Exact Cover

Denis Festa

October 12, 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).
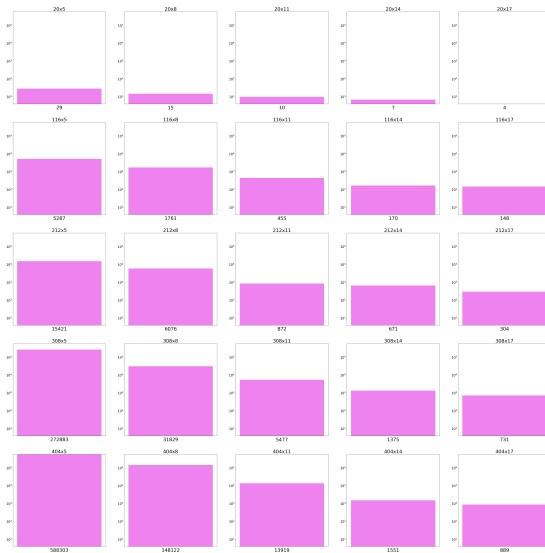
Explored nodes

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.
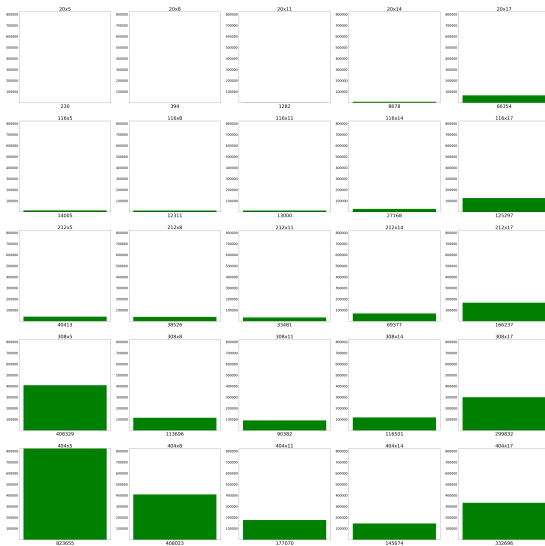
The implemented algorithm chooses to prune the nodes in the tree that cannot lead to a plausible solution. If the algorithm were to explore all the possible nodes, then for the i-th set the number of nodes to explore would be 2^i, hence, if n is the number of sets (rows of A), the number of nodes to 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).