Algorithm X in 30 lines!

If you were ever interested in writing a Sudoku solver, then you probably heard about the exact cover problem. It asks whether, for a given set *X* and a collection *Y* of subsets of *X*, there exists a subcollection *Y** of *Y* such that *Y** forms a partition of *X*.

Here's an example of the problem written in Python.

```
X = {1, 2, 3, 4, 5, 6, 7}
Y = {
    'A': [1, 4, 7],
    'B': [1, 4],
    'C': [4, 5, 7],
    'D': [3, 5, 6],
    'E': [2, 3, 6, 7],
    'F': [2, 7]}
```

The unique solution in this case is `['B', 'D', 'F']`.

The exact cover problem is NP-complete. Algorithm X was invented by Donald Knuth to solve it. He even suggested an efficient implementation technique called Dancing Links, using doubly-linked circular lists to represent the matrix of the problem.

However, Dancing Links can be tedious to code, and very hard to get right. That's where the magic of Python comes in! One day I decided to write Algorithm X in Python, and I came up with a very interesting variation on Dancing Links.

The algorithm

The main idea is to use dictionnaries instead of doubly-linked lists to represent the matrix. We already have *Y*. From it, we can quickly access the columns in each row. Now we still need to do the opposite, namely a quick access from the columns to the rows. For this, we can modify *X* by transforming it into a dictionnary. In the above example, it would be written like this

```
X = {
    1: {'A', 'B'},
    2: {'E', 'F'},
    3: {'D', 'E'},
    4: {'A', 'B', 'C'},
    5: {'C', 'D'},
    6: {'D', 'E'},
    7: {'A', 'C', 'E', 'F'}}
```

The sharp eye will notice this is slightly different from how we represented *Y*. Indeed, we need to be able to quickly remove and add rows to each column, which is why we use sets. On the other hand, and Knuth doesn't mention this, rows actually remain intact throughout the algorithm.

Here's the code for the algorithm.

```
def solve(X, Y, solution=[]):
    if not X:
        yield list(solution)
    else:
        c = min(X, key=lambda c: len(X[c]))
        for r in list(X[c]):
            solution.append(r)
            cols = select(X, Y, r)
            for s in solve(X, Y, solution):
                yield s
            deselect(X, Y, r, cols)
            solution.pop()

def select(X, Y, r):
    cols = []
    for j in Y[r]:
        for i in X[j]:
            for k in Y[i]:
                if k != j:
                    X[k].remove(i)
        cols.append(X.pop(j))
    return cols

def deselect(X, Y, r, cols):
    for j in reversed(Y[r]):
        X[j] = cols.pop()
        for i in X[j]:
            for k in Y[i]:
                if k != j:
                    X[k].add(i)
```

There, 30 lines exactly!

Formatting the input

Before solving an instance of the problem, we need to transform the input into the format described above. We can simply use

```
X = {j: set(filter(lambda i: j in Y[i], Y)) for j in X}
```

But this is slow. If the size of *X* is *m* and the size of *Y* is *n*, then the number of iterations will be *m*n*. In the case of a Sudoku grid of size *N*, this of the order of *N*^5. We can do better.

```
X = {j: set() for j in X}
for i in Y:
    for j in Y[i]:
        X[j].add(i)
```

This still has *O(m*n)* complexity, but only in degenerate cases. On average it will perform much better, because it won't iterate over all the empty cells. In Sudoku for example, there are exactly 4 entries in each row of the matrix, regardless of size, so this will have complexity *N*^3.

Advantages

- **Simple:** No need to construct a complicated new data structure, everything we used is provided by Python.
- **Readable:** The very first example above is a word-for-word transcription of an example taken straight from Wikipedia!
- **Flexible:** It can *very* easily be extended to solve Sudokus.

Solving Sudokus

All we have to do is describe Sudoku as an exact cover problem. Here's a complete Sudoku solver that can handle grids of any size, be it 3x3, 5x5, or even 2x3, all in less than 100 lines, doctest included! (Thanks to Winfried Plappert and David Goodger for their comments and suggestions)