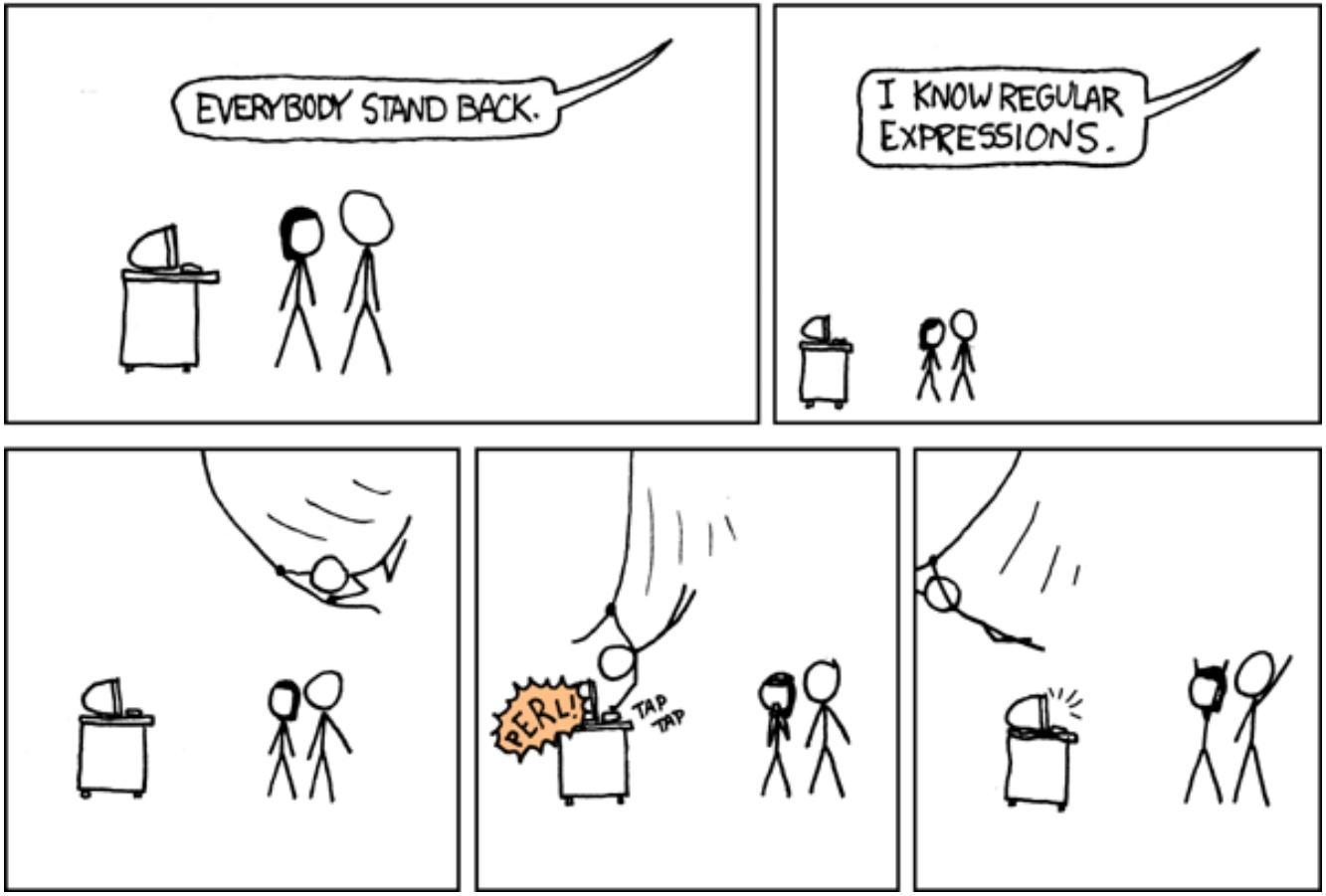
MAC0323 Algoritmos e Estruturas de Dados II

Edição 2020 – 2

AULA 30

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 e 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 $\texttt{re}$ 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 $(\texttt{re}_1\texttt{re}_2)$ obtida através da "*concatenação*" de duas expressões regulares $\texttt{re}_1$ e $\texttt{re}_2$; ou
- uma string $(\texttt{re}_1|\texttt{re}_2)$ obtida através da "*união*" de duas expressões regulares $\texttt{re}_1$ e $\texttt{re}_2$;
- uma string $(\texttt{re}*)$ 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, $\mathtt{A}$, $\mathtt{AA}$, $\mathtt{AAA}\ldots$

  $\mathtt{ABC}$, $\mathtt{ABBC}$, $\mathtt{ABCABC}\ldots$

  $\mathtt{ABA}$, $\mathtt{ABBA}$, $\mathtt{ABAABA}\ldots$

  $\mathtt{ABCABA}$, $\mathtt{ABAABC}$, $\ldots$

Abreviaturas

É conveniente utilizarmos abreviaturas como:

- "`.`": representa qualquer caractere, `AB.BA` representa `ABABA`, `ABBBA`, `ABCBA`, ...
- "`+`": 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` que representa as strings reconhecidas pelo `dfa`.

Plano

Procedemos 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

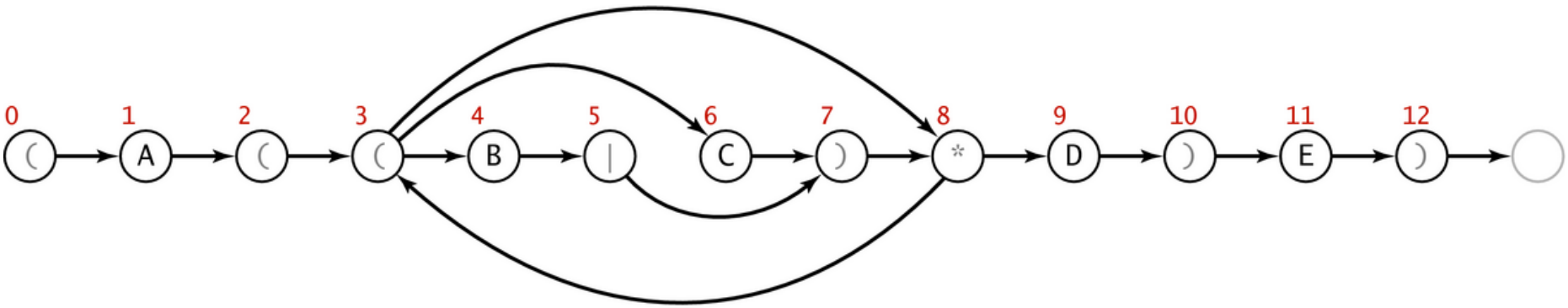
Utilizar outro tipo de autômato.

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` que
representa as strings reconhecidas pelo `nfa`.

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 ')'.

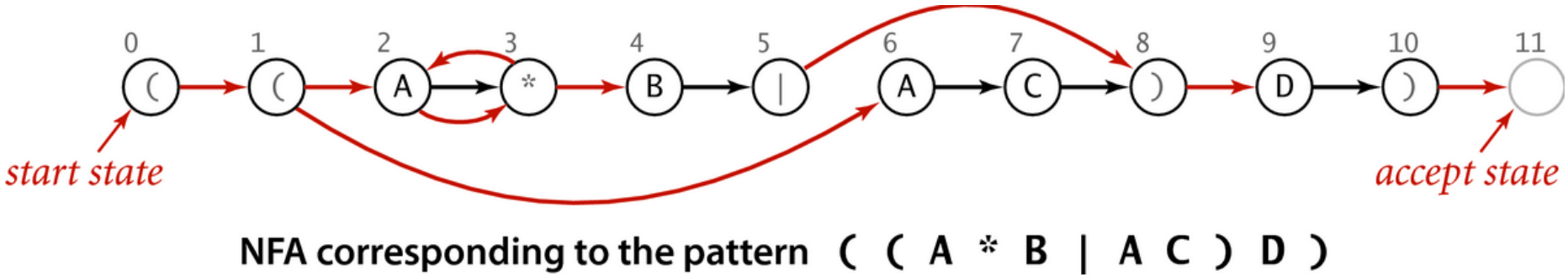
nfa tem um estado para cada caractere na regexp.

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

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

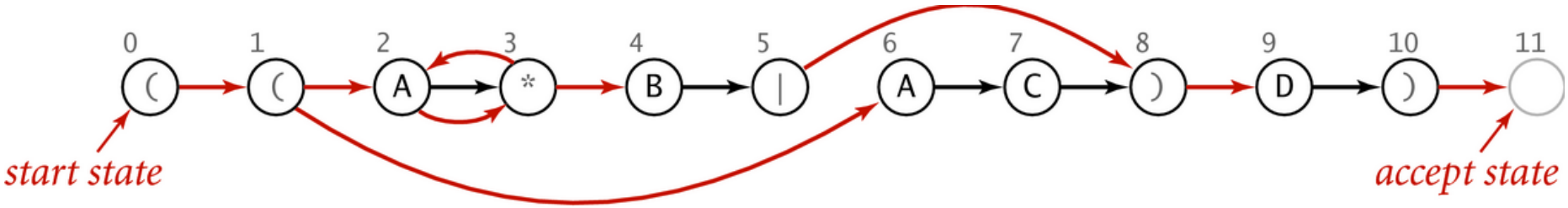
Aceita se **existe** uma sequência de transições que, após soletrar todos os caracteres em txt, termina em um estado de aceite.

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

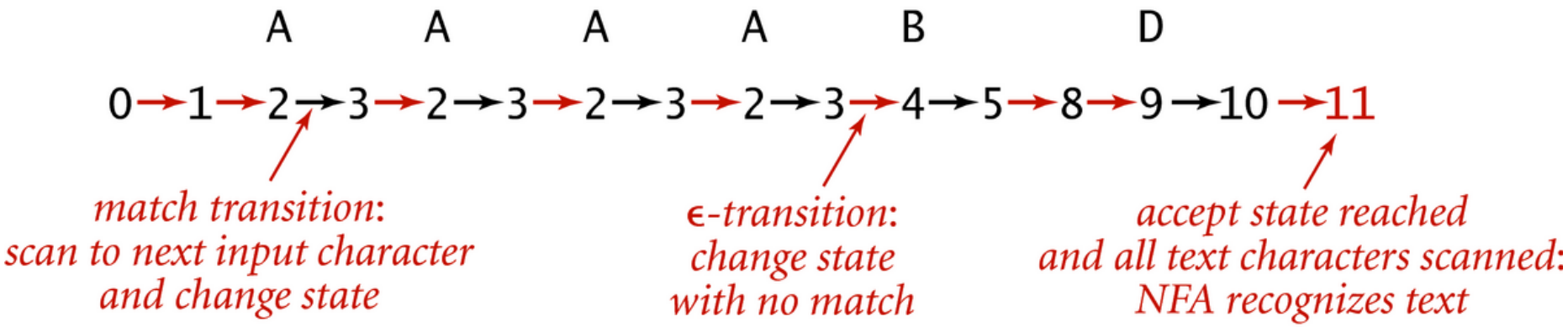


NFA corresponding to the pattern ((A * B | A C) D)

NFA: soletrando

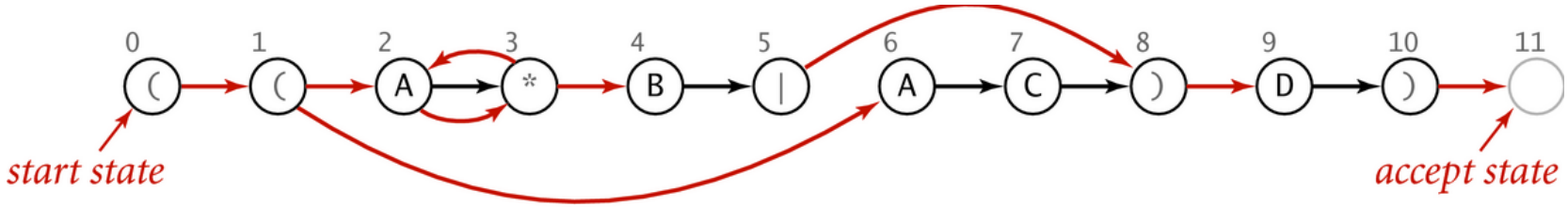


NFA corresponding to the pattern ((A * B | A C) D)



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

NFA: soletrando



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

A A A

0→1→2→3→2→3→4 ← *no way out of state 4*

wrong guess if input is

A A A A B D

A

0→1→6→7 ← *no way out of state 7*

A A A A C

0→1→2→3→2→3→2→3→2→3→4 ← *no way out of state 4*

NFA: mais estrutura

Um estado para cada caractere de `regexp`.

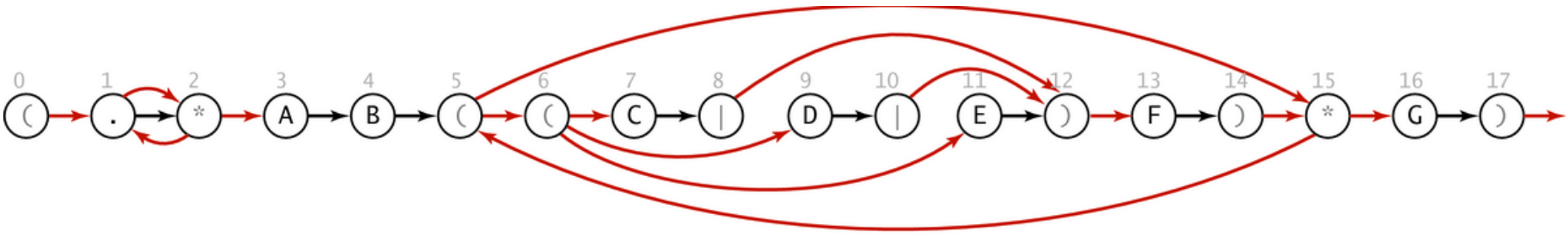
Estados correspondentes a letras têm apenas um **arco preto** saindo para o estado seguinte.

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?

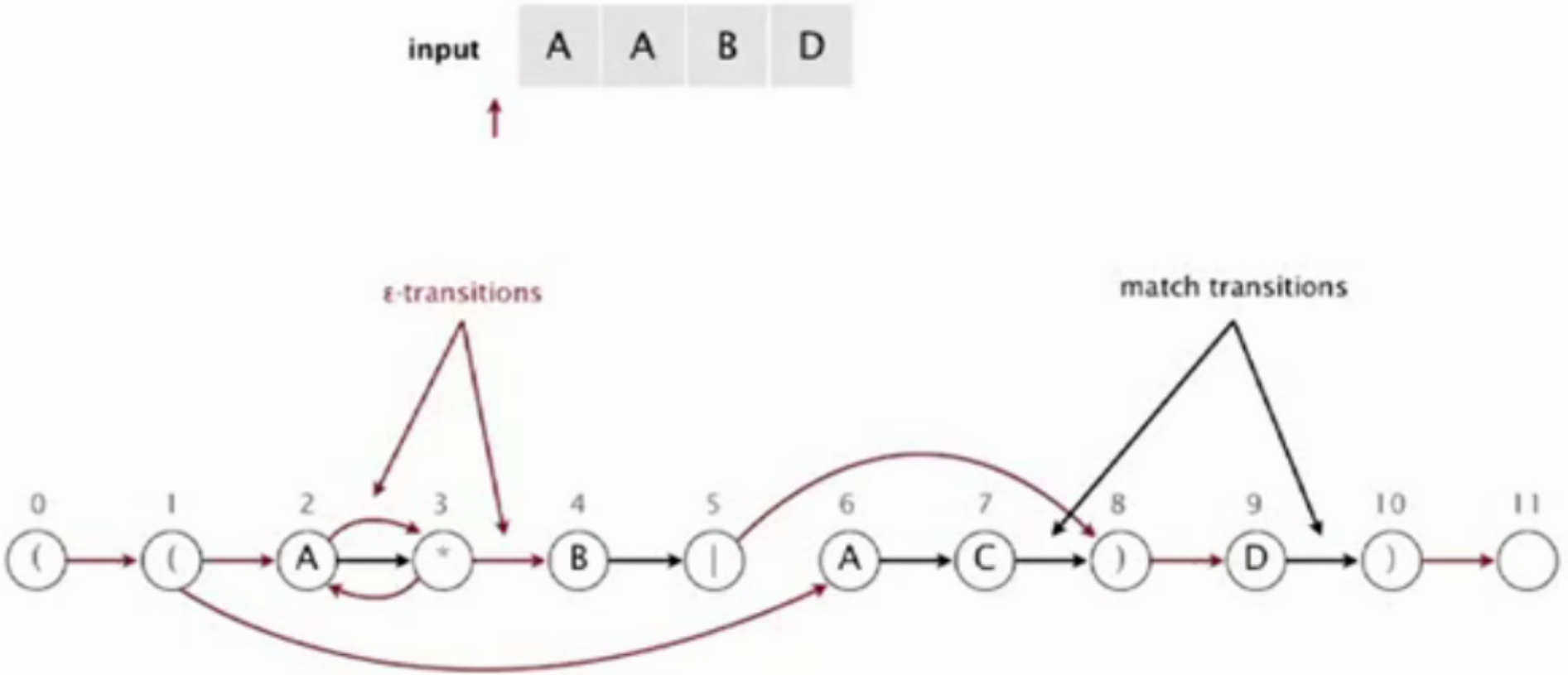
`DFA` $\Rightarrow$ soletrar `txt`,
aplicando **transições pretas**, fácil

`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.

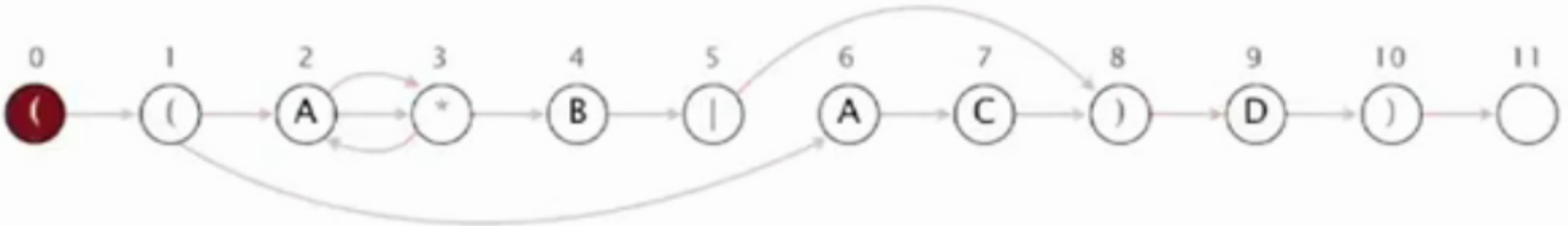


NFA corresponding to the pattern ((A * B | A C) D)

NFA simulation demo

Read next input character.

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

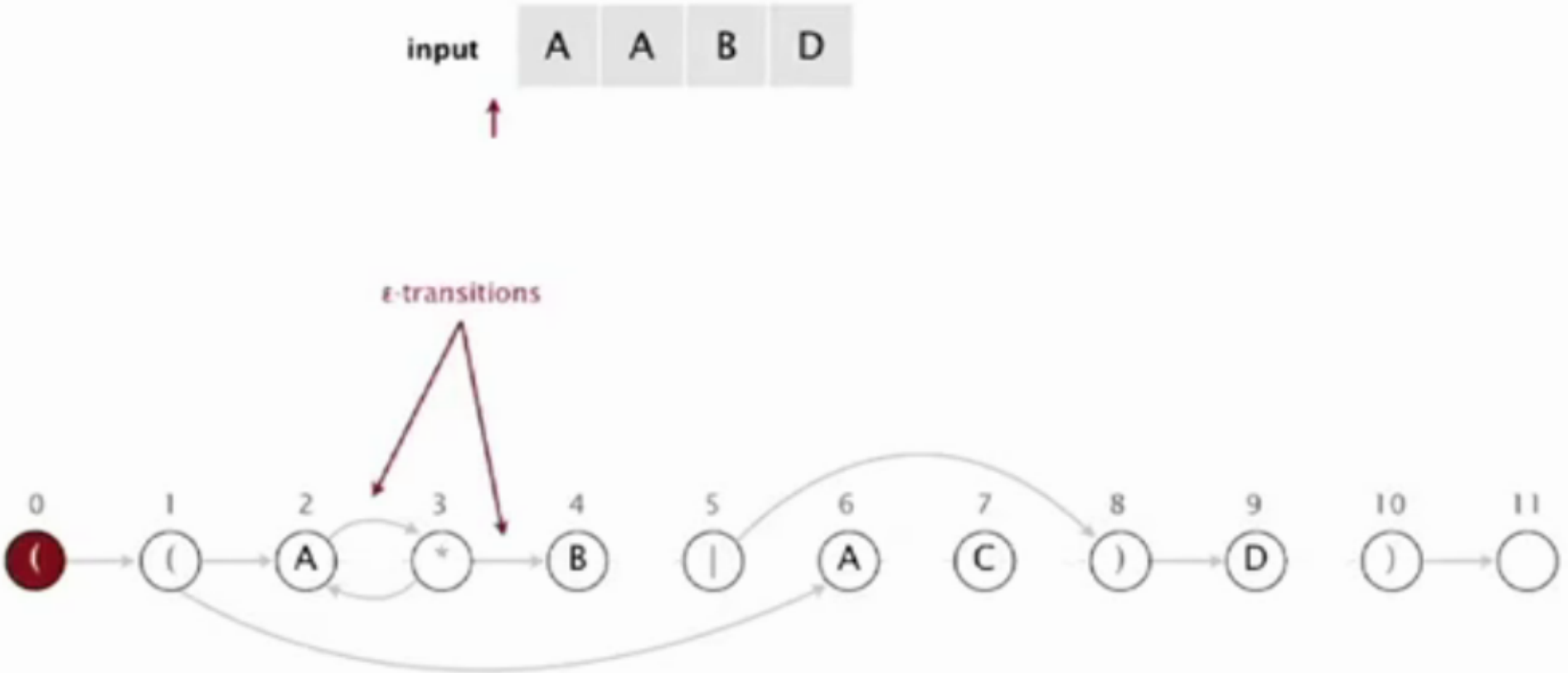


set of states reachable from start: 0

NFA simulation demo

Read next input character.

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

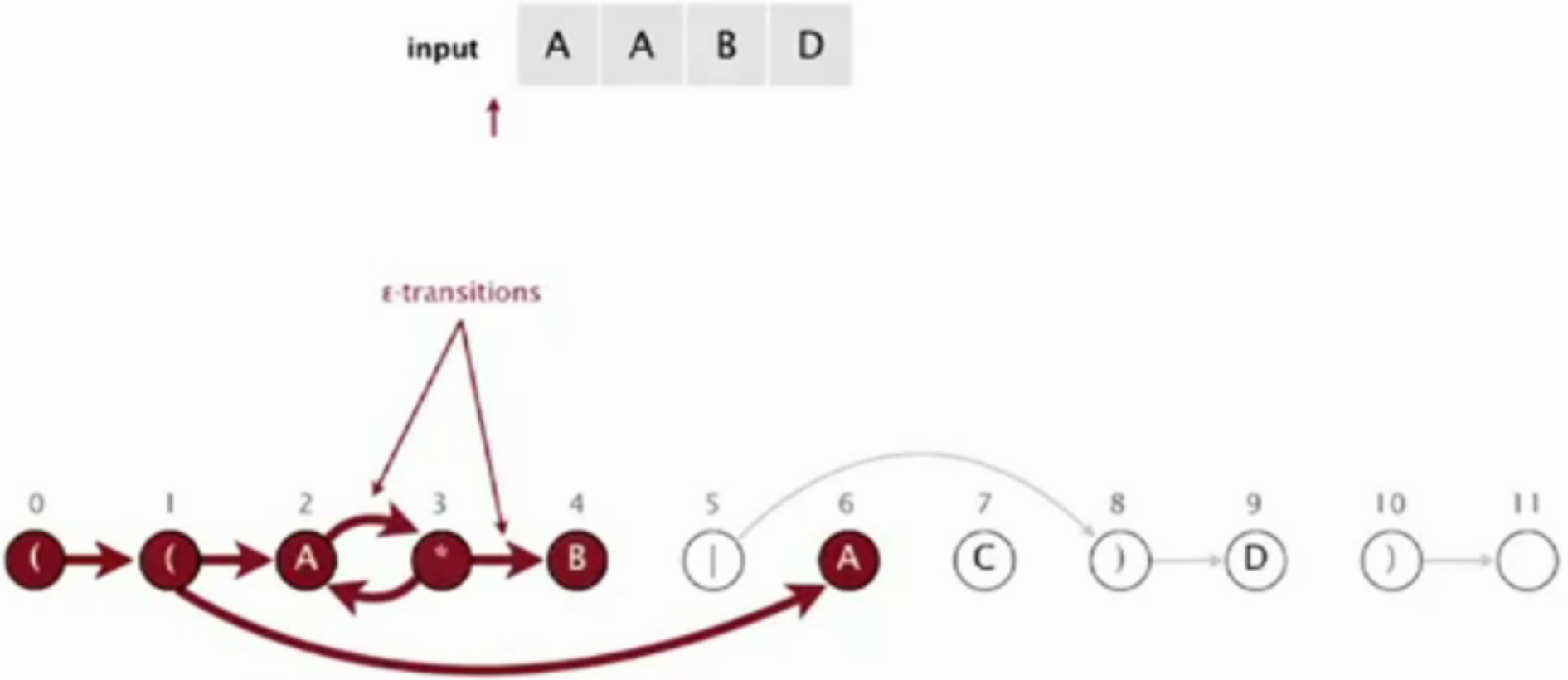


set of states reachable via ε-transitions from start

NFA simulation demo

Read next input character.

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

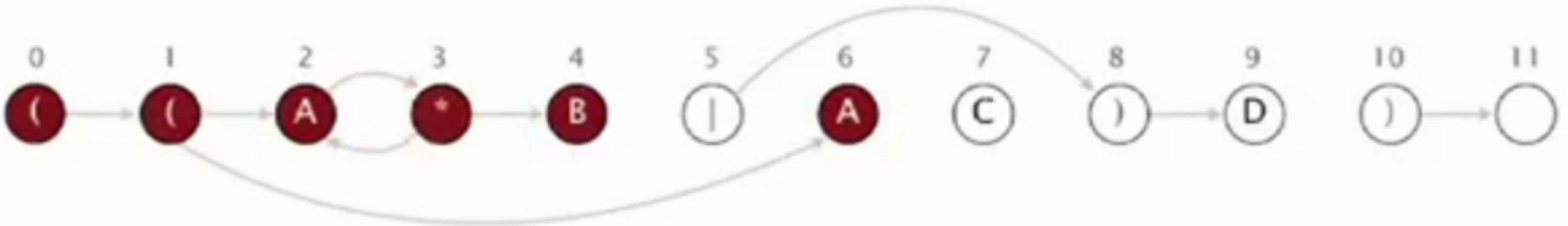


set of states reachable via ε-transitions from start

NFA simulation demo

Read next input character.

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

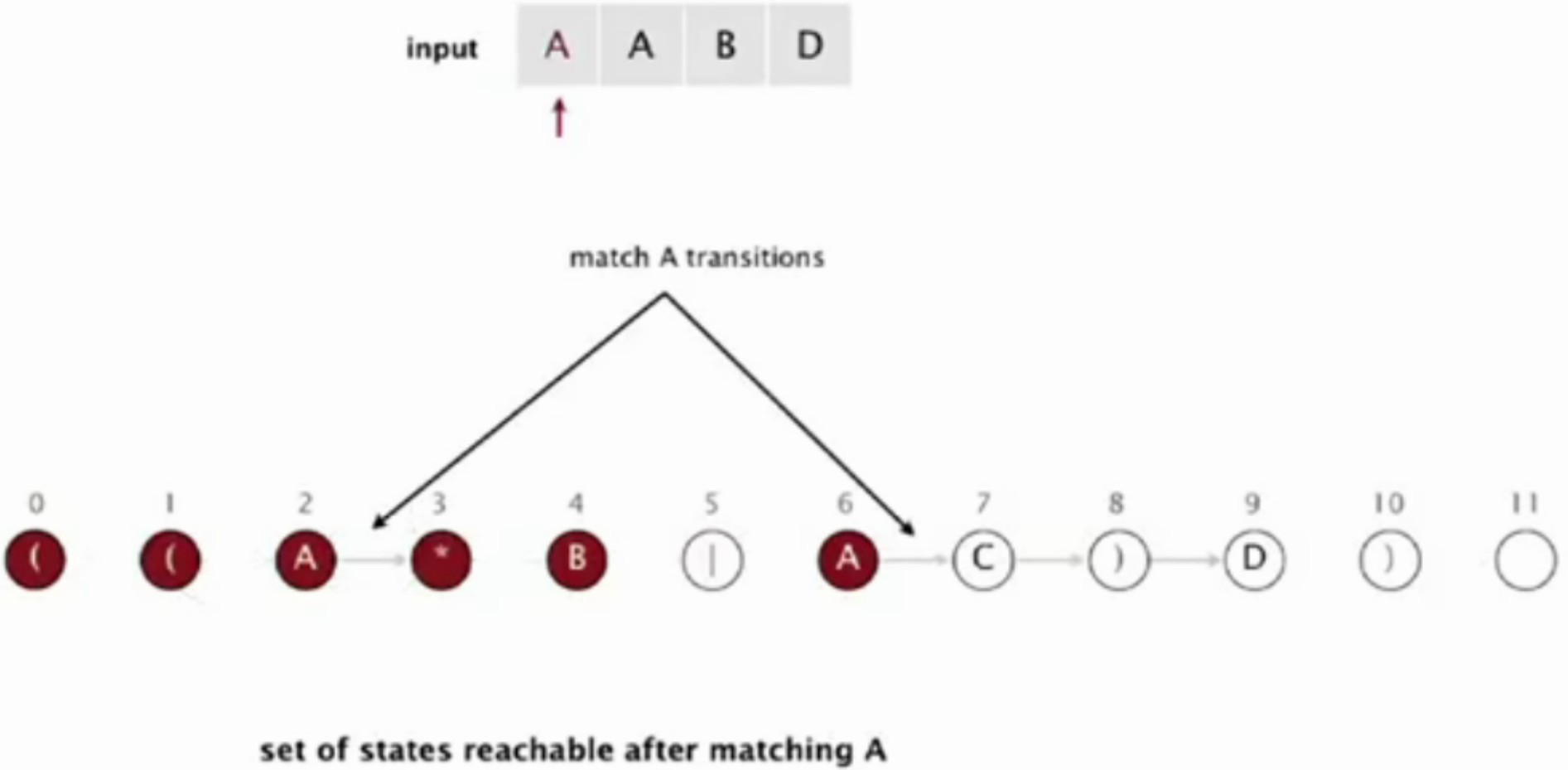


set of states reachable via ε-transitions from start : { 0, 1, 2, 3, 4, 6 }

NFA simulation demo

Read next input character.

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

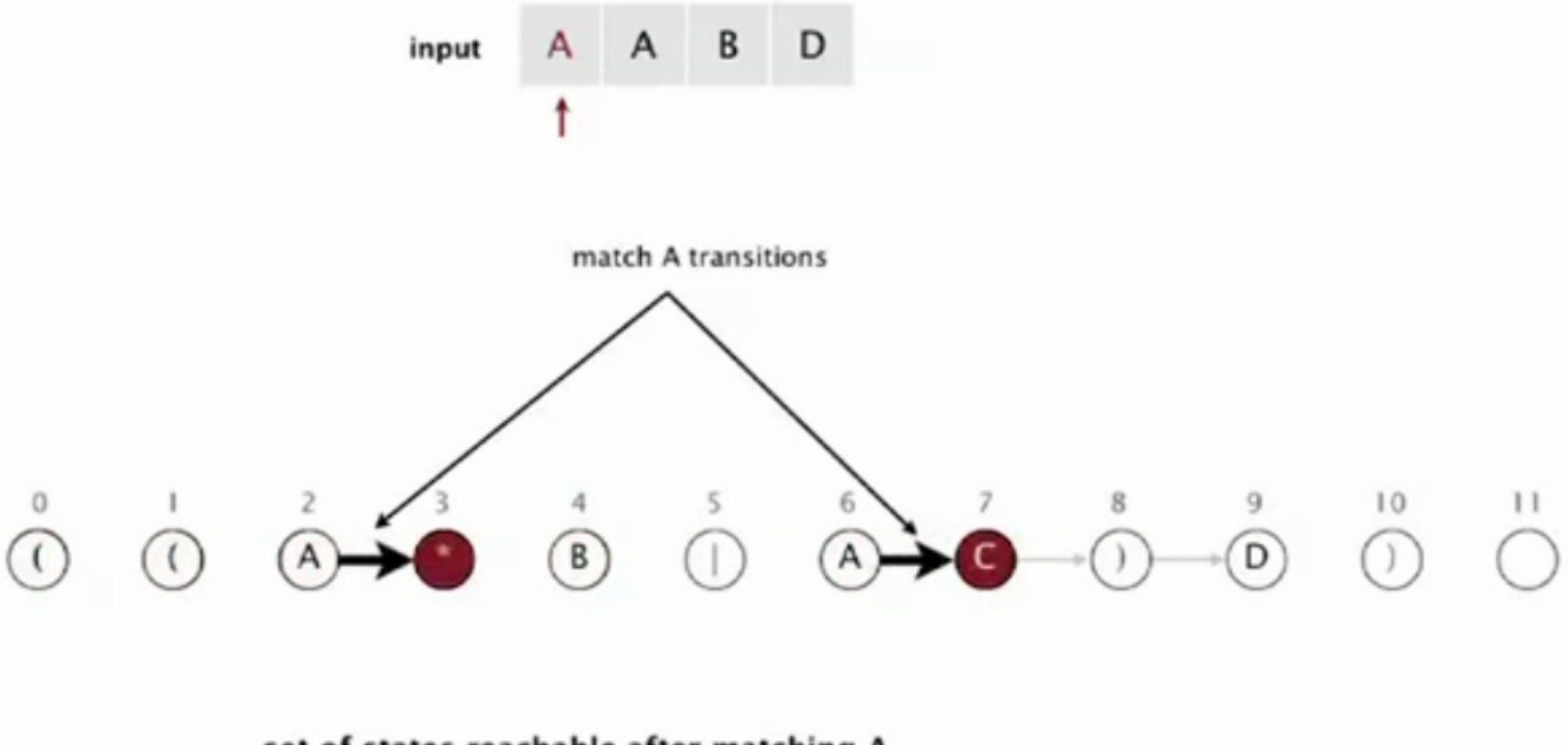


set of states reachable after matching A

NFA simulation demo

Read next input character.

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

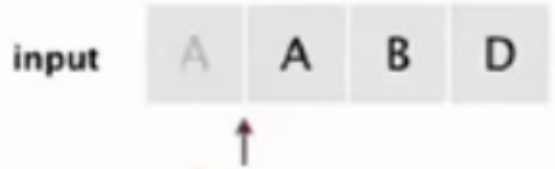


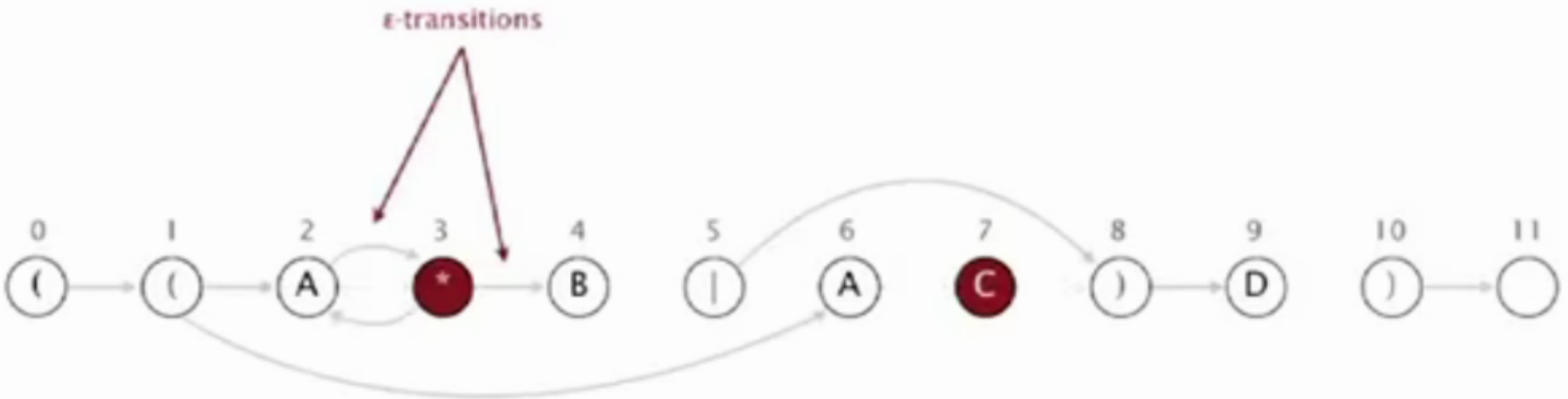
set of states reachable after matching A

NFA simulation demo

Read next input character.

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



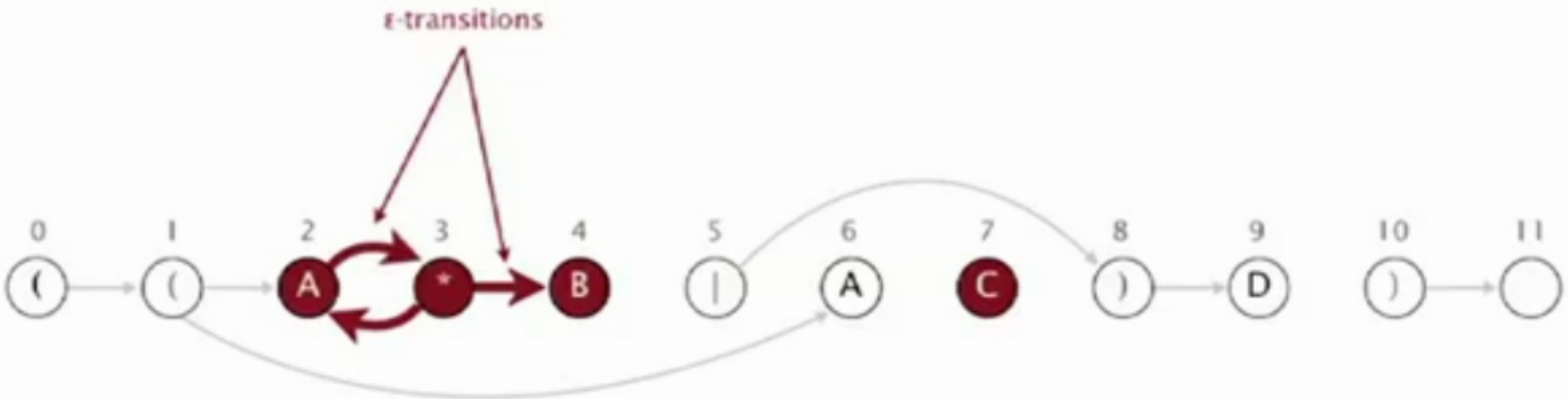


set of states reachable via ε-transitions after matching A

NFA simulation demo

Read next input character.

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

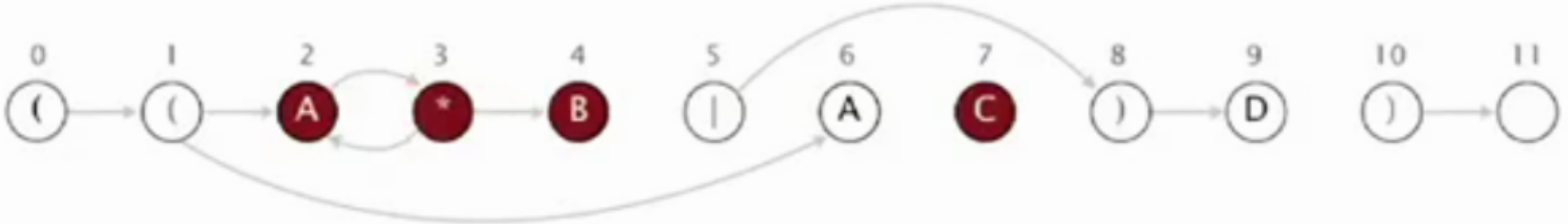


set of states reachable via ε-transitions after matching A

NFA simulation demo

Read next input character.

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

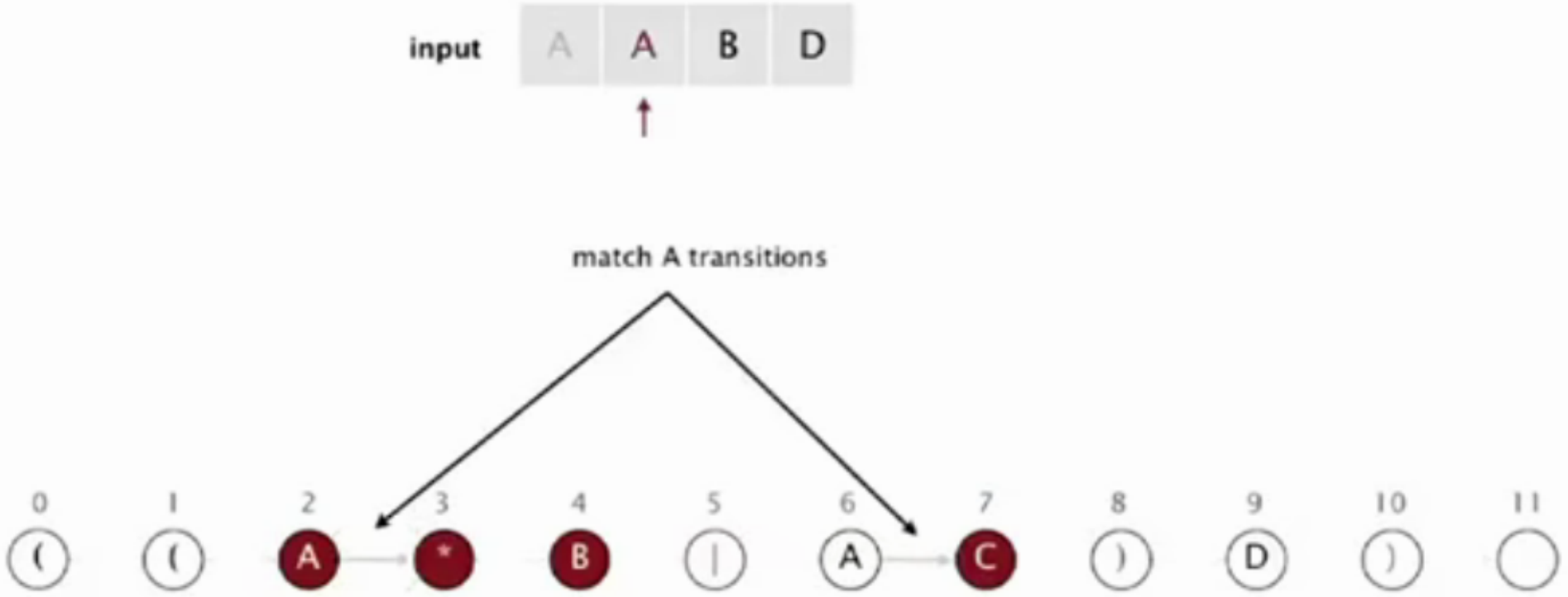


set of states reachable via ε-transitions after matching A : { 2, 3, 4, 7 }

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.

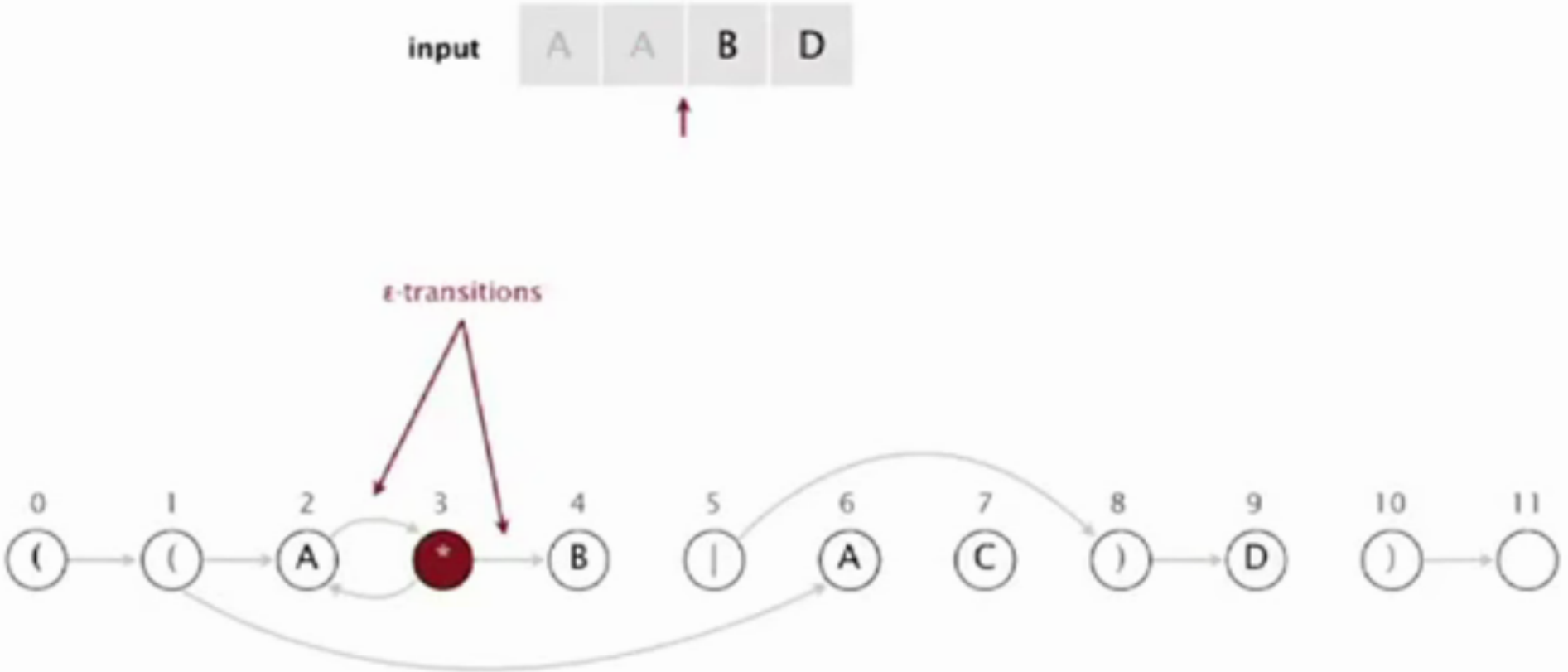
- Find states reachable by match transitions.
- Find states reachable by ε-transitions

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

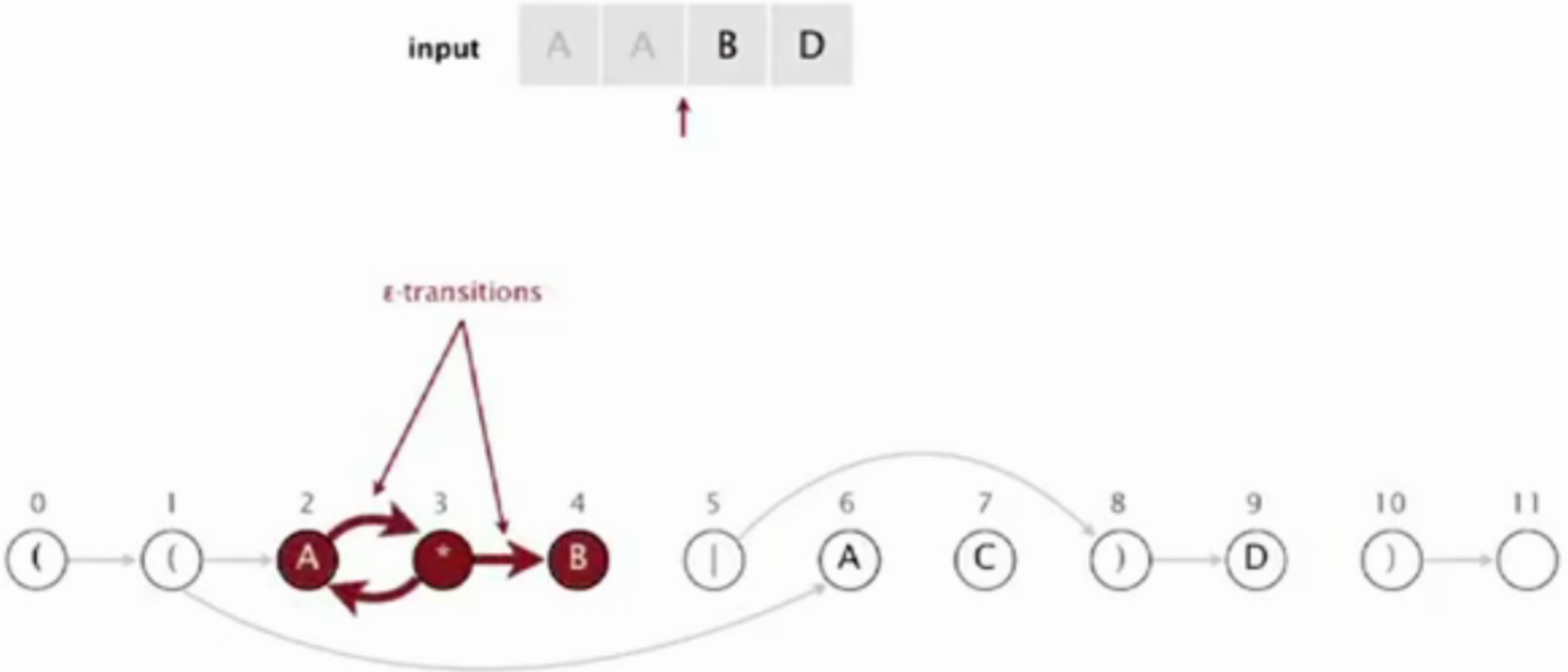


set of states reachable via ε-transitions after matching A A

NFA simulation demo

Read next input character.

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

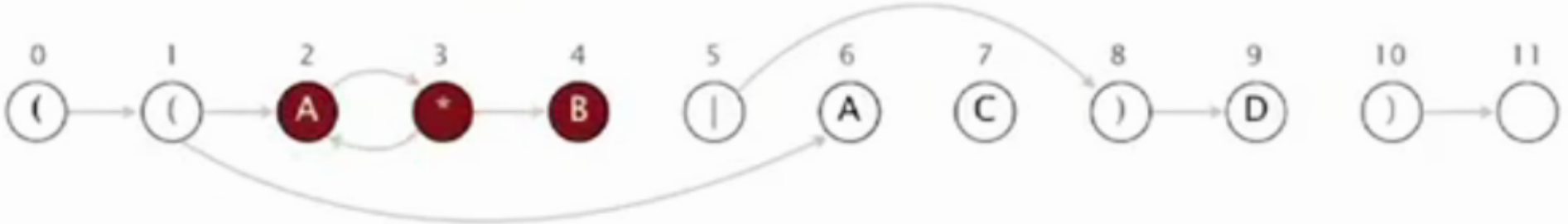


set of states reachable via ε-transitions after matching A A

NFA simulation demo

Read next input character.

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

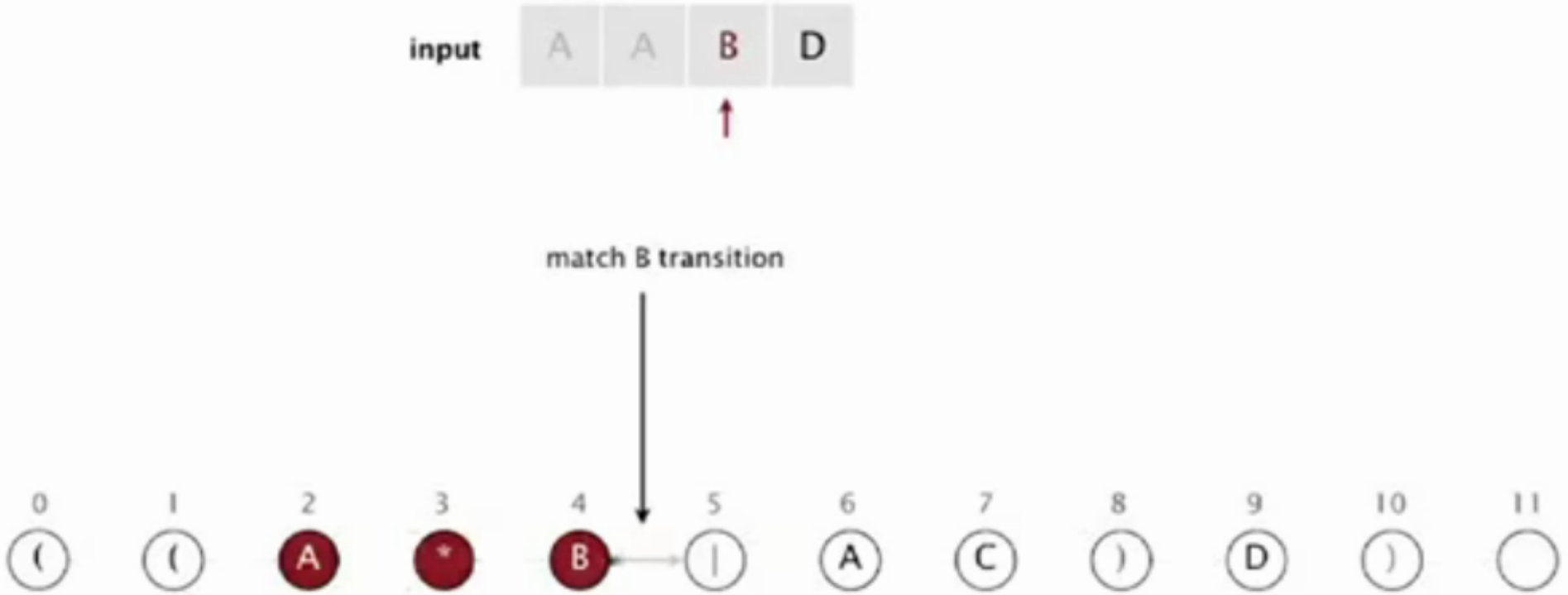


set of states reachable via ε-transitions after matching A A : { 2, 3, 4 }

NFA simulation demo

Read next input character.

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

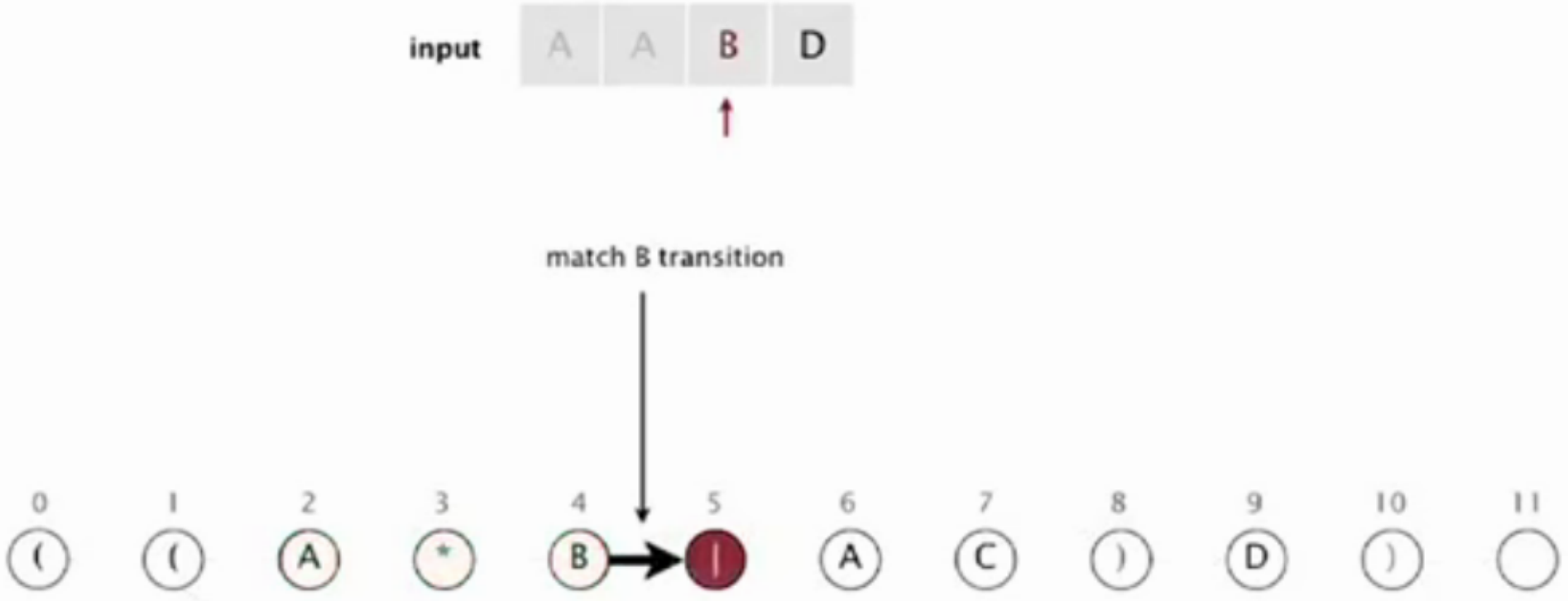


set of states reachable after matching A A B

NFA simulation demo

Read next input character.

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

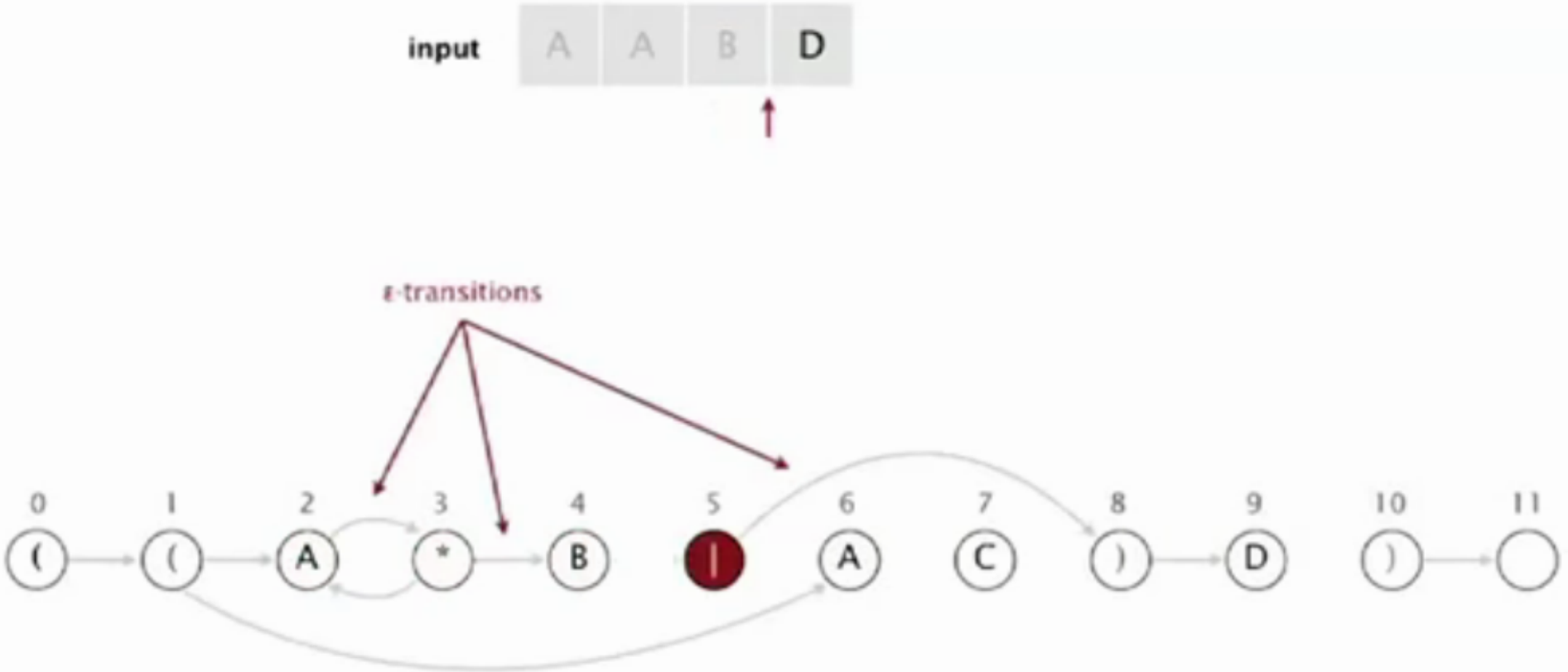


set of states reachable after matching A A B

NFA simulation demo

Read next input character.

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

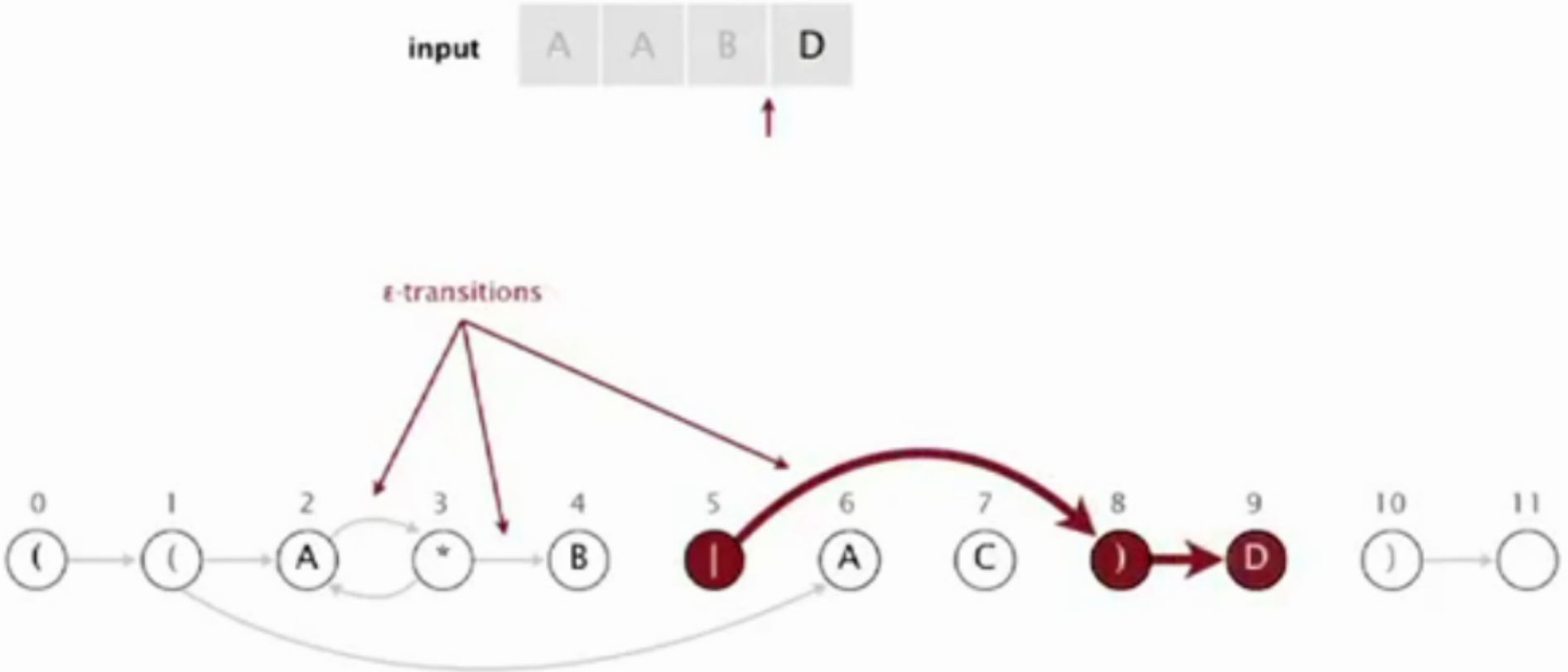


set of states reachable via ε-transitions after matching A A B

NFA simulation demo

Read next input character.

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

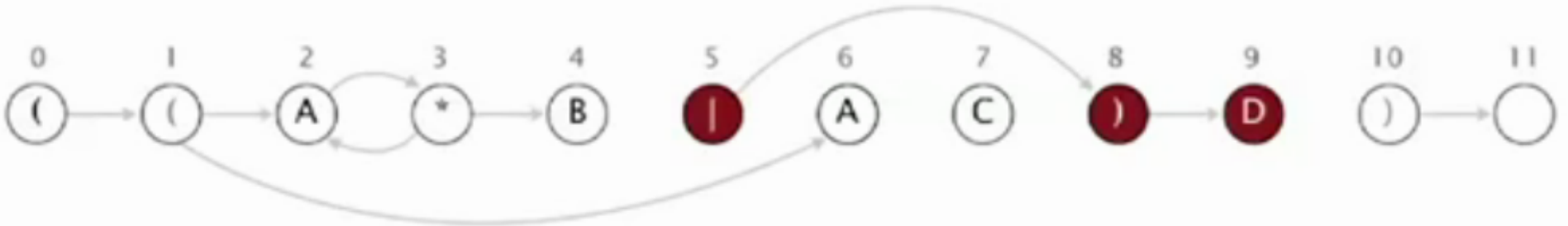


set of states reachable via ε-transitions after matching A A B

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.

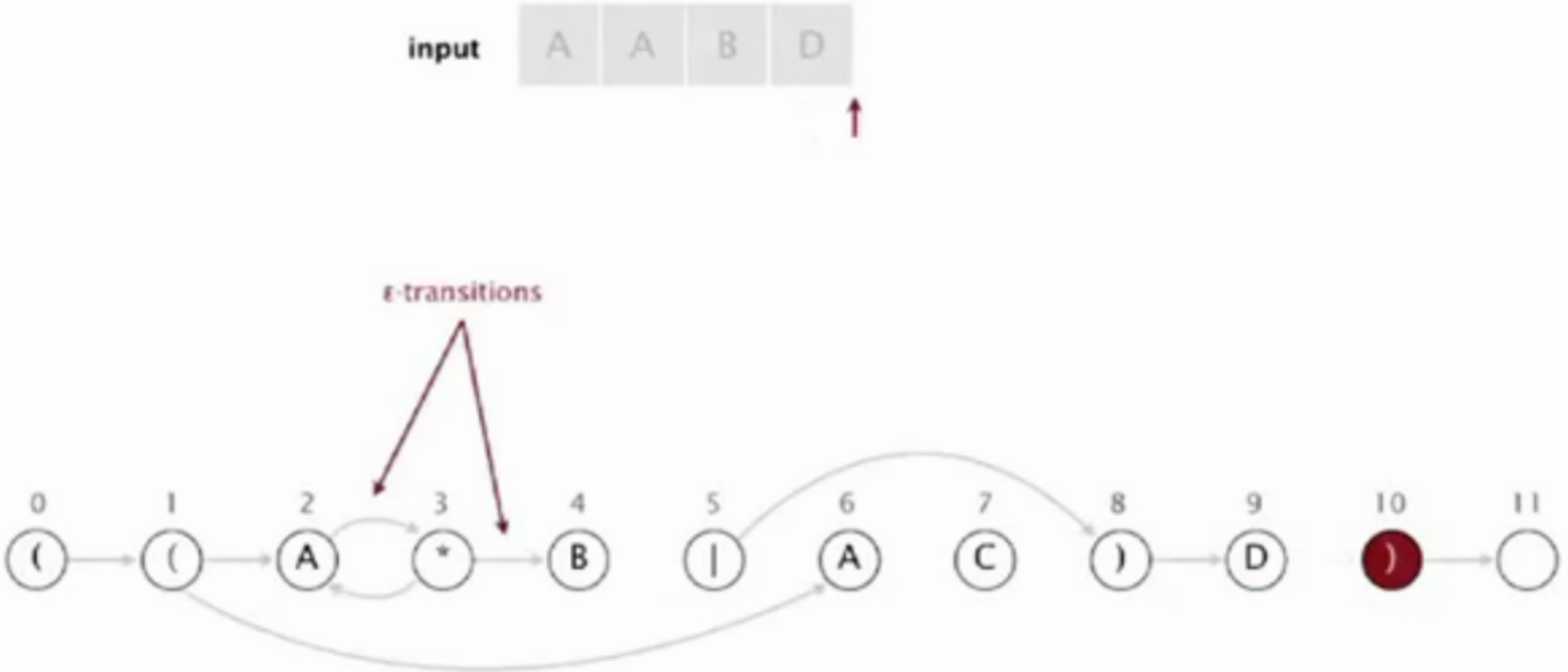
- Find states reachable by match transitions.
- Find states reachable by ε-transitions

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

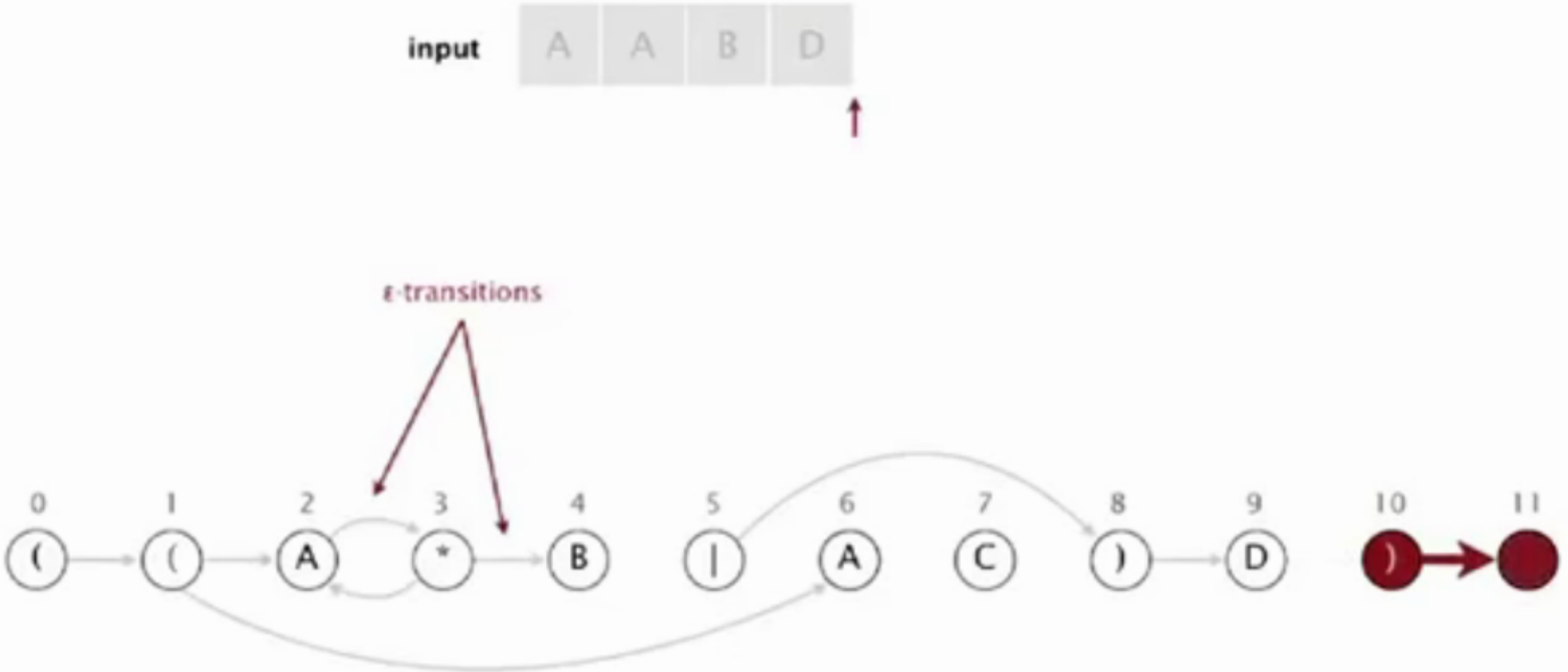


set of states reachable via ε-transitions after matching A A B D

NFA simulation demo

Read next input character.

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

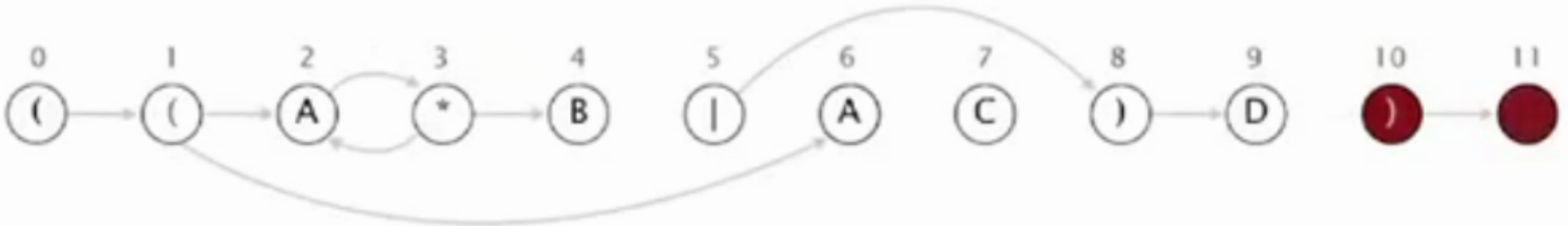


set of states reachable via ε-transitions after matching A A B D

NFA simulation demo

Read next input character.

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

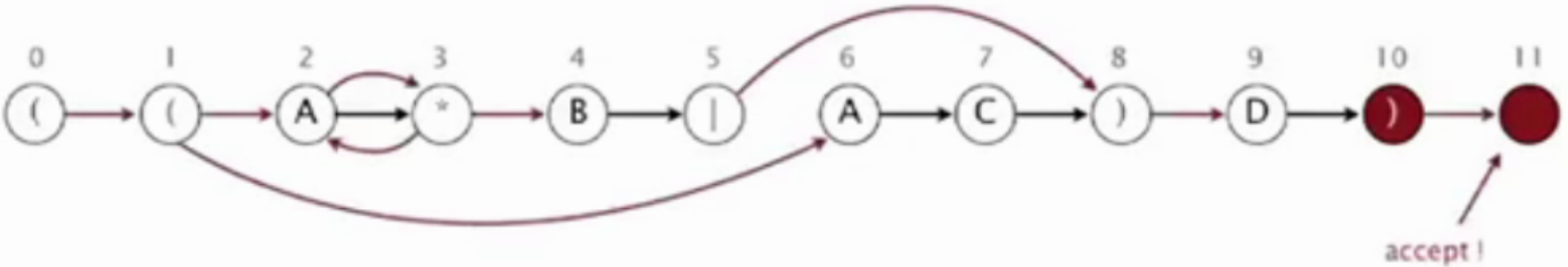


set of states reachable via ε-transitions after matching A A B D : { 10, 11 }

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 num 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]`.

Usaremos DFSpaths

```
dfsPaths DFSpaths(Digraph G, s) {...}
bool hasPath(dfsPaths T, 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 $O(m)$.

Biblioteca NFA: esqueleto

```
/* digrafo das transições epsilon */
static Digraph G;

/* expressão regular */
static char *re;

/* number of caracteres em re */
static int m;

void NFAInit(char *regexp) {...}

bool recognizes(char *txt) {...}
```

NFA: recognizes()

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

```
bool recognizes(char *txt) {
   int i, n = strlen(txt);
   DFSpaths dfs = DFSpathsInit(G, 0);
   Bag pc = bagInit();  /* saco de inteiros */
   Bag match;           /* saco de inteiros */
   for (int v = 0; v < G->V; v++)
      if (hasPath(dfs, v)) add(pc, v);
```

NFA: recognizes()

```
for (i = 0; i < n; i++) {
   match = bagInit();
   bagStartIterator(pc);
   while (bagHasNext(pc)) {
      int v = bagNext(pc);
      if (v == m) continue;
      if (re[v] == txt[i] || re[v] == '.')
         add(match, v+1);
   }
   dfs = DFSpathsInit(G, match);
   pc = bagInit();    bagFree(match);
   for (int v = 0; v < G->V; v++)
      if (hasPathTo(dfs, v)) add(pc, v);
}
```

NFA: recognizes()

```
    /* verifica se aceita */
    bagStartIterator(pc);
    while (bagHasNext(pc)) {
        int v = bagNext(pc);
        if (v == m) {
            bagFree(pc);
            return true;
        }
    }
    bagFree(pc);
    return false;
}
```

Conclusão

O consumo de tempo de `recognizes()` para decidir se uma string `txt` de comprimento `n` pertence à 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: no `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.

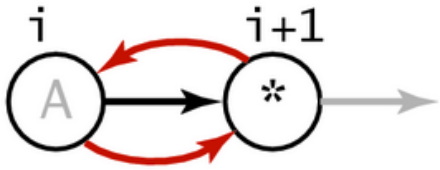
Depois de um caractere,
acrescente ϵ-transições para e do caractere.

Depois de um parênteses, acrescente ϵ-transição
para e do correspondente abre parênteses.

Acrescente uma ϵ-transição para o estado seguinte.

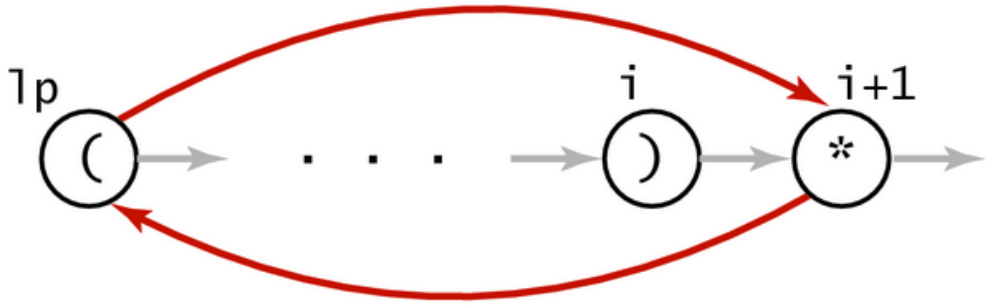
NFA: fecho

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);
```

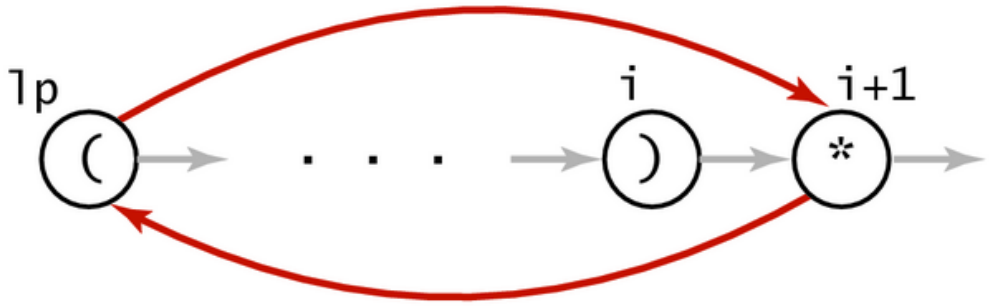
Construção do `nfa`

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 |.

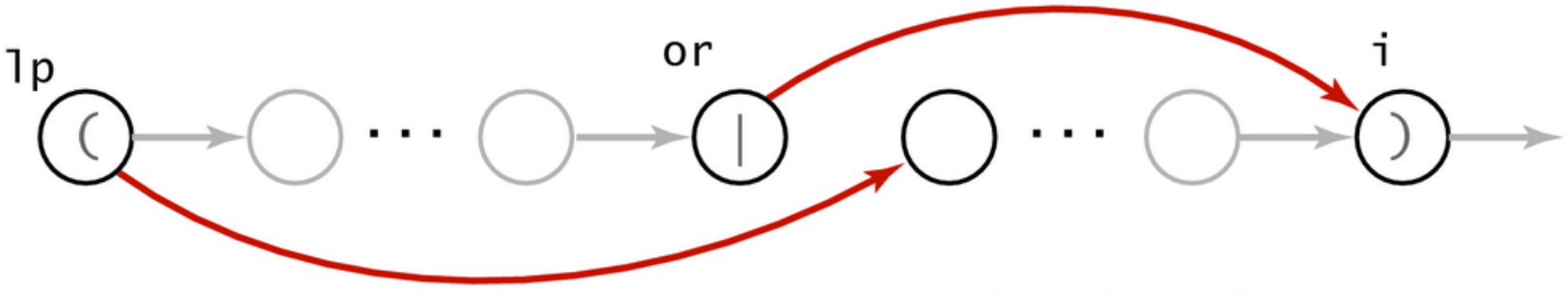
Acrecente uma ϵ-transição de |
para o estado de).

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

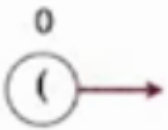
((A * B | A C) D)

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)

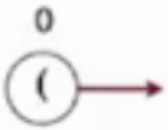
NFA construction demo

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)

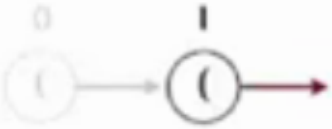
NFA construction demo

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

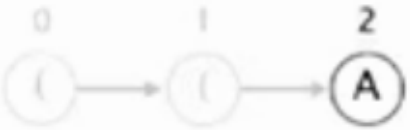
Alphabet symbol.

- 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

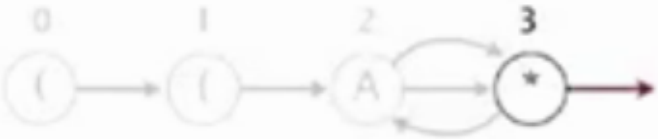
Closure symbol.

- 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.

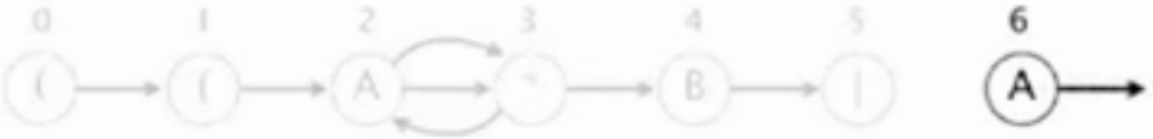
- Add match transition to next state.
- Do one-character lookahead:
  add ε-transitions if next character is *.

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.

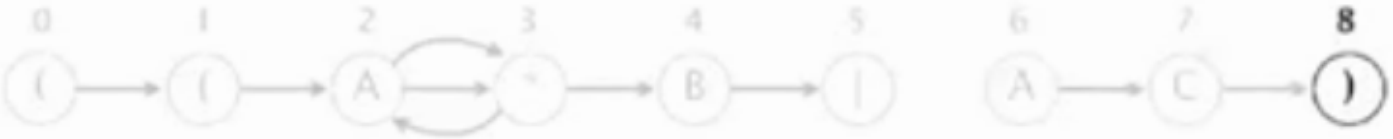
- Add ε-transition to next state.
- Pop corresponding (and possibly intervening |;
  add ε-transition edges for or.
- Do one-character lookahead:
  add ε-transitions if next character is *.

5

1

0

stack



((A * B | A C) D)

NFA construction demo

Right parenthesis.

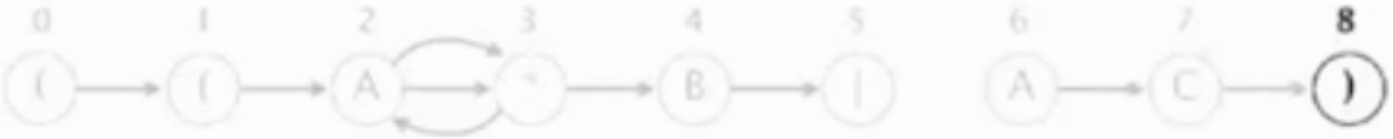
- Add ε-transition to next state.
- Pop corresponding (and possibly intervening |;
  add ε-transition edges for or.
- Do one-character lookahead:
  add ε-transitions if next character is *.

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)

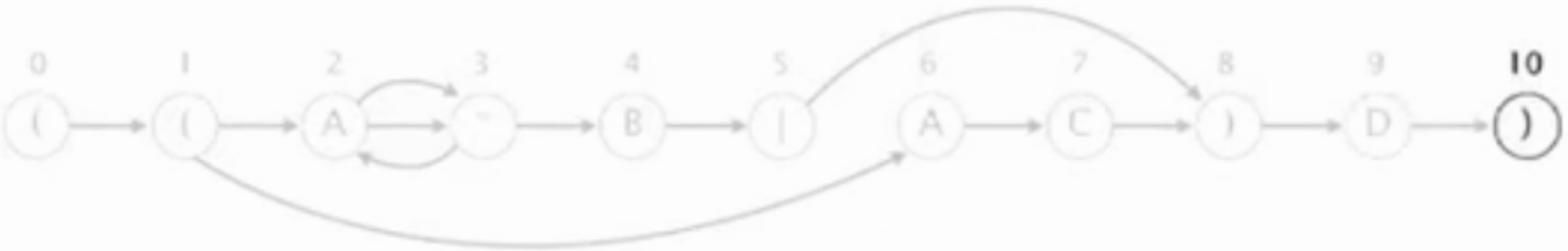
NFA construction demo

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.

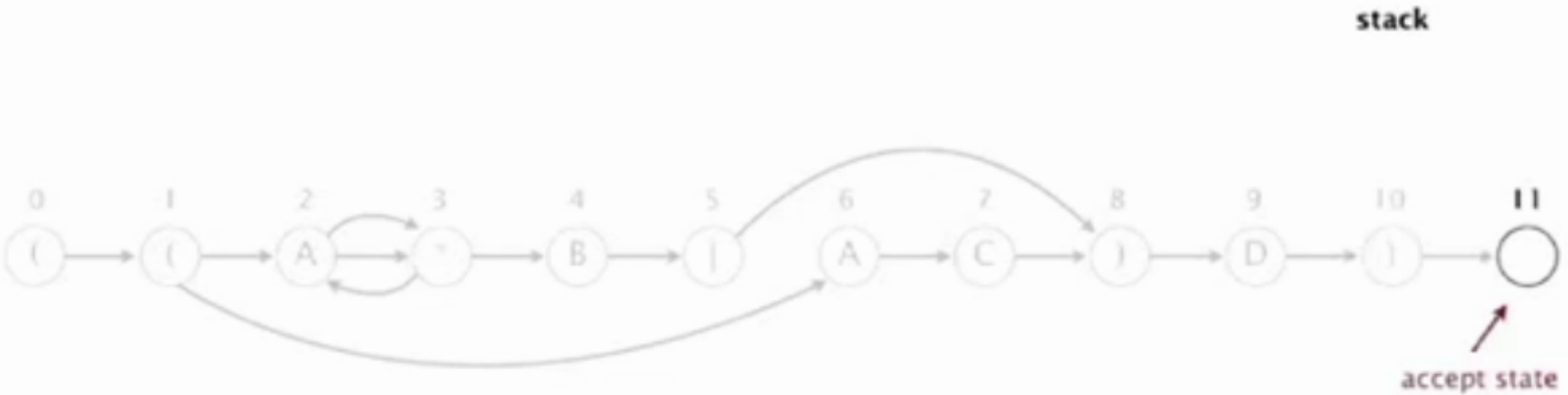
- Add ε-transition to next state.
- Pop corresponding (and possibly intervening |;
  add ε-transition edges for or.
- Do one-character lookahead:
  add ε-transitions if next character is *.

((A * B | A C) D)

NFA construction demo

End of regular expression.

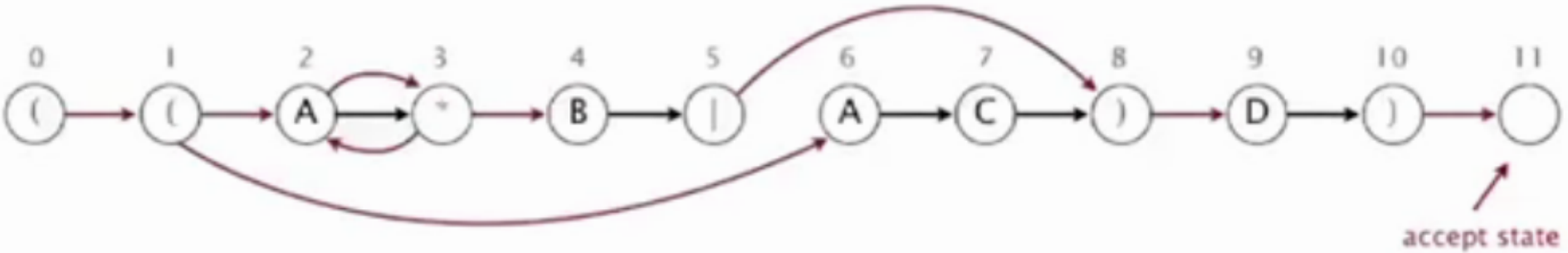
- Add accept state.



((A * B | A C) D)

NFA construction demo

stack



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

NFA: construtor

```
void NFAInit(char *regexp) {
   Stack ops = stackInit();
   m = strlen(regexp);
   re = mallocSafe((m+1)*sizeof(char));
   strcpy(re, regexp);
   G = newDigraph(m+1);
   for (int i = 0; i < m; i++) {
      int lp = i;
      if (re[i] == '(' || re[i] == '|')
         push(ops, i);
```

NFA: construtor

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

NFA: construtor

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

```
void main(int argc, char *argv) {
   char *regexp, *txt;
   int m = strlen(argv[0]);
   regexp = mallocSafe((m+7)*sizeof(char));
   strcpy(regexp, "(.*");
   strcat(regexp, argv[0]);
   strcat(regexp, ".*)");
   NFAInit(regexp);
   while (readLine(&txt))
      if (recognizes(txt))
         printf("%s\n", txt);
}
```

Conclusão

Dada uma 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.

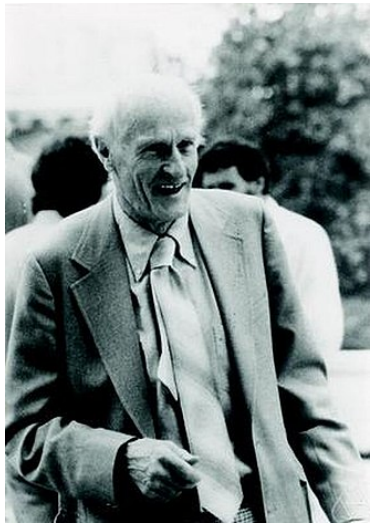
Mais referências

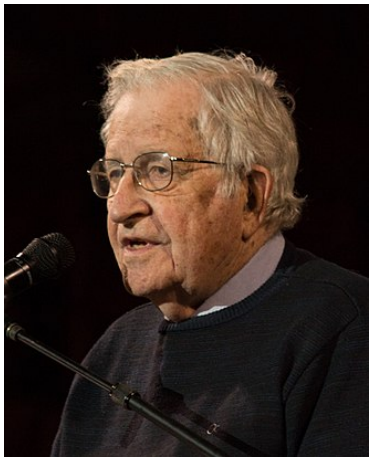
Algumas referências *da hora*:

- 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/

MACO323 – Edição 2020 – segundo semestre

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.

Agradecimentos e lembretes

Agradecimentos ao Prof. Coelho pelos slides...

- EP4: domingo, dia 17 de janeiro
- Exercício sobre remoção preguiçosa: sábado, 19 de dezembro
- Mais um sobre expressões regulares.
- Demais exercícios: quarta-feira, 6 de janeiro

Agradecimentos e lembretes

Algoritmos no próximo semestre:

- MAC0345 Desafios de Programação Avançados
- MAC0320 Introdução à Teoria dos Grafos

Algoritmos no segundo semestre:

- MAC0328 Algoritmos em Grafos
- MAC0414 Autômatos, Computabilidade e Complexidade
- MAC0385 Estruturas de Dados Avançadas
- MAC0473 Otimização Inteira

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