

ACISM: Aho-Corasick with Interleaved Arrays

Mischa Sandberg, 2010

For large sets of strings, 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](#).

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.

For example, **Fig.1** is the Aho-Corasick tree for ['op', 'open', 'retorts', 'tort', 'stop']:

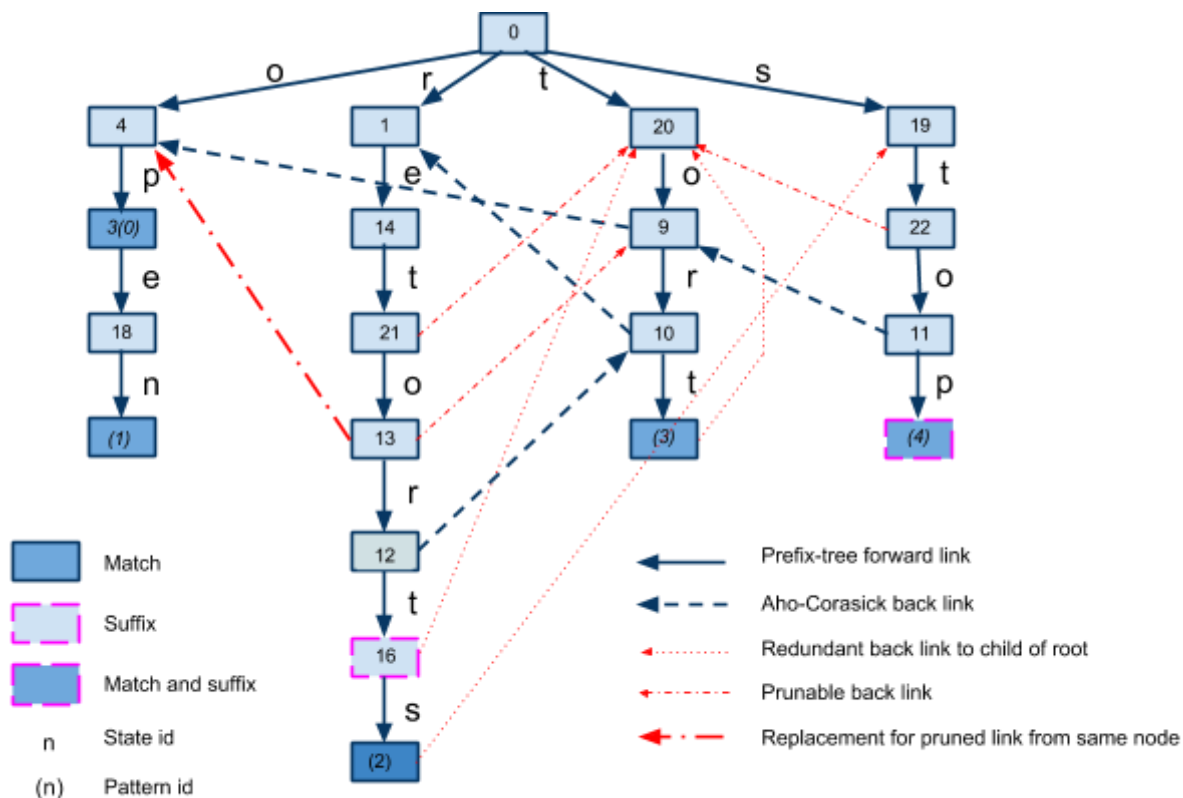


Fig.1 Aho-Corasick Prefix Tree with PSEARCH Optimizations

(The apparently arbitrary *state id* numbers are a result of ACISM's interleaved-array optimization; see below.)

State [11] 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 [11]→[9].

Suppose the target text is *store*. The scan processes *sto*, moving forward via [0]→[19]→[22]→[11]. *r* does not match any forward link from [11], so the scan follows the back link to [9], matches *r*, and moves to [10]. The next text character is *e*, which does not match any forward link from [10], so the scan follows the back link to [1], matches *e* and moves to [14].

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. **[3]**) to pattern-vector indices (*string numbers*).

A state may match multiple strings that are suffixes of one another. For example, both `stop` and `op` match at **[15]**. In typical Aho-Corasick implementations, there would be an explicit record of the match-set; e.g. a link **[15]→[3]**. 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 **[16]**, 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 **[17]**, which is a match. Since **[17]**'s *suffix* bit is not set, the backward search stops there.

ACISM also implements a minor enhancement of Aho-Corasick: pruning useless back links. For example, in the above diagram, **[13]→[9]→[4]** is a standard back link chain. However, if a match fails at **[13]**, then it will necessarily fail at **[9]**, since the only transition from **[9]** is `r`, which was also a transition from **[13]**. ACISM changes this to a link **[13]→[4]**, since **[4]** has a transition other than `r`. ACISM does not prune a link such as **[12]→[10]** if it needs it to find a suffix match. In practice, about 5% of backlinks are prunable.

Interleaved Arrays

ACISM interleaves the sparse rows of the state-machine matrix in one flat vector. Each non-leaf state 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 offset by 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 rows (states) to which they belong. 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 string number; for non-leaf nodes, a hash table maps the state to the string number.

Interleaved arrays amount to S.J. Ziegler's "row-displacement" method; see [TAR78](#). Ziegler decided that the rows should be inserted in descending order of size, following the metaphor of how to fill a bucket with sand, gravel and rocks (rocks first...). That produces near-ideal packing, but $O(n^2)$ execution time. Random order of insertion, with the $B_{2,N}$ hint array described below, produces packing with less than 1% waste, with $O(n^{1.5})$ execution time.

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 the common scan case (input byte occurs nowhere in **P**) as small and cache-friendly as possible.
- **X**: a vector of interleaved sparse arrays of *transitions* with the following bit-packed fields:
 - *code*: the code to which the transition corresponds; or **0** for a back link.
 - *match*: a bit flag indicating the end of a matched string.
 - *suffix*: a bit flag indicating the end of some matched suffix(es) of the current string.
 - *next*: a back link if *code* is **0**; the next state (number), if *next* < **size(X)**; otherwise the X element corresponds to a leaf node, and *next* - **size(X)** is the matching string number. $t = \text{TRANSITION}(\text{state}, \text{code})$ is a valid transition iff $t.\text{code} = \text{code}$.
- **H**: a hash table that maps non-leaf *match* offsets in **C** to *string numbers* (**P** indices).

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 the interleaved-array construction ensures that locations are essentially random. Hash table probes are guaranteed to find a value; so open addressing is reasonable with a table $1.2 \times \text{size}(\mathbf{P})$. If 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.2](#) is the pseudo-code for this scan.

Fig 2. Scan routine

```
state ← 0
for i ← 0 to length(T)-1
  code ← C[T[i]]
  if code = 0
    state ← 0
    continue

  while !VALID(t ← TRANSITION(state,code))
    and VALID(back ← TRANSITION(state,0))

    state ← back.state
  if not VALID(t)
    state ← 0
    continue
  if not t.match and not t.suffix
    state ← t.next
    continue

  s ← state
  state ← t.isleaf ? 0 : t.next
  repeat
    if VALID(t)
      if t.match
        p ← t.isleaf ? t.pattno : SEARCH(H, state+code)
        PROCESS(p, i)

        if state = 0 and not t.isleaf
          state ← t.state
        if state != 0 and not t.suffix
          break
    if s = 0
      break
  back ← TRANSITION(s,0)

  s ← VALID(back) ? back.state: 0
  t ← TRANSITION(s, code)
```

Comments

"0" is the root node state.

T[i] is a character not found in any pattern string.

On average, this loop executes once.

No more links, return to root.

Successful forward transition, but no complete match yet.

Found a match and/or suffix-match.

(t) is always valid on the first iteration. For a leaf node, t.next is a **P** index.

Perform action for each match.

Find a relevant back link, in the course of following the chain of possible suffix matches.

Compilation

Construct (**C**, **X**, **H**) 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:
 - *sibling, first_child*: 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*: reference 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 **size(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 is not the target of any back links, and its *backlink* points at a target node whose children are a subset of the first node's children, and the target isn't the parent of a *match* or *suffix* node, then the *backlink* can be changed to the target's *backlink*, and the process repeated. When changing a *backlink*, the old target's *refcount* value is decremented and the new target's *refcount* value is incremented. If the old target's *refcount* value is decremented to 0, then the old target is an immediate candidate for pruning.
5. 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 **B**_{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.
6. Populate **X** by traversing **T**, using the *state* offsets.
7. 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.3 Interleaved Allocation

USED = 1, BASE = 2

root.state \leftarrow 0

for c in root.child

$U[\text{root.state} + c.\text{code}] \leftarrow \text{USED}$

for i \leftarrow 1 to N

$B[i,0] \leftarrow B[i,1] \leftarrow 1$

for n in nodes

 first \leftarrow n.child[0].code

 found \leftarrow false

 if n.back = root

 need \leftarrow BASE

 base $\leftarrow B[\text{first},0]$

 else

 need \leftarrow BASE + USED

 base \leftarrow max($B[\text{first},0]$, $B[\text{first},1]$)

 repeat

 while $U[\text{base}] \& \text{need}$

 base \leftarrow base + 1

 fits \leftarrow 1

 for c in n.child

 fits $\leftarrow U[\text{base} + c.\text{code}] \& \text{USED}$

 if fits = 0

 break

 if not (found or c = n.child[0])

 found \leftarrow true

$B[\text{first}, \text{need} \& \text{USED}] \leftarrow \text{base} + \text{fits}$

 until fits

 n.state \leftarrow base

$U[\text{base}] += \text{need}$

 for c in n.child

$U[\text{base} + c.\text{code}] += \text{USED}$

Comments

Bit constants. USED means 'filled', BASE means allocated as a state base-offset.

Allocate places for the root's children.

Initialize starting points for base searches to 1.

Traverse all non-leaf nodes, breadth-first.

Find the start of a search for an unused *base*.

Find the start of a 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.

The interleaved array for the example (Fig.1) is shown in [Fig.4](#). For clarity, the state that "owns" an element is shown; for example, **X**₈ contains the transition from **[4]** for **p**. The *code* values are also shown as their original characters. In this case, the children of **[0]** are in **X**₃, **X**₅, **X**₆ and **X**₇. The back link and the one child node for **[9]** are in **X**₉ and **X**₁₄, respectively. Back links have a **0** *code* value. The largest field can contain a **patt** index (leaf hence match), a **next** link (non-leaf state) or a **back** link (for **code=0** only). In this example, **X**₁ and **X**₁₆ are unused; the interleaving algorithm could not fit anything else in them, so they are left empty/invalid (0). **X**₀ is always unused. Leaf nodes for **P**₀, **P**₁, **P**₂, **P**₃ encode the pattern ids (0..3) in the **Next** field by adding **size(X)** -- i.e. 27 -- to them. The hash table **H** contains the single pair (8,0) for the non-leaf node which identifies (**[4]**,**p**) as matching **P**₀ i.e. **op**.

Fig.4: Interleaved state transition array (X)

Index	Base State	Code	Match	Suffix	Patt or Next or Back
1	1	0			
2	1	e			14
3	0	o			4
4	3	e			18
5	0	r			1
6	0	s			19
7	0	t			20
8	4	p	Y		3
9	9	0			4
10	10	0			1
11	11	0			9
12	12	0			10
13	13	0			4
14	9	r			10
15	11	p	Y	Y	3+27=30
16					
17	10	t	Y		2+27=29
18	13	r			12
19	12	t		Y	16
20	18	n	Y		0+27=27
21	14	t			21
22	16	s	Y		1+27=28
23	20	o			9
24	21	o			13

25	22	o			11
26	9	t			22

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...aac`. 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. Performance statistics for compiling `/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	1,060,024
Size(X)	2,514,975
Unused elements in X	20,086
Size(H)	131,086
Pruned back links	79,125
Innermost loop iterations in interleave	160,673,264

References

- [NOR04] Optimized Pattern Matching for IDS; Marc Norton, 2004
<http://docs.idsresearch.org/OptimizingPatternMatchingForIDS.pdf>
[TAR78] Storing a Sparse Table; Robert Tarjan, 1978
<http://historical.ncstrl.org/litesite-data/stan/CS-TR-78-683.pdf>