SatisfyMe!

Pedro Brandimarte

November 24, 2018

1 Objective

Given a set $U = \{x_1, x_2, \ldots, x_n\}$ of boolean variables (literals) and a boolean formula k-CNF (conjunctive normal form composed by "and" of clauses in $\{c_1, c_2, \ldots, c_m\}$, where each clause consists in "or" of k literals belonging to the set U plus respective negations), the main objective of this program consists on searching attributions (true and false) for the set of literals such that the k-CNF is satisfied (true). This is a NP-complete problem known as k-SAT.

The k-SAT problem can be interpreted as a generalized cover set problem and, therefore, can be solved by a backtracking algorithm based on the D. Knuth's "Dancing Links X" algorithm [1].

Besides solving the k-SAT problem, the code is also able to determine the solutions, if any, of an exact cover set problem.

2 Input and Output

For the k-CNF, the program reads an input from `stdin` containing integer numbers, where the first line corresponds to "k n m" (amount of literals, cardinal of U and number of clauses, respectively) and the following lines correspond to the clauses, where a number i represents the variable x_i and $-i$ its negation $\neg x_i$.

The program output is send to `stdout`. With the command line option `-n` the code prints out the number of different attributions that satisfy the input. With the option `-N`, besides printing the number of attributions that satisfy the input, the code also prints all these attributions, line by line and in lexicographical order. In case no option is passed through the command line, the code prints 1 if the k-CNF is satisfiable or 0, otherwise.

The command line option `-C` indicates that the input contains an exact cover set problem. In this case, the input represents the problem of covering a set $[n] = \{1, 2, \ldots, n\}$ with elements from a collection of m subsets of $[n]$. The first line corresponds to "n m" (indicating the set $[n]$ and the amount of subsets of $[n]$, respectively) and the following m lines correspond to the subsets of $[n]$, represented by integers belonging to $[n]$ in ascending order. As output, the code prints out the number of different solutions for the problem, followed by the respectively solutions, line by line and represented by numbers belonging to $[n]$ in ascending order.

3 *k*-SAT as a Cover Set Problem

The k-SAT problem can be interpreted as a generalized cover set problem. Adopting the convention "1 = true" and "0 = false", one can represent a k-SAT by a matrix with the rows containing the possible values for which the variables $\{x_1, x_2, \ldots, x_n\}$ can assume (0 or 1), and with the primary columns corresponding to the literals (columns that should be exactly covered, i.e., the subset of rows from a solution must contain exactly one "1" at each of those columns) and the secondary columns corresponding to the clauses $\{c_1, c_2, \ldots, c_m\}$ which should be covered at least once by the set of rows of a given solution.

For example, consider the input:

$$\begin{matrix} 3 & 4 & 5 \\ -1 & 2 & 3 \\ 1 & -2 & 4 \\ 1 & -3 & 4 \\ -1 & -2 & 3 \\ -1 & 2 & -3 \end{matrix}$$

which corresponds to the following k-CNF:

$$(\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_4) \wedge (x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3)$$

Therefore, this boolean formula can be represented by the matrix:

$$\begin{array}{r} \\ \mathrm{x}_1 = 0 \\ \mathrm{x}_1 = 1 \\ \mathrm{x}_2 = 0 \\ \mathrm{x}_2 = 1 \\ \mathrm{x}_3 = 0 \\ \mathrm{x}_3 = 1 \\ \mathrm{x}_4 = 0 \\ \mathrm{x}_4 = 1 \end{array} \begin{array}{c} \begin{array}{ccccccccc} \mathrm{x}_1 & \mathrm{x}_2 & \mathrm{x}_3 & \mathrm{x}_4 & \mathrm{c}_1 & \mathrm{c}_2 & \mathrm{c}_3 & \mathrm{c}_4 & \mathrm{c}_5 \end{array} \\ \left[\begin{array}{ccccccccc} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \end{array}\right] \end{array}$$

Thus, as the primary columns (first four) correspond to an exact cover set problem, once selected the row $\mathrm{x}_\mathrm{i} = 0$, the row $\mathrm{x}_\mathrm{i} = 1$ cannot be in the solution, and vice-versa. However, the same criteria cannot be used for the columns representing the clauses, since a clause is satisfied when one or more literals are true.

In the example above, one attribution that satisfies the k-CNF is x_1, x_2 and x_3 true and x_4 false, represented by the rows $x_1 = 1$, $x_2 = 1$, $x_3 = 1$ and $x_4 = 0$ (or, in a compact form, 1110).

4 The X Algorithm

The X algorithm from D. Knuth [1] is a backtracking search algorithm that can be applied to a class of combinatorial puzzle problems, that is to find all solutions of an exact cover problem. It's based on a technique called *dancing links*: given a pointer x to an element from a double linked list, and let $L[x]$ and $R[x]$ be the pointers to the precedent and posterior elements, respectively, then the removal of x from the list is given by the operations:

$$\begin{aligned} L[R[x]] &\longleftarrow L[x] \\ R[L[x]] &\longleftarrow R[x] \end{aligned}$$

However, since after removal the element pointed by x is still allocated in memory, its return to the list is given by the operations:

$$\begin{aligned} L[R[x]] &\longleftarrow x \\ R[L[x]] &\longleftarrow x \end{aligned}$$

what allow thus return back to the previous stage (*backtrack*).

Given a matrix A of "0s" and "1s", the X algorithm searches for all sets of rows containing exactly one "1" at each column. The matrix A is then represented only by the "1s", with circular double linked lists for the rows and columns.

Each "1" in the matrix is represented by an structure containing four pointers for the preceding and posterior elements in the row and column, in addition to a pointer to the head of the column which it belongs to:

```
typedef struct node_struct {
   struct node_struct *left, *right; /* predecessor and successor at row */
   struct node_struct *up, *down; /* predecessor and successor at column */
   struct col_struct *col; /* column containing this node */
} node;
```

The columns' lists possess a head and the heads themselves also form a circular double linked list with a head called "root". Each element from a column list has an identifier given by a *string*, a counter with the updated amount of elements in the column and pointers to the precedent and posterior elements in the list and to the root:

```
typedef struct col_struct {
   node head; /* list head (root) */
   int len; /* actual number of items in this column list (not including head) */
   char name[max_name]; /* symbolic identifier of the column for printing */
   struct col_struct *prev, *next; /* neighbor columns */
} column;
```

The X algorithm scans the matrix A searching recursively for solutions as follows:

If A is empty, print actual solution and return.
Otherwise, choose a column c and do:
  Cover column c.
  For $r \leftarrow D[c], D[D[c]], \ldots,$ while $r \neq c$, do:
    Include r in the partial solution: $partial \leftarrow r$
    For $j \leftarrow R[r], R[R[r]], \ldots,$ while $j \neq r$, do:
      Cover column from j (i.e. $C[j]$).
    Recursively call this algorithm for the reduced matrix A.
    Assign previous values: $r \leftarrow partial$ and $c \leftarrow c[r]$
    For $j \leftarrow L[r], L[L[r]], \ldots,$ while $j \neq r$, do:
      Uncover column $C[j]$.
Uncover column c.

where D, U, L and R are pointers to the elements above and below in the column, and to the left and to the right in the row, respectively, and $C[j]$ represents the column at which the element j belongs to. The operation of covering a column consists in removing the column head and all the rows it contains from top to bottom:

$L[R[c]] \leftarrow L[c]$
$R[L[c]] \leftarrow R[c]$
For $i \leftarrow D[c], D[D[c]], \ldots,$ while $i \neq c$, do:
  For $j \leftarrow R[i], R[R[i]], \ldots,$ while $j \neq i$, do:
    $U[D[j]] \leftarrow U[j]$
    $D[U[j]] \leftarrow D[j]$
    $S[C[j]] \leftarrow S[C[j]] - 1$

where $S[C[j]]$ is the counter of the amount of elements at column j. The operation of uncovering a column is done in a reverse way as the covering procedure, i.e., from bottom to top:

```
For i ← U[c], U[U[c]], ..., while i ≠ c, do:
   For j ← L[i], L[L[i]], ..., while j ≠ i, do:
      S[C[j]] ← S[C[j]] + 1
      U[D[j]] ← j
      D[U[j]] ← j
L[R[c]] ← c
R[L[c]] ← c
```

5 Adaptation to the *k*-SAT Problem

As discussed in `Section 3`, the k-SAT problem can be mapped to a generalized cover problem, where the primary columns correspond to literals and the secondary ones to clauses. That is, the primary columns representing the literals have to be exactly covered, while the columns corresponding to the clauses should be covered at least once, since a clause is satisfied when one or more literals are true.

In order to adequate the X algorithm to the k-SAT problem it's enough to modify the parts "`Cover column from` j" and "`Uncover column from` j". More specifically, these calls of the algorithms to "cover" and to "uncover" should be changed accordingly.

In the algorithm described in `Section 4`, besides removing the column's head, it removes also all its rows. This is fine for the primary columns, because once selected the row $x_i = 0$, then the row $x_i = 1$ cannot be in the solution, and vice-versa. However, when covering a secondary column (clause), its rows should not be removed as those rows could correspond to possible solutions that would discarded in case of removal. Therefore, for the secondary columns one should only remove the column's head.

Moreover, it's necessary to verify if the head of the secondary column was already removed. Since each clause has k literals, it's enough to check if the number of elements in the column is equal to k. Thus, the variation of the cover algorithm

for the secondary columns turns out to be:

```
If S[c] = k, do:
    L[R[c]] ← L[c]
    R[L[c]] ← R[c]
S[c] ← S[c] − 1
```

With a similar reasoning, a secondary column is uncovered only when the amount of elements is equal to $k - 1$ and, therefore, the uncover algorithm for secondary columns becomes:

```
If S[c] = k − 1, do:
    L[R[c]] ← c
    R[L[c]] ← c
S[c] ← S[c] + 1
```

The structure representing the k-SAT problem has to be created in such way that the columns' root is immediately before the primary columns and immediately after the secondary, and in the modified X algorithm one choses to cover always a primary column c next to the root. Therefore, the k-SAT problem can be seen as a scan over a search tree with depth k (`Figure 1`).

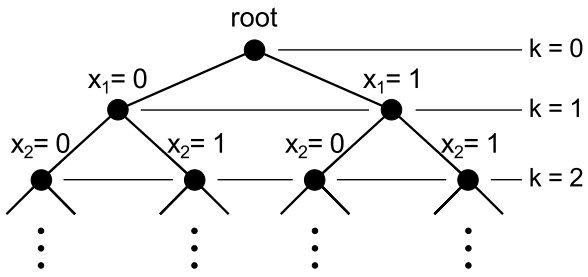

Figure 1: *k-SAT representation as a search tree.*

5.1 Observations

At "*Dancing Links*" [1] D. Knuth shows that choosing the column to be covered as the one with lesser elements, at each stage of the backtracking procedure, leads to a

search tree with lesser amount of nodes. This is an important observation specially for larger problems, because fewer updates (insertion and removal of elements) is realized. For this reason it's worth to include a counter S in the column structure shown above (where this counter is named as `len`). The way the k-SAT problem was constructed here, the counter S are only used as a criteria when covering the secondary columns, since the primary always have the same amount of elements (representing the values true and false of a literal). Another optimization mentioned by D. Knuth is to not cover columns with no elemnts, which again does not apply to the primary columns of the k-SAT problem.

In addition to the structures presented previously in `Section 4`, two other data structures were created to deal with the program output, which has to be given in terms of the boolean variables (for the example shown in `Setion 3`, the solution represented by the rows $x_1 = 1$, $x_2 = 1$, $x_3 = 1$ and $x_4 = 0$ should be output as 1110). To this end, the follow structure was used to point and identify the rows of the problem:

```
typedef struct line_struct {
   node *nodeR; /* first node of a row */
   char nameR[max_name]; /* row name */
} line;
```

and another structure was used to store the found solutions containing a array of the type *line* above with the solutions rows, an integer indicating the search level at which the solution was found (relevant only for the exact cover problem) and a pointer to the next solution:

```
typedef struct sol_struct {
   line *rowSol; /* array of solution rows */
   int level; /* amount of rows from solution */
   struct sol_struct *next; /* pointer to the next solution */
} solution;
```

6 Results and Critical Point Analysis

The probability that a k-CNF being satisfiable depends on the amount of clauses (m) and of literals (n). If the amount of clauses to be satisfied is large and the

mount of literals is small, then smaller is the chances of the k-CNF being satisfiable. On the other hand, if the number of literals is large and the amount of clauses is small, then higher are the chances of the k-CNF to be satisfiable. It is believed that for all k there exists a constant c_k with the following property:

> *Let ε a constant and suppose that $n \to \infty$. If $m \geq (c_k + \varepsilon)\, n$, then the k-CNF is almost-certainly not satisfiable, while if $m \leq (c_k - \varepsilon)\, n$, then it is almost-certainly satisfiable.*

To estimate the constant c_k for different values of k, the routine "*random_ksat.py*" from H. Yuen and J. Bebel has been used to generate pseudo-random instances of k-CNF[2], with a small modification to fulfill the input format described above. For a fixed value of k, the quantities $n \geq k$ and $m \geq 1$ were varied and, for each triple (k, n, m) a total of 50 pseudo-random instances of k-CNF were generated, from where the mean values of the amount of solutions and the execution time were calculated. Such procedure can be evaluated with the following example script (for larger n the size of the increment of m has to be increase accordingly):

```
#!/bin/bash

for k in ‘seq 2 5‘
do
  for n in ‘seq ${k} 20‘ # number of variables n: [k;20]
  do
    m=1
    aux=10 # just to get into the loop
    while [ ${aux} -gt 0 ] # while satisfiable, clauses m: [1;...]
    do
      count=0
      time=0
      for i in ‘seq 50‘ # 50 calculations, then take the mean
      do
        # Generate random input (k, n, m).
        python random_ksat.py ${k} ${n} ${m} -1 CriticalPointAnalysis/input

        ini=$(date +%s%N) # initial time

        # Execute the 'satisfyme' code.
        count=$(( ${count} + ‘./satifyme -n < CriticalPointAnalysis/input‘ ))

        fin=$(date +%s%N) # final time

        # Calculate the time in seconds.
        time=‘echo "scale = 10; ${time} + (${fin} - ${ini})/1000000000" | bc‘

      done

      # Calculate the mean values of the amount of solutions and the execution time.
      count=‘echo "scale = 10; ${count} / ${i}" | bc‘
      time=‘echo "scale = 10; ${time} / ${i}" | bc‘
      echo ${m} ${count} >> CriticalPointAnalysis/saida_k${k}_n${n}
      echo ${m} ${time} >> CriticalPointAnalysis/time_k${k}_n${n}

      aux=‘echo "${count} / 1" | bc‘
      m=$(( ${m} + 1 ))

    done
  done
done
```

The obtained estimates for the constant c_k are shown in the `Table 1` below:

k	2	3	4	5	8	10
c_k	2.44	5.18	10.76	21.87	177.00	709.79

Table 1: *Obtained estimates for the constant c_k.*

The graphs below show the mean values of the amount of solutions (log scale in both axis) and the execution time the plots (log sale in time), as a function of the number of clauses m and for different number of literals n. One observes that for $m \approx c_k n$, the execution time is fairly higher.

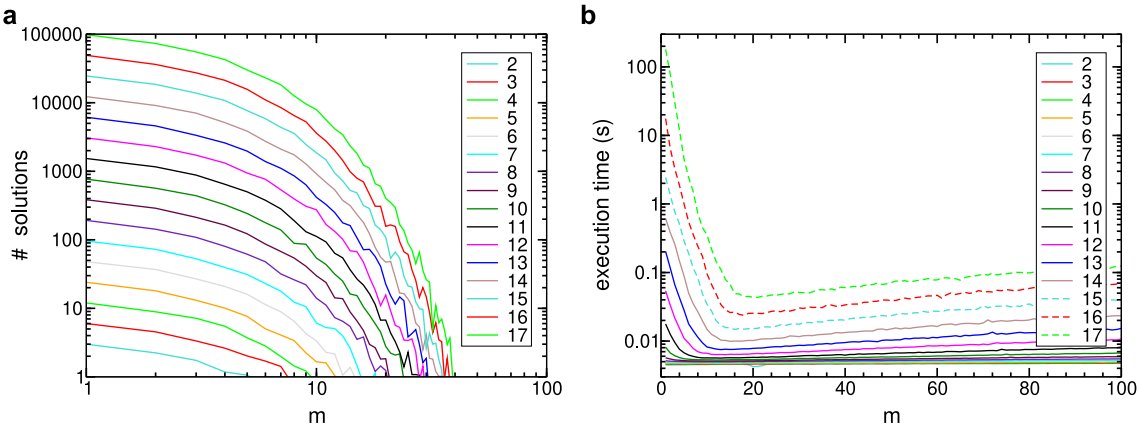

Figure 2: Mean values of the amount of solutions (**a**) and execution time in seconds (**b**) of a **2**-CNF as a function of m (number of clauses), taken for different n (number of literals shown at the legend).

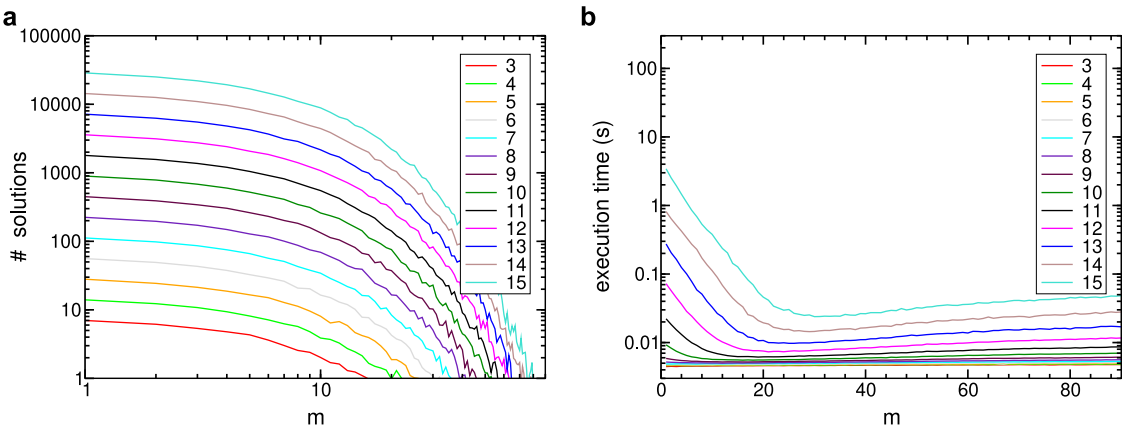

Figure 3: Mean values of the amount of solutions (**a**) and execution time in seconds (**b**) of a **3**-CNF as a function of m (number of clauses), taken for different n (number of literals shown at the legend).

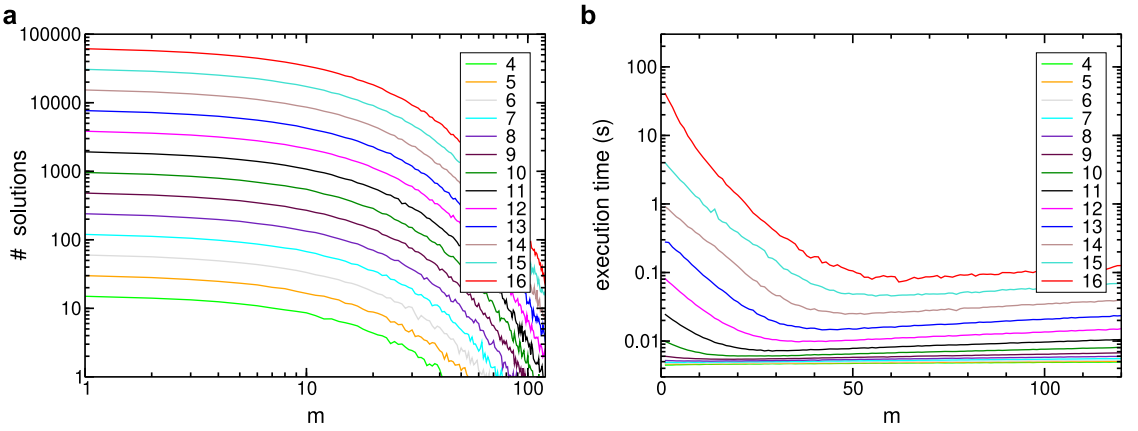

Figure 4: Mean values of the amount of solutions (**a**) and execution time in seconds (**b**) of a **4**-CNF as a function of m (number of clauses), taken for different n (number of literals shown at the legend).

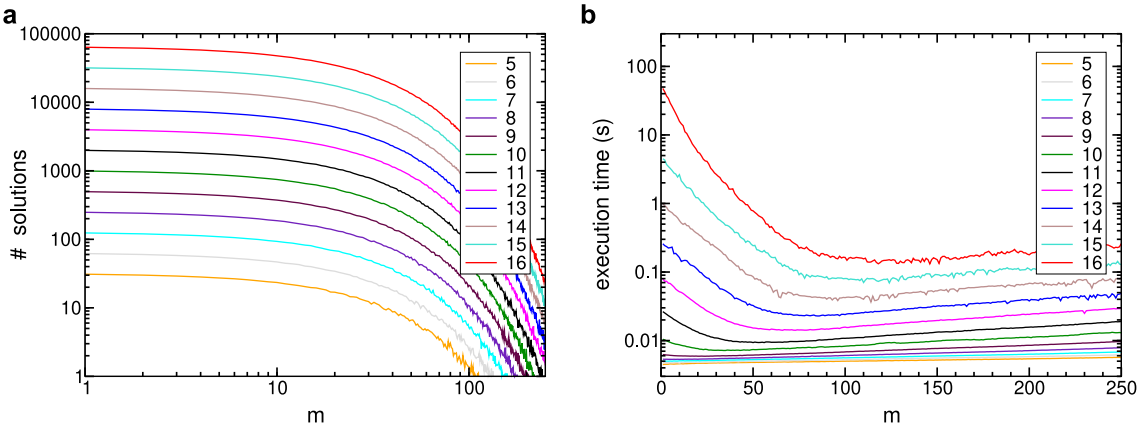

Figure 5: Mean values of the amount of solutions (**a**) and execution time in seconds (**b**) of a **5**-CNF as a function of m (number of clauses), taken for different n (number of literals shown at the legend).

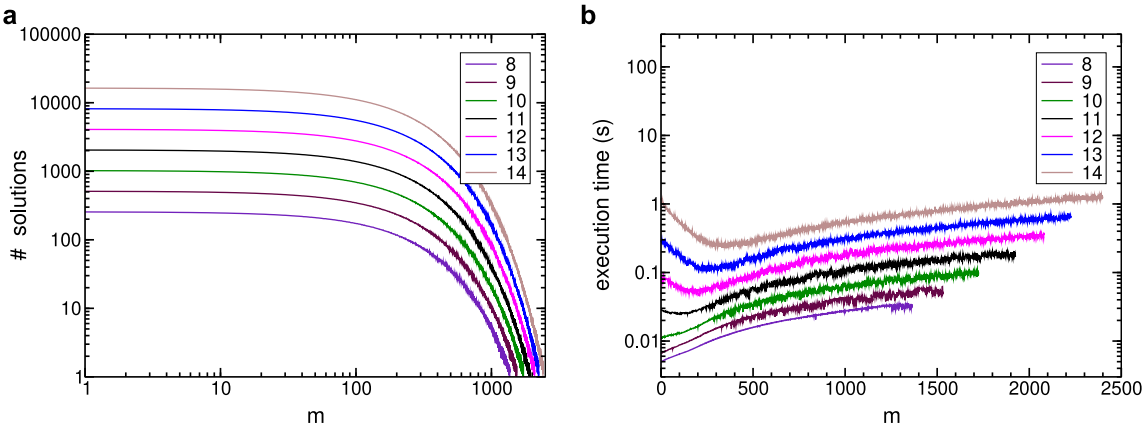

Figure 6: Mean values of the amount of solutions (**a**) and execution time in seconds (**b**) of a **8**-CNF as a function of m (number of clauses), taken for different n (number of literals shown at the legend).

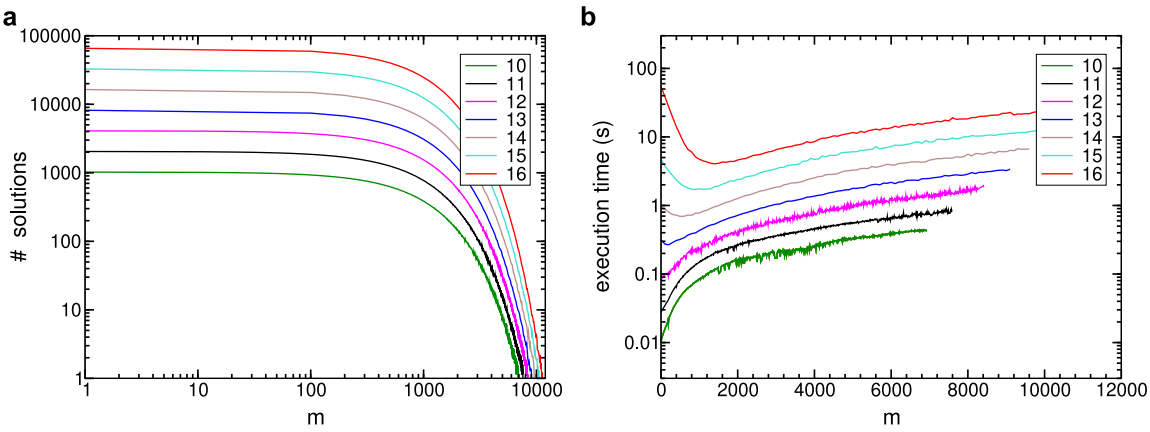

Figure 7: Mean values of the amount of solutions (**a**) and execution time in seconds (**b**) of a bf 10-CNF as a function of m (number of clauses), taken for different n (number of literals shown at the legend).

References

[1] D. Knuth, "*Dancing Links*," arXiv:cs/0011047 (2000), `https://arxiv.org/abs/cs/0011047`.

[2] H. Yuen and J. Bebel, "*ToughSat Project*," (2011), `http://toughsat.appspot.com`, routine `random_ksat.py` consulted in April 20th, 2012.