Exact Cover

Denis Festa

October 17, 2023

Introduction

The following presentation provides a self-contained report on the implementation of the exact cover problem provided by prof. Marina Zanella (University of Brescia, Italy). The language of choice is Python for its simplicity and readability. The code is available at `https://www.kaggle.com/code/denisfesta/exact-cover-problem`.

The following points discuss:

- the choice of the data structures for the algorithm in its basic form;
- the exploration of the solutions to different problems;
- the comparison between loading portions of the file and loading the whole file;
- the comparison between the basic algorithm and the *plus* algorithm;
- the application of the algorithm to the sudoku problem.

Data structures

To pursue the goal of implementing the algorithm, the matrices A and B need to be stored in an appropriate data structure. Since the matrices contain only 0s and 1s, the first idea might be to store boolean values instead of integers, however, using `True` and `False` gives no advantage. One popular library that provides boolean arrays and matrices is `numpy`, which I compare with the less popular `bitarray` library and show the results in figures (4) and (5). The elements of a `numpy` array or matrix occupy less memory than the elements of a `bitarray` array or matrix, however, the `numpy` variable storing the array occupies more memory than the `bitarray` variable storing the array. Given the absence of matrix operations, I don't see any particular advantage in using `numpy` over `bitarray`.

Built-in array of integers

```python
a = [1]*1000
print("Size of a, Size of a[0]")
sys.getsizeof(a), sys.getsizeof(a[0])
```

[13] ✓ 0.0s

```
Size of a, Size of a[0]
(8056, 28)
```

Built-in array of booleans

```python
b = [True]*1000
print("Size of b, Size of b[0]")
sys.getsizeof(b), sys.getsizeof(b[0])
```

[14] ✓ 0.0s

```
Size of b, Size of b[0]
(8056, 28)
```

Numpy array of bits

```python
p = np.ones(10_000, dtype=bool)
print("Size of p, Size of p[0]")
sys.getsizeof(p), sys.getsizeof(p[0])
```

[15] ✓ 0.0s

```
Size of p, Size of p[0]
(10112, 25)
```

Bitarray

```python
c = bitarray('1'*1000)
print("Size of c, Size of c[0]")
sys.getsizeof(c), sys.getsizeof(c[0])
```

[16] ✓ 0.0s

```
Size of c, Size of c[0]
(216, 28)
```

Built-in matrix of integers

```
N = 30
M = 1000
```

[17] ✓ 0.0s

```
a = [[1]*M for _ in range(N)]
print("Size of a, Size of a[0], Size of a[0][0]")
sys.getsizeof(a), sys.getsizeof(a[0]), sys.getsizeof(a[0][0])
```

[18] ✓ 0.0s

```
Size of a, Size of a[0], Size of a[0][0]
```

```
(312, 8056, 28)
```

Built-in matrix of booleans

```
b = [[True]*M for _ in range(N)]
print("Size of b, Size of b[0], Size of b[0][0]")
sys.getsizeof(b), sys.getsizeof(b[0]), sys.getsizeof(b[0][0])
```

[19] ✓ 0.0s

```
Size of b, Size of b[0], Size of b[0][0]
```

```
(312, 8056, 28)
```

Numpy matrix of bits

```
p = np.ones((N, M), dtype=bool)
print("Size of p, Size of p[0], Size of p[0][0]")
sys.getsizeof(p), sys.getsizeof(p[0]), sys.getsizeof(p[0][0])
```

[20] ✓ 0.0s

```
Size of p, Size of p[0], Size of p[0][0]
```

```
(30128, 112, 25)
```

Bitarray matrix

```
c = [bitarray('1'*M) for _ in range(N)]
print("Size of c, Size of c[0], Size of c[0][0]")
sys.getsizeof(c), sys.getsizeof(c[0]), sys.getsizeof(c[0][0])
```

[21] ✓ 0.0s

```
Size of c, Size of c[0], Size of c[0][0]
```

```
(312, 216, 28)
```

I tried to profile the code with `memory-profiler` to find whether the memory occupation advantages we expect to gain from choosing one data structure over the other are actually realized, but I couldn't see any significant difference in the memory usage while executing the code, I guess it's because the memory required to store the matrices is negligible compared to the memory required to store the set of solutions, the set of explored nodes and the stack of the recursive calls.

Exploring the solutions

Different choices of the cardinality of the domain (columns of the matrix A) and the number of sets (rows of the matrix A, rows and columns of the matrix B) lead to different values of:

- the number of solutions found;
- the number of explored nodes to find the solutions;
- the time required to find the solutions.

In the following slides I show the values of these quantities for different autmatically generated instances of the problem, for time constraints I kept the number of sets in a range between 20 and ~ 400 and the cardinality of the domain in a range between 5 and 14.

Solutions found

It's intuitive that the number of solutions found increases with the number of sets and decrease with the cardinality of the domain. In this case the intuition is confirmed by the solutions found (9) (at least, those found in the chosen dimensions of the problem).

UNIVERSITY OF BRESCIA

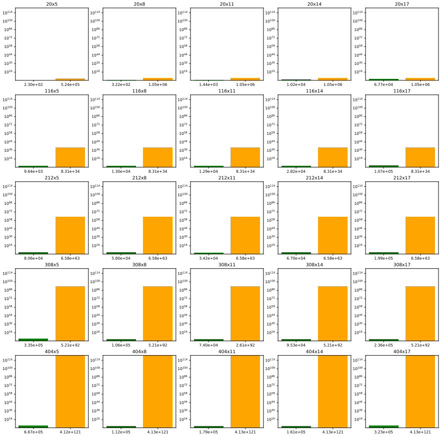
Explored nodes

It might seem intuitive that, similarly to the previous case, the number of explored nodes increases with the number of sets and decreases with the cardinality of the domain, however, the same problems that were solved in the previous case display a different and less intuitive behaviour in this case (11). The interpretation of this behaviour is that the algorithm has to work harder, that is to explore more nodes, to find the solutions for more complex problems, that is those problems with a higher number of rows and a higher number of columns. Why, fixed the number of rows, a small number of columns implies a higher number of explored nodes? I don't know.


20x5
230
20x8
322
20x11
1437
20x14
10158
20x17
67717
116x5
9635
116x8
13049
116x11
12914
116x14
28250
116x17
106896
212x5
80605
212x8
57957
212x11
34220
212x14
67012
212x17
199058
308x5
334824
308x8
106176
308x11
73991
308x14
95302
308x17
236495
404x5
666920
404x8
111592
404x11
178605
404x14
160570
404x17
322704

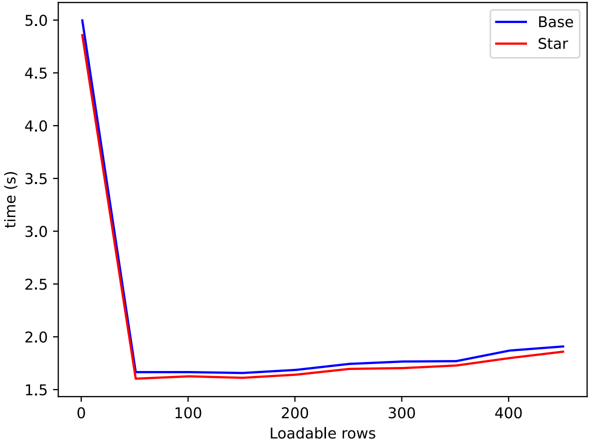
Explorable nodes

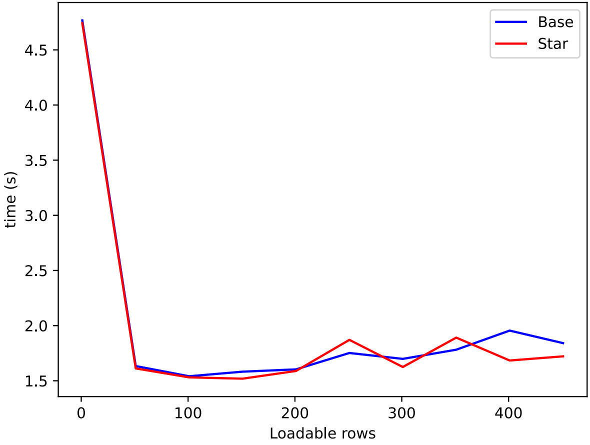
The implemented algorithm chooses to prune the nodes in the tree that cannot lead to a plausible solution. If the algorithm were to explore all the possible nodes, then for the i-th set the number of nodes to explore would be 2^i, hence, if n is the number of sets (rows of A), the number of nodes to explore would be $\sum_{i=0}^{n-1} 2^i = 2^n - 1$.
In figure (13) the enormous difference between the number of explored nodes and the number of explorable nodes is shown. the number of explorable nodes is not exactly $2^n - 1$ because the actual computation to represent the explorable nodes is $\sum_{i \in \mathcal{E}} 2^i$ where $\mathcal{E}$ is the set of explorable indices pointing to rows of A different from an all-zero row (which would be, by definition, compatible with any other row).

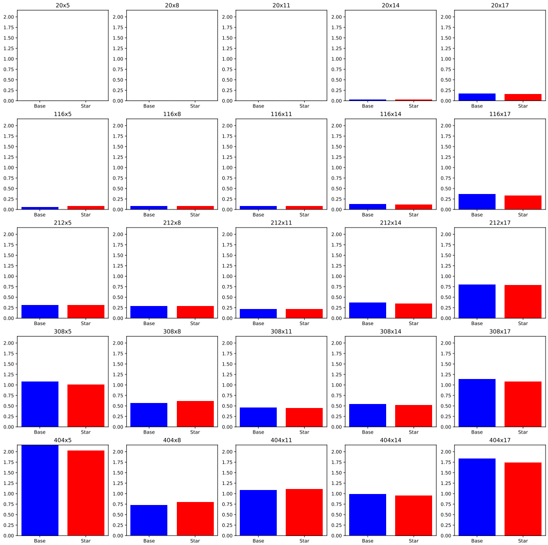

20x5
2.30e+02
5.24e+05
20x8
3.22e+02
1.05e+06
20x11
1.44e+03
1.05e+06
20x14
1.02e+04
1.05e+06
20x17
6.77e+04
1.05e+06
116x5
9.64e+03
8.31e+34
116x8
1.30e+04
8.31e+34
116x11
1.29e+04
8.31e+34
116x14
2.82e+04
8.31e+34
116x17
1.07e+05
8.31e+34
212x5
8.06e+04
6.58e+63
212x8
5.80e+04
6.58e+63
212x11
3.42e+04
6.58e+63
212x14
6.70e+04
6.58e+63
212x17
1.99e+05
6.58e+63
308x5
3.35e+05
5.21e+92
308x8
1.06e+05
5.21e+92
308x11
7.40e+04
2.61e+92
308x14
9.53e+04
5.21e+92
308x17
2.36e+05
5.21e+92
404x5
6.67e+05
4.12e+121
404x8
1.12e+05
4.13e+121
404x11
1.79e+05
4.13e+121
404x14
1.61e+05
4.13e+121
404x17
3.23e+05
4.13e+121

Loading portions

The problem to solve might require a huge matrix A, to prevent the computer from running out of memory one of the possible solutions is to load portions of the matrix and solve the problem for each portion, growing incrementally the set of solutions. What is expected is that loading the whole matrix and solving at once is faster than loading portions of the matrix. The results shown in figure (15) and (16) say something more. What can be noticed is that the greatest advantage that comes from decreasing the dimension of the chunk of loaded rows is gained when the dimension is small, then there is no clear advantage, sometimes loading a greater number of rows at once is faster and sometimes it's slower.


Base
Star
time (s)
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
0
100
200
300
400
Loadable rows


Base
Star
4.5
4.0
3.5
3.0
2.5
2.0
1.5
time (s)
0
100
200
300
400
Loadable rows


20x5
20x8
20x11
20x14
20x17
116x5
116x8
116x11
116x14
116x17
212x5
212x8
212x11
212x14
212x17
308x5
308x8
308x11
308x14
308x17
404x5
404x8
404x11
404x14
404x17
Base
Star

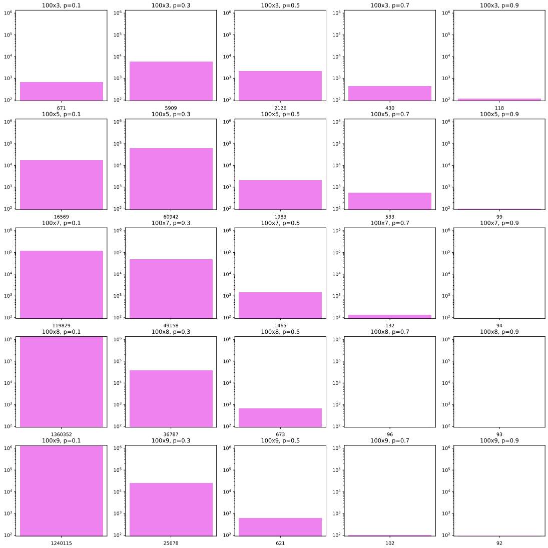
I manually tried to test the behaviour of the algorithm for different dimensions of the chunk on a larger problem (1000 rows and 15 columns) and

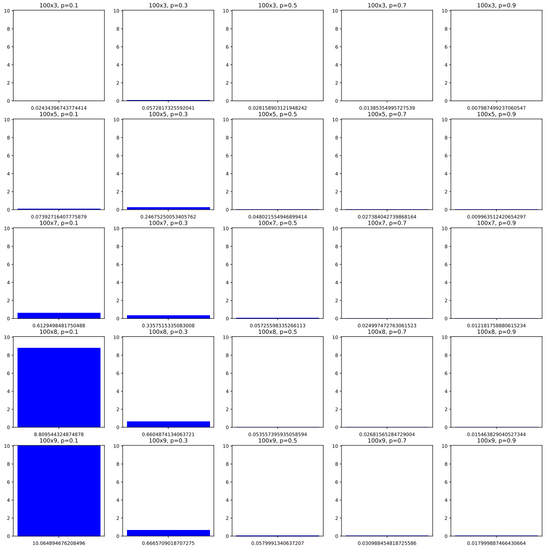
- when loading 1 row at a time the algorithm takes 24.5 seconds;
- when loading 2 rows it takes 15.3 seconds;
- when loading 3 rows it takes 12.3 seconds;
- when loading 5 rows it takes 9.7 seconds;
- when loading 10 rows it takes 7.9 seconds;
- when loading 20 rows it takes 6.9 seconds;
- when loading 50 rows it takes 6.5 seconds;
- when loading 100 rows it takes 6.5 seconds;
- when loading 400 rows it takes 6.6 seconds;
- when loading 900 rows it takes 7.1 seconds.
- when loading 1000 rows it takes 7.6 seconds

More or less 1s

For a better comprehension of how the number of explored nodes depends on the structure of the problem, one possibility is to generate random exact cover problems varying the sparsity of the matrix A. The intuition is that the more 1s there are in each row of the matrix A, the more difficult it is for the problem to have different rows that are compatible, hence, the more difficult it is for the algorithm to find the solutions and the more nodes it has to explore. The more 0s there are in each row of the matrix A, the more likely it is for the problem to find solutions, each partition will contain more rows since each row contains fewer 1s, hence, the algorithm will have less difficulties in finding the solutions and will explore less nodes. The figures (20), (21) and (22) show the results of the experiments and, the larger the number of columns (lower rows), the more the more our intuition is confirmed.


100x3, p=0.1
100x3, p=0.3
100x3, p=0.5
100x3, p=0.7
100x3, p=0.9
2126
9128
5468
3148
1791
100x5, p=0.1
100x5, p=0.3
100x5, p=0.5
100x5, p=0.7
100x5, p=0.9
28728
90213
7563
4985
2137
100x7, p=0.1
100x7, p=0.3
100x7, p=0.5
100x7, p=0.7
100x7, p=0.9
270234
109700
8256
4697
2642
100x8, p=0.1
100x8, p=0.3
100x8, p=0.5
100x8, p=0.7
100x8, p=0.9
3186069
125966
7298
4936
3156
100x9, p=0.1
100x9, p=0.3
100x9, p=0.5
100x9, p=0.7
100x9, p=0.9
3983404
100525
8738
5581
3567
1e6
0.5
1.0
1.5
2.0
2.5
3.0
3.5


100x3, p=0.1
671
100x3, p=0.3
5909
100x3, p=0.5
2126
100x3, p=0.7
430
100x3, p=0.9
118
100x5, p=0.1
16569
100x5, p=0.3
60942
100x5, p=0.5
1983
100x5, p=0.7
533
100x5, p=0.9
99
100x7, p=0.1
119829
100x7, p=0.3
49158
100x7, p=0.5
1465
100x7, p=0.7
132
100x7, p=0.9
94
100x8, p=0.1
1360352
100x8, p=0.3
36787
100x8, p=0.5
673
100x8, p=0.7
96
100x8, p=0.9
93
100x9, p=0.1
1240115
100x9, p=0.3
25678
100x9, p=0.5
621
100x9, p=0.7
102
100x9, p=0.9
92



UNIVERSITY OF BRESCIA

EC wrapper

In the following slides the mechanism wrapping the EC algorithm is presented. The idea is to load a portion of the matrix A, execute the EC algorithm and store the solutions found in a set, then load the next portion of the matrix A, execute the EC algorithm and grow the set of solutions. The important thing to keep in mind is that when the algorithm is executing on a portion of the matrix A, say from row i to row j, the rows from 0 to $i-1$ are not immediately available, they will be read from within the EC algorithm, again in portions. This explains the need for two variables: offset and the reading offset (the same colors will be used in the schema and the animation to follow.)

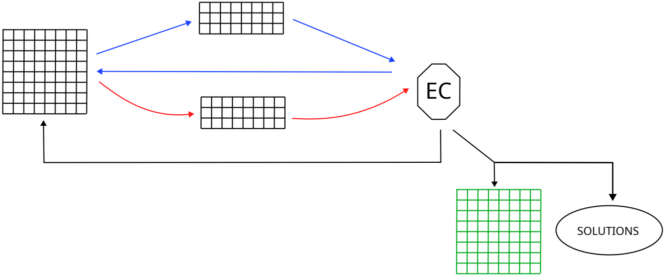
- offset is the index of the first row of the portion of the matrix A that has been passed to the EC algorithm from outside
- reading offset is the index of the first row of the portion of the matrix A that is being read by the EC algorithm, reading offset will always start from 0 and will be incremented until the rows are read from 0 to the last row of the portion of the matrix A that has been passed to the EC algorithm from outside.

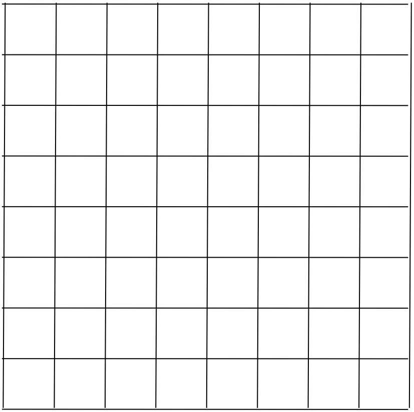
```
function IncrementalExactCover(filename)
    rows ← GetRows(filename)
    columns ← GetColumns(filename)
    B ← zeros(rows, columns)
    COV ← set(∅)
    offset ← 0
    while true do
        A ← GetPortionOfA(filename, offset, loadableRows)
        if A is empty then
            break
        end if
        EC(A, B, COV, offset, filename, loadableRows)
    end while
    return COV
end function
```

```
function EC(A, B, COV, offset, filename, loadableRows)
    N ← ROWS(A)                                          ▷ Number of rows of A
    M ← COLUMNS(A)                                       ▷ Number of columns of A
    for i ← 0 to N − 1 do
        row ← A[i]
        if SUM(row) = 0 then                             ▷ If the row is all 0s
            for t ← 0 to N − 1 do       ▷ set the relative row and column of B to 0
                B[t][i + offset] ← 0
                B[i + offset][t] ← 0
            end for
            continue
        end if
        if SUM(row) = M then                             ▷ If the row is all 1s
            for t ← 0 to N − 1 do       ▷ set the relative row and column of B to 0
                B[t][i + offset] ← 0
                B[i + offset][t] ← 0
            end for
            COV ← COV ∪ {(offset + i, )}
            continue
        end if
        readingOffset ← 0
        WHILECYCLE(A, B, COV, offset, filename, loadableRows, i, readingOffset)
    end for
end function
```

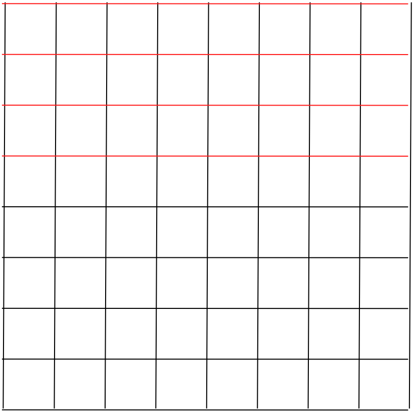
```
function WhileCycle(A, B, COV, offset, filename, loadableRows, i, readingOffset)
    while readingOffset < i + offset do
        oldA ← GetPortionOfA(filename, readingOffset, loadableRows)
        numRows ← rows(oldA)
        endVal ← min(numRows, i + offset − readingOffset) − 1
        for j ← 0 to endVal do
            oldRow ← oldA[j]
            if sum(oldRow) ∈ {0, M} then
                continue
            end if
            if intersect(oldRow, row) then
                B[j + readingOffset][i + offset] ← 0
            else
                I ← (offset + i, j + readingOffset)
                U ← union(oldRow, row)
                if sum(U) = M then
                    COV ← COV ∪ {I}
                    B[j + readingOffset][i + offset] ← 0
                else
                    B[j + readingOffset][i + offset] ← 1
                    intersect ← []
                    for k ← 0 to j + readingOffset − 1 do
                        if B[k][i + offset] = 1 and B[k][j + readingOffset] = 1 then
                            append(intersect, k)
                        end if
                    end for
                    if intersect ≠ ∅ then
                        kA ← GetSpecificRowsFromA(filename, intersect)
                        Explore(I, U, intersect, COV, kA, B, offset)
                    end if
                end if
            end if
        end for
    end while
end function
```

function $\textsc{Explore}(I, U, intersect, COV, kA, B, offset)$
  for $k \in intersect$ **do**
    $Itmp \leftarrow I \cup \{(k)\}$
    $kRow \leftarrow kA[k]$
    $Utmp \leftarrow \textsc{union}(U, kRow)$
    if $\textsc{sum}(Utmp) == M$ **then**
      $COV \leftarrow COV \cup \{Itmp\}$
    else
      $intersectTmp \leftarrow \{l \mid l \in intersect \text{ and } l < k \text{ and } B[l][k]\}$
      if $intersectTmp \neq \emptyset$ **then**
        $\textsc{Explore}(Itmp, Utmp, intersectTmp, COV, kA, B, offset)$
      end if
    end if
  end for
end function

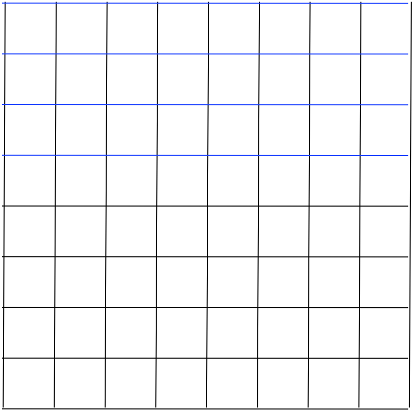

EC
SOLUTIONS



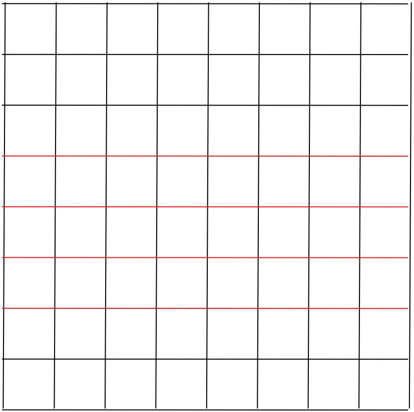
Offset	Reading offset
0	0



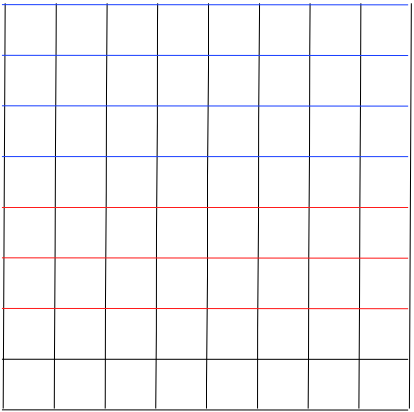
Offset	Reading offset
0	0



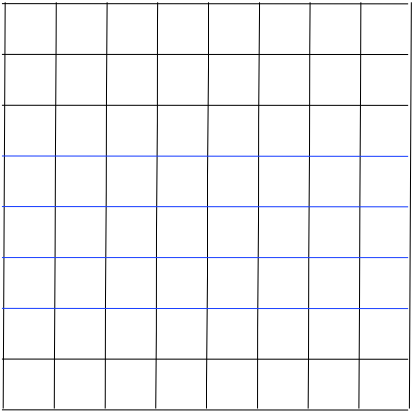
Offset	Reading offset
3	0



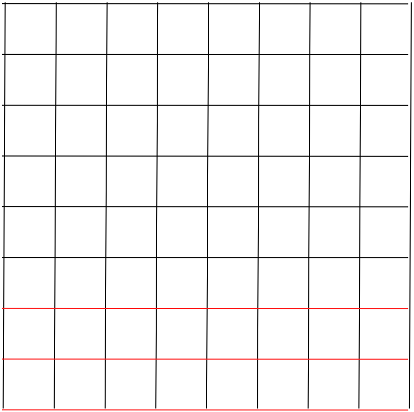
Offset	Reading offset
3	0



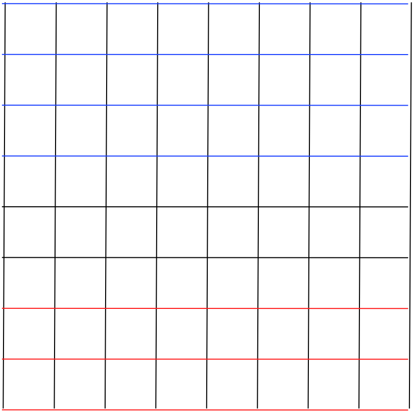
Offset	Reading offset
3	3



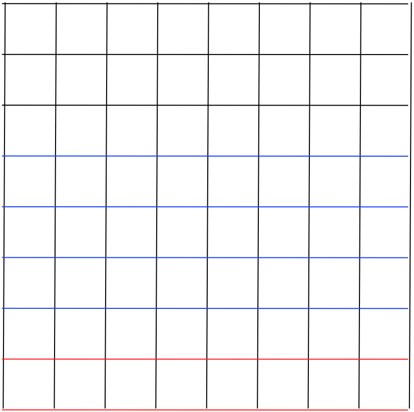
Offset	Reading offset
6	0



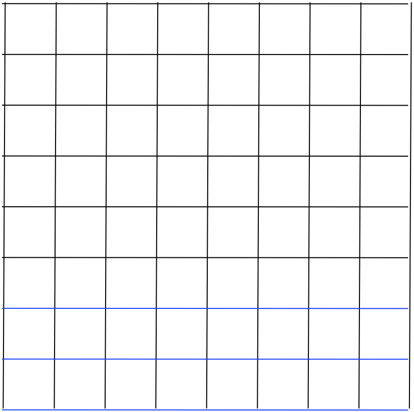
Offset	Reading offset
6	0



Offset	Reading offset
6	3



Offset	Reading offset
6	6



Sudoku [1]

Any instance of the sudoku problem can be mapped to an isntance of the exact cover problem. Independently of the size of the sudoku grid, which we call N (where N is the number of rows, the number of columns, the number of boxes and the number of possible elements to put in a cell), the exact cover problem will have 4 constraints:

1. every cell contains exactly one of the N possible elements;
2. every row has exactly one of the N possible elements in each of its N cells;
3. every column has exactly one of the N possible elements in each of its N cells;
4. every $N \times N$ box has exactly one of the N possible elements in each of its N cells.

[1] The following content is taken from http://www.ams.org/publicoutreach/feature-column/fcarc-kanoodle

Each of these constraints maps to N^2 columns of the matrix A, that is to say, every set of the exact cover problem has $4 \cdot N^2$ elements. The number of sets of the exact cover problem is N^3, where N^2 is the number of cells and N is the number of possible elements to put in a cell. This is how a 1 is mapped in a row of A, where
$r, c, n \in \{0, \ldots, N-1\}$, $i \in \{0, \ldots, 4 \cdot N^2 - 1\}$,
r is the row,
c is the column,
n is the element,
i is the index of the column of A:

1. a 1 in the i-th cell from the first N^2 cells of a row of A means that the r-th row and c-th cell of the sudoku grid has been filled with an element, without specifying which one, where $i = r \cdot N + c$;
2. a 1 in the i-th cell from the second N^2 cells of a row of A means that the r-th row has been filled with the n-th element, where $i = N^2 + r \cdot N + n$;
3. a 1 in the i-th cell from the third N^2 cells of a row of A means that the c-th column has been filled with the n-th element, where $i = 2 \cdot N^2 + c \cdot N + n$;
4. a 1 in the i-th cell from the fourth N^2 cells of a row of A means that the r-th row has been filled with the n-th element, where $i = 3 \cdot N^2 + \left(\left\lfloor \frac{r}{\sqrt{N}} \right\rfloor \sqrt{N} + \left\lfloor \frac{c}{\sqrt{N}} \right\rfloor \right) \cdot N + n$.

In a sudoku problem there always are some numbers already written in the grid, the presence of the numbers make the problem easier to solve, this reflects on the exact cover problem being easier to solve, the way it becomes easier to solve is that the number of sets to explore decreases. One possibility is to simply remove the sets that correspond to the cells that already have a number written in them but, in order to be able to write an algorithm that maps the solution to the exact cover problem back to the solution to the sudoku, it's easier to keep the sets and make them all 0s rows, we can do this because our algorithm ignores the only 0s rows, so it's like pruning the tree.

function $\textsc{SudokuToExactCover}(\mathit{sudoku}[], N)$ $\triangleright$ *sudoku* is an array of length N^2 containing the numbers written in the sudoku grid

$\quad nConstraints \leftarrow 4$

$\quad coverMatrix \leftarrow \textsc{zeros}(N^3, N^2 \cdot nConstraints)$

$\quad \textsc{FillA}(\text{coverMatrix, N})$

$\quad rowsToRemove \leftarrow \textsc{set}(\emptyset)$

$\quad \textsc{PruneRows}(rowsToRemove, sudoku, N)$

$\quad$ **for** $idx \in rowsToRemove$ **do**

$\qquad coverMatrix[idx] \leftarrow \textsc{zeros}(N^2 \cdot nConstraints)$

$\quad$ **end for**

$\quad$ **return** *coverMatrix*

end function

```
function FillA(coverMatrix, N)
    for r ← 0 to N − 1 do
        for c ← 0 to N − 1 do
            for n ← 0 to N − 1 do
                idx ← (r · N + c) · N + n
                coverMatrix[idx][r · N + c] ← 1
                coverMatrix[idx][N^2 + r · N + n] ← 1
                coverMatrix[idx][2 · N^2 + c · N + n] ← 1
                boxRow ← ⌊r/√N⌋
                boxCol ← ⌊c/√N⌋
                boxNum ← boxRow · √N + boxCol
                coverMatrix[idx][3 · N^2 + boxNum · N + n] ← 1
            end for
        end for
    end for
end function
```

function PRUNEROWS(*rowsToRemove*, *sudoku*, *N*)
    for $r \leftarrow 0$ to $N - 1$ **do**
        for $c \leftarrow 0$ to $N - 1$ **do**
            $symbol \leftarrow sudoku[r \cdot N + c] - 1$ ▷ Numbers in a sudoku usually start from 1, but in the matrix A they start from 0
            if *num* is not **NULL then**
                $startIdx \leftarrow (r \cdot N + c) \cdot N$
                for $i \leftarrow 0$ to $N - 1$ **do**
                    if $i \neq num$ **then**
                        $rowsToRemove \leftarrow rowsToRemove \cup \{startIdx + i\}$
                    end if
                end for
            end if
        end for
    end for
end function

function $\textsc{ExactCoverSolutionToSudoku}(ecSolution[], N)$

$sudokuSolution \leftarrow \textsc{zeros}(N, N)$

for $idx \in ecSolution$ **do**

$r \leftarrow \left\lfloor \frac{idx}{(N^2)} \right\rfloor$

$c \leftarrow \left\lfloor (\frac{idx}{N}) \mod N \right\rfloor$

$n \leftarrow (idx \mod N) + 1$

$sudokuSolution[r][c] \leftarrow n$

end for

return $sudokuSolution$

end function

Generating Exact Cover problems

Having a module that generates exact cover problems given the number of sets and the cardinality of the domain of the sets is useful for testing purposes. The idea of my choice to generate the exact cover problem is to ensure that the problem has at least a solution (that is, at least a set of sets that cover all the elements of the domain creating a partition), this is done by setting randomly M rows of the matrix A to the canonical base of $\mathbb{R}^M$. The precondition for this is that $N \geq M$. The rows of the matrix A that are not set to the canonical base are set randomly, each of their M cells is set to 1 or 0 with probability $p = 0.5$.

```
function GenerateExactCover(N, M)
    if N < M then
        return NULL
    end if
    A ← zeros(N, M)
    idxCanonicalBase[] ← RandomSample({0, ..., N − 1}, M)    ▷ Sample from {0, ..., N − 1} without replacement for M times
    for i ← 0 to M − 1 do
        A[idxCanonicalBase[i]][i] ← 1
    end for
    for i ← 0 to N − 1 do
        if i ∉ idxCanonicalBase then
            for j ← 0 to M − 1 do
                A[i][j] ← RandomChoice({0, 1})
            end for
        end if
    end for
    return A
end function
```