PAB-PACE Programming Project
Version 3 - October 26, 2022

António Ravara and João Aires de Sousa

1 Introduction

This document presents the problem you need to implement in Python to obtain the grade to the "Programming Project" assessment component. You should use the approaches and methodologies presented in the classes and use only language constructs covered in the course.[1]

The project is to be carried out in a group of **two students**. Section 5 describes in detail how and when to deliver your work.

Organisation of this document. The next section describes the concepts consider in the project, the main functionalities you should implement, and clarify how the interaction with the user is supposed to happen. To illustrate, Section 3 shows examples of input and respective output for the commands. We will publish shortly the tests.

Section 4 presents the project's evaluation criteria, clarifying the conditions to get a top grade, and Section 5 explain how to submit the project.

2 Project Concepts and Objectives

Efficient storage of information is crucial for computer applications. Not only it allows for better algorithms, but is also critical for performance. To understand how data representation plays a key role when building algorithms to manipulate it, think about numerals: the Roman notation is clearly less practical than the Arabic one for arithmetic operations (just try to come up with an algorithm for addition).

Molecular structures are rich elements that can be naïvely represented as graphs. But then, storage (for instance, in databases), search, composition, establishment of structure-property relationships, or the calculation of observable properties, to name a few, become quickly quite expensive operations, computationally. Moreover, computational structures like dictionaries or databases, require molecular representations that are non-ambiguous — a a representation corresponds to a single molecular structure — and unique — each molecule has only one possible representation.

The SMILES notation. Linear textual notations are particularly useful: they are simple, of low computational cost, usable straightforwardly in text messages or in web forms, and being non-ambiguous can be used as keys to access dictionaries or databases. Chemical line

[1]Check with the lecturer if you can use a feature not introduced in the material presented in classes and available on the CLIP web page of the UC.

notations represent molecular structural formulas by sequences of ASCII characters (usually known as strings).

The SMILES notation (Simplified Molecular Input Line Entry System) [1, 2] provides a compact intuitive representation of molecules by strings: atoms are represented by their symbols and bonded atoms follow each other in the sequence. For example, n-propanol can be represented by "`CCCO`".

The main rules to generate a SMILES string are, in a simplified way:

- Atoms are represented by their atomic symbols enclosed in square brackets (symbols '[' and ']'). Elements in the "organic subset" (`B`, `C`, `N`, `O`, `P`, `S`, `F`, `Cl`, `Br`, and `I`) may be written without brackets.

- Adjacent atoms are assumed to be connected to each other. Single, double, triple, and aromatic bonds are represented by the symbols `-`, `=`, `#`, and `:`, respectively (single and aromatic bonds may be omitted, as well as implicit hydrogen atoms). For example, `C=CC` represents propene.

- Aromaticity can be specified with lower-case atomic symbols.

- Branches are specified by enclosing them in parentheses. For example acetone can be represented as `CC(=O)C`.

- Cyclic structures are represented by breaking one bond in each ring. The bonds are numbered in any order, designating ring opening (or ring closure) bonds by a digit immediately following the atomic symbol at each ring closure. For example, the string `c1ccccc1` represents benzene.

- Configuration around double bonds is specified by the characters '/' and '\'.

- The configuration around tetrahedral centers is specified by @ or @@ written as an atomic property following the atomic symbol of the chiral atom inside the square brackets, and it is based on the order in which neighbors occur in the SMILES string. Looking from the first neighbor of the chiral atom to the chiral atom, the symbol '@' indicates that the three other neighbors appear anticlockwise in the order that they are listed; whereas '@@' indicates that the neighbors are written clockwise.

Main objectives of the project. The aim of the project is to construct a list of SMILES strings with organic atoms `B`, `C`, `N`, `O`, `P`, `S`, `F`, `Cl`, `Br`, and `I`, either reading from a file or interactively (via user input); and then offer operations to manipulate the list constructed, including:

1. counting the number of times each sub-structure from an external list (a given file) occurs in the SMILES strings of the list;

2. counting the number of times each atomic element occurs in the strings in the list and obtain the molecular formula in the simplest form: number of atoms of each element (e.g., `C8NO2`);

3. comparing molecules from their SMILES representation, calculating their dissimilarity – the sum of squared differences of the number of occurrences of the given list of sub-structures in two SMILES.

Imagine you want to compare SMILES s_1 and s_2 with respect to sub-structures ss_1, ss_2, and ss_3. Let's say s_1 contains ss_1 twice and ss_2 once, and s_2 contains ss_1 and ss_3 once each. The dissimilarity of s_1 and s_2 is $(2-1)^2+(1-0)^2+(0-1)^2=3$.

To understand if a structure is a sub-structure of a SMILES, you can look at them as weighted graphs (where single bonds are weigh 1, double 2, etc) and simply look for (proper) subgraphs (just edge containment).

In the context of this project, ignore the hydrogen atoms on molecules – SMILES omit them and we will never consider the omitted ones when implementing the functionalities demanded below. Functionalities printing numbers of occurrences should do so in descending order; functionalities printing strings of occurrences should do so in lexicographic (alphabetic like) order.

Notice that not all letters or pairs of letters correspond to atomic symbols (`A` or `Aa` are not atomic symbols). Valid atomic symbols are only the organic ones (and thus there is no need to use square brackets). Moreover, a SMILES does not start with a number, a round parenthesis, or a bond symbol. It is thus important to validate the input read by your application, to ensure strings are valid SMILES. So, you need to define validating functions for the sub-strings with special symbols, according to the generation rules above (for example, parentheses must be first open then closed; after opening a parenthesis, a bond symbol should occur; etc). Use recursive functions and/or regular expressions to do this validation.

Interaction with the user. The application starts by asking the user if it should load a set of SMILES from a file. If the user responds yes, it then asks for the file name; otherwise, it proceeds to the interactive phase. The file has a positive integer in the first line (the number of strings it contains) and then, in each line, a string that may be a SMILE (needs to be validated. In the interactive phase, the application communicates with the user via a command menu. Valid options are:

C Count the number of times each sub-structure from an external list (a given file) occurs in the SMILES strings of the list.

M Count the number of times each atomic element occurs in the strings in the list and obtain the molecular formula (number of atoms of each element, e.g., `C8NO2`). The output of the command should appear in the terminal and be in lexicographic order.

D Compare a given pair of molecules from their SMILES representation, calculating their *dissimilarity* – the sum of squared differences of the number of occurrences of the given list of sub-structures in two SMILES.

I Input a new SMILES string to be added to the current list, if valid (if not, the application reports it found a problem and waits for the user's to input a new command).

H Help: list all commands.

Q Quit: quit the application.

After selecting an option and providing the required input (if any), the application prints in the terminal the information corresponding to the command and waits for the user's next input. When the user wish to terminate using the application, it types Q to quit and the application asks if the user wants to save the current SMILES list to a file. If the user responds no, it simply terminates. Otherwise, it then asks the file name, writes on its first line a positive integer that is the number of SMILES it will store in the file, write each SMILES string in the list in one line of the file, and afterwards, terminates.

3 User sessions using the application.

To clarify the input/output expected, and to illustrate the information provided by the application, consider the following examples of execution. To help the user understanding when the application is waiting for their input, a *prompt* should be used. It will be the character '>'. When the application starts, it presents the message

```
Load input from file (Y/N)?
```

If the user answers with something different from `Y` or `N`, the application replies

```
Answer invalid
```

and asks again

```
Load input from file (Y/N)?
```

If the user responds `Y`, the application presents the prompt, waits for the file name (say `in.txt`) and tries to read it. If the application fails to read the file, it outputs the error message

```
Failed reading file in.txt
```

If it succeeds, validates each string in the file, loads the valid ones, and either prints

```
SMILES list empty
```

if no string in the file was validated, or prints the valid ones in lexicographic order, one per line. Afterwards (as when the user responded `N`), it presents the menu (see below the command `H`), the prompt in a new line, and waits for the next user input.

After executing each command (but `H` or `Q`), the application presents the command menu followed by the prompt, in a new line. The user either inputs a valid command and the application proceeds as appropriate, or the application replies

```
Command invalid
```

and presents the prompt again.

Command C. When the user types **C**, the application outputs the following message:

```
Input source (F/T):
```

where `F` means File and `T` terminal. In the former case, the application waits for the file name (say `in.txt`) and tries to read it. There will be no validation of the strings provided. If the application fails to read the file, it outputs the error message

```
Failed reading file in.txt
```

If it succeeds, for each substring it prints the number of times it occurs in each SMILES in the list. For instance, if the file `in.txt` contains the strings `Cl`, `CC` and `O`, and the list of SMILES currently stored by the application contain exactly the molecules `CCO`, `C1CCCCC1` and `O1CCOCC1`, the output is:

```
CCO contains Cl 0 times, CC 1 time and O 1 time
C1CCCCC1 contains Cl 0 times, CC 2 times and O 0 times
O1CCOCC1 contains Cl 0 times, CC 2 times and O 2 times
```

If the user chooses to input the strings via the terminal, they should first input the number of strings they are going to provide and then input each string, one per line. The application provide the same output shown above (note the SMILES are presented in lexicographic order).

If the user answered with something different from `F` or `T`, the application replies

```
Input invalid
```

and asks again

```
Input source (F/T):
```

Command M. When the user types **M**, the application either outputs

```
SMILES list empty
```

or for each SMILES in the list outputs its molecular formula:

```
CCO is C2O
C1CCCCC1 is C6
O1CCOCC1 is C4O2
```

Command D. When the user types **D**, the application either outputs

```
SMILES list empty or singular
```

if there are less than two SMILES in the list; or asks for a file and behaves as explained before. If it succeeds reading the file and loads at least a valid SMILE to compare with the ones in the list of the application, then it outputs

```
Give two SMILES to compare:
```

the user enters them (for instance w_1 and s_2), one per line; after reading each the application looks for them in the list and:

- when it does not find one (say, s_1,) it outputs

  `SMILES` s_1 `unknown; input a valid one:`

  and waits the user input;

- after having two valid ones, it outputs

  `Dissimilarity degree between SMILES` s_1 `and` s_2`:` n

  being n the calculated value.

Command I. When the user types **M**, the application waits for the user to then input a string and, if the string s is a valid SMILES still not present in the list, it outputs

`SMILES list updated:` s `inserted`

otherwise, if the string is not a valid SMILES, it outputs

`String` s `is not a valid SMILES`

and when the string is a SMILES already in the list, it outputs

`SMILES` s `already loaded`

Command H. When the user types **H**, the application shows the command menu:

`C`: count the number of times each sub-string from an external list (given file) occurs in the SMILES strings of the list.
`M`: Count the number of times each atomic element occurs in the strings in the list and obtain the molecular formula (number of atoms of each element, e.g., `C8NO2`). The output of the command should appear in the terminal and be in lexicographic order.
`D`: compare a given pair of molecules from their SMILES representation (calculate their dissimilarity, i.e., sum of squared differences between the number of occurrences of the sub-strings in two SMILES).
`S`: select all SMILES strings in the list containing a given (set of) sub-structure(s). The set may be given by the user or read from a file (one substructure per line). The output of the command should appear in the terminal and be in lexicographic order.
`I`: input a new SMILES string to be added to the current list, if valid (if not, the application reports it found a problem and waits for the user's to input a new command).
`H`: help – list all commands.
`Q`: quit – quit the application.

and waits for the next command input by the user.

Command Q. When the user types **Q**, the application asks

```
Save SMILES list to file (Y/N)?
```

If the user answers with something different from `Y` or `N`, the application replies

```
Answer invalid
```

and asks again

```
Save SMILES list to file (Y/N)?
```

If the user responds `Y`, the application presents the prompt, waits for the file name (say `out.txt`) and tries to write it (putting the number of SMILES in the list in the first line of the file and each SMILES in a line of the file). If the application fails to write the file, it outputs the error message

```
Failed writing file out.txt
```

and (as when the user respond `N`) terminates, printing `Goodbye`.

4 Project's Evaluation Criteria

The work submitted will be evaluated with respect to the functionalities correctly implemented and to code quality. Each criteria provides up to 10 values (being 20 the top possible mark). Each point in Mooshak is 0,1 values of the possible 10 values of the functionality part. The grading of each part is independent and does not affect each other (*i.e.*, the quality grade cannot decrease the functionality one nor vice-versa). For example, if you get 90 points in Mooshak and 8,5 values in the quality part, your grade for the project is 17,5 values, if all goes well in the defence.

The **defence** of the work is mandatory, and may take the form of a written or/and an oral discussion. The grade of each group member depends on the grade of the work and the individual performance in the discussion. Consequently, the grades of the two elements of the group may be different.

Functionality evaluates correction and completeness of the results produced. If your code passes all tests of Mooshak, getting 100 points, it means you implemented all functionalities asked and (at least with respect to the tests we defined) your code behaves correctly. Some of the Mooshak tests will be published soon.

Quality evaluates the readability, organisation, and efficiency of your code. You should:

1. divide the code in classes and functions that should be coherent logical units;
2. write simple, well-structured and (as much as possible, given your knowledge) **efficient** algorithms implemented with the most appropriate instructions;

3. choose identifiers that express the concepts they represent, written according to the conventions taught;

4. comment the code as suggested in the course.

Ethics. According to Regulation of Knowledge Assessment at FCT NOVA:

- Fraud exists when:
    (a) you use or attempt to use, in any way, in a test, exam, or other form of assessment, face-to-face or remotely, unauthorized information or equipment;
    (b) you provide or receive unauthorized collaboration in carrying out of exams, tests, or any other individual knowledge assessment test;
    (c) you give or receive collaboration, not permitted by applicable rules in each case, in carrying out practical work, reports or other evaluation elements.
- Students directly involved in fraud do not approve in the discipline and will not earn attendance.

5 Project Submission

Where to submit: To deliver your solution to the problem presented herein, you should submit your code to Mooshak in the **PAB-PACE-Proj** contest. The submission consists of a `.zip` file containing all files to successfully run your program and must contain a file `main.py` file with the `main` function.

Deadline: November 6, 23:59:59.

How to submit: Each group must register for the **PAB-PACE-Proj** contest, according to the following rules:

- The **username** (in Mooshak) must have the form **xxxxx_yyyyy**, where **xxxxx** and **yyyyy** denote the student numbers of the group members.
- The **email address** (to which Mooshak sends the *password*) must be the institutional address of one of the group members.

For example, the group made up of students with numbers 76543 and 76545 is the user with the name 76543_76545

You can resubmit the work as often as you like, until the submission deadline. Only the program with the **highest score** on Mooshak will be evaluated; if there are multiple programs with the highest score, among these, the **last** that has been submitted will be evaluated.

References

[1] Daylight Theory Manual v. 4.9, Daylight Chemical Information Systems, Inc. http://www.daylight.com/dayhtml/doc/theory. Accessed: 2022-09-29.

[2] J. Aires de Sousa. Processing of SMILES, InChI, and Hashed Fingerprints. In A. Varnek, editor, *Tutorials in Chemoinformatics*, pages 75–81. John Wiley & Sons, Inc., 2017.