayo.
(12) Jim did not order relish.
(13) The sandwich with lettuce did not have relish.
(14) The sandwich with cheese did not have mustard. Moreover it did not go to Tim.
(15) Neither the sandwich with cheese nor the one with mayo was on white bread.
(16) Elmer does not eat rye bread.
(17) Tim did not order his with whole wheat bread.
(18) The sandwich with whole wheat did not have mayo.
(19) Nick did not order tuna salad.
(20) Jim did not order seven−grain bread.
(21) Neither Jim nor Nick used lettuce.

We first use the information just to figure out the five items in each of five categories. The men are Frank, Jim, Nick, Tim, and Elmer. The fillings are salami, tuna salad, turkey, ham, and roast beef. The toppings are tomatoes, lettuce, onions, cheese, and pickles. The spreads are butter, ketchup, mustard, mayo, and relish. The breads are pumpernickel, seven−grain, white, rye, and wheat. Note that this preprocessing step is in itself a bit of a challenge. (Is relish a topping or spread? Is cheese a filling or topping?)

As there are a scant $(5!)^4$ possibilities we could, with some work, do a brute force search. A better approach would be the enumerate−and−prune method used in [Trott 1999]. But we will use a constraint satisfaction approach via linear programming as we believe it is instructive to demonstrate handling of inequations in this setting.

Conceptually, we arbitrarily number the men Frank = 1, ..., Elmer = 5. We create variables for each of the categories: filling, top, spread, and bread. For each such variable we create subvariables, one per sandwich number. These must be zero or one and sum to one. For example, we will have the equation

$$\sum_{j=1}^{5} \texttt{mayo}[j] = 1$$

Moreover we have transposes of the above type of equation. That is, we know the sum of all subvariables for a particular sandwich and category must be also be one. For example, we have

$$\texttt{mayo}[1] + \texttt{ketchup}[1] + \texttt{mustard}[1] + \texttt{relish}[1] + \texttt{butter}[1] = 1$$

We furthermore have equations relating the basic variables to their corresponding subvariables, such as

$$\texttt{mayo} = \texttt{mayo}[1] + 2\,\texttt{mayo}[2] + 3\,\texttt{mayo}[3] + 4\,\texttt{mayo}[4] + 5\,\texttt{mayo}[5]$$

For efficiency we only work with the knapsack variables, using tactics equivalent to these above equations at the end to put the solution into a reasonable form.

Finally we have all the inequations and the handful of equations implied by the 21 rules above. These we handle as follows. From rule 7, say, we know that $\texttt{turkey} = \texttt{wheat}$. This means we equate all the corre−sponding subvariables. From rule 11 we know that $\texttt{pickles} \neq \texttt{mayo}$. Some reflection shows that this equivalent to the set of subvariable inequalities $\texttt{mayo}[j] + \texttt{pickles}[j] \le 1$ for $1 \le j \le 5$. When we equate a pair of items, e.g. turkey and wheat, we realize that by equating each pair of their five subvariables we can remove one. Similarly observe that, for example, Jim did not take lettuce, so we have an equation for the variable representing lettuce used by Jim (it is zero). Again we can remove it from further consideration after we use it to simplify whatever constraints contained that variable. Use of such equations as means to remove variables and constraints tends to make the computation much faster. In this case it gave a speed improvement by a factor of 10.

The code below will quit after finding one valid solution. We are trusting that the author used adequate conditions to ensure that a solution exists and is unique (which is in fact the case). As in previous examples we see that it is not hard to modify the code to allow for other possibilities.

```
findSandwiches[] :=
 Module[{consumers, fills, tops, spreads, breads, frank, jim, nick, tim
   elmer, salami, tuna, turkey, ham, roastbeef, tomatoes, lettuce,
   onions, cheese, pickles, butter, ketchup, mustard, mayo, relish,
   pumpernickel, sevengrain, white, rye, wheat, fillvars, topvars,
   spreadvars, breadvars, varlists, all01vars, pv, prodvars,
   constraints1, constraints2, constraints3, constraints4,
   constraints, vars, program, stack, soln, soln01, badpos,
   counter = 0, eps = 10^(-6), names, suffix, partsoln, psvars},

  consumers = {frank, jim, nick, tim, elmer};
  fills = {salami, tuna, turkey, ham, roastbeef};
  tops = {tomatoes, lettuce, onions, cheese, pickles};
  spreads = {butter, ketchup, mustard, mayo, relish};
  breads = {pumpernickel, sevengrain, white, rye, wheat};

  fillvars = Outer[#1[#2] &, fills, consumers];
  topvars = Outer[#1[#2] &, tops, consumers];
  spreadvars = Outer[#1[#2] &, spreads, consumers];
```

```
breadvars = Outer[#1[#2] &, breads, consumers];
varlists = {fillvars, topvars, spreadvars, breadvars};
all01vars = Flatten[varlists];

constraints1 = Map[0 ≤ # ≤ 1 &, all01vars];
constraints2 = Map[Apply[Plus, #] == 1 &,
   {fillvars, topvars, spreadvars, breadvars}, {2}];
constraints3 = Map[Apply[Plus, #] == 1 &,
   Map[Transpose, {fillvars, topvars, spreadvars, breadvars}], {2}];

ineq[v1_, v2_, v3__] := {ineq[v1, v2], ineq[v1, v3], ineq[v2, v3]};
ineq[v1_, v2_] := Map[v1[#] + v2[#] ≤ 1 &, consumers];
eq[v1_, v2_] := Map[v1[#] == v2[#] &, consumers];

constraints4 =
  {ineq[mustard, tuna], ineq[ketchup, tuna], ineq[onions, turkey],
   ineq[roastbeef, rye], ineq[roastbeef, pickles],
   ineq[tuna, sevengrain], ineq[tuna, lettuce, mayo],
   ineq[turkey, mustard], ineq[mustard, lettuce],
   ineq[ketchup, white], ineq[ham, pickles], ineq[ham, onions],
   ineq[pickles, mayo], ineq[lettuce, relish], ineq[cheese, mustard],
   ineq[cheese, mayo, white], ineq[mayo, wheat]};

constraints = Flatten[
   Join[constraints1, constraints2, constraints3, constraints4]];

partsoln = Flatten[{eq[turkey, wheat], eq[ketchup, onions],
     onions[tim] == 0, rye[elmer] == 0, wheat[tim] == 0, white[nick] == 0,
     onions[nick] == 0, relish[jim] == 0, cheese[tim] == 0, tuna[nick] == 0,
     sevengrain[jim] == 0, lettuce[jim] == 0, lettuce[nick] == 0}];
psvars = Map[First, partsoln];
partsoln = partsoln /. Equal → Rule;

constraints = constraints //. partsoln /.
   {True → Sequence[], ((aa_ /; Head[aa] =!= Plus) ≤ 1) → Sequence[]};
all01vars = Complement[all01vars, psvars];

program = constraints;
stack = {program, {}};

While[stack =!= {},
 counter++; program = stack[[1]]; stack = stack[[2]];
 Internal`DeactivateMessages[
   soln = NMinimize[{1, program}, all01vars]];
 If[! FreeQ[soln, Indeterminate], Continue[]];
 soln = Chop[soln[[2]], eps];
 soln01 = all01vars /. soln;
 badpos = Position[soln01,
    (aa_ /; Chop[aa - Round[aa], eps] =!= 0), {1}, 1, Heads → False];
 If[badpos == {},
   partsoln = partsoln /. soln;
   soln = Join[soln, partsoln];
   soln = Split[Sort[(soln /. aa_?NumberQ :→ Round[aa]) /.
        {HoldPattern[_ → 0] :→ Sequence[], HoldPattern[hh_[bb_] → 1] :→
          bb == hh, HoldPattern[Rule[_, bb_ /; Not[NumberQ[bb]]]] :→
          Sequence[]}], #1[[1]] === #2[[1]] &];
   names = Map[#[[1, 1]] &, soln];
   soln = MapThread[Prepend, {soln /. aa_ == bb_ :→ bb, names}];
   soln = Map[ToString, soln, {2}];
   suffix = soln[[1, 1]];
   suffix =
    StringDrop[suffix, StringPosition[suffix, "$"][[1, 1]] - 1];
   soln = Map[StringReplace[#, suffix → ""] &, soln, {2}];
   Break[],
   badpos = badpos[[1, 1]];
   stack = {Append[program, all01vars[[badpos]] == 0], stack};
```

```
      stack = {Append[program, all01vars[[badpos]] == 1], stack};
     ];
    ];
    {counter, soln}
   ]

SetOptions[LinearProgramming, Method→ InteriorPoint];

Timing[orders = findSandwiches[]]

{0.95606 Second,
 {33, {{elmer, butter, lettuce, turkey, wheat}, {frank, cheese, relish,
     rye, tuna}, {jim, ketchup, onions, pumpernickel, roastbeef},
    {nick, ham, mayo, sevengrain, tomatoes},
    {tim, mustard, pickles, salami, white}}}}
```

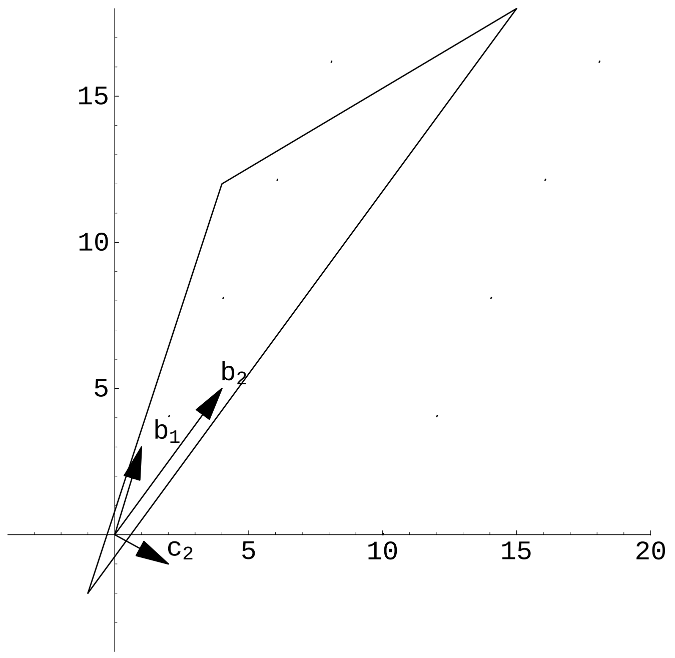
11. Keith numbers via branch−and−cut

One might wonder whether this branching method using relaxed linear problems might be help in the Keith number problem. In that case we deal with simple constraint satisfaction and there is no objective function per se; hence the bounding aspect is not of use. A first attempt similar to the code above took tremendous effort to handle even 6 digits. The problem is that the variables can take on fractional values in ways that combine to make most but not all of the constraints integer valued. Branching on the vector (that is, the digit) values is thus a slow process because this persists even after many such constraints have been added.

What turns out to work well is to branch on the variables themselves. We know that solutions arise when they are integer valued. Forcing them away from fractional values tends to take us to actual solutions in a fairly efficient manner. A bare−bones implementation revealed that this works reasonably well although, not surprisingly, it exhibits exponential behavior. Specifically it takes approximately twice as long for each increase by one digit. The output reveals that the number of subproblems for which a relaxation was solved is small compared to the total size of the search space.

This method is quite similar to one employed in [Aardal and Lenstra 2002] for solving an ILP known as the Frobenius instance problem. Their work has a simple but important refinement. In the setting of a branching algorithm it is as follows. We order the null vectors by decreasing norm and branch on the first one encoun−tered with noninteger coefficient. The idea, roughly, is that this tends to move us more quickly through the polytope corresponding to the relaxed LP, by forcing us first to take directions in which that polytope is thinner. (More technically, the polytope is likely to intersect fewer hyperplanes orthogonal to larger direction vectors and spaced by integral multiples of those vectors.) For difficult Frobenius instance problems one lattice vector will be significantly larger than the rest and this reordering is crucial. For finding Keith num−bers typically the vector lengths in the lattice spans a factor of two or so. In such cases we do not see tremen−dous improvement but experiments indicate the advantage may be as much as a factor of two or more.

The picture below may give some idea of why this vector norm based branching is a good 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 multiples of the good direction c_2 (which comes from the lattice reduction of $\{b_1, b_2\}$) will stay inside it.



Among other possible ways to improve speed we mention the following.

• Use `LinearProgramming` directly (as mentioned earlier for set covering). Possibly this should be done with nondefault option settings.

• Use the dual simplex method on the relaxed problems after adding new constraints. It can be started at the previous LP solution and hence might be expected to finish faster than an LP begun from scratch.

• Use other methods to spawn more subproblem constraints. This might be done, say, with cutting planes (see [Schrijver 1986]). For example we can use relaxations to find minimal and maximal values of the separate variables and then enforce integrality. That is, we take ceilings and floors to obtain what are often tighter constraints. This in turn may yield new restictions so iterating the process can give bounds that are tighter still.

• Use some basis vectors from `integerNullSpace2` as they are often very close to solutions to the fully constrained problem (as we saw, in a few cases they even are solutions). We would not want to have linear dependencies as dimensional components of null vectors tend to make the branching difficult. So we would need to augment these with vectors produced by `integerNullSpace` in such a way as to generate the same lattice. One could check the Hermite normal form to see that this has been done correctly. In some preliminary experiments we obtained a factor of two or so improvement in speed by using a solution basis so constructed.

The idea of utilizing cutting planes has appeal in part because we do not have a specific function to optimize. This being the case, we are free to utilize any linear function of the variables. For example,we could the values of those variables or their negatives, or small integer combinations of them. For each minimal value obtained for such an objective function we might then add a constraint that in future the actual value be no less than its ceiling. We show below a version that works reasonably well. It does several rounds of initial cuts for all variables and then makes subsequent cuts based on extremal values of randomly chosen variables. We note that, as cutting methods go, this one is naive. So one might hope for substantial improvement by more detailed analysis of the inequalities.

We also make small modification so that once a solution is found, further branching will be done as needed until all solutions have been obtained. In this way we can generate all solutions for such numbers of a specified size.

```
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= 1/10^5, rndvar},
  nulls = Reverse[onulls〚Ordering[Norm /@ N[onulls]]〛];
  vars = Array[x, len]; vecs = vars.nulls;
  constraints = Join[{1 ≤ First[vecs] ≤ 9}, (0 ≤ #1 ≤ 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[vars ≤ Floor[maxs + eps]], Thread[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〚RandomInteger[{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]]]}]
```

The function above returns all solutions for a candidate Keith number equation, as well as the number of linear programs that were actually solved (after the initial ones used to find cuts). We used this code to find all Keith numbers up to 23 digits. We first find the appropriate sets of integer equations along with spanning sets of solutions that do not in general satisfy the digit constraints.

```
Do[
  keqns[j] = keithEquations[j];
  Do[vecs = integerNullSpace[keqns[j][[k]]];
   nulls[j, k] = Reverse[vecs[[Ordering[Map[Norm, N[vecs]]]]]],
   {k, Length[keqns[j]]}],
  {j, 2, 23}
 ];
```

Now we solve these. For brevity the full output will not be shown.

```
Timing[knums = Table[Print["digits ", j];
   Print[Timing[digits = Table[
        {j, k, First[Timing[knumlist = keithSolutions[nulls[j, k]];]],
         knumlist}, {k, Length[keqns[j]]}]]]; digits, {j, 2, 23}]]
```

Here are all the Keith numbers between 20 and 23 digits. Note that some had not been found using the heuristic methods from the earlier section. The total run time was about four hours on a machine roughly twice as fast as that used for the other computations in this paper. The time spent in recovering those already known since 1998 (up through 19 digits) was roughly a half hour. We leave it as an open problem whether a better type of cutting or other optimizations might lead to substantial further improvement in speed.

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

A similar but substantially longer run yielded all such numbers through 29 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}
{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}
```

It is interesting that most of these larger ones prior to 29 digits all have leading digit in the set {1, 2, 9}. One might well wonder if there is a deep reason for this, and whether the trend returns after 29 digits.

This general method of computing bounded solutions by feeding a lattice−reduced set of vectors to linear programming code also appears in [Aardal, Hurkens, and Lenstra 2000]. They solve several problems that are demonstrably difficult for classical linear programming branching methods alone. A related possibility is to convert to a $0-1$ problem by creating, for each digit, ten auxiliary variables, similar to how we handled the sandwich problem. This would give a far larger set of Diophantine equations to solve but in return the lattice and branching steps would now work with smaller values.

The ILP refinements of [Aardal and Lenstra 2002] are motivated in large part by Frobenius instance prob− lems. Recent work of [Einstein, Lichtblau, Strzebonski, and Wagon 2005] indicates how it may play a role in the more difficult task of finding Frobenius numbers. In brief, we have a set $B = (b_1, \ldots, b_n)$ of positive integers with $\gcd(B) = 1$. There is a largest integer m such that m cannot be represented as a nonnegative integer combination of elements of B, but every integer $s > m$ can be thusly represented. This m is called the Frobenius number of the set B.

We finish with general remarks regarding the branching methods presented in this and preceding sections. It is clear that the implementations are quite similar. This is, not surprisingly, because the main ideas behind them are essentially the same. While one may not be terribly interested in, say, set covering, or in finding large Keith numbers, the important thing is that the method (along with a basic code framework) is simple and applies to a very large class of integer programming problems. Indeed, the entire body of code presented above for finding Keith numbers is but a few dozen lines. The task of putting together the right tools (e.g. lattice reduction and linear programming) is much easier than that of building the tools themselves. Better still, future versions of *Mathematica* will automatically apply branching methods similar to those we have seen, in the functions `FindInstance`, `Minimize,` and `Reduce`.

12. Notes on implementations and related work

There is a large body of literature regarding attatcks on knapsacks via lattice reduction methods. Methods and applications to cryptosystems are discussed in [Lagarias and Odlyzko 1985], [Schnorr and Euchner 1991], [von zur Gathen and Gerhard 1999], and [Nguyen 1999]. It seems that many such cryptosystems were vanquished in the 1980s and 1990s due to lattice methods. In [Schnorr 1993] there is moreover an attempt to apply lattice methods to integer factorization and computation of discrete logarithms (which could have the effect of breaking RSA−type cryptosystems). This has not yet been successful (to my knowledge!).

Computation of reduced lattices may be done in various ways. The original method of [Lenstra, Lenstra, and Lovász 1982] utilized rational arithmetic. It was recognized even that that integer arithmetic sufficed. A nice exposition may be found in [Cohen 1993, chapter 2]. Efficient variations using floating point (machine and higher precision) arithmetic and integer arithmetic appear in [Schnorr and Euchner 1991] and [Storjohann 1996] respectively. At various times in the past the default *Mathematica* implementation has utilized approxi− mate arithmetic but as of this writing it uses techniques from the latter paper. A considerable amount of unpleasant experience (my own) indicates that an approximate arithmetic version can be difficult to keep both fast and free of bugs; other programmers/implementations may have fared better in this regard. One might experiment with `LatticeReduce` in *Mathematica* via approximate arithmetic by using the line below.

```
Developer‘SetSystemOptions["LatticeReduceOptions "→
   {"LatticeReduceArithmetic "→ ApproximateNumbers}];
```

For restoration of default behavior one sets it to `Integers`.

There are several methods for ILPs and knapsack problems that we did not show herein. One with origins in computational commutative algebra is done via Gröbner bases. A nice *Mathematica* demonstration notebook (with some nontrivial examples) may be found in [Kapadia 2003].

Another example discussed in [Lichtblau 2002b] is as follows. Take the set of reciprocals of the first 100 integers. Divide it into two subsets each of size 50 in such a way that the difference between the sums is minimized (that is, they are the closest pair to half the total). Clearly this can be set up as an approximate subset sum problem; in the reference it is handled in two ways (one as a knapsack), both utilizing the *Mathe− matica* function `NMinimize` as the underlying solver. It would be interesting to see a successful attack based on lattice reduction. We remark that it is effectively a high density subset problem and this alone makes for trouble with lattice methods. But the real problem seems to be the presence of large null sets containing many small vectors. Possibly the tandem of lattice reduction and integer programming methods might be of use? We leave this as an open problem.

As `NMinimize` uses several methods and is, in my opinion, an interesting polyalgorithm, some remarks about its history are in order. First, as one might notice, the example involving the coin problem is not terribly exciting in and of iteself. It is included because it was the first sort of discrete optimization problem we successfully made to work at WRI using differential evolution. In 1999 I had the good fortune to teach a course on nonlinear programming as a visitor in the mathematics department at the University of Illinois. In the class were two graduate students who went on to do work at Wolfram Research: Serguei Chebalov and Brett Champion (the official spelling of the former has since changed twice and is now Sergey Shebalov; I'd not venture to guess what it might be in future). Sergey spent that summer and the next as an intern at WRI working primarily on what would become `NMinimize`. By the end of that first summer we had the coin example working using a sort of penalty method to enforce integrality. This is similar in spirit to a method from [Gisvold and Moe 1972]. We had the advantage that we worked with a method that made no require−ment of smoothness and hence we were free to use penalties with properties better suited to push solutions toward integer values.

The next summer was spent by Sergey in tuning the code, developing tests, and adding simulated annealing and random search to the existing Nelder−Mead and differential evolution methods. As he wrapped up his work and headed back to school, Brett obtained a degree, joined the company, and jumped (fell?) right into the project. He set to work finding and fixing bugs, engaged in code refinement and robustification (the process by which solid proof−of−concept work gets transformed into actual production code), vastly extended the test suite, wrote substantial elementary and advanced documentation, participated in design review for the interface to the functionality, and overhauled various tactics for handling of constraints (equalities, in particular, can be troublesome). He also performed the requisite if messy deity placification (an elaborate process whereby both major and lesser software gods are propitiated according to their rank; the specifics involve proprietary trade secrets and in any case are not for the faint of heart).

We decided after substantial experimentation on his part that enforcing integrality could better be accom−plished by judicious use of `Round` in function evaluation. A year or so later we obtained a copy of [Storn 1999] and learned that the inventors of differential evolution had had much the same experience with this manner of discrete optimization (though curiously enough, 0−1 knapsack problems such as those presented above often seem to be an exception in that they can behave better when integrality is enforced via penal−ties). Moreover, in the book containing that reference some of the immediately following chapters discuss examples where differential evolution works well with discrete and mixed optimization problems. Even so, in private correspondence the inventors were mildly (albeit pleasantly) surprised that we had, with modest success, applied differential evolution to nontrivial examples that are entirely discrete and in some cases combinatorial in nature.

My own role in the development of `NMinimize` was, primarily, to offer the two basic types of advise (that is to say, wanted and unwanted). Mostly this involved the heuristics for handling various sorts of missbehaved examples. As for actual code, I wrote the original code for the `DifferentialEvolution` method. Brett and Sergey being a pair of fine young cannibals, I rather doubt any shred of it remains.

While linear programming code has been a part of *Mathematica* since the early days, it was only in version 4.2 that code was in place that was both fast and reliable. This work was done by Yifan Hu. Since that time he has worked on development of various methods such as interior point, so for some classes of problems it might get faster still As we now had strong linear programming we decided it should become a specialized method in `NMinimize`, as linear problems are not infrequent. The preprocessing code used by `NMinimize` to determine that a system is linear was written by Rob Knapp with some assistance and debugging by Brett Champion. While one sees in some examples above that on occasion the heuristics need to be kicked by option settings, a nice feature is that often does its magic with little or no intervention on the part of the user.

Puzzle problems of the sort we showed are often handled using an enumerate−and−prune mechanism. Another approach is via constraint satisfaction, utilizing the type of linear programming we employed for set covering and Keith numbers. A distinguishing feature is that problems with mostly equality constraints are readily handled in this way. For example, the infamous "Who owns the zebra" problem, which an internet search indicates to be over four decades old, can be formulated as a linear CSP in terms of (mostly) $0-1$ variables. It can also be done by enumerate−and−prune, as in [Trott 1999]. The only troublesome part is in handling inequalities e.g. "Kools are smoked in the house next to the house where the horse is kept." As there are only a few such inequalities one can simply split into a disjunction of several problems and solve each separately; all but one give an empty solution.

Alternatively one might use methods from nonlinear programming. For example a constraint such as $(\text{Kools} - \text{horse})^2 = 1$ can be rewritten as a quadratic in $0-1$ variables. But such quadratic constraints can in turn be reformulated as linear ones (with new variables) so that $0-1$ solutions also satisfy the original quadratic. One fairly common method for doing this is discussed in [Adams and Sherali 1986]. One might then work with a relaxation of the problem, imposing linearity via branching and/or cutting planes. This method is quite general for quadratic knapsack problems and can even be used to factor integers (albeit not competitively with methods used in practice).

Surveying the technologies we used to attack knapsack and related problems, we find linear— and, to some extent, nonlinear— programming, integer lattice normal forms and reduction, and solving systems of nonlin−ear algebraic equations. These all contain sophisticated mathematical algorithms underneath the hood. For our purposes, however, they by and large can be taken as "black boxes" that do very useful things. The code we show above, while not in all instances trivial, is by no means strictly for experts. While the tools we used are themselves complex, they may be put together in ways that are relatively simple in order to solve knap−sack and related problems.

13. References

[Aardal, Hurkens, and Lenstra 2000] 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.

[Aardal and Lenstra 2002] K. Aardal and A. K. Lenstra. Hard equality constrained knapsacks. 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 233, 350−366. Springer−Verlag, 2002.

[Adams and Sherali 1986] W. P. Adams and H. D. Sherali. A tight linearization and an algorithm for zero−one quadratic programming problems. Management Science **32**(10):1274−1290, 1986.

[Aho, Hopcroft, and Ullmanh 1974] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison−Wesley Publishing Company, 1974.

[Boneh 1997] D. Boneh. Private communication, 1997.

[Champion 2002] B. Champion. Numerical Optimization in *Mathematica*: An Insider's View of `NMinimize`. In SCI2002, *Proceedings of the 6th World Multiconference on Systemics, Cybernetics, and Informatics*. Volume 16, N. Callaos, T. Ebisuzaki, B. Starr, J. M. Abe, D. Lichtblau, eds., pages 136−140. International Institute of Informatics and Systemics, 2002.
A *Mathematica* notebook version may be found at:
http://library.wolfram.com/infocenter/Conferences/4311/
Up to date documentation regarding `NMinimize` may be found at:
http://documents.wolfram.com/v5/Built−inFunctions/AdvancedDocumentation/Optimization/NMinimize/

[Cohen 1993] H. Cohen. *A Course in Computational Algebraic Number Theory*. Graduate Texts in Mathematics 138. Springer−Verlag, 1993.

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

[Einstein, Lichtblau, Strzebonski, and Wagon 2005] D. Einstein, D. Lichtblau, A Strzebonski, and S. Wagon. Frobe−nius numbers by lattice enumeration. In preparation.

[von zur Gathen and Gerhard 1999] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, 1999.

[Gisvold and Moe 1972] K. M. Gisvold and J. Moe. A method for nonlinear mixed−integer programming and its application to design problems. Journal of Engineering for Industry, pages 353−364, 1972.

[Hannagan 2004] J. Hannagan. Dinner on the run. Dell Variety Puzzles and Word Games **135**:62. May 2004.

[van Hoeij 2002] M. van Hoeij. Factoring polynomials and the knapsack problem. Journal of Number Theory **95**:167−181, 2002.

[Kampas 2004] F. Kampas. Private communication, 2004.

[Kapadia 2003] D. Kapadia. Integer Programming with Groebner Bases. *Mathematica* Demo notebook, 2003.
Available electronically at:
http://library.wolfram.com/infocenter/Demos/4825/

[Keith 1998] M. Keith. Determination of all Keith Numbers up to 10^{19}. Electronic manuscript, 1998.
Available electronically at:
http://users.aol.com/s6sj7gt/keithnum.htm
See also:
http://users.aol.com/s6sj7gt/mikekeit.htm

[Lagarias and Odlyzko 1985] J. C. Lagarias and A. M. Odlyzko. Solving low−density subset sum problems. Journal of the Association for Computing Machinery **32**(1):229−246, 1985.

[Lenstra 1984] A. K. Lenstra. Polynomial factorization by root approximation. In EUROSAM 84, *Proceedings of the International Symposium on Symbolic and Algebraic Computation*. Lecture Notes in Computer Science 174, pages 272−276. Springer, 1984.

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

[Lichtblau 1996] D. Lichtblau. "doubly stochastic graph to permutations", Usenet news group comp.soft−sys.math.mat‐hematica communication, 1996.
Available electronically at:
http://forums.wolfram.com/mathgroup/archive/1996/Mar/msg00283.html

[Lichtblau 2000] D. Lichtblau. Solving finite algebraic systems using numeric Gröbner bases and eigenvalues. In SCI2000, *Proceedings of the World Conference on Systemics, Cybernetics, and Informatics*. Volume 10 (Concepts and Applications of Systemics, Cybernetics, and Informatics), M. Torres, J. Molero, Y. Kurihara, and A. David, eds., pages 555−560. International Institute of Informatics and Systemics, 2000.

[Lichtblau 2002a] D. Lichtblau. "Re: need a function for sums of subsets", Usenet news group comp.soft−sys.math.mat hematica communication, 2002.
Available electronically at:
http://library.wolfram.com/mathgroup/archive/2002/Feb/msg00410.html

[Lichtblau 2002b] D. Lichtblau. Discrete optimization using *Mathematica*. In SCI2002, *Proceedings of the World Conference on Systemics, Cybernetics, and Informatics*. Volume 16, N. Callaos, T. Ebisuzaki, B. Starr, J. M. Abe, D. Lichtblau, eds., pages 169−174. International Institute of Informatics and Systemics, 2002.
A *Mathematica* notebook version may be found at:
http://library.wolfram.com/infocenter/Conferences/4317/

[Lichtblau 2003a] D. Lichtblau. Ordered heaps and fast marching method. 2003. *Mathematica* notebook available at:
http://library.wolfram.com/infocenter/Demos/4928/

[Lichtblau 2003b] D. Lichtblau. Revisiting strong Gröbner bases over Euclidean domains. Manuscript, 2003.

[Lichtblau 2004] D. Lichtblau. Solving knapsack problems. *Proceedings of the Sixth International Mathematica Symposium* (conference CD−ROM), P. Mitic, ed. Banff, Canada, 2004.

[Matthews 2001] K. R. Matthews. Short solutions of A X=B using a LLL−based Hermite normal form algorithm. Manuscript, 2001.

[McCluskey 1956] E. J. McCluskey, Jr. Minimization of boolean functions. Bell System Technical Journal **35**:1417−1444, 1956.

[Nguyen 1999] P. Nguyen. Cryptanalysis of the Goldreich−Goldwasser−Halevi cryptosystem from Crypto '97. Advances in Cryptology, *Proceedings of CRYPTO 1999*, Santa Barbara, CA, 1999.
Available electronically at:
http://www.di.ens.fr/~pnguyen/pub.html#Ng99

[Price and Storn 1997] K. Price and R. Storn. Differential evolution. Dr. Dobb's Journal, April 1997, pages 18−24 and 78.

[Rasmusson 2003] L. Rasmusson. "Re: *Mathematica* commands needed to solve problem in Set Theory!", Usenet news group comp.soft−sys.math.mathematica, 2003.
Available electronically at:
http://forums.wolfram.com/mathgroup/archive/2003/Sep/msg00258.html
The original statement of the problem is from Georgia College & State University BITS & BYTES **3**:3, February 1998.
http://www.gcsu.edu/acad_affairs/coll_artsci/mathcomp_sci/bits/V3N3Feb98.html
This work has also appeared in The *Mathematica* Journal **9**(2): 289−291, "Tricks of the Trade" column, edited by P. Abbott.

[Reinholtz 2003] K. Reinholtz. "Re: Notebook for low density subset sum?", Usenet news group comp.soft−sys.math.−mathematica, 2003.
Available electronically at:
http://forums.wolfram.com/mathgroup/archive/2003/Apr/msg00409.html

[Schnorr 1993] C. P. Schnorr. Factoring integers and computing discrete logarithms via diophantine approximation. Advances in Cryptology−Eurocrypt '91. Published in Lecture Notes in Computer Science 547, 171−182. Springer−Verlag, 1993.

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

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

[Storjohann 1996] A. Storjohann. Faster algorithms for integer lattice basis reduction. Technical Report 249, Departe−ment Informatik, ETH Zürich, 1996.

[Storn 1999] R. Storn. An introduction to differential evolution. Chapter 6 (pages 79−108) of *New Ideas in Optimiza−tio*n, D. Corne, M. Dorigo, and F. Glover, eds. Advanced Topics in Computer Science Series, McGraw−Hill, 1999.

[Strzebonski 2004] A. Strzebonski. Private communication, 2004.

[Trott 1999] M. Trott. Solving puzzles with *Mathematica*. In the column "Trott's corner", The *Mathematica* Journal **7**(3):291−307, 1999.

[Wolfram 2003] S. Wolfram. *The Mathematica Book* (5th edition). Wolfram Media, 2003.