● ACISM: Aho-Corasick Interleaved State-transition Matrix

Mischa Sandberg <mischasan@gmail.com> 2010
https://tinyurl.com/mta67h3r

For large sets of strings (patterns), Aho-Corasick multi-string search implementations typically must make a time/space trade-off. ACISM is an Aho-Corasick implementation that proves that you can have your cake and eat it, minimizing both space and scanning time, with an **O(n)** compile-time. The representation is simple to persist and share between processes. In a 32-bit implementation, ACISM can handle about 10MB of pattern text. It averages under 4 bytes per pattern character and under 20 machine instructions per input byte. For a comparison with other implementations of Aho-Corasick state machines, see NOR04. For the source code of the implementation, see http://github.com/mischasan/aho-corasick

Aho-Corasick Algorithm

Aho-Corasick is a multi-string search algorithm with excellent worst-case behaviour: it performs a bounded (small) amount of work for each input-text character. Bounded worst-case behaviour is particularly important in the hostile environment of network intrusion detection systems, where a denial-of-service attacker can tune exploits against published data and algorithms.

Aho-Corasick amounts to a DFA state machine, implemented by adding *back links* to nodes of a prefix tree. Each node of a prefix tree corresponds to a prefix of some pattern string(s). A back link connects node X to the node Y, when Y is the longest suffix of X that is a prefix of some other string in the set. The search follows a back link when there is no forward link; when there is neither forward nor back link, the search returns to the root.

Fig. 1 Example Tree

This is the Aho-Corasick tree for `['op', 'open', 'retorts', 'tort', 'stop']`:

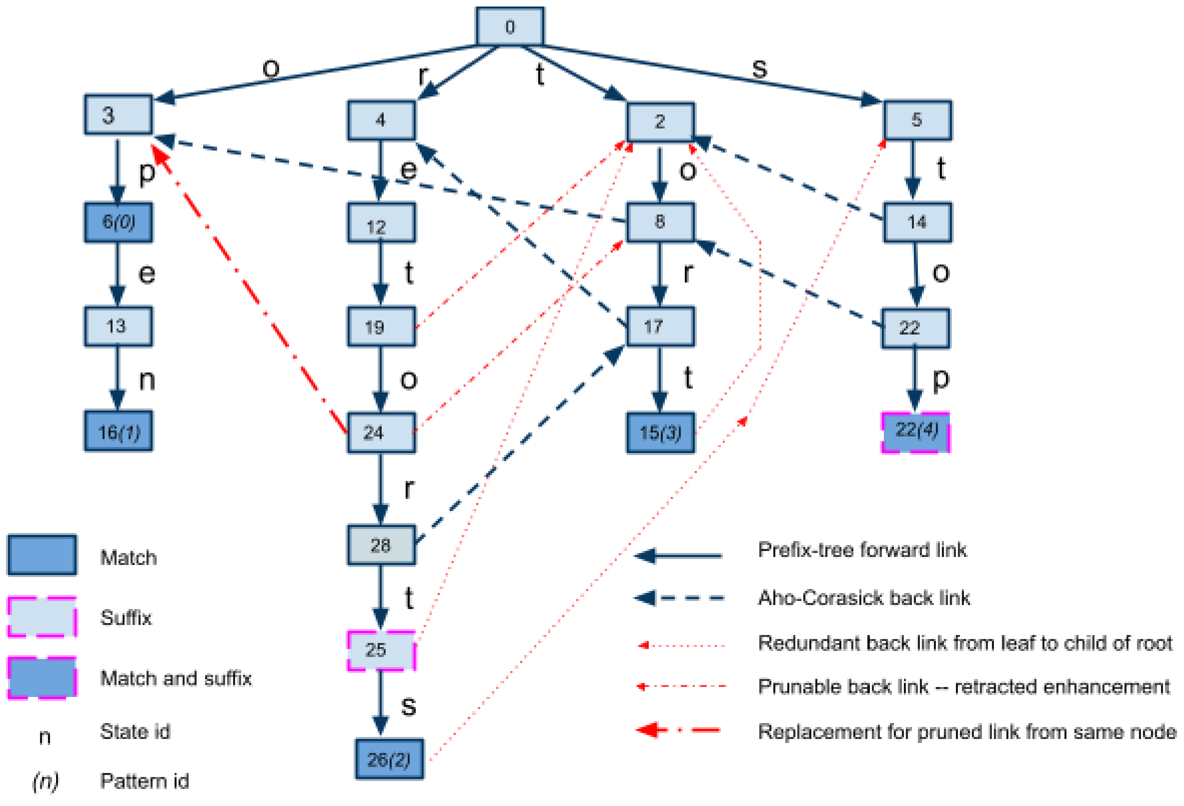

Fig.1 Aho-Corasick Prefix Tree with ACISM Optimizations

(The apparently arbitrary *state id* numbers are a result of ACISM's interleaved-array optimization)
State **[22]** corresponds to matching `sto`. The longest suffix of `sto`, that is also a prefix in this tree, is `to`, as in `tort`.
So a back link connects **[22]→[8]**.

Suppose the text to scan for matches is `store`. The scan processes `sto`, moving forward via **[0]→[5]→[14]→[22]**. `r` does not match any forward link from **[22]**, so the scan follows the back link to **[8]**, matches `r`, and moves to **[17]**. The next text character is `e`, which does not match any forward link from **[17]**, so the scan follows the back link to **[4]**, matches `e` and moves to **[12]**. At this point there is no more input and no match.

Algorithm

The fastest state machine possible is one that uses a state-transition matrix, indexed by [*state,code*], to determine the next state. ACISM represents such a matrix as an *interleaved array*. The information normally associated with a state is either stored in the transition to that state, or in a hash table that maps non-leaf matches (e.g. **[6]**) to pattern-vector indices (*pattern numbers*).

A state may match multiple strings that are suffixes of one another. For example, both `stop` and `op` match at **[22]**. In typical Aho-Corasick implementations, there is an explicit record of the match-set; e.g. a link **[22]**→**[6]**. ACISM does not store such links, because the list of multiple matches will always be a subset of the back link chain. ACISM sets a *match* bit in each transition to a match state; it also sets a *suffix* bit, in each transition with a match higher up the back link chain. For example, the string `retort`, leading to **[25]**, is not a match; but a suffix of that string (`tort`) is a match. For such a node the *suffix* bit is set, so scan follows the back link chain to find **[15]**, which is a match. Since **[15]**'s *suffix* bit is not set, the backward search for matches stops there.

Interleaved Arrays

ACISM interleaves the sparse rows of the state-machine matrix in one flat vector. Each non-leaf state row is assigned a unique base offset in the vector. The back link, if any, and child links, are stored at offsets from that base: the back link is stored at offset **0** and the transitions at offsets of the character *codes* (1..**N**). Rows may overlap, as long as they do not collide on any transition. Elements contain their own offset from their state base, to distinguish which of the overlapping state rows they belong to. Treating the back link like another child in the state-machine row means that no space is wasted if there is no back link. No space is wasted in mapping states to strings: for leaf nodes, the *next* field contains the pattern number; for non-leaf nodes, a hash table maps the state to the string number.

Interleaved arrays amount to S.F. Ziegler's "row-displacement" method; see TAR78. For building such an array, Ziegler figured that the rows should be inserted in descending order of size, analagous to filling a bucket with sand, gravel and rocks (rocks first...). That produces near-ideal packing, but **$O(n^2)$** compilation time. Insertion that ignores node width, plus the **$B_{2,N}$** hint array described below, produces packing with less than 1% waste, with **$O(n)$** compilation time. It adds nodes to the vector top-down, breadth-first; so the top nodes of the tree are packed into the beginning of the vector --- best cache performance.

The interleaved array for the example (**Fig.1**) is shown in **Fig.2**. For clarity, the **State** column shows which state "owns" a cell; for example, $\mathbf{X}_7$ contains the transition from **[3]** for `p` to **[6]**, which is also a match (`op`). The ***code*** values are also shown as their original characters. [t, o, r, p, s, e, n] In this case, the children of **[0]** are in $\mathbf{X}_1$, $\mathbf{X}_2$, $\mathbf{X}_3$ and $\mathbf{X}_5$. The back link and the one child node for **[8]** are in $\mathbf{X}_8$ and $\mathbf{X}_{11}$, respectively. The largest field is a *back* link when *code*=**0,** a **(patt)** index for a leaf node, or a **next** link. In this example, $\mathbf{X}_{19}$ is unused; the interleaving algorithm could not fit anything else in it, so it is left empty/invalid (0). $\mathbf{X}_0$ is always unused. Leaf nodes for $\mathbf{P}_1$, $\mathbf{P}_2$, $\mathbf{P}_3$, $\mathbf{P}_4$ encode the pattern ids (0..3) in the ***next*** field by adding **size**(**X**) -- i.e. 31 -- to them. The hash table **H** contains the single pair (state=**7**, pattno=***0***) for the non-leaf node which identifies (**[3]**,`p`) as matching $\mathbf{P}_0$ i.e. `op`**.**

Fig.2: Interleaved state transition array (X)

Index	State	Code	Match	Suffix	**Next**/*Back*/(**Patt**)
1	0	t			**2**
2	0	o			**3**
3	0	r			**4**
4	2	o			**8**
5	0	s			**5**
6	5	t			**14**
7	3	p	Y		**6**
8	8				*3*
9					
10	4	e			**12**
11	8	r			**17**
12	6	e			**13**
13	12	t			**19**
14	14				*2*
15					
16	14	o			**22**
17	17				*4*
18	17	t	Y		**(3)**+size(**X**)=34
19	19				*2*
20	13	n	Y		**(1)**+size(**X**)= 32
21	19	o			**24**
22	22				*8*
23					
24	24				*8*
25					
26	22	p	Y		**(4)**+size(**X**)=35
27	24	r			**28**
28	28				*17*
29	28	t		Y	**25**
30	25	s	Y		**(2)**+size(**X**)=33

Representation

Let **P** be the input set of pattern strings.The state machine comprises:

- **C**: a vector that maps input byte values (0..255) to a smaller range of *code* values. **C** maps the byte values found in **P** to the code values (1..**N**). **C** maps all other input bytes to **0**. Using codes instead of bytes saves bits in a transition, and makes a common case fast (input byte occurs nowhere in **P**).

- **X**: a vector of interleaved sparse arrays of <u>transitions</u> with the following bit-packed fields:
    - *code*: the code (1..**N**) to which the transition corresponds; or **0** for a back link.
      t = **X**[**state** + **code**] is a valid transition iff **code = t**.*code*.
    - *match*: a bit flag indicating the end of a matched string.
    - *suffix*: a bit flag indicating some suffixes of the current string are matches.
    - *next:* a back link, if **t.***code* is **0**; the next state (number), if **t.***next* < **size**(**X**);
      otherwise the matching pattern number for a leaf node, encoded as pattern_number + **size**(**X**)

- **H**: a hash table that maps (state+code) to a pattern number for non-leaf matches in **X**.
- **Z**: a byte-vector of tails, each identified by offset: { match id, nbytes, tail bytes}

For **P** of about 10MB or less, a transition fits in 32 bits.

H maps the *location* of a transition to the value (**P** index) of a **non-leaf match** in **X**. That location uniquely identifies the pattern; and interleaved-array construction ensures that locations are essentially random (no further “hashing” required). Hash table probes are guaranteed to find a value; so open addressing is reasonable with a table 1.2 x size(**P**)***.*** **I**f the code value is stored in the lowest **K** bits of a transition value, then the low **K** bits of (*transition* XOR *code)* will be 0 for a valid transition or back link.

Execution

Let **T** be a text string to be scanned for matches. The scan algorithm starts at the first byte of **T**, and traverses the prefix tree from the root, following forward links corresponding to successive characters in **T**. When it reaches a node marked as *match* or *suffix*, it reports the match(es). When it reaches a node with no branch corresponding to the next character in **T**, it uses the *back link* to jump to a prior node at a shallower depth, in effect advancing the start-of-string index. When there is no back link, scan restarts from the tree root, at the next byte in **T**. Fig.3 is the pseudo-code for this scan.

Fig 3. Scan routine	**Comments**

state ← ***ROOT***
for i ← **0 to** size(**T**)-1
  code ← **C**[**T**[i]]
  if code = ***BACK***
	"0" is the root-node state.

state ← ***ROOT***
    continue
	T[i] is a character not found in any pattern string.

while code **≠** (t ← **X**[state+code]).*code*
   and state **≠** ***ROOT***
    back ← **X**[state+***BACK***]
	On average, this loop executes once.
	BACK is offset **0**

state ← back.*code* = ***BACK*** **then** back.*next* **else** ***ROOT***
  if code **≠** t.*code*	

continue	No more backlinks, try transition from root.

if not (t.*match* **or** t.*suffix)*
    state ← t.*next*
	Successful forward transition, but no complete match yet.

else s ← state
    state ← IsLeaf(t) **then** ***ROOT*** **else** t.*next*
    repeat
	Found a match and/or suffix-match.

if t.*code* **=** code
        if t.*match*
	(t) is always valid on the first iteration.

p ← IsLeaf(t) **then** PattNo(t)
           else SEARCH(**H**, state+code)
	For a leaf node, t.*next* encodes a **P** index.

PROCESS(p, i)	Perform action for each match.

if state **=** ***ROOT*** **and not** IsLeaf(t)
          state ← t.*next*
        if state ≠ ***ROOT*** **and not** t.*suffix*
          break
      if s **=** ***ROOT***
        break
      back ← **X**[s+***BACK***]
      s ← (back.*code* **=** ***BACK***) **then** back.*next* **else** ***ROOT***
      t ← **X**[s + code]
	Find a relevant back link, in the course of following the chain of possible suffix matches.

IsLeaf(t): t.*next* ≥ size(**X**)
PattNo(t): t.*next* - size(**X**)

Compilation

Construct (**C, X, H, Z**) in these steps:

1. Compute **C**. There is a small *compilation* performance advantage to assigning codes in descending order of frequency of occurrence in **P**: nodes with more than one child are more likely to use and advance the same entry in **B**; see "Allocate **X** offsets" below.

2. Build **T**: a prefix tree of **P**. **T** will have no more nodes than there are characters in **P**, and usually many less. Each node will eventually contain:
    - *first_child, sibling*: tree structure implemented in linked lists.
    - *code*: the code leading to this state node.
    - *pattern_id*: the **P** index of the matching string, or a null value for non-match nodes.
    - *backlink*: Aho-Corasick back link.
    - *suffix*: true if the back link chain from this node contains a match.
    - *refcount*: count of back links pointing at this node.
    - *state*: offset in **X**.

3. Add Aho-Corasick DFA information to **T** (*backlink, suffix, refcount*) for each node. This uses a breadth-first (level-by-level) tree traversal. Any one level of the tree can have no more than **count(P)** elements. *backlink* is a pointer from a branch node to some other (shallower) node. Most *backlinks* point to root(**T**). *suffix* is true if a match can be found by following the chain of back link pointers. It is used in the scan to identify multiple matches by different strings at the same endpoint.

4. ~~Prune backlinks in **T**. If a node **D** is not the target of any back links, and its *backlink* points at a target node **E** whose children are a subset of **D**'s children, and **E** isn't the parent of a *match* or *suffix* node, then **D**'s *backlink* can be changed to **E**'s *backlink*, and the process repeated. When changing a *backlink*, **E**'s *refcount* value is decremented and the new target's *refcount* value is incremented. If **E**'s *refcount* value is decremented to 0, then it is an immediate candidate for pruning, itself.~~

5. Extract tails. *TBD*. Byte block **Z** contains sequences: { match ID, nbytes, tail-bytes }.
   The transition contains the byte-block offset, including sizes . (nbytes) may be a variadic int. Final stage catenates (X,H,Z)

6. Allocate **X** offsets (states), setting the *state* field of **T** nodes. See **Fig.3**. The root node is always allocated at offset 0. This step uses matrix $\mathbf{B}_{\mathbf{c,b}}$ to track the first position where a search could find an available place for a non-leaf node whose first child code is **c**; earlier positions having been proven invalid by previous searches. **b** is 1 for a node with a back link, and 0 for a node without. **T** is traversed breadth-first, as in **(3)**, to improve memory cache behaviour, by allocating nodes near the root of the tree densely together.

7. Populate **X** by traversing **T**, using the *state* offsets.

8. Populate **H** by traversing **T**, adding (*match id*, *pattern id*) to **H** for non-leaf matches. In this implementation, the hash table is filled in two passes, the first pass only inserting non-colliding entries.

Fig.4 Interleaved Allocation

```
USED = 1, BASE = 2
root.state ← 0

for i ← 1 to N
        B[i,0] ← B[i,1] ← 0
for n in nodes
        first ← n.child[0].code
        found ← false
        if n.back = root
                need ← BASE
                base ← B[first,0]
        else    need ← BASE + USED
                base ← max( B[first,0], B[first,1] )

        repeat
                while U[base] & need
                        base ← base + 1
                fits ← 1
                for c in n.child
                        fits ← U[base + c.code] & USED
                        if fits = 0
                                break
                if not (found or c = n.child[0])
                        found ← true
                B[first, need & USED] ← base + fits
        until fits
        n.state ← base
        U[base] += need
        for c in n.child
                U[base + c.code] += USED
```

Comments

Bit constants. USED: holds a transition;
BASE means allocated as a state base-offset.

Initialize starting points for base searches to 1.

Traverse all non-leaf nodes, breadth-first.

Need to search for an unused *base*.

Need to search for an unused *base* that is also free to hold a back link.

Advance to an unallocated base position, that is also an unoccupied element if *n* has a back link.

Record state base for next step.

Mark *base* allocated and possibly used.

Mark child positions as used.

Performance

The basic per-byte search loop, compiled with gcc 4.2.1 -O3 for ia86-32 , averages 20 machine instructions, with 2 look-ups in **X** and 3 jumps. The worst-case for the loop through back links is a pattern set such as [b, ab, aab, aaab, ...] being matched against aaaa...aaac. The effort is bounded by the length of the longest pattern string; in practice, the longest back link chain is far shorter. On the test machine, running fgrep -xf /usr/share/dict/words takes 1.8 secs, versus 1.2 secs for ACISM to compile and execute, most of the time being in compilation.

Fig 5. Compilation performance

Sample is /usr/share/dict/words

Statistic	**Count**
Size(**P**)	479,829
Total chars in **P**	4,473,870
Unique chars in **P**	70
Non-leaf nodes in **T**	1,060,025
Size(**X**)	2,519,340
Unused elements in **X**	24,451
Size(**H**)	131,057
Pruned back links	79,125
Innermost loop iterations in **interleave**	88,602,305

Enhancements

Tail strings

A common case is long pattern strings that are distinguished uniquely in their leading bytes --- e.g. GUID's. The tail of such strings take a whole transition (4 bytes) per char. If the tail nodes are not the source or target of any backlink, the tail chars can be stored as a byte block outside the state machine array, rather than as a chain of state transitions. This makes ACISM take less space, run faster *and* compile faster. In Fig.1 , **[6]** "`en`" is an example of a tail.

From such a state, the search uses a simpler loop, scanning for the first input byte that does not match the tail (byte block). The no-backlinks requirement means that, at that point, or after a match, state can/must be reset to 0 (root) and the input pointer skips the matched bytes. This works for block-streamed search -- qv **acism_more**(). If input matches bytes of the tail up to the current block's end, state carries the (offset,end) of remaining bytes to the next acism_more call; scan resumes by comparing the start of the next input block, to the remainder of the tail.

The performance gain mostly from match successes: a tail loop iteration being faster than a state transition iteration.
However, this in turn only counts if there are many matches --- and a match executes the callback!

This enhancement applies where none of the string prefixes occur at the end of long strings. The no-backlinks-to-root optimization means that at least two chars of pattern prefix must occur in the tail, to make it ineligible.

Pruning backlinks

ACISM originally pruned "useless" back links. For example, in the above diagram, **[24]**→**[8]**→**[3]** is a standard back link chain. However, if a match fails at **[24]**, then it will necessarily fail at **[8]**, since the only branch(es) from **[8]** is `rt`, which is a subset of the branch(es) from **[24]**. ACISM originally changed this to a link **[24]**→**[3]**, since **[3]** has a transition other than `rt`. ACISM does not prune a link such as **[25]**→**[17]** because it needs it to find a suffix match. It also doesn't prune **[14]**→**[2]**, because the `or` branch from **[2]** is not covered by the branches under **[14]**. In practice, about 5% of backlinks are prunable..

The original pruning implementation only checked target children as a subset of source children, not target branches as a subset of source branches (including MATCH bits). This is broken, and was removed. Removing backlinks increases the number and length of tail strings, making it worth implementing correctly

References

[NOR04] Optimized Pattern Matching for IDS; Marc Norton, 2004
http://goo.gl/xy8R1s

[TAR78] Storing a Sparse Table; Robert Tarjan, 1978
http://goo.gl/rz0VWb