DAAR project 1

Reda BERRADA, Sofiane CHIKH BLED

November 2020

Contents

1. Problem definition

In this project we are interested in implementing a solution that searches for a regular expression in a text. Regular expression is a sequence of characters that define a search pattern.
We will look at the data structures and algorithms that need to be implemented, then, we will do some tests with different parameters and finally we will discuss the performance test results that this solution give.

2. Implementation

To implement a solution for this problem, a process needs to be defined alongside with some data structures.

1 The process

The process to get the optimized automaton from a regex is as follows:

1. Build the regular expression tree for the given regular expression.
2. Given the regular expression tree we construct the corresponding NDA automaton.
3. We eliminate epsilon-transitions of that automaton.
4. We minimize the automaton to have the minimum number of states.
5. Using the minimized automaton, we search for the regular expression in the given text by going through the automaton states.

2 Data structures

2.1 RegEx

This class has multiple methods:

- parse(): parses the regular expression and returns a regular expression tree.
- buildOperandAutomaton(c): builds a single-character automaton.
- buildAlternateAutomaton(a1, a2): builds a1—a2 automaton.
- buildConcatAutomaton(a1, a2): builds a1.a2 automaton.
- buildStarAutomaton(a): builds a* automaton.
- RegExTreeToAutomaton(tree): constructs the NDA automaton of the regular expression given its tree.

2.2 RegExTree

A class that defines the regular expression tree. It is constituted of:

- A root: Can be an ASCII code of a character or a code of an operation (parenthesis, alternative, concatenation, star).
- A list of RegExTree: Holds the subtrees of the actual root.

2.3 Automaton

A class that represents an automaton. It includes the following:

- start: An integer that represents the starting state
- end: A list of integers that represent the final states
- transitions: Integer matrix that holds the transitions between states, transitions[i][j] represents the transition character from i to j
- table: ArrayList¡Integer¿ matrix, it is used to operate the epsilon-transition elimination and the minimization of the automaton. It allows us to group states together in the epsilon-transition elimination step. table[i][j] holds the list of states we transit to from the list of states i using the character j.

This class also holds methods to manipulate the automaton, the most important are:

- toTable(): converts the transition matrix from int[][] to ArrayList¡Integer¿[][].
- eliminateEpsilonTransitions(): eliminates epsilon-transitions of the automate using the epsilons-closures of each state.
- n_equivalence(): gets the n_equivalence of the automaton without epsilon-transitions (used in the minimization step).
- minimize(): minimizes the automaton after the epsilon-transition elimination using the n_equivalence.
- search(text): searches for the actual regular expression in the text given the parameter using the minimized automaton.

3 Algorithms

3.1 Building the regular expression tree

We used the one provided in the boiler code and will not be discussed here to save space.

3.2 Building the NDA automaton from the regular expression tree

This is done by going through the tree nodes recursively from the bottom to the root and constructing the sub-trees automatons until we get to the root automaton which is the final one.
The recursive algorithm makes it a lot easier to understand and to implement in few lines of code.

3.3 Eliminating epsilon-transitions [Academy (2017a)]

This done by going through all the states and calculating the epsilon-closure of each state. The epsilon-closure of a state is the set of states which the actual state can reach with only epsilon-transitions (including the state itself).
Then we add the non-epsilon transitions of each state of the epsilon-closure to the transitions of that state.
We have as a result an equivalent epsilon-free automaton. Final states are the states which the epsilon-closure contains an old final state

3.4 Minimizing the automaton [Academy (2017b)]

This is done building the n_equivalence of the automaton as follows:

1. The 1_equivalence is constituted by 2 sets of states, the final ones set and non-final ones set. We exclude the non-reachable states that result from the epsilon-transition elimination.

2. loop over the sets of the i_equivalence and construct the i+1_equivalence by keeping together the states that are equivalent. 2 states are equivalent if for each character the transit to states belonging to the same set.

3. We stop when i+1_equivalence = i_equivalence.

After that, the sets of the n_equivalence will constitute the new states of the minimized automaton, and, the new transitions are the transitions from one set to another. The initial state is the set that contains the old initial state, and the final state is the set that contains the old final states.

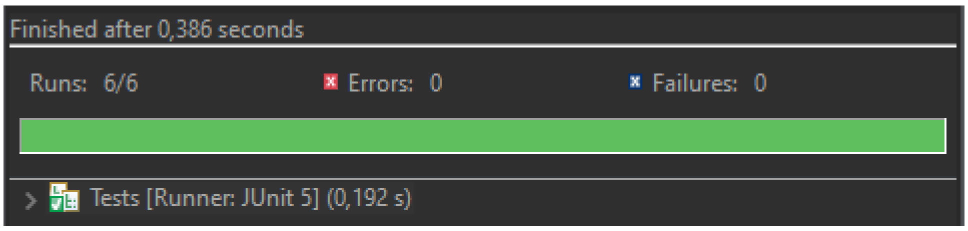
3.5 Searching in a text

We go through the text line by line and for each line we set the automaton at its initial state, then, we keep transiting from the current state to a next state if we are reading a character that allows us to do so, otherwise, we reset to the initial state and if we were at a final state we save the prefix that we just read. If we transit from an initial state we save of the column number, this is where an eventual good prefix starts.

3. Tests

The implementation is tested with *JUnit 5* on a scenario-based form, as shown in the following in the format **Test** → **Expected result** :

- Test a regex of an existing word in the text file → No error.
- Test a regex of characters without any operand → No error.
- Test a regex of a non-existing word in the text file → No error and empty result.
- Test a regex of an upper case word that only exists in a lower case form in the text file → No error and empty result.
- Test if the end states after epsilon elimination are correct → No error.
- Test a regex with allowed operands but no characters → Error.



```
Finished after 0,386 seconds
Runs: 6/6        Errors: 0        Failures: 0
Tests [Runner: JUnit 5] (0,192 s)
```

Figure 3.1: Tests' results

As illustrated in the figure above, all tests successfully passed.

4. Performance

The next step is to test the performance of the solution, and to compare it to *egrep*'s performance. To do so, the focus is on the execution time when running both methods. Since this statistic fluctuate, we decided to compute it by repeating the process 200 times and taking the mean of the obtained durations. We proceeded this way with all the following.

First, we compare the regex "S(a—r—g)*on" between the two of them in the Babylon book, as seen in the course. Here are the results :

Execution time (in ms)	
Our solution	egrep
21	85

Table 4.1: Our solution VS egrep with the example given in the course

We see that the results are very convenient. In fact, our solution seems to be 4 times faster than egrep for this regex in this book. That said, it is still not enough to jump into conclusions. That's why an interesting performance test was to compute the execution times for different books, and plot them versus their sizes. For that, we chose 20 books from the Gutenberg database (including the babylon book), and decided to use as regex "t(h—e—a)*(n—m)" because we were looking for a frequent word in the English language since all the books are in English. We obtained :

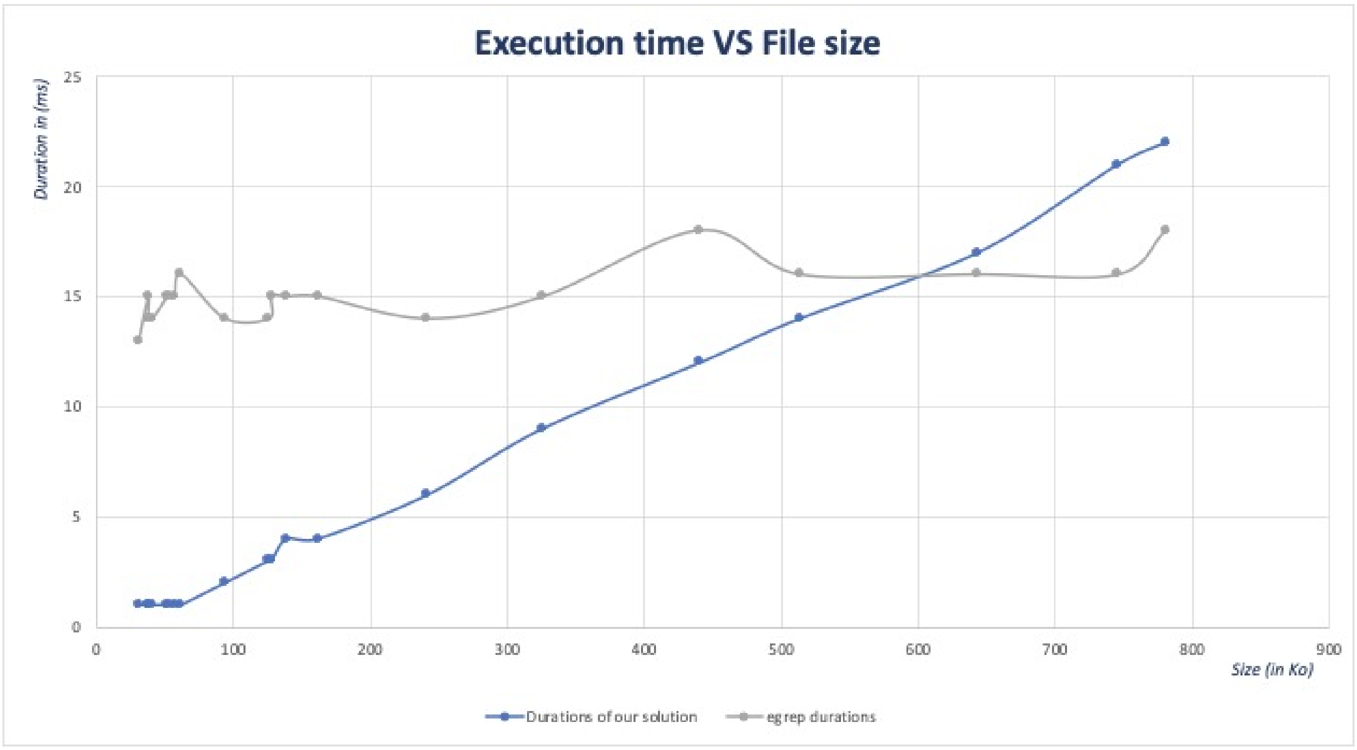


Figure 4.1: Execution time VS File size

We observe in Figure 4.1 that the execution time of our solution is linear, while egrep is quite constant. In fact, for small files, our solution seems to be more efficient, but egrep is more optimized for big files.

In addition, we constructed a bar chart that emphasizes the differences between our solution and egrep. We chose to compare the execution time for books of small, medium and large sizes. One should note that these labels are attributed relatively to our sample of 20 books, approximately 30 Ko, 326 Ko and 781 Ko respectively. We used the same regex as previously :

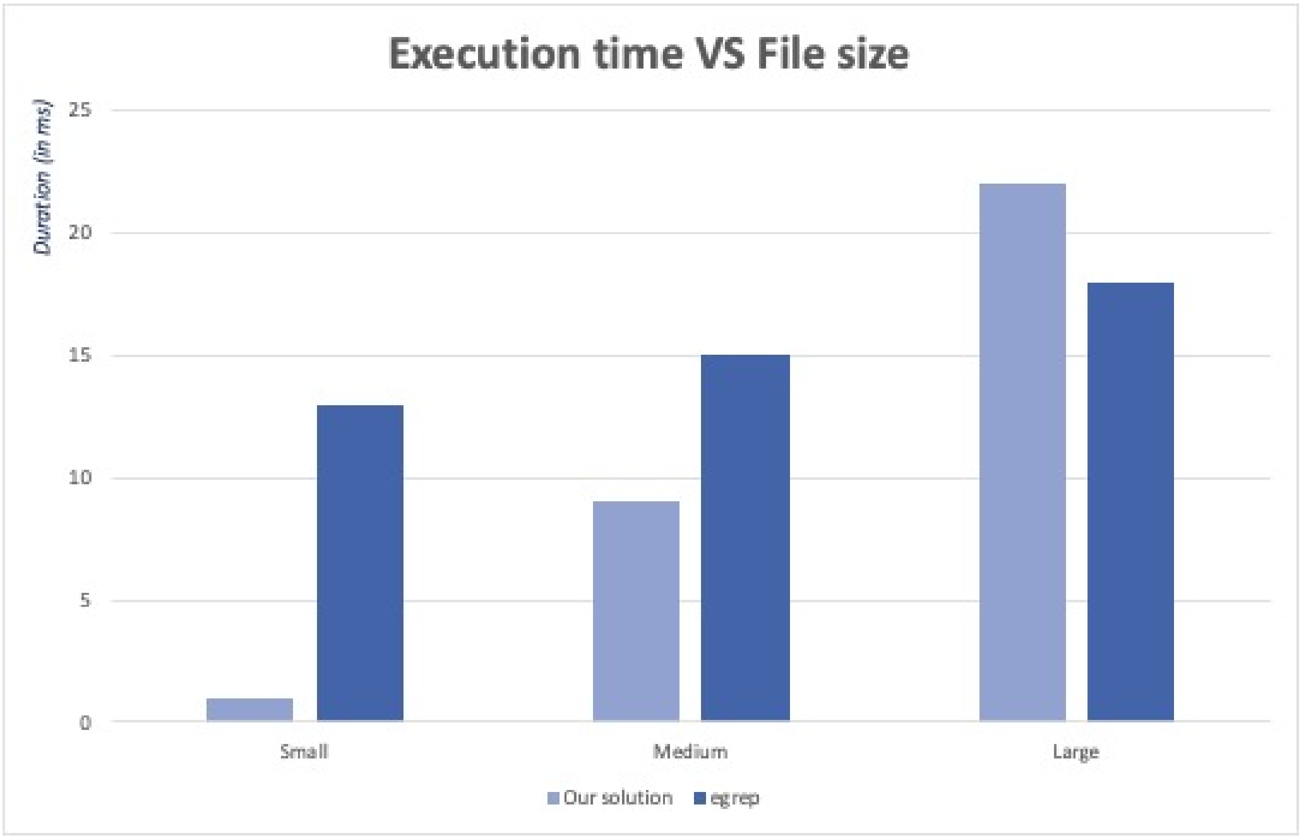


Figure 4.2: Bar chart of execution time VS File size

Figure 4.2 enhances the conclusion we made above. Our solution is way more efficient for small and medium files, and we clearly observe that the bigger the file is, the better the egrep solution gets compared to ours.

Conclusion

All in all, we managed to implement a clone of the egrep command, since we obtained an operational version that eventually gives the same result, and which on top of that is very efficient for small files. However, our solution still needs to be optimized to compete with egrep, especially for big files.

Bibliography

Neso Academy. 2017a. Conversion of Epsilon NFA to NFA. *https://www.youtube.com/watch?v=WSGcmaHNBFMab_channel=NesoAcademy* (2017).

Neso Academy. 2017b. Minimization of Deterministic Finite Automata (DFA). *https://www.youtube.com/watch?v=hOzc4BUIXRkab_channel=NesoAcademy* (2017).