DAAR: Développement des Algorithmes d'Application Réticulaire

Project 1: Automaton. Cloning egrep command supporting simplified ERE.

Ufuk Bombar
Elif Ohri

INTRODUCTION

A **regular expression** is the formalization of an algebraic language that is acceptable by a deterministic finite automaton (DFA), in the sense of formal language theory. They are powerful, flexible and defined patterns that are used for pattern matching, locating or validating specific strings or patterns of text in a sentence, document, or any other character input.

Regular expressions can be used to define regular languages and it provides a human-readable way to represent these languages. A regular language can be recognized and generated by finite automata. Similarly, the language accepted by finite automata can be easily described by regular expressions. It is the most effective way to represent any language. However, designing deterministic automata can be hard and it is usually easier to design a non-deterministic automata. Methods such as subset construction, allows us to turn a non-deterministic automaton into a deterministic automaton.

In this project we implemented the conversion of a regular expression into a finite state automaton. The following states are implemented respectively:

- Parse the regular expression into a syntax tree:
    - Tokenize the regular expression and parse the input regex.
    - Convert the regex into a syntax tree representation.
- Convert syntax tree into a Non - Deterministic Finite Automaton (NDFA):
    - Create a class for NDFA states and transitions.
    - Traverse the syntax tree to build an NDFA, keeping track of states and transitions.
    - Handle concatenation (“.”), alternation (“|”), plus(“+”) and closure (“*”) operations.
- Convert the NDFA into a non-epsilon transition NDFA
- Implement Regex Matching:
    - Use the DFA to match the regex pattern with the text from the file.

DATA STRUCTURES

Regex to NDFA: During the conversion from a regex to an NDFA, some types of data structures are used. There are linear ones which are also known as the basic ones. They are named lists, and sets. Python's default library provides implementations of those structures. We have used the lists and sets when we are tokenizing the raw input string. Additionally, to remove any ambiguity we needed to represent the regular expression in a syntax tree or also known as a parse tree. Each node in the syntax tree represents a subexpression, and the tree's structure reflects the hierarchy and precedence of operations in the regex. The tree can be traversed to build the corresponding NDFA. We did our implementation of a syntax tree node as a python class with left, right nodes and the data field.

NDFA Representation: The core data structure used in this part of the project to represent the NDFA is a graph. The nodes of the graph represent states in the NFA, and the edges represent transitions between states based on input characters or ε (epsilon) transitions. To store the starting state, accepting states and the transitions, the object NDFA is used. The transitions between the states are stored in a dictionary structure in the NDFA object.

IMPLEMENTATION

- **Parse the regular expression into a syntax tree:**

To be able to convert the regular expression into a syntax tree, we first have to tokenize the regular expression by following the below steps:

1. The regular expression is converted into a list of characters where every individual character is added as an element to the list
2. An object named **“Token”** is created. This object has two components: **“type”** and **“data”.** The component **“type”** indicates the type of the character. The values that it can take are as follows:

   ```
   CHAR = 0        # characters
   CONCAT = 1      # concatenation
   OR = 2          # alternation
   RP = 3          # right parenthesis
   LP = 4          # left parenthesis
   STAR = 5        # Kleene star (closure)
   ANY = 6         # any characters
   PLUS = 7        # plus (closure without the epsilon transition)
   ```

   The **“data”** component is for storing the actual character as a string. After every character is stored in a Token object, they are stored in a list. At the end, every element of the character list becomes a Token object in a new list called “tokenized_list”..

3. The list of Token objects are then optimized by storing all concatenated ASCII characters in one Token object. This is done by storing all the ASCII characters, that are concatenated, as a string of characters, paying attention to the precedence of the operators.

4. For the last step of the tokenization of the regular expression, we add a concatenation token when two strings are concatenated in the regular expression to be able to build the syntax tree.
5. An example of the tokenization step can be seen in the example below:

1st step: The regular expression is: “S(aaaa|gg|r)+on”

2nd step: Regular expression is turned into a list of characters: [“S(aaaa|gg|r)+on']

3rd step: The character list is converted into a list of tokens: [CHAR(S), LP, CHAR(a), CHAR(a), CHAR(a), CHAR(a), OR, CHAR(g), CHAR(g), OR, CHAR(r), RP, PLUS, CHAR(o), CHAR(n)]

4rd step: The optimization of the token list is done by storing concatenated characters in one token: [CHAR(S), LP, CHAR(aaaa), OR, CHAR(gg), OR, CHAR(r), RP, PLUS, CHAR(on)]

5th step: Concatenation token is added for the concatenation of characters and other operators: [CHAR(S), CONCAT, LP, CHAR(aaaa), OR, CHAR(gg), OR, CHAR(r), RP, PLUS, CONCAT, CHAR(on)]

To convert the tokenized regular expression into a syntax tree, we convert the tokenized regular expression that has been written in infix notation to postfix notation, then we convert this postfix notation into a tree structure.

To convert the infix notation to postfix notation, we use the **Shunting Yard Algorithm**. Shunting Yard Algorithm is a method for parsing mathematical expressions that are written as infix notation and converting them into postfix notation. In this project, we use the Shunting Yard Algorithm for converting regular expressions instead of mathematical operations. We transform the algorithm so that it can process the regular expression while considering operator precedence, including unary operators.

Here is how the ***regex_to_postifx()*** actually works:

1. First we indicate the precedence of the operators in a dictionary structure.
2. We start iterating over the list. When the function encounters an **alphanumeric character** or a **dot “.”**, it appends it directly to the **output** list, as these symbols don't require any special handling.
3. When it encounters an **opening parenthesis “(“**, it pushes it onto the **stack** list.
4. When it encounters a **closing parenthesis “)”**, it pops operators from the **stack** list and appends them to the **output** list until it encounters the matching opening parenthesis “(“. This step ensures that expressions within parentheses are properly evaluated.
5. When it encounters an **operator (“|”,”+”,”*”)**, it determines the precedence of the operator. Unary operators, like the Kleene star ('*'), have the highest precedence.
    - It checks whether the current operator is unary by looking at the character just before it in the regular expression. If the previous character is an alphanumeric character or a closing parenthesis, it treats the current operator as binary.
    - If the current operator is a unary operator, it appends it directly to the **output** list.
    - If the current operator is binary, it pops operators from the **stack** and appends them to the **output** list until it finds an operator with lower

precedence (or an opening parenthesis) at the top of the **stack**. Then, it pushes the current operator onto the **stack**.

6. After processing all characters in the regular expression, the function may have some operators remaining on the **stack**. It pops these operators and appends them to the **output** list to ensure that all operators are processed correctly.

This ***regex_to_postifx()*** function takes a regular expression in infix notation as an argument and returns the postfix notation of this regular expression.

6th step: Postfix notation of the regular expression: [CHAR(S), CHAR(aaaa), CHAR(gg), OR, CHAR(r), OR, PLUS, CONCAT, CHAR(on), CONCAT]

After the conversion to the postfix notation, the syntax tree can be easily built with the ***build_expression_tree()*** function. For the syntax tree, an object named **"SyntaxTreeNode"** object is created. This object has a node and left and right children. ***build_expression_tree()*** function reads the postfix notation and builds the syntax tree.

7th step: Syntax tree notation of the regular expression:

```
'CONCAT'
        'CONCAT'
                'CHAR(S)'
                'PLUS'
                        'OR'
                                'OR'
                                        'CHAR(aaaa)'
                                        'CHAR(gg)'
                                'CHAR(r)'
        'CHAR(on)'
```

- **Convert syntax tree into a Nondeterministic Finite Automaton (NFA):**

The ***generate_ndfa()*** function is responsible for converting a syntax tree representing a regular expression into a Non-Deterministic Finite Automaton (NDFA). The NDFA is constructed by recursively traversing the syntax tree and building the NDFA components for each node.

To do that, we created an **"NDFA"** object. This object has four components which store the following information about an **NDFA**: a starting state, a set structure for accepting states, a dictionary storing the transitions of the state and a last_generated_state variable for giving an id to each state.

generate_ndfa() function takes a SyntaxTreeNode which has a node and left and right children. Function iterates over the nodes and the leaves of the syntax tree. Here is how the function ***generate_ndfa()*** actually works:

1. Function takes the top node of a syntax tree and iterates over its children. It checks if the node is a leaf or a node.
2. If the node is a leaf, then it checks for the value of the node. The value of the node can be either a dot "." or any alphanumeric character.
    a. If the value is a dot "." then the function returns the ***new_symbol_matcher()*** function with an argument of "."
    b. If the value is anything else, then the function returns the ***new_symbol_matcher()*** function with an argument of the value of the node.
3. If the node is not a leaf, it checks if the node is a unary operator or a binary operator. If the node has both children then it is a binary operator.
    a. If the binary operator is an **alternation** operator, then the function calls ***generate_ndfa()*** function for both of its children and returns ***new_or_ndfa()*** function.
    b. If the binary operator is a concatenation operator, then the function calls ***generate_ndfa()*** function for both of its children and returns ***new_concat_ndfa()*** function.
4. If the node has only one left child then it is a unary operator.
    a. If the binary operator is a **closure** operator, then the function calls ***generate_ndfa()*** function for its left children and returns ***new_star_ndfa()*** function.
    b. If the binary operator is a **plus** operator, then the function calls ***generate_ndfa()*** function for its left children and returns ***new_plus_ndfa()*** function.

new_symbol_matcher() function creates a new NDFA object and calls the ***new_state()*** function which creates a state and returns the id of that state. This state is added to the **accepted_states** set of the NDFA and then the **add_transitions()** function is called.

add_transition() function takes the id of the from_state, the id of the to_state and the symbol for this transition. It adds this transition to the transition dictionary of the NDFA.

add_ndfa() function adds two NDFA with an epsilon transition by connecting the from_state to the starting state of the child NDFA and connecting the accepting states of the child NDFA to the to_state and. Then it calls the ***merge_ndfa()*** to merge the NDFAs.

The algorithm behind the conversion of each operator case into an NDFA is made by the following four functions: ***new_or_ndfa(), new_plus_ndfa(), new_star_ndfa()*** and ***new_concat_ndfa().***

Finally the ***generate_ndfa()*** algorithm returns the final NDFA object.

- **Convert the NDFA into a non-epsilon transition NDFA**

After generating the NDFA object, the epsilon transition states are removed from the graph. To do that two new functions are added: **create_non_epsilon_ndfa()** and **delete_unreachable_states().**

create_non_epsilon_ndfa() function copies the states of the old NDFA to a new NDFA without any transitions/accepting states. Then, by doing a BFS algorithm on the NDFA state, **epsilon_closure()** function iterates over the epsilon transition states.All the epsilon-transition states are visited until it sees a non-epsilon transition. Whenever it sees a non-epsilon transition, **input_closure()** function iterates to the other state with the transition input.

delete_unreachable_states() function deletes the states that have no incoming transitions towards them except the starting state.

- **Implement Regex Matching**

After deleting the epsilon transitions from the NDFA, the code is tested. match_line() function tries to match the input with the line. If there is a match, it returns True; if there is not then it returns False.

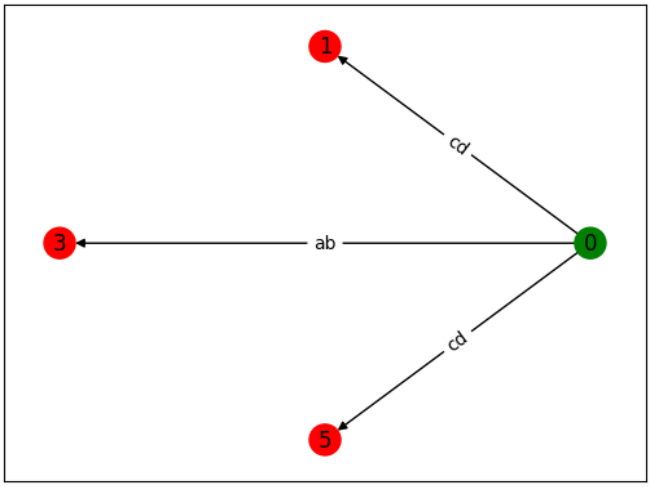

Figure 1: Example non-optimized NDFA of the regex “ab|cd”. As seen there are duplicate nodes.

PERFORMANCE TEST

Table 1. Performance test results compared to `egrep` implementation Tested on M2 mac.

RegEx	Our implementation	egrep method
'(t.h)|(T.h)'	**0.459s**	**1.319s**
''	**0.021s**	**0.023s**
' '	**0.020s**	**0.002s**
'a|b|c|d|e'	**0.723s**	**0.006s**
'.'	**0.164s**	**0.004s**
'..'	**0.661s**	**0.005s**
'...'	**2.426s**	**0.005s**
'....'	**9.117s**	**0.005s**
'(0|1|3|4)+'	**1.499s**	**0.026s**

Test results for the egrep command have clearly better results than our implementation. The main reasons may be:

- The epsilon transitions are deleted from the final NDFA however it is not converted into a DFA in the end. This results in a lot of duplùcated (mirrored) states. Since they are not concatenated and optimized, this results in the current set of states to grow near exponentially.
- Python implementation of the dictionaries are not optimized to store integers. There can be more optimized implementations additionally Python's garbage collector also reduces the performance