facile Documentation

Release 1.3

Xavier Olive

May 20, 2018

Contents

Python constraint programming library.

Github | PyPI | FaCiLe

Facile library is an excellent tool for solving constraint programming problems in OCaml. Facile stands for “Functional Constraint Library”. Besides, “facile” means “easy” in French!

This package is a Cython-based wrapping in Python of basic functionalities of this library.

CHAPTER 1

Installation

Facile is provided for Python 2.7, 3.4 and 3.5, for Linux, MacOS (10.10) and Windows (no 2.7).

```
pip install facile
```

The Linux version cannot be provided as wheel file: it has been compiled under a Debian Stable (Wheezy) environment for an older version of glibc.

```
easy_install facile
```

If the automatic installation process does not work on your environment, considering compiling your own version.

1.1 Compilation

In order to compile the tool, I recommend the following method:

```
# Install opam, then have PIC-compiled version of OCaml:
opam switch 4.01.0+PIC # other versions may work as well

# Follow opam instructions to configure the environment

# Install ocamlfind and facile library
opam install ocamlfind
opam install facile

# Build/install the Python library
python setup.py install

#  (optional) Prepare a package
python setup.py bdist_wheel
```

I detailed my approach and some lessons learned here

For building Windows packages, I use cross-compilation from Linux or Mac (see the specifically designed cross command of setup.py)

Do not hesitate to share if you find a better compilation experience.

CHAPTER 2

Quickstart

2.1 Introduction

A guy walks into a 7-11 store and selects four items to buy. The clerk at the counter informs the gentleman that the total cost of the four items is $7.11. He was completely surprised that the cost was the same as the name of the store. The clerk informed the man that he simply multiplied the cost of each item and arrived at the total. The customer calmly informed the clerk that the items should be added and not multiplied. The clerk then added the items together and informed the customer that the total was still exactly $7.11.

What are the exact costs of each item?

2.1.1 Resolution

```
from facile import *

a = variable(0, 330)
b = variable(0, 160)
c = variable(0, 140)
d = variable(0, 140)

constraint(a + b + c + d == 711)
constraint(a * b * c * d == 711000000)

p = array([a, b, c, d])
assert solve(p)
print ("Solution found a=%d, b=%d, c=%d, d=%d" % tuple(p.value()))

# Solution found a=316, b=150, c=120, d=125
```

So we found the costs of each item, resp. $3.16, $1.50, $1.20 and $1.25.

2.1.2 So, what happened?

The problem has been modelled using costs in cents so as to manipulate only integers. Four variables were chosen, taking values over a finite interal: the upper bounds are $0, and we had to adjust the upper bounds so as to not cause integer overflows through the multiplication.

Next come the constraints. The sum and the product are expressions of variables; we can state relations between expressions (operators like ==, <=, etc.). A constraint can then be manipulated as a Python reference, or posted to the solver (keyword: constraint)

The solve function takes the variables to be assigned as parameters and returns True if a solution is found. The .value() method applied to each variable yields the appropriate values.

2.2 Specific constraints

2.2.1 All different

Constraint programming uses global constraints that optimize the resolution process. In particular, we like to be able to state that the values assigned to a set of variables are all different.

See the following illustration through the 8-queen problem:

```
from facile import *

queens = [variable(0, 7) for i in range(8)]

constraint(alldifferent(queens))
constraint(alldifferent([q - i for q, i in enumerate(queens)]))
constraint(alldifferent([q + i for q, i in enumerate(queens)]))

assert solve(queens)
```

We can then script a bit to pretty-print the solution:

```
def print_line(val, n):
    cumul = n * '- '
    print (cumul[:2*val] + 'o' + cumul[2*val+1:-1])

for q in queens:
    print_line(q.value(), 8)
```

```
o - - - - - - -
- - - - o - - -
- - - - - - - o
- - - - - o - -
- - o - - - - -
- - - - - - o -
- o - - - - - -
- - - o - - - -
```

2.2.2 Arrays

Facile offers an array structure which helps stating meaningful expressions, such as:

- min() (resp. max()) yields a variable bound to the minimum (resp. maximum) value of a set of variables;

- the bracket notation [] accepts expressions made of variables to index an array;
- sort() returns an array of variables bound to the variables of the original array but sorted according to their values after resolution.

We can illustrate the usage of arrays with the *stable marriage problem*: all men and all women rank their possible partners. The goal is to match them so that for each pair, it is not possible to prefer each other over their current partners.

In the following modelisation, we need the arrays of variables husband and wife by other variables. Similarly, we need to index the ranks of integers by variables (and therefore transform lines of the 2D array into Facile arrays).

```
from facile import *

n = 5
men = ["Richard", "James", "John", "Hugh", "Greg"]
women = ["Helen", "Tracy", "Linda", "Sally", "Wanda"]

rank_women = [[1, 2, 4, 3, 5],
              [3, 5, 1, 2, 4],
              [5, 4, 2, 1, 3],
              [1, 3, 5, 4, 2],
              [4, 2, 3, 5, 1]]

rank_men = [[5, 1, 2, 4, 3],
            [4, 1, 3, 2, 5],
            [5, 3, 2, 4, 1],
            [1, 5, 4, 3, 2],
            [4, 3, 2, 1, 5]]

wife = array([variable(0, n-1) for i in range(n)])
husband = array([variable(0, n-1) for i in range(n)])

# You are your wife's husband, and conversely
for m in range(n):
    constraint(husband[wife[m]] == m)
for w in range(n):
    constraint(wife[husband[w]] == w)

for m in range(n):
    for w in range(n):
        # m prefers this woman to his wife
        c1 = rank_men[m][w] < array(rank_men[m])[wife[m]]
        # w prefers her husband to this man
        c2 = array(rank_women[w])[husband[w]] < rank_women[w][m]
        # trick: alias for c1 => (implies) c2
        constraint( c1 <= c2 )
        # w prefers this man to her husband
        c3 = rank_women[w][m] < array(rank_women[w])[husband[w]]
        # m prefers his wife to this woman
        c4 = array(rank_men[m])[wife[m]] < rank_men[m][w]
        # trick: alias for c3 => (implies) c4
        constraint( c3 <= c4 )

if solve(list(wife) + list(husband)):
    for i in range(n):
        print ("%s <=> %s" % (men[i], women[wife[i].value()]))

# Richard <=> Helen
```

(continues on next page)

(continued from previous page)

```
# James <=> Tracy
# John <=> Linda
# Hugh <=> Sally
# Greg <=> Wanda
```

2.3 Satisfaction & optimisation

In addition to the solve() function, Facile offers two other ways to solve a constraint satisfaction (resp. optimisation) problem: solve_all() and minimize().

Note that both methods return a list of values assigned to variables, in the same order as passed in parameter. The .value() method applied on values will return None.

- solve_all() returns a (possibly empty) list of assignations of variables.
- minimize() returns the value of the criterion and the assignations of variables that minimized it.

The following problem illustrates how to use the minimize() function.

2.3.1 The Golomb ruler

> A Golomb ruler is a set of integers (marks) $a_1 < \cdots a_k$ such that all the differences $a_i - a_j$ (assuming $i > j$) are distinct. Clearly we may assume $a_1 = 0$. Then a_k is the length of the Golomb ruler. For a given number of marks, we want to find the shortest Golomb rulers. Such rulers are called optimal.

As we build the ticks variables, we must set an upper bound. 2^n is a good candidate: considering the binary representation of $a_i - a_j = 2^i - 2^j = 2^j \cdot (2^{i-j} - 1)$, we may assert they are all differents.

The problem will be to find a shorter Golomb ruler.

In the following modelisation, we build a list of all differences (expressions) on which we pose an alldifferent constraint. Then, instead of using the regular solve function, we are to minimize the highest tick, i.e. ticks[n-1].

```
from facile import *

def golomb(n):
    ticks = [variable(0, 2**n) for i in range(n)]

    # First tick at the start of the ruler
    constraint(ticks[0] == 0)

    # Ticks are ordered
    for i in range(n-1):
        constraint(ticks[i] < ticks[i+1])

    # All distances
    distances = []
    for i in range(n-1):
        for j in range(i + 1, n):
            distances.append(ticks[j] - ticks[i])

    constraint(alldifferent(distances))

    # Redundant constraint
    for d in distances: constraint(d > 0)
```

(continues on next page)

(continued from previous page)

```
    # Breaking the symmetry
    size = len(distances)
    constraint(distances[size - 1] > distances[0])

    return (minimize(ticks, ticks[n-1])[1])

if __name__ == "__main__":
    import sys
    n = int(sys.argv[1])
    print (golomb(n))
```

So after resolution:

```
python golomb.py 10
[ 0  1  6 10 23 26 34 41 53 55]
```

2.4 Constraint reification

Some constraints can be transformed into new variables taking their values in {0, 1}. (resp. False and True) This process is called *reification*. Not all constraints are reifiable; an appropriate exception is raised if need be.

> Say we want to build an array of n variables $a_0, \cdots a_n$, where a_i represents the total number of occurrences of i in the array.

In the following example, the constraint x == i is automatically reified as it is summed with other constraints.

```
from facile import *

array = [variable(0, 10) for i in range(10)]

for i, a in enumerate(array):
    constraint(sum([x == i for x in array]) == a)

assert solve(array)
print ([v.value() for v in array])

# [6, 2, 1, 0, 0, 0, 1, 0, 0, 0]
```

CHAPTER 3

Examples

In order of complexity, you can read the following example programs, (almost) all coming from the set of examples from the original Facile library:

- examples/basic.py, basic example problems;
- examples/seven_eleven.py, find four numbers such that their sum is 711 and their product is 711000000;
- examples/coins.py, which coins do you need to give back change for any amount between 0 and maxval, using a given set of coins;
- examples/golomb.py, the Golomb ruler;
- examples/queens.py, the n-queen problem;
- examples/magical.py, a magical sequence where a[i] is equal to the number of i in a;
- examples/marriage.py, a problem of stable marriages;
- examples/buckets.py, a problem of filling buckets;
- examples/tiles.py, placing tiles on a board;
- examples/golf.py, the golf tournament problem.