Naive and DPLL algorithms

I have implemented the naive and DPLL algorithms in three languages :

- OCamL : clearly the fastest
- Python : the slowest (not surprisingly)
- Cython : when it comes to performance, Cython is quite tricky

OCamL

I used the Set data structure : to quote the OCamL documentation :

> The implementation [of Sets] uses balanced binary trees, and is therefore reasonably efficient: insertion and membership take time logarithmic in the size of the set, for instance.

The CNF clauses are nothing else than a set of integer sets (the clauses), which is convenient, given that the order doesn't matter, by commutativity.

How to use it

```
cd src

./OCamL/DPLL ./Examples/test6.txt

# The hardest examples (far too much for Python) were placed in the "Difficult" folder

./OCamL ./Difficult/test4.txt
```

Python (3.5/3.6)

I tried to stick with the pythonic way to implement such a recursive algorithm as much as possible (although I did resort to functools assets such as `reduce` and `map` (which are anything but pythonic)), especially by making an extensive use of iterators.

How to use it

```
cd src

python ./Python/DPLL.py ./Examples/test6.txt
```

Cython

It would be far too long to gat into the details : if the situation had to be described in one word, I finally managed to compile to `.pyx` file in an executable with the following commands :

```
cython --embed -o DPLL_compiled.c DPLL.pyx

gcc -v -Os -I /Users/younessekaddar/.pyenv/versions/3.6.0/include/python3.6m -L /Library/Frameworks/Python.framewo
rk/Versions/3.6/lib  -o DPLL_compiled DPLL_compiled.c  -lpython3.6  -lpthread -lm -lutil -ldl
```

(which are obviously only suited for my particular case)

How to use it

```
cd src

./Cython/DPLL_compiled ./Examples/test6.txt
```

> Concerning the `choose` function (which chooses the next literal before a recursive call), I tried two different implementations :

- by randomly picking a literal amongst the available ones

- by taking the “next” one :
    - in the iterator, for Python/Cython
    - with the `choose` function of the Set module for OCamL

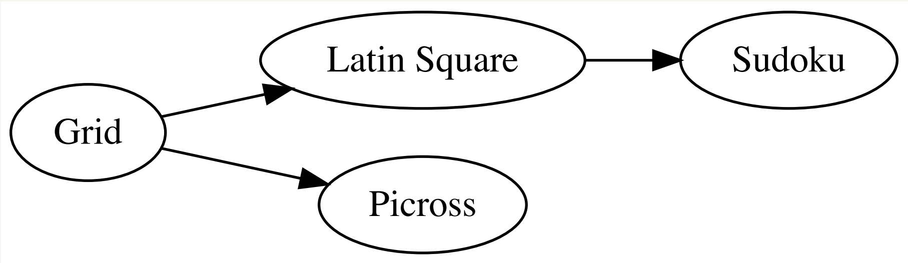
But the randomized approach was so bad (5 to 6 time slower) that I gave it up.

```
def choose(clauses, choice = 'default'):
    if choice == 'random':
        return choice(tuple(reduce(set.union, clauses, set())))
    else:
        return next(iter(clauses[0]))
```

Logic games

Logic games have been implemented as Python classes with regards to their outer structure, but the actual SAT solving process resorts to the OCamL DPLL algorithm, through the `subprocess` Python library.

The inheritance relation between the used classes is depicted in the following graph :



Here is a brief overview of theses classes.

Grid

It is the main class from which the different logic games inherit.

Once an instance of a given game has been translated into CNF clauses, *via* the `generate_file` method, the `show` method resorts to the OCamL DPLL algorithm so as to translate any satisfiable truth valuation back into a game grid.

The following classes share a similar structure, which is underlain by three key methods :

- `generate_file` : actually translates an instance of the problem into CNF clauses (the ouput format follows the DIMACS format conventions)
- `generate_grid` : given a truth valuation, creates the corresponding grid to be displayed with matplotlib
- `decode_literals` (*static method*) : given an integer literal (appearing in the CNF DIMACS format), translates it back into a tuple that is about to be properly interpreted in the matplotlib graph

Latin Squares

The literals are triplets (k, i, j), given that :

$$(k, i, j) \text{ is true } \iff \text{ the variable k is in position (i,j)}$$

Besides, (k, i, j) corresponds to the literal number $k \times (n^2) + i \times n + j + 1$

> **NB** : *It’s a little detail, but it bugged me bit at the beginning* :
>
> when it comes to encoding and decoding literals, as such operations are carried out multiple times, one might be tempted to use a dictionary (for which setting and getting items has a constant average time cost), but it turns out that the overhead time cost don’t turn out to be that advantageous :

```
# Latin Square
# With a dictionary
>>> timeit 100000 from latin_square import LatinSquare; LatinSquare(5, 4)
0.001320116535720008 seconds on average for 100000 iterations

# Without any dictionary, by computing the literal number each time (provided the value of n**2 is stored)

>>> timeit 100000 from latin_square import LatinSquare; LatinSquare(5, 4)
0.0012361194491600327 seconds on average for 100000 iterations.
```

```
# (k,i,j) is true iff the variable k is in position (i,j)
# (k,i,j) corresponds to the literal number k*(n**2) + i*n + j + 1

for k in range(n):
    for i in range(n):
        not_two_in_one_row = ''
        for j in range(n):
            output_str += str(k*n_squared + i*n + j + 1) + ' '
            for j2 in range(j):
                not_two_in_one_row += '-' + str(k*n_squared+i*n+j2+1) + ' -' + str(k*n_squared+i*n+j+1) + ' 0\n'
        output_str += '0\n'
        output_str += not_two_in_one_row

for k in range(n):
    for j in range(n):
        not_two_in_one_col = ''
        for i in range(n):
            output_str += str(k*n_squared+i*n+j+1) + ' '
            for i2 in range(i):
                not_two_in_one_col += '-' + str(k*n_squared+i2*n+j+1) + ' -' + str(k*n_squared+i*n+j+1) + ' 0\n'
        output_str += '0\n'
        output_str += not_two_in_one_col

for i in range(n):
    for j in range(n):
        for k in range(n):
            output_str += str(k*n_squared+i*n+j+1) + ' '
        output_str += '0\n'

self.outputs.append(output_str)
```

VS :

```
# (k,i,j) is true iff the variable k is in position (i,j)
# (k,i,j) corresponds to the literal number k*(n**2) + i*n + j + 1

d = {}
count = 1

for k in range(n):
    for i in range(n):
        not_two_in_one_row = ''
        for j in range(n):
            d[(k,i,j)] = count
            output_str += str(count) + ' '
            count +=1
            for j2 in range(j):
                not_two_in_one_row += '-' + str(d[(k,i,j2)]) + ' -' + str(count) + ' 0\n'
        output_str += '0\n'
        output_str += not_two_in_one_row

for k in range(n):
    for j in range(n):
        not_two_in_one_col = ''
        for i in range(n):
            output_str += str(d[(k,i,j)]) + ' '
            for i2 in range(i):
                not_two_in_one_col += '-' + str(d[(k,i2,j)]) + ' -' + str(d[(k,i,j)]) + ' 0\n'
        output_str += '0\n'
        output_str += not_two_in_one_col

for i in range(n):
    for j in range(n):
        for k in range(n):
            output_str += str(d[(k,i,j)]) + ' '
        output_str += '0\n'

self.outputs.append(output_str)
```

As a result, I didn't store the interpretations of the literals for the other game classes either.

Sudoku

Same literals as the LatinSquare, but with additional block-related constraints.

On top of the other methods, the `random` method generates a randomly partially filled Sudoku grid.

```
# Sudoku(size_of_grid, random=number_of_fixed_coefficients, solvable=necessarily_solvable_or_not)
M = Sudoku(9, random=37, solvable=False)
# M.show()

N = Sudoku(9, random=27, solvable=True)
N.show()
```

The random generation is twofold :

- if the `solvable` parameter is set to `True` :
  the randomly generated grid is guaranteed to be solvable, insofar as it is build out of a totally filled random Sudoku grid.
- if the `solvable` parameter is set to `False` :
  the randomly generated grid is indeed a partial Sudoku grid, but may not be solvable.

Picross

Let n (resp. m) be the number of rows (resp. columns).

The literals fall into two categories :

- the cell-related literals, in the form of couples (i, j)
- the block-related ones, in the form of triplets (k, i, j)

$$(i, j) \text{ is true } \iff \text{ the cell in position } (i, j) \text{ is colored}$$

and

$$(k, i, j) \text{ is true } \iff \text{the } i\text{-th block of the } k\text{-th sequence (that is, line or column) starts from the } j\text{-th position.}$$

The former are associated with the literals

$$1 \leq i \times \max(n, m) + j \leq \max(n, m)^2$$

the latter with the literals

$$\max(n, m)^2 + 1 \leq \max(n, m)^2 + k \times (m + n)^2 + i \times (m + n) + j + 1 \leq (m + n)^3 + \max(n, m)^2 + 1$$

Moreover :

- for each sequence (*i.e* row or column), the i-th block is somewhere
- for each sequence, the i-th block is at one position at most
- each cell of each block is colored
- each colored cell is in one block of its row
- each colored cell is in one block of its column

⟶ I also implemented a verbose mode to trace back the literals more conveniently in the DIMACS generated file.

⟶ An instance can also be initialized by providing the URL of a “picross” file in the **CWD format**.

The CWD format

```
number_of_rows
number_of_columns
rows

columns
```

e.g :

```
10
5
2
2 1
1 1
3
1 1
1 1
2
1 1
1 2
2

2 1
2 1 3
7
1 3
2 1
```

Benchmarks

benchmark.py

This file aims at comparing the performances of the OCamL, Python and Cython DPLL algorithms on the files located in the `./Example` folder.

benchmarks.sh

This bash script creates a file, the content of which is the following markdown table, comparing the performances of the OCamL, Python and Cython DPLL algorithms on the files located in the `./Example` folder.

```bash
#!/bin/bash
files=./Examples/*
output=time.txt
echo "||OCamL|Cython|Python">$output
echo "-|-|-|-">>$output
for f in $files
do
  echo "Processing $f file..."
  ocaml="$( TIMEFORMAT='%lU';time ( ./OCamL/DPLL $f ) 2>&1 1>/dev/null )"
  cython="$( TIMEFORMAT='%lU';time ( ./Cython/DPLL_compiled $f ) 2>&1 1>/dev/null )"
  python="$( TIMEFORMAT='%lU';time ( python ./Python/DPLL.py $f ) 2>&1 1>/dev/null )"
  echo "${f#$files}|$ocaml|$cython|$python">>$output
done
```

How to use it

```bash
cd src

chmod +x ./benchmarks.sh
./benchmarks.sh
```

	OCamL	Cython	Python
latin_square_3.txt	0m0.006s	0m0.093s	0m0.155s
latin_square_4.txt	0m0.019s	0m0.126s	0m0.215s
latin_square_9.txt	0m0.916s	0m6.290s	0m7.101s
picross_10_10_1B0A7J8GFE0I947QOP210BVVZGHGSK.txt	0m0.067s	0m0.152s	0m0.279s
picross_10_20_YSCI430K89C3PPAY5MY0Z4XBYFG5OOFO95M0D6X50CZKPET9VH.txt	0m0.367s	0m1.038s	0m1.175s
picross_10_5_2S0MZ3SE45E68E661.txt	0m0.173s	0m0.217s	0m0.307s
picross_15_15_3BWYQ1KSBSMP2YHH4LWN87XBA6CXMHTNO3IYP.txt	0m0.327s	0m1.009s	0m1.286s
picross_5_5_EGJL3MLF.txt	0m0.007s	0m0.050s	0m0.140s
picross_8_8_JRF431YX10EVG1R.txt	0m0.120s	0m0.160s	0m0.256s
picross_9_5_9JT4G0T57YGWP22H.txt	0m0.113s	0m0.108s	0m0.226s
picross_9_8_EJ8UN1C2M2O4TG5R.txt	0m0.148s	0m0.299s	0m0.493s
sudoku_4.txt	0m0.016s	0m0.126s	0m0.263s
sudoku_4_1ZYLI8KVPC.txt	0m0.013s	0m0.075s	0m0.197s
sudoku_4_TXL8PXIXU1.txt	0m0.006s	0m0.055s	0m0.140s
sudoku_6.txt	0m0.013s	0m0.068s	0m0.153s
sudoku_9_1GX4FPP2MXTZ34Z5G06EWWQV7IK2BZVALK6TY0FISVNGVC5051C.txt	0m0.063s	0m0.260s	0m0.401s
sudoku_9_3VQXKC2ZBJLXQ48PJHH3LAMW08H6DZ8MUTGE6RDZ1JHUPFIA02YU.txt	0m0.968s	0m5.804s	0m6.245s
sudoku_9_42Y4WY8HTNMCE6XGGQS30JYQT2W6XF2ZS5Y81PMBQNGH2CUA1AA8.txt	0m0.107s	0m0.382s	0m0.542s
sudoku_9_492NPRN1HIOX0LM1WLI7140DWV76XFPXB6B3DZTR840HFR2D91JC.txt	0m0.199s	0m0.469s	0m0.375s
sudoku_9_662DTRFIQ3L1OSCWZH75GUQKFNZ3U7DY17KRZ73AW1BM7GSB27O.txt	0m12.665s	0m45.010s	0m48.086s
sudoku_9_CM2TXPZ68TCLEUQZV8PFGY3YTB4DC99N4JA3IKR0OJWKI49649Y.txt	0m0.077s	0m0.298s	0m0.366s
sudoku_9_D1HB6ZSYIO1PR55T1G9Z5904BZI1WWH91OX0MESBO5WD6FKP99DW.txt	0m0.069s	0m0.280s	0m0.367s