

Data Structures and Algorithms II

Lecture 10: Pattern Matching Algorithms

- Motivation
- Representing Patterns
- A Simple Pattern Matching Algorithm.

1

Representing Patterns: Regular Expressions

Patterns can be represented by just using two special characters:

- “|” represents alternatives.
ab|cd matches ab or cd.
(a|bc)d matches ad or bcd.
- “*” allows repetition (0 or more times).
ab* matches a, ab, abb, abbb etc.
a(bc)* matches a, abc, abcbc, etc.

Brackets may be used, as illustrate above. * has higher priority than |. Simple concatenation has priority in between, so a|bc means (a or (b followed by c)) but ab* means (a followed by (b*)).

Further characters are often used for conciseness:

- “?” matches any character at all.
a?b matches aab abb azb.
- “+” allows 1 or more repetitions.
ab+ matches ab, abb, abbb etc.

3

Pattern Matching: Motivation

For many applications we want tools to find if given string matches some criteria, e.g.,

- Find all filenames that end with .cpp
- Check whether word entered is yes, y, Yes, Y, or variant.
- Search for a line in a file containing both “data” and “abstraction”.

These can typically be done by checking if strings match a *pattern*.

We need:

- A notation to describe these patterns.
- An algorithm to check if a pattern matches a given string.

2

Regular Expressions

Given the following regular expressions, which of the example strings do you think it would match?

- (c|de)*
 1. cd
 2. ccc
 3. cdede
- (a*b|c+)
 1. b
 2. aaaa
 3. ccc
 4. ab
- ?*(ie|ei)?*
 1. ii
 2. piece
 3. sheik

4

Regular Expressions and Finite State Machines (FSMs)

- Can represent regular expressions in terms of a network of nodes and connections between them.
- These nodes represent states, and the connections represent transitions between them.
The nodes in our pattern matcher capture the state “in which a certain character in the pattern has been successfully matched”.
- The network is referred to as a finite-state machine (and has many applications in CS).
- In particular, it is a *non-deterministic* finite state machine, as it will need to have *choice* nodes. You won't be able to immediately determine which route to take in the network just by traversing the string.

5

Implementing the Machine

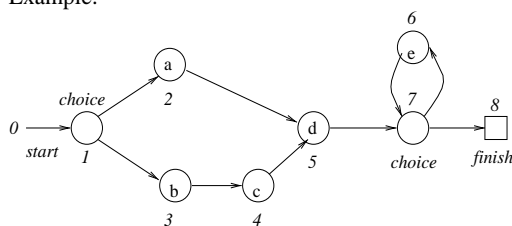
- The finite state machine (FSM) suggests a good way to represent patterns so that pattern matching algorithms can be easily implemented.
- We could represent our FSM using a general graph ADTs (discussed later). But we never allow a node to have more than two neighbours, so a simpler data structure is possible.
- Each state has one or two successor states.. so use an Nx2 array where N is number of states, and store in it the indices of successor states.
- Also need array to store characters in nodes - let content be “?” for choice nodes.

For the example FSM in earlier slide we'd have:

```
next[0][0]=1   ch[1]='?'
next[1][0]=2   ch[2]='a'
next[1][1]=3   ch[3]='b'
next[2][0]=5   etc.
```

7

Example:



(italicised bits are just for explanation or referring to it)

String matches if you can traverse network from start to finish, matching all the characters in the string. (e.g., bcdeee, ad).

6

Algorithm

We're now ready for an algorithm for pattern matching.

- We use need a data structure that allows us to keep track, as we go through the string, which characters are legal according to the pattern.
- We use a special list structure for this - it contains the possible states in the FSM corresponding to the current and next character in the string being analysed.
- This is updated as we simultaneously go through the FSM and move up in the string - based on possible state corresponding to current character in string, we can determine from FSM possible states for next character in string.

8

- A variant of the stack/queue is used as the ADT: *double ended queue* allows nodes to be put on front or end of queue: dq.put adds items to end, while dq.push adds items to start.
- Split the queue in two halves with special character
e.g., (1 2 + 5 6)
States before “+” represent possible states corresponding to current character.
States after “+” represent possible states corresponding to next character.

9

Pattern Matching Algorithm

(j=position in string)

```

dq.put('+'); j=0;
state=next[start][0];

```

While not at end of string or pattern.

- If (state=='+') {j++;
dq.put('+');}
(finished dealing with char j so move on..)
- else if (ch[state]==str[j])
dq.put(next[state][0])
(char matches one in pattern, so put next state on queue).
- else if (ch[state]=='?')
{dq.push(next[state][0]);
dq.push(next[state][1]);}
(choice node so put alternatives on front of queue.)
- state=dq.pop();
(remove state from dq)

11

Outline of Algorithm

Main step in algorithm is:

- Look at current character, and possible current state in FSM.
- If the state in the FSM is a choice node, that means the current character might correspond to either of the choices - so put them on the queue as possibilities for current character.
- If the state in the FSM contains a character matching the current character, then the next character should match the next state in the FSM - so put that next state on the (end of) the queue (as possibility for next char).

e.g.,

```

queue = (1 +)   str='ad'
  1 is a choice node
queue = (2 3 +)
  2 corresponds to 'a'
  state 2 has successor 5
queue = (3 + 5)

```

10

Example

Suppose:

```

next[0][0]=1  ch[1]=?   str='abd'
next[1][0]=2  ch[2]=a
next[1][1]=4  ch[3]=b
next[2][0]=3  ch[4]=c
next[3][0]=5  ch[5]=d
next[4][0]=5  node 6 = finish
next[5][0]=6

```

Working through algorithm we have,

dq	j	state	
(+)	0	1	
(2 4 +)	0	1	(as ch[1]='?')
(4 +)	0	2	(after dq.pop)
(4 + 3)	0	2	(as ch[2]=str[0])
(+ 3)	0	3	(after dq.pop)
(3)	0	+	(after dq.pop)
(3 +)	1	+	(state='+')
(+)	1	3	(after dq.pop)
(+ 5)	1	3	(ch[3]=str[1])
(5)	1	+	
(5 +)	2	+	
(+)	2	5	

12

(+ 6)	2	5
(6)	2	+
(6 +)	3	+
(+)	3	6

.. and we're at the end of the string and last state in the pattern, so the match succeeds.

13

Summary

- Regular expressions are equivalent to finite state machines, where each state in the machine represents a character.
- One algorithm for pattern matching involves:
 - Transforming a regular expression into an FSM, and representing this using arrays.
 - Searching the network using a search method which uses a double ended queue, and divides it so we know which states correspond to options for current character, and which for the next character.
 - This is a fairly simple and efficient algorithm for a difficult problem. Efficiency $O(MN*N)$ where M = number of states; N = length of string.

14