Operations on Finite Automata

Word recognition

Please take the time to read this document carefully. In particular, it contains certain instructions that you must absolutely comply with. Any failure to do so will inevitably influence your grade.

Further information (clarification, additional information) may be provided at a later date.

Table des matières

PROGRAM TO BE DEVELOPED

Advice: Initially, when you start working on the project, don't take into account the epsilon transitions and program the next steps directly. Afterwards, when the notion has been studied in lectures and TDs, you will be able to take epsilon transitions into account.

Your program is divided into several stages:

- reading a FA (in a file), storing it in memory and displaying it on the screen;
- testing if there is at least one epsilon transition in the FA (if the FA is asynchronous)
- if the automaton is not complete deterministic, obtaining the equivalent complete deterministic FA;
- calculating the equivalent minimal FA;
- testing word recognition ;
- creating an automaton recognizing the complementary language and testing word recognition on that automaton;

Standardization of the automaton without epsilon transitions should be available at any moment, but it will be a separate operation: all operations that follow will be carried out on the non-standardized automaton.

Reading a FA from a file

Some time before the presentations, we will send you the test automata. You will have to transcribe them into".txt" files. These files should be named as follows:

<team number>-#.txt

where the # will be replaced by the number of the test FA. For example, if you are the team n° Int2-3 and you are dealing with test FA number 8, the file must be named Int2-3-8.txt.

To begin with, your program must read the description of a FA in a text file, and save it in memory according to your choice of data structures.

During the execution of your program, the user will choose which test FA he or she wants to study. It must be possible to make this choice in a very simple way: for example, to choose the test FA n°8, it should be sufficient to type "8" in answer to the question "which FA do you want to use?"

As long as you have not yet received the test automata (which will be given to you in a graphic form!), you are free to work on any automaton of your invention, encoding it in a .txt file according to the model you will see below.

Entering a FA from the keyboard (without a file) is forbidden. It is also forbidden to "hard-define" a FA in the program.

The representation of a FA in memory will depend on your choice of data structures. You are free to choose the data structures that you think are most appropriate.

Displaying FA on screen

Displaying the FA stored in memory, with an explicit indication:

- of the initial state(s);
- of the terminal state(s) ;
- of the transition table.

The steps in reading and display can be represented by the following pseudo code:

```
FA ← read_automaton_from file (filename)
display_automaton(FA)
```

Determinization and completion

Let FA be the automaton obtained (and displayed) at the previous stage. The operations can be described by the following pseudo code:

```
IF is_an_asynchronous_automaton (FA)
THEN
      CDFA←determinization_and_completion_of_asynchronous_automaton(FA)
ELSE
      IF is_deterministic (FA)
      THEN
            IF is_complete(FA)
            THEN
```

```
                    CDFA←FA
                ELSE
                    CDFA←completion(FA)
                ENDIF
            ELSE
                CDFA←
                determinization_and_completion_of_synchronous_automaton(FA)
            ENDIF
        ENDIF
        display_complete_deterministic_automaton(CDFA)
```

where

`is_an_asynchronous_automaton (FA)`

Check whether it is an asynchronous FA (whether it contains epsilon transitions).

The result of the test is displayed. If the FA is asynchronous, your program must also tell the user which elements make it asynchronous.

`determinization_and_completion_of_asynchronous_automaton(FA)`

Construction of a synchronous, deterministic and complete finite automaton from an initial asynchronous automaton (FA).

You have a choice between:

1) a direct determinization (accompanied by an obligatory display of epsilon closures).

2) elimination of epsilon transitions using `epsilon_elimination_(FA)` followed by a usual determinization and completion:

`epsilon_elimination (FA)`

Replace all initial states and states that are targets of at least one non-epsilon transition by their epsilon closures, and keep only non-epsilon transitions. Properly identify initial and final composite states of the resulting automaton.

`determinization_and_completion_synchronous_automaton` **`(FA)`**

Construction of a complete deterministic automaton from a synchronous non-deterministic automaton Here, you should use the algorithm of determinization given in lectures and worked in problem sessions. It should be done on the initial automaton (except if it is already synchronous, deterministic and complete) and not on a standardized automaton even if standardization was previously requested by the user.

`is_deterministic` **`(FA)`**

Check whether the synchronous automaton FA is deterministic.

The result of the test is displayed on screen. If the FA is not deterministic, your program must tell the user why.

`is_complete` **(FA)**

Check whether the synchronous and deterministic automaton FA is complete.

The result must figure on screen. If the automaton is not complete, your program must tell the user why.

`completion`**(FA)**

Construction of a complete deterministic automaton from a deterministic automaton FA if the latter is not complete.

`display_complete_dererministic_automaton(CDFA)`

Display the automaton on screen using the same format as the one you used for displaying the initial automaton.

Besides displaying the complete deterministic automaton (CDFA), your program must explicitly show the composition of the CDFA states in terms of the states of the original automaton.

Remark: it is preferable to directly use the way of labeling the states of the CDFA as sets of the original states, like "123" for a state composed of 1, 2, and 3. For numbers greater than 9, it is important to make a difference between, say, {1,2,3} and {12,3}. You can implement that by, for example, using a separator of your choice, like "1.2.3" and "12.3".

Warnings:

- The various tests figuring in the pseudocode are obligatory. For example, you can launch determinization only if the FA has been identified as not being deterministic. Deterlminization of DFA will be considered as an error.
- Determinization of a FA does not include standardizing it. You must determinize the FA (if it is not deterministic) such as it figures in the .txt file imported by your program.

Minimization

There are no tests to do. Your program will minimize the CDFA obtained above and display the result. If the algorithm finds out that the automaton was already minimal, your program must tell the user about it.

Pseudocode:

```
MCDFA ← minimization(CDFA)
display_minimal_automaton(MCDFA)
```

where:

`minimization(CDFA)`

Constructing a synchronous, deterministic, complete and minimal finite automaton (`MCDFA`) from a synchronous, deterministic, and complete finite automaton (`CDFA`).

Notice that there is no verification if the automaton is already minimal.

Your program must display successive partitions, as well as transitions expressed in terms of parts (groups), all the way along the minimization process. Make that display easily readable `minimization(CDFA)`

Constructing a synchronous, deterministic, complete and minimal finite automaton (`MCDFA`) from a synchronous, deterministic, and complete finite automaton (`CDFA`).

Notice that there is no verification if the automaton is already minimal.

Your program must display successive partitions, as well as transitions expressed in terms of parts (groups), all the way along the minimization process. Make that display easily readable.

`display_minimal_automaton(MCDFA)`

Displaying the minimal automaton using the same form as before.

Your program must explicitly inform the user to which state(s) of the initial automaton corresponds each state of the minimal automaton. This correspondence may be directly used in the transition table, or, if you opt for renaming the states of the minimal automaton, there must be a separate "table of correspondence".

Word recognition

Your program should now analyze words typed on the keyboard by the user. It ***must*** allow the user to test several words one after the other, without having to re-select the word recognition operation.

Pseudocode

```
read_word ( word)
WHILE word ≠ « end » DO
      recognize_word ( word, A )
      read_word ( word)
REDO
```

where:

`read_word (word)`

storing in memory a string of characters typed by the user on the keyboard.

Warning:

-You must absolutely give the user a way to type in an empty word.

-The word must be fully read as a string before it is tested. There should be absolutely no letter-by-letter reading and testing.

-You must tell the user what to type in order to signal that he or she wants to stop reading words. The pseudocode above proposes the string "end", but you may choose otherwise.

`recognize_word ( word, A )`

Verifying if the automaton A recognizes the input word.

Result: Yes or No.

Optional: informing the user which is the first character of the word that prevents it from being recognized.

Since all automata your program has dealt with so far are equivalent, you can test the word recognition using any kind of automaton (non-deterministic, just

deterministic, deterministic and complete, minimal). It is, however, easier to use a deterministic automaton for that purpose than a non-deterministic one. If you opt for using a complete deterministic automaton, you may freely choose between a non-minimized or a minimized automaton.

Complementary Language

Your program must construct a FA that recognizes the language complementary to that recognized by a given FA, and test which words are recognized by this complementary FA.

Pseudo code:

```
AComp ← complementary_automaton ( A )
display-automaton ( AComp )
read_word (word)
WHILE word ≠ "end" DO
      recognize_word ( word, AComp )
      read_word (word)
REDO
```

where:

`complementary_automaton ( A )`

Constructing an automaton recognizing the complementary language.

The entry parameter A can be, at your choice, the previously obtained CDFA or MCDFA. Your program must tell the user which automaton has been used to obtain a complementary.

Standardization

Your program must be able to standardize an automaton obtained at any stage of the programmed sequence. Pseudocode:

```
ACompStd ← standard_automaton ( A )
display_automaton ( ACompStd )
```

```
read_word ( word)
WHILE word ≠ "end" DO
      recognize_word (word , ACompStd )
      read_word ( word)
REDO
```

where:

`standard_automaton ( A )`

Standardization of the previously obtained automaton 'A'.

If the automaton A is already standard, your program must inform the user about it and it should not modify it If you have succeeded in building all the code described above, the following sequence of automata will be obtained:

FA → CDFA → MCDFA → AComp

with lateral arrows of standardization that must be possible at any moment except if the automaton is asynchronous.

IF some of the stages have not been done successfully, you may obtain different sequences. For example:

Minimization not done:

FA → CDFA → AComp

Evidently, you'll have to clearly announce during your presentation what exactly your program is capable of doing.

The multiple automata loop

Il est impératif de mettre tous les traitements identifiés ci-dessous dans une boucle générale permettant de traiter plusieurs automates AF sans relancer votre programme.

ENVIRONNEMENT DE PROGRAMMATION

You can use the languages such as C, C++, Python, or Java.

Your program must be compilable and executable on my PC. I'll tell you the software versions I will have installed. (You may give me advice if you wish).

AUTOMATA YOUR PROGRAM MUST BE ABLE TO DEAL WITH:

The tests will be done on automata:

- dealing with the alphabet consisting of characters from 'a' to 'z':

  for example, an automaton whose alphabet consists of 3 symbols will use the characters 'a', 'b', and 'c';

- whose states are labeled by numbers starting from '0':

  for example, an automaton consisting of 5 states will have the states labeled from 0 to 4, without breaking the sequence.

There is no limit on the number of states a test automaton may contain.

The text file representing an automaton read by your program may have the following syntax (but you may prefer a slightly different structure if you wish):

Line 1: number of symbols in the automaton's alphabet.

Line 2: number of states.

Line 3: number of initial states, followed by their numeric labels.

Line 4: number of final states, followed by their numeric labels.

Line 5: number of transitions.

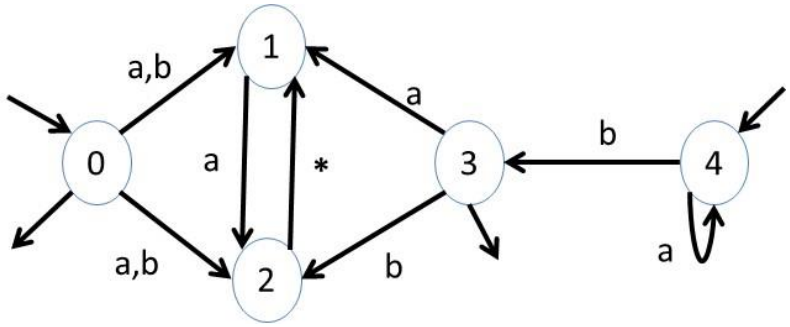
Lines 6 and the following lines: transitions in the form

<source state><symbol><target state >

For an asynchronous automaton, an epsilon-transition can be represented by the symbol "*".

For example:

Automate:



2 symbols in the alphabet

A={a,b}

5 states

Q={0,1,2,3,4}

2 initial states

I={0,4}

2 final states

T={0,3}

10 transitions (of which one is an epsilon-transition)

File:

```
2
5
2 0 4
2 0 3
10
0a1
0a2
0b1
0b2
1a2
2*1
3a1
3b2
4b3
4a4
```

TECHNICALITIES

The teams

Group Int1 consists of 37 students, which makes 5 teams of 5 + 3 teams of 4.

Group Int2 consists of 39 students, which makes 7 teams of 5 + 1 team of 4.

Group Int3 consists of 39 students, which makes 7 teams of 5 + 1 team of 4.

This means that group Int1 will form 7 teams of 5, and group Int2, teams of 4 and 4 teams of 5. The list of teams must be in my hands (sent to me by your delegate in a template file I'll provide) by March 19.

Tests (preparing the presentations)

Some time in the future you'll be given test automata you'll have to use during your presentation.

At the presentation, all the text files corresponding to test automata must be present on your computer and accessible to the program. It's your responsibility to ensure that your PC works well and that your program can be compiled and/or executed on your PC.

There will be no computer provided by EFREI.

The contents of what you should send:

- Source code: exclusively the files you typed yourselves. **Not a single file produced by your software** (Codeblocks, Anaconda, etc.) should figure in what you send. The code must be sufficiently commented so that it is easy to read it by somebody who has not participated in its development. This is especially important for a C/C++ code.
- All files you are sending must be named so that the file name starts with the team name : suppose you are a team Int2-3, and have a C++ file you would otherwise call « main » ; in your project, that file should be named Int2-3-main.cpp.
- All files .txt of the test automata, named in the same way, must figure in the same directory as the code files.
- ***Possibly***, a ppt or pdf file used during the presentation, named in the same way. (You may, if you prefer, send me the presentation file on the defense day).
- The execution traces (text files (.txt) created by your program and reproducing what the user will see on screen; they cannot be replaced by screen copies. There should be as many execution traces as there are test automata, and they should be named using the same principle.

Sending in your project

The deadline for sending in the project will be given you slightly later. (The deadline will be the same for all groups, be it the French or English groups).

Presentation

The calendar: the slots marked PRJ in your agenda. In case there is no lockdown, I'll ask your delegates to send me your choice of time slots by team. In case of lockdown, I'll simply assign slots to teams in some natural order.

Duration : 45 minutes per team.

The presentation sequence: an oral presentation, demonstration, possibly an additional discussion.

Oral presentation

length: 15 minutes.

Each team member will present a part of the total, without any help from the others. Make sure you know who presents what!

The presentation times of each team member, as well as the relative difficulties of the operations presented by each team member, must be comparable to each other. It's your responsibility to ensure that is the case.

You must prepare a written presentation support on paper (which, apart of being possibly sent to me together with other files, should be physically given to me – on paper! - in the very beginning of your presentation).

Your data structures and algorithms must be clearly described in that support.

***Advice**: prefer a clear and precise diagram or a well-structured pseudo-code that you will comment on, rather than a long text that cannot be read during the presentation! Your data structures and algorithms must be absolutely clear, and you must not use the C or C++ code directly for the presentation. Be complete and precise: saying, for example, that an*

automaton is represented by an object class is not enough if we do not know more precisely the data structures used and the functions allowing their manipulation; saying that it is an array is not enough either if we do not know what the cells of the array contain... ***Explaining a data structure means making people understand without having to ask additional questions about how a FA is represented in the machine memory****. It is also necessary to spontaneously specify, without waiting for a request, if you are using the same data structure or two different data structures for non-deterministic and deterministic automata.*

Warning:

It is advisable to prepare a Powerpoint presentation. If the defences take place on campus, the rooms are usually equipped with a video projector or with a TV screen. If this is not the case, a laptop with a sufficiently readable screen can usually suffice. And in case we are we must conduct the defences on Teams, there will be little difference: you must turn on your camera, share your screen and present as if we were at school.

Put all the text files representing the test automata in your working environment **before** the presentation.

You will compile your program (which must be exactly what you send to me, without any last-minute corrections or additions) in my presence, then you'll launch execution. I will choose the automaton to work on, and then I will test the recognition of some words. I will also have the right to suggest a little change in an existing automaton, that we'll save, read and test; or even an entirely new automaton, easy enough to type as text in a file.

Make sure your computer is working (the battery is charged, the system is on, …) **before** you enter the room!

Discussion / Questions and Answers

Naturally, I can ask you questions at any moment. I can also ask a specific person to answer, without any help from the others. You have to understand that even though you may not directly participate in work on all components of the program (which is perfectly normal for a team work), each must be capable to answer any sufficiently general question about the project. It is

advisable for each team to meet regularly, and during those meetings you should explain to each other the work each has done.

Besides giving to each of you the information necessary for the presentation, this will also force each of you to learn how to implement all algorithms you will have learned in this course.

SOME GRADING TIPS

Naturally, the grade will depend on the functioning of your program as well as on the quality of your presentation. The following elements will also influence on the grade:

- Clarity of the man-machine interface, allowing for a smooth testing of your program and for an easy and versatile demonstration (sequence of actions, execution trace, …). It should be easy for someone not having participated in the work on the program to check the results.
- The choice and the justification of the of data structure(s) representing automata.
- The readability of the code. This includes commenting. The absence of comments is usually perceived, except rare cases of extremely well written code, as a badly readable code.

Each team will get a common grade, but this grade may be raised or lowered individually depending on that person's oral presentation and/or answers to questions.