1 fst

1.1 FST representation

1.1.1 state is represented as a list of transitions

a state is represented as a list of transitions. The last transition of a state is marked as S. Transition for the last character of each accepted word is marked as F (means Final).

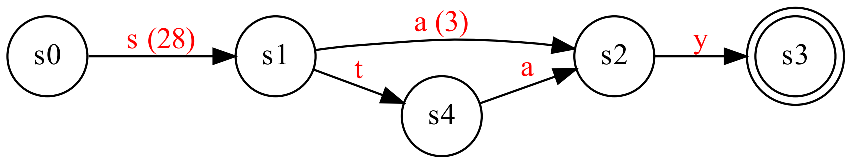

Figure 1.1: DFA of say (output 31) and stay (output 28)

	L	F	S	→
1	s		●	2
2	t			4
3	a		●	5
4	a		●	5
5	y	●	●	0

Figure 1.2: States seen as lists of transitions. L is a label, F marks final transitions. S is the marker for the last transition in the state. → is a pointer to the target state. The automaton recognizes words say and stay.

See the above example. There are
state s0, consists of transition 1 with label 's', to the state start with transition 2.
state s1, consists of transition 2 and 3. Transition 3 is marked 'S' as the last transition of current state.
state s4, consists of transition 4. Marked 'S' since there is only one transition in this state.
state s2, consists of transition 5. Marked 'S' since there is only one transition in this state. Also marked as 'F', since both words "say" and "stay" have their last transition here.
and for last state s3, there is no outgoing transitions, not show figure **??**.

1.2 Size Reduction Techniques

This section's content is from "Smaller Representation of Finite State Automata" by Jan Daciuk.

Next Pointers

Tomasz Kowaltowski et al. ([6]) note that most states have only one incoming and one outgoing transition, forming chains of states. It is natural to place such states one after another in the data structure. We call a state placed directly after the current one the next state. It has been observed in [6] that if we add a flag that is on when the target state is the next one, and off otherwise, then we do not need the pointer for transitions pointing to the next states. In case of the target being the next state, we still need a place for the flags and markers, but they take much less space (not more than one byte) than a full pointer. In case of the target state not being the next state, we use the full pointer. We need one additional bit in the pointer for the flag.

	L	F	S	N	→
1	s		●	●	2
2	t				3
3	a		●	●	4
4	y	●	●		0

Figure 1.3: The next flag (marked as 'N') with sharing of transitions.

Two transitions (number 3 and 4 from figure **??**) occupy the same space Transition Sharing is another compression technique we don't implement. See the paper "Experiments with Automata Compression by Jan Daciuk" for more about it.

Using next pointer, the bit layout of a single transition is as follows.

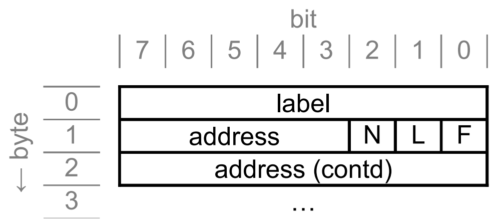

Figure 1.4: Binary layout of data fields in a single transition, version 1. N, F and L are bit flags, address field' s length is as large, as the largest state offset in the automaton (but constant for every transition).

V-coding of target addresses.

The above bit layout version uses fixed-length address encoding integrated with the flags byte. This has an effect of abrupt increases of automaton size once 1, 2, 3 or more bytes are needed to encode the largest state' s offset.

We used a simple form of variable length encoding for non-negative integers (v-coding), where the most significant bit of each byte is an indicator whether this is the last byte of the encoded integer and the remaining bits carry the integer' s data in 7-bit chunks, lowest bits first. For example, 0 is encoded as (binary representation) 0000 0000, 127 as 0111 1111, 128 using two bytes: 1000 0000 and 0000 0001, and so on.

Transitions with index-coded label.

In reality, for automata created on non-degenerate input, and in particular on text, the distribution of label values is often heavily skewed. There are many transitions with a small subset of the label range and a few transitions outside this range.

The observation that labels have uneven distribution leads to an optimization that has a profound effect on automaton size: we can integrate the 31 most frequent labels ($2^5 - 1$) into the flags byte as an index to a static lookup table. Zeros on all these bits would indicate the label is not indexed and is stored separately. Note that we tried to avoid any complex form of encoding (like Huffman trees); a fixed-length table with 31 most frequent labels is a balanced tradeoff between auxiliary lookup structures and label decoding overhead at runtime.

Combining v-coding of the target address and table lookup for the most frequent labels yields two alternative transition formats, as shown in Figure **??**. With such encoding most transitions take 1 + length(address) bytes. In an extreme case when the N bit is also set (target follows the current state' s last transition), the entire transition is encoded in a single byte.

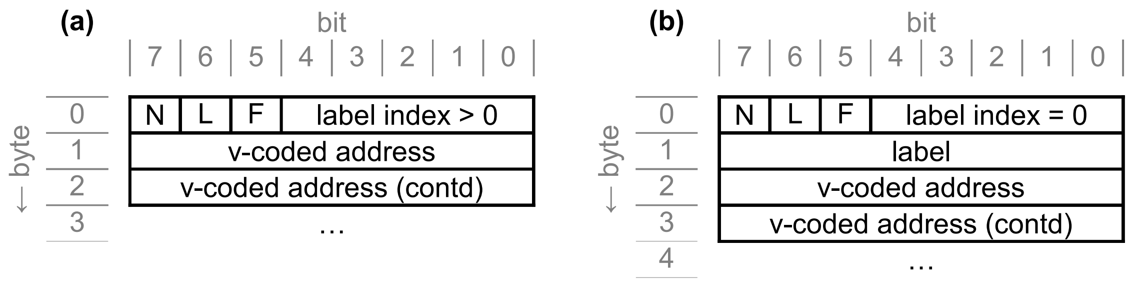

Figure 1.5: Binary layout of data fields in a single transition with v-coding of target address and indexed labels. Two variants of each transition are possible: (a) with the index to the label, (b) with the label directly embedded in the transition structure.

an example

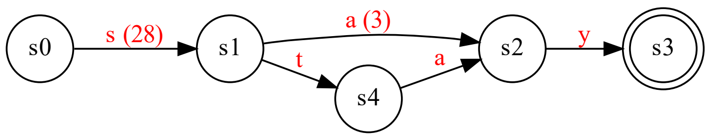

Figure 1.6: DFA of say (output 31) and stay (output 28)

Using the above techniques, dfa of say and stay is represented as follows.

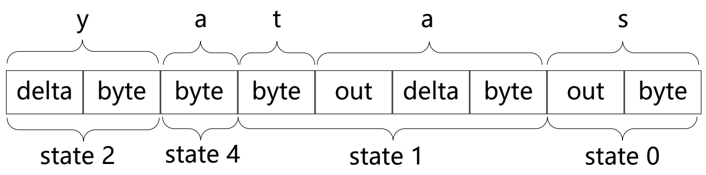

Figure 1.7: dfa byte layout. Notice that data is written in reversed order. last state s3 has no outging transitions, not shown. delta is the address delta between current state and target state. Use delta we can calculate target state address.

The dfa dump info is as follows, also in reversed order:

```
stateToState Address  Arc#idx N F L NextAddr Output StOuts Size(jt)
-----------------------------------------------------------------
    2->3     1        'y' #3    F L x                      2
    4->2     2        'a' #1  ↑   L                        1
    1->4     3        't' #4  ↑   L                        1
    1->2     6        'a' #1        1        3             3
    0->1     8        's' #2  ↑   L          28            2
```

Each row represents a single transition. stateToState shows from which stateId the transition go to which target stateId. Address is the end address of current transition. Arc is the transition label character, #idx is the index of the label.
NFL means Next Pointer (if true, next pointer is enabled), to Final state, is Last transition of current state. NexyAddr means the address of the target state.
Output and StOuts shows the output and stateOutput in current transition. Size means the current transition size in bytes.

1.3 use jump table for fast lookup

During search, we start from state 0, follow the transitions. For example, search for "sd" would go from s0, compare the first char of "sd", which is ’s’ with the the only transition’s label, find a match.

then go to state s1, compare the second char of "sd", which is 'd', with every transition's label going out from s1.

What if there are many transitions from a state? We still need to look at every trantition and check whether we got a match. To enable fast lookup, we can use a jump table.

We want to use binary search, which has O(log_2N) time complexity.To use binary search, first we need to sort the transitions by label alphabetically. Then store the addresses of all the transitons in an array. In practice, we can store the delta between each transition and the transition stored at last. Because delta can be small engough to store in 1 or 2 bytes.

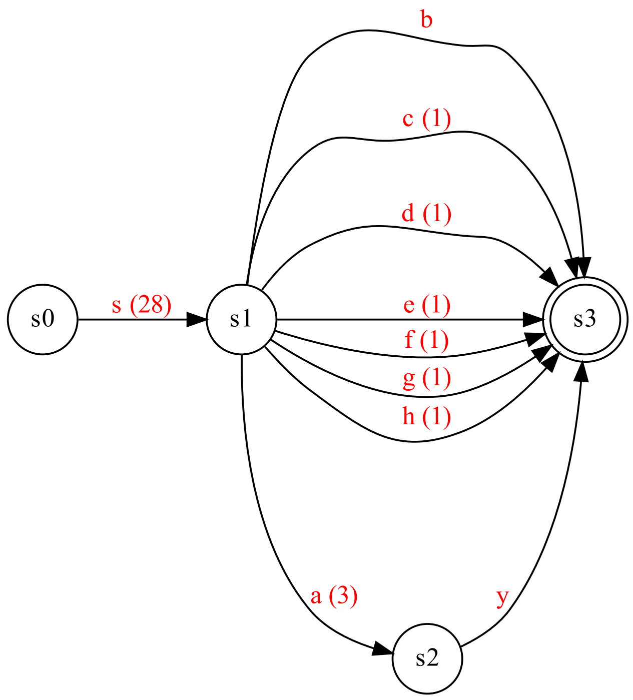

Figure 1.8: dfa With JumpTable Example

It's represented as

```
stateToState Address  Arc#idx N F L NextAddr Output StOuts Size(jt)
------------------------------------------------------------
    2->3     1        'y' #10   F L x                       2
    1->3     4        'h' #9    F L x        1              3
    1->3     7        'g' #8    F   x        1              3
    1->3     10       'f' #7    F   x        1              3
    1->3     13       'e' #6    F   x        1              3
    1->3     16       'd' #5    F   x        1              3
    1->3     19       'c' #4    F   x        1              3
    1->3     21       'b' #3    F   x                       2
    1->2     35       'a' #2        1        3              14 (11)
    0->1     37       's' #1  ↑   L          28             2

stateId=1 jumpTable, elements format is index:  delta(address)
0:  0(24)  3(21)  5(19)  8(16) 11(13) 14(10) 17( 7) 20( 4)
len=8 tag=11111101 header=111
```

In this dfa, state s1 has 8 transitions. First transition is 'a', last trantition is 'h', stored reversed. the end address of transition 'a' is actually 24, with size of 3 bytes(14 - 11 = 3). Jump table size is 11.

The shown address of transition 'a' is 35, is the address of the jump table: 24+11 = 35.

The jump table stores the delta between each transition's address and the address of transition stored at last, which is 'a'.

```
at index 0, is the address_of_a - address_of_a: 24 - 24 = 0.
at index 1, is the address_of_a - address_of_b: 24 - 21 = 3.
...
at index 7, is the address_of_a - address_of_h: 24 - 4 = 20.
```

the Fst on disk is

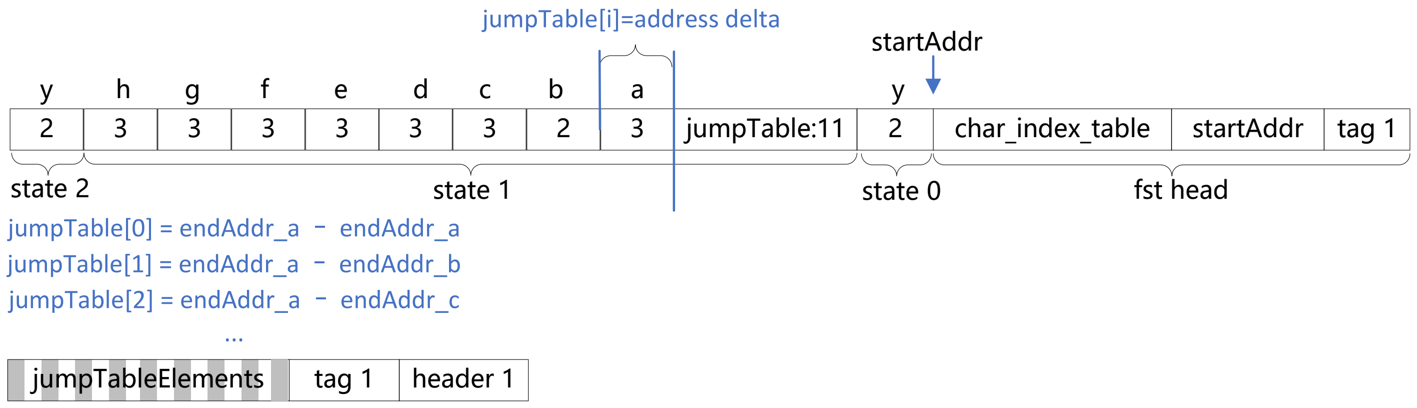
jumpTable[i]=address delta
startAddr
y
h
g
f
e
d
c
b
a
y
2
3
3
3
3
3
3
2
3
jumpTable:11
2
char_index_table
startAddr
tag 1
state 2
state 1
state 0
fst head
jumpTable[0] = endAddr_a - endAddr_a
jumpTable[1] = endAddr_a - endAddr_b
jumpTable[2] = endAddr_a - endAddr_c
...
jumpTableElements
tag 1
header 1

Figure 1.9: The fst on disk. left handside is low address, right handside is high address.

`char_index_table` is a table given a labelIndex we can find the label character.

1.4 implementation details

transition record format

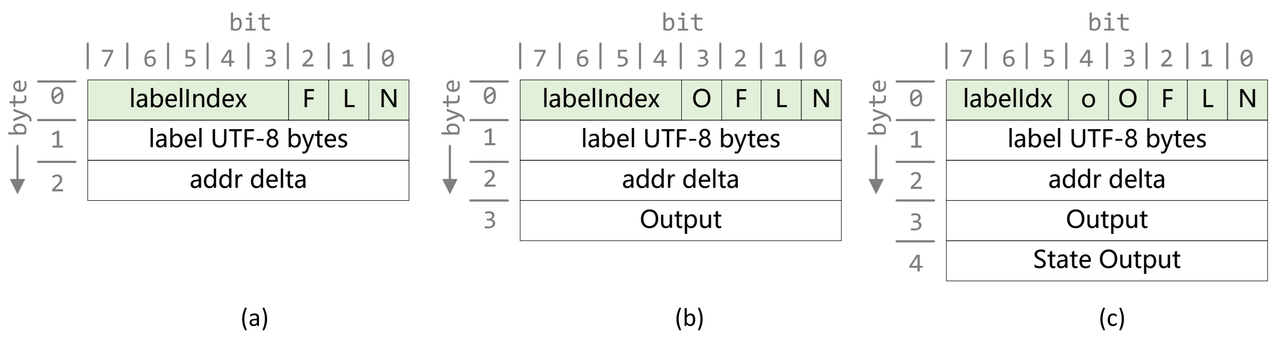

Figure 1.10: transition record format. first byte is required, call it header byte. O means output, o means state output in header byte. There are 3 kinds of record format. (a) for fst with no output. (b) for fst with output but without state output. (c) for fst with output and state output.

Only the header byte is required. if labelIndex is 0 then label character is encoded into UTF-8 bytes (may have 1-4 bytes) and written from next byte on. If labelIndex> 0, label UTF-8 bytes won't be written, the label character will be looked up in char index table.
if N flag in header byte is 1, then addr delta byte will not exist, else addr delta will be written. addr delta is the difference between address of current transition record and address of the next state. iT's used to calculate address of the next state.

avoid code collision

we use header byte `0b0000_0111` to denote there is jump table for a transition record. Note this header byte has label index 0, which means it has no label index in label lookup table. Then the label must be written in the next byte. In practice bytes of a transition record is written reversely. So the label byte is written in the byte before header byte.

To distinguish a jump table header with a normal transition record header, we write `0b1111_1101` or `0b1111_1110` in the previous byte for a jump table. Because a normal transition record header will need to write UTF-8 encoded bytes of the label in the previous byte and (`0b1111_1101` or `0b1111_1110`) is not a valid UTF-8 byte.

`0b1111_1101` is for jump table with element of 1 byte in size.
`0b1111_1110` is for jump table with element of 2 byte in size.