Compacto dos melhores momentos

AULA 25

Busca de substring

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
TGACTAATTTTTTTTATTTATGCAGAGGCCGAGGCCGCCTCGGCCTCTGAGCTATTCCAGAAGTAGTGAGGAGGCTTTT
TTGGAGGCCTAGGCTTTTGCAAAAAGCTGCCCAAGCTGATCCCCGGGGGCAATGAGATATGAAAAAGCCTGAACTCACC
GCGACGTCTGTCGAGAAGTTTCTGATCGAAAAGTTCGACAGCGTCTCCGACCTGATGCAGCTCTCGGAGGGCGAAGAAT
```

Algoritmo KMP

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

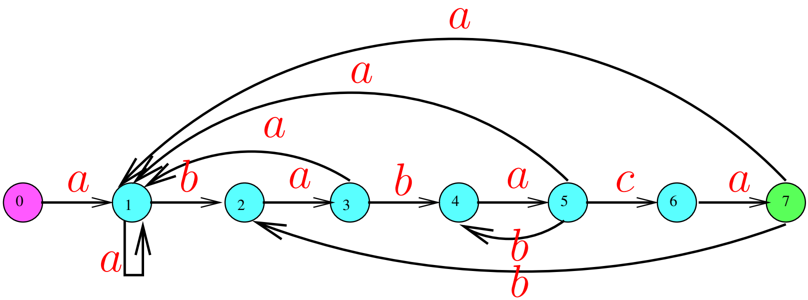
Em cada iteração, o algoritmo sabe qual posição `k` de `pat` deve ser emparelhada com a próxima posição `i+1` de `txt`.

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)



$0..7$ = conjunto de estados

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

δ = função de transição

0 é estado inicial e 7 é estado final

Exemplo: pat = ABABAC

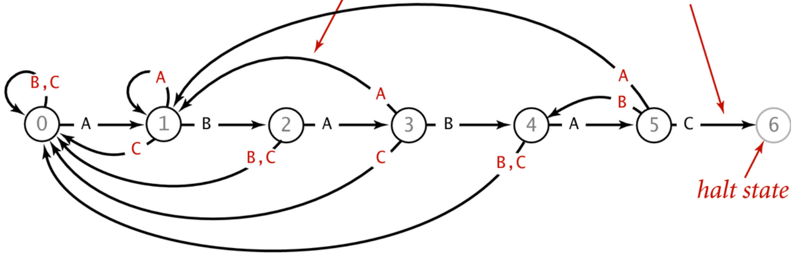
internal representation

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



halt state

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;
}
```

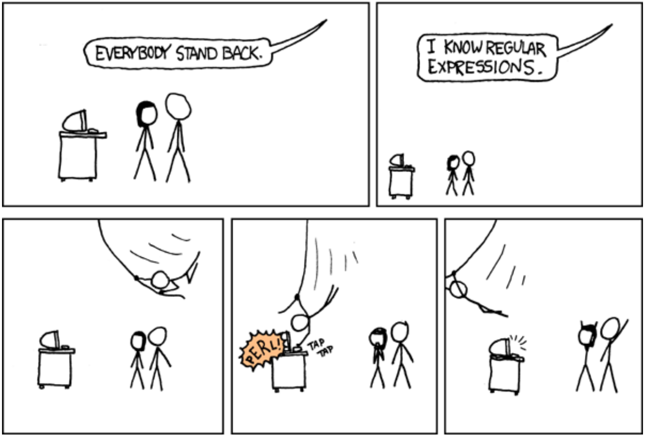
Próximo passo

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 26

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 `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: $(\mathtt{A}(\mathtt{B}*))$ 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

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

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 `dfa` que reconhece as strings representadas por `regexp`.
Para todo `dfa` existe uma `regexp` representa as strings reconhecidas por `dfa`.

Plano

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

- construir um auntômato 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

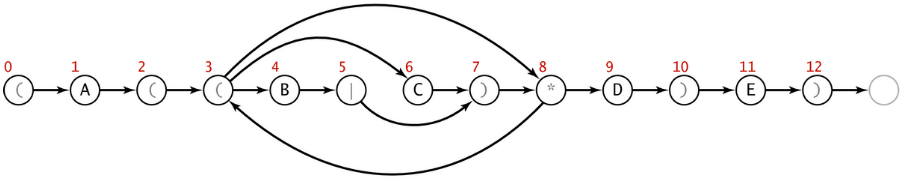
Utilizar outro tipo de autômato.

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).

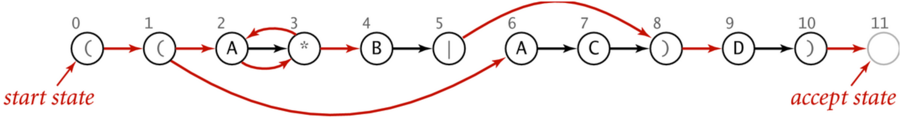
nfa tem um estado para cada caractere na regexp.

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

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

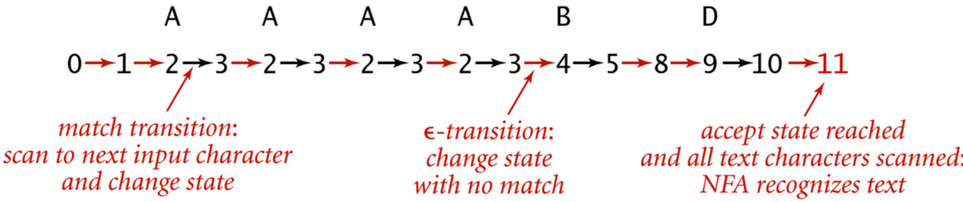
Aceita se **existe** uma sequência de transições, após soletrar todos os caracteres em txt, que termina em um estado de aceite.

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



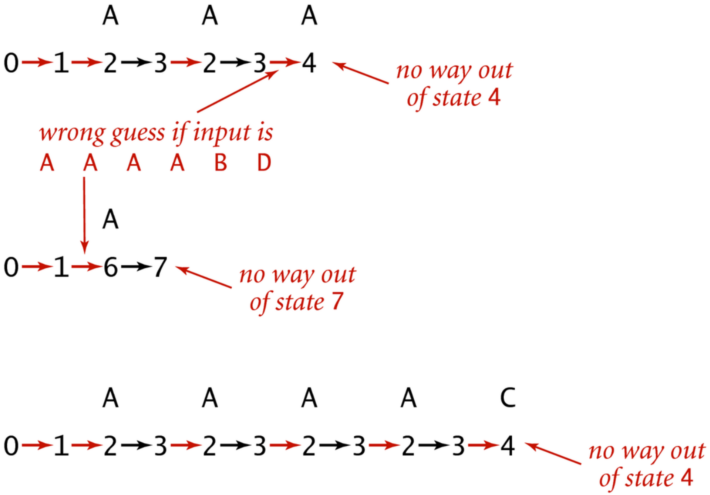
NFA corresponding to the pattern ((A * B | A C) D)

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`.

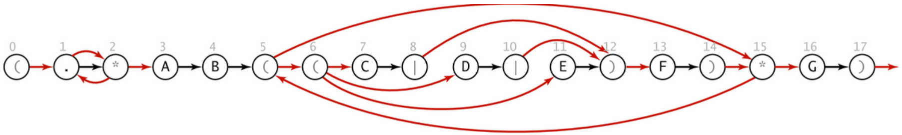
Estados correspondentes a letras tem apenas um **arco preto** saindo para o estado seguinte.

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

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

Rejeita se **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?

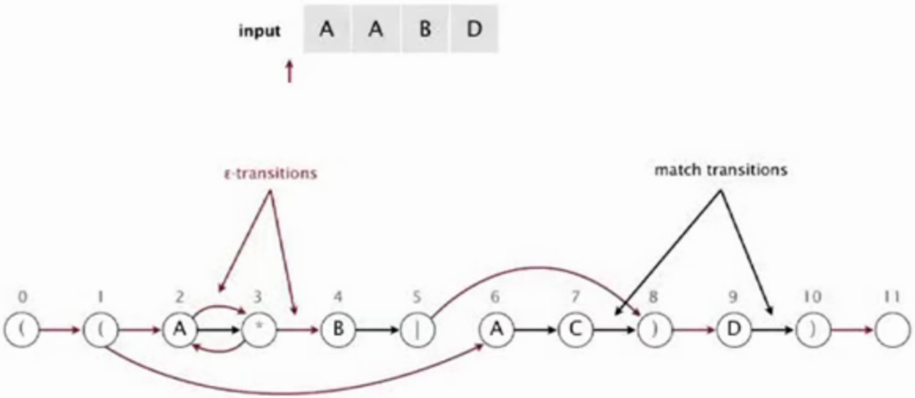
DFA$\Rightarrow$ soletrar `txt`, aplicando **transições pretas**, fácil

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

Para simular a NFA sistematicamente consideramos todas as transições possíveis.

NFA simulation demo

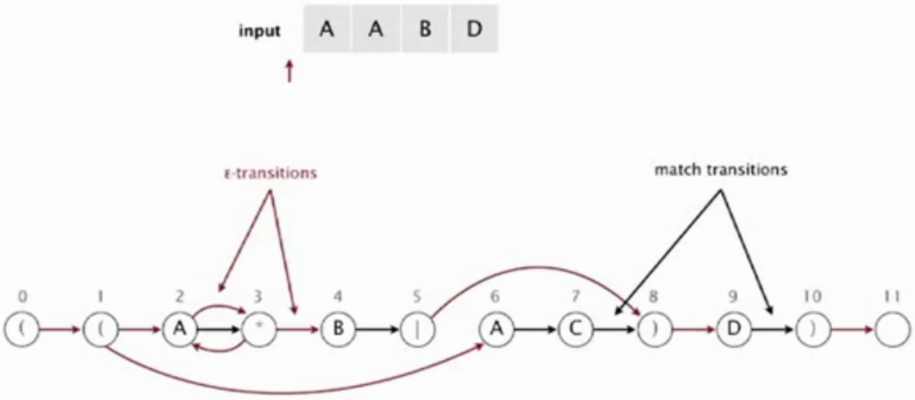
Goal. Check whether input matches pattern.



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

NFA simulation demo

Goal. Check whether input matches pattern.

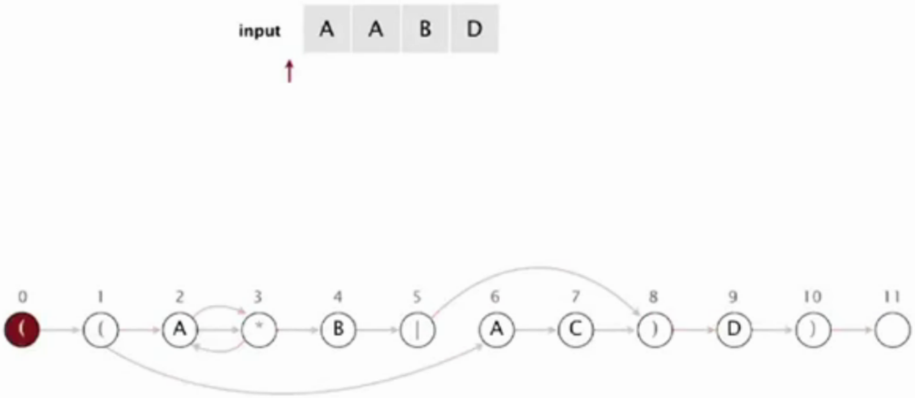


NFA corresponding to the pattern ((A * B | A C) D)

NFA simulation demo

Read next input character.

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

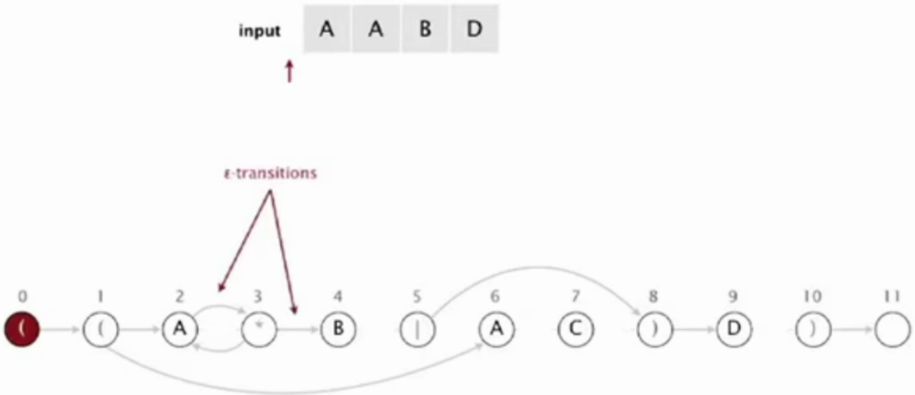


set of states reachable from start: 0

NFA simulation demo

Read next input character.

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

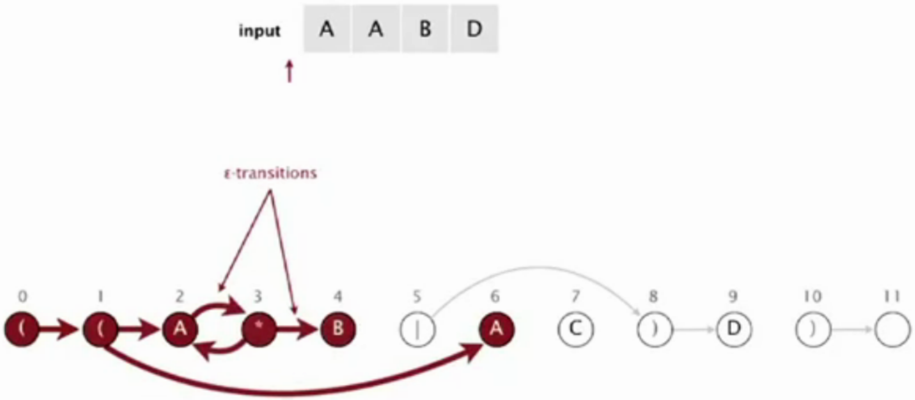


set of states reachable via ε-transitions from start

NFA simulation demo

Read next input character.

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

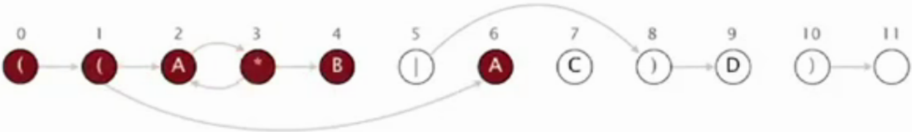


set of states reachable via ε-transitions from start

NFA simulation demo

Read next input character.

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

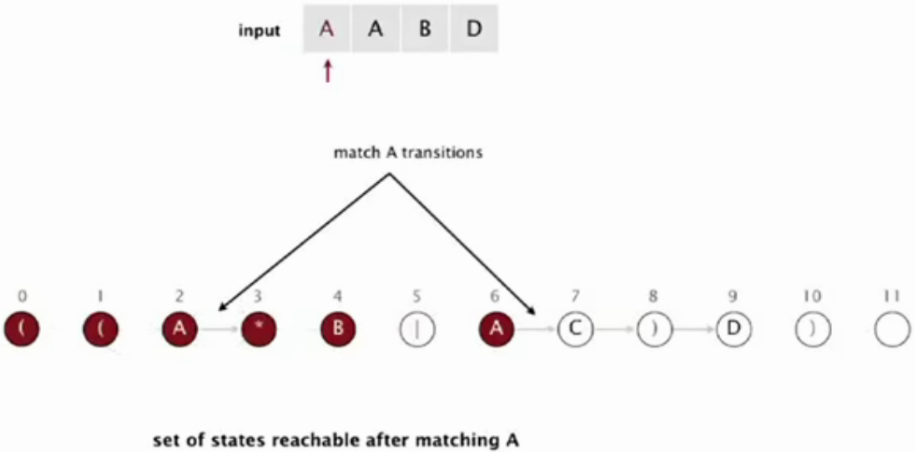


set of states reachable via ε-transitions from start : { 0, 1, 2, 3, 4, 6 }

NFA simulation demo

Read next input character.

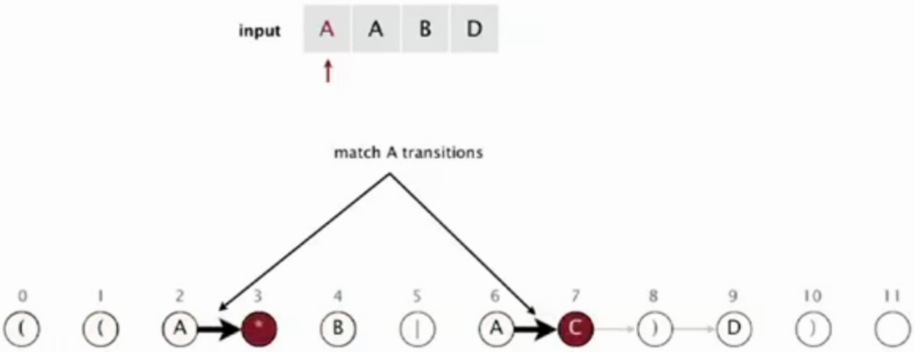
- Find states reachable by match transitions.
- Find states reachable by ε-transitions



NFA simulation demo

Read next input character.

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

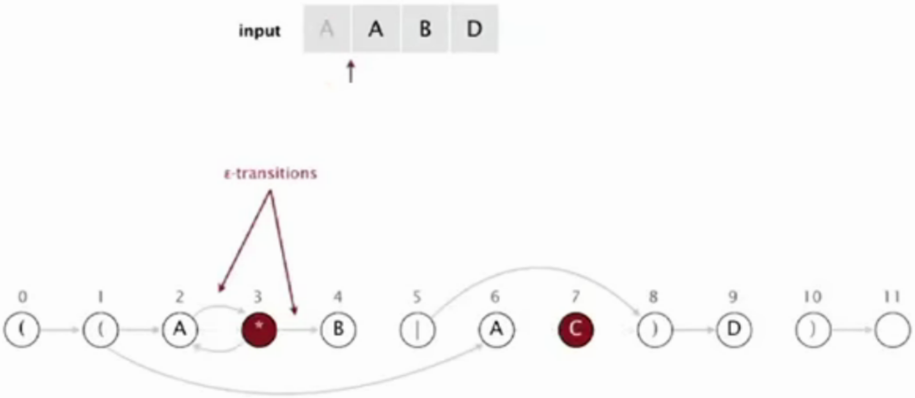


set of states reachable after matching A

NFA simulation demo

Read next input character.

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

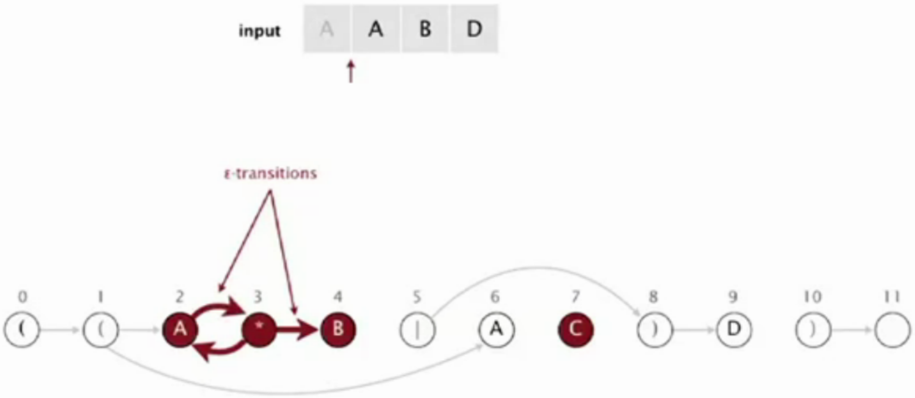


set of states reachable via ε-transitions after matching A

NFA simulation demo

Read next input character.

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

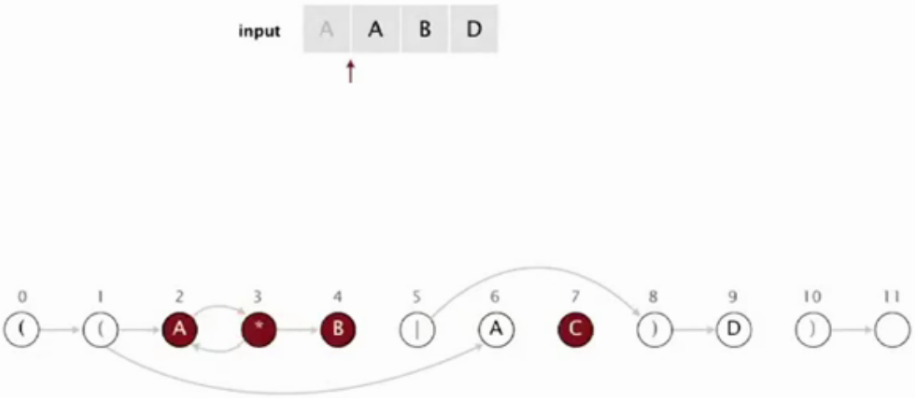


set of states reachable via ε-transitions after matching A

NFA simulation demo

Read next input character.

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

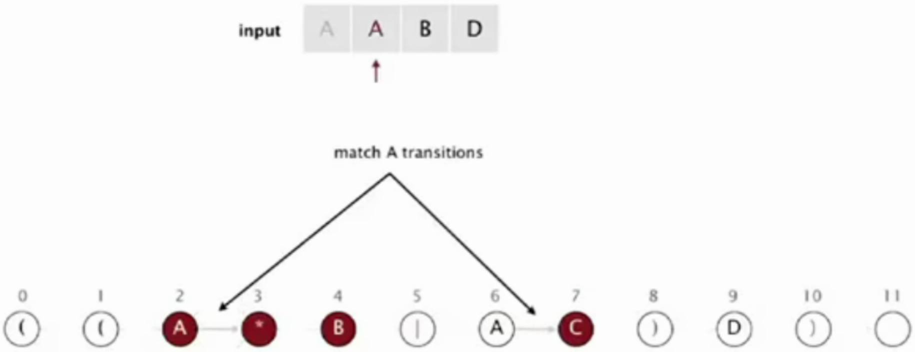


set of states reachable via ε-transitions after matching A : { 2, 3, 4, 7 }

NFA simulation demo

Read next input character.

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

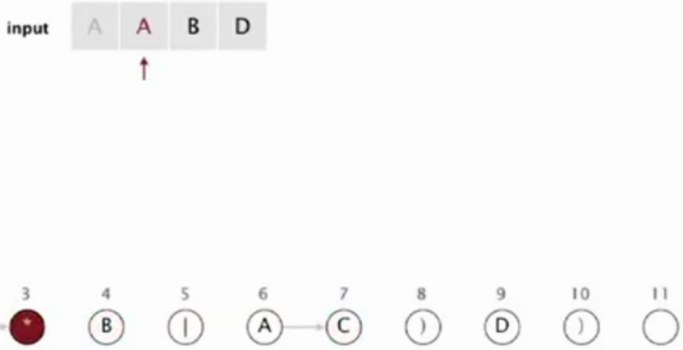


set of states reachable after matching A A

NFA simulation demo

Read next input character.

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

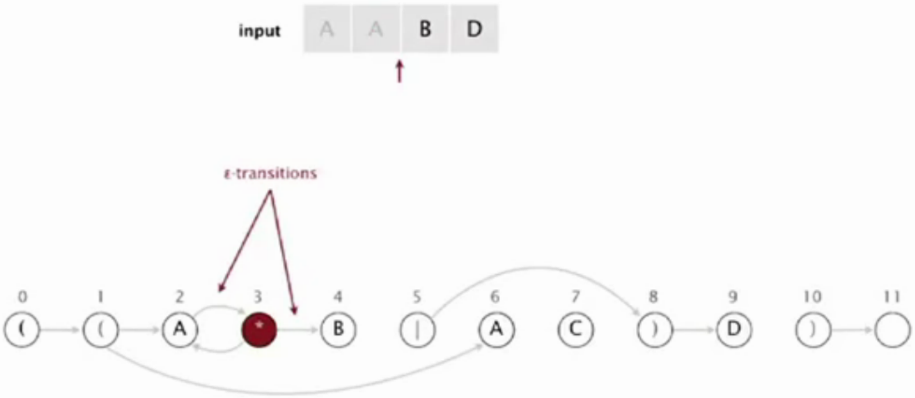


set of states reachable after matching A A : { 3 }

NFA simulation demo

Read next input character.

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

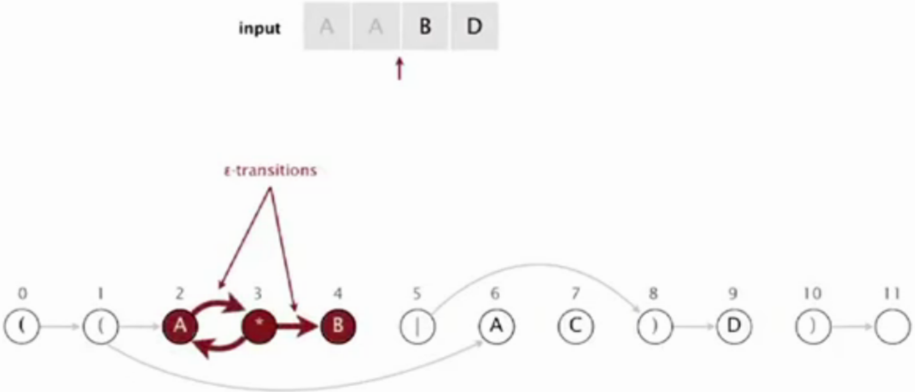


set of states reachable via ε-transitions after matching A A

NFA simulation demo

Read next input character.

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

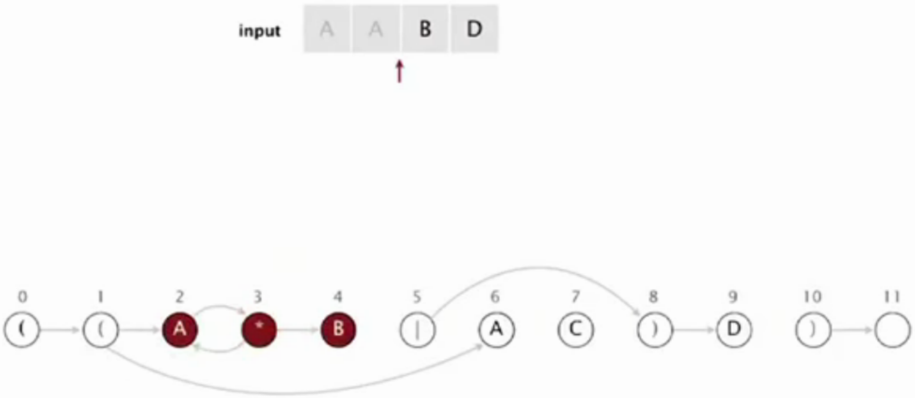


set of states reachable via ε-transitions after matching A A

NFA simulation demo

Read next input character.

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

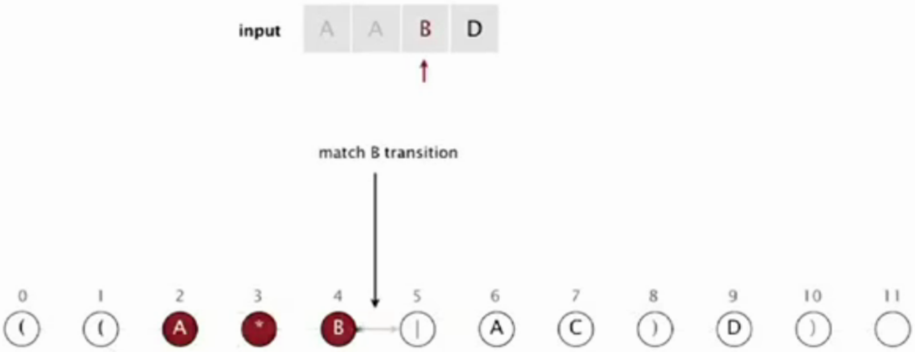


set of states reachable via ε-transitions after matching A A : { 2, 3, 4 }

NFA simulation demo

Read next input character.

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

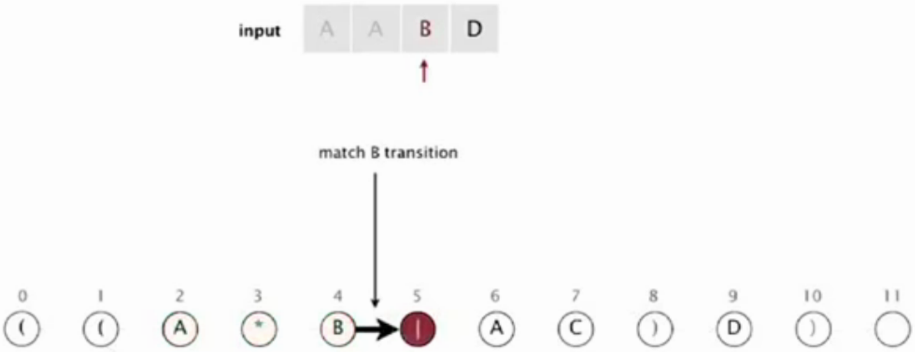


set of states reachable after matching A A B

NFA simulation demo

Read next input character.

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

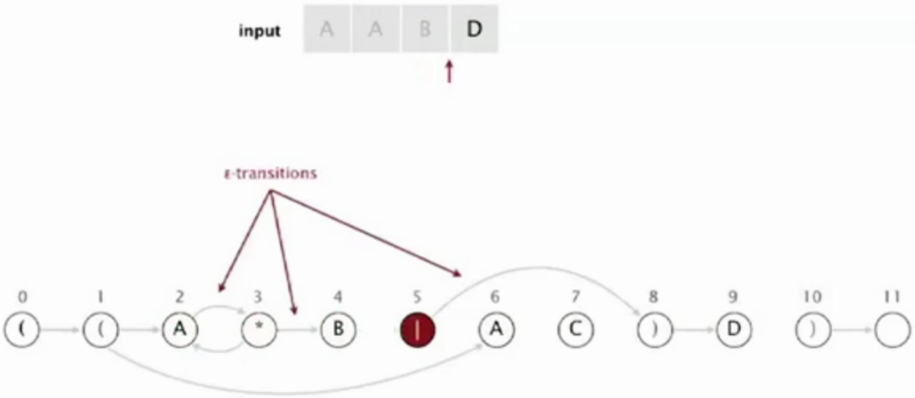


set of states reachable after matching A A B

NFA simulation demo

Read next input character.

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

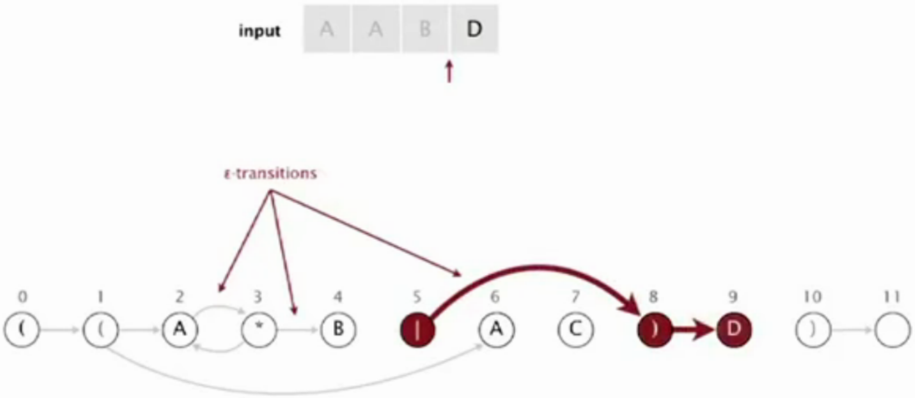


set of states reachable via ε-transitions after matching A A B

NFA simulation demo

Read next input character.

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

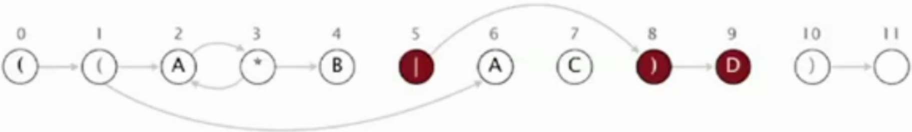


set of states reachable via ε-transitions after matching A A B

NFA simulation demo

Read next input character.

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

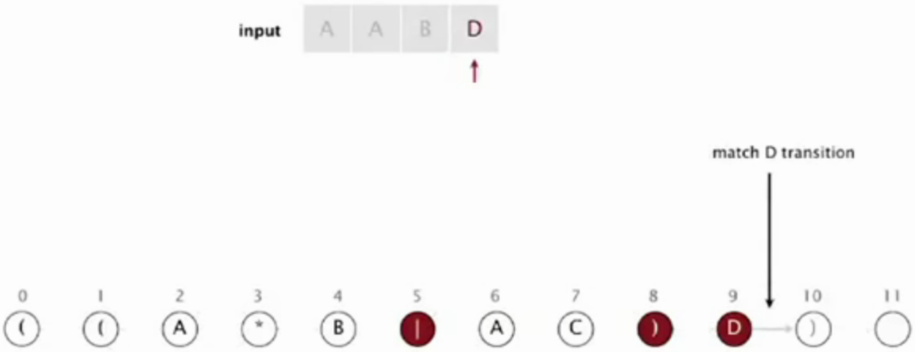


set of states reachable via ε-transitions after matching A A B : { 5, 8, 9 }

NFA simulation demo

Read next input character.

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

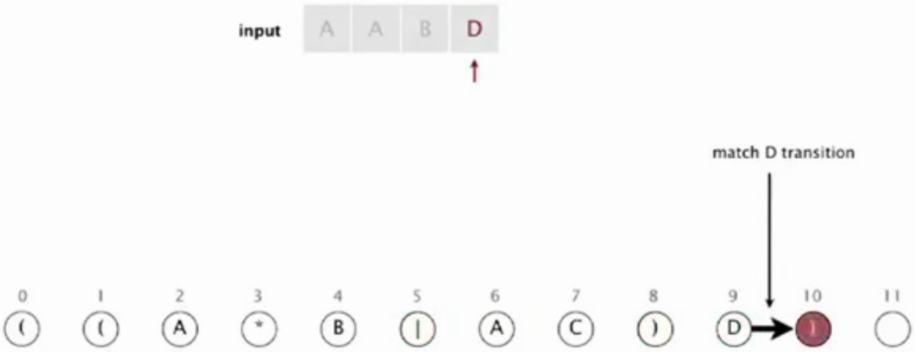


set of states reachable after matching A A B D

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.

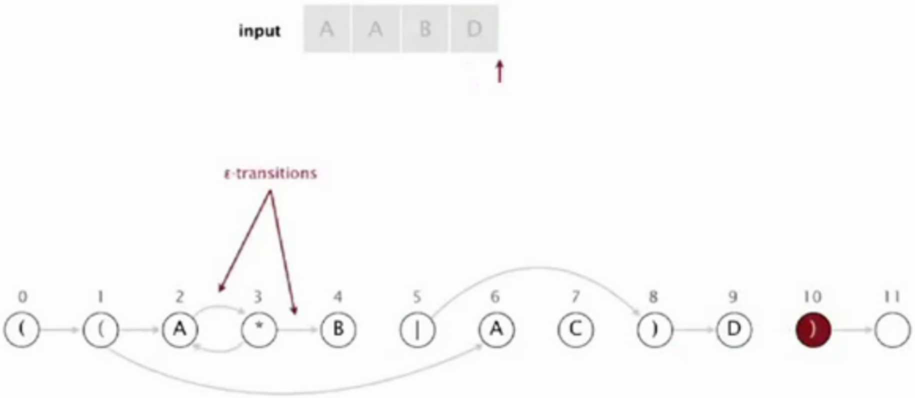
- Find states reachable by match transitions.
- Find states reachable by ε-transitions

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

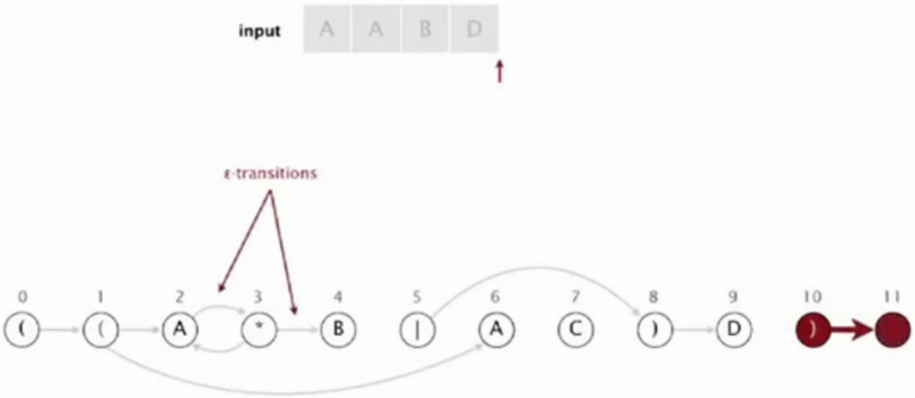


set of states reachable via ε-transitions after matching A A B D

NFA simulation demo

Read next input character.

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

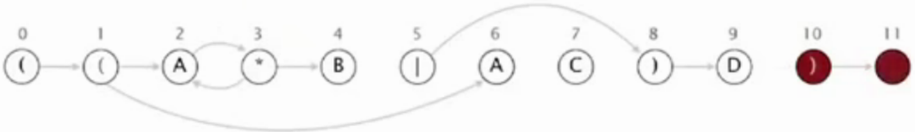


set of states reachable via ε-transitions after matching A A B D

NFA simulation demo

Read next input character.

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

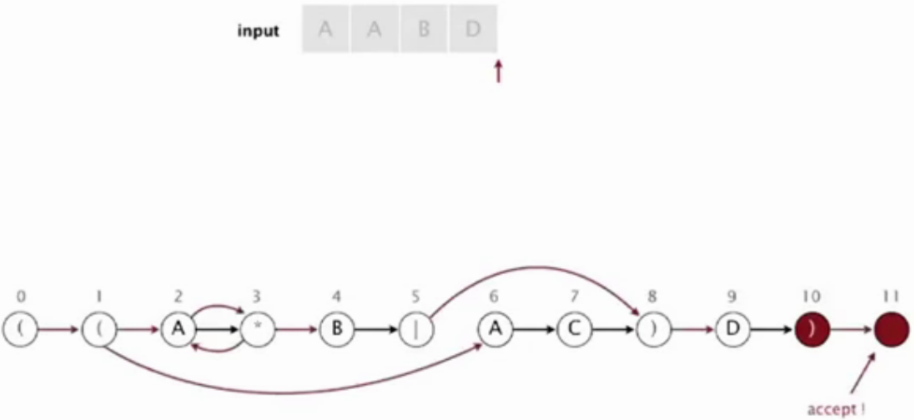


set of states reachable via ε-transitions after matching A A B D : { 10, 11 }

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 $E \leq 3m$ 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 `n m`.

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.

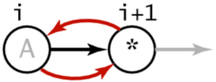
Parenteses: acrescente uma ϵ-transição para o estado seguinte.

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 e do correspondente abre parênteses.
Acrescente uma ϵ-transição para o estado seguinte.

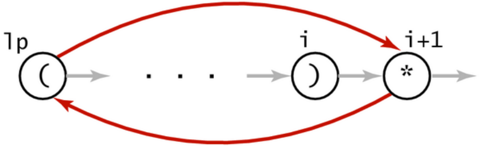
NFA: fecho

single-character closure




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

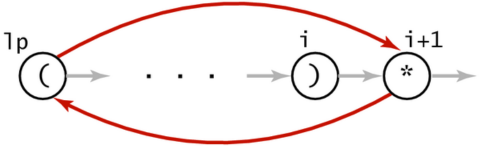
closure expression




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

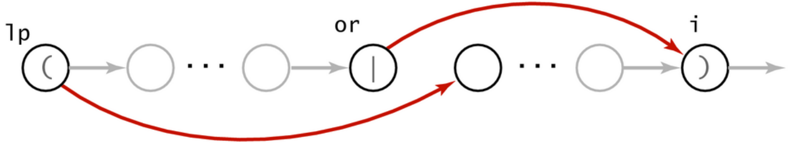
Construção do nfa

ou: temos $(\mathtt{re}_1|\mathtt{re}_2)$ onde $\mathtt{re}_1$ e $\mathtt{re}_2$ 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

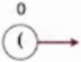
((A * B | A C) D)

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)

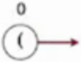
NFA construction demo

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.

NFA construction demo

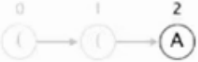
Alphabet symbol.

- 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

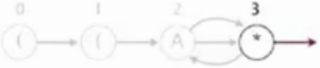
stack

((A * B | A C) D)

NFA construction demo

Closure symbol.

- Add ε-transition to next state.



((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

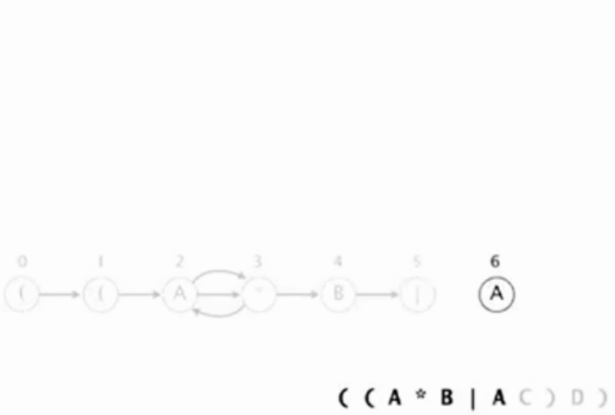
stack

((A * B | A C) D)

NFA construction demo

Alphabet symbol.

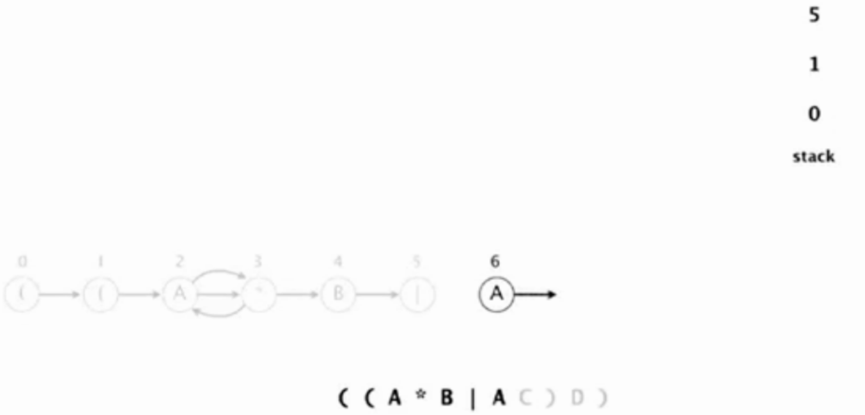
- Add match transition to next state.
- Do one-character lookahead:
  add ε-transitions if next character is *.



NFA construction demo

Alphabet symbol.

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



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.

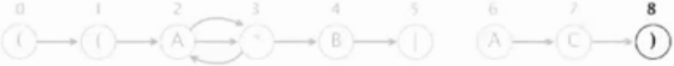
- Add ε-transition to next state.
- Pop corresponding (and possibly intervening |;
  add ε-transition edges for or.
- Do one-character lookahead:
  add ε-transitions if next character is *.

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 *.

((A * B | A C) D)

NFA construction demo

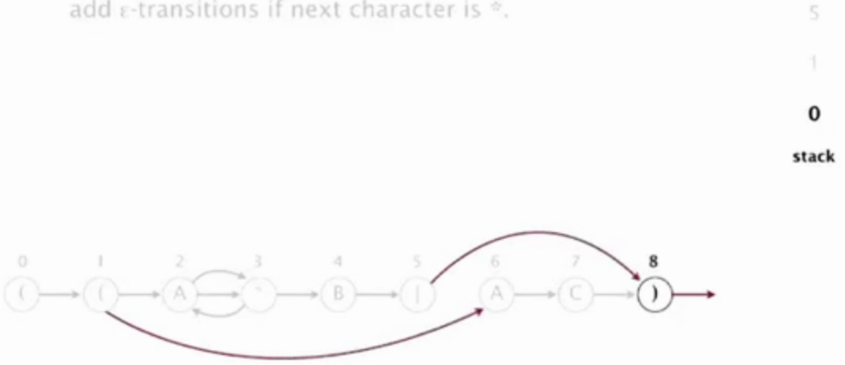
Right parenthesis.

- 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 *.

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

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 *.

((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

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

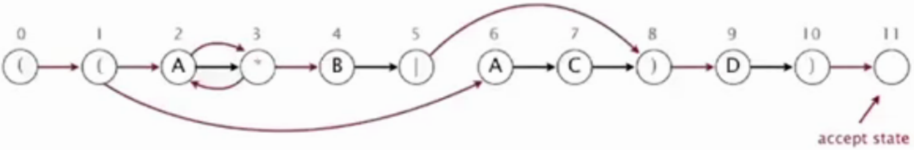
End of regular expression.

- 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 para construir um `NFA` correspondente a uma `regexp` de comprimento `m` cosome tempo e espaço 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"
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaac
1.6 seconds
java Validate "(a|aa)*b"
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac
3.7 seconds
java Validate "(a|aa)*b"
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac
9.7 seconds
java Validate "(a|aa)*b"
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac
23.2 seconds
java Validate "(a|aa)*b"
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac
62.2 seconds
java Validate "(a|aa)*b"
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac
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;