FINAL PROJECT OF AUTOMATA THEORY

AMIN MOHAMMAD SOLEIMANI ABYANEH 94100805[1]

CONTENTS

LIST OF FIGURES

* *Department of Computer Science, Sharif University of Technology*

1 INTRODUCTION

Automata theory is a theoretical field in CS, which mainly discusses the logic of computation with respect to basic and abstract machines, called automata.This study enables the scientists to understand the computers are able to comprehend the functions and solve difficult problems. In this context, we are mainly engaged in dealing with Finite State Machines, e.g. DFA and NFA, and Push Down Automata. As we already know, the Deterministic Finite Automata (DFA) and Non-deterministic Finite Automata(NFA) are employed to recognize a Regular Language while PDA is used to process a Context Free Language.

In this project, we are supposed to deliver three main algorithms which we have studied during the course. The main code consists of an I/O interface in command line which implements the desired input and output handling. In each of the next sections we present one of the following problems of the project. The three main functions that we have to implement are as the following.

- Convert a Regular Expression to NFA and vice versa.
- Conversion of NFA to DFA and ultimately to minimum DFA.
- Conversion of a Context Free Grammar to PDA.

The source project is also available in `https://github.com/aminabyaneh/Automata_Theory`. The repository also contains some simple and complex test cases, as well as the development history and corresponding algorithms.

WARNING The source code is implemented entirely in JAVA 11, so using lower versions of Java to run the code may cause unanticipated errors and warnings. The code is also explained in the sub-section of each respective task and the required objects and functions are mentioned under the sub-section of implementation, however, it also contains a comprehensive Javadoc which is attached to this report.

2 GENERAL STRUCTURE OF THE CODE

The code consists of several parts ranging from input handling to various automata algorithms. To chase the structure of the code, one should start with the function automata(), called in the main. This function is a state machine responding to various types of input data. The class IO is responsible for handling the input and output methods, which first reads the first line, and then calls a corresponding functions based on the type of the input: Regex, NFA, or CFG.

The code then creates a container which stores the values of input into one of the following classes: CFGEntry, NFAEntry, and RegExEntry which are all children of the super class Entry. These containers store the data and provide a complete set of functions to manipulate the entry data. And, are passed to the corresponding objects to generate the data needed for the algorithms.

Afterward, there are again three main objects: DFA, NFA, and CFG, where DFA and NFA are children of a super abstract class called FSM. These classes all have a field of the object StateTransitionMatrix which is the state transition matrix of the object. State transition matrix is a class that uses inheritance to build a 3-dimensional ArrayList. Each column is implemented as a 2-dimensional ArrayList and contains number of cells which are all 1-dimensional ArrayLists. In addition to the matrix, each of the mentioned classes comprise of a set of functions to implement algorithms like building NFA from Regex and creating the minimum DFA. For determining the structure, it is also useful to look at the following code snippet.

```
/** The main automata function. */
private static void automata() {
IO ioHandler = new IO();
switch (ioHandler.getDataType()) {

case RegEx:
RegEx regEx = new RegEx((RegExEntry)ioHandler.getData());
regEx.taskHandler();
break;
case NFA:
NFA nfa = new NFA((NFAEntry)ioHandler.getData());
nfa.taskHandler();
break;
case CFG:
CFG cfg = new CFG((CFGEntry)ioHandler.getData());
cfg.taskHandler();
default:
break;
  }
}
```

3 REGULAR EXPRESSION TO NFA AND VICE VERSA

In this section, we investigate the procedure of converting RE to NFA and vice versa. We first explain the forward direction of the problem and later explain the postulated reverse direction.

3.1 Regular Expression to NFA

To convert a regular expression to an NFA, we have to follow a recursive procedure based on Thompson construction rules. The recursion reduces the computation complexity and divides the problem into more understandable sub-problems which are then solved by constructing a trivial Automata. In the next sections we first discuss the algorithm behind the scene, then explain the implementation and finally the input output structures and the results.

3.1.1 *Algorithm*

To convert a Regex to NFA, we go through several stages. The goal is to parse the Regex correctly and break the expression recursively to reach trivial NFA, and then merge the NFAs into the final NFA. To build up the final NFA, we break down the Regex to one of the following situations in Figure 1 on the following page. As it appears from the figure, each cas 1 to 4 corresponds to an special case of operations which is listed here.

1. The trivial NFA, where we only have a symbol like a or b.
2. The union NFA, whenever we have two NFAs like $N(s)$ and $N(t)$, we construct the NFA of $N(s)|N(t)$ like this.
3. The concatenation NFA, whenever we have two NFAs like $N(s)$ and $N(t)$, we construct the NFA of $N(s).N(t)$ like the picture.
4. The start NFA, whenever we have a NFA like $N(s)$, we construct the NFA of $N(s)^*$ like mentioned in the picture.

To further illustrate the algorithm, consider the Figure 2 on page 6. Here you can se that we first have to break down the regular expression down to the trivial Automata, i.e. case 1 in Figure 1 on the following page, and then step up merging Automatas based on case 2 to 4 in Figure 1 on the next page.

3.1.2 *Implementation*

The implementation consists of several stages which are explained one by one in this section. The main function in this case generates a

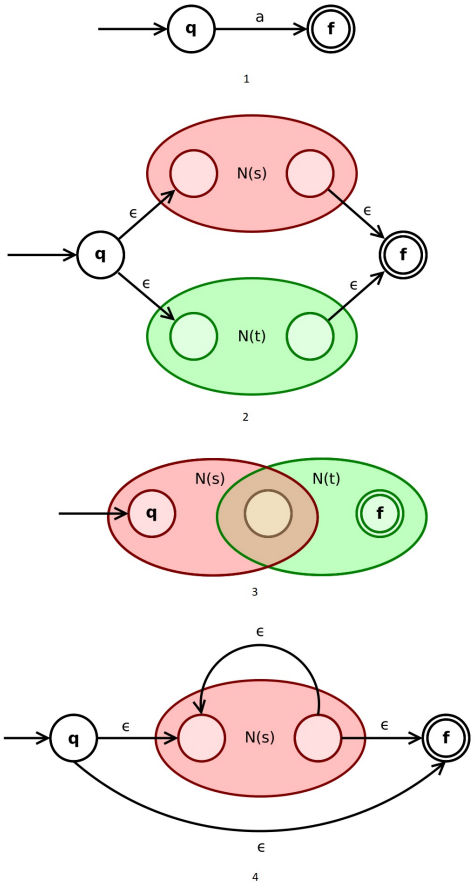


Figure 1: Thompson construction rules

Regex object and Regex entry class to store and manipulate the input data. Then the following stages come along. The first step is to call the function `createNFA()` in the `RegEx` class.

REFORM THE INPUT REGEX Here we add the symbol "." whenever there is a need for concatenation. By doing this we add "." to the set of operations that we already have, $|, *$. This is done by the function `addConcatenation()`. We also add parenthesis down to even a single symbol to make the processing more smooth using the `addParenthesis()` function. The code snippet in the next page is a more clear example of what is really happening in the `creatNFA()` function. Remember that after building the NFA we have to add the start and final state to the `requestedNFA` object which we are going to return.

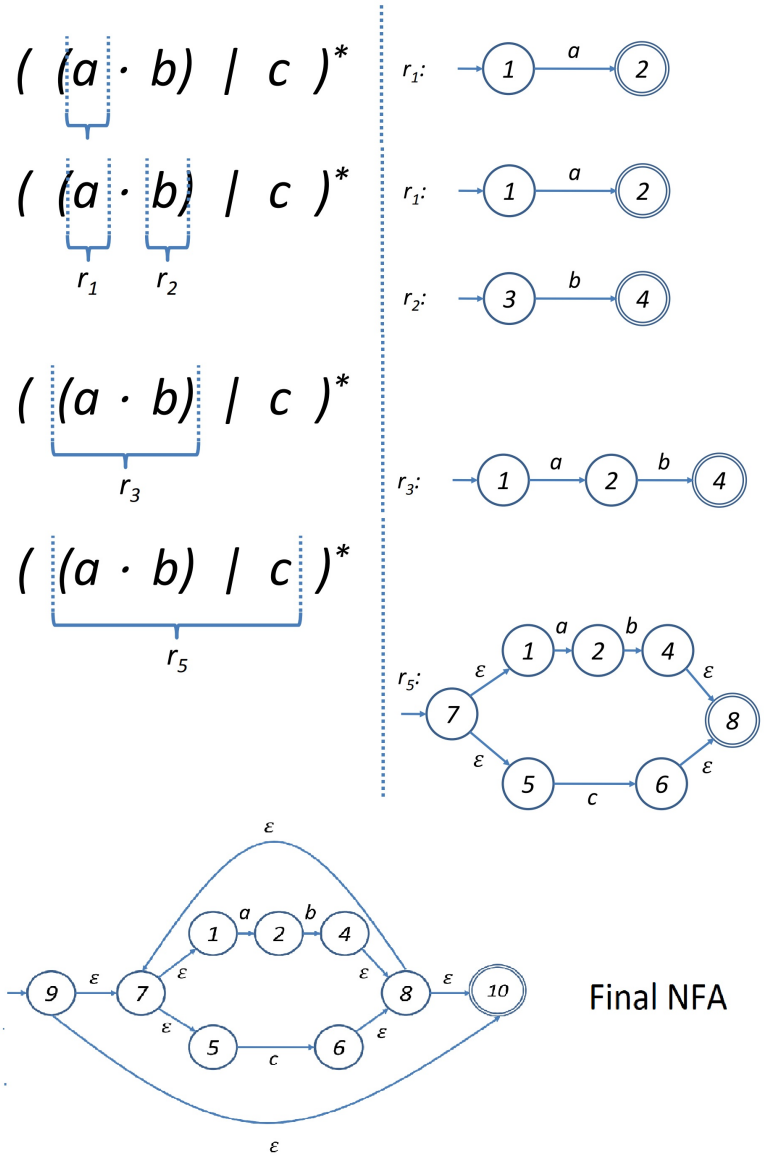


Figure 2: Thompson construction procedure

```
/**
* Creates the NFA. This function constructs and NFA recursively
    and
* returns the result.
*/
private NFA createNFA() {

  this.regex = this.addConcatenation();
  this.regex = this.addParenthesis();

  /** Build NFA from RegEx recursively. */
  NFA requestedNFA = this.buildNFA(this.regex);
```

```
  /** Set start and final state as they are first and last
      state. */
  ArrayList<Integer> finalStates = new ArrayList<Integer>();
  finalStates.add(requestedNFA.stmat.size() - 1);

  requestedNFA.setStartState(1);
  requestedNFA.setFinalStates(finalStates );

  return requestedNFA;
}
```

BUILDING THE NFA Now that we have reformed the Regex, we pass it to the function buildNFA. This function is a recursive function that creates the it is time to wrap it as a Phrase object. The build function first checks the trivial case, which is when the Regex length is equal to 1, and then uses the function extractPhrases(regex) to return the phrases inside a regex. Phrases are part of a regex with the lowest and same parenthesis priority. One may notice that when we are extracting the phrases, there based on the reformation which we performed previously, all the phrases with the same priority have the same operation between them, e.g., the operation between all of them is concatenation or union. Next, the function begins to check whether the phrases have the Kleene star or not. If it is the case, the code performs the starNFA(NFAnfa) function which then applies the star to the nfa and returns the starred nfa. Similarly, the concatNFA(nfaA, nfaB) function and the unionNFA(nfaA, nfaB) function are used to build the concatenation and union results. This is explained in details in the next paragraph.

MICRO OPERATIONS By convention, we call micro operations as the operations that are based on Thompson construction rules. To perform these kinds of operations, we need to apply the changes in the state transition matrix. To illustrate the method, let us consider the function buildUnionNFA(stmA, stmB) in StateTransitionMatrix. This functions works similar to other functions building concatenation and star operation. First, there are two NFAs which we want to perform union operation on. We pass their state transition matrix to the function buildUnionNFA(stmA, stmB). Then, inside the function, we first increment the state numbers if stmB and then merge it with stmA. Merging is done in a way that satisfies case 2 of the Figure 1 on page 5. At the end, two start and end states are added to the table and connect the two previous NFAs via epsilon, simulating the union function based on Thompson rules. The following code snippet may help the reader to understand the concept of micro operations on the state transition matrices.

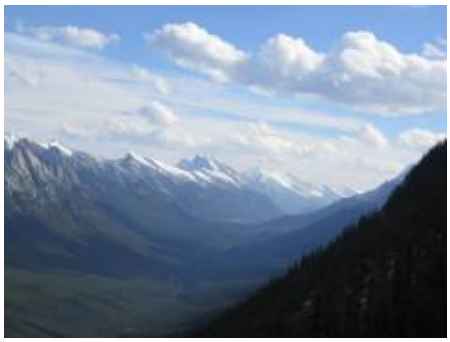
```
/** Building the state transition matrix for union NFA. */

/** Increment state numbers to add the new states. */
StateTransitionMatrix.incrementStateNumbers(nfaA);

/** Add two new states for the start and for the end. */
ArrayList<ArrayList<Integer>> startColumn =
new ArrayList<ArrayList<Integer>>();
ArrayList<ArrayList<Integer>> firstColumn =
nfaA.stmat.get(1);

/** Adding the start state to the beginning of state transition
    matrix. */
Integer index = -1;
for (ArrayList<Integer> c : firstColumn) {

  index++;
  changeCell = new ArrayList<Integer>();

  if (index == 0)
    changeCell.add(c.get(0) - 1);
  else if (index == firstColumn.size() - 1)
    changeCell.addAll(startStates);

  startColumn.add(changeCell);
}
```

3.1.3 *Results*

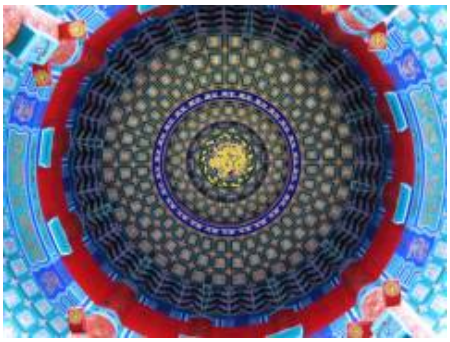
In this subsection, we present some of the major results of the implementation on some arbitrary regular expressions. The input output of the console are aligned with the regulations in the document provided with the project. Here we see results for three regular expression, from simple to rather complicated ones. The results are shown in Figure 3 on the next page.

(a) A city market.

(b) Forest landscape.



(c) Mountain landscape.



(d) A tile decoration.

Figure 3: Results of the Regex to NFA converter

3.2 NFA to Regular Expression

3.2.1 *Algorithm*

3.2.2 *Implementation*

3.2.3 *Results*

4 NFA TO MINIMUM DFA

This is the second required task of the project which comprises of two main sections: creating a DFA from a given NFA, conversion of DFA to minimum DFA. The methods related to this part of the project are implemented inside the DFA object. In each section, we use a given algorithm to convert an FSM to another FSM with certain specifications. We investigate each of the major parts of the algorithm in the next two sections. A picture of the conversion sequence may be found in Figure 4 on the following page.

4.1 NFA to DFA

In the book "Introduction to the Theory of Computation", a complete algorithm is suggested to convert an epsilon-NFA to a DFA. The con-

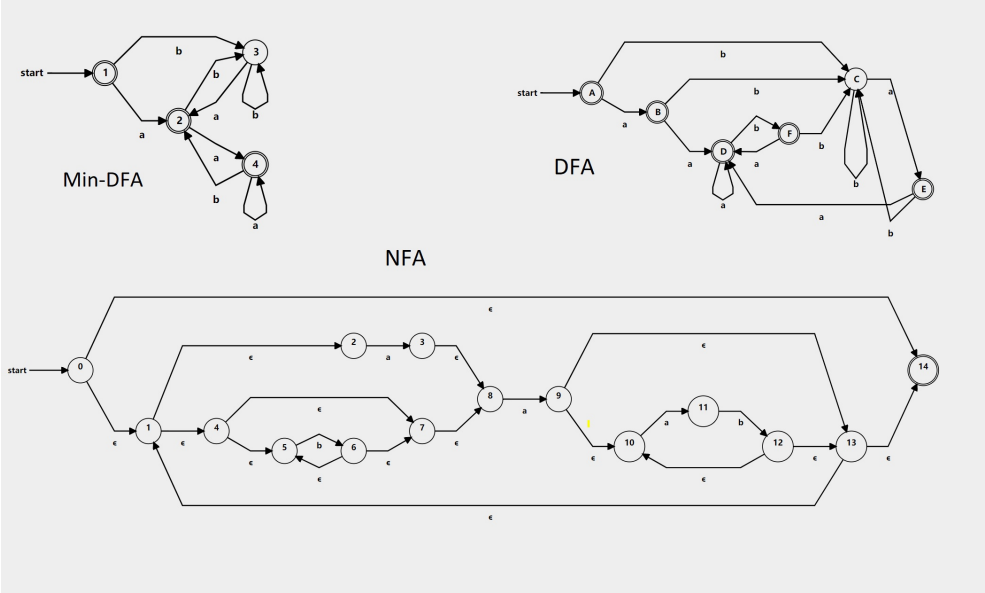


Figure 4: NFA → DFA → Minimum DFA

version, however, is not always minimal; it may be reduced to an even more compact DFA. This section explains the algorithm behind a non-optimum transformation of a desired NFA to corresponding DFA. We get here by calling the function `buildDFA()` on an arbitrary DFA object, where the algorithm gets the state transition matrix and final and start states of the NFA and passes it to DFA in order to start the conversion procedure.

4.1.1 *Algorithm*

The algorithm is rather straight forward. Starting from an start state, we have to find its epsilon closure, the epsilon closure is a set with respect to a state that includes all the sets to which we may travel from the mentioned state by only moving along the epsilon edges. Then we take a look at the outgoing edges from the epsilon-closure set and complete the state transition matrix of the DFA, by putting the union of outgoing states for a specific symbol in the place of next state in DFA state transition matrix. Finally, we build the set of final states to any state that contains an NFA final state in its state set. To make the case of the algorithm more concrete, we present the main algorithm here.

Suppose there is an NFA $N < Q, \sum, q0, \sigma, F >$ which recognizes a language L. Then the DFA $D < Q', \sum, q0, \sigma, F' >$ can be constructed for language L as:

1. Initially $Q' = \emptyset$
2. Add qo to Q'.

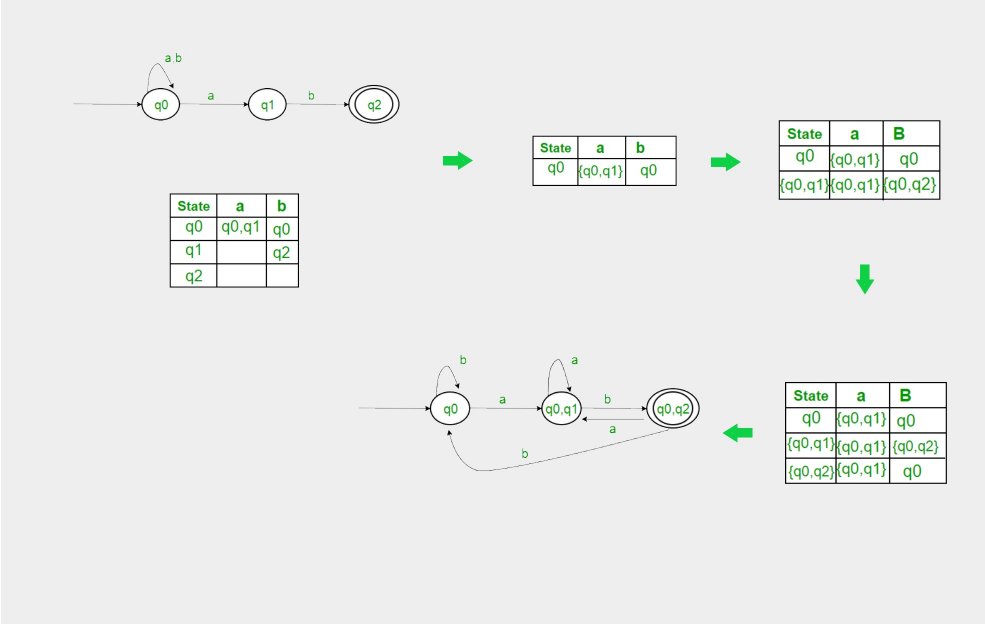


Figure 5: NFA → DFA detailed procedure

3. For each state in Q', find the possible set of states for each input symbol using transition function of NFA. If this set of states is not in Q', add it to Q'.
4. Final state of DFA will be all states with contain F. (final states of NFA)

A simple example of how the algorithm works is given in Figure 5.

4.1.2 *Implementation*

The implementation is, as stated before, started with the function `buildDFA`. In this function we first load the state transition matrix of the input NFA, and initialize an empty state transition matrix for our DFA object.

ADDING THE ALPHABET After the initialization, we add the NFA alphabet to DFA using the function `addNewColumn()`, except for the epsilon sign. The alphabet is added by just simply copying the first column of NFA state transition matrix into the DFA's.

FINDING EPSILON CLOSURES Finding the epsilon closures is not an easy task, i.e. it requires to find a set connected to a specific state via epsilon edges, and then also finding the same thing for each of these states. This yields a recursive function, implemented as `getEpsilonClosure()` to recursively find the epsilon closure set. To avoid getting stuck in a loop we have to keep track of seen states in a separate array.

FILLING THE STATE TRANSITION MATRIX Next we iterate over the columns of NFA state transition matrix. We keep the index of the last checked column in the variable lastCheckedColumn to help us investigate the columns one by one. During the investigation, whenever we reach a new state we keep its record and later add it as a new column to the DFA state transition matrix. The function fillColumnFromNFA(lastCheckedColumn) performs the union operation on the ongoing states and also updates the epsilon closure for each of the states. The following code gets the most of what is said in this paragraph.

```
/** See where the last state added goes, fill its column,
* add the new states to state transition matrix as well.
*/
int lastCheckedColumn = 0;

/** Continue until there is no new state. */
while (lastCheckedColumn < this.stmax.size() - 1) {

  /** Increase the last checked number. */
  lastCheckedColumn++;

  /** Fill the current column. */
  this.fillColumnFromNFA(lastCheckedColumn);

  /** Check whether a new state is found and add it. */
  this.addNewStates(lastCheckedColumn);
}
```

SIMPLIFYING THE MATRIX After what we performed in the previous code, we now rename the states from the complicated state names like [1,2,3] to simple integers. This state is not necessary but compensates to the reduction of complexity caused by the previous algorithms. We also sort the state numbers for more convenience.

4.1.3 *Results*

In this subsection, we present the results of conversion from NFA to DFA. We are aware that this conversion is only the first step to reach the minimum DFA but the results from this stage will act as a strong ground for the next stage's implementation. The results for a few cases are shown in the next section's result tab, alongside the minimum DFA results.

4.2 DFA to Minimum DFA

The term minimization of a DFA refers to converting a given DFA to its equivalent DFA with minimum number of states. To this extent, we use an algorithm to perform the conversion. The algorithm tries to find some partitions which are distinguishable from one another and mark them as correct states. The whole procedure starts with the function $makeMin()$, an attribute of a DFA class. This procedure, same as other operations, is based on manipulation of the state transition matrix to reach a minimized one.

4.2.1 *Algorithm*

The algorithm is mainly based on partitioning. A useful intuition to understand the way it works is to consider the case in which two states become indistinguishable. This happens when they have the same output for every symbol in the alphabet, and by the same output we mean that the output lies in the same partition. If for the first partitioning, we consider final and non-final states and loop over the partitioning procedure using the distinguishable states in each partition, we would finally reach the desired results. Hence, the formal algorithm is as the following.

Suppose there is a DFA $D < Q, \sum, q0, \sigma, F >$ which recognizes a language L. Then the minimized DFA $D < Q', \sum, q0, \sigma', F' >$ can be constructed for language L as:

1. We will divide Q (set of states) into two sets. One set will contain all final states and other set will contain non-final states. This partition is called P_0.
2. Initialize $k = 1$.
3. Find P_k by partitioning the different sets of P_{k-1}. In each set of P_{k-1}, we will take all possible pair of states. If two states of a set are distinguishable, we will split the sets into different sets in P_k.
4. Stop when $P_k = P_{k-1}$. (No change in partition)
5. All states of one set are merged into one. No. of states in minimized DFA will be equal to no. of sets in P_k.

A simple example of how the algorithm works is given in Figure 6 on the following page.

4.2.2 *Implementation*

The implementation is a bit complicated. To simplify the explanation we start from the base function and explain the big picture of implementation gradually.

DFA STATE	Min-DFA STATE	TYPE	a	a,b	b
{A,C}	1		2		1
{B}	2		2		3
{D}	3		2		4
{E,F,G,H,I}	4	accept		4	

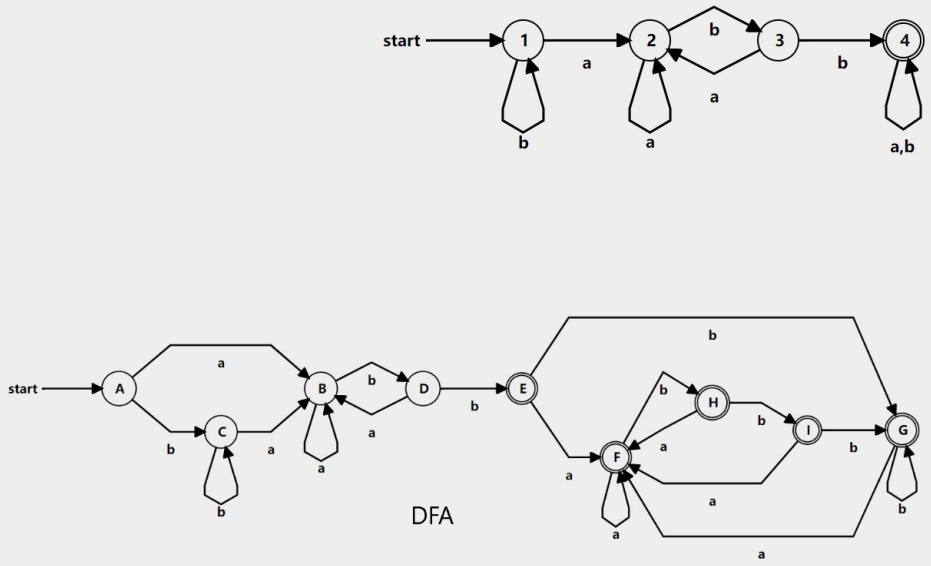


Figure 6: DFA → Minimum DFA detailed procedure

4.2.3 *Results*

5 CONTEXT FREE GRAMMAR TO PDA

A context free grammar is a grammar which satisfies certain properties. These properties and rules are given true a set of rules called the production rules. A grammar of this kind builds the language also known as formal language. Grammars work by proposing transformations so that we can make to construct a string in the language described by a grammar. Grammars will describe how to convert a start symbol (usually S) into some string of symbols.

5.0.1 *Algorithm*

Every CFG can be converted to an equivalent PDA. The constructed PDA will perform a leftmost derivation of any string accepted by the CFG. Here, and based on the algorithm mentioned in the text book, we convert a CFG to a PDA as mentioned in the following algorithm. However, we remove the two start and end states as here we have an

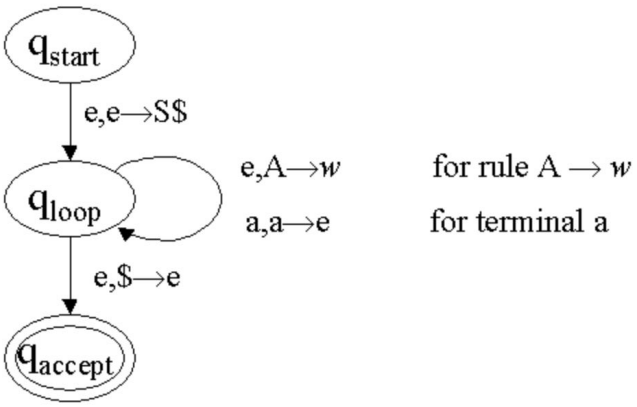


Figure 7: CFG → PDA state diagram

stack initializer and we can also check whether the stack gets empty or not. That said, we now illustrate the algorithm behind the scene.

1. Create three states, start, and end.
2. Fill the stack with the special character $ and the start variable. namely S.
3. As for the loop state, for reach rule in the form of $A \to \omega$ which A is a variable, pop A and push ω. For each terminal, also create the action $a \to \epsilon$ in the loop state.
4. Create a transition from loop to the end state when seeing the special character, $.

An state diagram of the algorithm is shown in the Figure 7.

WARNING The initial state of the stack is set to $. Hence, we do not need to create a transition to put the character inside the stack, based on the discussions previously made about the subject in the class.

WARNING The implementation is not unique. This is crucial to understand in order to evaluate the design and results.

5.0.2 *Implementation*

5.0.3 *Results*