Algorithmic complexity and graphs : 2024 - 2025 project

NICOLAS LE HIR
nicolas.lehir@epitech.eu

TABLE DES MATIÈRES

INTRODUCTION

Important : please invite me (using my epitech email address) to your repo when you are finished. Please send me an email with the url of your repo. Preferred method : epitech repo (instead of personal repo).

Please do not send me a fork of the course repo, with some added files. Instead, make a repo containing only your project files.

All processing should be made with python3.

Some form of report must accompany your code, in order to explain and comment it. General explanations and **conclusions** on your global approach are expected, instead of low level explanations on elementary functions.

— Example of explanations to include in your report : high-level discussions over the choice of the methods, heuristics, and conclusion on which method(s) worked best.

— Example of explanations **not** to include in your report : presentation of elementary python function, or library functions from numpy, matplotlib, or the libraries themselves. You should **not** present these.

Please write some sumary of your conclusions at the **beginning** of the report of each exercise (this actually facilitates reading a lot). For instance :

— (For exercice 1) : "we managed to obtain a solution that yields an average waiting time approximately equal to 85% of the waiting time obtained with the greedy solution"

— (For exercice 2) : "we tried 2 matching algorithms : on average, algorithm A was 13% better than algorithm B in terms of size the matching obtained, but is was also 22% slower."

You may use python scripts or notebooks, **except for the question 3 of exercise 1 where you have to use a python script, and for exercise 4** (see more details there), you may also write a pdf report. Note that you can write markdown inside a notebook. Short docstring at the top of files and functions will be appreciated, if relevant. There is no length constraint on the report, you do not need to write more than necessary. The goal of writing a report is to help me give you useful feedback.

The 4 parts of the project are independent.

1 PART 1 : THE SNOWPLOW PROBLEM

1.1 Introduction

A snowplow must clean the snow in front of n houses, $n \in \mathbb{N}$. The position of each house is represented by a float number (that might be < 0!), that represents meters, hence we assume that all houses are on a given one dimensional road. The snowplow travels at a speed of 1 meter per second. Initially, the snowplow is in 0.

Each house waits for a duration before the snow is cleaned in front of its doorstep. The snowplow must minimize the **average waiting time of the houses**. **This minimization is from the point of view of the houses** (and not from the point of view of the snowplow). The total distance traveled by the snowplow is not relevant itself, and it is important to note that this is not the same problem as the minimization of the distance traveled by the snowplow. **Please do not make this mistake**.

1.2 Exercise

1) In order to make sure that you understand the problem correctly, complete the following steps :

1a) Show the following average waiting times for these orders of cleaning of the houses :

— order : [-1.5, 1.2, 2.3, 5.0] average waiting time : 4.75s

— order : [5, -1.5, 2.3, 1.2] average waiting time : 12.05s

— order : [-15, 7, -6, -13] average waiting time : 39.75s

— order : [-13, -15, -6, 7] average waiting time : 22.25s

1b) Compute the average waiting times for these orders of cleaning of the houses :

— order : [-3, 9.5, -6.2, 2]

— order : [9.5, -3, 2, -6.2]

2) Propose a $n \in \mathbb{N}$ (it does not need to be large $n \in [5, 10]$ is fine) and a configuration of the houses where the optimal order of cleaning is **not** obtained by either :

— sorting the houses positions and cleaning them in that order.

— going to the closest house at each time step.

Like in question 1, **develop the exact computation** and compare the waiting times between the methods in order to prove it! You can also write a function that computes the average waiting time and use the outputs of this function.

3) Propose and implement a **polynomial-time algorithm** (in terms of the number of houses) based on the positions of the houses, in order to clean the snow. You must write a function that takes has input the positions of the n houses, and outputs the positions of the houses, sorted in the cleaning order, respecting the conventions listed below :

Conventions :

— **Name of the function (mandatory) : parcours()**.

— **Input of the function :** python list generated with $\mathrm{rng.normal}(\mathrm{loc} = 0, \mathrm{scale} = 1000, \mathrm{size} = 1000).\mathrm{tolist}()$ with **rng=numpy.random.default_rng()**. Hence we use $n = 1000$ in this exercise.

— **Output of the function :** python list of the houses positions in the order found by your algorithm.

— **Format :** for this question, you have to give me a python script that contains your function, and **not a notebook** (otherwise, I have to copy paste some code, which can lead to mistakes and loss of time). Please try to send me a **single module (=file)** that does not import other modules of yours (otherwise I have to copy several files, which can lead to mistakes and loss of time). Please also name your module, for instance with a short name representing your group.

Tests (very important) : You can find a test file named **test_parcours_exercise_1.py** in the **projet/** folder of the repo. This test file asserts whether :

— **parcours()** returns a python list

— the cleaning order has as many elements as the input list

— the sum of the positions of the houses in the cleaning order is identical to the sum before being sorted in that order (up to the numerical precision).

Please to not send me any solution that does not pass these tests !

Objective :

Your objective is that your solution yields a waiting time that is smaller to 90% of the time that we would obtain with a greedy solution, that consists in going to the closest house at each step. You are encouraged to try to obtain even lower values (it is possible to go below 70% of the greedy solution !). This is tested statistically, by generating a sequence of random input house positions and averaging the results, similarly to what was done during the course. An example workflow is to prototype you algorithm with small n and then check the behavior with larger n.

Remarks :

— Your main metric is the waiting time of the house, and not the execution time of the algorithm, although the execution time is also interesting and you are encouraged to produce algorithms with the smallest execution time possible (but this is not the main component of the notation).

— You can test your solution yourself in order to make sure that it meets the requirements, by also implementing the greedy solution, and comparing the cleaning time to that returned by your solution. **You should know the final performance of your algorithm**.

— You are encouraged to comment on the choices of datastructures used, and to comment on their impact on the program complexity.

4) In your report, prove that your solution runs in polynomial time.

The form of the proof might be similar to this example :

https://github.com/nlehir/AlgoGraph/blob/master/slides/11%20complexity%20proof.pdf

You can choose to compute the worst-case complexity or the average-case complexity : however, please be **explicit** about this choice.

2 PART 2 : CHOOSING BINOMS IN A COMPANY (MATCHING)

2.1 Introduction

A manager must choose binoms among a set of employees in a company. However, while some employees may work together, others may not. This information is represented in a graph G. Each employee is represented by a node in G. If two employees are represented by the nodes A and B, there is an edge between these two nodes if and only if they may work together. Choosing binoms corresponds to perform a matching in the graph G.

The manager would like to form a number of binoms that is as large as possible. Hence, The goal of this exercise if to perform a matching of large size. This does not necessary mean that you have to find an optimal matching in the strict sense of the term, but the larger the size of the matching, the better it is.

2.2 Random graphs

In this exercise, you will work with random graphs, in order to statistically evaluate the performance of different matching methods. The graphs can for instance be Erdös-Renyi graphs, also called G_{np} graph, with 120 nodes, and a probability $p = 0.04$ for the presence of each edge in the graph. In order to generate this kind of graphs, you can use the method **gnp_random_graph()** from **networkx**, with $n = 120$ and $p = 0.04$.

However, in order to compare and discuss the difference between heuristics (see the exercise below), you may also use different types of random graphs, on top of the G_{np} graphs mentioned above.

https://networkx.org/documentation/stable/reference/generated/networkx.generators.random_graphs.gnp_random_graph.html

https://en.wikipedia.org/wiki/Erd%C5%91s%E2%80%93R%C3%A9nyi_model

Figure 1 presents an example of such graph.

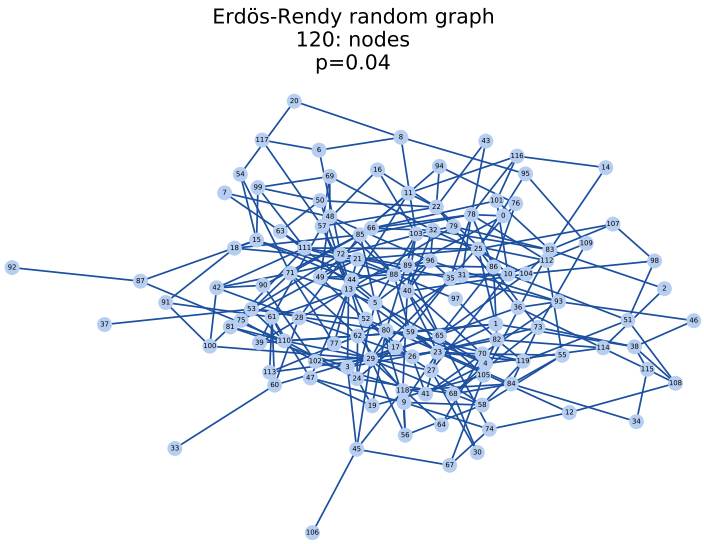

FIGURE 1 – Example 1 of a G_{np} graph.

To visualize the graphs, you may use and adapt the code from the repo.

2.3 Exercise

1) Pick **two** different degree-based heuristics of your choice and perform matchings on your generated graphs. Both heuristics must run in polynomial time. You can take inspiration from the document in the **documents/** folder in the repo. Please indicate your reference(s) if you use some.

2) Verify that your heuristics indeed return matchings by adding a test file **test_matching.py**. This file may use functions from the **networkx** library (or another library).

3) Compare the two heuristics statistically, in terms of :
— the size of the returned matching
— the measured computation time

Figure 2 presents the statistical result of the matching size obtained by applying the builtin method **maximal _matching.py** from **networkx**.

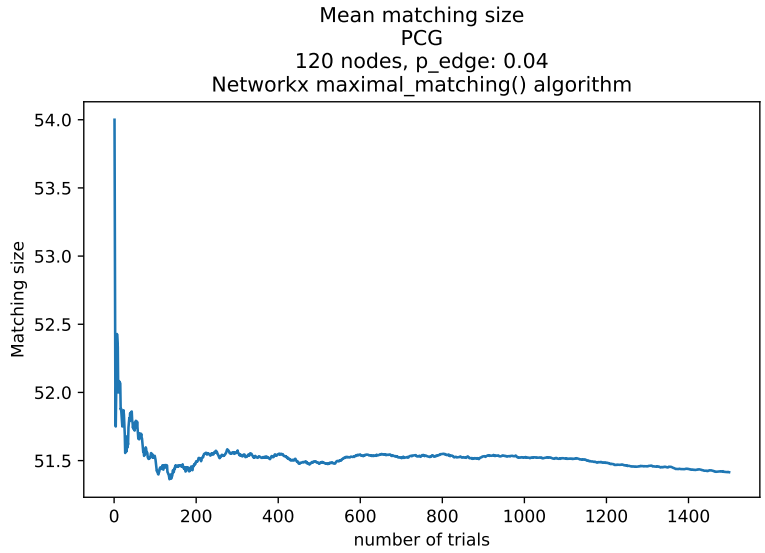

FIGURE 2 – Convergence of the size of the matching returned by the buitin method from networkx.

You should do enough trials to observe a convergence, like in figure 2. It is not mandatory that your heuristics perform better than this method !

4) Prove that both heuristics run in polynomial time (same instructions as in Exercise 1 for this proof).
Like in 1, you can choose to compute the worst-case complexity or the average-case complexity : however, please be **explicit** about this choice.

3 PART 3 : COMPLEXITIES

For each of the two functions **function_1()** and **function_2()** that you can find in **exercise_3/exercise_3_functions.py** :

— compute an upper bound of their complexity, as a function of their argument n. We look for the most informative bound, in the sense that it is for instance more informative to have a $\mathcal{O}(n^3)$ bound than a $\mathcal{O}(n^5)$ bound. For the format of the proof, the instructions are again similar to Exercise 1, question 4, and an example of proof for another function is shown here :
https://github.com/nlehir/AlgoGraph/blob/master/slides/11%20complexity%20proof.pdf

— verify the bound is correct by fitting a polynomial of the found degree to a curve representing the measured elapsed computation time as a function of n. For instance in figure 3, the function as a $\mathcal{O}(n^3)$ complexity, so a fitted polynomial of degree 2 can not represent well the curve. However, a polynomial of degree 3 can, as in figure 4.
You can use for instance **numpy.polyfit**. You have an example of usage of polyfit in the folder of this exercise in the repo, named **polyfit_example.py**.
https://numpy.org/doc/stable/reference/generated/numpy.polyfit.html

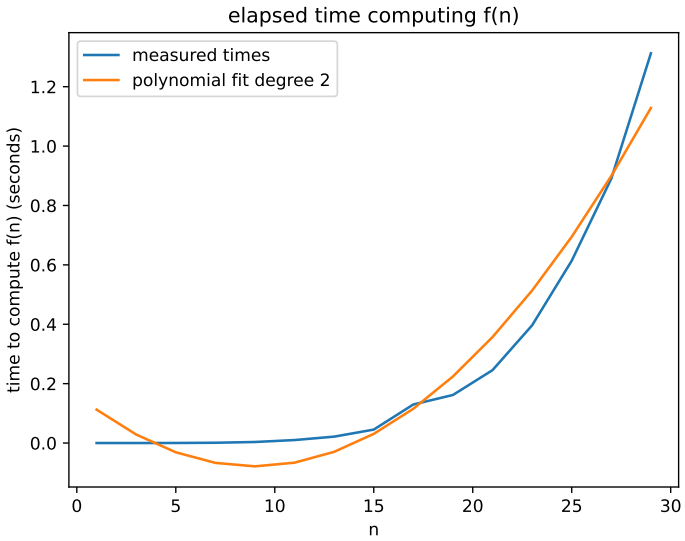

FIGURE 3 – Here, the degree of the polynomial used to fit the curve is too small.

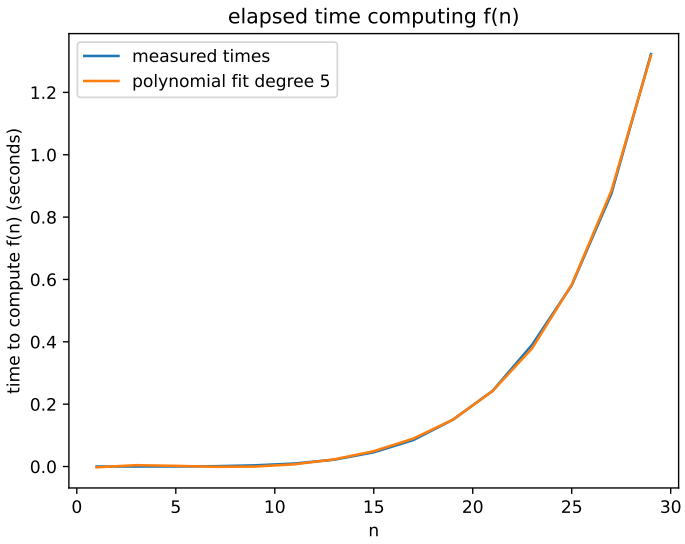

FIGURE 4 – Here, the degree of the polynomial used to fit the curve is correct.

4 PART 4 : AN ABSOLUTE VALUE MINIMIZATION PROBLEM

4.1 Introduction

We consider a vector x of $n = 4000$ random numbers : $x = (x_i)_{i\in[1,n]}$ that are distributed normally around 0, and another number $b \in \mathbb{R}$, also distributed normally around 0, but with a different scale.

For each number x_i with $i \in [1, n]$, we have a choice to "flip" (by multiplying it by -1) it or not. We consider the vector c of integers $(c_i)_{i\in[1,n]}$ that represent this choice : if $c_i = 1$, x_i is untouched, and if $c_i = -1$, x_i is flipped.

The minimization problem that we are looking at to find c that minimizes the **absolute value** of the sum of b and the components of x, flipped or not. Formally, this means :

$$\min_{c\in[-1,1]^n} S(x, c, b) = |b + \sum_{i=n}^{n} c_i x_i| \tag{1}$$

4.2 Exercise

Propose an algorithm that finds c in order to have the smallest possible $S(x, c, b)$. This time, the time complexity of the algorithm has to be smaller than $\mathcal{O}(n^2)$. This means that for instance a complexity of $\mathcal{O}(n \log n)$ is accepted, but not a complexity of $\mathcal{O}(n^3)$. Proove the time complexity of your algorithm (the instructions are the same as in the previous exercises).

Conventions :

— Put your algorithm inside a function named **choose_signs(numbers, b)**

— **Inputs to the function :**

  — numpy array numbers generated with $rng.normal(loc = 0, scale = 0.005, size = 4000)$.

  — float b generated with $b = rng.normal(loc = 0, scale = 0.05)$

with **rng=numpy.random.default_rng()**.

— **Output of the function :** numpy array containing only 1s and −1s, of the same size as *numbers*.

— **Format :** same instructions as in 1, question 3 (use a script instead of a notebook)

Tests : As in 1, there is a test file called **test_choose_sign.py** in the **projet/exercise_4/** folder of the repo. This test file asserts whether :

— **choose_sign()** returns a python list

— the returned array of signs has as many elements as numbers

— the returned array of signs contains only 1s and −1s.

Your solution must pass the tests. You can find some template files in the **exercice_4** folder of the repo, with a random solution and a main file in order to benchmark solutions.

Objective : You solution should be better than the random solution **random_sign()** (given **exercice_4** in the folder) : meaning that on average, your final score should smaller than 25% of that of **random_sign()**. But it is possible to go way lower. Try also to optimize the runtime of the solution (a **faster** solution is better).

5 THIRD-PARTY LIBRARIES

You may use libraries such as networkx or graphviz, for instance for visualisations of the graph, but not for the algorithmic part that is the subject of the corresponding exercise, unless specified.

6 CODE STYLE

Please add a short docstring at the top of each file and in functions, if relevant.
You are encouraged to use **type hints**, and to format your code.
https://github.com/psf/black
https://pycqa.github.io/isort/
https://docs.python.org/3/library/typing.html

7 ORGANISATION

Number of students per group : 3 or 4.

Deadline for submitting the project :

— 1st session (September 19th, 20th, 21th 2024) : October 20th 2024.

— 2nd session (October 24th, 25th, 26th 2024) : November 12th 2024.

The project should be shared through a github repo with contributions from all students. Please briefly indicate how work was divided between students (each student must have contributions to the repository).

Each exercise should be in its own folder.

If you used third-party libraries, please include a **requirements.txt** file in order to facilitate installations for my tests, but please specify whether or not you did use

libraries that were not used during the course. **If you used only libraries that were used during the course, you do not need to add a requirements.txt file.**

https://pip.pypa.io/en/stable/user_guide/#requirements-files

You can reach me by email if you have questions.