Python for Education: The Exact Cover Problem

Andrzej Kapanowski
Marian Smoluchowski Institute of Physics, Jagiellonian University, Cracow, Poland
andrzej.kapanowski@uj.edu.pl

Abstract

Python implementation of Algorithm X by Knuth is presented. Algorithm X finds all solutions to the exact cover problem. The exemplary results for pentominoes, Latin squares and Sudoku are given.

1. Introduction

Python is a powerful dynamic programming language that is used in a wide variety of application domains (Lutz, 2007). Its high level data structures and clear syntax make it an ideal first programming language (Downey, 2008) or a language for easy gluing together tools from different domains to solve complex problems (Langtangen, 2006). The Python standard library and third party modules can speed up programs development and that is why Python is used in thousands of real-world business applications around the world, Google and YouTube, for instance. The Python implementation is under an opes source licence that make it freely usable and distributable, even for commercial use.

Python is a useful language for teaching even if students have no previous experience with it. They can explore complete documentation, both integrated into the language and as separate web pages. Since Python is interpreted, students can learn the language by executing and analysing individual commands. Python is sometimes called "working pseudocode" because it is possible to explain an algorithm by means of Python code and next to run a program in order to check if it is correct. Our aim is to use Python to implement an algorithm of solving the exact cover problem. We prove that Python code is readable and can be used to solve many medium size problems in reasonable time.

The paper is organized as follows. In Section 2 the exact cover problem is defined. In Section 3 Python implementation of Algorithm X is presented. Sections 4, 5, and 6 are devoted to pentominoes, Latin squares, and Sudoku, respectively. A summary and conclusions are contained in Section 7.

2. The Exact Cover Problem

In mathematics, given a collection S of subsets of a set X, an exact cover is a subcollection S* of S such that each element in X is contained in exactly one subset in S*. In computer

science, the exact cover problem is a decision problem to find an exact cover or else determine none exists. The exact cover problem is NP-complete (Wikipedia, 2010b).

The relation "contains" can be represented by an incidence matrix A. The matrix includes one row for each subset in S and one column for each element in X. The entry in a particular row and column is 1 if the corresponding subset contains the corresponding element, and is 0 otherwise. In the matrix representation, an exact cover is a selection of rows such that each column contains a 1 in exactly one selected row.

Interesting examples of exact cover problems are: finding Pentomino tilings, finding Latin squares, and solving Sudoku. The standard exact cover problem can be generalized to involve not only "exactly-one" constraints but also "at-most-one" constraints. The N queens problem is an example of such generalization.

3. Python Implementation of Algorithm X

Algorithm X is a recursive, nondeterministic, backtracking algorithm (depth-first search) that finds all solutions to the exact cover problem. Knuth efficiently implemented his Algorithm X by means of the technique called Dancing Links (Knuth, 2000). Algorithm X functions as follows.

```
If the matrix A is empty, the problem is solved; terminate successfully.
Otherwise choose a column c (deterministically).
Choose a row r such that A[r,c] = 1 (nondeterministically).
Include row r in the partial solution.
For each column j such that A[r, j] = 1,
      delete column j from matrix A;
      for each row i such that A[i, j] = 1,
      delete row i from matrix A.
Repeat this algorithm recursively on the reduced matrix A.
```

Now we would like to present Python implementation of the Algorithm X. Extensive use of list comprehensions is present. The program was tested under Python 2.5. Let us define the exception `CoverError` and the function to read the incident matrix from a text file to the table A. The table A is represented by the list of nodes, where a node is a pair (row, column) for a 1 in the incident matrix. Any line of the text file should contain labels of incident matrix columns with 1 in a given row.

```
class CoverError(Exception):
    """Error in cover program."""
    pass

def read_table(filename):
    """Read the incident matrix from a file."""
    f = open(filename,"r")
    table = []
```

```
    row = 0
    for line in f:
        row = row + 1
        for col in line.split():
            table.append((row, col))
    f.close()
    return table

A = read_table("start.dat")
```

Let us define some useful global variables: `B` to keep the solution (selected rows of the incident matrix), `updates` to count deleted nodes on each level, `covered_cols` to remember if a given column is covered. The number of removed nodes is proportional to the number of elapsed seconds. The 2 GHz Intel Centrino Duo laptop did from 20 to 40 kilo-updates per second.

```
B = {}
updates = {}
covered_cols = {}
for (r, c) in A: covered_cols[c] = False
```

Here are some functions to print the solution and to choose the next uncovered column. In our program a column with the minimal number of rows is returned because it leads to the fewest branches.

```
def print_solution():
    """Print the solution - selected rows."""
    print "SOLUTION", updates
    for k in B:
        for node in B[k]:
            print node[1],
        print

def choose_col():
    """Return an uncovered column with the minimal number of rows."""
    cols = [c for c in covered_cols if not covered_cols[c]]
    if not cols:
        raise CoverError("all columns are covered")
    # Some columns can have no rows.
    tmp = dict([(c,0) for c in cols])
    for (r,c) in A:
        if c in cols:
            tmp[c] = tmp[c] + 1
    min_c = cols[0]
    for c in cols:
        if tmp[c] < tmp[min_c]:
            min_c = c
    return min_c
```

The most important is a recursive function `search(k)` which is invoked initially with k=0.

```
def search(k):
    """Search the next row k in the table A."""
    if not A:   # A is empty
        for c in covered_cols:
            if not covered_cols[c]:   # blind alley
                return
        print_solution()
        return
    c = choose_col()
    # Choose rows such that A[r,c]=1.
    rows = [node[0] for node in A if node[1]==c]
    if not rows:   # blind alley
        return
    for r in rows:
        box = []         # a place for temporaly removed rows
        # Include r in the partial solution.
        B[k] = [node for node in A if node[0]==r]
        # Remove row r from A.
        for node in B[k]:
            box.append(node)
            A.remove(node)
            updates[k] = updates.get(k,0) + 1
        # Choose columns j such that A[r,j]==1 (c is included).
        cols = [node[1] for node in B[k]]
        for j in cols:
            covered_cols[j] = True
            # Choose rows i such that A[i,j]==1.
            rows2 = [node[0] for node in A if node[1]==j]
            # Remove rows i from A to box.
            tmp = [node for node in A if node[0] in rows2]
            for node in tmp:
                box.append(node)
                A.remove(node)
                updates[k] = updates.get(k,0) + 1
        search(k+1)
        # Restore deleted rows.
        for node in box:
            A.append(node)
        del box
        del B[k]
        # Uncover columns.
        for j in cols:
            covered_cols[j] = False
    return
```

The program can be saved to the file *cover.py*. Next sections are devoted to the selected applications of the program.

4. Pentomino

Polyominoes are shapes made by connecting certain numbers of equal-sized squares, each joined together with at least one other square along an edge (Golomb, 1994). Pentominoes are made from five squares and they can form twelve distinctive patterns. Some letter names are recommended for them according to the shapes. All pentominoes can fill a board with 60

squares and of different shapes. The standard boards are rectangles of 6 × 10, 5 × 12, 4 × 15, and 3 × 20, but we can try a cross or a chessboard without the center four squares, see Figure 1. Pentominoes can be rotated (turned 90, 180, or 270 degrees) or reflected (flipped over). Note that one-side pentominoes can be also considered, where the reflection in forbidden.

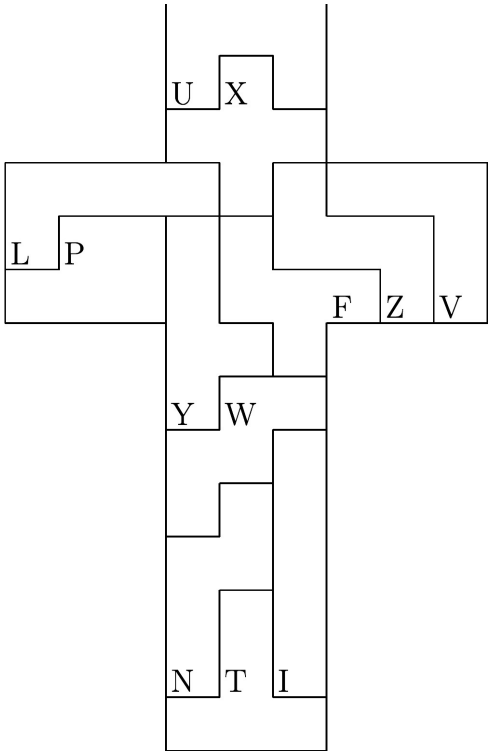

Figure 1. The 12 pentominoes form a cross. There are 21 unique solutions. The naming convention is also shown.

The problem of forming a 60 square board with twelve pentominos involves two kinds of constraints:

- **Pentomino**. For each of the 12 pentominoes, it must be placed exactly once. Columns names correspond to the pentominos: F, I, L, P, N, T, U, V, W, X, Y, Z.
- **Square.** For each of the 60 squares, it must be covered by a pentomino exactly once. A square name can be its successive number.

Thus there are 12+60 = 72 constraints in all. Our results are collected in Table 1.

Board	**Input**	**Solutions**	**Unique**	**Updates**
3 × 20	1100 × 72	8	2	9,770,304
4 × 15	1558 × 72	1472	368	237,324,570
5 × 12	1806 × 72	4040	1010	682,909,158

6 × 10	1928 × 72	9356	2339	1,296,313,446
Cross	1413 × 72	42	21	15,806,634
Chess	1568 × 72	520	65	127,145,172
8 × 8	2357 × 77	129,168	16,146	15,142,060,397

Table 1. Results for different pentomino boards. Input is the size of the incident matrix, Solutions are all solutions found by the program, Unique are different solutions, and Updates are numbers of temporaly removed nodes. The Cross board is shown in Figure 1. The Chess board is 8 × 8 without the center four squares. The 8 × 8 board includes the square tetromino.

There are many other problems connected with pentominoes that can be solved by means of the cover program. Some of them were collected by G. E. Martin in his book (Martin, 1996): the Double Duplication Problem, the Triplication Problem, for instance.

5. Latin Squre

Latin square is an n × n table filled with n different symbols (for example, numbers from 1 to n) in such a way that each symbol occurs exactly once in each row and exactly once in each column. An exemplary Latin square 4 × 4 is shown in Figure 2. Latin squares are used in the design of experiments and error correcting codes (Wikipedia, 2010c).

1	2	3	4
2	3	4	1
3	4	1	2
4	1	2	3

Figure 2.1. Latin square 4 × 4 normalized. There are 4 unique solutions.

The problem of finding Latin squares involves three kinds of constraints:

- **Square.** Each square must contain exactly one number (column name ij).
- **Row-Number.** Each row must contain each number exactly once (column name RxNy).
- **Column-Number.** Each column must contain each number exactly once (column name CxNy).

There are 3n^2 constrains and the incident matrix is n^3 × 3 n^2. The rows describing the Latin square shown in Figure 2 are

```
11 R1N1 C1N1                          12 R1N2 C2N2
```

```
13 R1N3 C3N3                    32 R3N4 C2N4
14 R1N4 C4N4                    33 R3N1 C3N1
21 R2N2 C1N2                    34 R3N2 C4N2
22 R2N3 C2N3                    41 R4N4 C1N4
23 R2N4 C3N4                    42 R4N1 C2N1
24 R2N1 C4N1                    43 R4N2 C3N2
31 R3N3 C1N3                    44 R4N3 C4N3
```

A Latin square is normalized if its fist row and first column are in natural order. For each n, the number of all Latin squares is n!(n-1)! times the number of normalized Latin squares. The exact values are known up to n=11 (McKay, 2005). Our results for normalized Latin squares are collected in Table 2.

Board	**Input**	**Solutions**	**Updates**
1 × 1	1 × 1	1	1
2 × 2	5 × 12	1	12
3 × 3	17 × 27	1	33
4 × 4	43 × 48	4	216
5 × 5	89 × 75	56	3,909
6 × 6	161 × 108	9,408	675,513
7 × 7	265 × 147	16,942,080	1,307,277,285
8 × 8	407 × 192	535,281,401,856	?
9 × 9	593 × 243	377,597,570,964,258,816	?

Table 2. Results for normalized Latin squares. Input is the size of the incident matrix, Solutions are all solutions found by the program, and Updates are numbers of temporaly removed nodes.

6. Sudoku

A standard Sudoku is like an order-9 Latin square, differing only in its added requirement that each subgrid (box) contain the numbers 1 through 9 (Delahaye, 2006). Generaly, a Sudoku of order k (n=k^2) is an n × n table which is subdivided into n k × k boxes. Each raw, column, and box must contain each of the numbers 1 through n exactly once. Any valid Sudoku is a valid Latin square. An exemplary Sudoku 4 × 4 is shown in Figure 3.

1	2	3	4
3	4	1	2
2	1	4	3
4	3	2	1

Figure 3. Sudoku 4 × 4. There are 288 unique solutions.

Note that the Latin square shown in Figure 2 is not a valid Sudoku.

A Sudoku delivers many interesting and sometimes difficult logic-based problems. Let us start from the problem of counting the number of valid Sudoku tables. The problem involves four kinds of constraints:

- **Square.** Each square must contain exactly one number (column name ij).
- **Row-Number.** Each row must contain each number exactly once (column name RxNy).
- **Column-Number.** Each column must contain each number exactly once (column name CxNy).
- **Box-Number.** Each box must contain each number exactly once (column name BxNy).

For the Sudoku board n × n, there are 4 n^2 constrains and the incident matrix is n^3 × 4 n^2. The exemplary rows for the Sudoku shown in Figure 3 are

```
11 R1N1 C1N1 B1N1                    31 R3N2 C1N2 B3N2
12 R1N2 C2N2 B1N2                    32 R3N1 C2N1 B3N1
13 R1N3 C3N3 B2N3                    33 R3N4 C3N4 B4N4
14 R1N4 C4N4 B2N4                    34 R3N3 C4N3 B4N3
21 R2N3 C1N3 B1N3                    41 R4N4 C1N4 B3N4
22 R2N4 C2N4 B1N4                    42 R4N3 C2N3 B3N3
23 R2N1 C3N1 B2N1                    43 R4N2 C3N2 B4N2
24 R2N2 C4N2 B2N2                    44 R4N1 C4N1 B4N1
```

Our results are collected in Table 3. A detailed calculation of the number of classic 9 × 9 Sudoku solutions was provided by Felgenhauer and Jarvis in 2005 (Felgenhauer and Jarvis, 2005) and the number is approximately 6.7 × 10^21. This is 1.2 × 10^(-6) times the number of 9 × 9 Latin squares. Felgenhauer and Jarvis identified 44 classes of different solutions, where first three rows are fixed for a given class when we are looking for solutions.

Board	**Input**	**Solutions**	**Updates**
1 × 1	1 × 1	1	1
4 × 4	64 × 64	288	21,712
9 × 9	729 × 324	6,670,903,752,021,072,936,960	?
16 × 16	4096 × 1024	?	?

Table 3. Results for Sudoku. Input is the size of the incident matrix, Solutions are all solutions found by the program, and Updates are numbers of temporaly removed nodes. Solutions for the 9 × 9 board are cited from the paper by Felgenhauer and Jarvis (2005).

A Sudoku puzzle is a partially completed table, which has a unique solution and has to be completed by a player. The problem of the fewest givens that render a unique solution is unsolved, although the lowest number yet found is 17. There are collected more than 38,000 17-Clou puzzles and there is one known 16-Clue puzzle with two solutions. Our program can easily complete a puzzle or can check that the unique solution exists in few seconds. Many puzzle enthusiasts are looking for the hardest Sudoku, i.e. the Sudoku which is the most difficult to solve for some solver programs. The hardest Sudoku for our program was 21-Clue Sudoku called col-02-08-071 (Wikipedia, 2010c) shown in Figure 4. Peter Norvig (Norvig, 2011) presented a Python program solving Sudoku puzzle which is based on two ideas: constraint propagation and search. The program is based on two mutually-recursive functions. Ali Assaf (Assaf, 2011) implemented the Algorithm X in Python using sets instead of doubly-linked lists.

.	2	.	4	.	3	7	.	.
.	.	.	.	.	.	.	3	2
.	.	.	.	.	.	.	.	4
.	4	.	2	.	.	.	7	.
8	.	.	.	5	.	.	.	.
.	.	.	.	.	1	.	.	.
5	.	.	.	.	.	9	.	.
.	3	.	9	.	.	.	.	7
.	.	1	.	.	8	6	.	.

Figure 4. The hardest Sudoku 9 × 9. There are 113,072 updates in our program.

A Sudoku solution is a special case of a gerechte design (Bailey et al., 2008) used in agricultural experiments. The existence of k^2 × k^2 Sudoku squares for any positive integer k was proved by Herzberg and Murty (2007). Is also possible to construct k^2 × k^2 Sudoku squares with distinct entries on each of the two diagonals for any k (Keedwell, 2007; Akman, 2008).

7. Conclusions

In this paper, we presented Python implementation of Algorithm X solving the exact cover problem. It has less than one hundred lines, counting comments. The program can be used to solve any medium size problem that can be formulated as the exact cover problem. It can handle the cases without solutions or with multiple solutions.

The program was used to solve some puzzles, to generate Latin squares or Sudoku boards. The problems can be analysed according to different criteria: the incident matrix size, number of 1 in a row, number of solutions, or a number of updates on any level of

backtracking. The number of 1 in a row of the incident matrix can be constant (3 for Latin square; 4 for Sudoku) or changing (5 and 6 for pentomino with tetromino; 4, 5 or 6 for Sudoku with distinct entries on the two diagonals).

The total number of updates (and computing time) strongly depends on the rules for choosing an uncovered column. The problem is how to limit a search tree during the backtracking. In our program the column with the minimum number of rows is taken. It is important that the column selection should be done efficiently. Sometimes there are problem-specific hints how to choose a column. In the case of the N queens problem, it is better to place queens near the middle of the board first (Knuth, 2000).

A backtrack program usually spends most of its time on only a few levels of the search tree (Knuth, 2000). In the case of normalized Latin squares, the sumarized number of updates on the corresponding levels is shown in Figure 5. On increasing the table size, the number of updates on the higher levels is increasing.

The presented implementation of Algorithm X can be easily extended to the case of "at-most-one" constraints. We hope that the presented program will be used for teaching or just for fun.

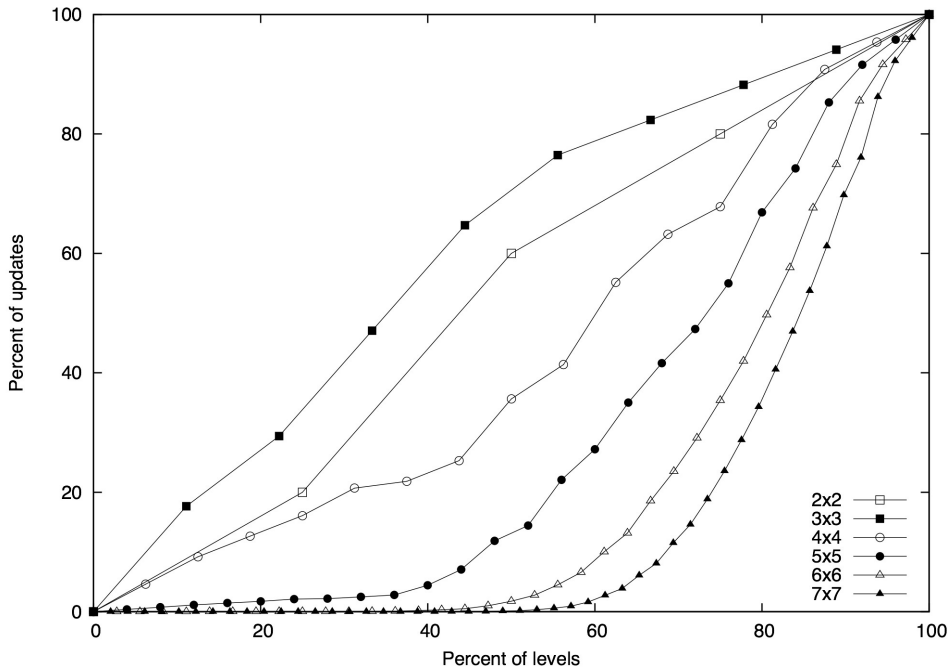

Figure 5. The sumarized number of updates on different levels of backtracking (percents) in the case of normalized Latin squares. Results for tables from 2×2 to 7×7 are shown.

8. References

Akman, F., 2008, Partial Chromatic Polynomials and Diagonally Distinct Sudoku Squares, arXiv:0804.0284v2 [math.CO].

Assaf, A., 2010, Algorithm X in 30 lines!, http://www.cs.mcgill.ca/%7Eaassaf9/python/algorithm_x.html.

Bailey, R. A., Cameron, P. J., Connelly, R., 2008, Sudoku, gerechte designs, resolutions, affine space, spreads, reguli, and Hamming codes, Amer. Math. Monthly 115, 383-404.

Delahaye, J.-P., 2006, The Science Behind Sudoku, Scientific American magazine, June.

Downey, A. B., 2008, Think Python: How to Think Like a Computer Scientist, Green Tea Press, Needham, Massachusetts, http://www.thinkpython.com/.

Felgenhauer, B., Jarvis, F., 2005, Enumerating possible Sudoku grids, http://www.afjarvis.staff.shef.ac.uk/sudoku/.

Golomb, S. W., 1994, Polyominoes: Puzzles, Patterns, Problems, and Packing, Revised and expanded second edition, Princeton, New Jersey: Princeton University Press.

Herzberg, A. M., Murty, M. R., 2007, Sudoku squares and chromatic polynomials, Notices Amer. Math. Soc. 54, 708-717.

Keedwell, D., 2007, On Sudoku squares, Bull. ICA 50, 52-60.

Knuth, D. E., 2000, Dancing Links, arXiv:cs/0011047v1 [cs.DS].

Langtangen, H. P., 2006, Python Scripting for Computational Science, Series: Text in Computational Science and Engineering, Vol. 3, second ed., Springer-Verlag Berlin Heidelberg.

Lutz, M., 2007, Learnig Python, Third Edition, O'Reilly Media.

Martin, G. E., 1996, Polyominoes, a guide to puzzles and problems in tiling, The Matematical Association of America, Cambridge University Press.

McKay, B. D., Wanless, I. M., 2005, On the Number of Latin Squares, Annals of Combinatorics 9, 335-344.

Norvig, P., 2011, Solving Every Sudoku Puzzle, http://www.norvig.com/sudoku.html.

Wikipedia, 2010a, Algorithmics of sudoku, http://en.wikipedia.org/wiki/Algorithmics_of_sudoku.

Wikipedia, 2010b, Exact cover, http://en.wikipedia.org/wiki/Exact_cover.

Wikipedia, 2010c, Latin square, http://en.wikipedia.org/wiki/Latin_square.