Exact Cover

Denis Festa

October 14, 2023

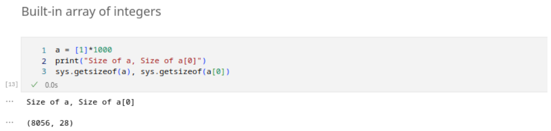
Introduction

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`.

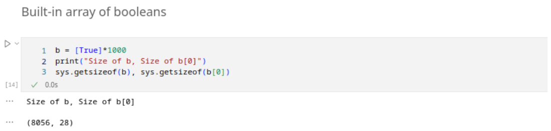


Built-in array of integers

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

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

(8056, 28)
```

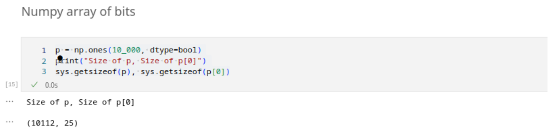


Built-in array of booleans

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

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

(8056, 28)
```

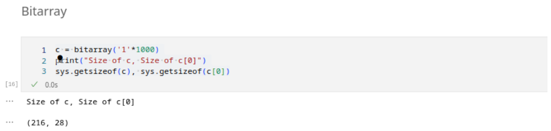


Numpy array of bits

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

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

(10112, 25)
```

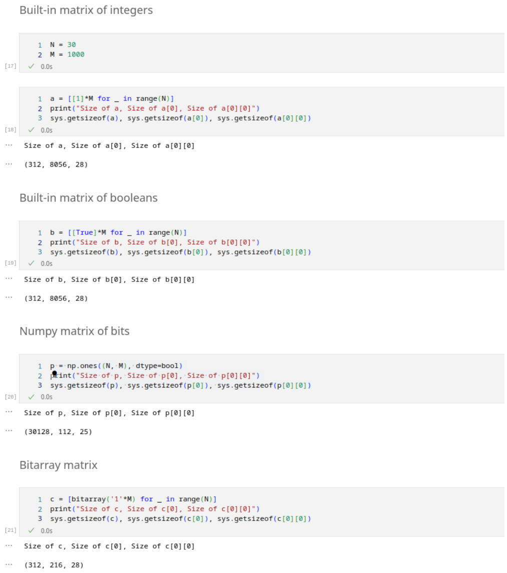


Bitarray

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

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

(216, 28)
```



Built-in matrix of integers

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

[17] ✓ 0.0s

```python
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

```python
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

```python
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

```python
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

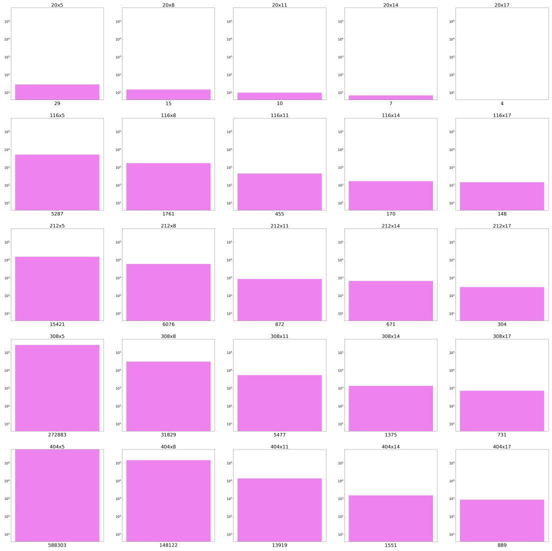
Different choices of the cardinality of the domain (columns of the matrix A) and the number of sets (rows of the matrix A, rows and columns of the matrix B) lead to different values of:

- 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.

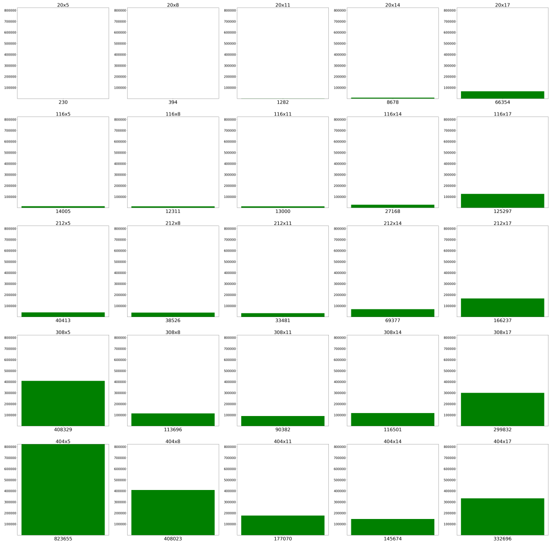
Solutions found

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).


20x5
29
20x8
15
20x11
10
20x14
7
20x17
4
116x5
5287
116x8
1761
116x11
455
116x14
170
116x17
148
212x5
15421
212x8
6076
212x11
872
212x14
671
212x17
304
308x5
272883
308x8
31829
308x11
5477
308x14
1375
308x17
731
404x5
588303
404x8
148122
404x11
13919
404x14
1551
404x17
889

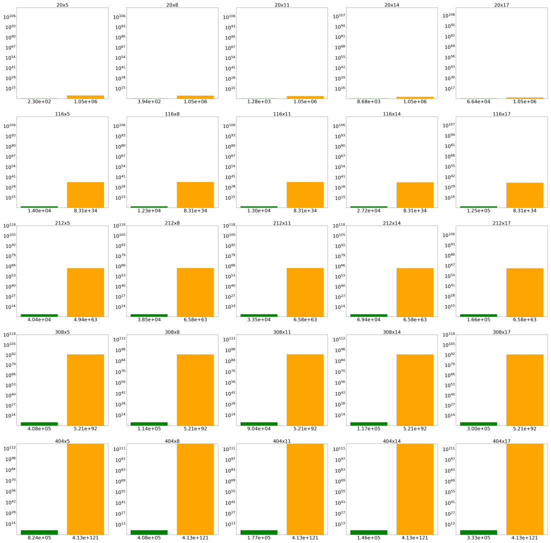
Explored nodes

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
394
20x11
1282
20x14
8678
20x17
66354
116x5
14005
116x8
12311
116x11
13000
116x14
27168
116x17
125297
212x5
40413
212x8
38526
212x11
33481
212x14
69377
212x17
166237
308x5
408329
308x8
113696
308x11
90382
308x14
116501
308x17
299832
404x5
823655
404x8
408023
404x11
177070
404x14
145674
404x17
332696
800000
700000
600000
500000
400000
300000
200000
100000

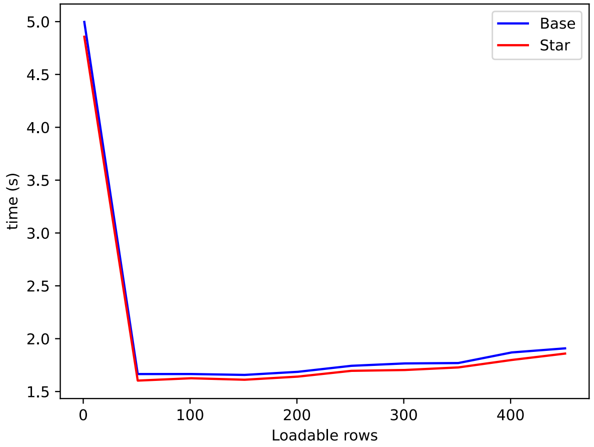
Explorable nodes

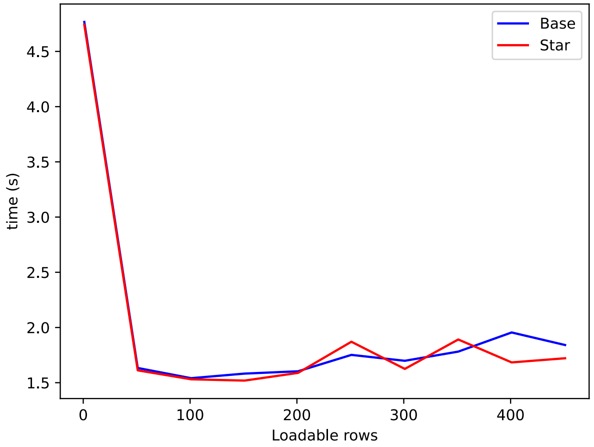
The implemented algorithm chooses to prune the nodes in the tree that cannot lead to a plausible solution. If the algorithm were to explore all the possible nodes, then for the *i*-th set the number of nodes to explore would be 2^i, hence, if n is the number of sets (rows of A), the number of nodes to 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
1.05e+06
20x8
3.94e+02
1.05e+06
20x11
1.28e+03
1.05e+06
20x14
8.68e+03
1.05e+06
20x17
6.64e+04
1.05e+06
116x5
1.40e+04
8.31e+34
116x8
1.23e+04
8.31e+34
116x11
1.30e+04
8.31e+34
116x14
2.72e+04
8.31e+34
116x17
1.25e+05
8.31e+34
212x5
4.04e+04
4.94e+63
212x8
3.85e+04
6.58e+63
212x11
3.35e+04
6.58e+63
212x14
6.94e+04
6.58e+63
212x17
1.66e+05
6.58e+63
308x5
4.08e+05
5.21e+92
308x8
1.14e+05
5.21e+92
308x11
9.04e+04
5.21e+92
308x14
1.17e+05
5.21e+92
308x17
3.00e+05
5.21e+92
404x5
8.24e+05
4.13e+121
404x8
4.08e+05
4.13e+121
404x11
1.77e+05
4.13e+121
404x14
1.46e+05
4.13e+121
404x17
3.33e+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
time (s)
Loadable rows
4.5
4.0
3.5
3.0
2.5
2.0
1.5
0
100
200
300
400

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

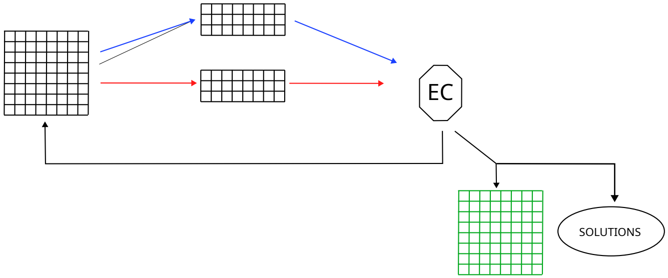
EC wrapper

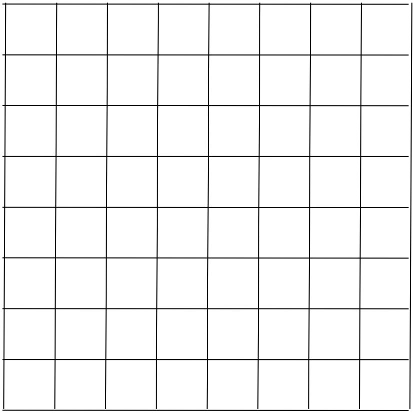
In the following slides the mechanism wrapping the EC algorithm is presented. The idea is to load a portion of the matrix A, execute the EC algorithm and store the solutions found in a set, then load the next portion of the matrix A, execute the EC algorithm and grow the set of solutions. The important 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.

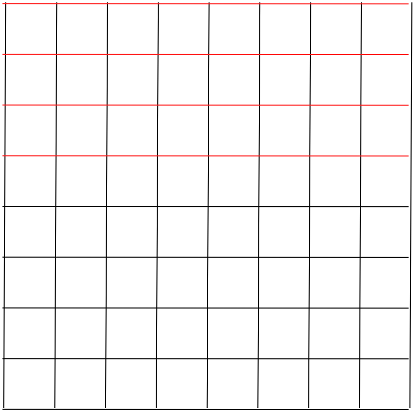
```
function $\textsc{IncrementalExactCover}(\mathit{filename})$
    $\mathit{rows} \leftarrow \textsc{GetRows}(\mathit{filename})$
    $\mathit{columns} \leftarrow \textsc{GetColumns}(\mathit{filename})$
    $B \leftarrow \textsc{zeros}(\mathit{rows}, \mathit{columns})$
    $COV \leftarrow \textsc{set}(\emptyset)$
    $\mathit{offset} \leftarrow 0$
    while true do
        $A \leftarrow \textsc{GetA}(\mathit{filename}, \mathit{offset}, \mathit{loadableRows})$
        if $A$ is empty then
            break
        end if
        $\textsc{EC}(A,\ B,\ COV,\ \mathit{offset},\ \mathit{filename},\ \mathit{loadableRows})$
    end while
    return $COV$
end function
```

```
1: function $\mathrm{EC}(A, B, COV, offset, filename, loadableRows)$
2:     $N \leftarrow \textsc{rows}(A)$                ▷ Number of rows of A
3:     $M \leftarrow \textsc{columns}(A)$             ▷ Number of columns of A
4:     $COV \leftarrow \textsc{set}(\emptyset)$
5:     $reading\_offset \leftarrow 0$
6:     $new\_COV \leftarrow$
   $\textsc{EC\_plus}(A, B, offset, reading\_offset, COV, rows, columns, loadable\_rows$
7:     return $new\_COV$
8: end function
```

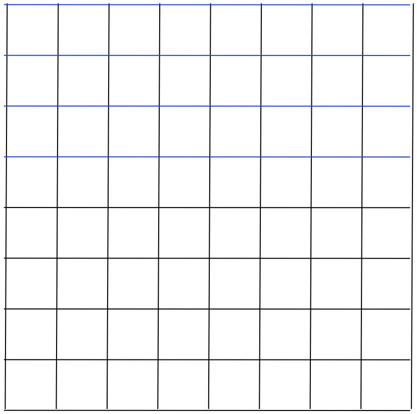

EC
SOLUTIONS



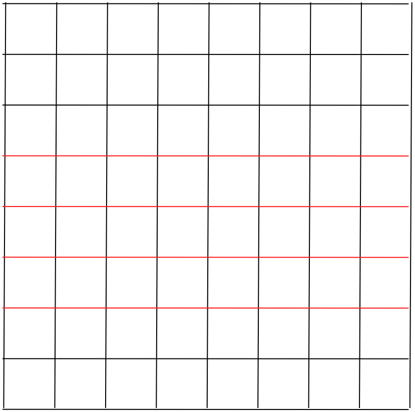
Offset	Reading offset
0	0



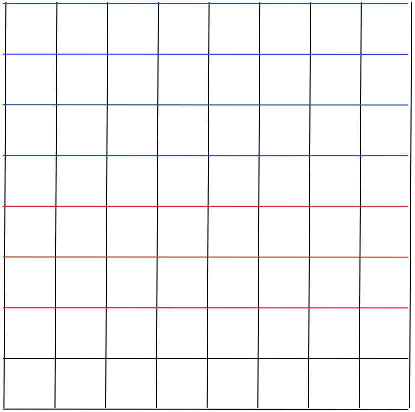
Offset	Reading offset
0	0



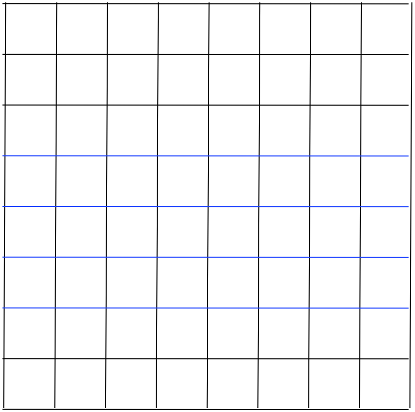
Offset	Reading offset
3	0



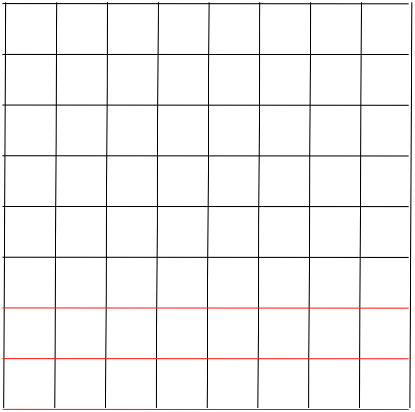
Offset	Reading offset
3	0



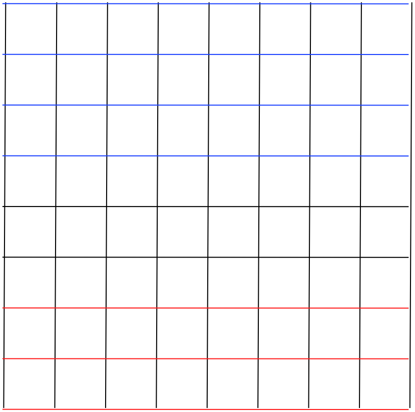
Offset	Reading offset
3	3



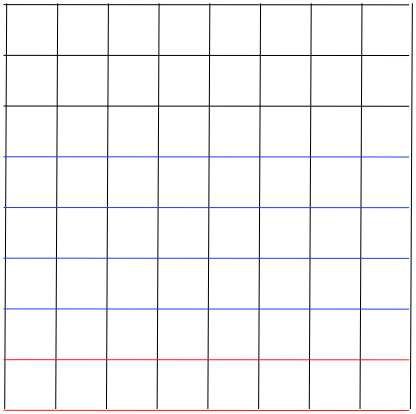
Offset	Reading offset
6	0



Offset	Reading offset
6	0



Offset	Reading offset
6	3



Offset	Reading offset
6	6