Exact Cover

Denis Festa

October 16, 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

```
1  a = [1]*1000
2  print("Size of a, Size of a[0]")
3  sys.getsizeof(a), sys.getsizeof(a[0])
```
[13] ✓ 0.0s

··· Size of a, Size of a[0]

··· (8056, 28)

Built-in array of booleans

```
1  b = [True]*1000
2  print("Size of b, Size of b[0]")
3  sys.getsizeof(b), sys.getsizeof(b[0])
```
[14] ✓ 0.0s

··· Size of b, Size of b[0]

··· (8056, 28)

Numpy array of bits

```
1  p = np.ones(10_000, dtype=bool)
2  print("Size of p, Size of p[0]")
3  sys.getsizeof(p), sys.getsizeof(p[0])
```
[15] ✓ 0.0s

··· Size of p, Size of p[0]

··· (10112, 25)

Bitarray

```
1  c = bitarray('1'*1000)
2  print("Size of c, Size of c[0]")
3  sys.getsizeof(c), sys.getsizeof(c[0])
```
[16] ✓ 0.0s

··· Size of c, Size of c[0]

··· (216, 28)

Built-in matrix of integers

```
1  N = 30
2  M = 1000
```
[17] ✓ 0.0s

```
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])
```
[18] ✓ 0.0s

⋯   Size of a, Size of a[0], Size of a[0][0]

⋯   (312, 8056, 28)

Built-in matrix of booleans

```
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])
```
[19] ✓ 0.0s

⋯   Size of b, Size of b[0], Size of b[0][0]

⋯   (312, 8056, 28)

Numpy matrix of bits

```
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])
```
[18] ✓ 0.0s

⋯   Size of p, Size of p[0], Size of p[0][0]

⋯   (30128, 112, 25)

Bitarray matrix

```
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])
```
[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).

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.

Exact Cover

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).

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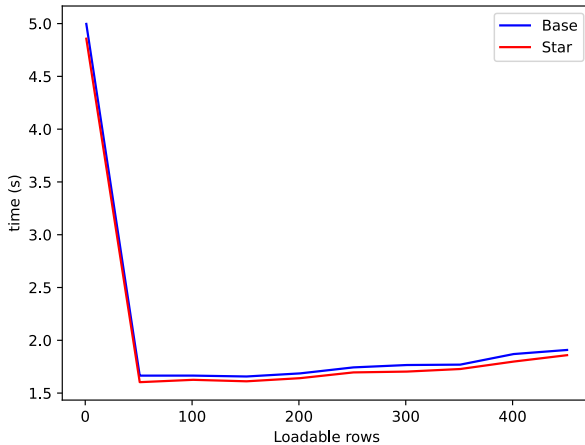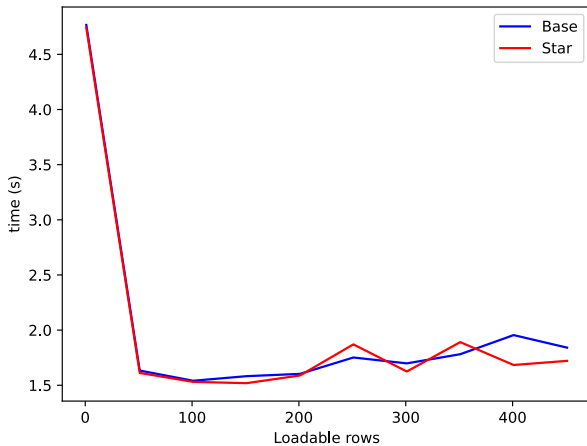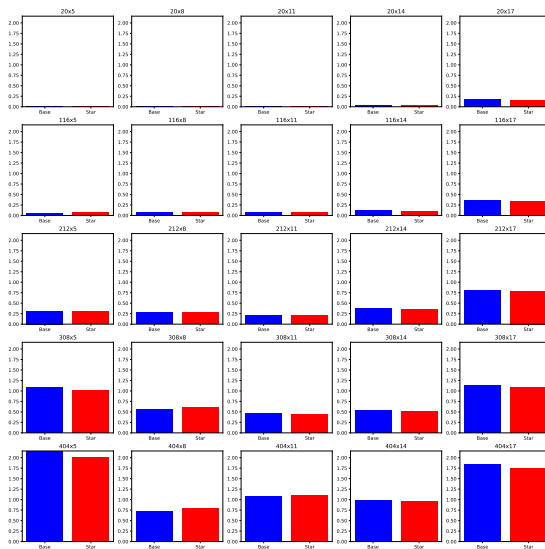
I manually tried to test the behaviour of the algorithm for different dimensions of the chunk on a larger problem (1000 rows and 15 columns) and

- 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 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*)
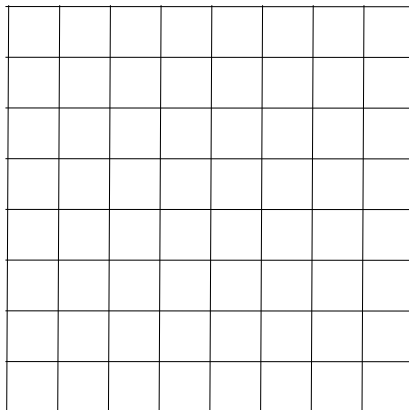   **end if**
            end if
         end if
      end for
   end while
end function

```
function EXPLORE(I, U, intersect, COV, kA, B, offset)
    for k ∈ intersect do
        Itmp ← I ∪ {(k)}
        kRow ← kA[k]
        Utmp ← UNION(U, kRow)
        if SUM(Utmp) == M then
            COV ← COV ∪ {Itmp}
        else
            intersectTmp ← {l | l ∈ intersect and l < k and B[l][k]}
            if intersectTmp ≠ ∅ then
                EXPLORE(Itmp, Utmp, intersectTmp, COV, kA, B, offset)
            end if
        end if
    end for
end function
```
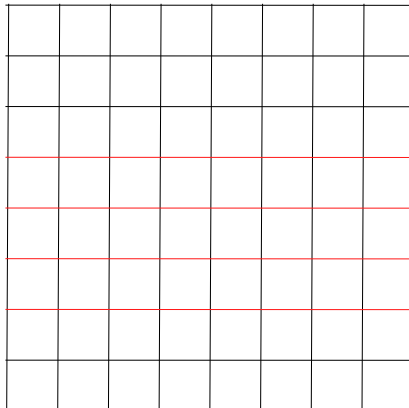
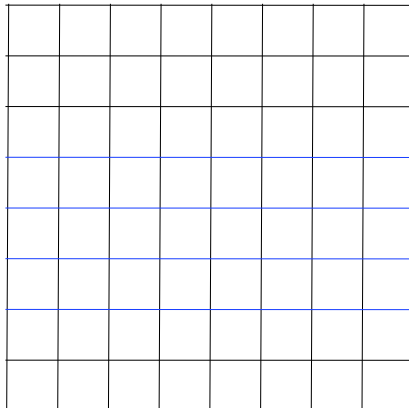| Offset	Reading offset
0	0

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
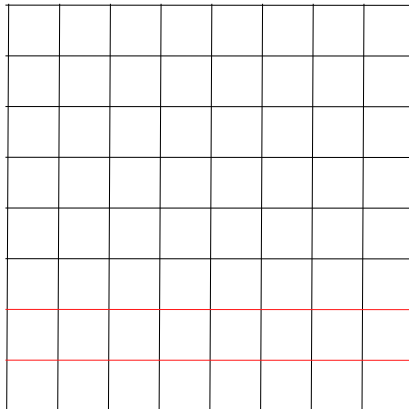
Offset	Reading offset
6	0

Offset	Reading offset
6	3
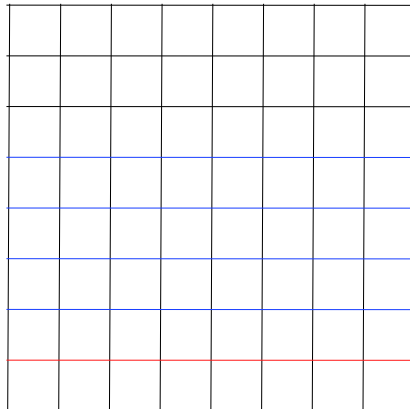
Offset	Reading offset
6	6

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 $r, c, b, n \in \{0, \ldots, N-1\}$, $i \in \{0, \ldots, 4 \cdot N^2 - 1\}$, r is the row, c is the column, b is the box, n is the element, i is the index of the column:

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,

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 $i = N^2 + r \cdot N + n$; where $i = r \cdot N + c$;

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 $i = 3 \cdot N^2 + b \cdot N + n$.