

Finite State Automata for Unicode

Tom Emerson
Sr. Computational Linguist

strategy • process • technology • results

www.basistech.com

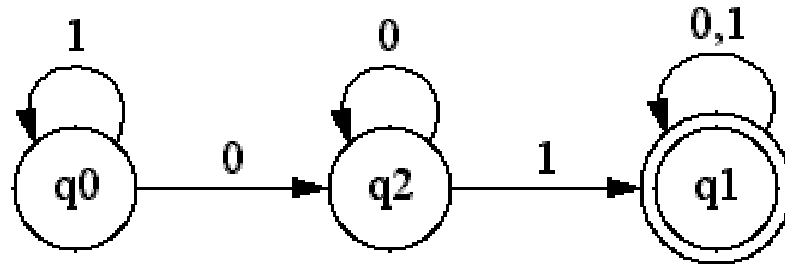
Overview

- What are Finite State Automata (FSA)?
 - Formal definition
- Unicode and FSA
- Solutions
- Conclusions

Finite State Automata

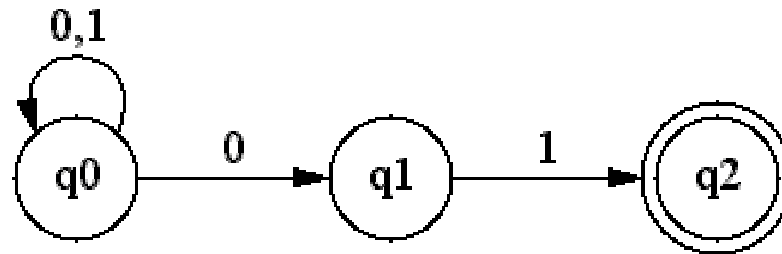
- An FSA consists of
 - A set of “states”
 - “Control” moves from state to state in response to some “input”
- Deterministic Automata
 - The machine is in only one state at a given time
- Non-deterministic Automata
 - The machine can be in multiple states at a given time

Deterministic FSA



	0	1
q0	q2	q0
*q1	q1	q1
q2	q2	q1

Non Deterministic FSA



	0	1
q0	{q0,q1}	{q1}
q1	-	{q2}
*q2	-	-

Formal Definition

- Formally a DFA is defined by
 - $A = (Q, \Sigma, \delta, q_0, F)$
 - Q is a finite set of states
 - Σ is a finite set of symbols, the alphabet
 - δ is a transition function, $Q \times \Sigma \rightarrow Q \cup \{\perp\}$
 - $q_0 \in Q$ is the start state
 - F is a set of final states, $F \subseteq Q$
- We deal with acyclic automata
 - $\delta(q_n, a) \neq q_n$

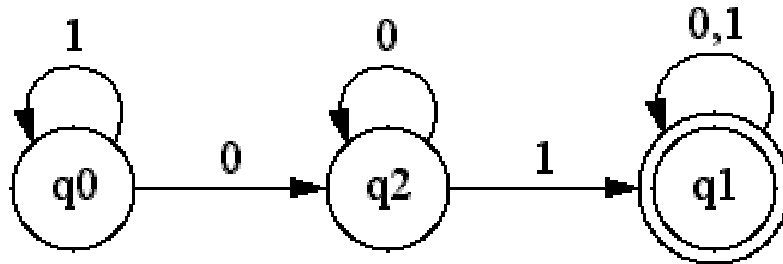
Unicode and ADFSA

- When dealing with Unicode, in the general case, the size of Σ is huge.
- Choosing the representation of the machine is important for both space and time complexity
- For regular expressions entire blocks of characters may be specified requiring more advanced representations than simple character tables.
- For dictionaries and ADFSA, each character is represented

Representation

- The straight forward representation for ADFSA's is a simple two-dimensional matrix
 - Columns represent the alphabet
 - Rows represent the states
- There is always one accepting state, and after minimization, one final state.

Representation



	0	1
q0	q2	q0
*q1	q1	q1
q2	q2	q1

- State machines are sparse
 - $1 - (|\delta| / |\Sigma| |Q|)$
 - An ADFSA for a dictionary of hiragana verb forms is 97% sparse before minimization:
 - 31 (hiragana) characters
 - 735 states
 - 733 transitions
 - minimization does not help all that much: the same dictionary after minimization is still 93% sparse.
 - 113 states
 - 231 transitions

Solutions

- An ADFSA containing one million lexemes with $|\Sigma| > 5800$ has sparsity beyond 98%.
- The brute-force method of building the ADFSA is to generate a trie in matrix form
 - A trie is an ADFSA after all!
- The trie is then minimized to exploit commonalities in the input
 - Minimizing the automata decreases the size of the matrix.
 - We cannot minimize the alphabet

Solutions

- The remaining savings need to be found by compressing the resulting matrix
- Utilize existing table-compression methods
- We use the method of Kiraz 1999
 - Row-indexed representation of the sparse automata

Row-Indexed Automata

- The automata is represented by three arrays
 - \mathbf{A} contains the entries of the original matrix in row-major order
 - \mathbf{A}_q contains the indices in \mathbf{A} where the transitions for q reside
 - \mathbf{As} contains, for each entry in \mathbf{A} , the symbol associated with the transition at $\mathbf{A}[n]$.
- Space complexity is $2|\delta| + |Q|$
 - Note that $|\Sigma|$ does not enter into the equation!

Row-Indexed Automata

```
static const BT_State kStartState = 65;
static const BT_State kFinalState = 112;

static const BT_State gTransitions[] = {
    112, 0, 5, 1, 5, 5, 92, 1,
    5, 5, 92, 106, 1, 5, 1, 1,
    // ...
};

static const BT_UInt8 gOffsets[] = {
    0, 1, 2, 5, 9, 14, 15, 17,
    19, 22, 24, 26, 27, 28, 29, 31,
    32, 34, 37, 41, 47, 48, 51, 55,
    // ...
};

static const BT_Char16 gCharacters[] = {
    12427, 12356, 12426, 65532, 12425, 12426, 12429, 65532,
    12425, 12426, 12429, 12377, 65532, 12425, 65532, 65532,
    12384, 12414, 65532, 12428, 12414, 65532, 12356, 12427,
    12427, 12428, 12400, 12435, 12383, 65532, 12425, 65532,
    // ...
};
```

Row-Indexed Automata

- Contains 152 fully inflected Japanese verbs in hiragana with their associated dictionary form.
- Recall:
 - Before minimization: 735 states, 733 transitions
 - After minimization: 113 states, 231 transitions
- Total size of the automata is 806 bytes.

Row-Indexed Automata

```
BT_State x_delta(BT_State state, BT_Char16 c)
{
    BT_State k1 = gOffsets[state];
    BT_State k2 = gOffsets[state + 1];

    while (k1 < k2) {
        if (gCharacters[k1] == c) {
            return gTransitions[k1];
        }
        ++k1;
    }
    return FAILURE_STATE;
}
```

Row-Indexed Automata

- Naive state lookup is linear on the average number of transitions on each state
- Word recognition is linear on the length of the input
- Faster state lookup can be implemented
- The row-indexed automata can be small enough to fit in the data cache, and the state lookup is small enough to fit in the instruction cache.

Row-Indexed Automata

- The obvious issue is that a slice in **As** can be quite large and searching linearly can be unoptimal
 - Binary search can be used to get better general performance
 - Hashing
 - Linear with implicit ordering

Issues

- Building a large automata requires a lot of space, perhaps more than is available in the computer.
- Minimization after building is excruciatingly slow.
- Incremental construction is necessary for large lexica
 - Several algorithms are available, some requiring sorted data, some not.
 - Watson and Daciuk looks promising for this task.

Conclusion

- ADFSA are quite usable for creating accepting automata, even for large alphabets.
- Such automata are very sparse, and require compact representation
- Table compression is important, but so is performance.

