Exact Cover

Denis Festa

November 1, 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 slides discuss:

- memory management in Python and chosen data structures;
- the exploration of the solutions to different problems;
- the mechanism of loading portions of the problem to solve;
- conversion of the sudoku problem to an exact cover problem;
- generating random exact cover problems.

UNIVERSITY OF BRESCIA

Memory management

The language of choice for this implementation of the algorithm to solve the Exact Cover problem is Python, next slides recall the important aspects about memory management that need to be considered when deciding which data structures work better for the given task. In Python, when a value is assigned to a variable, an object is created, this causes the variable to require memory overhead: after allocating `i` $= 1$ or `i` $=$ `True`, the memory occupied by i is 28 bytes. The variable i does not memorize the raw value of 1 or `True`, it holds a pointer to the object that, among other quantities useful to the Python runtime environment, contains the raw value of 1. The quantities stored for each object are:

- reference count: information on how many references there are to the object, so that Python's garbage collector can reclaim it when there are no references left.
- Type information: data on the object's type, which Python uses to figure out what operations can be performed on the object.
- Object-specific data: for integers, this is the numerical value. For booleans, this could be a simple flag indicating true or false.

The default list in Python is a linked list, each position `list[i]` holds a pointer to the respective object (a boolean, an integer, anything...). The variable `list[]` in itself does not contain any raw value but somewhere else `#len(list)` objects (of size 28 bytes when dealing with integers or booleans) have been stored and referenced to by the pointers contained in `list`. Popular libraries such as `Numpy` or `bitarray` work differently, a `Numpy` array does not contain a list of references to objects, it contains a reference to an object that handles contiguous blocks of memory, the needed values (integers or booleans) are not stored as objects (each of them containing additional metadata that, together with the value, would reach 28 bytes), they are stored in a raw format, this way a lot of memory is saved. `Numpy` dedicates 1 byte to store a `True` or `False`, it is much less than 28 bytes, and `bitarray` does an even better job, it allocates 1 bit for a boolean, drastically reducing memory usage.

There is an important detail to clarify: when analyzing the space used by an element (the first) of the `bitarray` by calling `sys.getsizeof(my_bitarray[0])` (where `my_bitarray` is a `bitarray` containing booleans) the result is 28 bytes instead of a single bit, this might look contradictory with the previous claim that `bitarray` dedicates a single bit for a boolean, the contradiction is only apparent because when invoking `my_bitarray[0]` the Python runtime environment checks the single bit allocated by the `bitarray` object and creates on the fly a new object containing the corresponding boolean value required plus the overhead that every object in Python has.

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 since both integers and booleans are objects and both bring an important overhead with themselves. The `bitarray` library is the most efficient in terms of memory usage, however, `Numpy` offers many other advantages, such as speed (operations are written in `C`). Since complex matrix operations are not required in this algorithm, I think that the advantage `Numpy` can give in terms of speed is less valuable than the advantage in terms of memory consumption offered by `bitarray`. The presented implementation has a focus on memory efficiency, the slide *EC wrapper* describes another choice to sacrifice time efficiency over memory efficiency which is mainly about loading portions of the file containing the problem rather than the whole file at once.

UNIVERSITY OF BRESCIA

Figure (8) shows different data structures used to store a list of boolean values. The first two scenarios show that the memory required to store a list of 1000 pointers to objects of type `boolean` and `integer` is the same (8056 bytes), each object referenced by each pointer requires 28 bytes, and these 28 bytes are to be considered as memory occupation. No advantage comes from using `booleans` instead of `integers`. The `Numpy` array requires 1112 bytes to store the raw values of the 1000 booleans (around 8.896 bits per boolean), then, when a single boolean is accessed, the Python runtime environment creates an object on the fly containing the boolean value plus the overhead, and it's 25 bytes: these 25 bytes are not to be considered as memory occupation, they are not used to store the values, they are used temporarly allocated just as the value is accessed. The `bitarray` array requires 216 bytes to store the raw values of the 1000 booleans (around 1.728 bits per boolean), then when a single boolean is accessed, the Python runtime, similarly as before, creates the object on the fly.

Built-in array of integers

```
[66] ✓ 0.0s
```

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

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

Built-in array of booleans

```
[67] ✓ 0.0s
```

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

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

Numpy array of bits

```
[68] ✓ 0.0s
```

```
p = np.ones(1000, dtype=bool)
print("Size of p, Size of p[0]")
sys.getsizeof(p), sys.getsizeof(p[0])
```

```
Size of p, Size of p[0]
(1112, 25)
```

Bitarray

```
[69] ✓ 0.0s
```

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

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

UNIVERSITY OF BRESCIA

Built-in matrix of integers

```
[6]  1 N = 30
     2 M = 1000
     ✓ 0.0s
```

```
[7]  1 a = [[1]*M for _ in range(N)]
     2 print("Size of a, Size of a[0], Size of a[0][0]")
     3 sys.getsizeof(a), sys.getsizeof(a[0]), sys.getsizeof(a[0][0])
     ✓ 0.0s
```

```
Size of a, Size of a[0], Size of a[0][0]
(312, 8056, 28)
```

Built-in matrix of booleans

```
[8]  1 b = [[True]*M for _ in range(N)]
     2 print("Size of b, Size of b[0], Size of b[0][0]")
     3 sys.getsizeof(b), sys.getsizeof(b[0]), sys.getsizeof(b[0][0])
     ✓ 0.0s
```

```
Size of b, Size of b[0], Size of b[0][0]
(312, 8056, 28)
```

Numpy matrix of bits

```
[9]  1 p = np.ones((N, M), dtype=bool)
     2 print("Size of p, Size of p[0], Size of p[0][0]")
     3 sys.getsizeof(p), sys.getsizeof(p[0]), sys.getsizeof(p[0][0])
     ✓ 0.0s
```

```
Size of p, Size of p[0], Size of p[0][0]
(30128, 112, 25)
```

Figure: The `Numpy` matrix requires 30128 bytes to store the raw values of the 30000 booleans (around 8.03 bits per boolean).

UNIVERSITY OF BRESCIA

Bitarray matrix

```
[10]
1 c = [bitarray('1'*M) for _ in range(N)]
2 print("Size of c, Size of c[0], Size of c[0][0]")
3 sys.getsizeof(c), sys.getsizeof(c[0]), sys.getsizeof(c[0][0])
✓ 0.0s

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

(312, 216, 28)
```

```
[11]
1 d = bitarray('1'*M*N)
2 print("Size of d, Size of d[0]")
3 sys.getsizeof(d), sys.getsizeof(d[0])
✓ 0.0s

Size of d, Size of d[0]

(4060, 28)
```

Figure: Notice the difference among the two cases. In the first scenario we have a list of bitarrays, although the bitarrays are stored in a contiguous block of memory, a different object is dedicated to each bitarray, creating an overhead. In the second scenario we have a single bitarray, a single object is dedicated to the bitarray, which has the shape of a flattened matrix, it occupies 4060 bytes to store 30000 booleans (around 1.082 bits per boolean). The second choice is the one I made.

I tried to profile the code with `memory-profiler` to find whether the memory occupation advantages I expected to gain from choosing one data structure over the other were actually realized, but I could not see any significant difference in memory usage while executing the code, I guess it is 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 cardinalities of the domain (columns of the matrix A) and different numbers 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 automatically generated instances of the problem. 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 to contain the duration of the experiments.

UNIVERSITY OF BRESCIA

Solutions found

It is intuitive that the number of solutions found increases with the number of sets and it decrease with the cardinality of the domain. In this case the intuition is confirmed by the solutions found (3).

Figure: The number of solutions found increases with the number of sets and decreases with the cardinality of the domain.

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 (4). 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 greater number of rows and a greater number of columns. Notice that the graphs in figure (4) follow the same pattern as the graphs in figure (6), which shows the time required to find all the solutions for the same problems. Why, fixing the number of rows, a small number of columns implies a greater number of explored nodes?

This behaviour is probably due to the fact that the rows of the matrix A are generated by sampling from a uniform distribution the values $\{0, 1\}$ (the slide *Generating Exact Cover problems* explains how the exact cover problems used here were generated), when the number of columns is small each row is more likely to be compatible with other rows, so it has to be explored more deeply, whereas when the number of columns is high, given that each row contains around half 1s and half 0s, it is more likely that a row is incompatible with other rows, so less rows have to be explored. Later in the presentation, in the section *More or less 1s*, an alternative is presented, which makes use of different distributions to generate the rows of the matrix A.

Figure: The number of explored nodes increases with the number of sets and increases when the cardinality of the domain is too small or too large. The only interpretation found is that the number of explored nodes increases with the complexity of the problem.

Explorable nodes

The implemented algorithm chooses to prune many subtrees that will not lead to a plausible solution. Professor Marina Zanella showed that if the algorithm was blind and didn't prune any subtree, then the number of nodes to be explored would be:

$$\sum_{i=1}^{N}\sum_{j=1}^{i}\binom{i}{j} = 2^N - 1.$$

In figure (5) the enormous difference between the number of explored nodes and the number of explorable nodes is shown. The number of explorable nodes is not always exactly $2^N - 1$ because the actual computation to find the explorable nodes takes into account only rows of A different from an all-zero row (which would be, by definition, compatible with any other row).

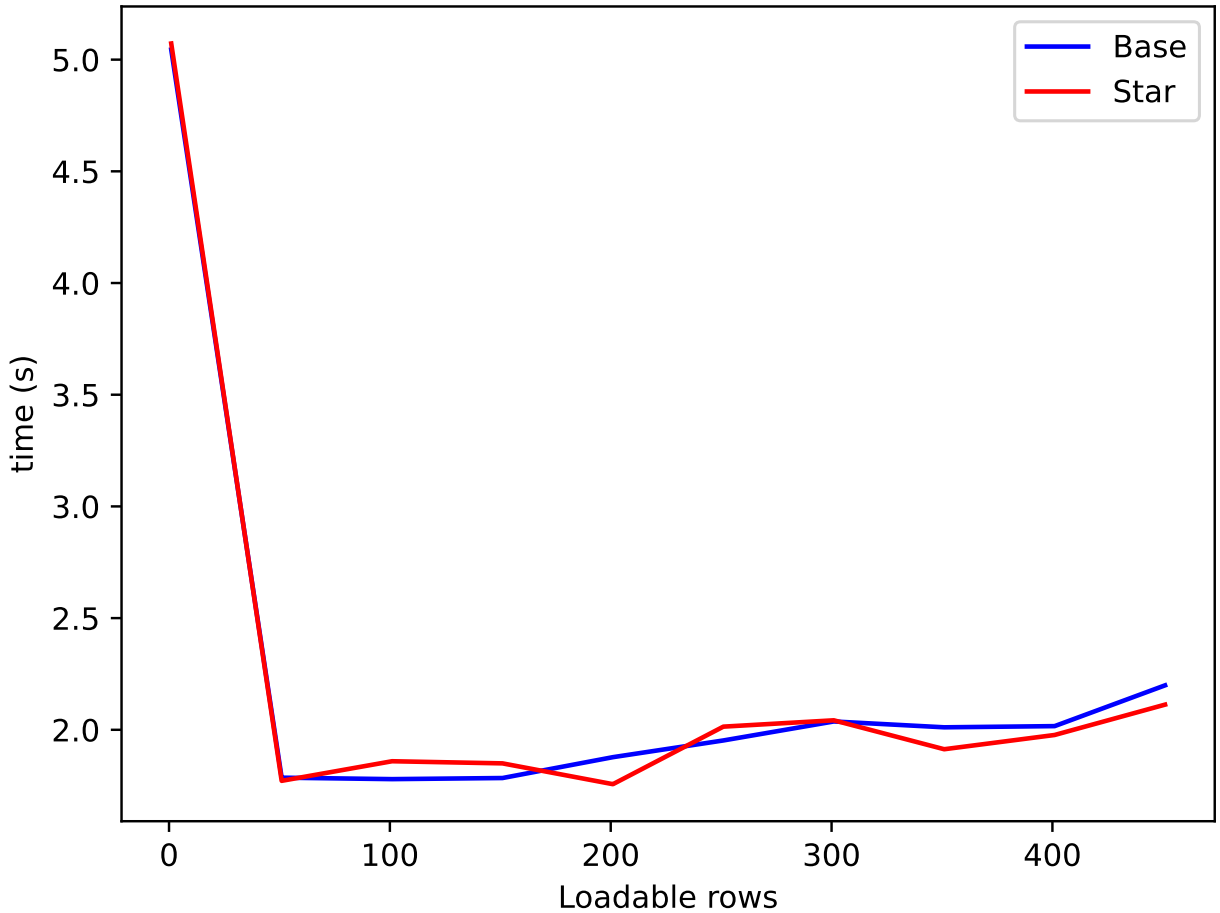
UNIVERSITY OF BRESCIA

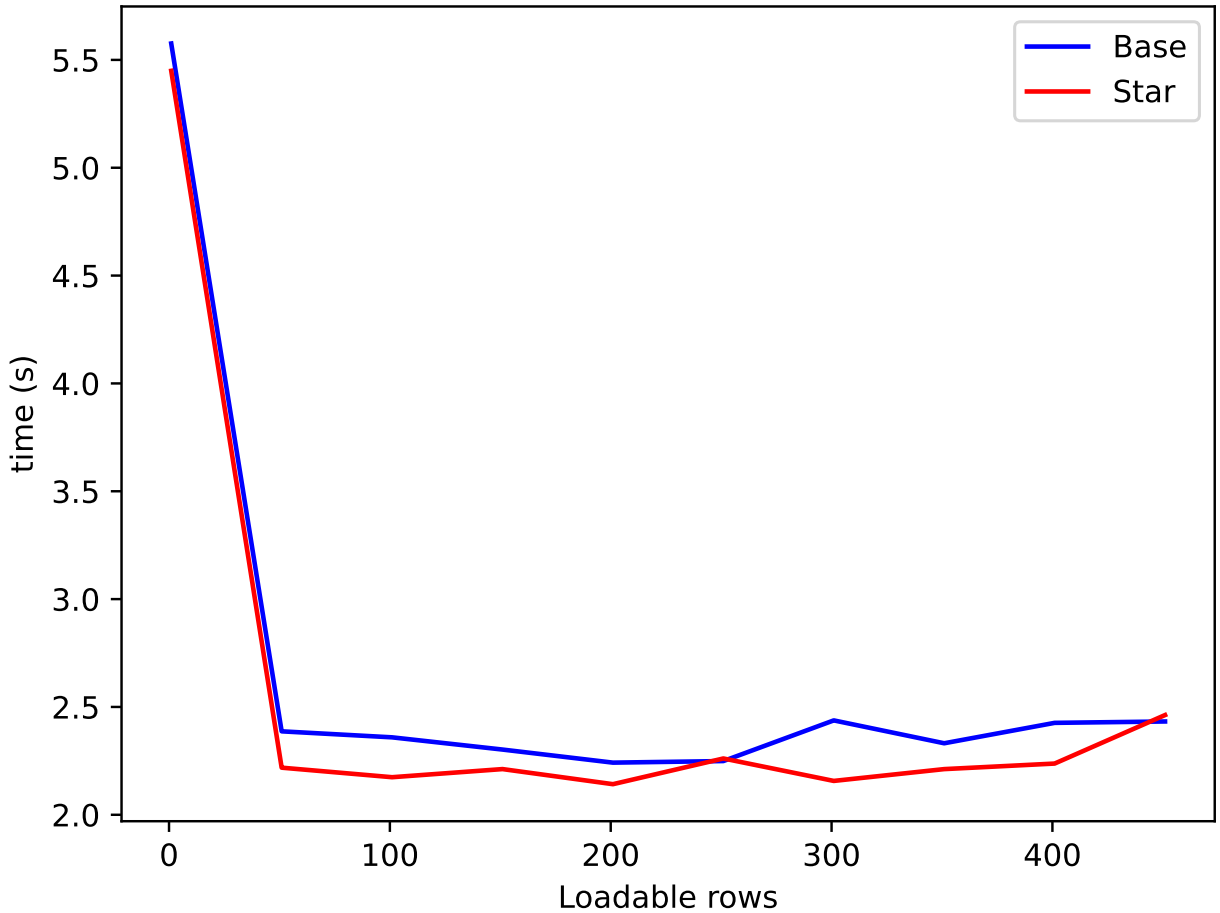
Figure: The number of nodes that would be explored by a blind algorithm is much greater than the number of nodes that are actually explored by the implemented algorithm.

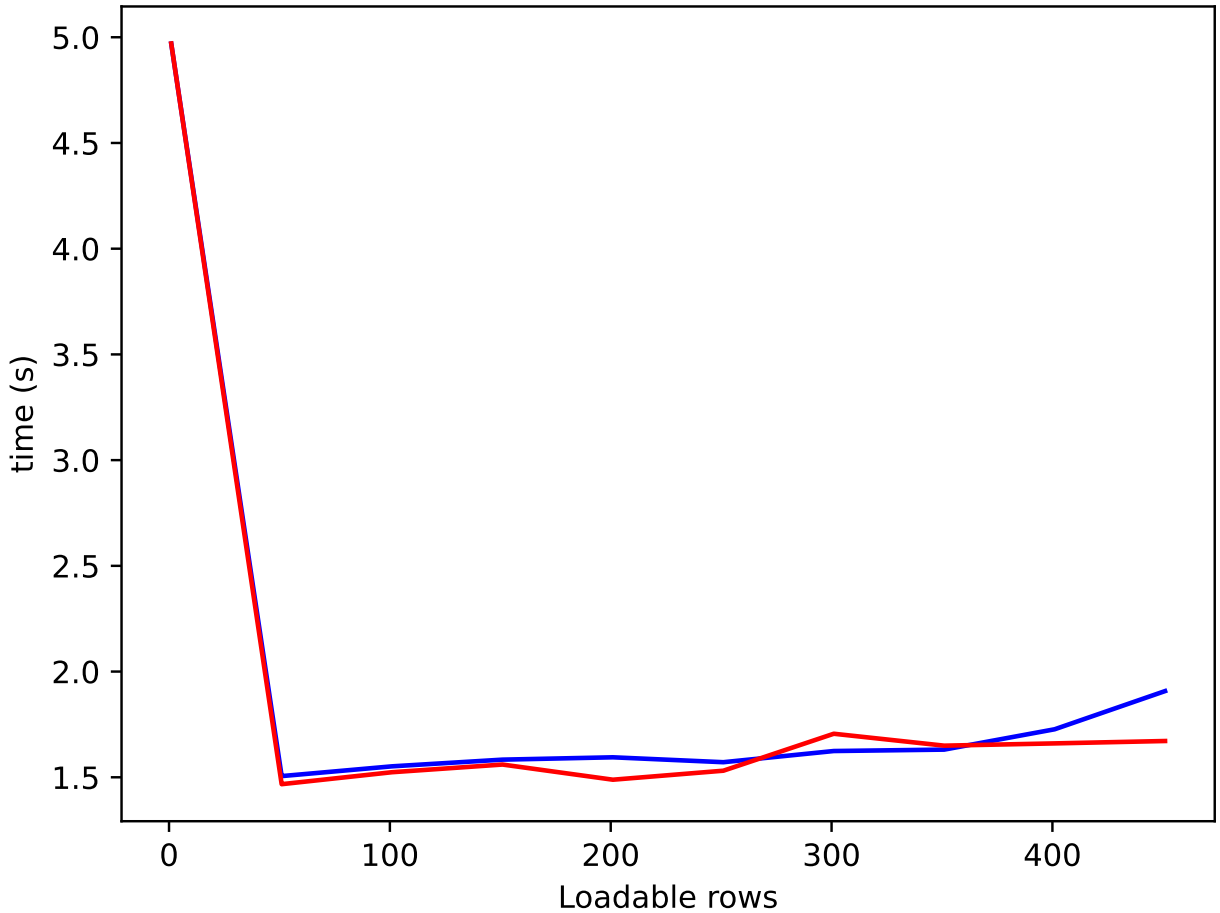
UNIVERSITY OF BRESCIA

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 (this mechanism is described in the slide *EC wrapper*). 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 (21), (22) and (23) show something different. 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. To make the order in which the problems were solved as influential as possible on the results, the problems were first solved with an increasing number of loadable rows (figure (21)), then with a decreasing number of loadable rows (figure (22)) and finally with a random number of loadable rows (figure (23)).


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


Base
Star
5.5
5.0
4.5
4.0
3.5
3.0
2.5
2.0
time (s)
0
100
200
300
400
Loadable rows


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

Base and *plus* algorithms

The results show that the *plus* algorithm, most of the times, is slightly faster than the *base* algorithm, but it remains unclear why sometimes this is not the case. The same unpredictable behaviour is shown in figure (6).

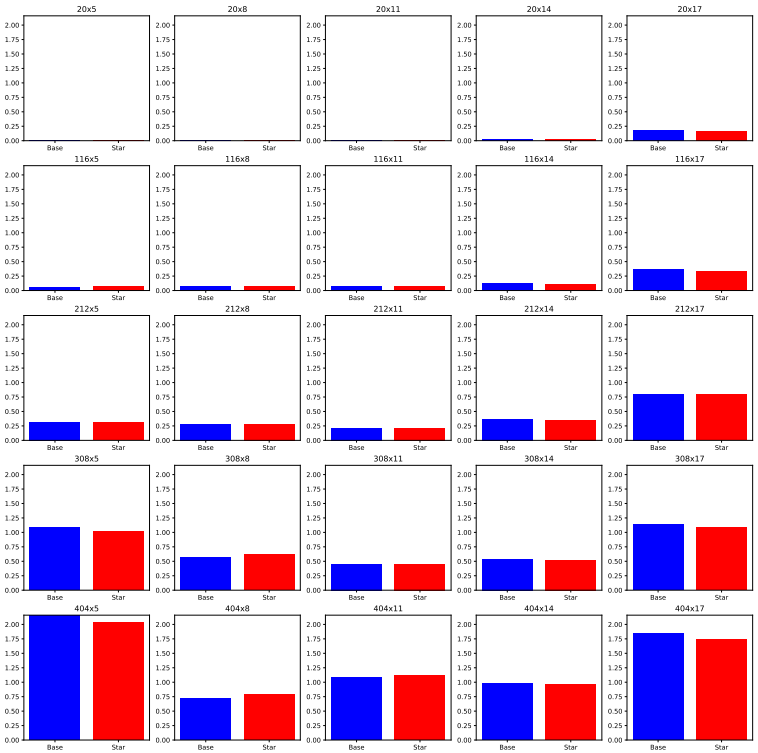


Figure: This figure shows the time required to find all the solutions for the same problems whose results were shown before, in particular it follows the same pattern in figure (4). The *plus* algorithm is slightly faster than the *base* algorithm, but it remains unclear why sometimes this is not the case.

UNIVERSITY OF BRESCIA

A series of manual tests on different dimensions of the chunk of loadable rows on a larger problem (1000 rows and 15 columns) brought to the following results:

- 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 (7), (8) and (9) show the results of the experiments and, the larger the number of columns, the more our intuition is confirmed.

Figure: The number of explored nodes increases with the number of rows and decreases with the number of 1s in each row, this is because the more 1s there are, the less likely it is for the problem to have compatible rows, hence, the less sets are to be explored, the more 0s there are, the more likely it is for the problem to have compatible rows, hence, the more sets are to be explored.

Figure: Once the problem has enough columns, the number of solutions found increases with the number of 0s in each row, the reason is the same as before (figure (7)).

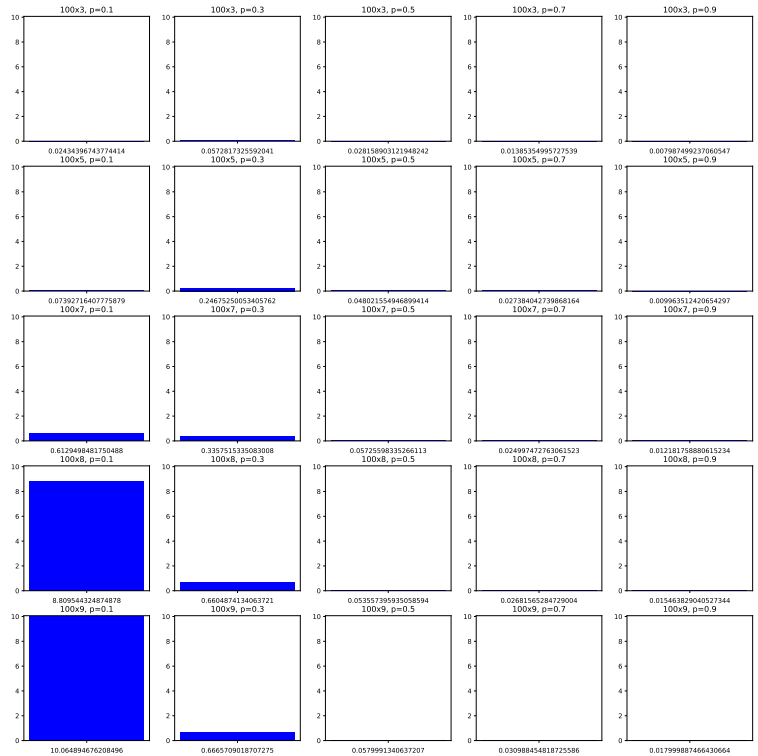


Figure: Again, the time required to find the solutions follows the same pattern as the number of explored nodes, shown in figure (7).

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 reading offset (the same colors will be used in the schema and the animation to follow.)

UNIVERSITY OF BRESCIA

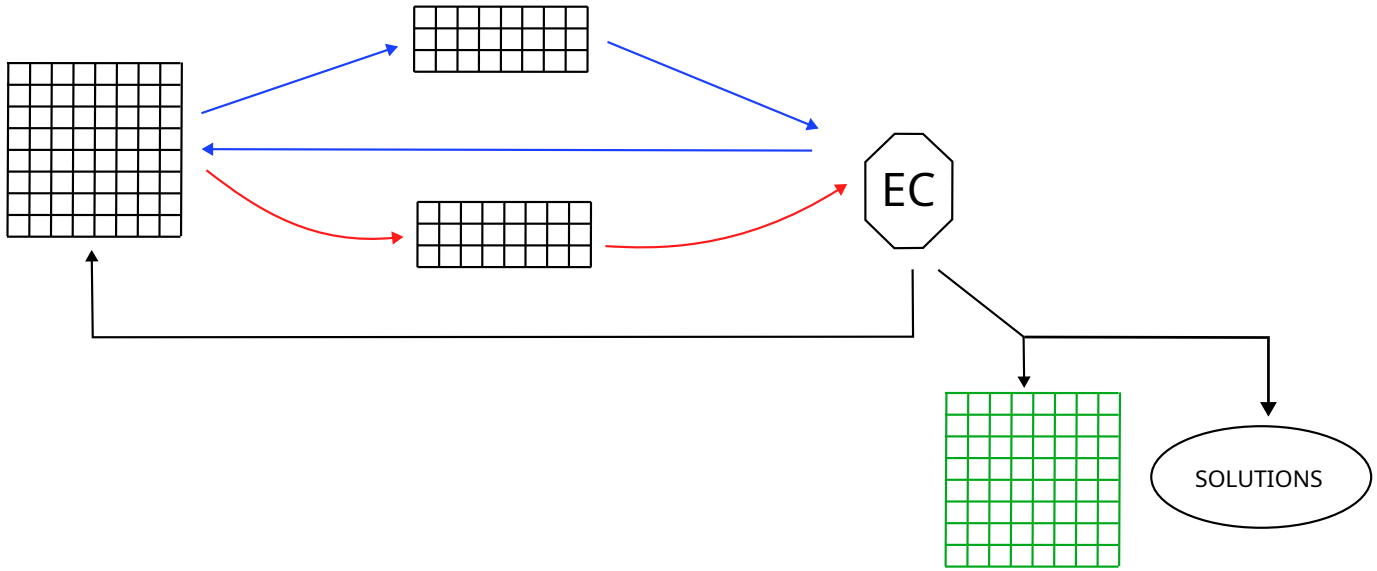
- offset is the index of the first row of the portion of the matrix A that has been passed to the EC algorithm from outside
- reading offset is the index of the first row of the portion of the matrix A that is being read by the EC algorithm, reading offset will always start from 0 and will be incremented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

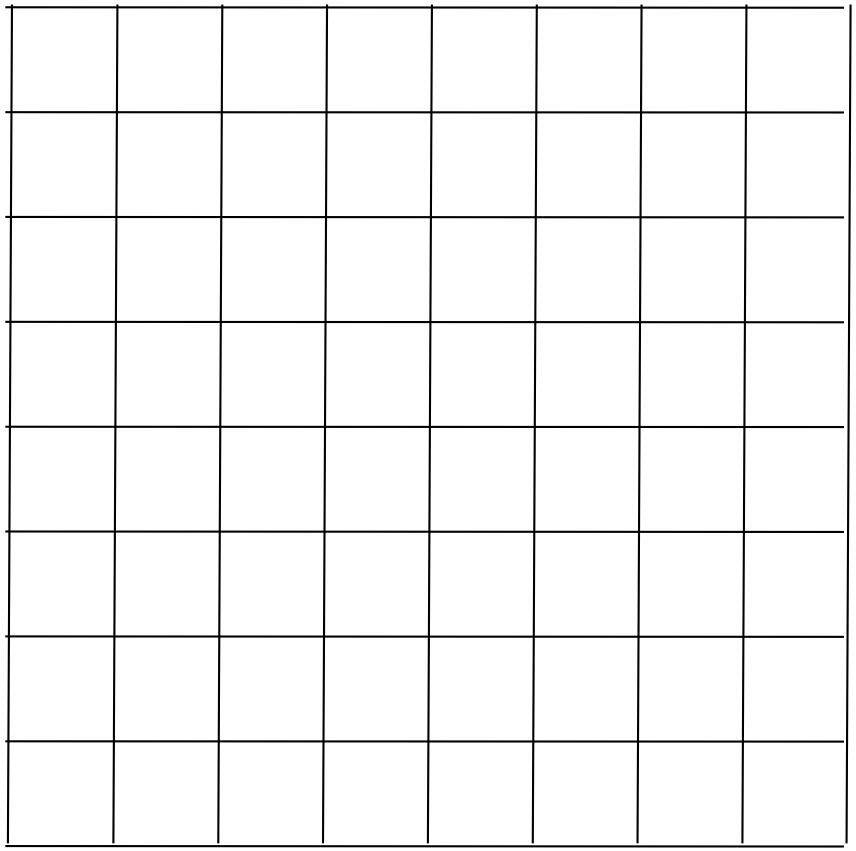
```
function IncrementalExactCover(filename)
    rows ← GetRows(filename)
    columns ← GetColumns(filename)
    B ← zeros(rows, columns)
    COV ← set(∅)
    offset ← 0
    while true do
        A ← GetPortionOfA(filename, offset, loadableRows)
        if A is empty then
            break
        end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

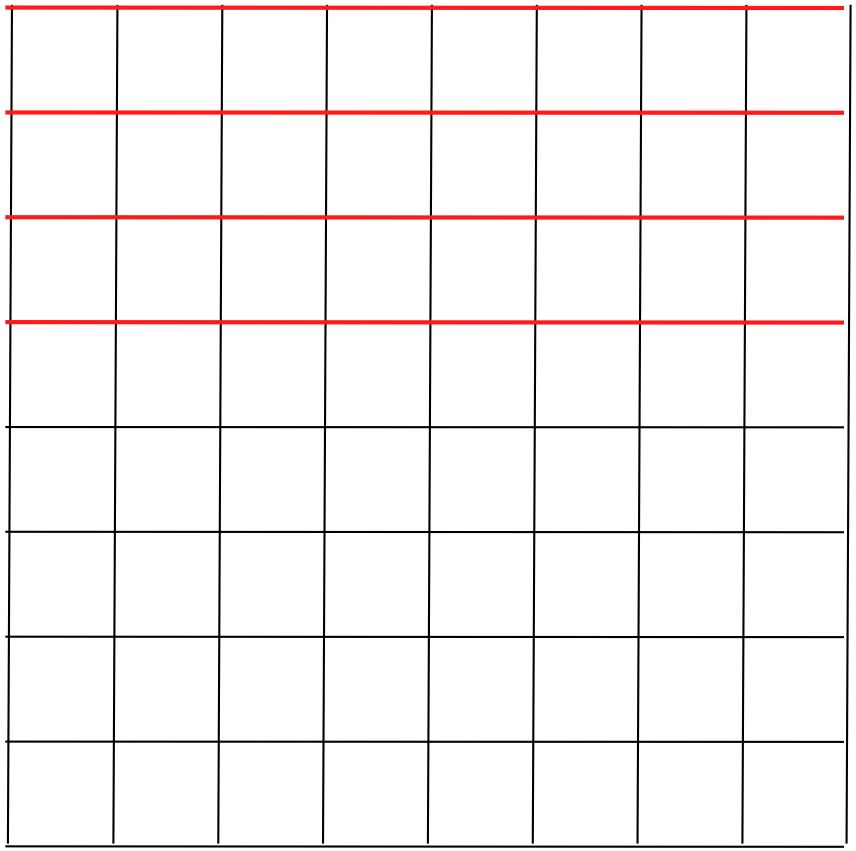
```
function WhileCycle(A, B, COV, offset, filename, loadableRows, i, readingOffset)
    while readingOffset < i + offset do
        oldA ← GetPortionOfA(filename, readingOffset, loadableRows)
        numRows ← rows(oldA)
        endVal ← min(numRows, i + offset − readingOffset) − 1
        for j ← 0 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 EXPLORE(I, U, $intersect$, COV, kA, B, $offset$)
  for $k \in intersect$ **do**
    $Itmp \leftarrow I \cup \{(k)\}$
    $kRow \leftarrow kA[k]$
    $Utmp \leftarrow \text{UNION}(U, kRow)$
    if $\text{SUM}(Utmp) == M$ **then**
      $COV \leftarrow COV \cup \{Itmp\}$
    else
      $intersectTmp \leftarrow \{I \mid I \in intersect \text{ and } I < k \text{ and } B[I][k]\}$
      if $intersectTmp \neq \emptyset$ **then**
        EXPLORE($Itmp$, $Utmp$, $intersectTmp$, COV, kA, B, $offset$)
      end if
    end if
  end for
end function

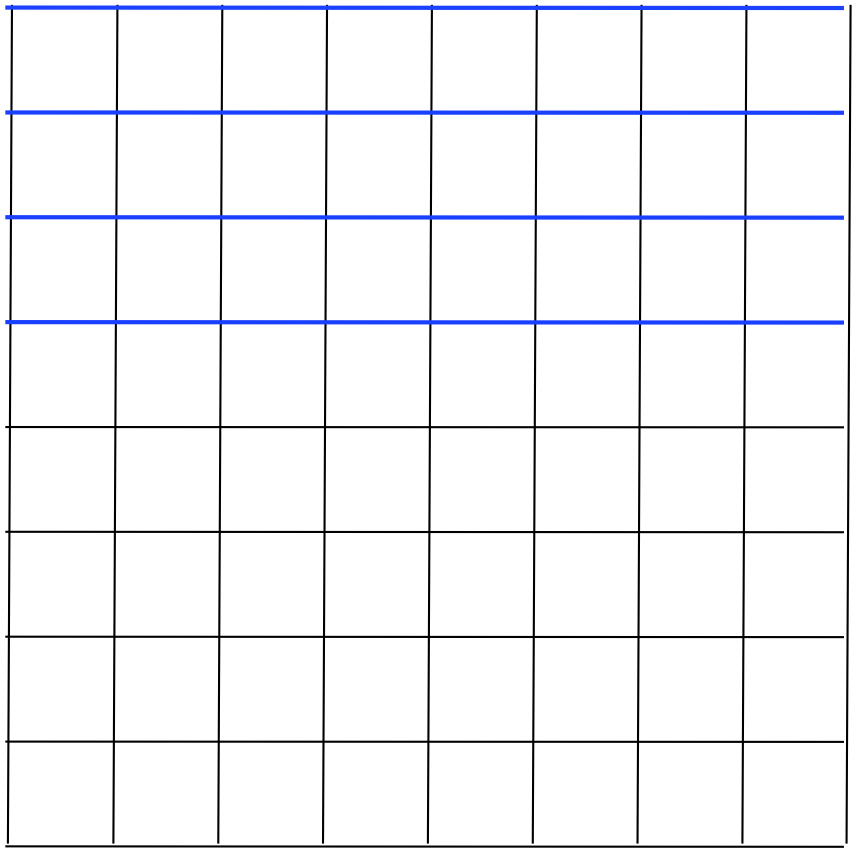

EC
SOLUTIONS



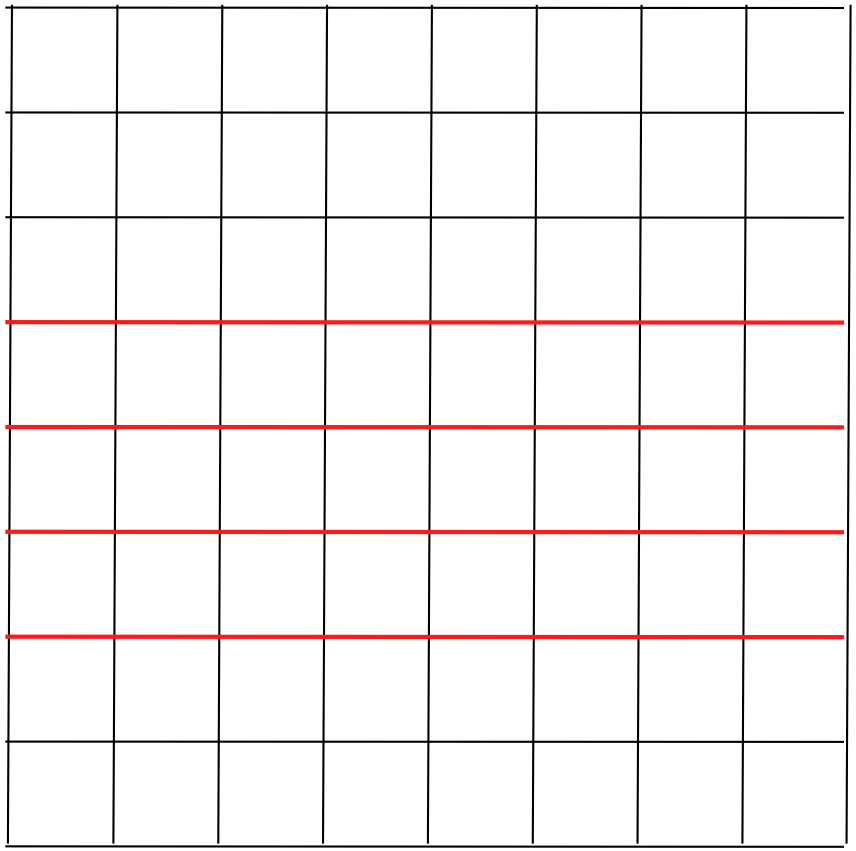
Offset	Reading offset
0	0



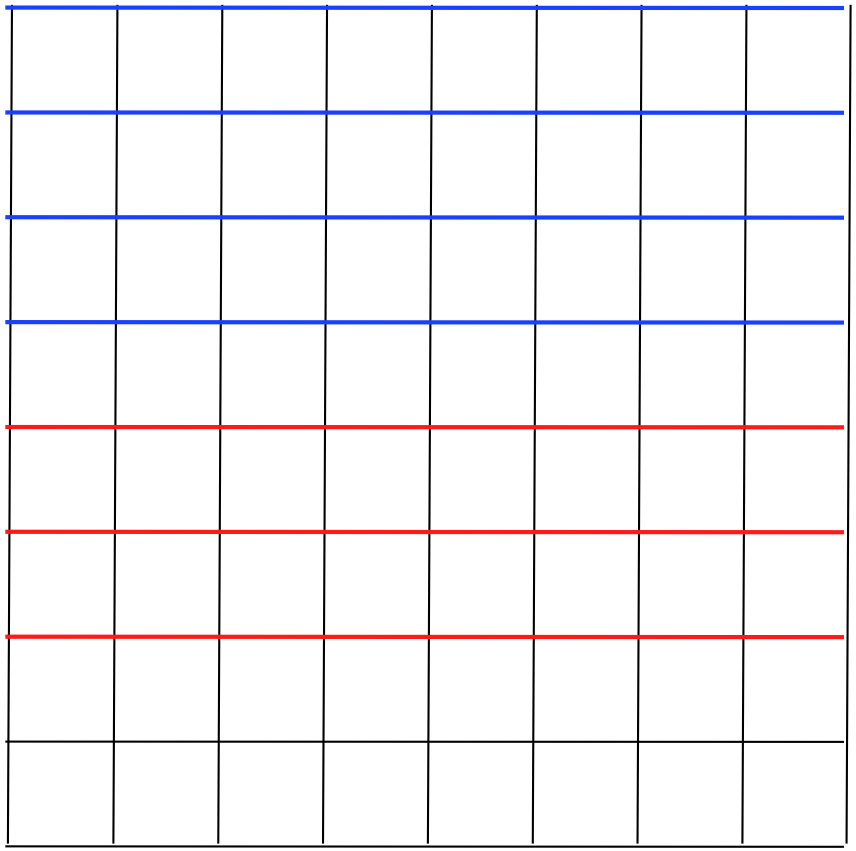
Offset	Reading offset
0	0



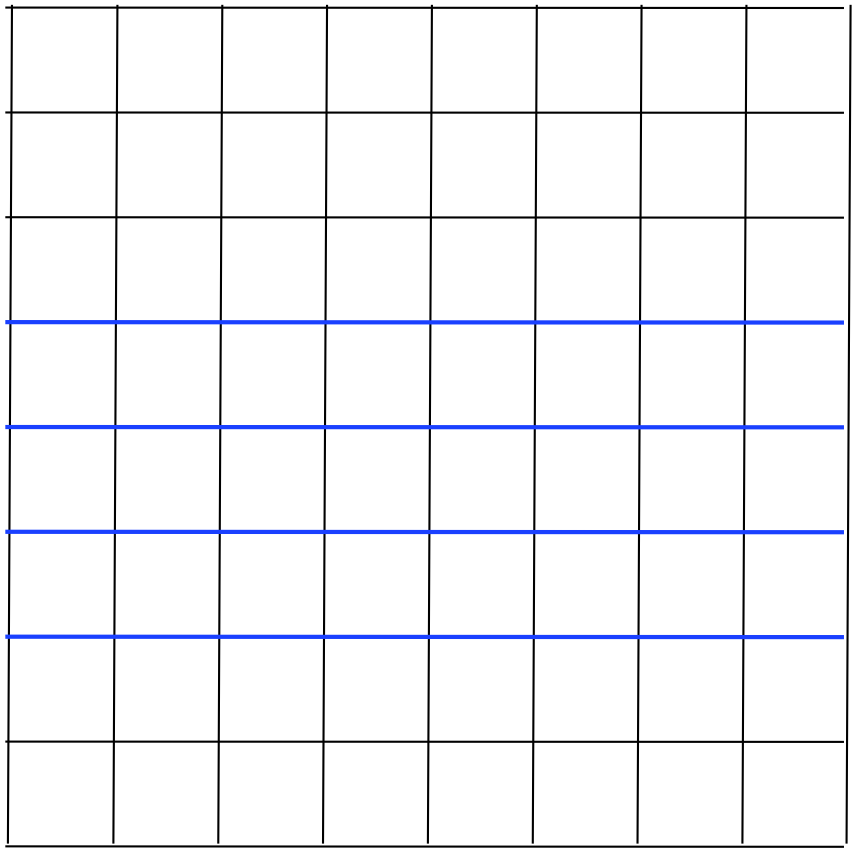
Offset	Reading offset
3	0



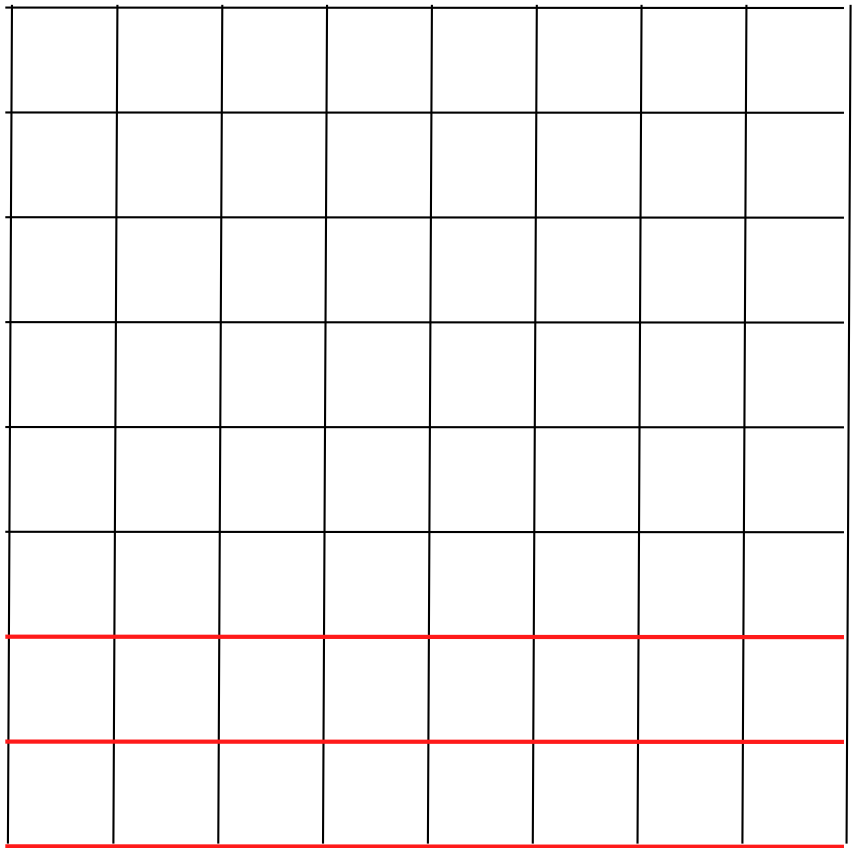
Offset	Reading offset
3	0



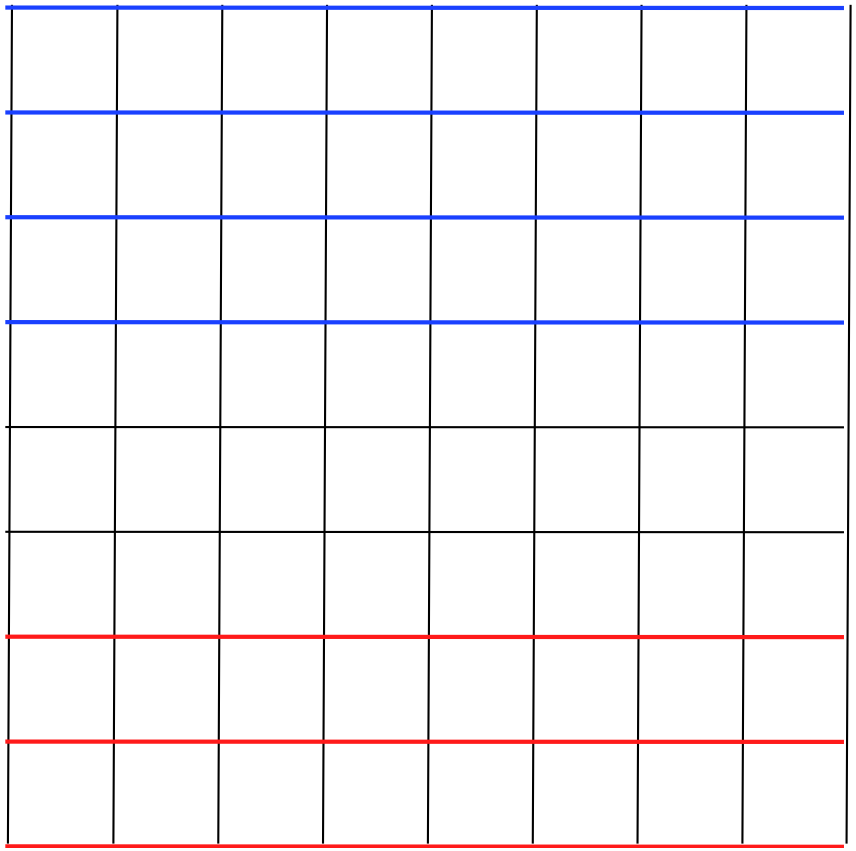
Offset	Reading offset
3	3



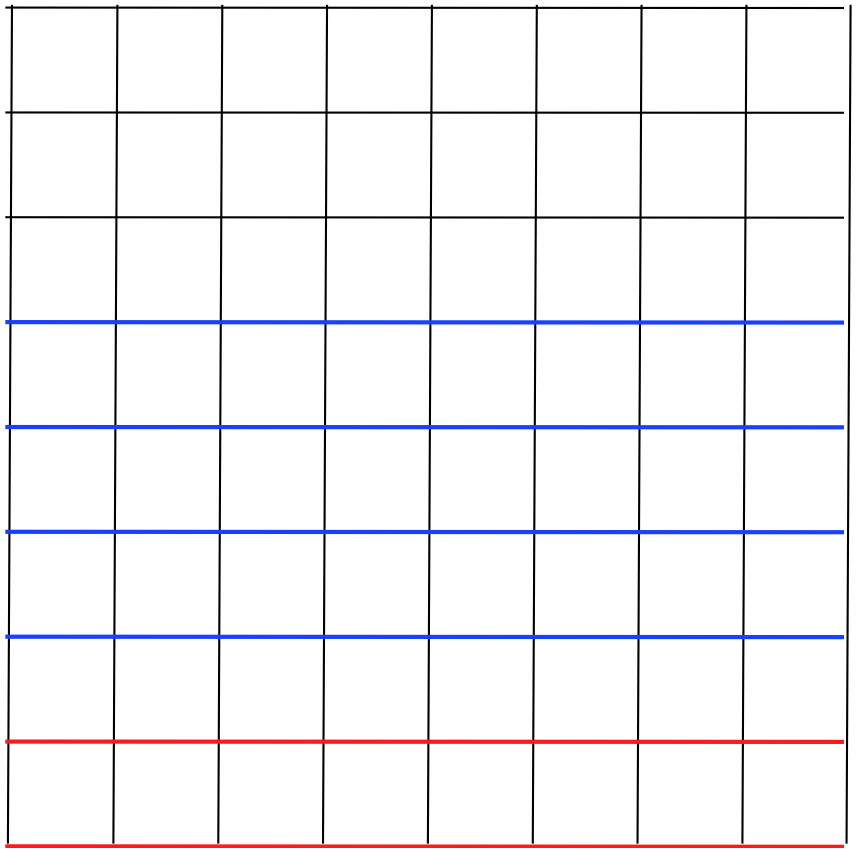
Offset	Reading offset
6	0



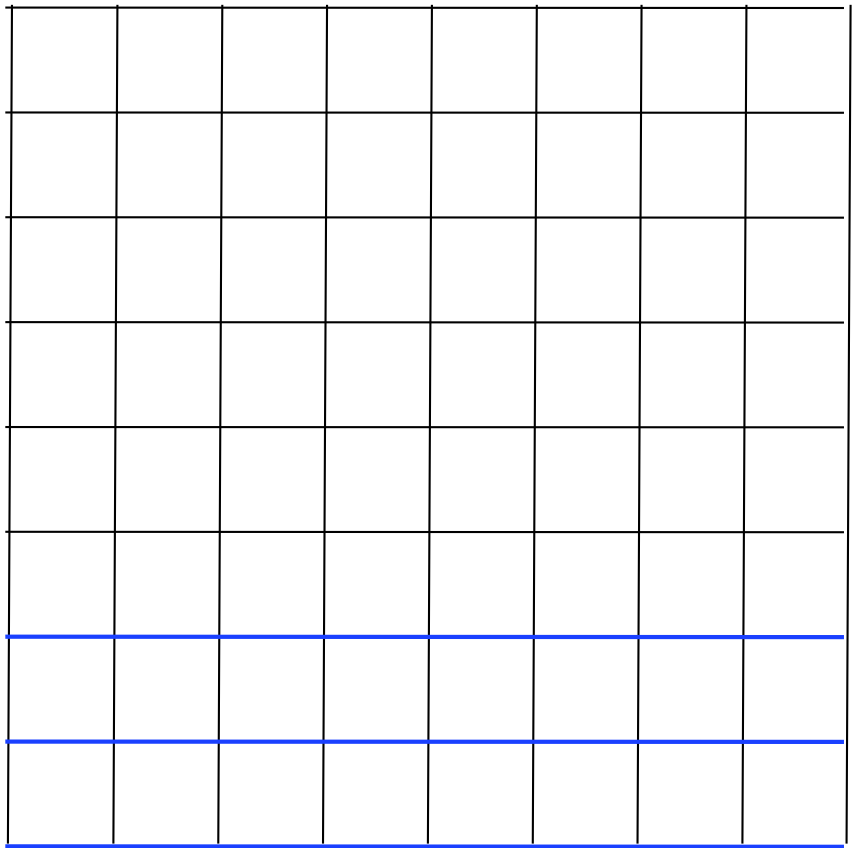
Offset	Reading offset
6	0



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`

UNIVERSITY OF BRESCIA

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 SudokuToExactCover(*sudoku*[], *N*) ▷ *sudoku* is an array of length N^2 containing the numbers written in the sudoku grid

nConstraints ← 4

coverMatrix ← zeros(N^3, $N^2 \cdot$ *nConstraints*)

FillA(coverMatrix, N)

rowsToRemove ← set(∅)

PruneRows(*rowsToRemove*, *sudoku*, *N*)

for *idx* ∈ *rowsToRemove* **do**

coverMatrix[*idx*] ← zeros($N^2 \cdot$ *nConstraints*)

end for

return *coverMatrix*

end function

function FILLA($coverMatrix$, N)
  for $r \leftarrow 0$ to $N-1$ **do**
    for $c \leftarrow 0$ to $N-1$ **do**
      for $n \leftarrow 0$ to $N-1$ **do**
        $idx \leftarrow (r \cdot N + c) \cdot N + n$
        $coverMatrix[idx][r \cdot N + c] \leftarrow 1$
        $coverMatrix[idx][N^2 + r \cdot N + n] \leftarrow 1$
        $coverMatrix[idx][2 \cdot N^2 + c \cdot N + n] \leftarrow 1$
        $boxRow \leftarrow \lfloor r/\sqrt{N} \rfloor$
        $boxCol \leftarrow \lfloor c/\sqrt{N} \rfloor$
        $b \leftarrow boxRow \cdot \sqrt{N} + boxCol$
        $coverMatrix[idx][3 \cdot N^2 + b \cdot N + n] \leftarrow 1$
      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 ExactCoverSolutionToSudoku($ecSolution[], N$)
    $sudokuSolution \leftarrow \text{zeros}(N, N)$
    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 a custom probability, the default one is $p = 0.5$.

function GENERATEEXACTCOVER(N, M)
    if $N < M$ **then**
        return NULL
    end if
    $A \leftarrow$ ZEROS(N, M)
    $idxCanonicalBase[] \leftarrow$ RANDOMSAMPLE$(\{0, \ldots, N-1\}, M)$ ▷ Sample from $\{0, \ldots, N-1\}$ without replacement for M times
    for $i \leftarrow 0$ to $M-1$ **do**
        $A[idxCanonicalBase[i]][i] \leftarrow 1$
    end for
    for $i \leftarrow 0$ to $N-1$ **do**
        if $i \notin idxCanonicalBase$ **then**
            for $j \leftarrow 0$ to $M-1$ **do**
                $A[i][j] \leftarrow$ RANDOMCHOICE$(\{0, 1\})$
            end for
        end if
    end for
    return A
end function