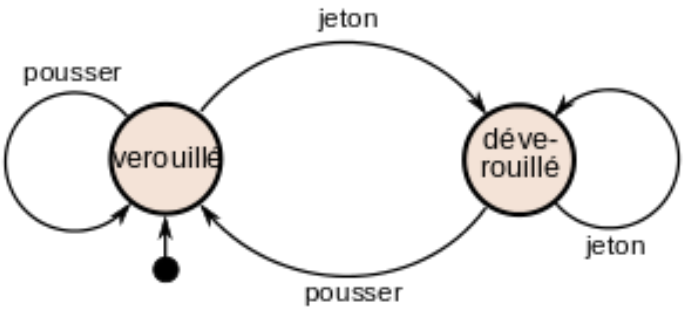


Rapport – Projet Python AEF

BEZET Camille

SALAVILLE Erwan

FORTIER Théo

EL MABROUK Imen

GARCIA-MEGEVAND Thibault

ING1 GIA1

4 Janvier 2024

A l'attention de Mme.ZAOUCHE DJAOUIDA

SOMMAIRE

I. Introduction

A finite automaton or an automaton with a finite number of states is a mathematical model of computation, used, for example, in the design of computer programs or communication protocols.

This project has enabled us to discover automata in greater depth and gain a better understanding of how they work.

The project can be found in this Git repository: www.github.com/camescopetech/ProjetAEF, access has been given to **dze@cy-tech.fr**

II. Organization

A. Tools

To better communicate with each other and work efficiently, we have chosen to work on several platforms simultaneously.

Discord:

We chose the discord platform to communicate and keep abreast of the progress of each other's work. This platform also enables us to hold daily voice meetings.

The different channels make it easy to find our way around the different issues and ensure that we don't lose important messages sent.

Google drive:

We've created a shared drive to pool different documents such as: a summary of problems encountered, meeting notes or this report.

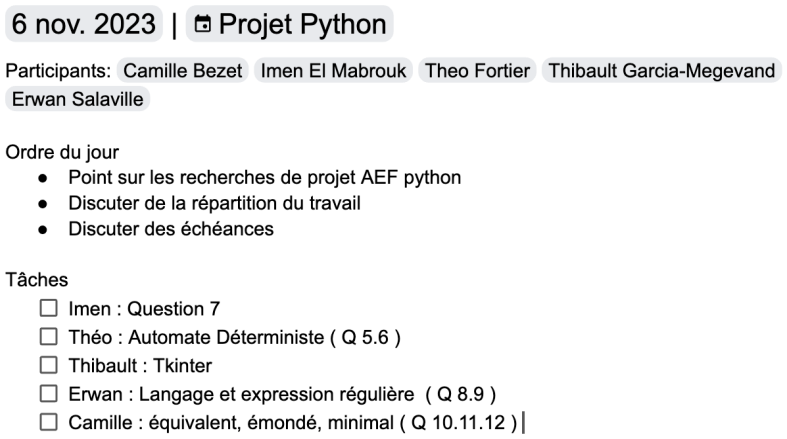
GitHub:

The GitHub platform is a hosting service for projects under development. The site also offers a collaborative space. This makes it easier for several people to work on the same code.

B. Work distribution & workload

Our group has met every week since the start of the project to take stock of what has been done, discuss what needs to be done and any difficulties encountered.

The weekly meetings have enabled us to organize ourselves each week according to what needed to be done most urgently, take the time to discuss it and divide up the tasks.


6 nov. 2023 | Projet Python

Participants: Camille Bezet Imen El Mabrouk Theo Fortier Thibault Garcia-Megevand Erwan Salaville

Ordre du jour
- Point sur les recherches de projet AEF python
- Discuter de la répartition du travail
- Discuter des échéances

Tâches
- [ ] Imen : Question 7
- [ ] Théo : Automate Déterministe (Q 5.6)
- [ ] Thibault : Tkinter
- [ ] Erwan : Langage et expression régulière (Q 8.9)
- [ ] Camille : équivalent, émondé, minimal (Q 10.11.12)

Fig 1. A presentation of how we conducted our reunion : what do we have to discuss and what is to be done for the project

III. Data structure

A. Tables

In our system, the data structure revolves around tables that are central to the functioning of automata.

Each column within these tables represents symbols in the language, serving as essential elements for the interpretation and processing of data. The rows, on the other hand, correspond to states, capturing the various stages or conditions that the automaton can assume during its operation.

Notably, the last two columns consistently denote the initial and final states, providing crucial reference points for understanding the automaton's behavior. This tabular representation not only organizes the language symbols and states efficiently but also facilitates the seamless navigation and analysis of the automaton's dynamics. It forms a foundational aspect of our data structure, playing a pivotal role in the successful implementation and interpretation of automata in our system.

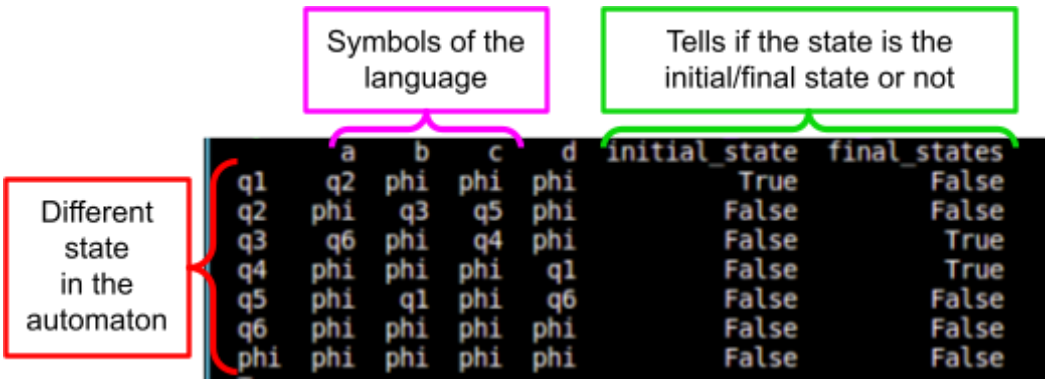


Fig 2. This figure presents how we represent the automaton and how we work with it

B. Dictionary

Our dictionary serves as a fundamental data structure in our system, encapsulating the essential components and specifications of finite automata. The format of the dictionary is designed to comprehensively capture the key attributes of an automaton.

```
{
    'alphabet': ['a','b'],
    'states' : ['0','1','2','3'],
    'initial_state' : '0',
    'final_states' : ['2','3'],
    'transitions': [
        ['0','1','b'],
        ['0','3','a'],
        ['1','1','a'],
        ['1','2','b'],
        ['3','3','a']
    ]
}
```

Fig 3. Example of an automaton in the format we use

We use 5 different keys :

Alphabet

The 'alphabet' key corresponds to a list containing all the symbols that constitute the language recognized by the automaton.
In the provided example, the alphabet consists of the symbols 'a' and 'b'.

States

The 'states' key holds a list enumerating all the possible states within the automaton.

In the given illustration, the states are labeled as '0', '1', '2', and '3'.

Initial State
The 'initial_state' key designates the initial state from which the automaton begins its computation.
In the provided illustration, the initial state is '0'.

Final States
The 'final_states' key contains a list of states that are considered accepting or final states.
In the example, '2' and '3' are marked as final states.

Transitions
The 'transitions' key encapsulates a list of transition rules, where each rule is represented as a sublist. Each sublist consists of three elements:

- the current state,
- the next state,
- and the symbol triggering the transition.

In the given dictionary, the transitions are defined as follows:

From state '0' to state '1' upon encountering symbol 'b'.
From state '0' to state '3' upon encountering symbol 'a'.
From state '1' to state '1' upon encountering symbol 'a'.
From state '1' to state '2' upon encountering symbol 'b'.
From state '3' to state '3' upon encountering symbol 'a'.

This dictionary format provides a clear and structured representation of finite automata, allowing for easy interpretation and manipulation of automaton properties within our system. It serves as a versatile and efficient means of storing and retrieving essential information about the automaton's language, states, transitions, and more.

IV. Final product

A. Main functions

We use many functions to have a working product.

These functions are used every time to have the desired form of an automaton based on the input.

- dict_to_table(dict_automaton) :

take an automaton under his dictionary form and return it under his dataframe form.

- table_to_dict(df_automaton) :
  take an automaton under his dataframe form and return it under his dictionary form.

For the **3rd** and **4th** questions, it is important to make sure an automaton is complete before having to complete or not. So we implemented those functions amongst others :

- is_complete(automaton) :
  return bool according to if the automaton is complete or not
- completing(automaton) :
  return the complete automaton (all missing transition goes to phi , a new state)

The **question 5** asks us to determine whether an automaton is determinist or not. A deterministic finite automaton consists of a finite set of states, a finite alphabet of input symbols, a transition function that specifies how the automaton transitions from one state to another based on the input symbol, an initial state, and a set of accepting (or final) states. The term "deterministic" indicates that for each state and input symbol, there is exactly one possible transition to another state.
The function used begins to check in what type the automaton is and transforms it into a dictionary if it wasn't, the function retrieves in variables the list of initial states, states and transitions. Then the function looks state by state in the transitions list if a letter returns several times in its transitions. Which means it’s not a deterministic automaton. The function does not check if the automaton has several initial states as the format of the project does not allow it.

In the **6th question** we have to make an automaton determinist. The function begins by checking in which type the automaton is and transforms it into a dictionary if it was not. The function initializes a new automaton that will be the deterministic one return by the function, the program thus recovers the states and the list of initial states to put them in the new automaton that will be called afd.
Then the function initializes a queue that will be the queue for unprocessed states. After that a "set" is created, at the expense of a list to avoid duplicates, which will correspond to the created states. We will treat all the elements of the states queue by letter by creating a 'set' of its states reachable by the letter and then checking if the reachable states

are already in the base states of the automaton, if the states are not null and if it is not a state already created. If all these conditions are met it means that it is a state to be created, so we will create it. When we did this we will check if one of the states was final which would make the state final also, then we add the transitions from our state to the state created.

For **question 7**, we need to perform various operations on the AEF (automaton_q7.py) :

- complem_automaton(automaton) :
  takes an automaton (represented by a dictionary) as input and returns its complement. The function then converts the modified dictionary into a data table for clear display.
- miroir_automaton(automaton):
  takes an automaton (represented by a dictionary) as input and constructs its mirror automaton. The function converts this mirror automaton into a data table for easy viewing.
- product_aefs(automaton1, automaton2):
  takes two automaton, and creates their product. The function then defines a find_transitions sub-function to determine the transitions for a given combined state, taking into account the transitions of the two original automata. The function converts this mirror automaton into a data table for easy visualization.
- concatAEF(automaton1, automaton2):
  takes two automaton, and creates their concatenation. The function converts this concatenated automaton into a data table for easy viewing.

The **8th** question is about finding the regular expression (regex) of an automaton (regex_q8_q9_q10.py).

- find_regex(automaton) :
  Take an automaton and return its regular expression

The process to do that is :

1. Look for the ending path with no loop
2. Then find every loop
3. For each ending path, spot every loop (an imbricated cycle is a loop who can be followed through the precedent loop) which could be used and who does not take the end path way.
4. Each loop can be followed only once per position of the path. Do this at every position of the ending_path

5. The regex will be cleaned and optimize with subfunction below

The **question 9** just returns the regular expression found in question 8 as a mathematical set. The form is the word that the regex satisfy

For the **10th question** we have to determine if 2 automaton are equivalent, which means answering the question “do they return the same language ?”

- is_automaton_equivalent(auto1,auto2):

Generate a list of words proportional to the automaton size for each of the automatons, using q2. Then verify that every word recognized by the first automaton is also recognized by the second one, and vice versa.
Returns False if a test failed, True else

The **11th** question is about excising an automaton (excise.py).

- remove_unreachable_states(automate) :
  remove all unreachable state
- remove_epsilon_transitions(automate):
  remove all epsilon transition

To do so we need to identify unreachable state and epsilon transition. An unreachable state is a state not accessible from the initial state. An epsilon transition is a transition only used to access a state from another, but doesn't change the structure of the automata.

For the **12th question** we have to look for ending paths with no loop and extract the states these paths borrow.
To find ending paths, from the initial state, we use a stack containing the paths borrowed. We then add to the stack all the states where we can go, ignoring states already borrowed. An ending path is found if the current state is a final state.
If the loop found add the states in the loop only if one or more of them is on the previous list (ending paths)
Then delete others states and transitions to those states

B. Graphic interface

We opted for tkinter as our graphical user interface library in Python due to its simplicity and effectiveness. The graphical interface is divided into three main components: a terminal, a text editor, and an options bar.

Before executing any function that requires an automaton as input, we perform a validation check using the "isAutomateSpring" function (located in interface.py). This function verifies whether the provided automaton adheres to the expected format and returns an error code corresponding to any identified issues. In cases where two automata are provided as input, they must be separated by a newline character in the following format :

```
{
json1
}
{
json2
}
```

If the function returns a modified automaton, the text editor's content is replaced with the new automaton in JSON format (using “addJsonOnTextArea”). Otherwise, the result is displayed in the graphical interface's terminal (using “insertTerminal”).

Our system incorporates a comment system using the symbols ',*' and '*,'. We have implemented functions to extract the automaton without comments (“getTextAreaJson”) and to retrieve comments separately (“getTextAreaComment”), ensuring that comments are preserved during potential automaton rewriting.

Furthermore, we have implemented a function to visually represent the automaton by drawing (“drawAutomate”) it in separate windows. The first step of this function arranges the states in a circular pattern, while the second step involves adding links and details, highlighting the initial and final states for clarity in the graphical representation.

This visual editor also gives the possibility to import an automaton from a file, save it in a file or delete it.

In order to start the editor only the following command is needed :

```
pyhton3 index.py #if your python version is > 3
pyhton index.py #otherwise
```

Some libraries may need to installed such as :

- Pandas
- Tkinter

V. Conclusion

In conclusion, this Python project has been a very rewarding experience for our group. We acquired new skills and knowledge throughout the process of designing and building this product.

Thanks to this project, we were able to familiarize ourselves with the key technologies studied in this module, such as the Pandas library or dictionaries. We were also able to interact with a visual interface to display and manage information.