String Matching

String Matching
Topics

- *String matching terminology*
- *String matching applications*
- *Algorithms for pattern matching*
- *Naïve String matching algorithm*
- *Analysis of Naïve algorithm*
- *Building Finite State Automaton (FSA) for string matching*
- *Analysis of preprocessing and matching FSA procedures*

String Matching
Terminology

The string matching algorithms are used to determine occurrences of a sequence of characters in a large block of text. The string may, in general, consist of any set of symbols, such as language alphabets, decimal digits, binary digits, or arithmetic symbols, which are drawn from a finite set Σ called *alphabet*. For the purpose of matching, the text and pattern are assumed to be stored in arrays T and P of sizes m and n, where $m \leq n$.

In matching procedure, often the *beginning characters* or the *ending characters* of the pattern are used to perform comparisons. A sequence of beginning characters in a pattern P is called *prefix* of P. The prefix is denoted by the symbol $\sqsubset$ .A sequence of ending characters of P is called suffix of P. The suffix is denoted by the symbol $\sqsupset$. The prefixes and suffixes can be of different lengths. For example, if P=*computer*, *comp* $\sqsubset P$ is of length 4. Likewise, *ter* $\sqsupset P$ is of length 3.

A string of length zero is called *empty string*. It is denoted by the Greek letter ε (epsilon). The operation of joining of two strings is referred to as *concatenation*.

String Matching
Applications

String Matching is used in several applications in diverse fields. Some notable examples are as follows :

- *Text editing (spell check etc)*
- *Searching Digital Libraries (keywords and phrases)*
- *Web surfing and Web Mining*
- *Virus scanning*
- *Classifying DNA patterns*
- *Parsing for compiler construction*
- *Maintaining directories of files*
- *Processing text and XML data*

String Matching
Algorithms

In order to solve the string matching problems, a number of algorithms have been developed. A basic algorithm that makes direct character-by-character comparisons is known as *naïve or brute force*. Some more efficient algorithms *preprocess* the pattern to extract useful information which is helpful to avoid unnecessary comparisons. The important algorithms in this category are *Rabin-Karp*, *Knuth-Morris-Pratt*, and *Finite State Automaton*

The *Rabin Karp* algorithm uses hash function to transform the Pattern and the Text into strings of digits, which are compared to find matches. The *Knuth-Morris-Pratt* algorithm preprocesses the pattern to determine number of shifts of the pattern when mismatch occurs. The Finite State Automaton algorithm, preprocesses the pattern to construct an automaton, which is used to find matches in the input text. The running time of matching algorithm depends both on the preprocessing time and the time required to make actual matches. The table below summarizes the time complexities for pattern of size m and text of size n *(Ref. T Cormen et al)*.

Algorithm	Preprocessing time	Matching Time
Naïve (Brute Force)	0	$O((n-m+1)m)$
Rabin-Karp	$O(m)$	$O((n-m+1)m)$
Finite Automaton	$O(m^3 \Sigma)$	$O(n)$
Knuth-Morris-Pratt	$O(m)$	$O(n)$

Brute Force Matching

Algorithm

The Brute Force(Naïve) approach makes *character-by-character comparison* between a *pattern* and the given *text-block* to examine all matching possibilities. The algorithm consists of following steps:

Step #1*: *Align the pattern with the first text character

***Step #2*:** *Make character-by-character comparison. If match occurs print search successful*

***Step # 3*:** *Slide the pattern to next character*

***Step #4*:** *If last character of pattern moves past terminal character of text-block exit; otherwise, go to step #2*

Brute Force Matching

Example

Consider the pattern 1 0 0 1 of binary digits, which is to matched with the binary string 1 0 1 1 1 1 1 0 1 0 0 1 1 1 0 0 1 0 1 1 1. The array storing the pattern is shown in figure *(i)*. The array for the sample text is depicted in figure *(ii)*

1	0	0	1

(i) Pattern

1	0	1	1	1	1	1	0	1	0	0	1	1	1	0	0	1	0	1	1	1

(ii) Sample text

➢The steps involved in matching the pattern by using the *naïve* method are illustrated in the following diagrams

Brute Force Matching

Example

1	0	0	1																	
1	0	1	1	1	1	1	0	1	0	0	1	1	1	0	0	1	0	1	1	1

Offset: 0 Status: *No match*

	1	0	0	1																
1	0	1	1	1	1	1	0	1	0	0	1	1	1	0	0	1	0	1	1	1

Offset: 1 Status: *No match*

		1	0	0	1															
1	0	1	1	1	1	1	0	1	0	0	1	1	1	0	0	1	0	1	1	1

Offset: 2 Status: *No match*

Brute Force Matching

Example

				1	0	0	1													
1	0	1	1	1	1	1	0	1	0	0	1	1	1	0	0	1	0	1	1	1

Offset: 4 Status: *No match*

					1	0	0	1												
1	0	1	1	1	1	1	0	1	0	0	1	1	1	0	0	1	0	1	1	1

Offset: 5 Status: *No match*

						1	0	0	1											
1	0	1	1	1	1	1	0	1	0	0	1	1	1	0	0	1	0	1	1	1

Offset: 6 Status: *No match*

Brute Force Matching

Example

1	0	0	1

1	0	1	1	1	1	1	0	1	0	0	1	1	1	0	0	1	0	1	1	1

Offset: 7 Status: *No match*

1	0	0	1

1	0	1	1	1	1	1	0	1	0	0	1	1	1	0	0	1	0	1	1	1

Offset: 8 Status: *Pattern Matches*

1	0	0	1

1	0	1	1	1	1	1	0	1	0	0	1	1	1	0	0	1	0	1	1	1

Offset: 9 Status: *No match*

Brute Force Matching

Example

										1	0	0	1							
1	0	1	1	1	1	1	0	1	0	0	1	1	1	0	0	1	0	1	1	1

Offset: 10 Status: *No match*

											1	0	0	1						
1	0	1	1	1	1	1	0	1	0	0	1	1	1	0	0	1	0	1	1	1

Offset: 11 Status: *No match*

												1	0	0	1					
1	0	1	1	1	1	1	0	1	0	0	1	1	1	0	0	1	0	1	1	1

Offset: 12 Status: *No match*

Brute Force Matching

Example

													1	0	0	1				
1	0	1	1	1	1	1	0	1	0	0	1	1	1	0	0	1	0	1	1	1

Offset: 13 Status: *Pattern matches*

														1	0	0	1			
1	0	1	1	1	1	1	0	1	0	0	1	1	1	0	0	1	0	1	1	1

Offset: 14 Status: *No match*

															1	0	0	1		
1	0	1	1	1	1	1	0	1	0	0	1	1	1	0	0	1	0	1	1	1

Offset: 15 Status: *No match*

Brute Force Matching

Example

																1	0	0	1	
1	0	1	1	1	1	1	0	1	0	0	1	1	1	0	0	1	0	1	1	1

Offset: 16 Status: *No match*

																	1	0	0	1
1	0	1	1	1	1	1	0	1	0	0	1	1	1	0	0	1	0	1	1	1

Offset: 17 Status: *No match*, *End of text*

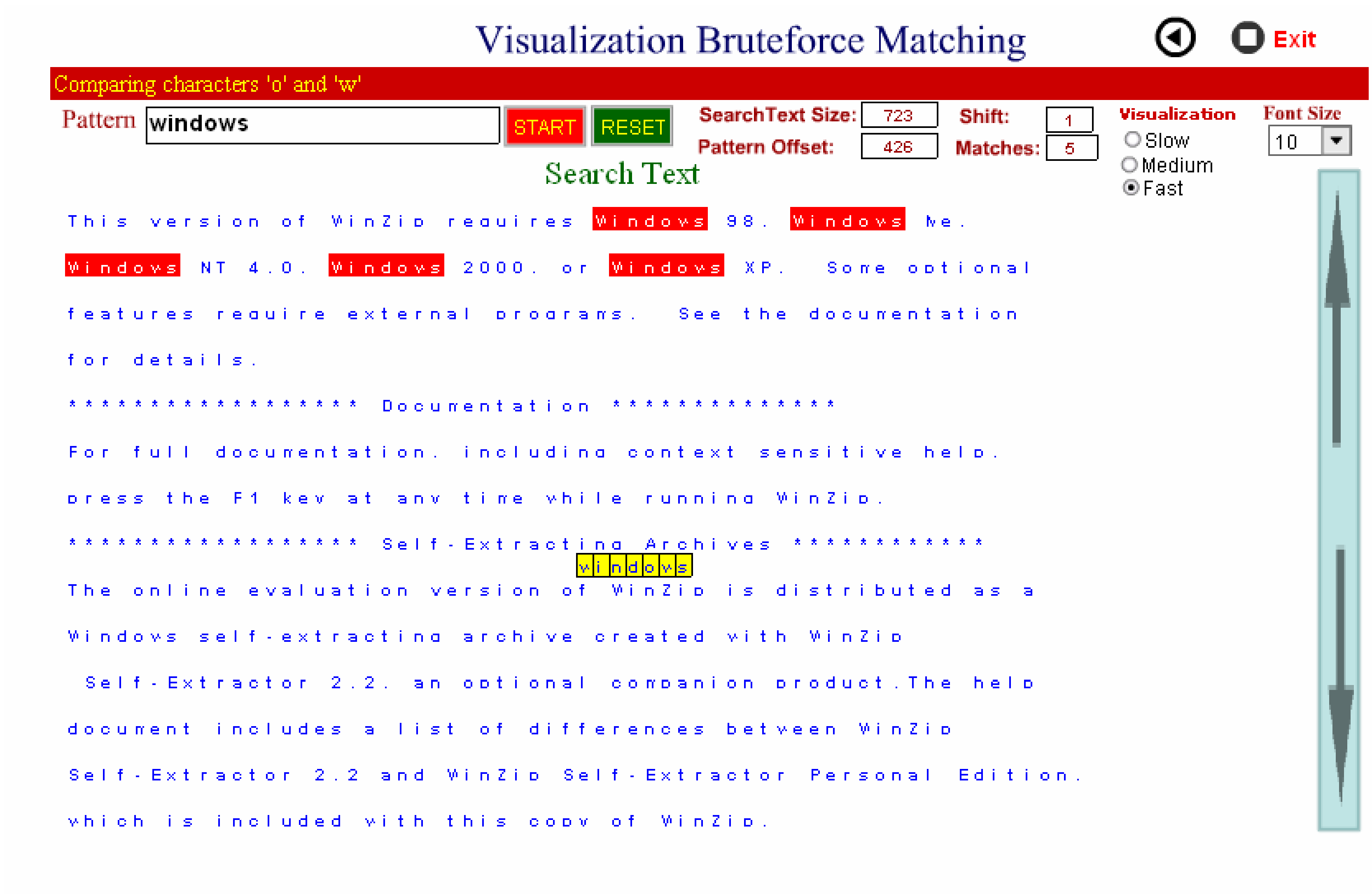
Brute Force Algorithm
Implementation

The code consists of two nested loops. The index of the inner loop accesses the characters stored in the pattern. The controlling index of the outer loop accesses the characters stored in the text. A comparison is made between a pair of characters. If all characters in the pattern match with the consecutive characters in the text, the search for pattern matching is successful, and execution terminates. The index for the text, where pattern matches , is returned. If both the loops are exhausted, and no match is found, -1 is returned

```
MATCH(P,T)
n←length[T]                    ► n stores length of text block
m ←length[P]                   ► m stores length of patter
for i ←0 to n-m  do
  for j ←1 to m do
    if  T[i+j]=P[j]
      then  if  j = m           ► All characters of pattern match
                then return i   ► i is index of text where match occurred
     else break inner loop
return -1                       ► No match is found
```

BRUTE

Visualization



Analysis of Brute Force Algorithm

Running Time

The code consists of two nested loops

for *i* $\leftarrow$ *0 to n-m* ***do***
  for *j* $\leftarrow$ *1 to m* ***do***

The outer loop iterates $n-m+1$ times. The inner loop iterates m times. Thus, in worst case, altogether $m(n-m+1)$ iterations are performed.

Thus, running time for the algorithm is $O(m(n-m+1))$, which is simplified as $O(mn)$

Finite State Automaton

Finite State Automaton

String Matching

A *Finite State Automaton (FSA)* for *string pattern matching* consists of five components, called *5-tuple* $(Q,\ q_0,\ q_t,\ \Sigma,\ \delta)$, where

1) Q is a finite *set of states* which represent all possible conditions for pattern matching

2) q_0 is the *initial* or *starting state* , where $q_0 \in Q$

3) q_t is the *terminal* or *accepting state*, where $q_t \in Q$

4) Σ is the *alphabet*, which is a collection of *symbols* used to compose the pattern

5) δ is the *transition function,* which determines the next state of FSA when a new character of the text is input to the FSA

- An FSA is *constructed for a given pattern* to be searched for in a block of text

Finite State Automaton

Alphabet (Σ)

An *alphabet* , Σ, is *set of symbols* that are used to compose the text and pattern. Depending upon the nature of application, the alphabet may consist of *characters*, *bits*, *Decimal digits*, *arithmetical symbols*, etc, as follows.

(i) {0, 1}, alphabet of bits

(ii) { 0, 1 ,2, ..9}, alphabet of decimal digits

(iii) {(+, - ,* , / }, alphabet of arithmetic symbols

(iv) {a, b, c, d}, alphabet of pattern

Finite State Automaton

Initial State

The initial state, q_0, refers to the configuration of the FSA when the string matching procedure is started. During subsequent processing, the FSA can again move back to the initial state , depending upon the current state and the nature of input alphabet.

- By convention, the initial state of FSA for string matching is the ***zeroth state***.

Thus, $q_0=0$

Finite State Automaton
Terminal State

The *terminal state* , q_t, refers to the configuration of the FSA when a given pattern successfully matches with a set of characters in the text. It is the *last state* of the FSA, and is often referred to as the *accepting state*. A string is said to be *accepted* if match occurs, and said to be *rejected* otherwise. It follows that there can be only a *single accepting state.* By contrast, in other applications of FSA there can be *multiple* terminal states.

- If a pattern consists of *m* characters, the accepting state would be $(m+1)^{st}$ state, since FSA would move into this state after successful matching. Thus, $q_t = m+1$

Finite State Automaton
Intermediates States

The *intermediate states* refer to the configuration of the FSA for partially matched patterns. The number of intermediate states equals the number of characters in the pattern. The states are numbered serially, *1 through m,* where *m* is the number of characters in the pattern.

If P is the string of characters in the pattern, the *sub-patterns* are denoted by $P_1, P_2, \ldots P_m$., that is, P_k refers to *first k characters in the pattern*. For example, if $P=abca$, $P_1=a$, $P_2=ab$, $P_3=abc$, $P_4=abca$. By convention P_0 refers to *empty string* ε $P_1, P_2, P_3, \ldots, Pm$ are also called *prefixes* of P, of lengths *1, 2, 3, .,.m* . The *FSA for P is said to be in state k* if the input string has partially matched with P_k,i.e, first *k* characters of pattern have matched. The table shows the intermediate states of FSA and partial matching of pattern

FSA State	***Partial Matching***
1	P_1
2	P_2
3	P_3
..	
m	P_m

Finite State Automaton

Transition Function

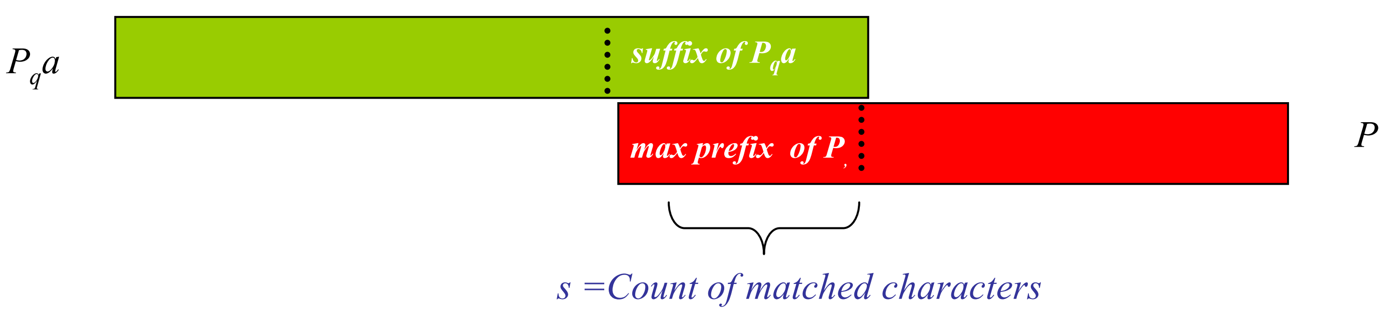
The transition function determines the dynamics of FSA for pattern matching. It returns the next state of FSA when a character is input. The next state is dependent on the current state the FSA is in, and the nature of input alphabet. Formally, the transition function $\delta(q,\alpha)$ is defined as

$$\delta(q,\alpha) = s$$

where q is the current state of FSA, and ***s*** is the new state when alphabet α is input to FSA

Suppose that FSA is currently in state q, and a symbol a in input. In state q, the partially matched sub-pattern is P_q. Thus, when a is input, we need to examine the sub-string $P_q\,a$ in order to find the next state. Further, the next state should be the one which corresponds to maximum partial matching. In other words, we need to examine the maximum prefix of P ,which matches with the *suffix* of P_qa. Since P_1, P_2, P_3….are prefixes of P, formally, the transition function is defined by the following formula, illustrated in the diagram

δ(q, a)= maximum prefix of P which matches the with suffix of P_qa = s



Computing Transition Function

Algorithm

The transition function for a given pattern P of size m can be tabulated, for all possible states ($q=0,1,..m$) and all characters in the alphabet Σ, by the following procedure:

Step #1: Consider FSA in the state q

Step #2: Select a character α from the alphabet Σ

Step #3: Select the string P_q corresponding to partial matching in state q

Step #4: *Concatenate* character α with P_q to obtain the string $P_q\,\alpha$

Step #5: Determine the *maximum size of the prefix* of pattern P that matches with the *suffix* of string $P_q\alpha$. If the count of matching characters is ***s*** then $\boldsymbol{\delta(q, \alpha) = s}$

Step #6: Repeat *Step #1* through *Step #5* for all q and all α

Computing Transition Function

Example

Consider the pattern $P = abcabca$. The table below lists the substrings that match in different states of FSA

FSA State	*Partial Match*	*Substring*
0	P_0	ε (empty)
1	P_1	*a*
2	P_2	*a b*
3	P_3	*a b c*
4	P_4	*a b c a*
5	P_5	*a b c a b*
6	P_6	*a b c a b c*
7	P_7	*a b c a b c a*

- The procedure for computing the transition table for the pattern *abcabca* is illustrated by the following diagrams

Computing Transition Function

FSA in state 0, partial match $P_0 = \varepsilon$ *(empty string)*

(1) *Input character* : **a**

$\mathbf{P_0 = \varepsilon}$ ***(empty),*** $\mathbf{P_0\, a = a}$

One *character in the prefix of pattern* P *matches with the suffix of* P_0a. *The transition function:*

$$\delta(0, a) = 1$$

P_0a

a

a b c a b c a

Character **a** *matches*

Transition Table

	a	b	c
0	1		
1			
2			
3			
4			
5			
6			
7			

(2) *Input character* : **b**

$\mathbf{P_0 = \varepsilon}$ ***(empty),*** $\mathbf{P_0\, b = b}$

No character in the prefix of pattern P *matches with the suffix of* P_0b. *The transition function:*

$$\delta(0, b) = 0$$

P_0b

b

a b c a b c a

No character matches

	a	b	c
0	1	0	
1			
2			
3			
4			
5			
6			
7			

(3) Input character : **c**

$\mathbf{P_0 = \varepsilon}$ ***(empty),*** $\mathbf{P_0\, c = c}$

No character in the prefix of pattern P *matches with the suffix of* P_0c. *The transition function:*

$$\delta(0, c) = 0$$

P_0c

c

a b c a b c a

No character matches

	a	b	c
0	1	0	0
1			
2			
3			
4			
5			
6			
7			

Computing Transition Function

FSA in state 1, partial match $P_1 = a$

(4)*Input character* : **a**

$\mathbf{P_1 = a, \quad P_1\,a = aa}$

One *character in the prefix of pattern* P *matches with the suffix of* P_1a. *The transition function:*

$$\delta(1, a) = 1$$

P_1 a

a a

a b c a b c a

Character a *matches*

Transition Table

	a	b	c
0	1	0	0
1	1		
2			
3			
4			
5			
6			
7			

(5) *Input character* : **b**

$\mathbf{P_1 = a, \quad P_1\,b = ab}$

Two *characters in the prefix of pattern* P *match with the suffix of* P_1b. *The transition function:*

$$\delta(1, b) = 2$$

P_1 b

a b

a b c a b c a

Substring ab *matches*

	a	b	c
0	1	0	0
1	1	2	
2			
3			
4			
5			
6			
7			

(6) *Input character* **c**

$\mathbf{P_1 = a, \quad P_1\,c = ac}$

No character in the prefix of pattern P *matches with the suffix of* P_1c. *The transition function:*

$$\delta(1, c) = 0$$

P_1 c

a c

a b c a b c a

No character matches

	a	b	c
0	1	0	0
1	1	2	0
2			
3			
4			
5			
6			
7			

Computing Transition Function

FSA in state 2, partial match P_2 = ab

(7) *Input character*: **a**

P_2 = ab, P_2 a =aba

One *character in the prefix of pattern* P *matches with the suffix of* P_2a. *The transition function:*

$$\delta(2, a) = 1$$

P_2 a

a b a

a b c a b c a

Character a matches

Transition Table

	a	b	c
0	1	0	0
1	1	2	0
2	1		
3			
4			
5			
6			
7			

(8) *Input character*: **b**

P_2 = ab, P_2 b =abb

No character in the prefix of pattern P *matches with the suffix of* P_2b. *The transition function:*

$$\delta(2, b) = 0$$

P_2 b

a b b

a b c a b c a

No character matches

	a	b	c
0	1	0	0
1	1	2	0
2	1	0	
3			
4			
5			
6			
7			

(9) *Input character*: **c**

P_2 = ab, P_2 c =abc

Three *characters in the prefix of pattern* P *match with the suffix of* P_2c. *The transition function:*

$$\delta(2, c) = 3$$

P_2 c

a b c

a b c a b c a

Substring **abc** *matches*

	a	b	c
0	1	0	0
1	1	2	0
2	1	0	3
3			
4			
5			
6			
7			

Computing Transition Function

FSA in state 3, partial match P_3 = abc

(10) *Input character*: **a**

P_3 = abc, P_3 a =abca

Four *characters in the prefix of pattern* P *matche with the suffix of* P_3a. *The transition function:*

δ(3, a) = 4

P_3 a

a	b	c	a			
a	b	c	a	b	c	a

Substring **abca** *matches*

Transition Table

	a	b	c
0	1	0	0
1	1	2	0
2	1	0	3
3	4		
4			
5			
6			
7			

(11) *Input character*: **b**

P_3 = abc, P_3 b =abcb

No character in the prefix of pattern P *matches with the suffix of* P_3b. *The transition function:*

δ(3, b) = 0

P_3 b

a	b	c	b						
				a	b	c	a	b	c	a

No character matches

	a	b	c
0	1	0	0
1	1	2	0
2	1	0	3
3	4	0	
4			
5			
6			
7			

(12) *Input character:* **c**

P_3 = abc, P_3 c =abcc

No character in the prefix of pattern P *matches with the suffix of* P_3c. *The transition function:*

δ(3, c) = 0

P_3 c

a	b	c	c							
				a	b	c	a	b	c	a

No character matches

	a	b	c
0	1	0	0
1	1	2	0
2	1	0	3
3	4	0	0
4			
5			
6			
7			

Computing Transition Function

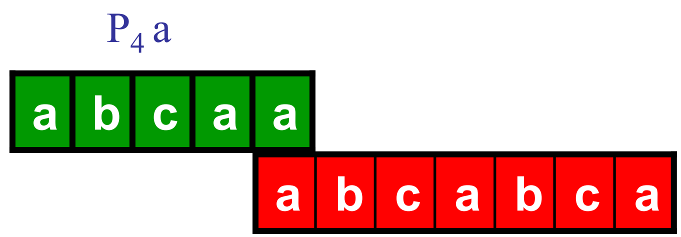
FSA in state 4, partial match P_4 = abca

(13) *Input character*: **a**

$\mathbf{P_4}$ **= abca,** $\mathbf{P_4}$ **a =abcaa**

One *character in the prefix of pattern* P *matches with the suffix of* P_4a. *The transition function:*

$$\delta(4, a) = 1$$



Character **a** *matches*

Transition Table

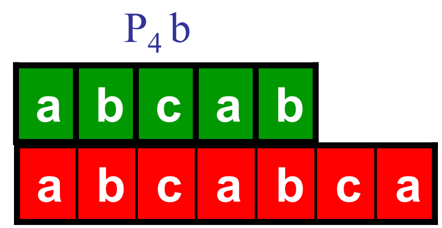
	a	b	c
0	1	0	0
1	1	2	0
2	1	0	3
3	4	0	0
4	1		
5			
6			
7			

(14) *Input character*: **b**

$\mathbf{P_4}$ **= abca,** $\mathbf{P_4}$ **b =abcab**

Five *characters in the prefix of pattern* P *match with the suffix of* P_4b. *The transition function:*

$$\delta(4, b) = 5$$



Substring **abcab** *matches*

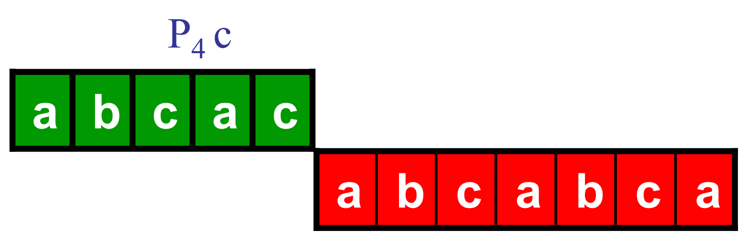
	a	b	c
0	1	0	0
1	1	2	0
2	1	0	3
3	4	0	0
4	1	5	
5			
6			
7			

(15) *Input character*: **c**

$\mathbf{P_4}$ **= abca,** $\mathbf{P_4}$ **c =abcac**

No character in the prefix of pattern P *matches with the suffix of* P_4c. *The transition function:*

$$\delta(4, c) = 0$$



No character matches

	a	b	c
0	1	0	0
1	1	2	0
2	1	0	3
3	4	0	0
4	1	5	0
5			
6			
7			

Computing Transition Function

FSA in state 5, partial match P_5 = abcab

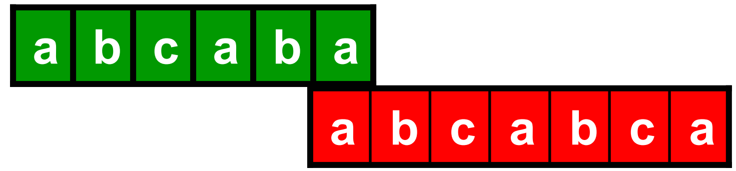
(16) *Input character*: **a**

P_5 = abcab, P_5 a =abcaba

One *character in the prefix of pattern* P *matches with the suffix of* P_5a. *The transition function:*

$$\boldsymbol{\delta(5, a) = 1}$$

P_5 a



Character **a** *matches*

Transition Table

	a	b	c
0	1	0	0
1	1	2	0
2	1	0	3
3	4	0	0
4	1	5	0
5	1		
6			
7			

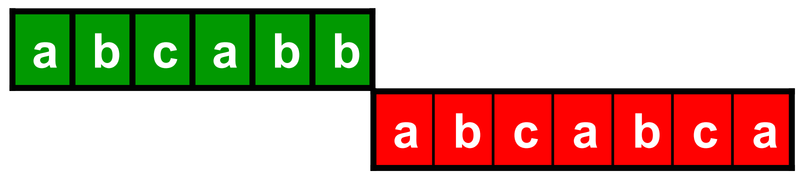
(17) *Input character*: **b**

P_5 = abcab, P_5 b =abcabb

No *character in the prefix of pattern* P *matches with the suffix of* P_5b. *The transition function:*

$$\boldsymbol{\delta(5, b) = 0}$$

P_5 b



No character matches

	a	b	c
0	1	0	0
1	1	2	0
2	1	0	3
3	4	0	0
4	1	5	0
5	1	0	
6			
7			

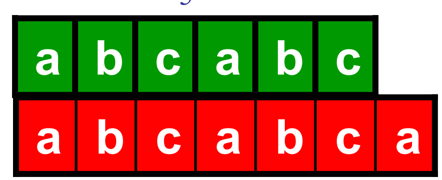
(18) *Input character*: **c**

P_5 = abcab, P_5 c =abcabc

Six *characters in the prefix of pattern* P *match with the suffix of* P_5c. *The transition function:*

$$\boldsymbol{\delta(5, c) = 6}$$

P_5 c



Substring **abcabc** *matches*

	a	b	c
0	1	0	0
1	1	2	0
2	1	0	3
3	4	0	0
4	1	5	0
5	1	0	6
6			
7			

Computing Transition Function

FSA in state 6, partial match P_6 = abcabc

(19) *Input character*: **a**

P_6 = abcabc, P_6 a =abcabca

Seven *characters in the prefix of pattern* P *match with the suffix of* P_6a. *The transition function:*

$\delta(6, a) = 7$

P_6 a

a	b	c	a	b	c	a
a	b	c	a	b	c	a

Substring **abcabca** *matches*

Transition Table

	a	b	c
0	1	0	0
1	1	2	0
2	1	0	3
3	4	0	0
4	1	5	0
5	1	0	6
6	7		
7			

(20) *Input character*: **b**

P_6 = abcabc, P_6 b =abcabcb

No *character in the prefix of pattern* P *matches with the suffix of* P_6b. *The transition function:*

$\delta(6, b) = 0$

P_6 b

a	b	c	a	b	c	b							
							a	b	c	a	b	c	a

No character matches

	a	b	c
0	1	0	0
1	1	2	0
2	1	0	3
3	4	0	0
4	1	5	0
5	1	0	6
6	7	0	
7			

(21) *Input character*: **c**

P_6 = abcabc, P_6 c =abcabcc

No character in the prefix of pattern P *matches with the suffix of* P_6c. *The transition function:*

$\delta(6, c) = 0$

P_6 c

a	b	c	a	b	c	c							
							a	b	c	a	b	c	a

No character matches

	a	b	c
0	1	0	0
1	1	2	0
2	1	0	3
3	4	0	0
4	1	5	0
5	1	0	6
6	7	0	0
7			

Computing Transition Function

FSA in state 7, partial match P_7 = abcabca

(22) *Input character*: **a**

P_7 = abcabca, P_7 a = abcabcaa

One *character in the prefix of pattern* P *matches with the suffix of* P_7a. *The transition function:*

$$\delta(7, a) = 1$$

P_7 a

a b c a b c a a

a b c a b c a

Character **a** *matches*

Transition Table

	a	b	c
0	1	0	0
1	1	2	0
2	1	0	3
3	4	0	0
4	1	5	0
5	1	0	6
6	7	0	0
7	1		

(23) *Input character*: **b**

P_7 = abcabca, P_7 b = abcabcab

Five *characters in the prefix of pattern* P *match with the suffix of* P_7b. *The transition function:*

$$\delta(7, b) = 5$$

P_7 b

a b c a b c a b

a b c a b c a

Substring **abcab** *matches*

	a	b	c
0	1	0	0
1	1	2	0
2	1	0	3
3	4	0	0
4	1	5	0
5	1	0	6
6	7	0	0
7	1	5	

(24) *Input character*: **c**

P_7 = abcabca, P_7 c = abcabcac

No character in the prefix of pattern P *matches with the suffix of* P_7c. *The transition function:*

$$\delta(7, c) = 0$$

P_7 c

a b c a b c a c

a b c a b c a

No character matches

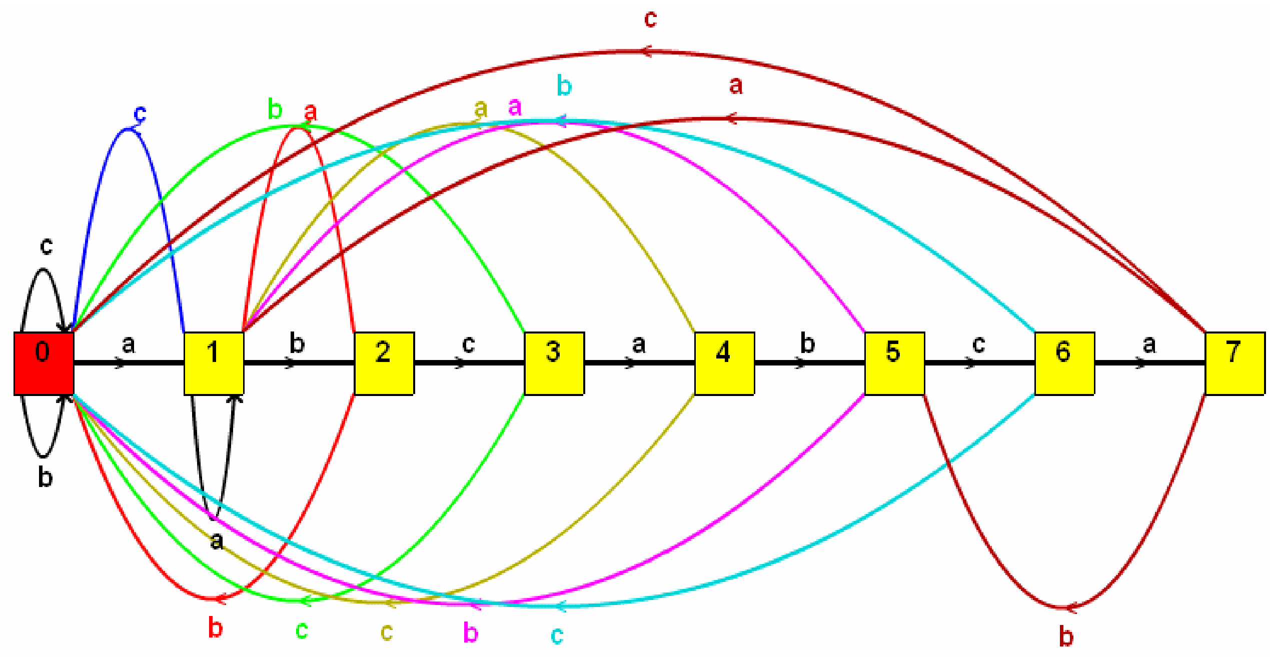
	a	b	c
0	1	0	0
1	1	2	0
2	1	0	3
3	4	0	0
4	1	5	0
5	1	0	6
6	7	0	0
7	1	5	0

Finite State Automaton

Graph Representation

A Finite State Automaton for string matching is often represented as a *directed graph,* also called *flow chart*. The *vertices* of the graph indicate the *states*. The *edges* are labeled with *alphabets*. The direction of an edge shows the transition to the next state when a labeled alphabet is input. A special sequence of edges, referred to as *spine*, is laid out horizontally. It indicates direction of *successful matching.*

Figures shows the graph representation of FSA for pattern '*abcabca*' and the associated transition table . For clarity, the edges are shown in different colors.



Finite State Automaton (FSA)

	a	b	c
0	1	0	0
1	1	2	0
2	1	0	3
3	4	0	0
4	1	5	0
5	1	0	6
6	7	0	0
7	1	5	0

Transition Table

Computing Transition Function
Implementation

The following code (*Source: Cormen et al*) computes transition function for a given pattern *P* defined over the alphabet Σ .

```
TRANSITION(P,Σ)
m←length[P]
for q←0 to m do                                ►Process all possible states
  for each symbol a in Σ do                    ►Process all characters in the alphabet
     for k← minimum ( m+1, q+2) do             ►Select size of the smaller of the two substrings
       repeat k←k-1
        until prefix P_k matches with the suffix of P_q a
        δ(q, a)←k
return δ
```

The outermost loop iterates $m+1$ times. The next loop cycles through |Σ| times. The next loop controlled by k runs at most $m+1$ times. The final repeat loop compares up to m characters to match suffix with prefix. Thus, the running time T_{tran} for computing the transition function is given by

$$T_{tran} = O((m+1).|\Sigma|.(m+1).m\ ) \text{ which simplifies to } O(m^3\ .|\Sigma|\ )$$

String Matching Using FSA

Algorithm

Once the transition function for pattern is computed, the FSA can be used to match given pattern with any text block. The matching procedure works as follows:

Step# 1: *Start with the initial state*

Step #2: *Extract a character, c, from the text block.*

Step #3*: Use transition function, with character c as argument, to determine the next state*

Step #4: *If the next state is the accepting state, print match is found*

Step #5: *Repeat Step#1 through Step #5 , until the text block is exhausted*

- The working of the matching algorithm is illustrated by the following diagrams

Finite State Automaton

String Matching

The following steps illustrate the use of Finite State Automaton to match pattern *abcabca* with an input string. *abcacbabaabcabcaacc* Initially the FSA is in state 0 (shown in red shade). It moves to the next state depending on the input character. The transition table gives the states of FSA when any of the valid character is entered The pattern is matched when the FSA ends up in the *accepting state 7*.

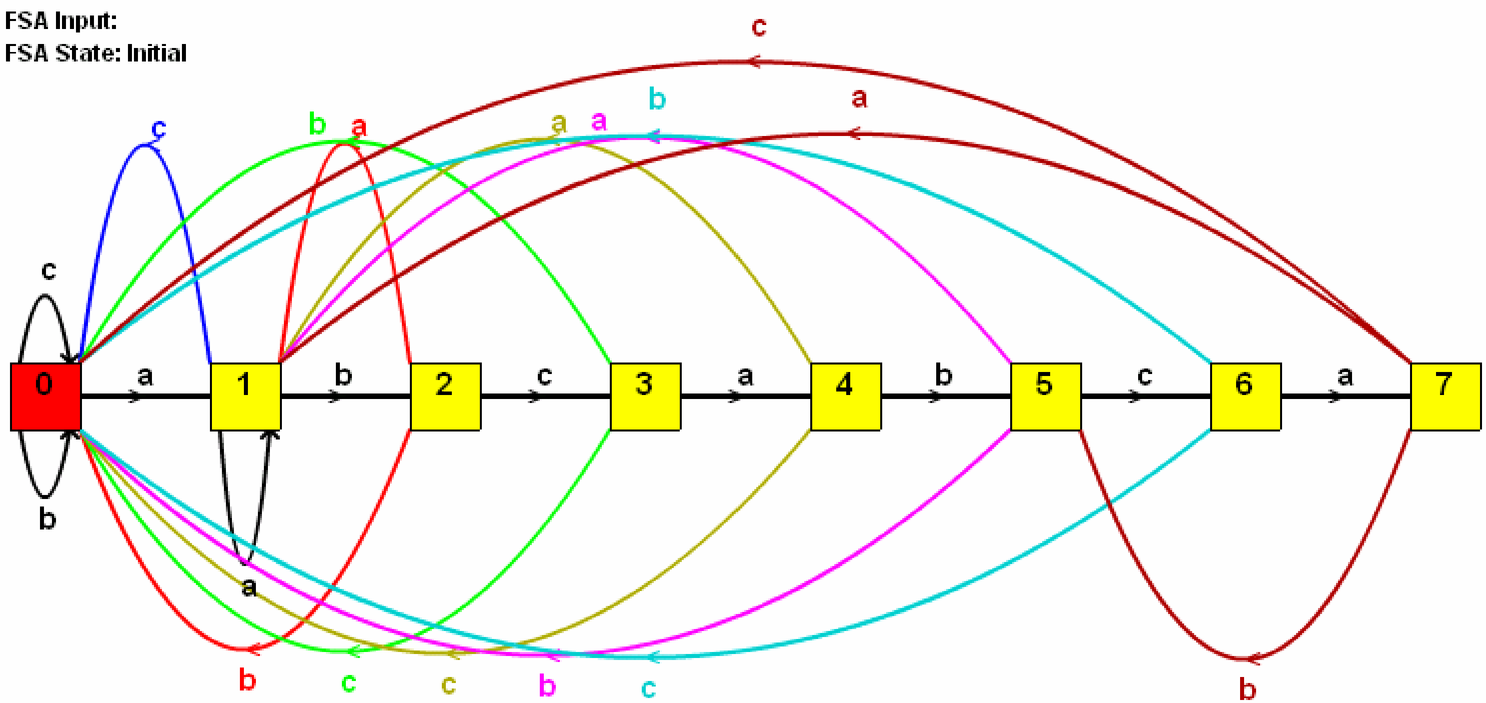
Input Text: a b c a c b a b a a b c a b c a a c c

Pattern: a b c a b c a

Step#: 1

FSA Input:

FSA State: Initial



Finite State Automaton (FSA)

	a	b	c
0	1	0	0
1	1	2	0
2	1	0	3
3	4	0	0
4	1	5	0
5	1	0	6
6	7	0	0
7	1	5	0

Transition Table

Finite State Automaton
String Matching

The input character is ‘*a*’. The FSA moves from state 0 to state 1

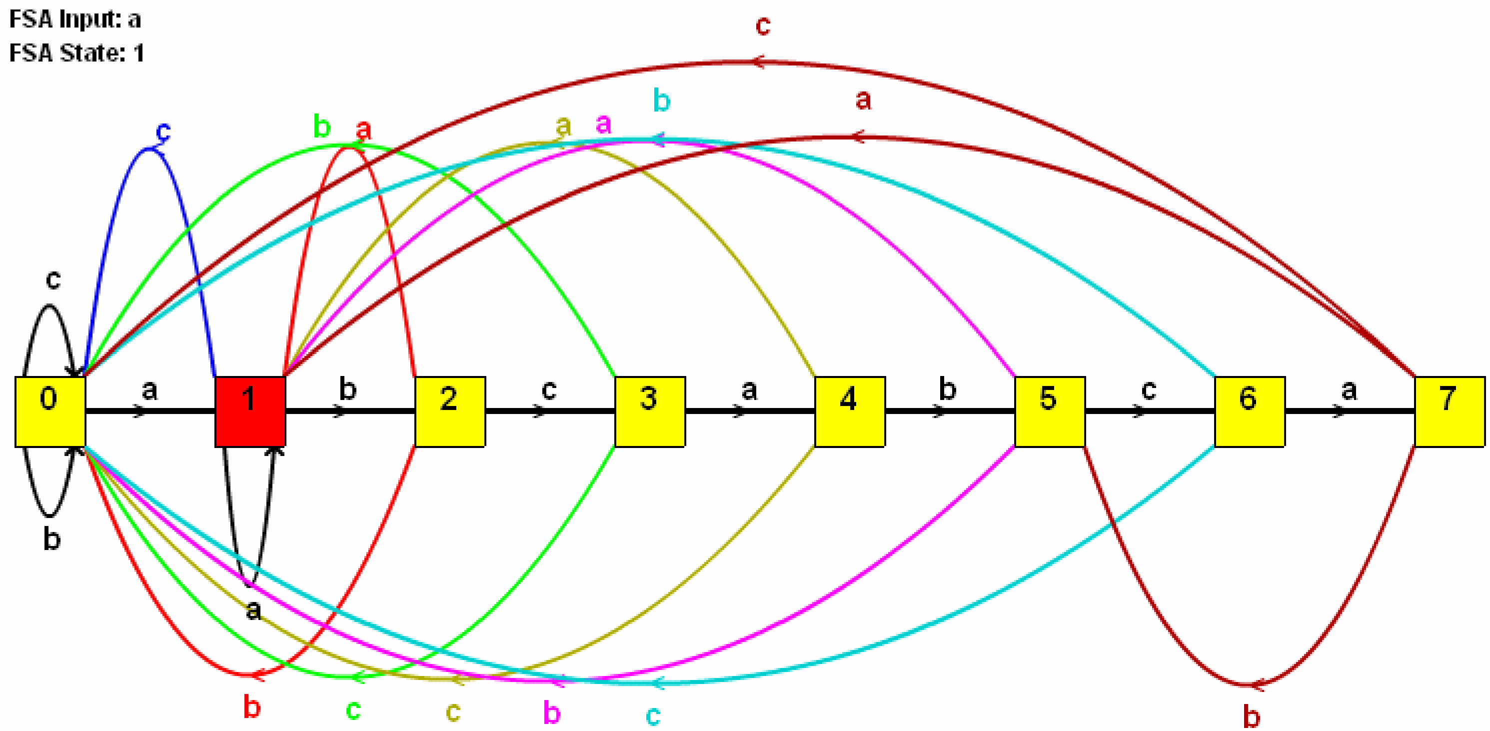
Input Text: a b c a c b a b a a b c a b c a a c c

Pattern: a b c a b c a

Step#: 2

FSA Input: a

FSA State: 1



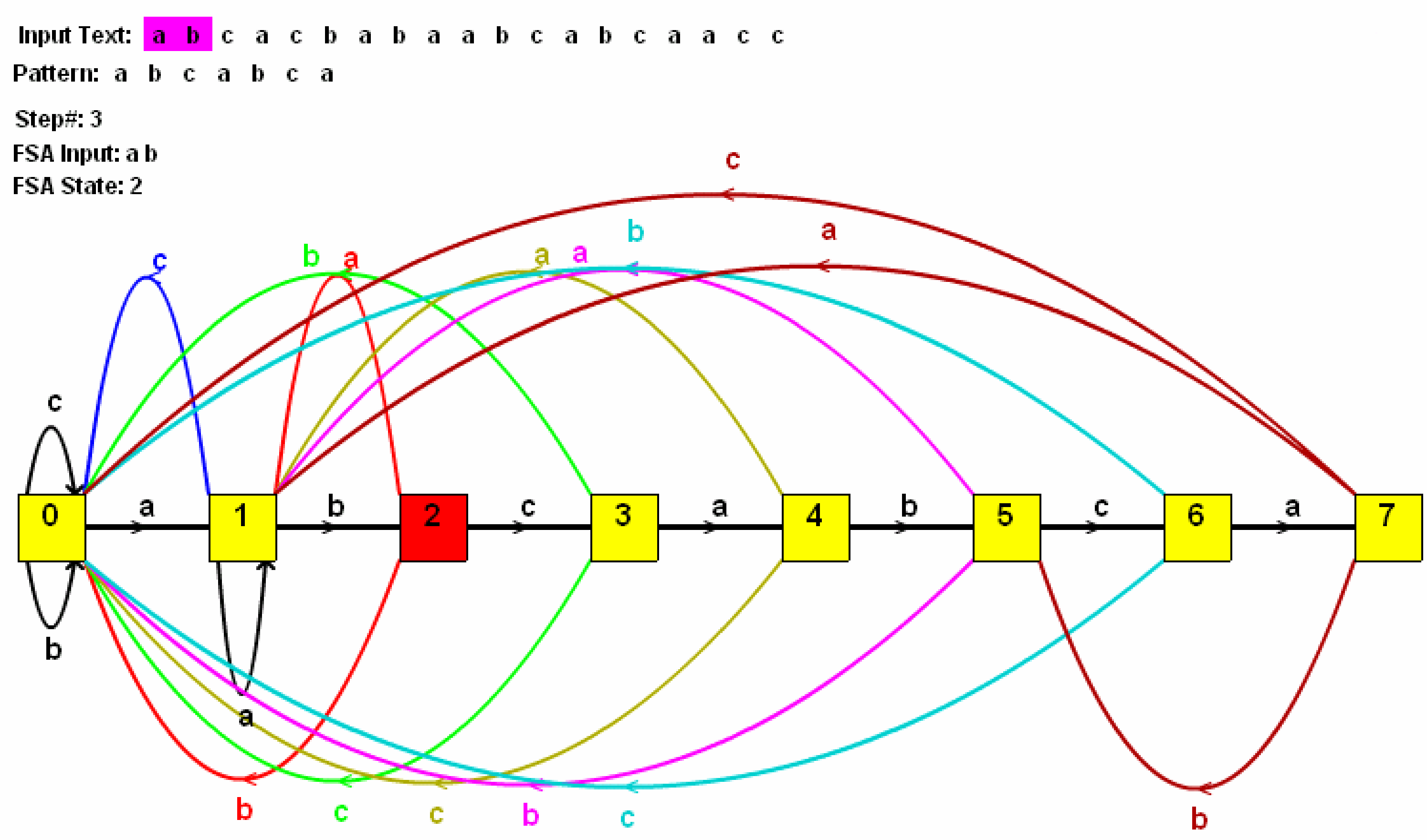
Finite State Automaton (FSA)

	a	b	c
0	1	0	0
1	1	2	0
2	1	0	3
3	4	0	0
4	1	5	0
5	1	0	6
6	7	0	0
7	1	5	0

Transition Table

Finite State Automaton
String Matching

The input character is ‘*b*’. The FSA moves from state 1 to state 2



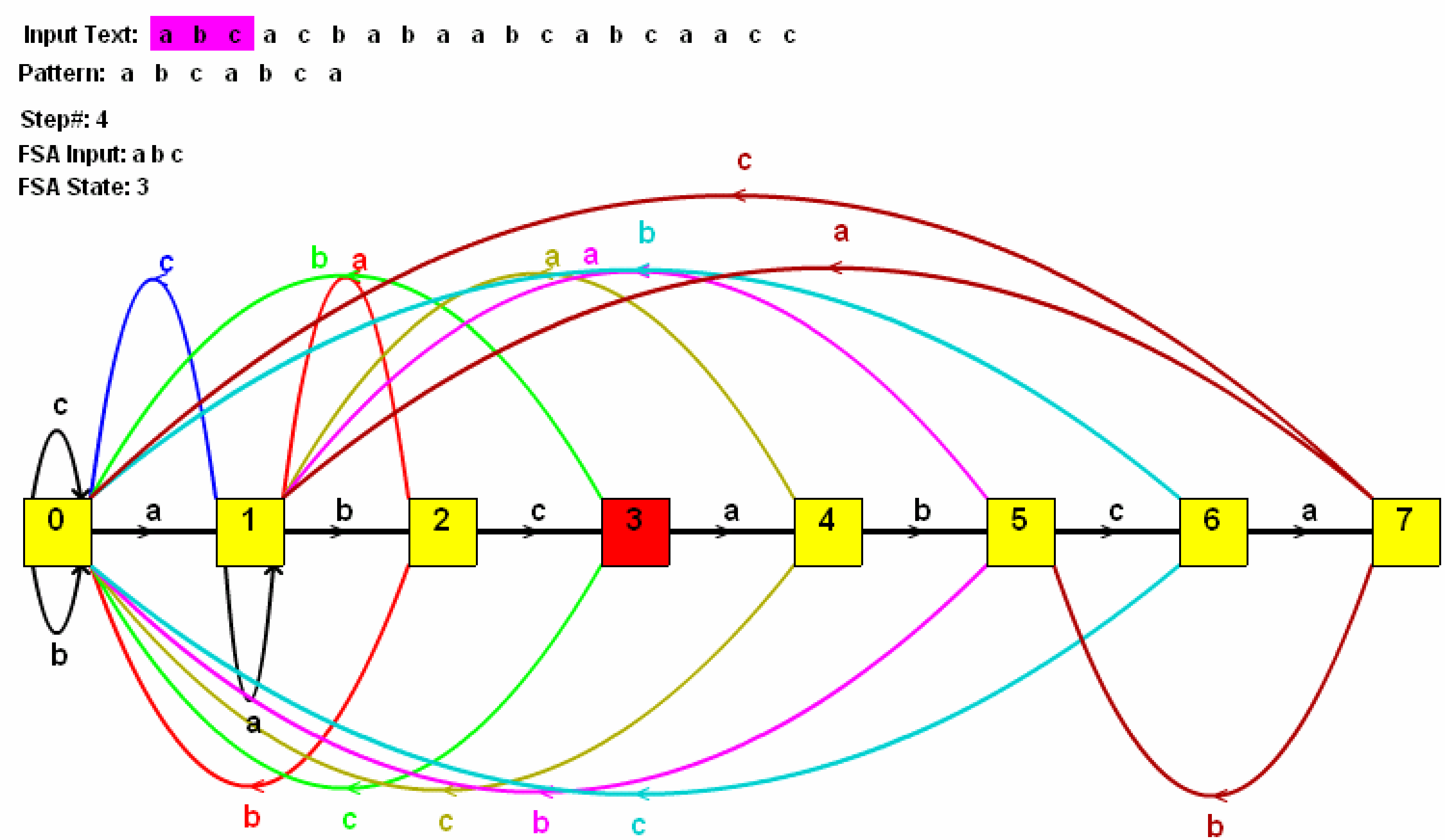
Finite State Automaton (FSA)

	a	b	c
0	1	0	0
1	1	2	0
2	1	0	3
3	4	0	0
4	1	5	0
5	1	0	6
6	7	0	0
7	1	5	0

Transition Table

Finite State Automaton
String Matching

The input character is '*c*'. The FSA moves from state 2 to state 3



Finite State Automaton (FSA)

	a	b	c
0	1	0	0
1	1	2	0
2	1	0	3
3	4	0	0
4	1	5	0
5	1	0	6
6	7	0	0
7	1	5	0

Transition Table

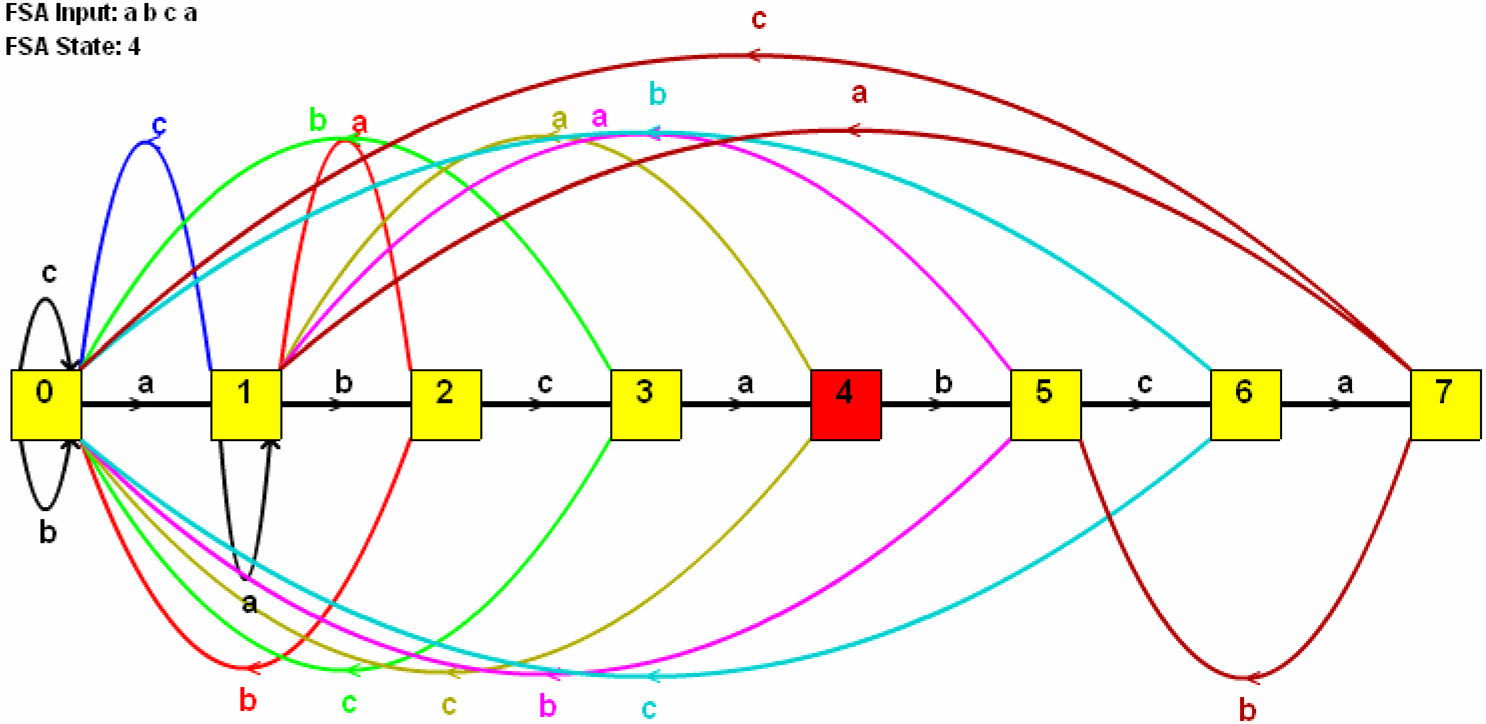
Finite State Automaton
String Matching

The input character is ‘*a*’. The FSA moves from state 3 to state 4

Input Text: a b c a c b a b a a b c a b c a a c c

Pattern: a b c a b c a

Step#: 5
FSA Input: a b c a
FSA State: 4



Finite State Automaton (FSA)

	a	b	c
0	1	0	0
1	1	2	0
2	1	0	3
3	4	0	0
4	1	5	0
5	1	0	6
6	7	0	0
7	1	5	0

Transition Table

Finite State Automaton

String Matching

The input character is ‘*c*’. The FSA moves from state 4 to state 0

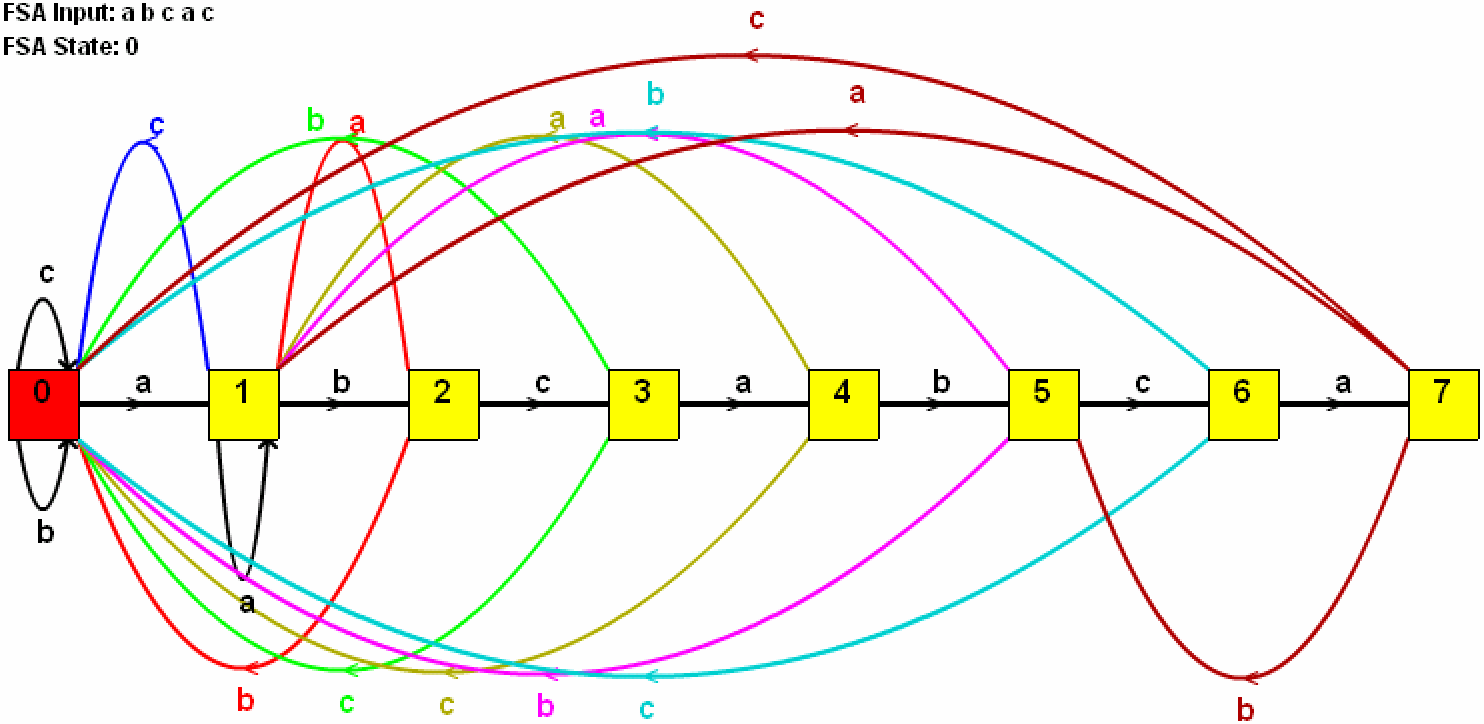
Input Text: a b c a c b a b a a b c a b c a a c c

Pattern: a b c a b c a

Step#: 6

FSA Input: a b c a c

FSA State: 0



Finite State Automaton (FSA)

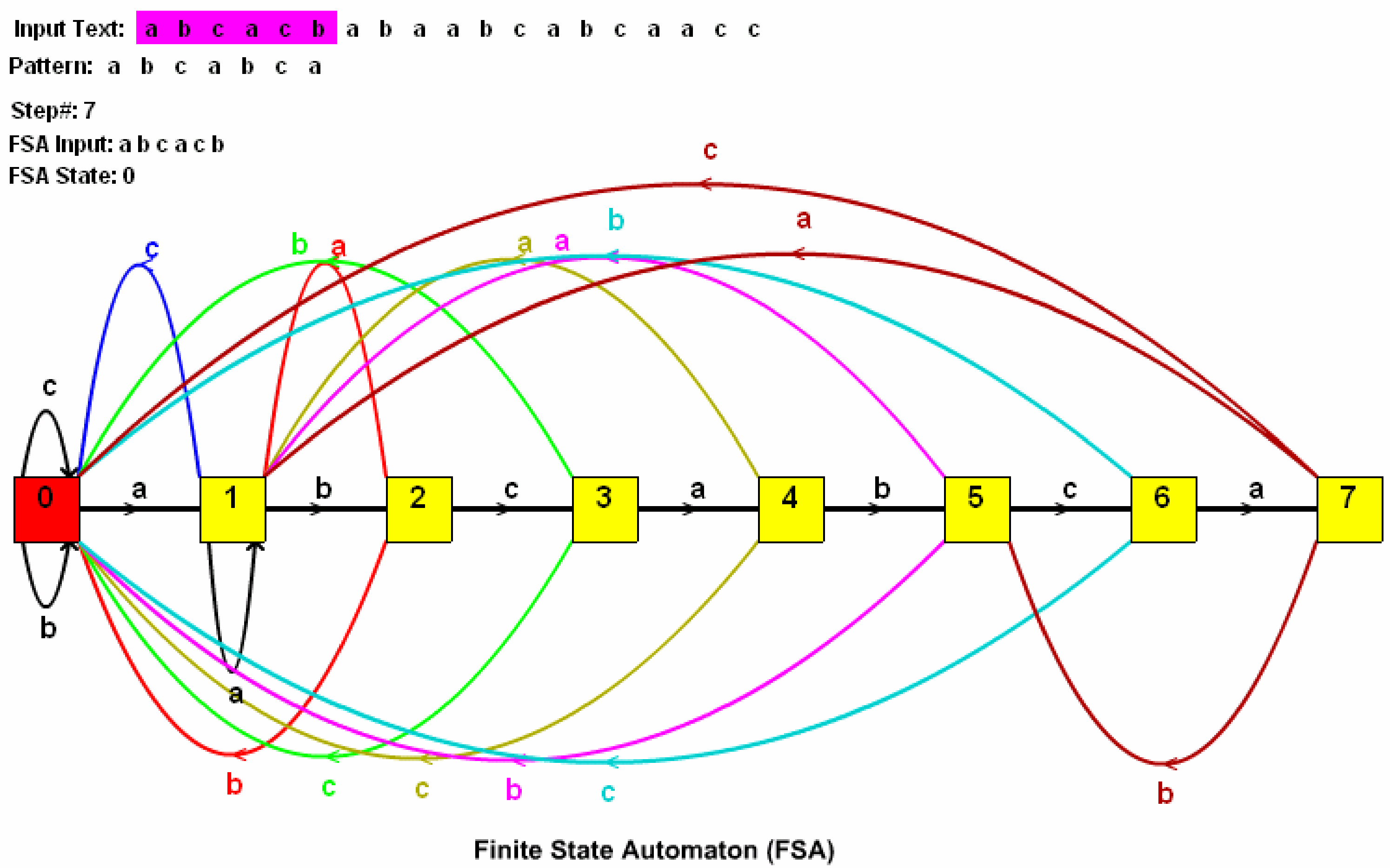
	a	b	c
0	1	0	0
1	1	2	0
2	1	0	3
3	4	0	0
4	1	5	0
5	1	0	6
6	7	0	0
7	1	5	0

Transition Table

Finite State Automaton

String Matching

The input character is ‘*b*’. The FSA moves back to state 0



Finite State Automaton (FSA)

	a	b	c
0	1	0	0
1	1	2	0
2	1	0	3
3	4	0	0
4	1	5	0
5	1	0	6
6	7	0	0
7	1	5	0

Transition Table

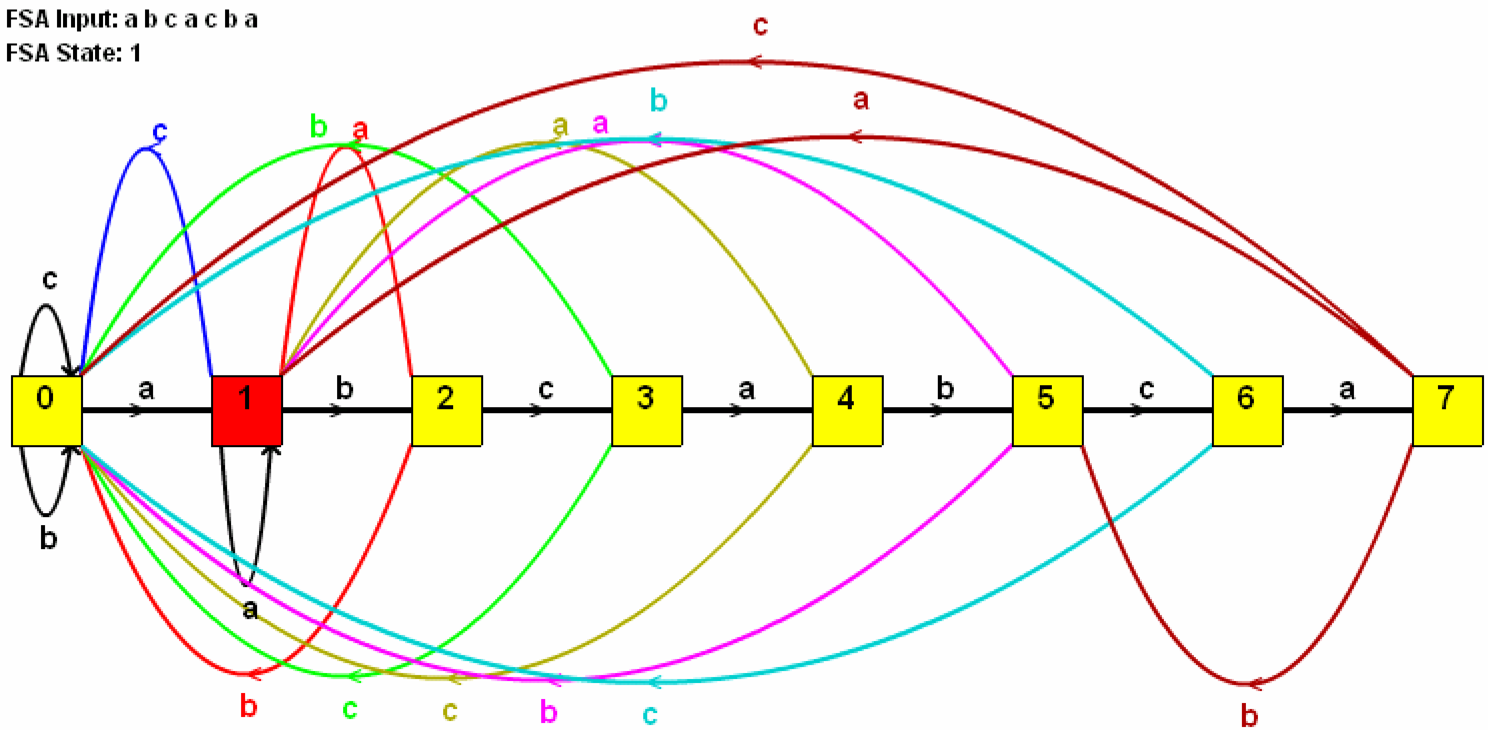
Finite State Automaton
String Matching

The input character is ‘*a*’. The FSA moves from state 0 to state 1

Input Text: a b c a c b a b a a b c a b c a a c c

Pattern: a b c a b c a

Step#: 8
FSA Input: a b c a c b a
FSA State: 1



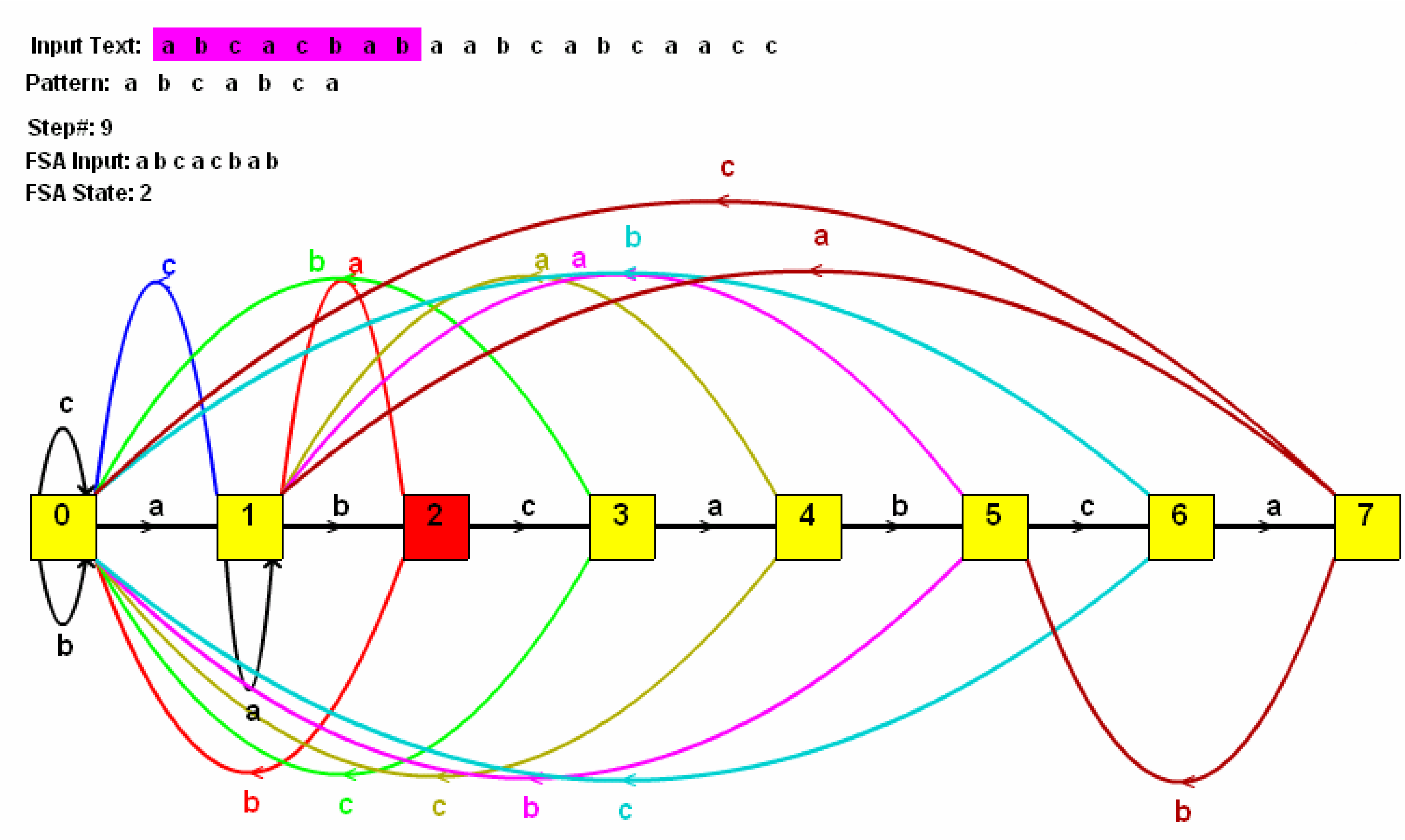
Finite State Automaton (FSA)

	a	b	c
0	1	0	0
1	1	2	0
2	1	0	3
3	4	0	0
4	1	5	0
5	1	0	6
6	7	0	0
7	1	5	0

Transition Table

Finite State Automaton
String Matching

The input character is ‘*b*’. The FSA moves from state 1 to state 2



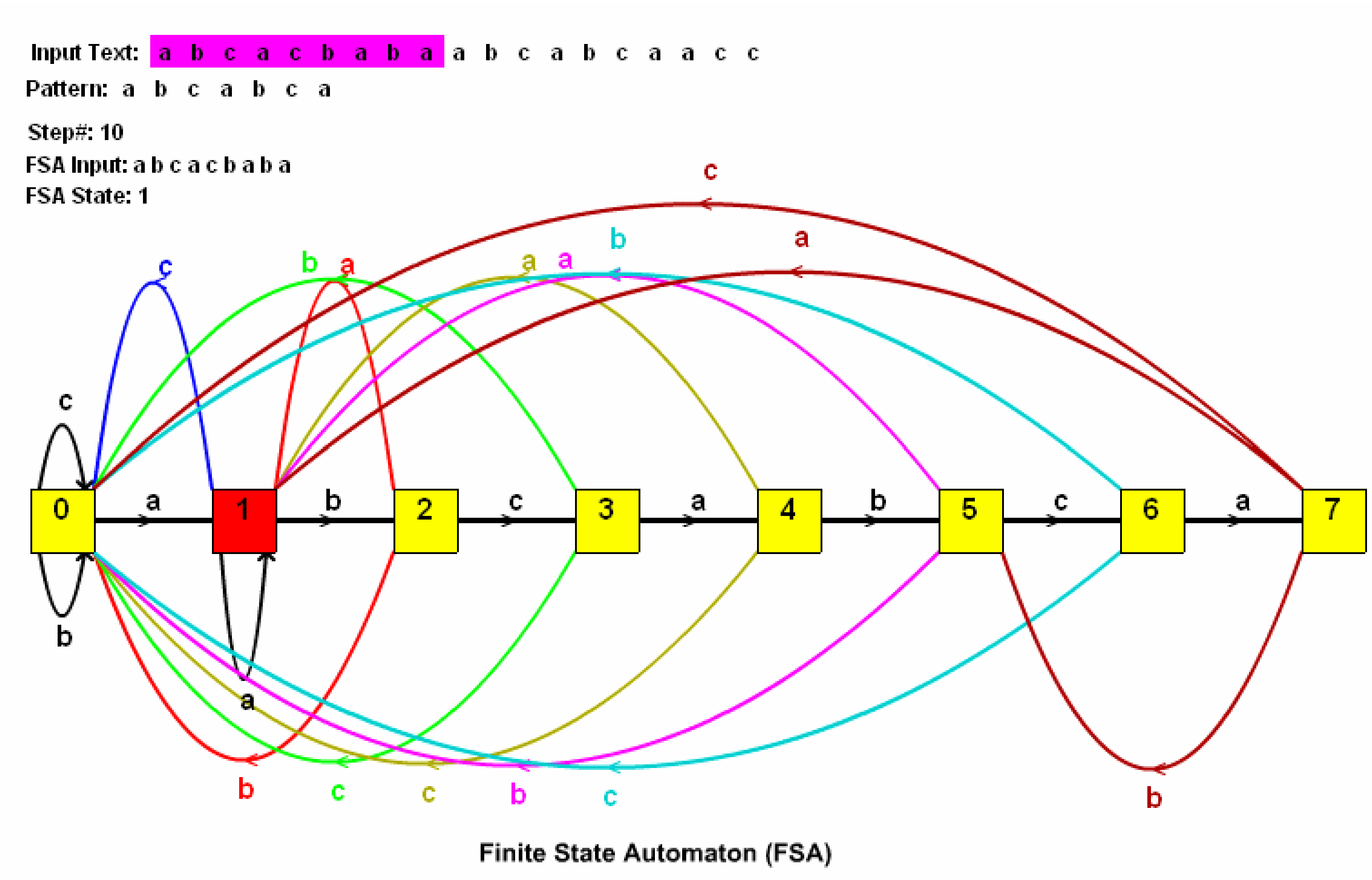
Finite State Automaton (FSA)

	a	b	c
0	1	0	0
1	1	2	0
2	1	0	3
3	4	0	0
4	1	5	0
5	1	0	6
6	7	0	0
7	1	5	0

Transition Table

Finite State Automaton
String Matching

The input character is ‘*a*’. The FSA moves from state 2 to state 1

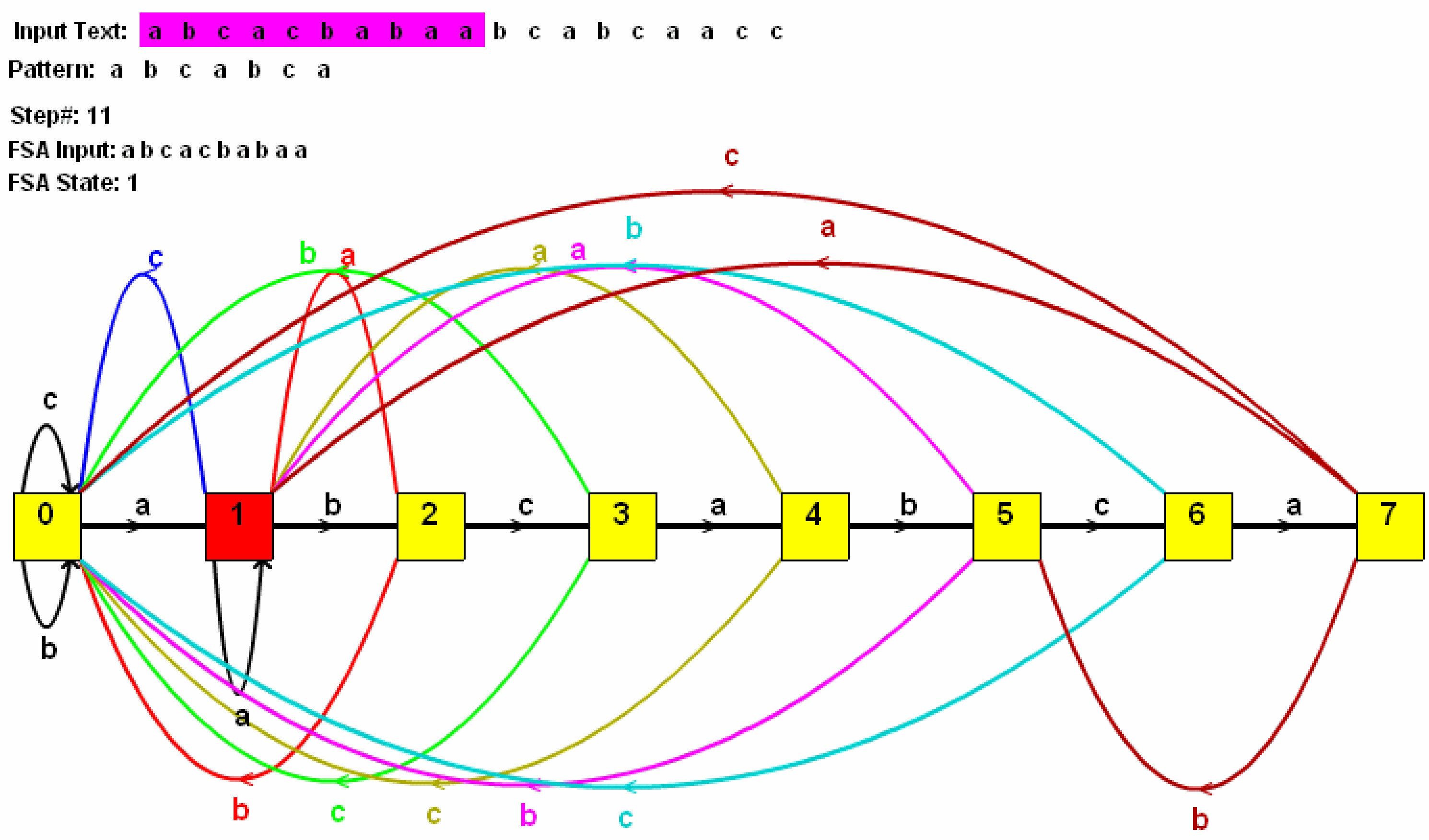


	a	b	c
0	1	0	0
1	1	2	0
2	1	0	3
3	4	0	0
4	1	5	0
5	1	0	6
6	7	0	0
7	1	5	0

Transition Table

Finite State Automaton
String Matching

The input character is '*a*'. The FSA moves back to state 1



Finite State Automaton (FSA)

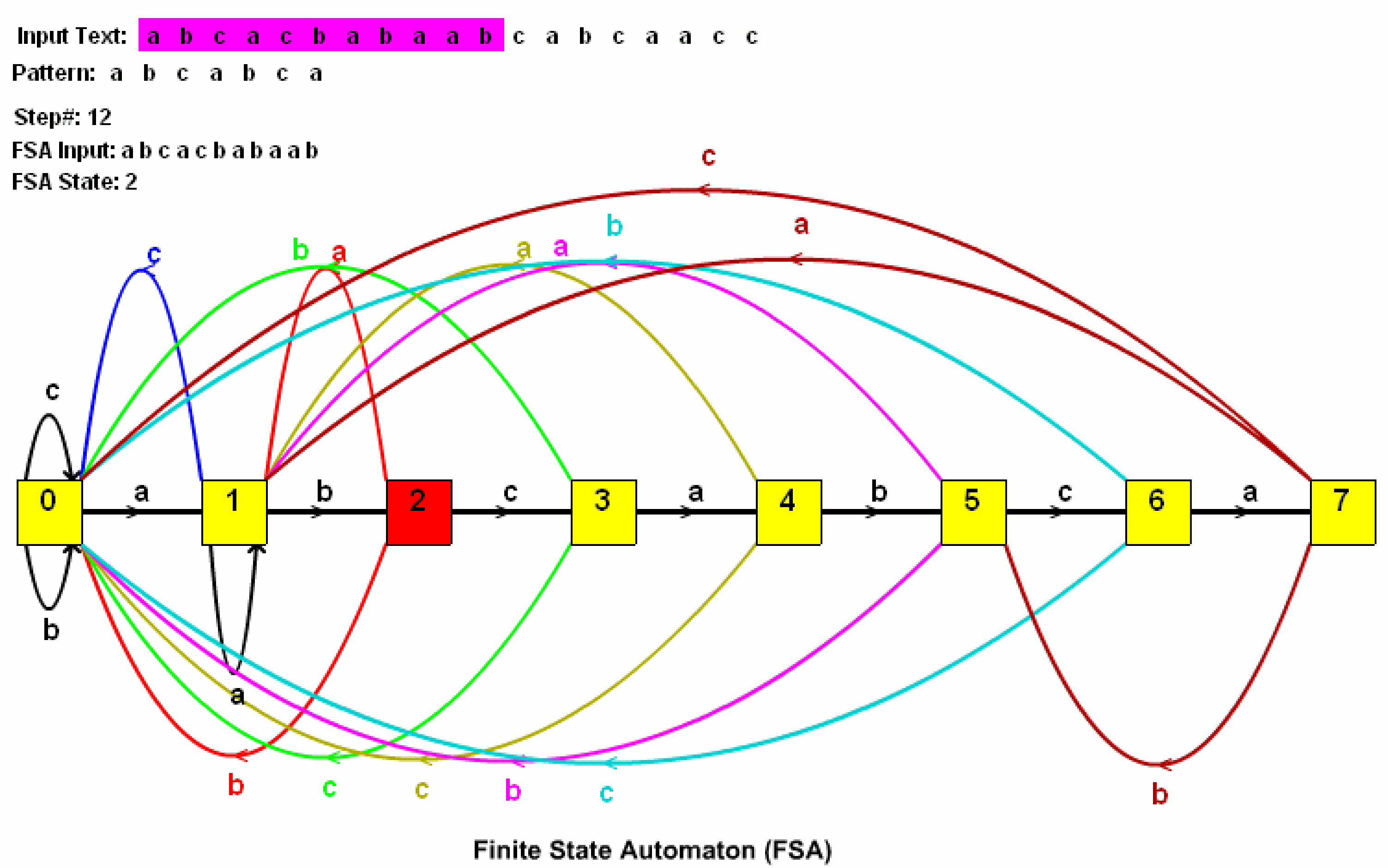
	a	b	c
0	1	0	0
1	1	2	0
2	1	0	3
3	4	0	0
4	1	5	0
5	1	0	6
6	7	0	0
7	1	5	0

Transition Table

Finite State Automaton

String Matching

The input character is ‘*b*’. The FSA moves from state 1 to state 2



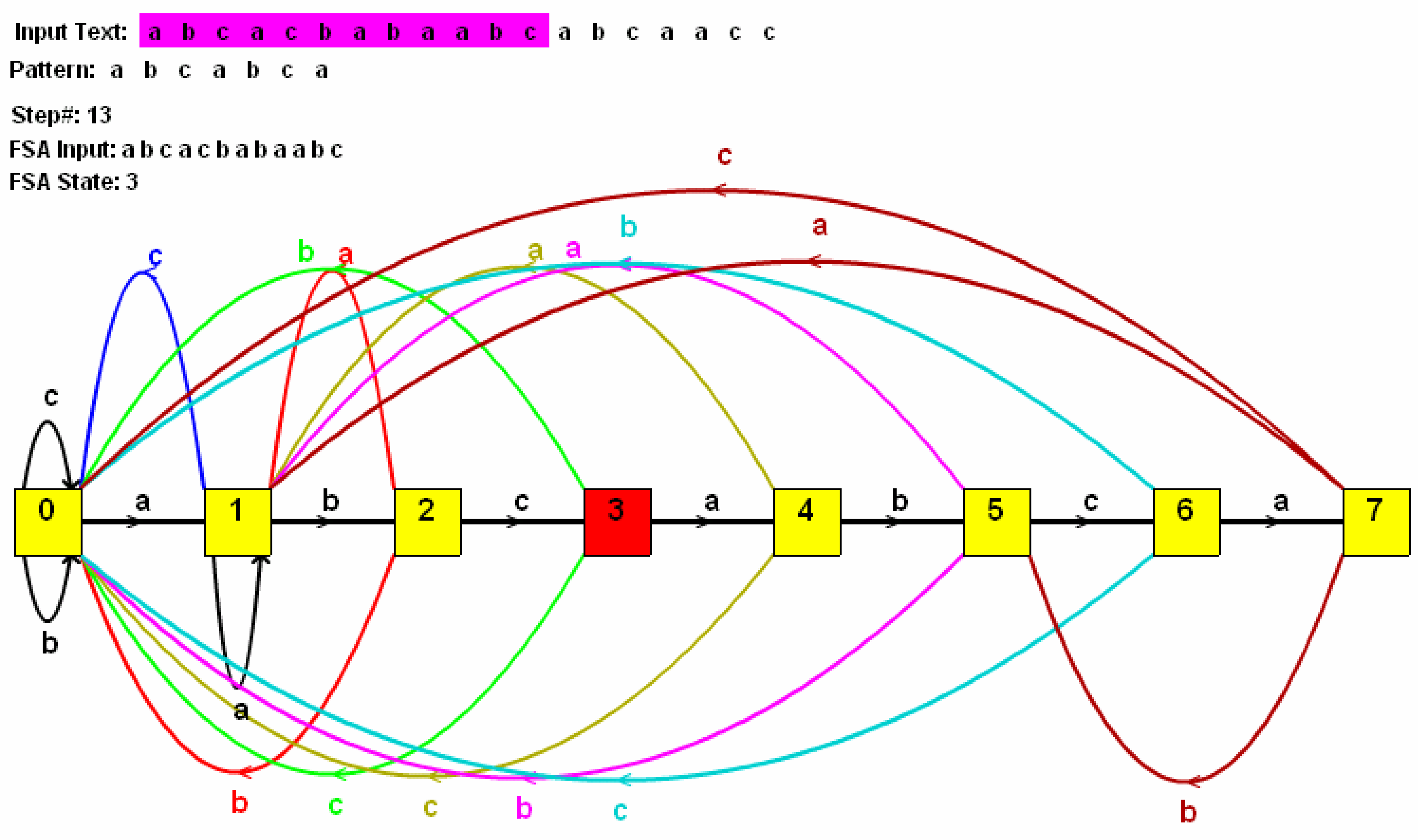
	a	b	c
0	1	0	0
1	1	2	0
2	1	0	3
3	4	0	0
4	1	5	0
5	1	0	6
6	7	0	0
7	1	5	0

Transition Table

Finite State Automaton

String Matching

The input character is ‘*c*’. The FSA moves from state 2 to state 3



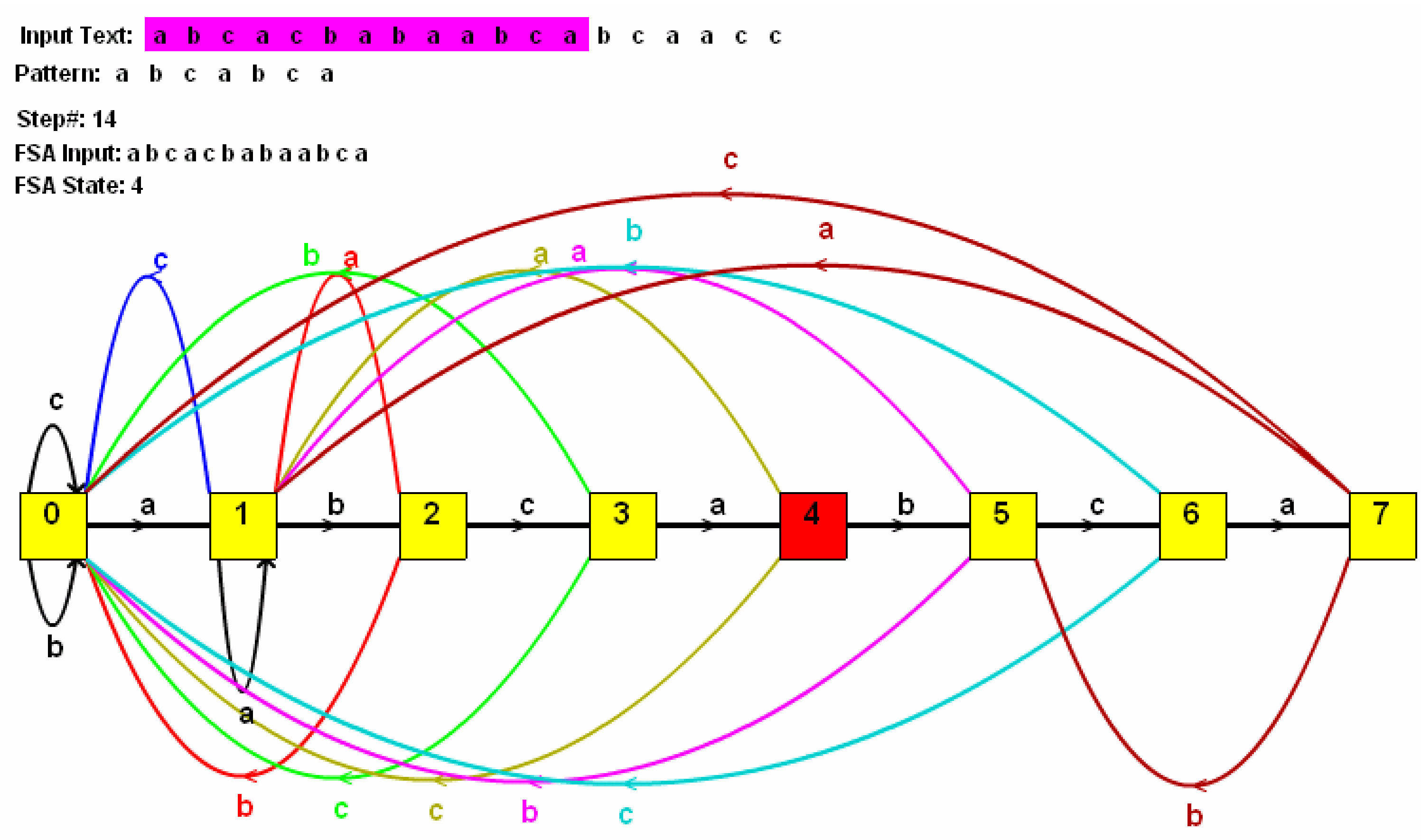
Finite State Automaton (FSA)

	a	b	c
0	1	0	0
1	1	2	0
2	1	0	3
3	4	0	0
4	1	5	0
5	1	0	6
6	7	0	0
7	1	5	0

Transition Table

Finite State Automaton
String Matching

The input character is ‘*a*’. The FSA moves from state 3 to state 4



Finite State Automaton (FSA)

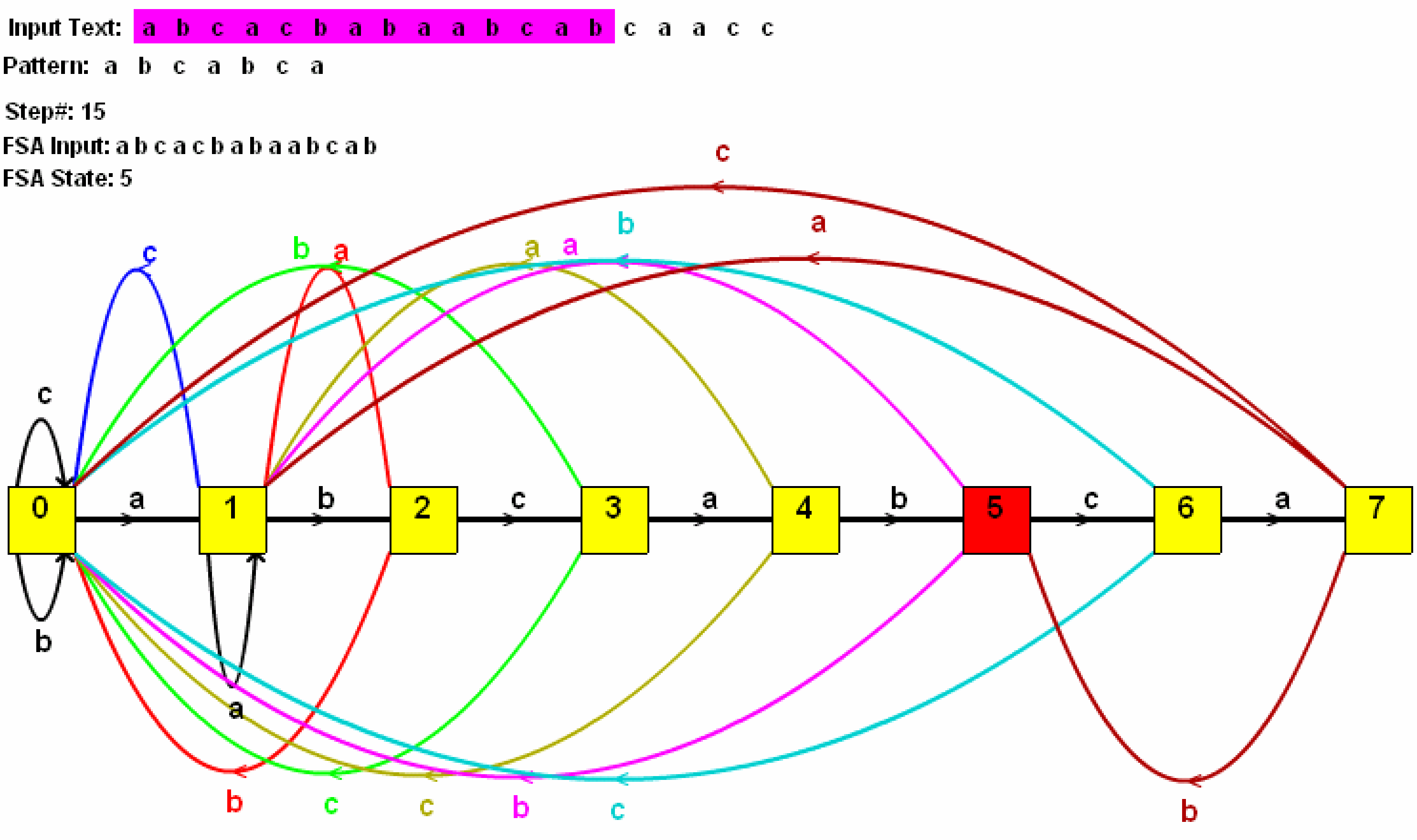
	a	b	c
0	1	0	0
1	1	2	0
2	1	0	3
3	4	0	0
4	1	5	0
5	1	0	6
6	7	0	0
7	1	5	0

Transition Table

Finite State Automaton

String Matching

The input character is ‘*b*’. The FSA moves from state 4 to state 5



Finite State Automaton (FSA)

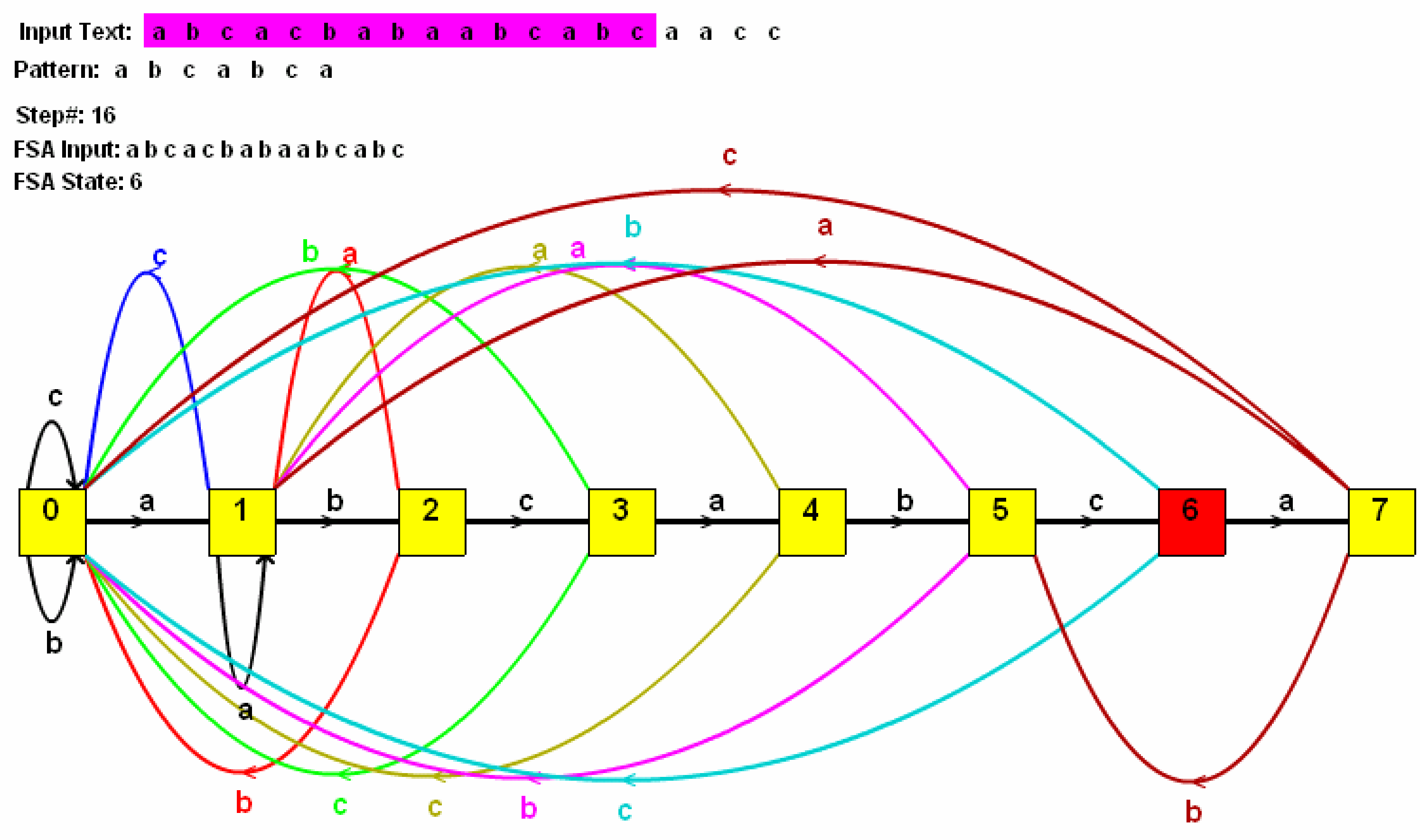
	a	b	c
0	1	0	0
1	1	2	0
2	1	0	3
3	4	0	0
4	1	5	0
5	1	0	6
6	7	0	0
7	1	5	0

Transition Table

Finite State Automaton

String Matching

The input character is ‘*c*’. The FSA moves from state 5 to state 6



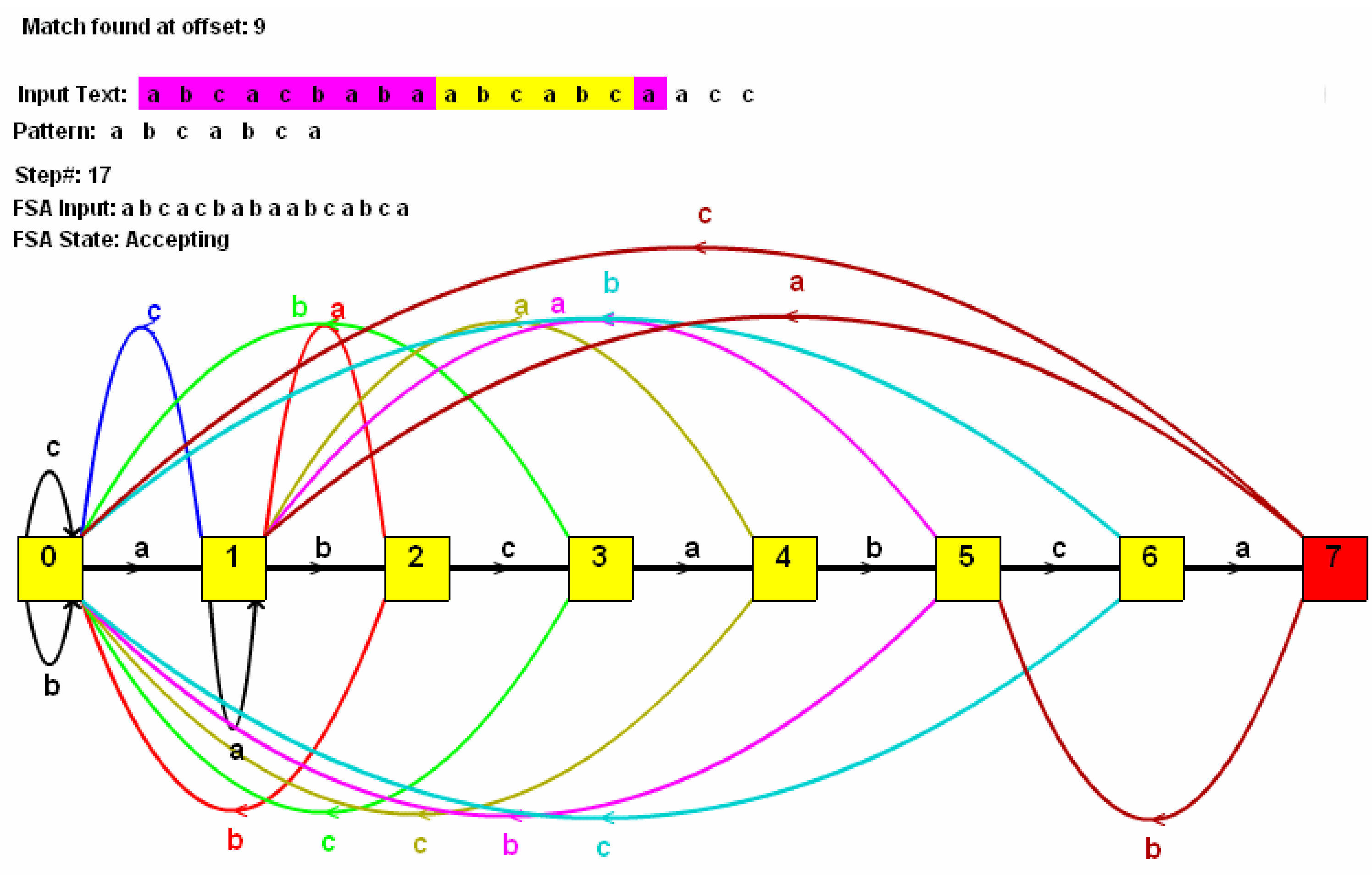
Finite State Automaton (FSA)

	a	b	c
0	1	0	0
1	1	2	0
2	1	0	3
3	4	0	0
4	1	5	0
5	1	0	6
6	7	0	0
7	1	5	0

Transition Table

Finite State Automaton
String Matching

The input character is ‘*a*’. The FSA moves from state 6 to state 7, which is the ***accepting state***. Thus, the ***pattern matches*** with input string



Finite State Automaton (FSA)

	a	b	c
0	1	0	0
1	1	2	0
2	1	0	3
3	4	0	0
4	1	5	0
5	1	0	6
6	7	0	0
7	1	5	0

Transition Table

String Matching

Implementation

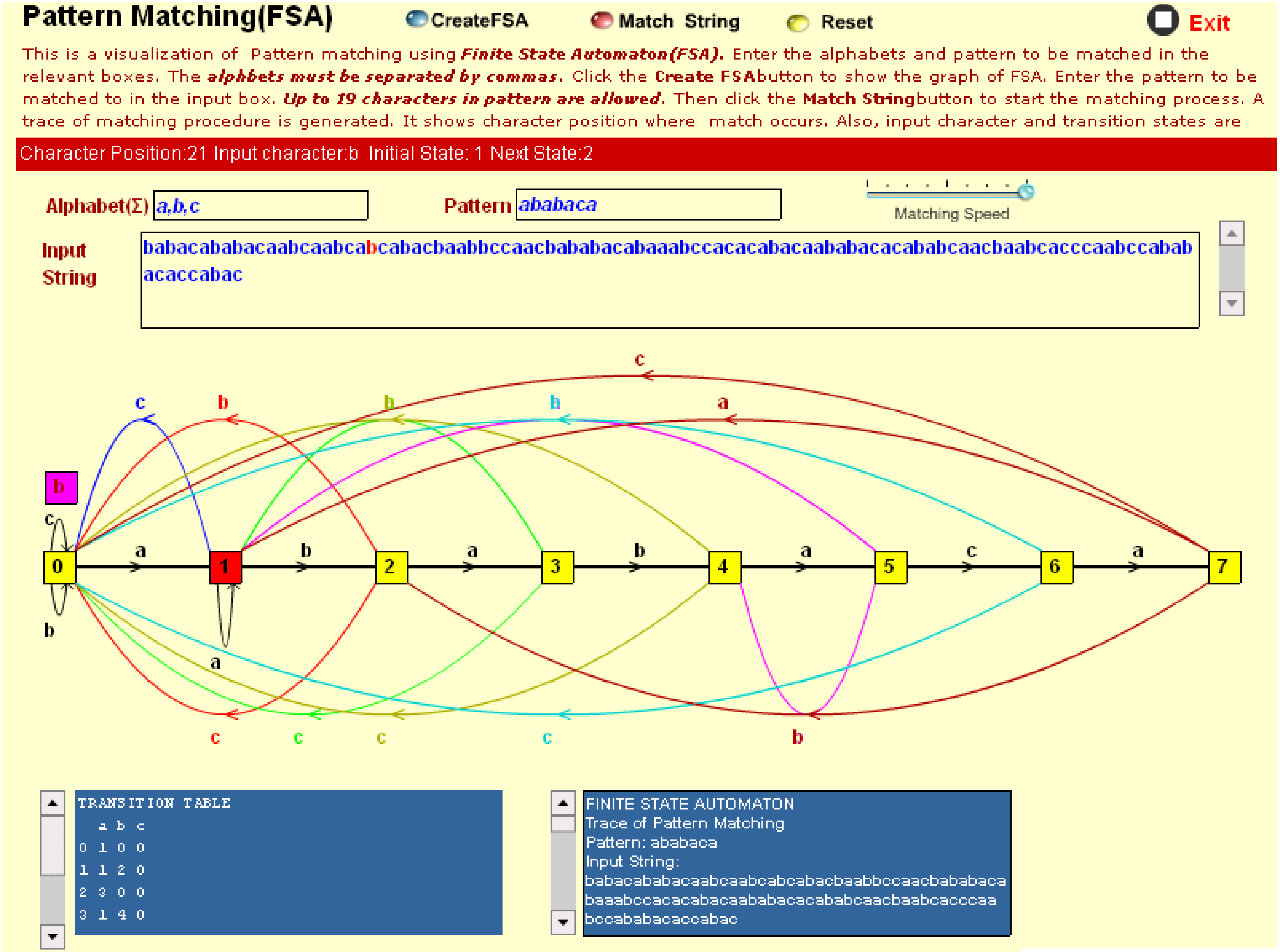
The following code perform the pattern matching procedure by using the transition function . It accepts the text block, transition function , and length of pattern as inputs

```
MATCHER(T, δ, m)
n←length[T]
 q←0                          ► start with initial states
    for i←1 to n do
        c←T[i]                ► extract character from text
         q← δ(q, c)           ► move to next FSA state
         if q=m               ► check if current state equal pattern length
            then print match occurs with shift i-m     ► if so  pattern matches
```

➢ The procedure consists of a *single loop which runs n times*. Thus, running time of string matcher *O(n)*, where *n* is number of characters in the text

FSA

Visualization



String Matching Algorithms
Comparison

Assuming that length of pattern, *m,* is 20 and $|\Sigma|=5$, a comparison of ***Brute Force*** and ***FSA*** algorithms is given in the table below, for text blocks of different lengths. For small *n*, the Naive algorithm performs better, but for large *n* the FSA algorithm is far superior.

n	FSA	Brute
1000	41000	20000
2000	42000	40000
3000	43000	60000
4000	44000	80000
5000	45000	100000
6000	46000	120000
7000	47000	140000
8000	48000	160000
9000	49000	180000
10000	50000	200000
11000	51000	220000
12000	52000	240000
13000	53000	260000
14000	54000	280000
15000	55000	300000

Running time of Naïve and FSA algorithms