Automata Examples

Operations on Regular Languages

Automata Representing Operations on Regular Languages

Hamming and Levenshtein Distance

Approximate Text Search

Automaton Bit Arrays Simulation
Automaton $C_1$ accepts union of sets
$L_1 = \{00, 0011, 001100, 00110011, 0011001100, ...\} \cup \{11, 1100, 110011, 11001100, 1100110011, ...\}$.

Automaton $C_2$ accepts language $L_2$ over $\Sigma = \{0, 1\}$, in each word of $L_2$:
-- there is at least one symbol 1,
-- each symbol 1 is followed by exactly two or three symbols 0.
Automaton $C_3$ accepts all binary nonnegative integers divisible by 3, any number of leading zeros may be included.

Automaton $C_4$ accepts all binary positive integers divisible by 3, no leading zeros are allowed.
Language operations

Operations on regular languages revisited

Let $L_1$ and $L_2$ be any languages. Then

$L_1 \cup L_2$ is union of $L_1$ and $L_2$. It is a set of all words which are in $L_1$ or $L_2$.

$L_1 \cap L_2$ is intersection of $L_1$ and $L_2$. It is a set of all words which are simultaneously in $L_1$ and $L_2$.

$L_1 \cdot L_2$ is concatenation of $L_1$ and $L_2$. It is a set of all words $w$ for which holds $w = w_1w_2$ (concatenation of words $w_1$ and $w_2$), where $w_1 \in L_1$ and $w_2 \in L_2$.

$L_1^*$ is Kleene star or Kleene closure or iteration of language $L_1$. It is a set of all words which are concatenations of any number (incl. zero) of any words of $L_1$ in any order.

Closure

Whenever $L_1$ and $L_2$ are regular languages then $L_1 \cup L_2$, $L_1 \cap L_2$, $L_1 \cdot L_2$, $L_1^*$ are regular languages too.

Automata support

When $L_1$ is regular language accepted by automaton $A_1$ and $L_2$ is regular language accepted by automaton $A_2$ then there also are automata $A_3$, $A_4$, $A_5$, $A_6$, which accept $L_1 \cup L_2$, $L_1 \cap L_2$, $L_1 \cdot L_2$, $L_1^*$, respectively.
Automaton $A_3$ accepting union of two regular languages $L_1$, $L_2$ accepted by automata $A_1$, $A_2$ respectively.

Automaton $A_3$ is constructed using $A_1$ and $A_2$:

Do not change $A_1$ and $A_2$.
Create new additional start state $S_0$, add $\varepsilon$-transitions from $S_0$ to start states $S_1$ and $S_2$ of $A_1$ and $A_2$ respectively.
Define set of final states of $A_3$ as union of final states of $A_1$ and $A_2$. 
Automaton $B_3$ accepts any word from sets
{00, 0011, 001100, 00110011, 0011001100, ...}
{11, 1100, 110011, 11001100, 1100110011, ...}
and also any binary nonnegative integer divisible by 3
with any number of leading zeros
Automaton $A_5$ accepting concatenation of two regular languages $L_1, L_2$ accepted by automata $A_1, A_2$ respectively.

Automaton $A_5$ is constructed using $A_1$ and $A_2$:

- Do not change $A_1$ and $A_2$.
- Add $\varepsilon$ - transitions from each final state $F_k$ of $A_1$ to the start state $S_2$ of $A_2$.
- Define start state of $A_5$ to be equal to the start state of $A_1$.
- Define set of final states of $A_5$ to be equal to the set of final states of $A_2$. 

![Diagram showing the construction of $A_5$ from $A_1$ and $A_2$.]
Automaton $B_5$ accepts any word over \{0, 1\} which can be split into two consecutive words $w_1$ and $w_2$, where word $w_1$ is described by regular expression $0^*(100+1000)(100+1000)^*$, word $w_2$ represents binary positive integer divisible by 3 w/o leading 0's.
Automaton $A_{6}$ accepting iteration of language $L_{1}$ accepted by automaton $A_{1}$.

Automaton $A_{6}$ is constructed using $A_{1}$:

Do not change $A_{1}$.
Create new aditional start state $S_{0}$ and add $\varepsilon$ - transition from $S_{0}$ to start state $S_{1}$ of $A_{1}$
Add $\varepsilon$ - transitions from all final states $F_{k}$ of $A_{1}$ to state $S_{1}$.
Define start state of $A_{6}$ to be $S_{0}$.
Define set of final states of $A_{6}$ as union of final states $F_{k}$ and $S_{0}$.
Automaton $B_6$ accepts any word created by concatenation and repetition of any words accepted by $C_2$ including empty word.

Maybe you can find some more telling informal description of the corresponding language?
Automaton $A_4$ accepting intersection of two regular languages $L_1$, $L_2$ accepted by automata $A_1$, $A_2$ respectively.

Automaton $A_4$ is constructed using $A_1$ and $A_2$:

Create Cartesian product $Q_1 \times Q_2$, where $Q_1$, $Q_2$ are sets of states of $A_1$, $A_2$.

Each state of $A_4$ will be an ordered pair of states of $A_1$, $A_2$.

State $(S_1, S_2)$ will be start state of $A_4$, where $S_1, S_2$ are start states of $A_1$, $A_2$.

Final states of $A_4$ will be just those pairs $(F, G)$, where $F$ is a final state of $A_1$ and $G$ is a final state of $A_2$.

Create transition from state $(p_1, p_2)$ to $(q_1, q_2)$ in $A_4$ labeled by symbol $x$ if and only if

there is a transition $p_1 \rightarrow q_1$ labeled by $x$ in $A_1$ and also
there is a transition $p_2 \rightarrow q_2$ labeled by $x$ in $A_2$. 
Scheme of an automaton $A_4$ accepting the intersection of two regular languages $L_1$, $L_2$ accepted by automata $A_1$, $A_2$ respectively.
Automaton $A_4$ accepting binary integers divisible by 3 ($C_4$) in which each symbol 1 is followed by exactly two or three symbols 0 ($C_2$).
Hamming distance

Hamming distance of two strings is equal to \( k \) (\( k \leq 0 \)), whenever \( k \) is the minimal number of rewrite operations which when applied on one of the strings produce the other string. Rewrite operation rewrites one symbol of the alphabet by some other symbol of the alphabet. Symbols cannot be deleted or inserted. Hamming distance is defined only for pairs of strings of equal length.

**Informally:** Align the strings and count the number of mismatches of corresponding symbols.
Automaton $A_1$ for approximate pattern matching. It detects all occurrences of substrings whose Hamming distance from the pattern $p_1 p_2 p_3 p_4$ is less or equal to 3.
Automaton $A_2$ for approximate pattern matching. It detects all occurrences of substrings whose Hamming distance from the pattern 'rose' is less or equal to 3.

Automaton $A_2$ detects among others also the words:

- rose (distance = 0)
- dose (distance = 1)
- rest (distance = 2)
- list (distance = 3)
- and more...
NFA accepting any word with subsequence $p_1p_2p_3p_4$ anywhere in it.

Example

NFA accepting any word with subsequence $p_1p_2p_3p_4$ anywhere in it, one symbol in the sequence may be altered.

Alternatively: NFA accepting any word containing a subsequence $Q$ whose Hamming distance from $p_1p_2p_3p_4$ is at most 1.
Hamming distance of the found pattern Q from pattern P = "rose" cannot be deduced from the particular end state. E.g.: "rope":
\[ r - 1 \cdot o - 2 \cdot p - 7 \cdot e - 8. \]
\[ r - 5 \cdot o - 6 \cdot p - 10 \cdot e - 11. \]

Notation: \( \overline{x} = \Sigma - \{x\} \) means: Complement of x in \( \Sigma \).

Hamming distance from the pattern P = "rose" to the found pattern Q corresponds exactly to the end state.
Levenshtein distance

Levenshtein distance of two strings A and B is such minimal \( k (k \geq 0) \), that we can change A to B or B to A by applying exactly \( k \) edit operations on one of A or B. The edit operations are Remove, Insert or Rewrite any symbol of the alphabet anywhere in the string. (Rewrite is also called Substitution.)

Levenshtein distance is defined for any two strings over a given alphabet.

\[
\begin{align*}
\text{B R U X E L L E S} & \quad \text{Delete X.} \\
\text{B E T E L G E U S E} & \quad \text{Rewrite R->E, U->T, L->G.} \\
\text{Distance} & = 6 \\
\text{Insert U, E.}
\end{align*}
\]

Note

Although the distance is defined unambiguously (prove!), the particular edit operations transforming one string to another may vary (find an example).
**Calculating Levenshtein distance**

Apply a simple Dynamic Programming approach.

Let $A = a[1].a[2]. \ldots .a[n] = A[1..n]$, $B = b[1].b[2]. \ldots .b[m] = b[1..m]$, $n, m \geq 0$.

\[
Dist(A, B) = |m - n| \quad \text{if } n = 0 \text{ or } m = 0
\]

\[
Dist(A, B) = 1 + \min (\text{Dist}(A[1..n - 1], B[1..m]), \text{Dist}(A[1..n], B[1..m - 1]), \text{Dist}(A[1..n - 1], B[1..m - 1])) \quad \text{if } n > 0 \text{ and } m > 0
\]

\[
Dist(A, B) = \text{Dist}(A[1..n - 1], B[1..m - 1]) \quad \text{if } n > 0 \text{ and } m > 0 \text{ and } A[n] = B[m]
\]

**Calculation corresponds to** ... **Operation**

- $1 + \text{Dist}(A[1..n - 1], B[1..m])$, ... **Insert**($A$, $n - 1$, $B[m]$) or **Delete**($B$, $m$)
- $1 + \text{Dist}(A[1..n], B[1..m - 1])$, ... **Insert**($B$, $m - 1$, $A[n]$) or **Delete**($A$, $n$)
- $1 + \text{Dist}(A[1..n - 1], B[1..m - 1])$, ... **Rewrite**($A$, $n$, $B[m]$) or **Rewrite**($B$, $m$, $A[n]$)
Levenshtein distance

Dist("BETELGEUSE","BRUXELLES") = 6

```
B   E   T   E   L   G   E   U   S   E
0  1  2  3  4  5  6  7  8  9  10
B  1  0  1  2  3  4  5  6  7  8  9
R  2  1  1  2  3  4  5  6  7  8  9
U  3  2  2  2  3  4  5  6  6  7  8
X  4  3  3  3  3  4  5  6  7  7  8
E  5  4  3  4  3  4  5  5  6  7  7
L  6  5  4  4  4  3  4  5  6  7  8
L  7  6  5  5  5  4  4  5  6  7  8
E  8  7  6  6  5  5  5  4  5  6  7
S  9  8  7  7  6  6  6  5  5  5  6
```

D[0][j] = j; D[i][0] = i;

```
for( i = 1; i <= n; i++ )
    for( j = 1; j <= m; j++ )
            D[i][j] = D[i-1][j-1];
        else
            D[i][j] = 1+ min(D[i-1][j-1], D[i-1][j], D[i][j-1]);
```
NFA searches in a text for a string within Levenshtein distance 3 from the pattern "rose".

More transitions than in Hamming distance NFA

vertical ... Insert operation
epsilon ... Delete operation

Self-check question
Label vertical transitions by $\Sigma$ (whole alphabet). How will it change the functionality of this NFA?
Bit representation of NFA

Size of transition table $T$ is $|Q| \times |\Sigma|$ and each its element $T[i, k]$ corresponds to state $q_i \in Q$ and symbol $a_k \in \Sigma$. $T[i, k]$ is vector of length $|Q|$ and it holds: $T[i, k][j] = 1 \iff q_j \in \delta(q_i, a_k)$.

For bit vector $F$ of final states holds $F[j] = 1 \iff q_j \in F_A$.

Automaton $A$ detects pattern $aba$ in a text.
Text Search

starting configuration

sets of states represented by bit arrays during computation

A

sets of states

Text Symbols:

- text symbols:
  - a, c, a, b, c, a, a, b, a

<table>
<thead>
<tr>
<th>Time</th>
<th>State 0</th>
<th>State 1</th>
<th>State 2</th>
<th>State 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>i=0</td>
<td>1 1 1 1</td>
<td>1 1 1 1</td>
<td>1 1 1 1</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>i=1</td>
<td>0 1 0 0</td>
<td>0 0 1 1</td>
<td>0 0 1 1</td>
<td>0 0 1 1</td>
</tr>
<tr>
<td>i=2</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>i=3</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
</tr>
</tbody>
</table>

Example:

Automaton is in states \{0,1\}, it reads symbol b

\[ F = 00001 \]
Simulation of work of a NFA without $\varepsilon$-transitions
Basic method, implemented with bit vectors.

Input: Bit table $T$ of transitions, bit vector $F$ of final states,
number of states $Q$.size, text in array $t$ (indexed from 1).
Output: Simulated run and output of the automaton.
(notation in format $[0101\ldots00]$ denotes characteristic vector of set of states)

```plaintext
S[0] = [100..0]; i = 1;  // init
while( (i <= t.length) && (S[i-1]!=000...0) ) {
    for( j=0; j < Q.size; j++ )
        if( (S[i][j] == 1) && (F[j] == 1) )
            print( q[j].final_state_info );
    S[i] = [000…0];
    for( j=0; j < Q.size; j++ )
        if( S[i-1][j]==1 )
            S[i] = S[i] | T[j][t[i]]; // "|
    i++;
}
```