Fonte: ash.atozviews.com

Compacto dos melhores momentos

AULA 24

Introdução

Problema: Dada uma string `pat` e uma string `txt`, encontrar uma (todas) ocorrência(s) de `pat` em `txt`.

Exemplo: encontre `ATTGG` em:

```
TGGTAAGCGGTTCCTGCCCCGGCTCAGGGCCAAGAACAGATGAGACAGCTGAGTGATGGGCCAAACAGGATATCTGTGG
TAAGCAGTTCCTGCCCCGGCTCGGGGCCAAGAACAGATGGTCCCCAGATGCGGTCCAGCCCTCAGCAGTTTCTAGTGAA
TCATCAGATGTTTCCAGGGTGCCCCAAGGACCTGAAAATGACCCTGTACCTTATTTGAACTAACCAATCAGTTCGCTTC
TCGCTTCTGTTCGCGCGCTTCCGCTCTCCGAGCTCAATAAAAGAGCCCACAACCCCTCACTCGGCGCGCCAGTCTTCCG
ATAGACTGCGTCGCCCGGGTACCCGTATTCCCAATAAAGCCTCTTGCTGTTTGCATCCGAATCGTGGTCTCGCTGTTCC
TTGGGAGGGTCTCCTCTGAGTGATTGACTACCCACGACGGGGGTCTTTCATTTGGGGGCTCGTCCGGGATTTGGAGACC
CCTGCCCAGGGACCACCGACCCACCACCGGGAGGTAAGCTGGCCAGCAACTTATCTGTGTCTGTCCGATTGTCTAGTGT
CTATGTTTGATGTTATGCGCCTGCGTCTGTACTAGTTAGCTAACTAGCTCTGTATCTGGCGGACCCGTGGTGGAACTGA
CGAGTTCTGAACACCCGGCCGCAACCCTGGGAGACGTCCCAGGGACTTTGGGGGCCGTTTTTGTGGCCCGACCTGAGGA
AGGGAGTCGATGTGGAATCCGACCCCGTCAGGATATGTGGTTCTGGTAGGAGACGAGAACCTAAAACAGTTCCCGCCTC
CGTCTGAATTTTTGCTTTCGGTTTGGAACCGAAGCCGCGCGTCTTGTCTGCTGCAGCATCGTTCTGTGTTGTCTCTGTC
TGACTGTGTTTCTGTATTTGTCTGAAAATTAGGGCCAGACTGTTACCACTCCCTTAAGTTTGACCTTAGGTCACTGGAA
AGATGTCGAGCGGATCGCTCACAACCAGTCGGTAGATGTCAAGAAGAGACGTTGGGTTACCTTCTGCTCTGCAGAATGG
CCAACCTTTAACGTCGGATGGCCGCGAGACGGCACCTTTAACCGAGACCTCATCACCCAGGTTAAGATCAAGGTCTTTT
CACCTGGCCCGCATGGACACCCAGACCAGGTCCCCTACATCGTGACCTGGGAAGCCTTGGCTTTTGACCCCCCTCCCTG
GGTCAAGCCCTTTGTACACCCTAAGCCTCCGCCTCCTCTTCCTCCATCCGCCCCGTCTCTCCCCCTTGAACCTCCTCGT
TCGACCCCGCCTCGATCCTCCCTTTATCCAGCCCTCACTCCTTCTCTAGGCGCCGGAATTCGTTAACTCGAGGATCCGG
CTGTGGAATGTGTGTCAGTTAGGGTGTGGAAAGTCCCCAGGCTCCCCAGCAGGCAGAAGTATGCAAAGCATGCATCTCA
ATTAGTCAGCAACCAGGTGTGGAAAGTCCCCAGGCTCCCCAGCAGGCAGAAGTATGCAAAGCATGCATCTCAATTAGTC
AGCAACCATAGTCCCGCCCCTAACTCCGCCCATCCCGCCCCTAACTCCGCCCAGTTCCGCCCATTCTCCGCCCCATGGC
TGACTAATTTTTTTATTTATGCAGAGGCCGAGGCCGCCTCGGCCTCTGAGCTATTCCAGAAGTAGTGAGGAGGCTTTT
TTGGAGGCCTAGGCTTTTGCAAAAAGCTGCCCAAGCTGATCCCCGGGGGCAATGAGATATGAAAAAGCCTGAACTCACC
GCGACGTCTGTCGAGAAGTTTCTGATCGAAAAGTTCGACAGCGTCTCCGACCTGATGCAGCTCTCGGAGGGCGAAGAAT
```

Algoritmo de força bruta

pat = a b a b b a b a b b a

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	
	a	b	a	a	b	a	b	a	b	b	a	b	a	b	a	b	b	a	b	a	b	b	a	txt
0	a	b	a	b	b	a	b	a	b	b	a													
1		a	b	a	b	b	a	b	a	b	b	a												
2			a	b	a	b	b	a	b	a	b	b	a											
3				a	b	a	b	b	a	b	a	b	b	a										
4					a	b	a	b	b	a	b	a	b	b	a									
5						a	b	a	b	b	a	b	a	b	b	a								
6							a	b	a	b	b	a	b	a	b	b	a							
7								a	b	a	b	b	a	b	a	b	b	a						
8									a	b	a	b	b	a	b	a	b	b	a					
9										a	b	a	b	b	a	b	a	b	b	a				
10											a	b	a	b	b	a	b	a	b	b	a			
11												a	b	a	b	b	a	b	a	b	b	a		
12													a	b	a	b	b	a	b	a	b	b	a	

Conclusões

O consumo de tempo de `search()` força-bruta no pior caso é $O((n - m + 1)m)$.

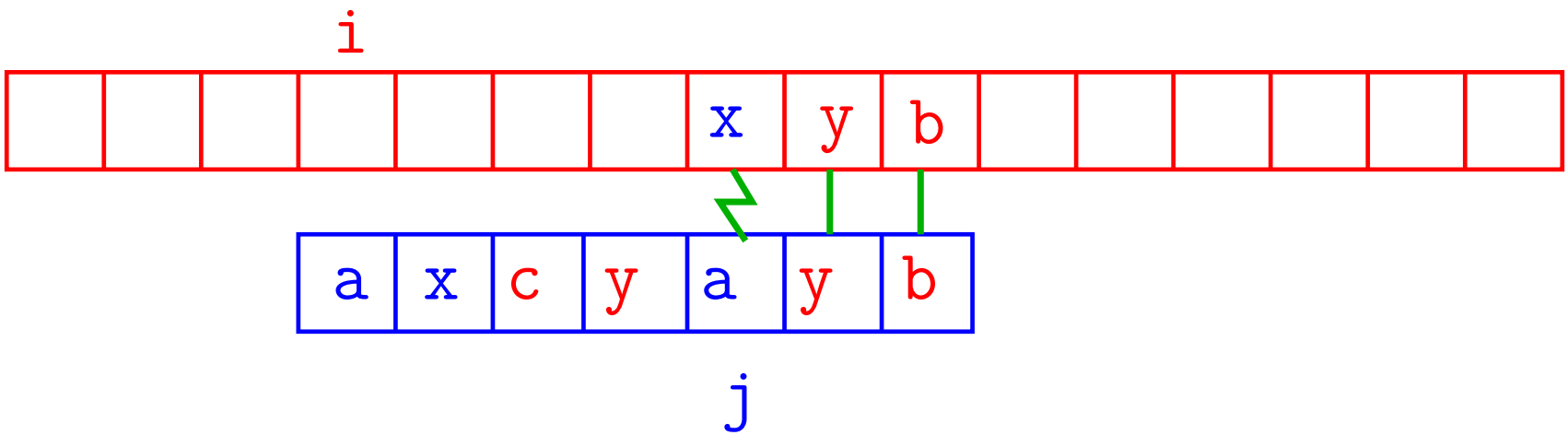
O consumo de tempo de `search()` força-bruta no melhor caso é $O(n - m + 1)$.

Isto significa que no pior caso o consumo de tempo é essencialmente proporcional a mn.

Em geral o algoritmo é rápido e faz não mais que $1.1 \times n$ comparações.

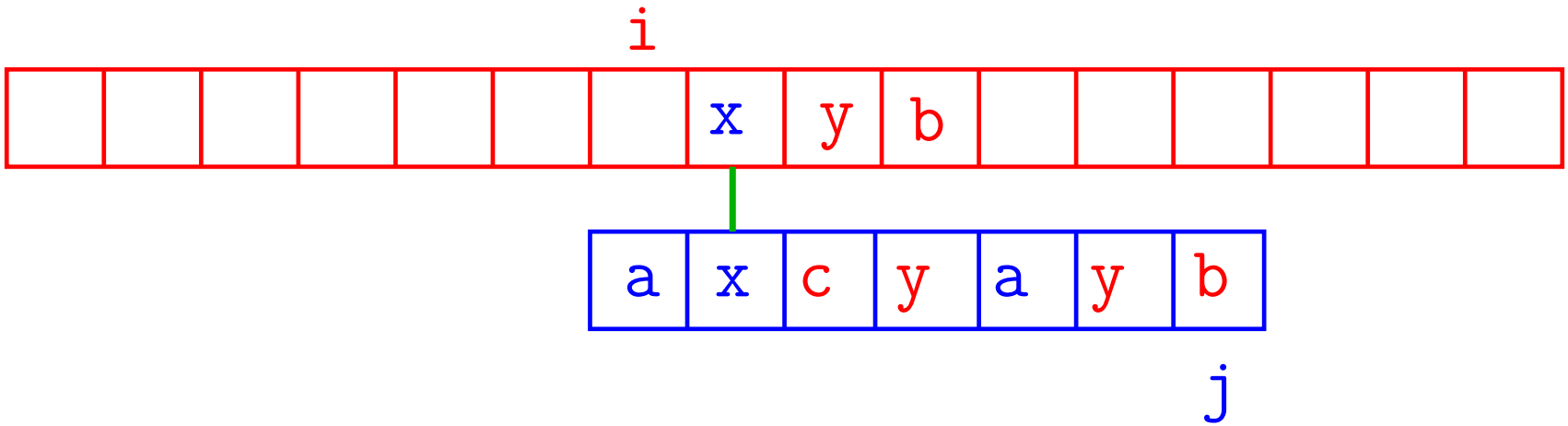
Boyer-Moore

O primeiro algoritmo de R.S. Boyer e J.S. Moore (1977) é baseado na seguinte heurística.



Boyer-Moore

O primeiro algoritmo de R.S. Boyer e J.S. Moore (1977) é baseado na seguinte heurística.



Bad-character heuristic

Para implementar essa ideia fazemos um pré-processamento de `pat`, determinando para cada símbolo `x` do alfabeto a posição de sua última ocorrência em `pat`.

`pat`

0	1	2	3	4	5	6
a	n	d	a	n	d	o

`right`

0	...	'a'	'b'	'c'	'd'	...	...	'n'	'o'	'p'	...	255
-1	$\ldots$	3	-1	-1	5	$\ldots$	$\ldots$	4	6	-1	$\ldots$	$\ldots$

Boyer-Moore

pat = a n d a n d o

```
  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
  a s   a n d o r i n h a s   a n d a m   a n d a n d o   a l t o txt
1 a n d a n d o
```

Boyer-Moore

pat = a n d a n d o

```
  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
  a s   a n d o r i n h a s   a n d a m   a n d a n d o   a l t o txt
1 a n d a n d o
2       a n d a n d o
```

Boyer-Moore

pat = a n d a n d o

```
  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
  a s   a n d o r i n h a s   a n d a m   a n d a n d o   a l t o txt
1 a n d a n d o
2     a n d a n d o
3       a n d a n d o
```

Boyer-Moore

pat = a n d a n d o

```
  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
  a s   a n d o r i n h a s   a n d a m   a n d a n d o   a l t o txt
-------------------------------------------------------------------
1 a n d a n d o
2       a n d a n d o
3           a n d a n d o
4               a n d a n d o
```

Boyer-Moore

pat = a n d a n d o

```
  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
  a s   a n d o r i n h a s   a n d a m   a n d a n d o   a l t o txt
-------------------------------------------------------------------
1 a n d a n d o
2     a n d a n d o
3         a n d a n d o
4             a n d a n d o
5                 a n d a n d o
```

Boyer-Moore

pat = a n d a n d o

```
   0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
   a s   a n d o r i n h a s   a n d a m   a n d a n d o   a l t o txt
  ---------------------------------------------------------------------------------------
1 a n d a n d o
2       a n d a n d o
3           a n d a n d o
4               a n d a n d o
5                   a n d a n d o
6                       a n d a n d o
```

Boyer-Moore

pat = a n d a n d o

```
   0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
   a s   a n d o r i n h a s   a n d a m   a n d a n d o   a l t o txt
  ---------------------------------------------------------------------------------------
1 a n d a n d o
2       a n d a n d o
3           a n d a n d o
4               a n d a n d o
5                   a n d a n d o
6                       a n d a n d o
7                           a n d a n d o
```

Boyer-Moore

pat = a n d a n d o

```
  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
  a s   a n d o r i n h a s   a n d a m   a n d a n d o   a l t o txt
-------------------------------------------------------------------
1 a n d a n d o
2       a n d a n d o
3           a n d a n d o
4               a n d a n d o
5                   a n d a n d o
6                       a n d a n d o
7                           a n d a n d o
8                               a n d a n d o
```

Conclusões

O consumo de tempo do algoritmo BoyerMoore no pior caso é $O((n - m + 1)m)$.

O consumo de tempo do algoritmo BoyerMoore no melhor caso é $O(n/m)$.

Isto significa que no pior caso o consumo de tempo é essencialmente proporcional a mn e no melhor caso o algoritmo é **sublinear**.

Algoritmo KMP

Examina os caracteres de `txt` um a um, da esquerda para a direita, sem nunca retroceder.

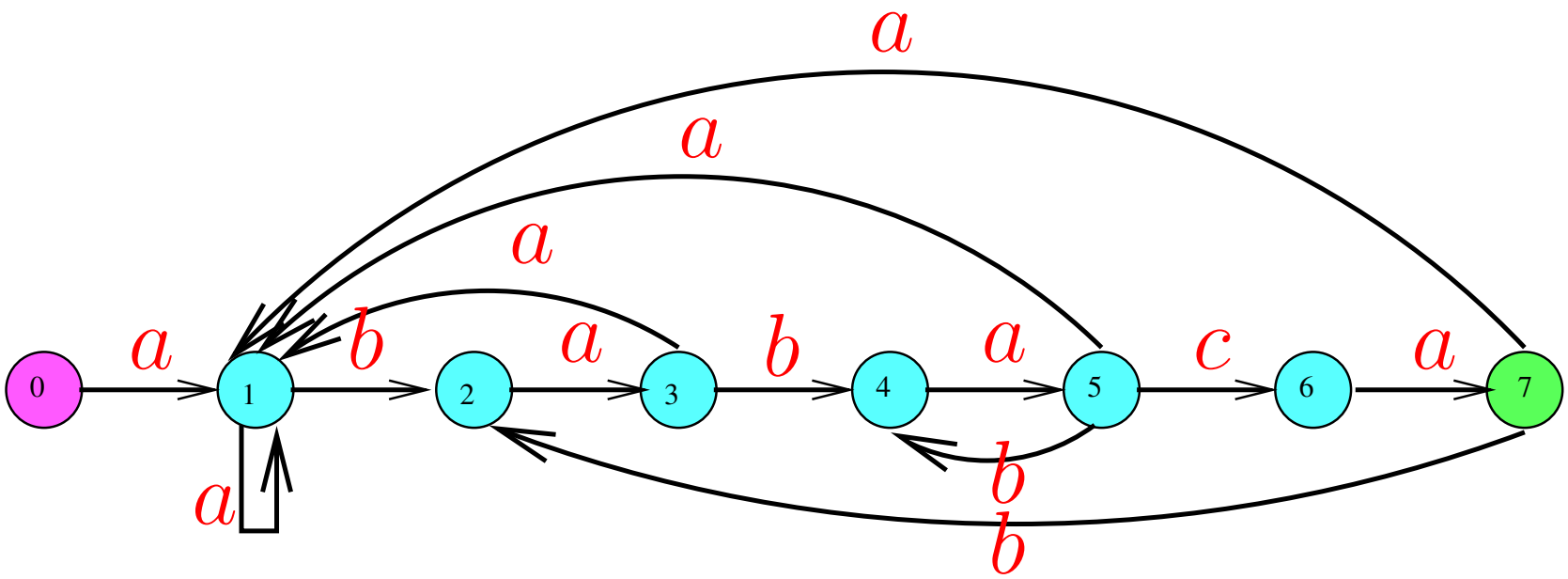
Em cada iteração, o algoritmo sabe qual posição `k` de `pat` deve ser emparelhada com a próxima posição `i+1` de `txt`.

O algoritmo `KMP` usa uma tabela `dfa[][]` que armazena os `índices mágicos k`.

O nome da tabela deriva da expressão *deterministic finite-state automaton*.

O algoritmo KMP simula o funcionamento do autômato de estados.

Autômato de estados determinístico (DFA)



$0 \ldots 7 =$ conjunto de estados

$\Sigma = \{\text{a, b, c}\} =$ alfabeto

$\delta =$ função de transição

0 é estado inicial e 7 é estado final

Exemplo: pat = ABABAC

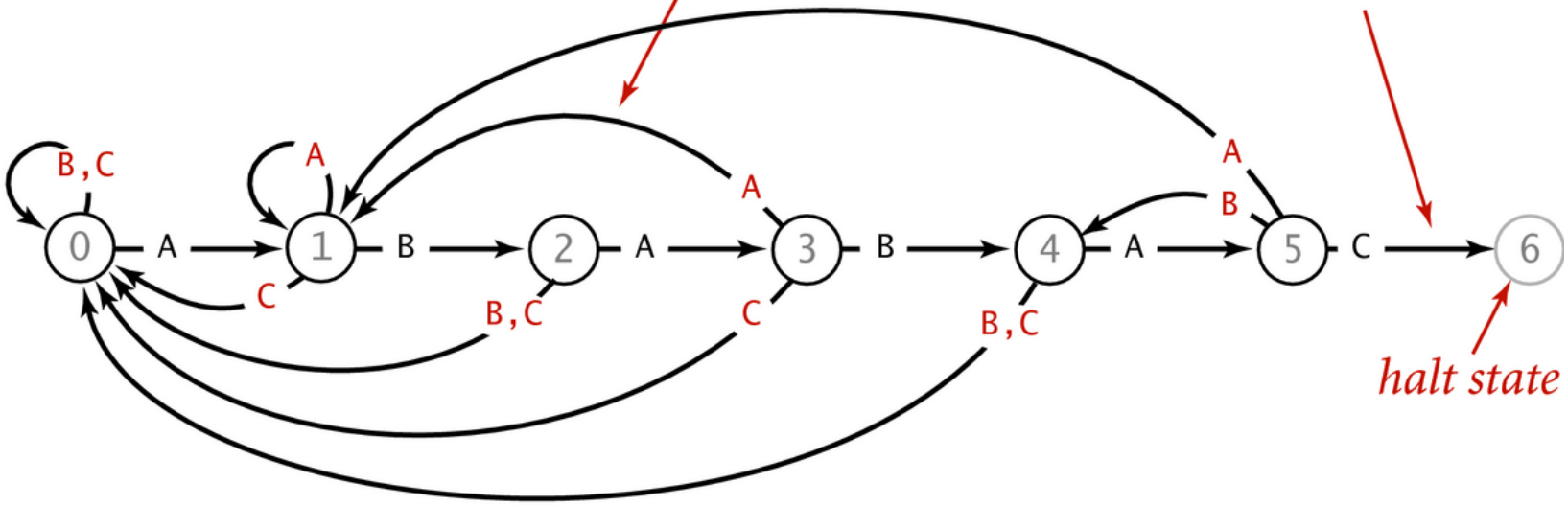
internal representation

	j	0	1	2	3	4	5
pat.charAt(j)		A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5	1
	B	0	2	0	4	0	4
	C	0	0	0	0	0	6

mismatch transition (back up)

match transition (increment)

graphical representation



Algoritmo KMP

Retorna a posição a partir de onde pat ocorre em txt se pat não ocorre em txt retorna n.

```
public int search(String txt) {
   int i, n = txt.length();
   int j, m = pat.length();

   for (i = 0, j = 0; i < n && j < m; i++)
       j = dfa[txt.charAt(i)][j];

   if (j == m) return i - m;
   return n;
}
```

Consumo de tempo

O consumo de tempo do algoritmo KMP é $O(m + n)$.

Proposição. O algoritmo KMP examina não mais que m + n caracteres.

Se levarmos em conta o tamanho do alfabeto, R, o consumo de tempo para construir o DFA é mR.

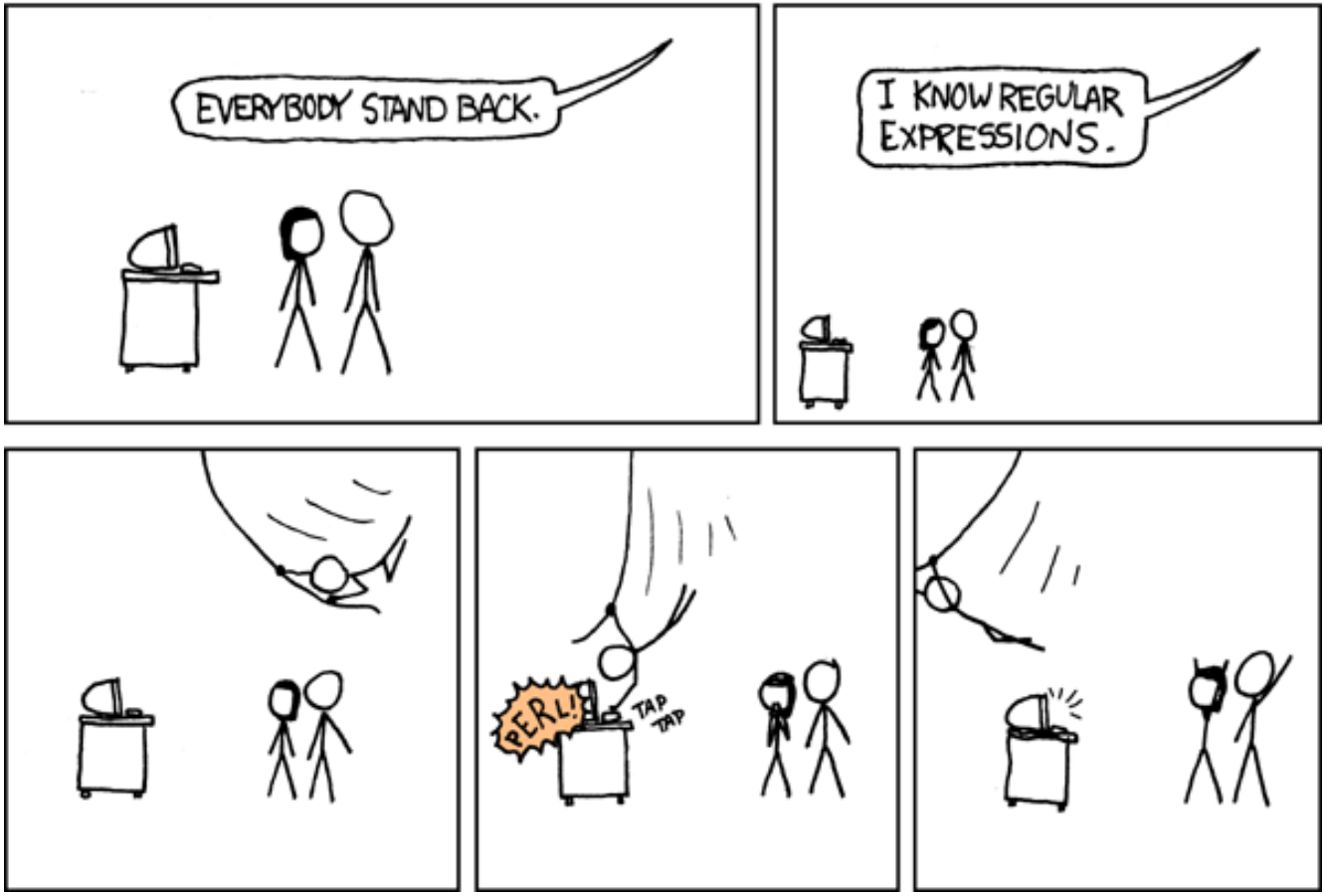
Prato do dia

Que acontece se o padrão não é apenas uma string mas um conjunto de strings descrito por uma **expressão regular** como `A*|(A*BA*BA*)*` ou `((A*B|AC)D)`, por exemplo?

Essa generalização do problema de busca é muito importante. A solução envolve o conceito de autômato de estados não determinístico.

AULA 25

Expressões regulares



Fonte: https://xkcd.com/208/

Referências: Regular expressions (SW), slides (SW), vídeo (SW).

Busca de padrões

Problema: Dado um conjunto `L` de strings, uma string `txt`, encontrar uma (todas) ocorrência(s) de padrões `pat` de `L` em `txt`.

Essa é uma generalização do problema de busca de substring.

O conjunto `L` será uma linguagem regular.

Linguagem regulares, mesmo infinitas, admitem uma representação bem compacta através de uma string que é uma expressão regular.

Uma expressão regular define um conjunto de strings ou padrões sobre um alfabeto.

Expressões regulares

Uma string $\mathtt{re}$ sobre um alfabeto é **expressão regular** se é:

- a string vazia; ou
- a string formada por apenas um caractere/símbolo do alfabeto; ou
- uma string $(\mathtt{re}_1\mathtt{re}_2)$ obtida através da "*concatenação*" de duas expressões regulares $\mathtt{re}_1$ e $\mathtt{re}_2$; ou
- uma string $(\mathtt{re}_1|\mathtt{re}_2)$ obtida através da "*união*" de duas expressões regulares $\mathtt{re}_1$ e $\mathtt{re}_2$;
- uma string $(\mathtt{re}*)$ obtida através do "*operador fecho de Kleene*".

Exemplos

- concatenação: se `ABC` e `DEF` são padrões (`ABCDEF`) representa o padrão `ABCDEF`;
- *ou*: `((((A | E) | I) | O) | U)` ou simplesmente `A | E | I | O | U` representa os padrões *vogais*;
- fecho: `(A(B*))` ou simplesmente `AB*` representa todos os padrões `A`, `AB`, `ABB`, `ABBB`, ...

Parênteses e precedência

Os parênteses em uma expressão regular podem ser omitidos.

Se isso ocorre, o cálculo é feito na ordem da precedência:

- estrela/fecho;
- concatenação;
- união/ou;

Exemplos

- $\mathtt{A}(\mathtt{B}|\mathtt{C})\mathtt{D}$ representa $\mathtt{ABD}$ e $\mathtt{ACD}$;
- $\mathtt{A}^*|(\mathtt{AB}^*\mathtt{B}(\mathtt{C}|\mathtt{A}))^*$ representa
  ϵ, $\mathtt{A}$, $\mathtt{AA}$, $\mathtt{AAA}\ldots$
  $\mathtt{ABC}$, $\mathtt{ABBC}$, $\mathtt{ABCABC}\ldots$
  $\mathtt{ABA}$, $\mathtt{ABBA}$, $\mathtt{ABAABA}\ldots$
  $\mathtt{ABA}$, $\mathtt{ABBA}$, $\mathtt{ABCABA}\ldots$

Abreviaturas

É conveniente utilizarmos abreviaturas como:

- "·": representa qualquer caractere, AB·BA representa ABABA, ABBBA, ABCBA, ...
- "+": fecho um uma ou mais cópias, A+B representa AB, AAB, AAAB, AAAAB, ...
- "?": zero ou uma cópia, (AB)?C* representa C, CC, CCC, ...ABC, ABCC, ABCCC, ...
- {k}: k cópias, (AB){3} representa ABABAB
- [ ]: conjunto , [AEIOU]* representa todos os padrões de vogais.

E muitas mais ...

Busca de padrões

Problema: Dada uma expressão regular `regexp` e uma string `txt`, encontrar uma (todas) ocorrência(s) de `padrões pat` de `regexp` em `txt`.

Teorema de Kleene

Para toda `regexp` existe um `dfa` que reconhece as strings representadas pela `regexp`.

Para todo `dfa` existe uma `regexp` representa as strings reconhecidas pelo `dfa`.

Plano

Proceder como no algoritmo KMP, dadas as strings regexp e txt:

- construir um autômato dfa que reconhece as strings em regexp;
- examinar os caracteres de txt andando no autômato.

Dificuldade: o autômato dfa pode ter um número exponencial de estados no tamanho m da regexp.

Solução

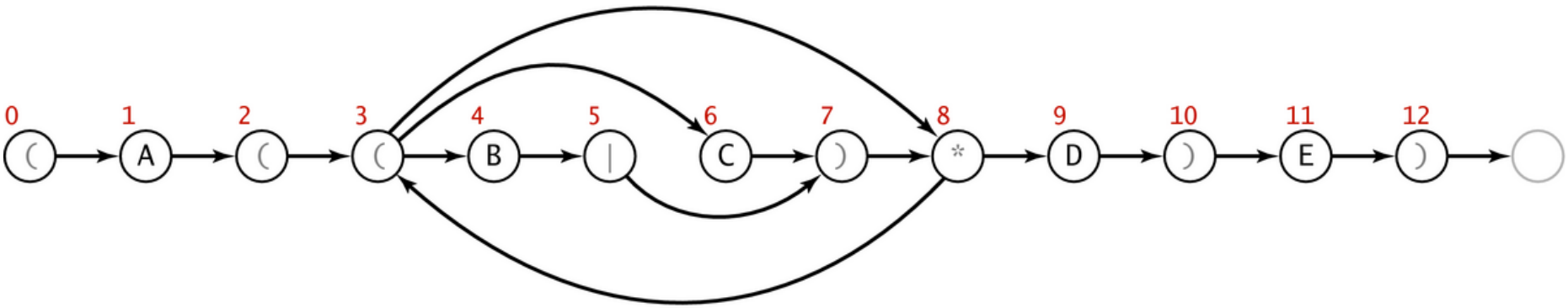
Utilizar outro tipo de autômato.

Substituir um `DFA` por um `NFA` (*nondeterministic* finite-state automata).

Teorema de Kleene
Para toda `regexp` existe um `nfa` que reconhece as strings representadas por `regexp`.
Para todo `nfa` existe uma `regexp` representa as strings reconhecidas por `dfa`.

Boa notícia: o autômato `nfa` tem `m`+1 estados.

NFA: (A ((B | C) * D) E)



One-state-per-character NFA corresponding to the pattern (A ((B | C) * D) E)

regexp para nfa

Por simplicidade, o algoritmo supõe que o primeiro caractere da regexp é '(' e o útimo é ')'.

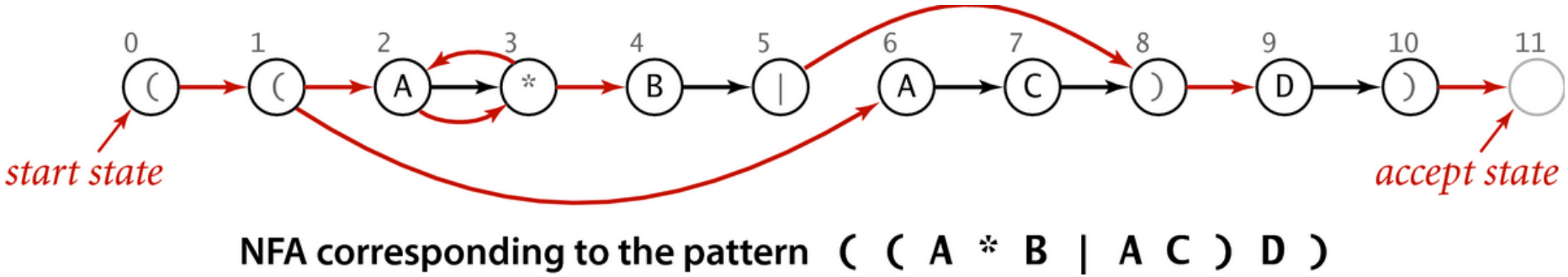
nfa tem um estado para cada caractere na regexp.

Arcos vermelho correspondem a ϵ-**transições**: mudamos do estado sem soletrar caractere de txt.

Arcos pretos correspondem a transições que mudamos de estado após soletrar um caractere de txt; como em um dfa.

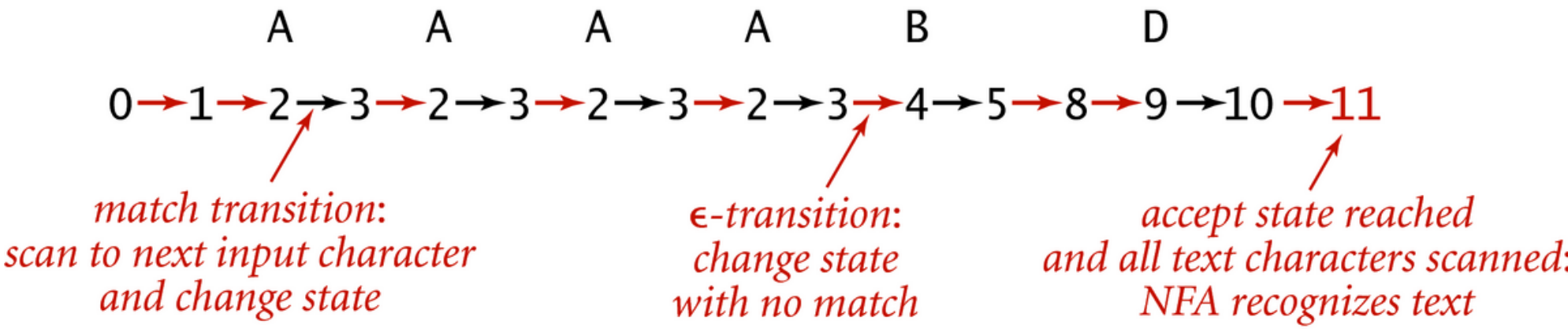
Aceita se **existe** uma sequência de transições que, após soletrar todos os caracteres em txt, que termina em um estado de aceite.

NFA: ((A * B | A C) D)



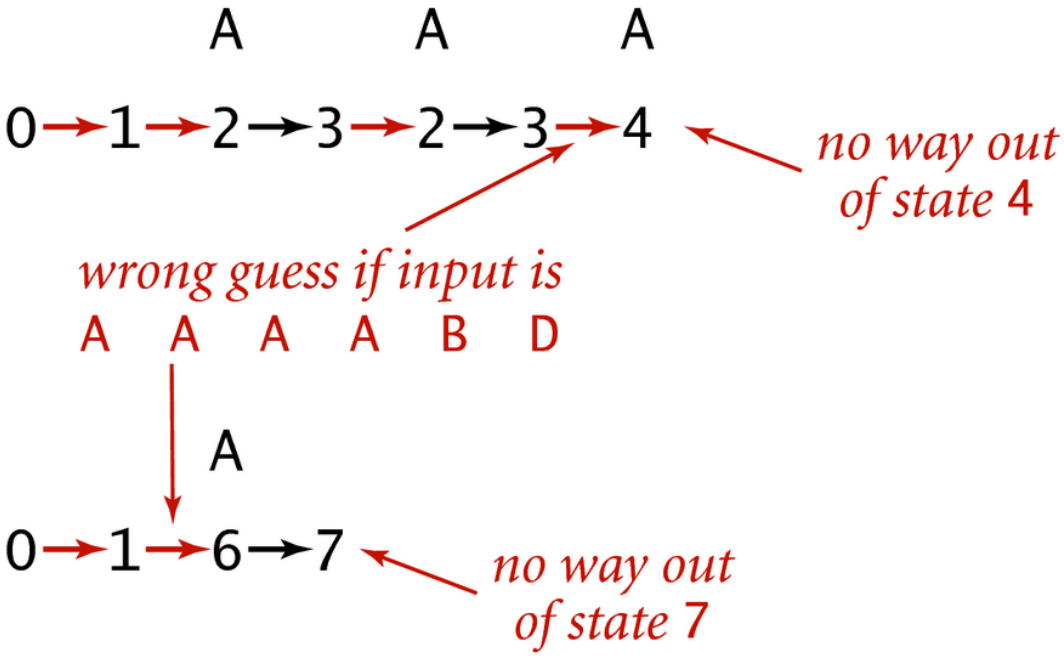
NFA corresponding to the pattern ((A * B | A C) D)

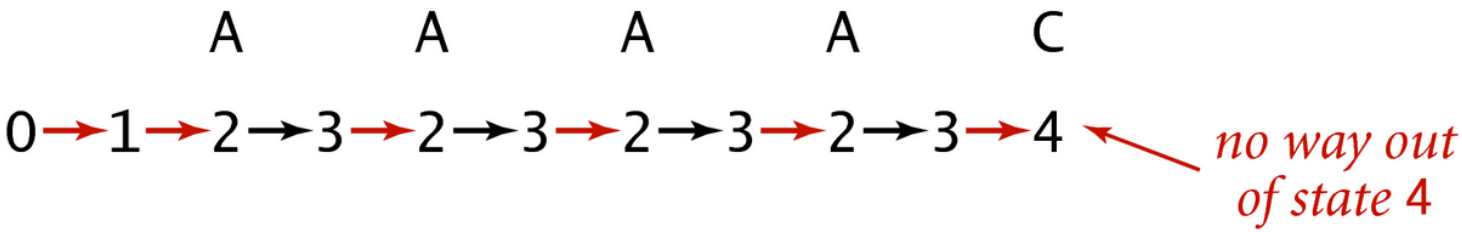
NFA: soletrando



Finding a pattern with ((A * B | A C) D) NFA

NFA: soletrando





Stalling sequences for ((A * B | A C) D) NFA

NFA: mais estrutura

Um estado para cada caractere de `regexp`.

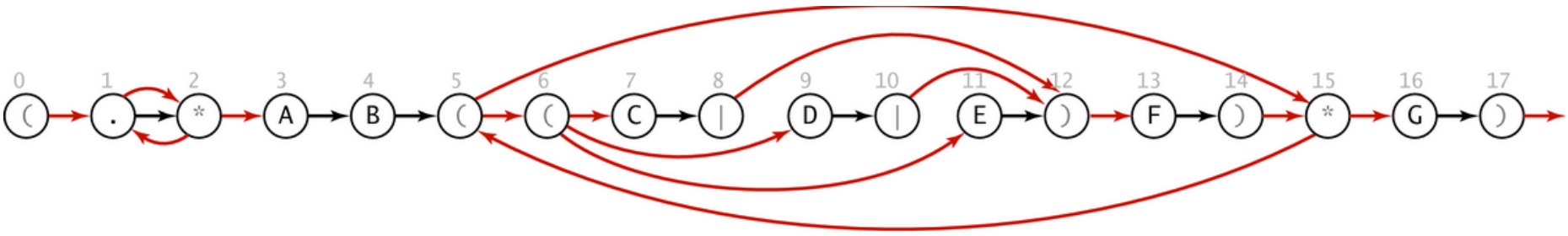
Estados correspondentes a letras tem apenas um **arco preto** saindo para o estado seguinte.

Estados correspondents a `(`, `*`, `|`, `)` têm apenas arcos vermelhos saindo.

Estados têm no máximo um **arco preto** entrando.

Rejeita se **não existe** uma sequência de transições, após soletrar todos os caracteres em `txt`, que termina em um estado de aceite.

NFA: (.* A B ((C | D | E) F) * G)



NFA corresponding to the pattern (. * A B ((C | D | E) F) * G)

Plano

Proceder como no algoritmo KMP, dadas as strings `regexp` e `txt`:

- construir um autômato `nfa` que reconhece as strings em `regexp`;
- examinar os caracteres de `txt` andando no autômato.

Como determinar aceitação de uma string?

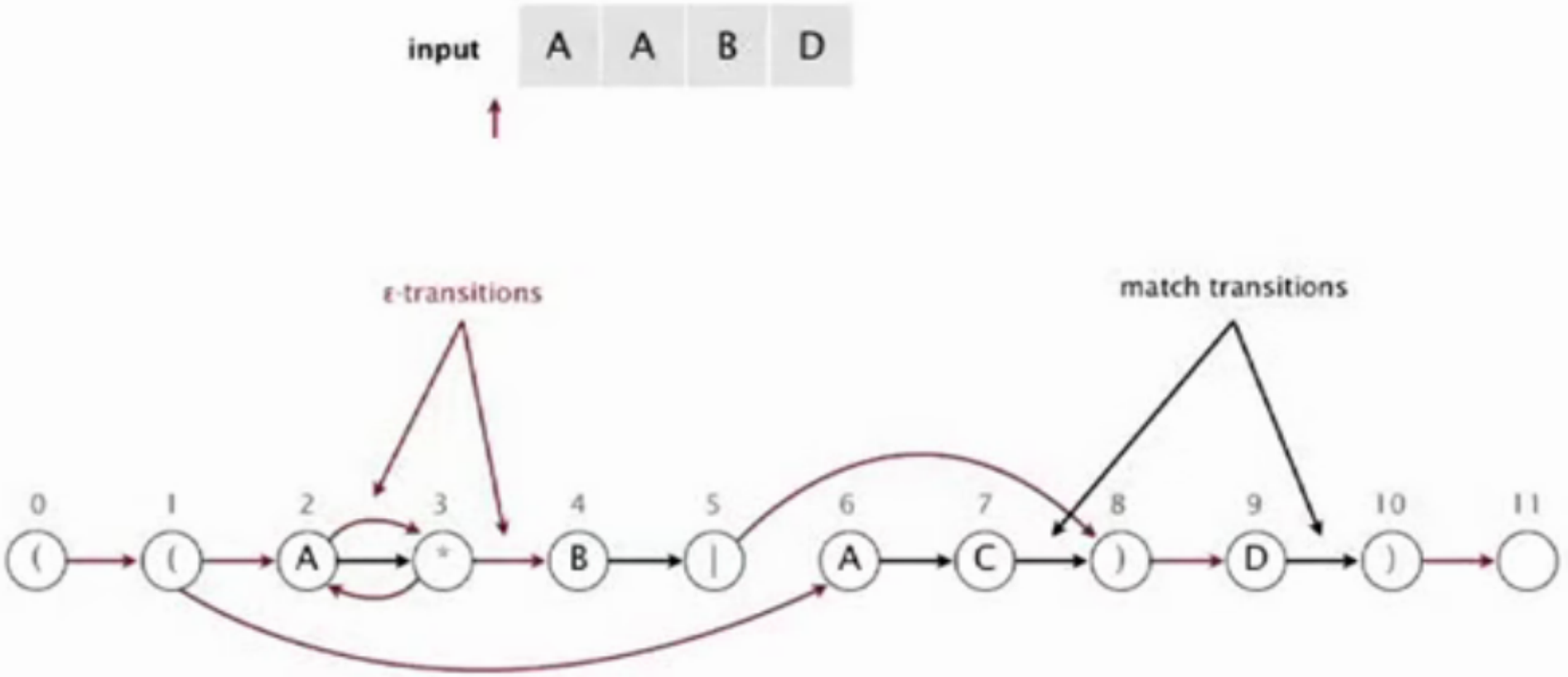
`DFA`$\Rightarrow$ soletrar `txt`, aplicando **transições pretas**, fácil

`NFA`$\Rightarrow$ podemos aplicar várias transições...

Para simular a `NFA` sistematicamente consideramos **todas(!)** as transições possíveis.

NFA simulation demo

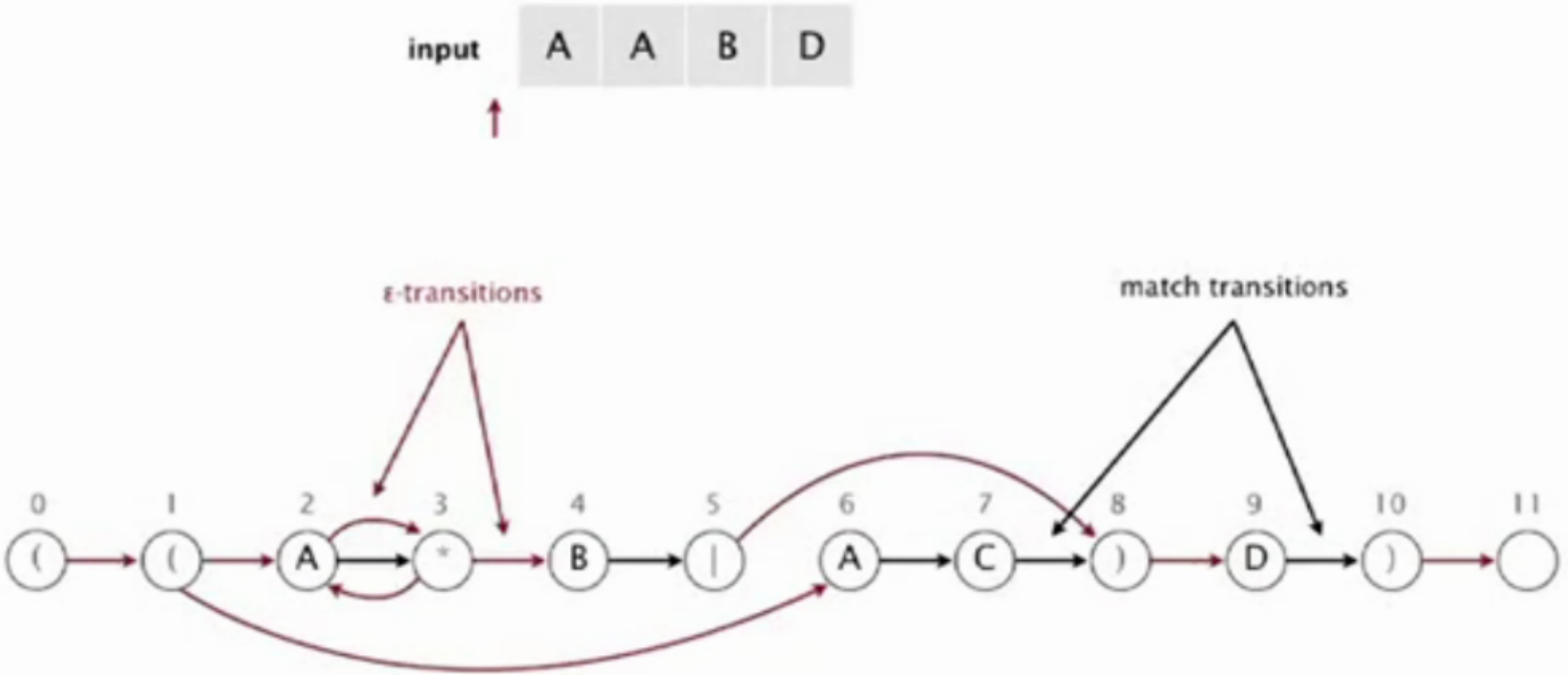
Goal. Check whether input matches pattern.



NFA corresponding to the pattern ((A * B | A C) D)

NFA simulation demo

Goal. Check whether input matches pattern.

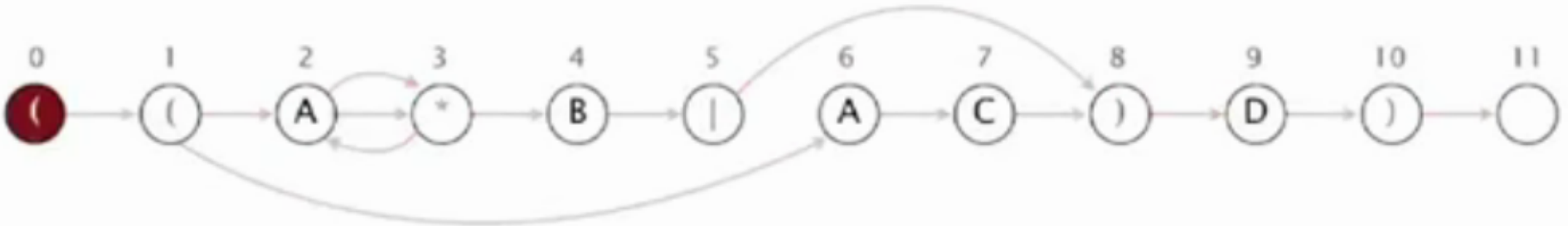


NFA corresponding to the pattern ((A * B | A C) D)

NFA simulation demo

Read next input character.

- Find states reachable by match transitions.
- Find states reachable by ε-transitions

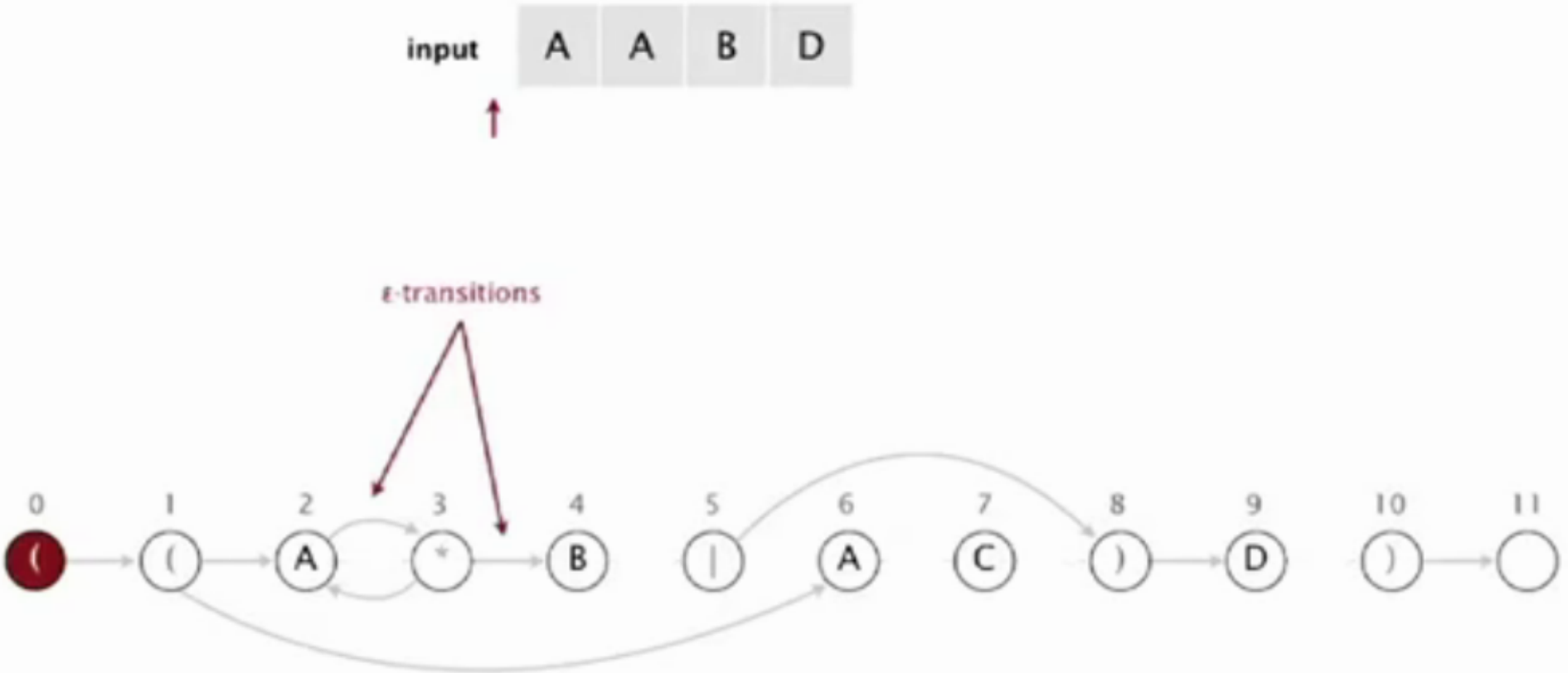


set of states reachable from start: 0

NFA simulation demo

Read next input character.

- Find states reachable by match transitions.
- Find states reachable by ε-transitions

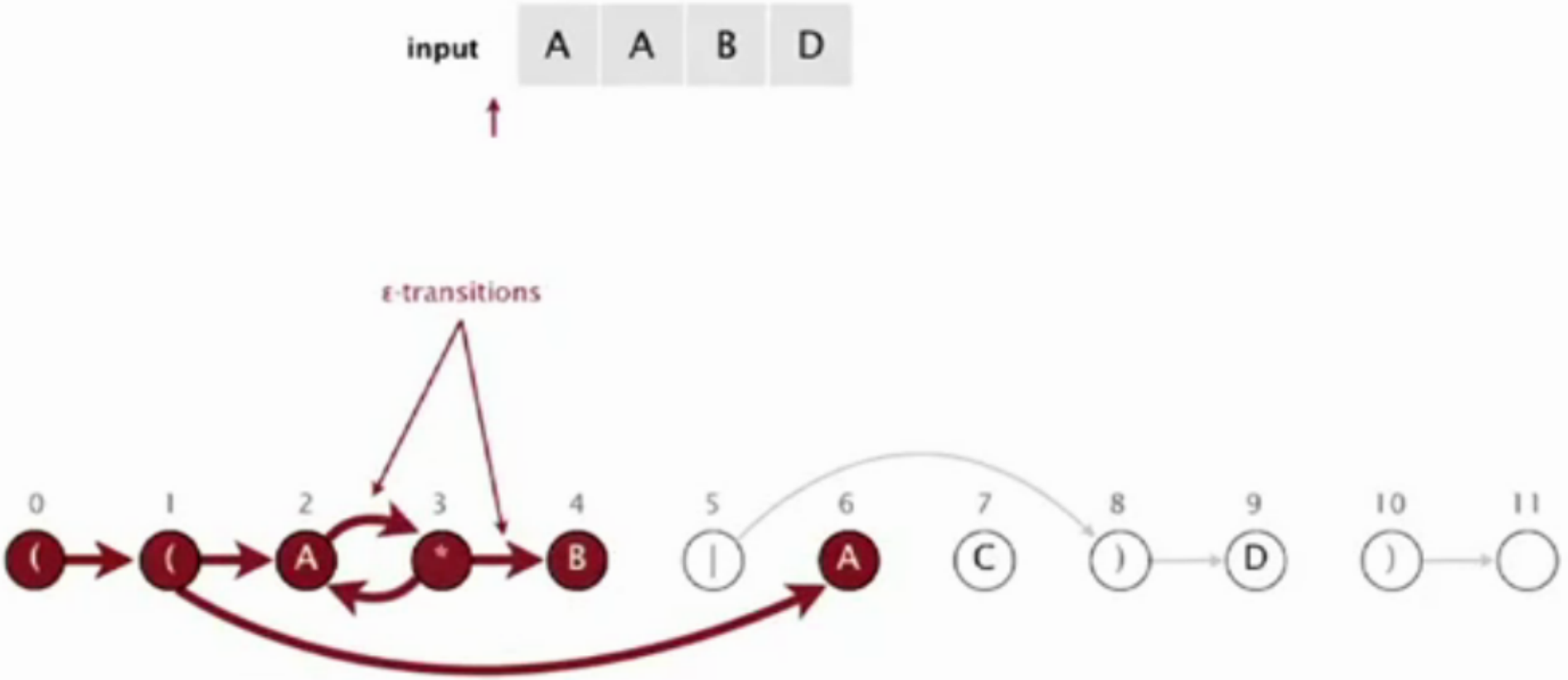


set of states reachable via ε-transitions from start

NFA simulation demo

Read next input character.

- Find states reachable by match transitions.
- Find states reachable by ε-transitions

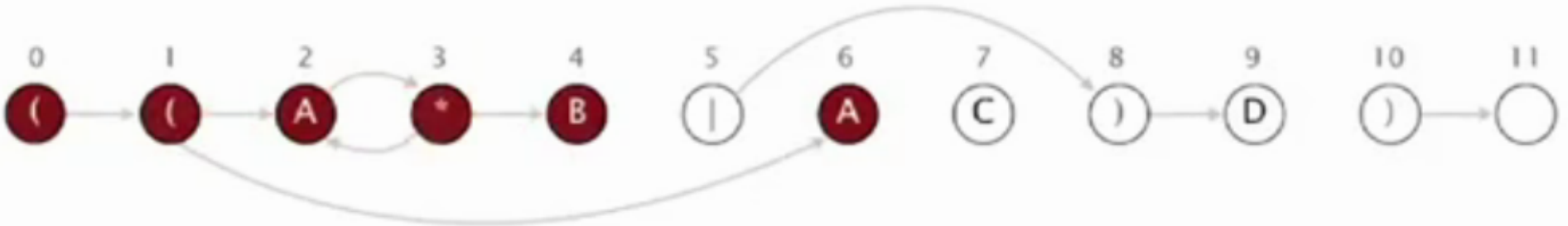


set of states reachable via ε-transitions from start

NFA simulation demo

Read next input character.

- Find states reachable by match transitions.
- Find states reachable by ε-transitions

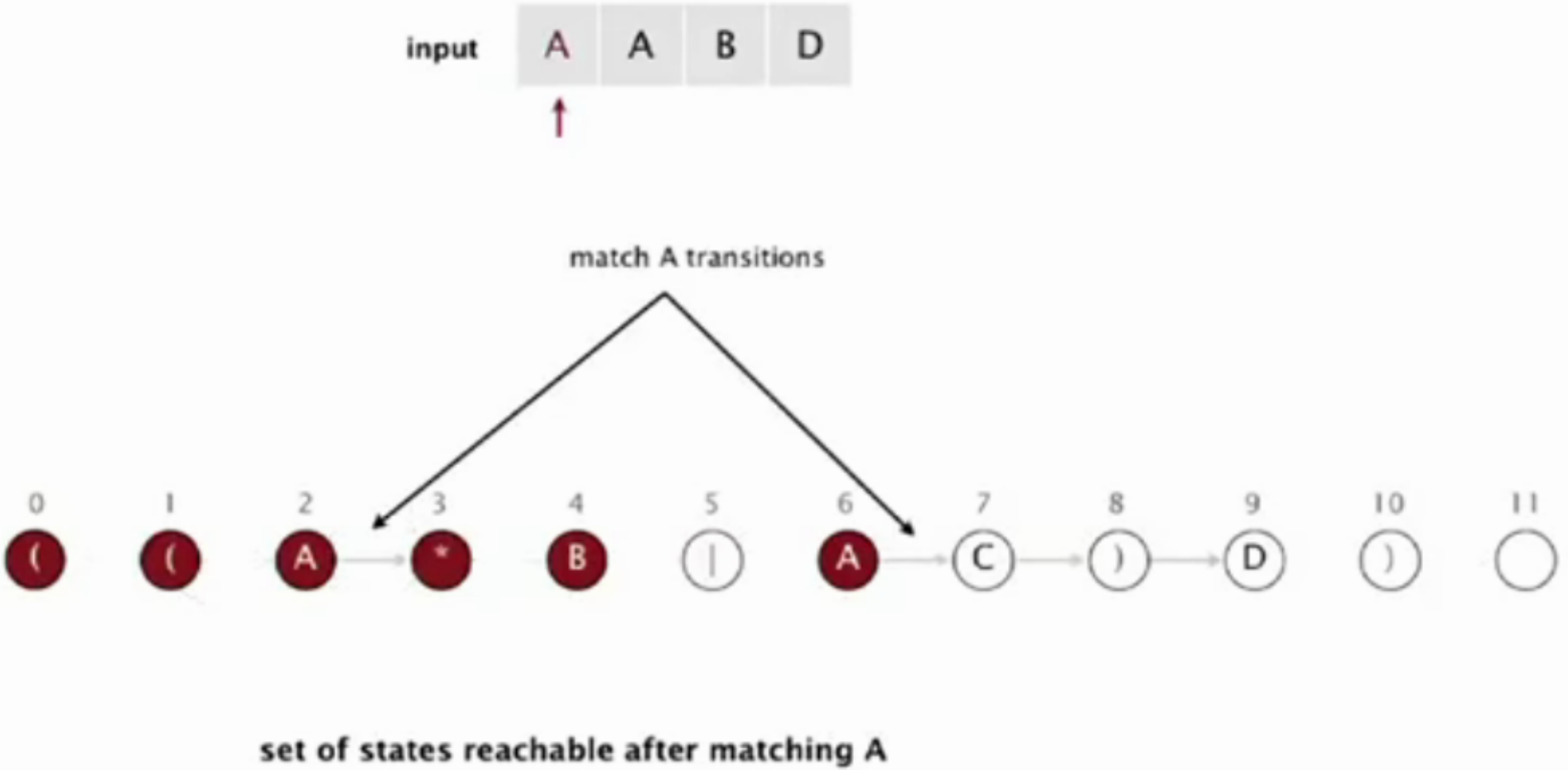


set of states reachable via ε-transitions from start : { 0, 1, 2, 3, 4, 6 }

NFA simulation demo

Read next input character.

- Find states reachable by match transitions.
- Find states reachable by ε-transitions

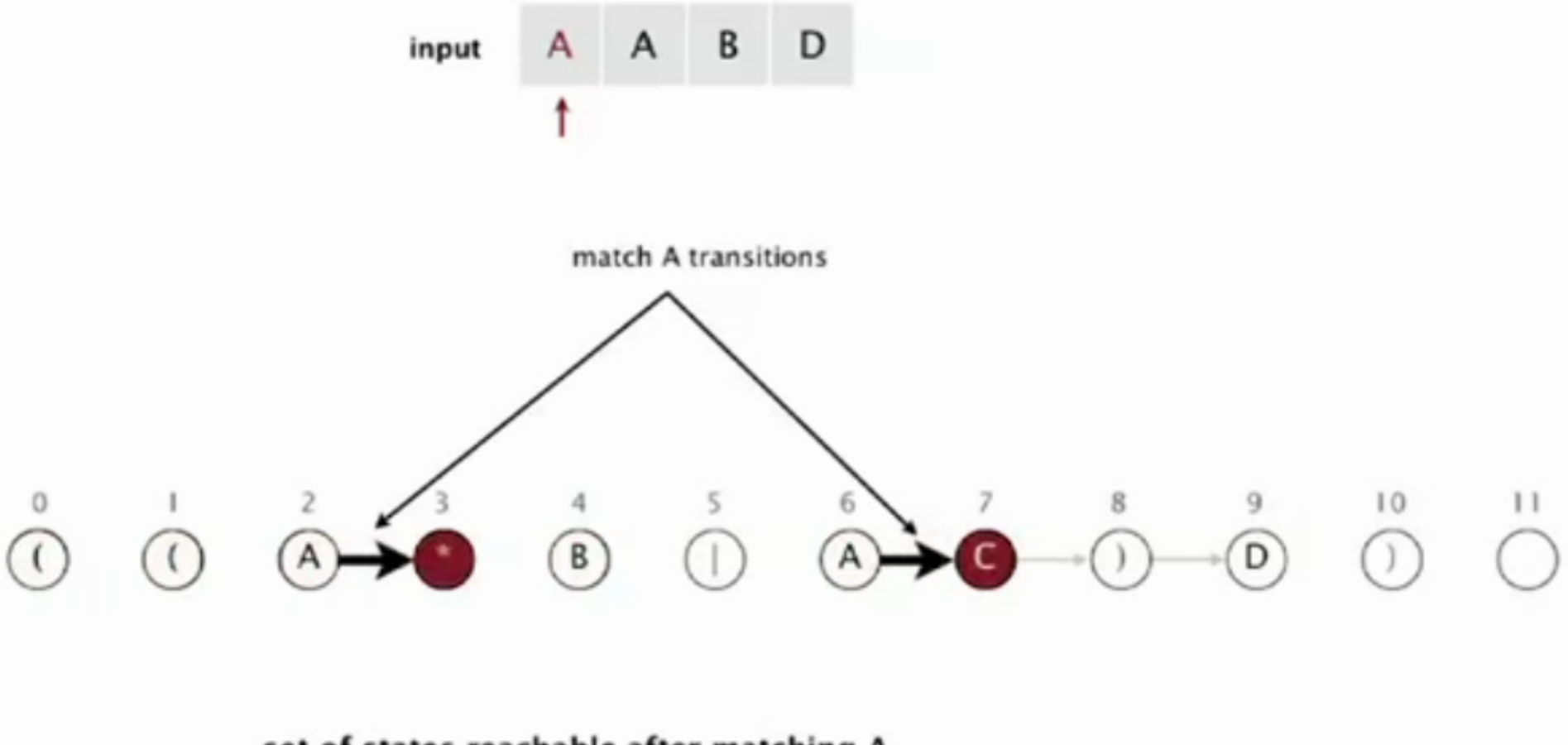


set of states reachable after matching A

NFA simulation demo

Read next input character.

- Find states reachable by match transitions.
- Find states reachable by ε-transitions

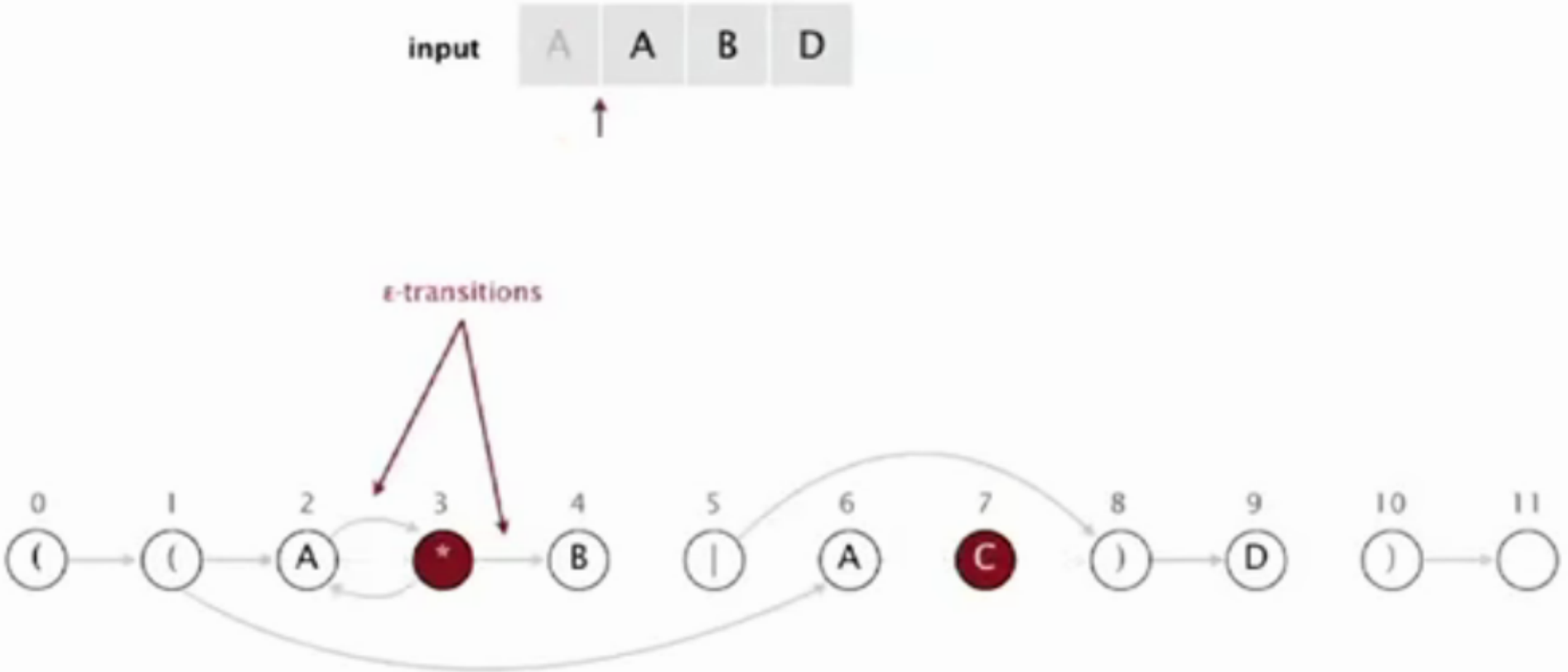


set of states reachable after matching A

NFA simulation demo

Read next input character.

- Find states reachable by match transitions.
- Find states reachable by ε-transitions

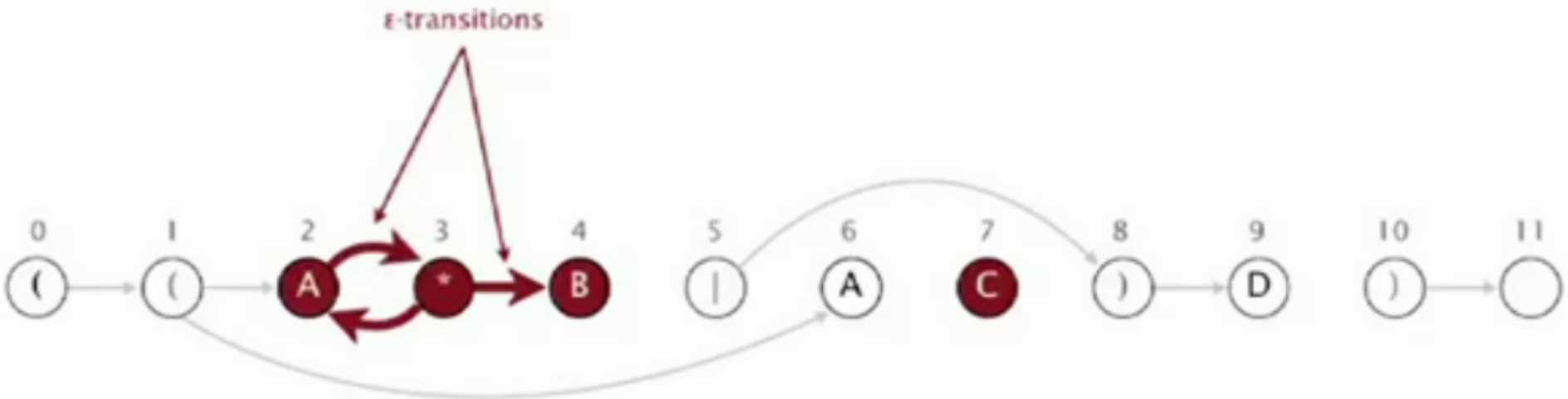


set of states reachable via ε-transitions after matching A

NFA simulation demo

Read next input character.

- Find states reachable by match transitions.
- Find states reachable by ε-transitions

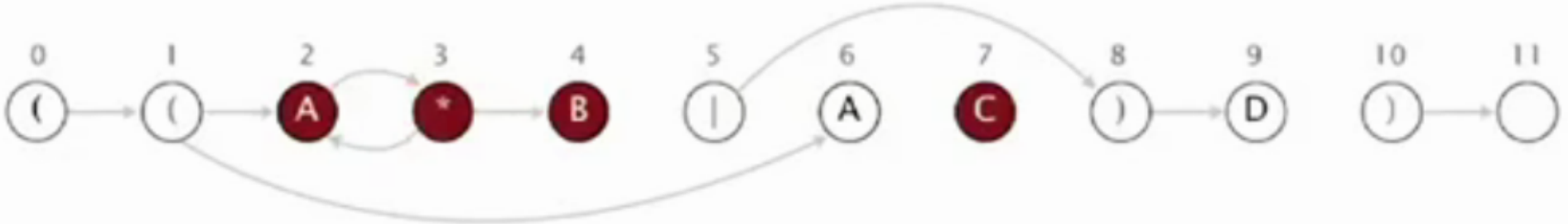


set of states reachable via ε-transitions after matching A

NFA simulation demo

Read next input character.

- Find states reachable by match transitions.
- Find states reachable by ε-transitions

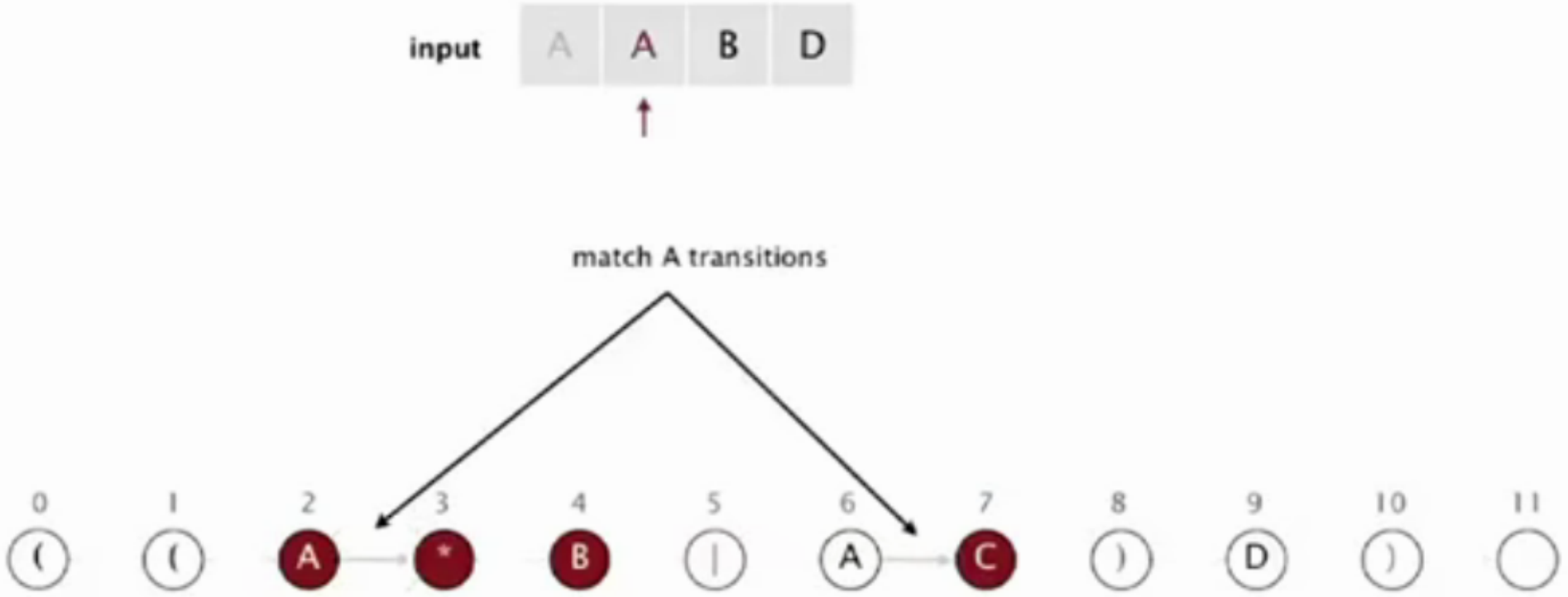


set of states reachable via ε-transitions after matching A : { 2, 3, 4, 7 }

NFA simulation demo

Read next input character.

- Find states reachable by match transitions.
- Find states reachable by ε-transitions



set of states reachable after matching A A

NFA simulation demo

Read next input character.

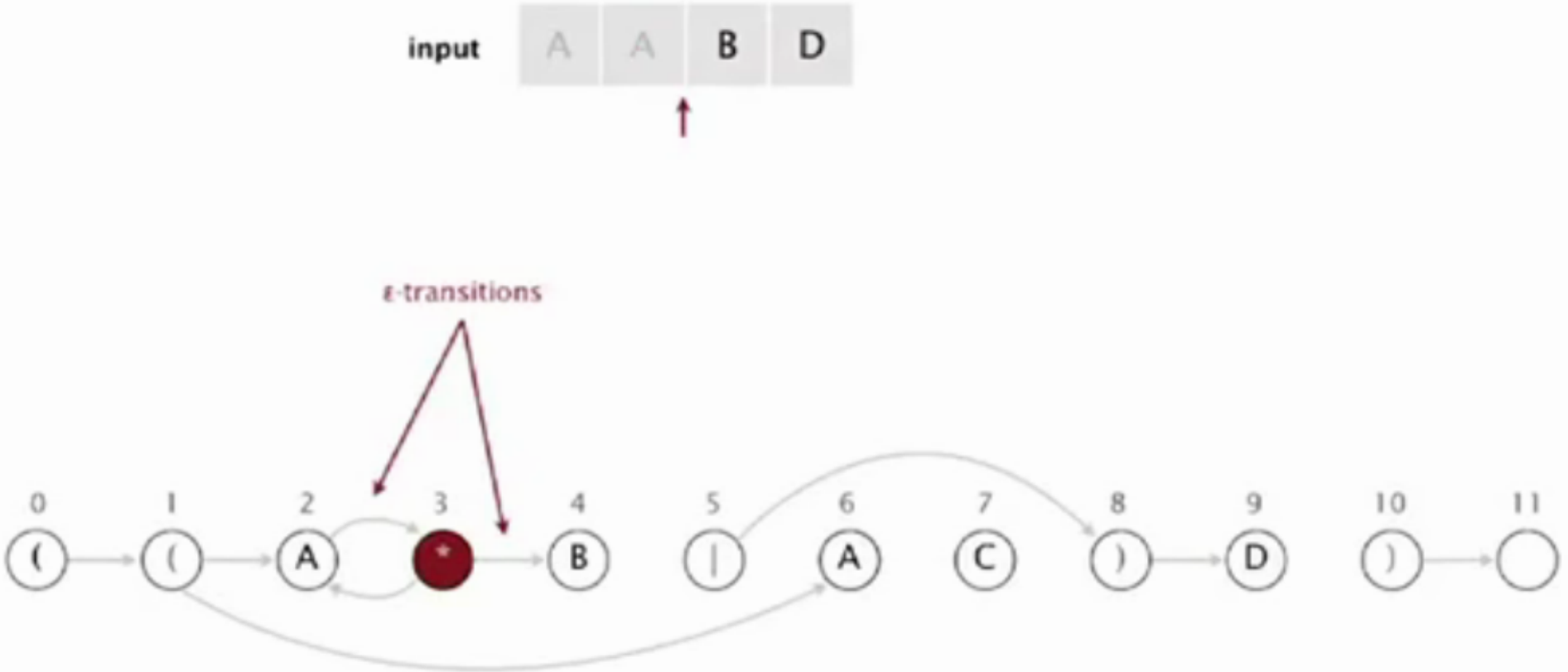
- Find states reachable by match transitions.
- Find states reachable by ε-transitions

set of states reachable after matching A A : { 3 }

NFA simulation demo

Read next input character.

- Find states reachable by match transitions.
- Find states reachable by ε-transitions

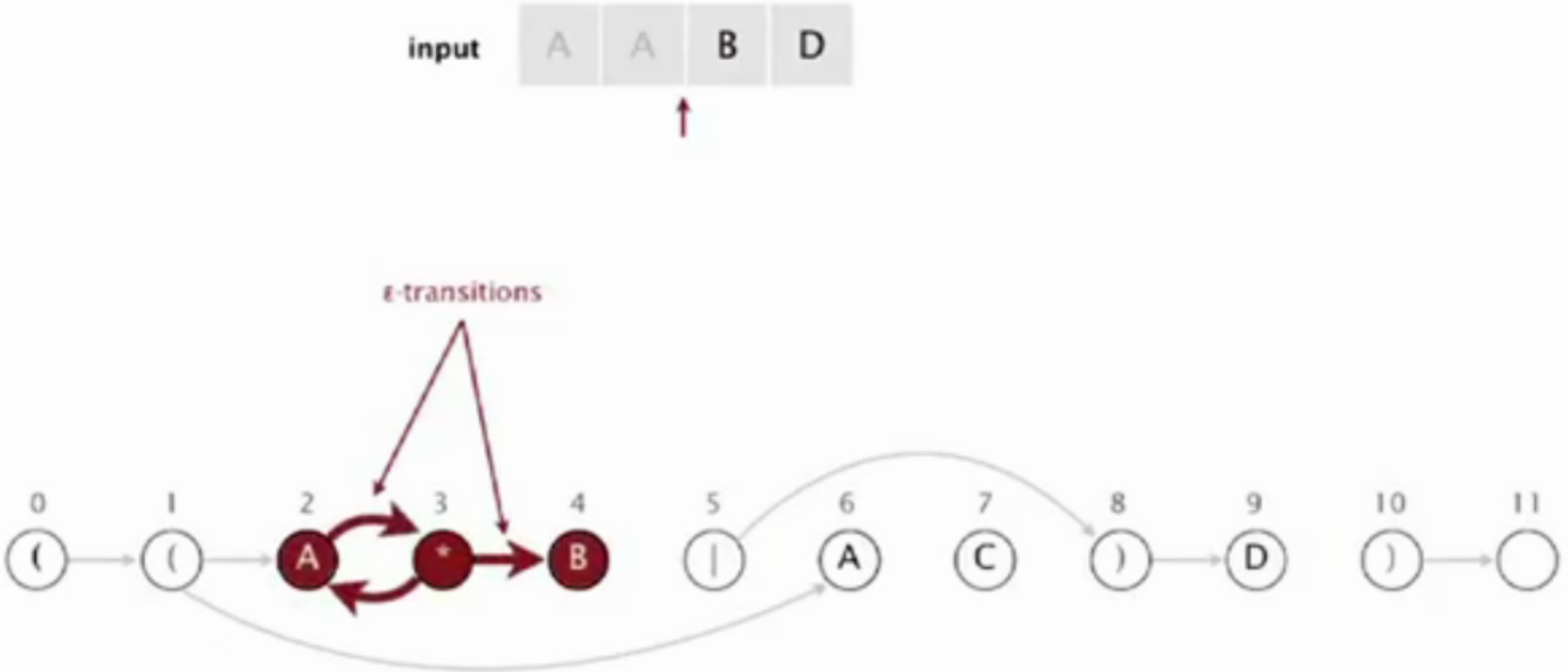


set of states reachable via ε-transitions after matching A A

NFA simulation demo

Read next input character.

- Find states reachable by match transitions.
- Find states reachable by ε-transitions

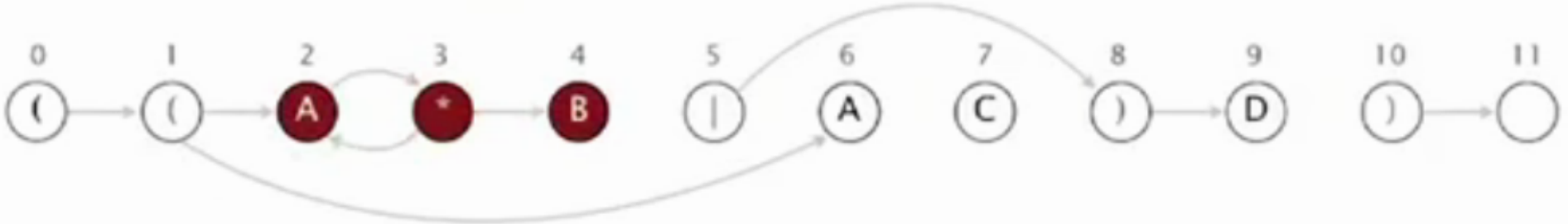


set of states reachable via ε-transitions after matching A A

NFA simulation demo

Read next input character.

- Find states reachable by match transitions.
- Find states reachable by ε-transitions

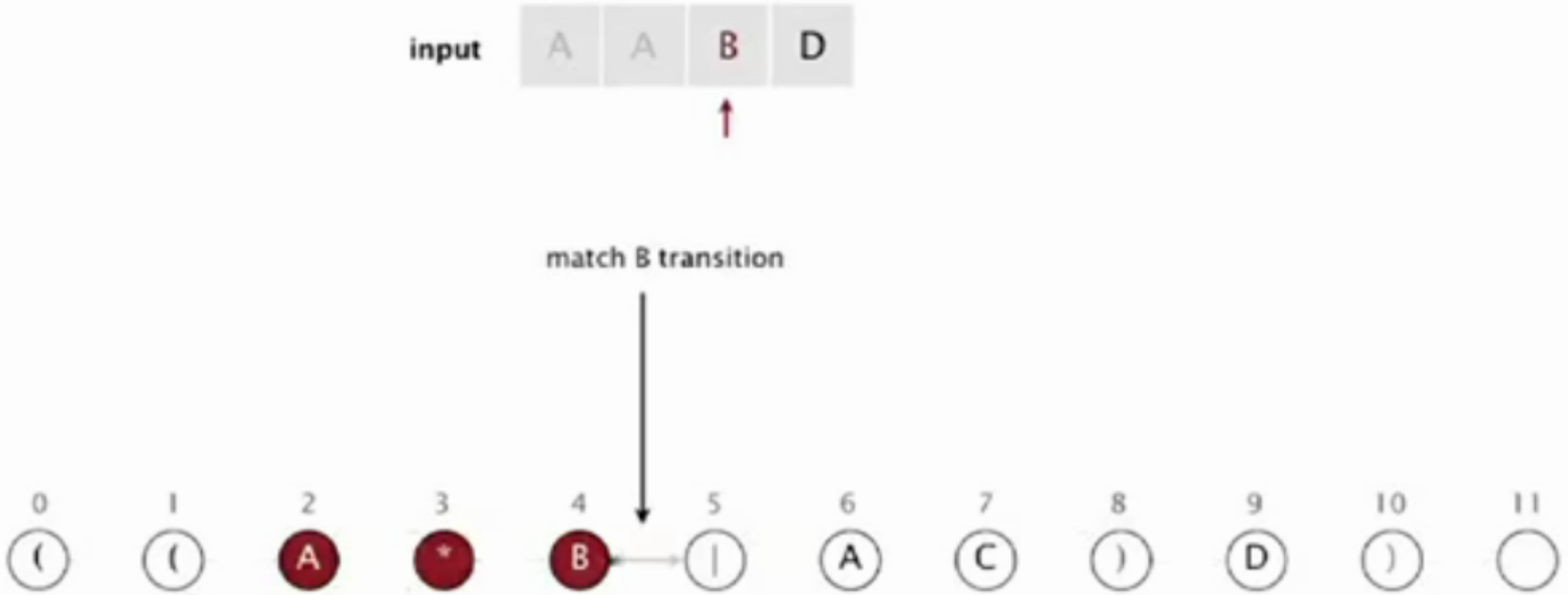


set of states reachable via ε-transitions after matching A A : { 2, 3, 4 }

NFA simulation demo

Read next input character.

- Find states reachable by match transitions.
- Find states reachable by ε-transitions

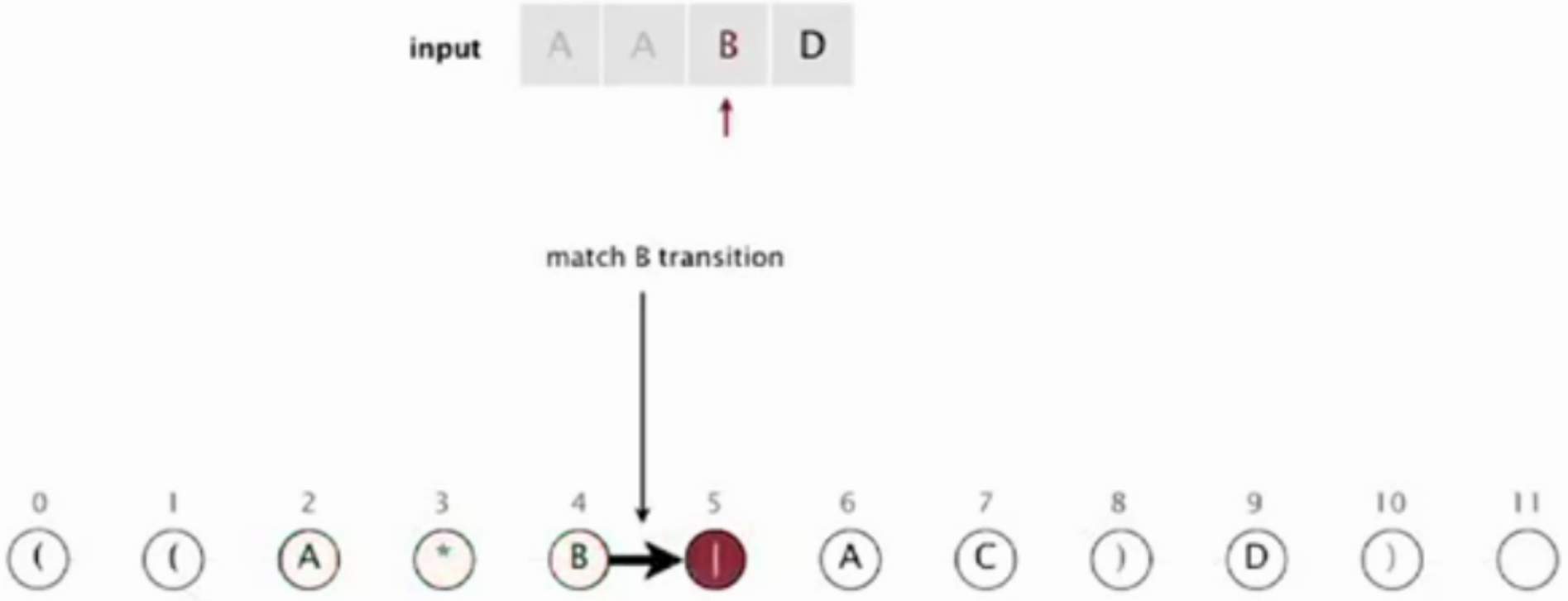


set of states reachable after matching A A B

NFA simulation demo

Read next input character.

- Find states reachable by match transitions.
- Find states reachable by ε-transitions

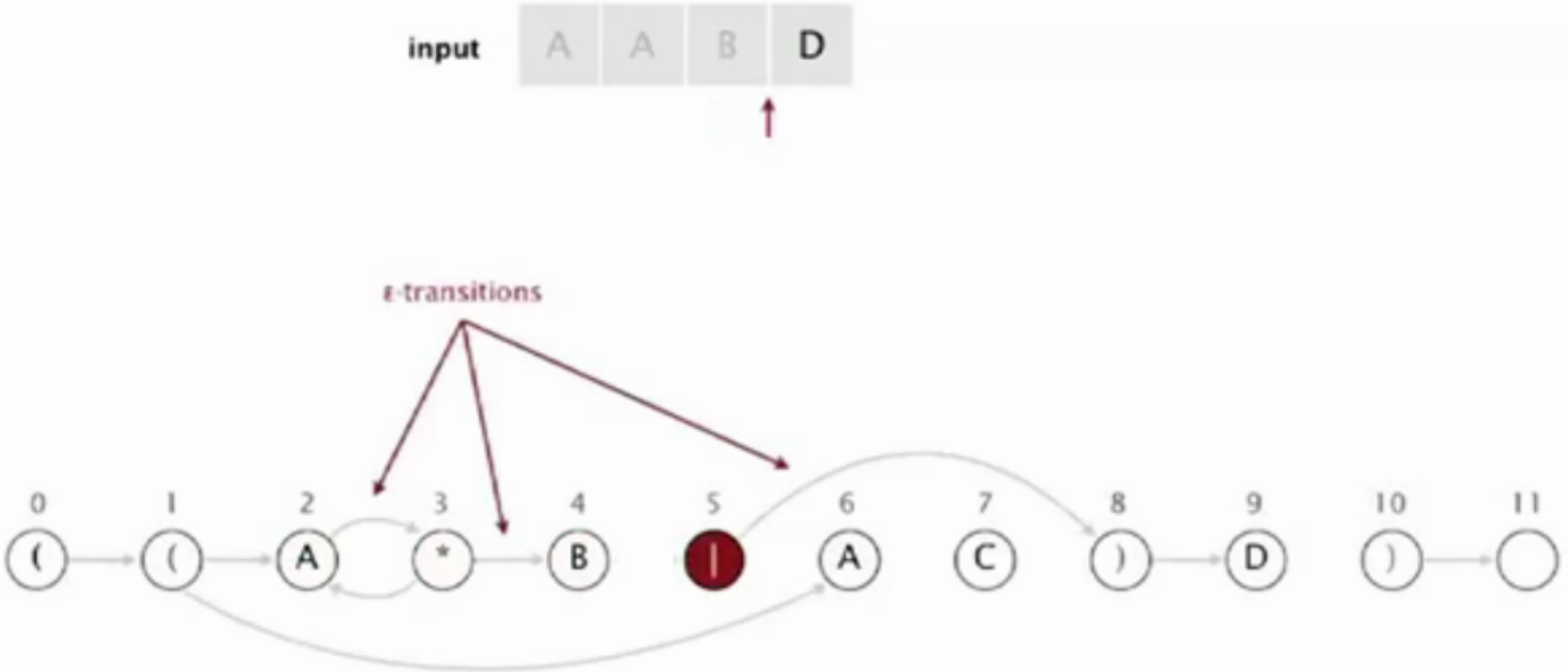


set of states reachable after matching A A B

NFA simulation demo

Read next input character.

- Find states reachable by match transitions.
- Find states reachable by ε-transitions

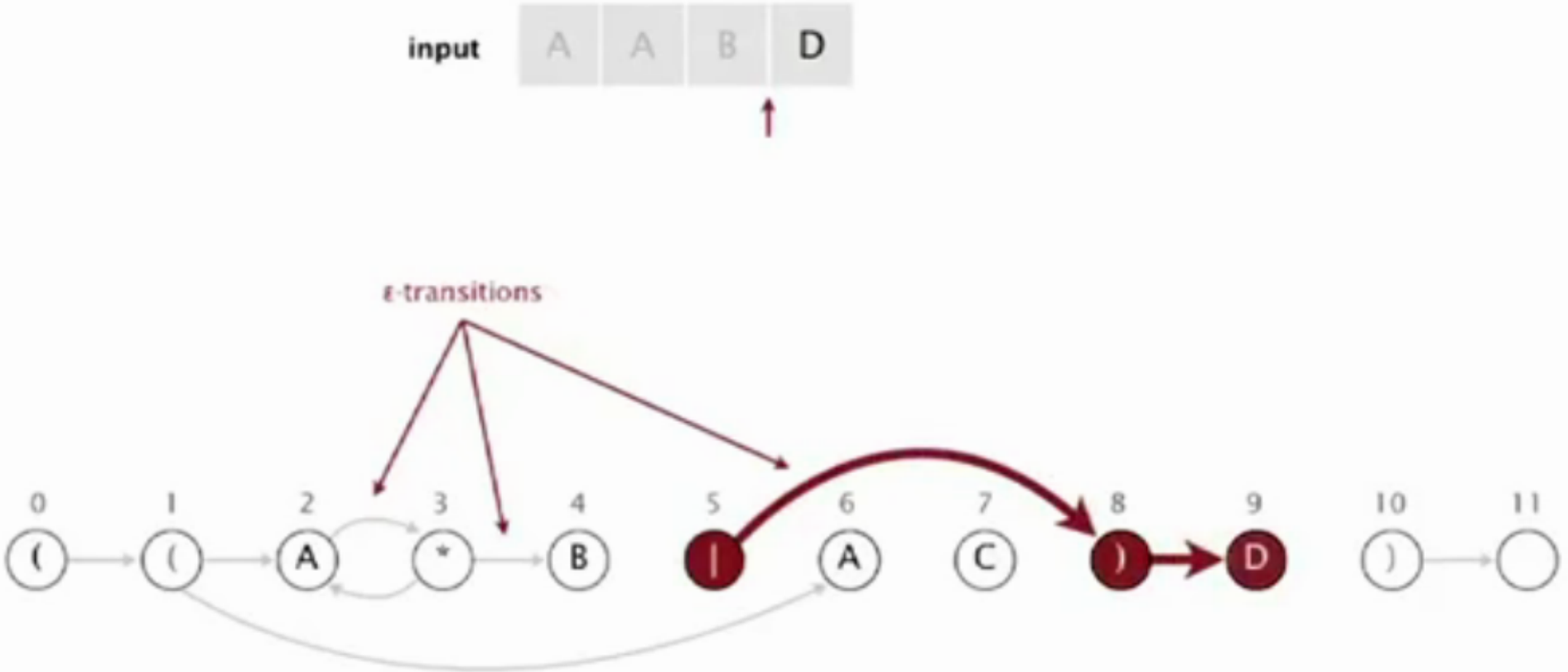


set of states reachable via ε-transitions after matching A A B

NFA simulation demo

Read next input character.

- Find states reachable by match transitions.
- Find states reachable by ε-transitions

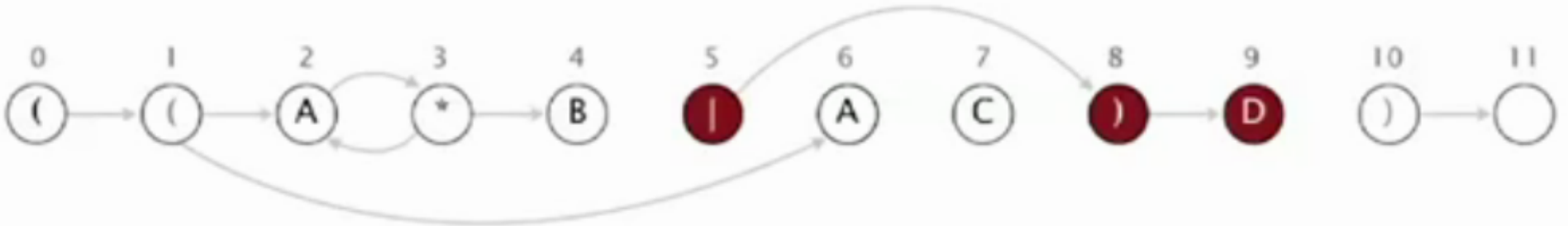


set of states reachable via ε-transitions after matching A A B

NFA simulation demo

Read next input character.

- Find states reachable by match transitions.
- Find states reachable by ε-transitions



set of states reachable via ε-transitions after matching A A B : { 5, 8, 9 }

NFA simulation demo

Read next input character.

- Find states reachable by match transitions.
- Find states reachable by ε-transitions

set of states reachable after matching A A B D

NFA simulation demo

Read next input character.

- Find states reachable by match transitions.
- Find states reachable by ε-transitions

set of states reachable after matching A A B D

NFA simulation demo

Read next input character.

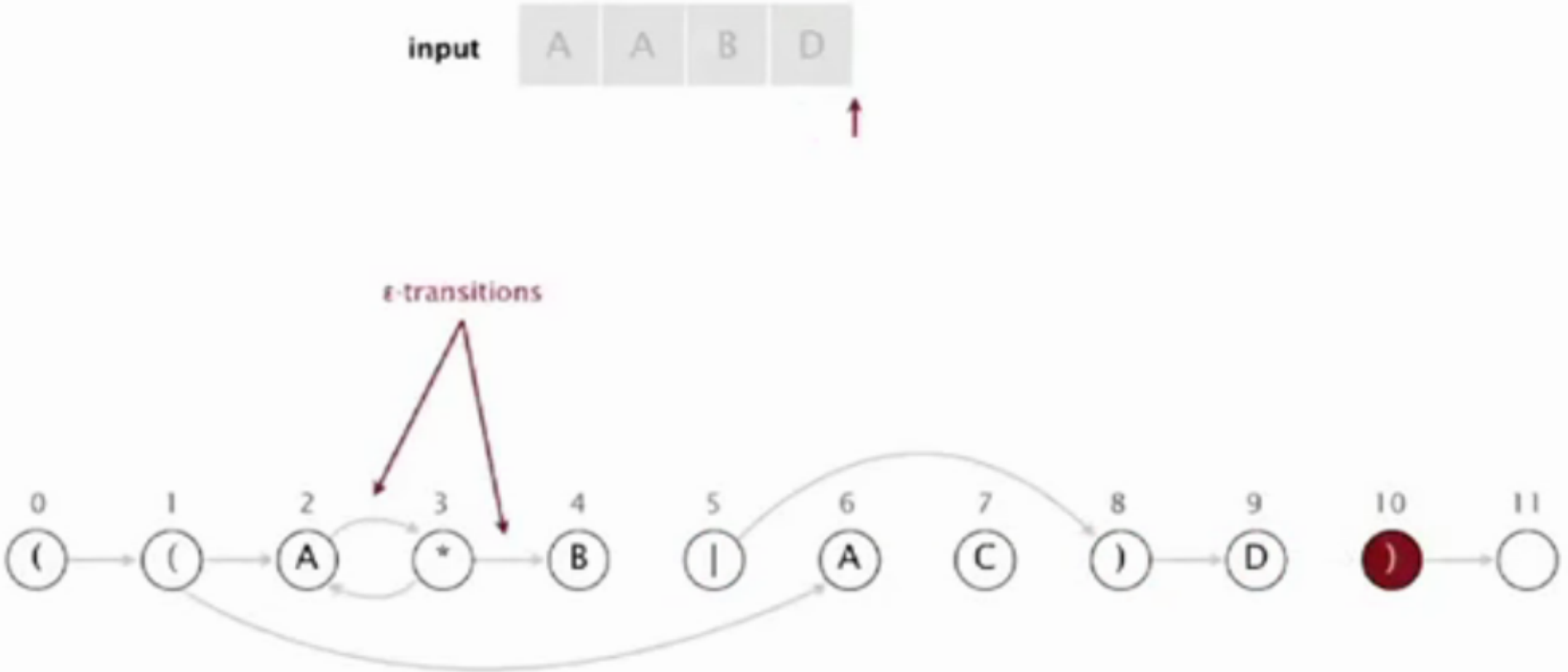
- Find states reachable by match transitions.
- Find states reachable by ε-transitions

set of states reachable after matching A A B D : { 10 }

NFA simulation demo

Read next input character.

- Find states reachable by match transitions.
- Find states reachable by ε-transitions

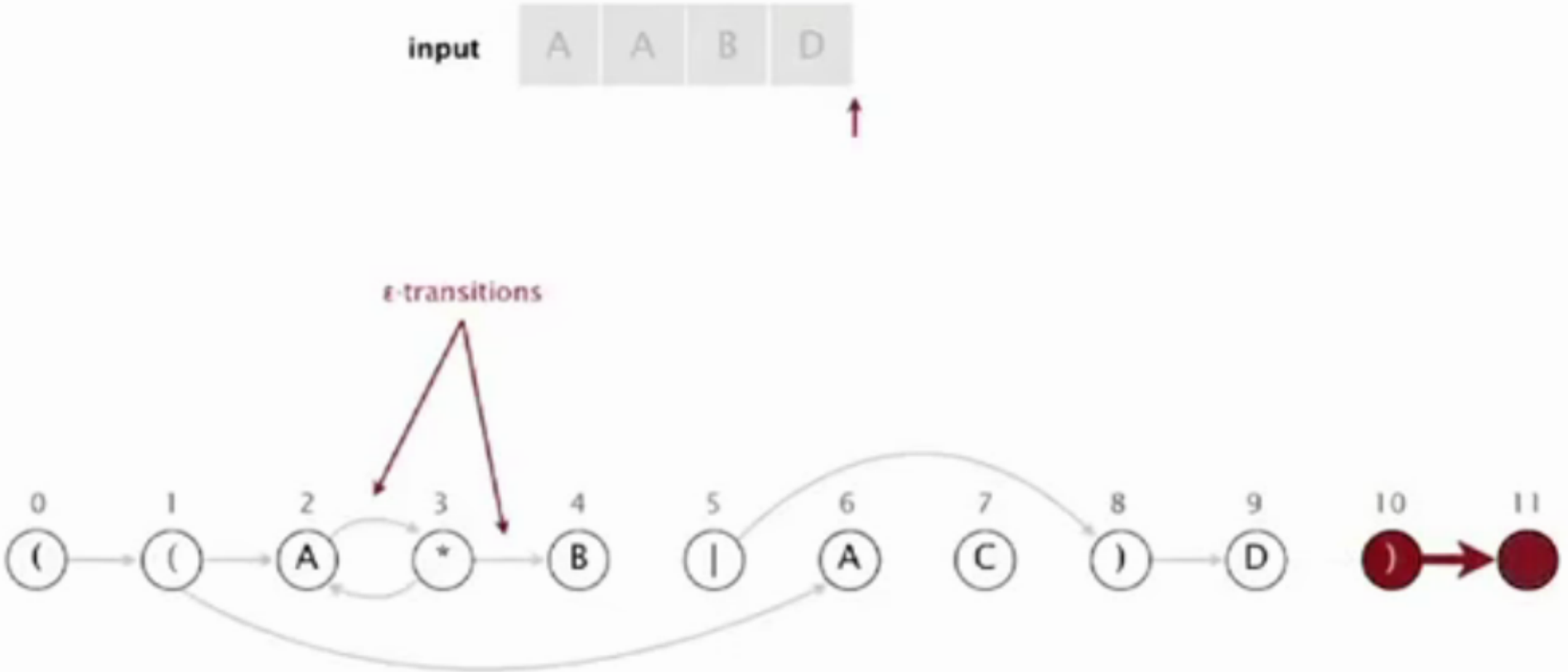


set of states reachable via ε-transitions after matching A A B D

NFA simulation demo

Read next input character.

- Find states reachable by match transitions.
- Find states reachable by ε-transitions

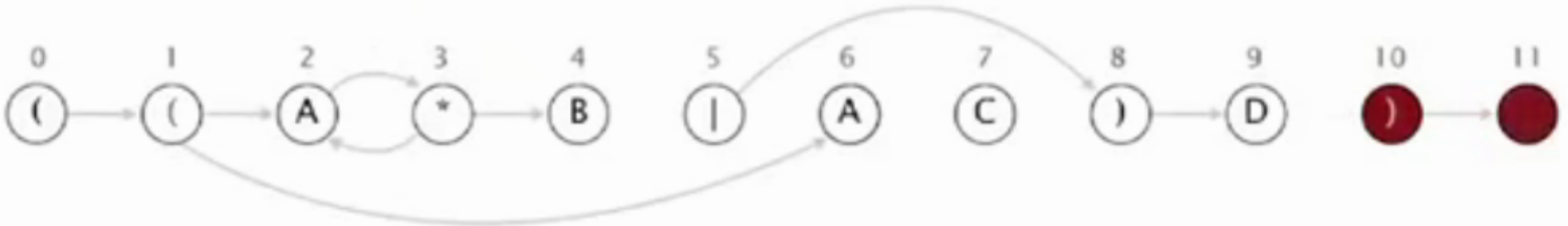


set of states reachable via ε-transitions after matching A A B D

NFA simulation demo

Read next input character.

- Find states reachable by match transitions.
- Find states reachable by ε-transitions

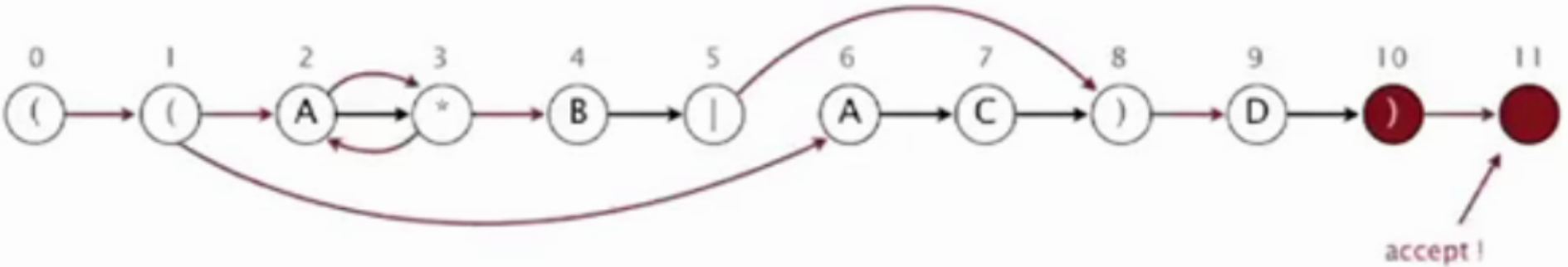


set of states reachable via ε-transitions after matching A A B D : { 10, 11 }

NFA simulation demo

When no more input characters:

- Accept if any state reachable is an accept state.
- Reject otherwise.



set of states reachable : { 10, 11 }

Representação de `nfa`

Os caracteres da `regexp` são mantidos em um vetor `re[]`.

Os estados são os vértices `0,1,...,m` de um digrafo `G`.

O estado inícial é 0 e o de aceitação é `m`.

Os arcos do digrafo `G` correspondem **apenas** a ϵ-transições.

Cada vértice `j` corresponde a um caractere `re[j]`.

Classe DFSpaths

```
public class DFSpaths {

   public DFSpaths(Digraph G, s) {...}

   public DFSpaths(Digraph G,
          Iterable<Integer> S) {...}

   public hasPath(int v) {...}

}
```

Consumo de tempo para vetores de listas de adjacência é $O(V + E)$.

Como a construção do `nfa` garante que $E \leq 2m$ temos que esse consumo de tempo é $O(m)$.

Classe NFA: esqueleto

```
public class NFA {
   // digrafo das transições epsilon
   private Digraph G;

   // expressão regular
   private String re;

   // number of caracteres em re
   private final int m;

   public NFA(String regexp) {...}

   public boolean recognizes(String txt)
   {...}
```

NFA: recognizes()

Decide se o string txt pertence a linguagem determinada pela expressão regular re.

```
public boolean recognizes(String txt) {
   int i, n = txt.length();
   DFSpaths dfs = new DFSpaths(G, 0);
   Bag<Integer> pc = new Bag<Integer>();
   for (int v = 0; v < G.V(); v++)
       if (dfs.hasPath(v)) pc.add(v);
```

NFA: recognizes()

```
for (i = 0; i < n; i++) {
   Bag<Integer> match= new Bag<Integer>();
   for (int v :  pc) {
       if (v == m) continue;
       if (re[v] == txt.charAt(i)
           || re[v] == '.')
           match.add(v+1);
   }
   dfs = new DFSpaths(G, match);
   pc = new Bag<Integer>();
   for (int v = 0; v < G.V(); v++)
       if (dfs.hasPathTo(v)) pc.add(v);
}
```

NFA: recognizes()

```
    // verifica se aceita
    for (int v:  pc)
        if (v == m) return true;
    return false;
}
```

Conclusão

O consumo de tempo de `recognizes()` para decidir se um string `txt` de comprimento `n` pertence a linguagem determinada por uma expressão regular `regexp` de comprimento `m` é proporcional a `nm`.

Construção do `nfa`

Inclua um estado para cada caractere na `regexp` mais um estado de aceitação.

Metacaracteres: `( ) * . |`

Concatenação: na `nfa` corresponde a uma simples transição para o estado seguinte; a transição saindo de metacaracteres é uma ϵ-transição.

Parenteses: acrescente uma ϵ-transição para o estado seguinte.

Construção do `nfa`

fecho: um $*$ ocorre depois de um caractere ou de um fecha parênteses.

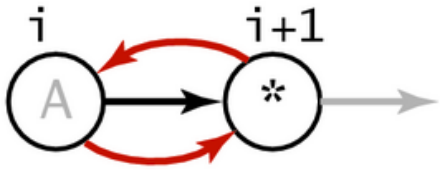
Depois de um caractere acrescente ϵ-transições para e do caractere.

Depois de um parênteses acrescente ϵ-transições para o correspondente abre parênteses.

Acrescente uma ϵ-transição para o estado seguinte.

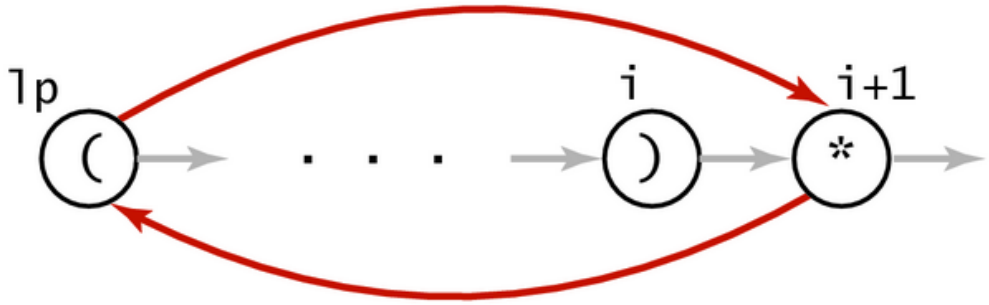
NFA: fecho

single-character closure




```
G.addEdge(i, i+1);
G.addEdge(i+1, i);
```

closure expression




```
G.addEdge(lp, i+1);
G.addEdge(i+1, lp);
```

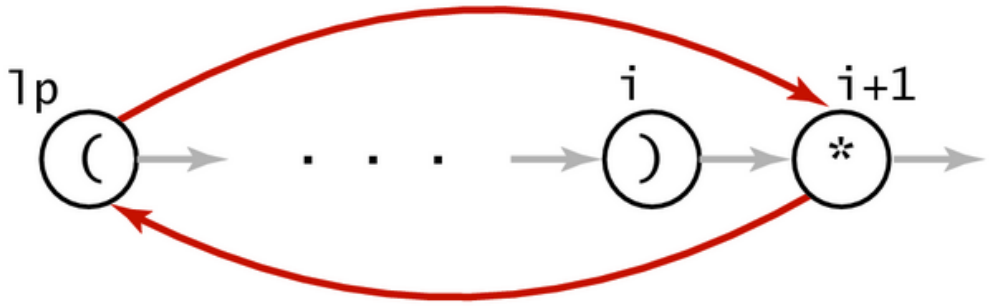
Construção do `nfa`

ou: temos $(\texttt{re}_1|\texttt{re}_2)$ onde $\texttt{re}_1$ e $\texttt{re}_2$ são expressões regulares.

Acrecente uma ϵ-transição de (para o estado depois de |.

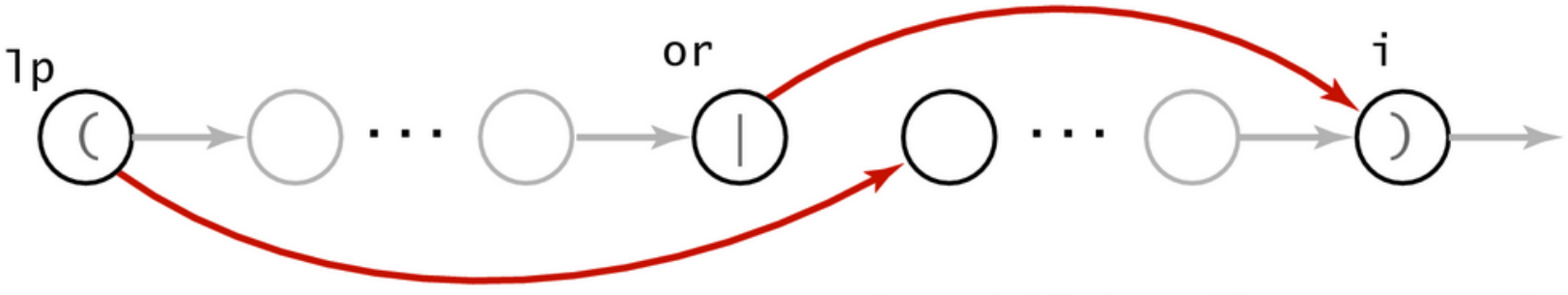
Acrecente uma ϵ-transição de | para o estado de).

Acrecente uma ϵ-transição de) para o estado seguinte.




```
G.addEdge(lp, i+1);
G.addEdge(i+1, lp);
```

or expression




```
G.addEdge(lp, or+1);
G.addEdge(or, i);
```

NFA construction rules

NFA construction demo

stack

((A * B | A C) D)

NFA construction demo

Left parenthesis.

- Add ε-transition to next state.
- Push index of state corresponding to (onto stack.

stack

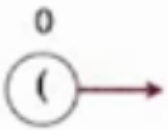
((A * B | A C) D)

NFA construction demo

Left parenthesis.

- Add ε-transition to next state.
- Push index of state corresponding to (onto stack.

stack



((A * B | A C) D)

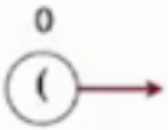
NFA construction demo

Left parenthesis.

- Add ε-transition to next state.
- Push index of state corresponding to (onto stack.

0

stack



((A * B | A C) D)

NFA construction demo

Left parenthesis.

- Add ε-transition to next state.
- Push index of state corresponding to (onto stack.

0

stack

((A * B | A C) D)

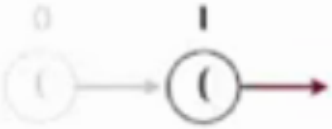
NFA construction demo

Left parenthesis.

- Add ε-transition to next state.
- Push index of state corresponding to (onto stack.

0

stack



((A * B | A C) D)

NFA construction demo

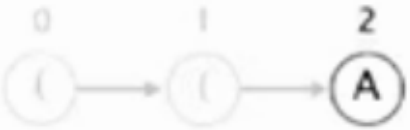
Alphabet symbol.

- Add match transition to next state.
- Do one-character lookahead:
  add ε-transitions if next character is *.

1

0

stack



((A * B | A C) D)

NFA construction demo

Alphabet symbol.

- Add match transition to next state.
- Do one-character lookahead:
  add ε-transitions if next character is *.

1

0

stack

((A * B | A C) D)

NFA construction demo

Alphabet symbol.

- Add match transition to next state.
- Do one-character lookahead:
  add ε-transitions if next character is *.

1

0

stack

((A * B | A C) D)

NFA construction demo

Closure symbol.

- Add ε-transition to next state.

1

0

stack

((A * B | A C) D)

NFA construction demo

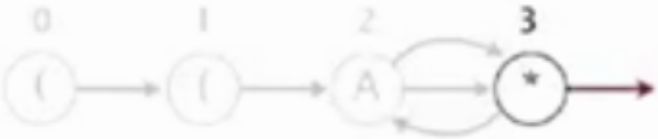
Closure symbol.

- Add ε-transition to next state.

1

0

stack



((A * B | A C) D)

NFA construction demo

Alphabet symbol.

- Add match transition to next state.
- Do one-character lookahead:
  add ε-transitions if next character is *.

1

0

stack

((A * B | A C) D)

NFA construction demo

Or symbol.

- Push index of state corresponding to | onto stack.

1

0

stack

((A * B | A C) D)

NFA construction demo

Alphabet symbol.

- Add match transition to next state.
- Do one-character lookahead:
  add ε-transitions if next character is *.

5

1

0

stack

((A * B | A C) D)

NFA construction demo

Alphabet symbol.

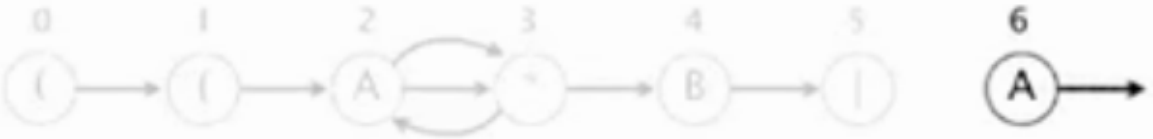
- Add match transition to next state.
- Do one-character lookahead:
  add ε-transitions if next character is *.

5

1

0

stack



((A * B | A C) D)

NFA construction demo

Alphabet symbol.

- Add match transition to next state.
- Do one-character lookahead:
  add ε-transitions if next character is *.

5

1

0

stack

((A * B | A C) D)

NFA construction demo

Right parenthesis.

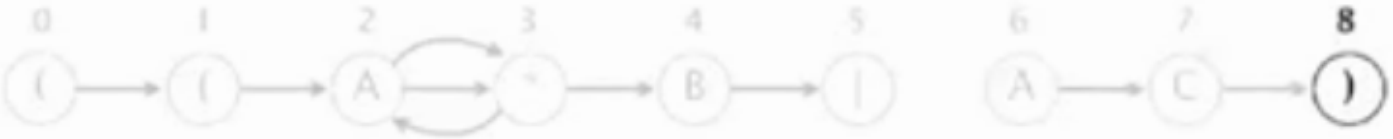
- Add ε-transition to next state.
- Pop corresponding (and possibly intervening |;
  add ε-transition edges for or.
- Do one-character lookahead:
  add ε-transitions if next character is *.

5

1

0

stack



((A * B | A C) D)

NFA construction demo

Right parenthesis.

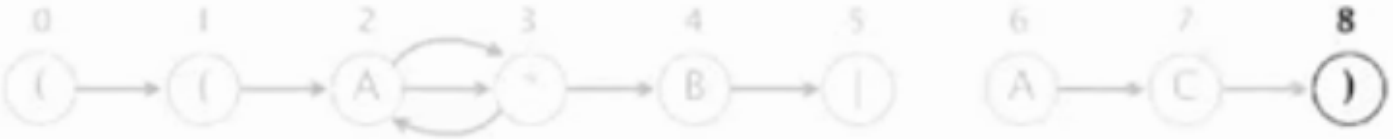
- Add ε-transition to next state.
- Pop corresponding (and possibly intervening |;
  add ε-transition edges for or.
- Do one-character lookahead:
  add ε-transitions if next character is *.

5

1

0

stack



((A * B | A C) D)

NFA construction demo

Right parenthesis.

- Add ε-transition to next state.
- Pop corresponding (and possibly intervening |;
  add ε-transition edges for or.
- Do one-character lookahead:
  add ε-transitions if next character is *.

5

1

0

stack

((A * B | A C) D)

NFA construction demo

Right parenthesis.

- Add ε-transition to next state.
- Pop corresponding (and possibly intervening |;
  add ε-transition edges for or.
- Do one-character lookahead:
  add ε-transitions if next character is *.

5

1

0

stack

((A * B | A C) D)

NFA construction demo

Alphabet symbol.

- Add match transition to next state.
- Do one-character lookahead:
  add ε-transitions if next character is *.

0

stack

((A * B | A C) D)

NFA construction demo

Alphabet symbol.

- Add match transition to next state.
- Do one-character lookahead:
  add ε-transitions if next character is *.

0

stack

((A * B | A C) D)

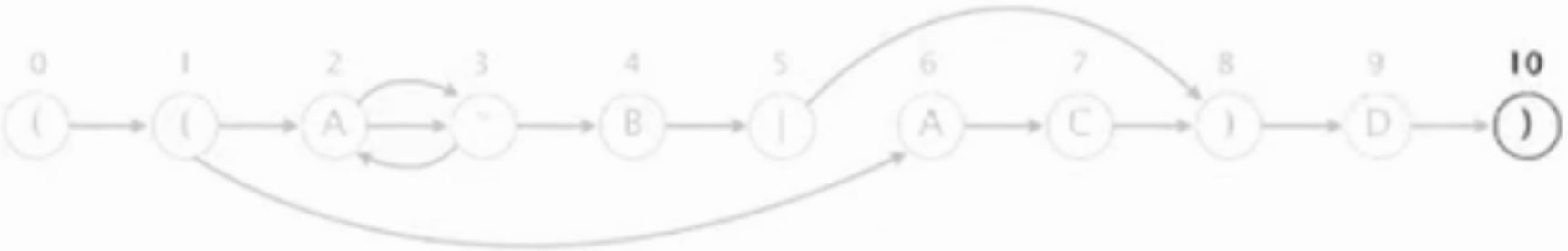
NFA construction demo

Right parenthesis.

- Add ε-transition to next state.
- Pop corresponding (and possibly intervening |;
  add ε-transition edges for or.
- Do one-character lookahead:
  add ε-transitions if next character is *.

0

stack



((A * B | A C) D)

NFA construction demo

Right parenthesis.

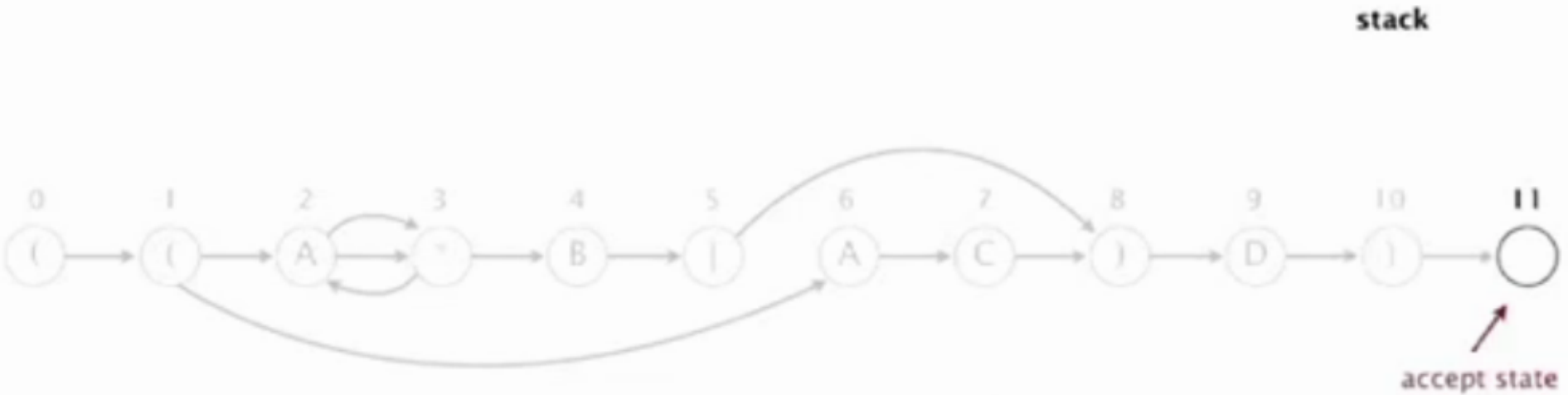
- Add ε-transition to next state.
- Pop corresponding (and possibly intervening |;
  add ε-transition edges for or.
- Do one-character lookahead:
  add ε-transitions if next character is *.

((A * B | A C) D)

NFA construction demo

End of regular expression.

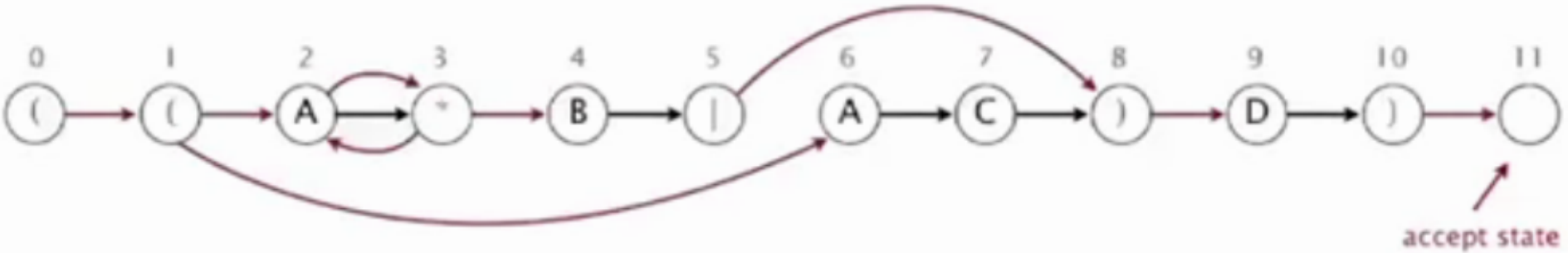
- Add accept state.



((A * B | A C) D)

NFA construction demo

stack



NFA corresponding to the pattern ((A * B | A C) D)

NFA: construtor

```
public NFA(String regexp) {
   re = regexp.toCharArray();
   m = re.length;
   Stack<Integer>ops=new Stack<Integer>();
   G = new Digraph(m+1);
   for (int i = 0; i < m; i++) {
       int lp = i;
       if (re[i] == '('
           || re[i] == '|')
           ops.push(i);
```

NFA: construtor

```
for (int i = 0; ...
    else if (re[i] == ')') {
        int or = ops.pop();
        if(re[or] == '|') {
            lp = ops.pop();
            G.addEdge(lp, or+1);
            G.addEdge(or, i);
        }
        else if(re[or] == '(')
            lp = or;
    }
```

NFA: construtor

```
    // fecho:  usa um caractere lookahead
    if (i < m-1 && re[i+1] == '*') {
        G.addEdge(lp, i+1);
        G.addEdge(i+1, lp);
    }
    if (re[i] == '('
        || re[i] == '*'
        || re[i] == ')')
        G.addEdge(i, i+1);
    }
}
```

Conclusão

O consumo de tempo e espaço para construir um `NFA` correspondente a uma `regexp` de comprimento `m` é proporcional a `m`.

GREP

O clássico cliente grep para reconhecimento de padrões.

```
public class GREP {
   public static void main(String[] args){
       String regexp ="(.*"+args[0]+".*)";
       NFA nfa = new NFA(regexp);
       while (StdIn.hasNextLine()) {
           String txt= StdIn.readLine();
           if (nfa.recognizes(txt))
              StdOut.println(txt);
       }
   }
}
```

Conclusão

Dada um expressão regular `regexp` de comprimento `m` representando uma linguagem `L` e um texto `txt` de comprimento `n` o consumo de tempo de `GREP` para reconhecer as linhas de `txt` que contêm uma substring `pat` em `L` é proporcional a `nm`.

Comentários

O utilitário `grep` parece construir um `dfa` e não um `nfa`.

Vejam o arquivo `dfasearch.c` que está no diretório `glibc` ou baixem o fonte do `grep` da página https://www.gnu.org/software/grep.

Mais comentários

A página Regular expressions do algs4 tem alguns comentários interessantes sobre bibliotecas com implementação de busca por expressões regulares.

Segundo essa página a busca em várias dessas bibliotecas utiliza uma algoritmo backtracking que pode consumir tempo exponencial.

Os exemplos a seguir, copiados da página do algs4 são devidos ao método
`public boolean matches(String regexp)`
da classe `String` do Java.

Mais comentários

```
java Validate "(a|aa)*b"
aaaaaaaaaaaaaaaaaaaaaaaaaaaaac
1.6 seconds
java Validate "(a|aa)*b"
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaac
3.7 seconds
java Validate "(a|aa)*b"
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaac
9.7 seconds
java Validate "(a|aa)*b"
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaac
23.2 seconds
java Validate "(a|aa)*b"
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaac
62.2 seconds
java Validate "(a|aa)*b"
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaac
161.6 seconds
```

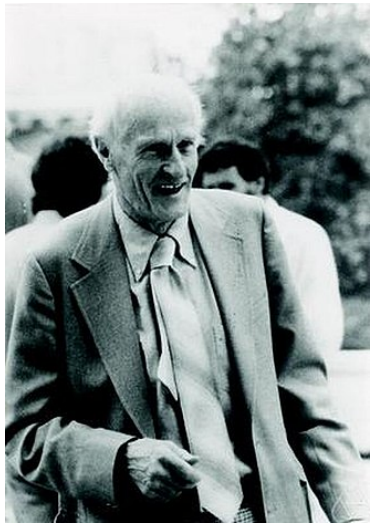
Mais referências

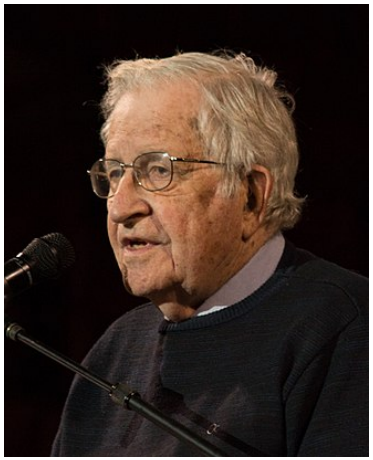
Mais algumas referências *da hora*.

- Regular Expression Matching Can Be Simple And Fast (but is slow in Java, Perl, PHP, Python, Ruby, ...) por Russ Cox;
- Building a RegExp machine por Dmitry Soshnikov;

História

1951: Stephen Kleene
matemático,
inventou expressões
regulares





1956: Noam Chomsky
linguista, filósofo, ativista político...
definiu a hierarquia de Chowsky:
expressões regulares são
reconhecidas por `dfa`s

História

1961: Ken Thompson
cientista da computação,
hacker,
popularizou o uso de `regexp`s:

- `grep` e
- análise léxica

Comentários finais

Fonte: http://www.quickmeme.com/

MAC0323 – Edição 2019

Livro

Nossa referência básica foi o livro

SW = *Sedgewick & Wayne,*
Algorithms, 4th Editions
http://algs4.cs.princeton.edu/

Notas de aula de Paulo Feofiloff baseadas no livro *Algorithms*

http://www.ime.usp.br/~pf/estruturas-de-dados.

MAC0323

MAC0323 **foi** uma disciplina introdutória em:

Projeto de algoritmos: BFS, DFS, heurísticas, pré-processamento, aleatorização, redimensionamento, move to front, Dijkstra, ~~Kosaraju~~, Knuth, Morris e Pratt, Rabin e Karp, Boyer e Moore, Run-length enconding, Huffman, Burrows-Wheeler, Lempel, Ziv, Welch, radix-sort, LSD, MSD , 3-way string quicksort, prefix doubling, A-estrela, DFA, NFA.
que nasceram de aplicações cotidianas em ciência da computação:

MAC0323

MAC0323 **foi** uma disciplina introdutória em:

Estruturas de dados: bags, stacks, queues, heaps, leftist heaps, Tries, TSTs, BSTs, árvores rubro-negras, hashing com encadeamento, hashing com sondagem linear, skip-lists, union-find, digrafos, grafos.

que nasceram de aplicações cotidianas em ciência da computação:

Pausa para nossos comerciais

- EP15: 23/JUN
- Prova 3: terça-feira, 25/JUN
- Prova Sub: média das provinhas

Pausa para nossos comerciais

Algoritmos no próximo semestre:

- MAC0338 Análise de Algoritmos
- MAC0315 Otimização Linear
- MAC0328 Algoritmos em Grafos
- MAC0414 Autômatos, Computabilidade e Complexidade

Fonte: http://dawallpaperz.blogspot.com.br/