Problem's solution by Exhaustive Search algorithm

Pietro Zafferani

BSc in Genomics, University of Bologna

March 31, 2021

1 Introduction

The problem given asked to take on the shoes of a pharmaceutical company that has to decide a purchase plan, the task consisted in selecting the specific suppliers that allowed the company to maximize the amount of chemical substance to be purchased. The rules to follow in the creation of the purchase plan were the following:

- given n suppliers, each supplier is incompatible with exactly $\frac{n}{2}$ other suppliers selected randomly.
- The amount of chemical substances that each supplier can offer is randomly drawn in a range from 0 to 1.

In order to solve this optimization problem it is necessary to reduce its elements to the minimum and to abstract its features.

2 Combinatorial Problem

Abstracting the problem at the combinatorial level is mandatory if we want to define a clear strategy to tackle it:

INPUT: an array $s = (s_1, \ldots, s_n)$ of n suppliers , each supplier is associated with a list of $\frac{n}{2}$ incompatible suppliers and with a list of $\frac{n}{2} - 1$ compatible suppliers.

OUTPUT: an array L of compatible suppliers that returns the highest amounts of possible compounds. Because of the restrictions imposed by the problem, L can have a maximum length of $\frac{n}{2} - 1$ elements.

3 Designing the algorithm

The strategy chosen was to define a class called *Supplier*, each instance of this class represents a supplier with associated a label, the weight of compounds offered, a so called 'black list' and a 'white list', with all the respective 'getters' methods. These two last attributes will be fundamental for the implementation of the algorithm.

```
    class Supplier(object):
        def __init__(self, label: int, weight: float, blacklist:
    list, interactions: list):

            self.__label = label
            self.__weight = weight
            self.__blacklist = blacklist
            self.__interactions = interactions
```

The external function that creates the array s returns a dictionary, this choice of design is due to the fact that retrieving information from a dictionary takes constant time, so especially for large numbers of suppliers this feature speeds the process up. The algorithm takes as only parameter the suppliers' dictionary, and it iterates over each instance of it.

The main issue of an Exhaustive Search algorithm is that it is very slow since by definition it looks in the whole search space before returning always the best solution possible, so we need to implement in the algorithm a smart way to look inside the search space.

This particular implementation takes advantage of the constrains imposed by the very same problem, namely that every supplier can be contained in the same array with at max $(\frac{n}{2} - 1)$ other suppliers which are the compatible ones. So for every supplier we need to check the compatibility only with those suppliers, then a double-sense check is performed to see if the compatibility is reciprocal. Every time a total amount of compounds surpasses the previous highest one, the latter is set as the new best one and also the array of suppliers that generated it is placed as the best one.

Eventually after the algorithm repeated this process n times, a tuple containing the highest amount of compounds and its respective array is returned.

```
#in this example we consider 10 suppliers

        if __name__ == '__main__':
# external function that creates the suppliers' dictionary
        suppliers=create_suppliers(10)
# running the algorithm
        print(ExhaustiveSearch(suppliers))

#the tuple contains the best solution possible for this example
>> (1.892, [2, 10])
```

4 Testing the algorithm

In order to test the practical efficiency of the algorithm it was exploited an external library called *codetiming*, it allowed to record the running time of the algorithm, then thanks to a function that repeated its execution exactly 1000 times it was possible to return the average running time. By giving to the function different numbers of suppliers we obtain a set of running times that is needed for the plotting of the algorithm's behaviour.

The theoretical complexity of the algorithm is polynomial, as can be extrapolated in section 3 since the amount of total operations depends on the size of the input and of the size of the white list which asymptotically tend to $n * n$. The others operations can be neglected since do not strictly depend on the input's size.

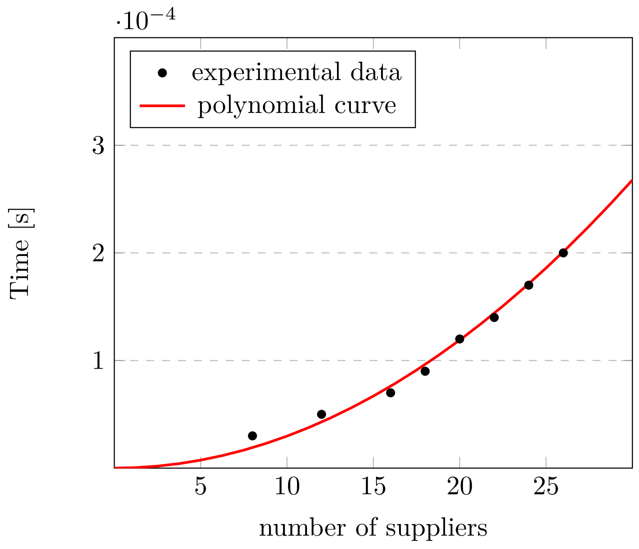
The *scipy* library allowed instead to predict a mathematical model of the algorithm's behaviour based on the data returned by the testing functions.

```
from codetiming import Timer
import scipy.optimize as optimization

#represents the mathematical model
def poly(x, a):
    return a * x ** 2  # complexity is polynomial

# each element is the numbers of suppliers considered
Set = [8, 12, 16, 18, 20, 22, 24, 26]
# average running times produced by testing the Set list
times = DataSets(Set)
printPairs(Set, times)
print()
#returns the values to use for plotting the mathematical model
print(optimization.curve_fit(poly, Set, times))

>> (8, 3e-05) (12, 5e-05) (16, 7e-05) (18, 9e-05) (20, 0.00012)
    (22, 0.00014) (24, 0.00017) (26, 0.0002)

#the first value is the coefficient of the polynomial function
>>(array([2.97475065e-07]), array([[2.09292323e-17]]))
```

5 Plotting the results

From the previous calculations we obtain a set of pairs that can be displayed in the following table:

Time [s]	3e-05	5e-05	7e-05	9e-05	1.2e-04	1.4e-04	1.7e-04	2e-04
Suppliers	8	12	16	18	20	22	24	26



When plotting the data on a Cartesian plane it is noticeable that the experimental data fit the theoretical model previously predicted.

So it is safe to say that this particular implementation of the Exhaustive Search algorithm runs in polynomial time.