Recommendation System - Report

Armela Ligori
aligori18@epoka.edu.al
Epoka University - Theory of Computation (CEN 350)

Abstract

This assignment represents a practical application of Finite Automaton in searching a series of patterns in a given data set, with the aim of constructing a Finite Automaton-based Recommendation System. For this purpose, a data set comprised of ten web-pages and a set of keywords related to a specific topic, are taken into consideration. Using python, an approach to classify these web-pages from most to least recommended is developed.

Keywords – Finite Automata, pattern matching, preprocessing, searching phase, state, occurrence

1 Introduction

In many information retrieval applications, it is necessary to be able to locate quickly some or all occurrences of user-specified patterns of words and phrases in the data set. A recommendation system, at its core, is an information retrieval application, which relies on pattern matching algorithms to learn consumers behavior and to produce outcomes that satisfy their needs. The recommendation system in this assignment, supposes the behaviour of the consumers to be already known, so the list of patterns related to the users needs is already determined. Its job is to search that specific set of keywords into a set of webpages using string matching algorithms. There are several exact pattern matching algorithms, but here I consider a Finite Automata approach to build a basic recommendation system for a chosen list of webpages.

Finite Automata A finite automaton is an abstract state machine represented by a five tuple (Q, Σ, δ, q_0, F) consisting of [1]:
1. Q: a finite set of states
2. Σ: a finite input alphabet
3. $q_0 \in Q$: a fixed element of Q called the initial state
4. F $\subset$ Q: a set of final or accepting states
5. δ: a transition function

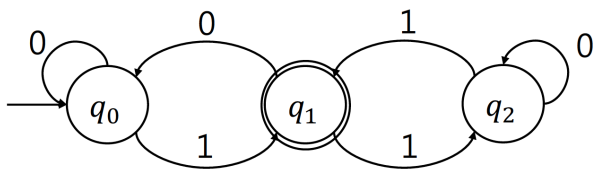


Figure 1: **Finite Automata** - Illustration of a finite state automaton

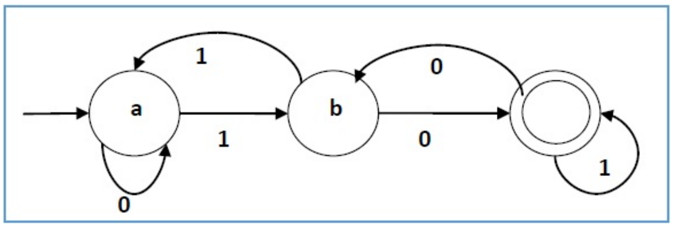


Figure 2: **Finite Automata** - Illustration of a finite state automaton

A finite state automaton successively reads each symbols of the input string, and changes its state according to a particular control mechanism. It is constructed to have a set of states and rules for moving from one state to another but it depends upon the applied input symbol [2]. If the machine, after reading the last symbol of the input string, is in one of a set of particular states, then the machine is said to accept the input string.

2 Dataset

This experiment will be limited to a dataset comprised of ten random websites related to the topic of art. These webpages will be the dynamic text in which a list of keywords (patterns) related to *art* will be searched. The text is extracted from the webpages and saved into *.txt* files, namely *file1.txt,file2.txt* ... *file10.txt*.

2.1 String-matching automata

As an online searching algorithm, string-matching automaton consists of two parts:

1. Preprocessing Phase of the pattern P
2. Searching Phase of P in dynamic text T

Preprocessing Phase For each given pattern P, a string-matching automaton is constructed in a preprocessing step before it is used to search the dynamic text. This phase consists in building the transition table representing the finite automaton for the given pattern.

- The state set Q is $\{0, 1....m\}$. Each state represents a prefix of the pattern P. The starting state q0 is state 0, and state $m = len(P)$ is the only accepting state.
- The transition function δ is computed for any state q and any character a $\in$ alphabet. It represents the longest prefix of the pattern P that has matched the text T so far [3].

- There are two cases to consider when finding the transition function:
    1. Next character in T will continue to match the pattern → Increment the state.
    2. Next character in T does not continue to match the pattern → Backtrack by finding a smaller prefix of P that is also a suffix of T.

Searching Phase
To search the text, the finite automaton will begin in state 0 and will examine every incoming character of T. Based on the current state and current character, the automaton will use the already-computed transition function to determine the next state. If next state is the final state, then the automaton is said to have accepted the string read so far.

2.2 Implementation

This implementation of String Matching Finite Automata is based on the code contributed by GeeksForGeeks [4]. In order to represent a Finite State Automaton with all its properties, I modified the code and used a class-based representation. An instance of this class is the finite automaton constructed for a given pattern. Each FA instance has as attributes: the pattern it represents, the finite set of alphabet, the finite set of all states, the accepting/final state and the transition table representing this automaton. For each instance, the pattern is preprocessed once in the moment that the instance is created and **search** function can be called with any dynamic text as an argument, and will return the number of occurrences of that specific pattern in the current text. View in Repl.it

Listing 1: Finite Automata class implementation in python

```python
class FiniteAutomata:

  def __init__(self, pattern):
    self.pattern = pattern
    self.alphabet = 256  # extended ASCII chars
    self.states = [i for i in range(len(pattern) + 1)]
    self.finalState = len(pattern)
    self.transitionTable = self.computeTable()

  def computeTable(self):
    '''Preprocess the transition Table representing this FA'''

    table = [[self.getNextState(state, char)\
              for char in range(self.alphabet)]\
              for state in self.states]
    return table

  def getNextState(self, state, ch):
    '''This function calculates the next state'''
    if state<self.finalState and ch == ord(self.pattern[state]):
      return state + 1
    i = 0
    for nextstate in range(state, 0, -1):
      if ord(self.pattern[nextstate - 1]) == ch:
        while(i < nextstate - 1):
          if self.pattern[i] != self.pattern[state-nextstate+1+i]:
            break
          i += 1
        if (i == nextstate - 1):
          return nextstate
    return 0

  def search(self, text):
    ''' Searching for occurrences of pattern P in dynamic text ↩
    T '''
    currentState = 0
    occurrences = 0
    text = list(map(self.convertToAscii, text))
    for t in text:
      try:
        currentState = self.transitionTable[currentState][t]
      except(IndexError):
        currentState = 0
      if currentState == self.finalState:
        occurrences += 1
    return occurrences

  def convertToAscii(self,t):
    return ord(t)
```

2.3 PseudoCode

```
COMPUTE-TRANSITION-FUNCTION ($P$, $\Sigma$)
$m = P.length$;
for $q = 0$ to $m$ do
    for each character $a \in \Sigma$ do
        $k$ = min($m+1, q+2$);
        repeat $k = k - 1$;
        until $P_k$ suffix $P_q$ a;
        $\delta(q,a) = k$
    end
end
return $\delta$;
```
Algorithm 1: Preprocessing Phase with Finite Automata

```
FINITE-AUTOMATON-MATCHER ($T$, $\delta$, $m$)
$n = T.length$;
$q$ = 0;
for $i = 1$ to $n$ do
    $q = \delta(q, T[i])$;
    if $q == m$ then
        print "Pattern occurs with shift i=m";
    end
end
```
Algorithm 2: String Matching with Finite Automata

Complexity
Since each character of the text is processed only once, the complexity of searching a pattern *P* in a text *T* of length *n* is $\Theta(n)$. However the preprocessing phase required to build the finite automaton takes $\Theta(m^3|\Sigma|)$ time, where m is the length of the pattern and $|\Sigma|$ is the size of the alphabet.

3 Experiment and Results

The topic of interest for this assignment is *Art*. After extracting the data from the websites, I defined a set of keywords to search in the choosen text files:

patterns = ['art','artwork','artist','aesthetic','painting']

I preprocessed each pattern once by constructing instances of *FiniteAutomata* class, and saved all the finite automata into a python list to use them later on with any text file.

```python
finite_automata = []

for pattern in patterns:
    fa = FiniteAutomata(pattern)
    finite_automata.append(fa)
```

I extracted the text from each file and converted all the characters into lowercase, so that 'Art' and 'art' are considered a match. I iterated through the list of finite automata and for each automaton, the function **search** is called to find the occurrences of the pattern that the automaton represents into the current text. With these two for-each loops, I was able to search all the patterns in all the text files.

The recommendation system should filter the pages based on relevance to the topic. To accomplish this, for each file I needed to store the results of searching and sort them according to a sorting criteria. The most recommended pages, will be the ones that have the most occurrences of the given patterns.

To store the searching results, I created a helper class called Node that has as instance properties: the filename, a dictionary where each key is the pattern and each corresponding value is the frequency of that pattern in the file, and the average frequency for the file. This average frequency will be the sorting criteria and is created only for the purpose of ordering the files.

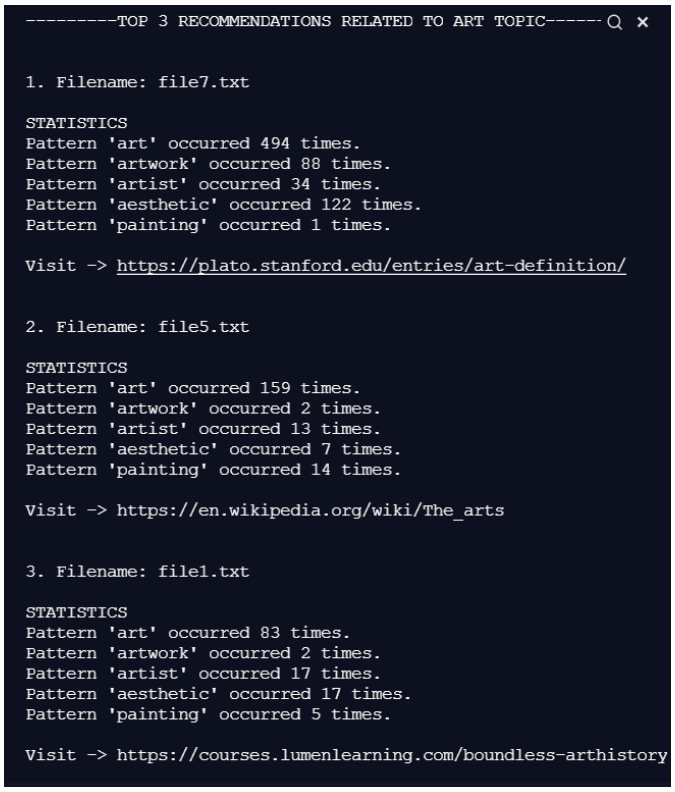
```
class Node:
  def __init__(self, filename, patternFrequencies):
    self.filename = filename
    self.patternFrequencies = patternFrequencies
    self.avgFrequency = self.calculateAvg()

  def __lt__(self, other):
    return self.avgFrequency > other.avgFrequency

  def calculateAvg(self):
    sum = 0
    for pattern, frequency in self.patternFrequencies.items():
      sum += frequency
    return sum/len(self.patternFrequencies)
```

This class is called Node because heap-sort algorithm will be used. For this, I created a Priority Queue using a Max Heap, to store each Node. A priority queue is an abstract data type similar to a regular queue in which each element has a "priority" associated with it and an element with high priority is served before an element with low priority. A Heap is a complete binary tree that gives the most efficient implementation of a priority queue. As mentioned before, the priority in this case for each file will be the average frequency.

Implementing a heap data structure is out of the scope of this assignment, therefore I am using the **heapq** built-in module in python to create the heap. When searching process finishes for a file, a node with the results of that file is created and is pushed into the heap. This *heappush* operation will maintain the heap invariant. This means that the Node with the highest avg pattern frequency will be the root node in the heap.

```
maxHeap = []

for filename in files:
  with open('websites/' + filename, 'r') as infile:
  text = infile.read().lower()
  patternsFrequencies = {}

  for fa in finite_automata:
      # Search current pattern in current text
      frequency = fa.search(text)
      patternsFrequencies[fa.pattern] = frequency

  # Create a new node for the current file
  node = Node(filename, patternsFrequencies)
  # Push the new node to the heap, maintaining the heap invariant.
  heapq.heappush(maxHeap, node)

displayRecommendations(maxHeap)
```

After all the files are searched, the final Max Heap is constructed. If *n* pop operations are performed on the heap, the result will be a descending list of Nodes. Hence, the webpages will be ordered starting from most to least recommended. I displayed only the top 3 most recommended webpages, together with the statistics for each pattern.



```
---------TOP 3 RECOMMENDATIONS RELATED TO ART TOPIC------

1. Filename: file7.txt

STATISTICS
Pattern 'art' occurred 494 times.
Pattern 'artwork' occurred 88 times.
Pattern 'artist' occurred 34 times.
Pattern 'aesthetic' occurred 122 times.
Pattern 'painting' occurred 1 times.

Visit -> https://plato.stanford.edu/entries/art-definition/


2. Filename: file5.txt

STATISTICS
Pattern 'art' occurred 159 times.
Pattern 'artwork' occurred 2 times.
Pattern 'artist' occurred 13 times.
Pattern 'aesthetic' occurred 7 times.
Pattern 'painting' occurred 14 times.

Visit -> https://en.wikipedia.org/wiki/The_arts


3. Filename: file1.txt

STATISTICS
Pattern 'art' occurred 83 times.
Pattern 'artwork' occurred 2 times.
Pattern 'artist' occurred 17 times.
Pattern 'aesthetic' occurred 17 times.
Pattern 'painting' occurred 5 times.

Visit -> https://courses.lumenlearning.com/boundless-arthistory
```

Figure 3: **Recommendation Results**

4 Conclusion

After testing the String Matching Automaton in this simple recommendation system, it is visible that the algorithm itself is very efficient, because each character in the text is examined only once. However, the program required not only one, but multiple patterns to be searched in multiple files. This increased the total complexity of the program. The number of patterns and files were limited, but assuming that we have *n* files, *n* patterns, *n* character text for each file, the total complexity for this recommendation system would be $\Theta(n^3)$. By constructing distinct finite automata for each pattern, the patterns are searched in a serial mode. If the patterns are instead searched in parallel (simultaneously), this would reduce a lot of unnecessary time, needed when rescanning the same text all over again.

References

[1] M. V. Lawson, *Finite Automata*. CRC Press, 2003.

[2] GeeksForGeeks, "Introduction of finite automata." https://www.geeksforgeeks.org/introduction-of-finite-automata/. Accessed: 2021-2-2.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.

[4] A. Kumar, "Finite automata algorithm for pattern searching." https://www.geeksforgeeks.org/finite-automata-algorithm-for-pattern-searching/. Accessed: 2021-2-2.