Aho-Corasick String Matching

Arthur Freitas Ramos
David Barros Hulak
Ruy J. G. B. de Queiroz

June 25, 2026

Abstract

This development formalizes the Aho-Corasick multi-pattern string matching algorithm [1] over lists. The verified executable reference automaton uses the canonical Aho-Corasick state: after reading a text prefix, the state is the longest suffix of that prefix that is also a pattern prefix. The development also proves a state-only transition theorem, a fold-based scan invariant, and a recursive failure-link transition refining the canonical transition, together with equivalent state-carrying and failure-link search procedures. The main theorem shows that the search procedure reports exactly the pattern occurrences ending at each text position. AI assistance was used for proof engineering; all final definitions, statements, and proofs are checked by Isabelle.

Contents

theory *Aho-Corasick*
imports *HOL−Library.Sublist*
begin

1 Aho-Corasick String Matching

The original Aho-Corasick string matching algorithm was introduced by Aho and Corasick [1]. It searches for all occurrences of a finite dictionary of patterns in a text. Its central automaton invariant is that, after reading a

text prefix, the current state is the longest suffix of the processed text that is also a prefix of a dictionary pattern.

This theory formalizes that canonical Aho-Corasick state and an executable reference search procedure over arbitrary lists. For each consumed text prefix, the reference transition recomputes the canonical state by scanning the finite list of pattern prefixes; this is the abstract automaton semantics that the usual failure-link implementation realizes more efficiently. We also derive a state-only transition theorem, a fold-based scan invariant, and a recursive failure-link transition that refines the canonical transition. The main theorem proves that the reports produced by the search are exactly the pattern occurrences ending at each text position.

AI assistance was used for proof engineering. The final definitions, statements, and proofs are checked by Isabelle.

1.1 Pattern-prefix states

definition *ac-states* :: *'a list list* ⇒ *'a list list* **where**
  ac-states pats = *remdups* ([] # *concat* (*map prefixes pats*))

lemma *set-ac-states*:
  set (*ac-states pats*) = *insert* [] {*q*. ∃ *p*∈*set pats*. *prefix q p*}
  ⟨*proof*⟩

lemma *Nil-in-ac-states* [*simp*]: [] ∈ *set* (*ac-states pats*)
  ⟨*proof*⟩

fun *longest-list* :: *'a list list* ⇒ *'a list* **where**
  longest-list [] = []
| *longest-list* (*x* # *xs*) =
    (*let y* = *longest-list xs in if length x* < *length y then y else x*)

lemma *longest-list-member*:
  assumes *xs* ≠ []
  shows *longest-list xs* ∈ *set xs*
  ⟨*proof*⟩

lemma *longest-list-longest*:
  assumes *x* ∈ *set xs*
  shows *length x* ≤ *length* (*longest-list xs*)
  ⟨*proof*⟩

1.2 The canonical Aho-Corasick state

The canonical state for a word w is chosen among the finite list of all pattern prefixes. Since the empty list is always a state and always a suffix, the candidate list is nonempty. The selected state is therefore a suffix of w, a valid pattern-prefix state, and at least as long as any other valid suffix state.

definition *state-candidates* :: *'a list list* ⇒ *'a list* ⇒ *'a list list* **where**
state-candidates pats w = *filter* (λ*q*. *suffix q w*) (*ac-states pats*)

definition *ac-state* :: *'a list list* ⇒ *'a list* ⇒ *'a list* **where**
ac-state pats w = *longest-list* (*state-candidates pats w*)

lemma *state-candidates-nonempty*: *state-candidates pats w* ≠ []
⟨*proof*⟩

lemma *ac-state-in-states*: *ac-state pats w* ∈ *set* (*ac-states pats*)
⟨*proof*⟩

lemma *ac-state-suffix*: *suffix* (*ac-state pats w*) *w*
⟨*proof*⟩

lemma *ac-state-longest*:
assumes *q* ∈ *set* (*ac-states pats*)
assumes *suffix q w*
shows *length q* ≤ *length* (*ac-state pats w*)
⟨*proof*⟩

lemma *pattern-is-state*:
assumes *p* ∈ *set pats*
shows *p* ∈ *set* (*ac-states pats*)
⟨*proof*⟩

lemma *ac-state-snoc-prefix-closed*:
assumes *r* @ [*x*] ∈ *set* (*ac-states pats*)
shows *r* ∈ *set* (*ac-states pats*)
⟨*proof*⟩

lemma *pattern-suffix-ac-state-iff*:
assumes *p* ∈ *set pats*
shows *suffix p* (*ac-state pats w*) ⟷ *suffix p w*
⟨*proof*⟩

lemma *ac-state-Nil* [*simp*]: *ac-state pats* [] = []
⟨*proof*⟩

1.3 State-only transition

The next theorem captures the automaton nature of the construction. Although the canonical state is defined from the whole consumed text prefix, the next canonical state can be computed from only the current state and the next symbol.

definition *ac-step* :: *'a list list* ⇒ *'a list* ⇒ *'a* ⇒ *'a list* **where**
ac-step pats q x = *ac-state pats* (*q* @ [*x*])

lemma *suffix-snoc-mono*:

assumes *suffix q w*
shows *suffix* (*q* @ [*x*]) (*w* @ [*x*])
⟨*proof*⟩

lemma *state-suffix-next-window*:
assumes *q*: *q* = *ac-state pats w*
assumes *r-state*: *r* ∈ *set* (*ac-states pats*)
assumes *r-suffix*: *suffix r* (*w* @ [*x*])
shows *suffix r* (*q* @ [*x*])
⟨*proof*⟩

theorem *ac-state-snoc*:
ac-state pats (*w* @ [*x*]) = *ac-step pats* (*ac-state pats w*) *x*
⟨*proof*⟩

Iterating the state transition over a text fragment therefore tracks the canonical state for the whole consumed prefix.

theorem *ac-state-foldl-from*:
assumes *q* = *ac-state pats w*
shows *foldl* (*ac-step pats*) *q xs* = *ac-state pats* (*w* @ *xs*)
⟨*proof*⟩

theorem *ac-state-foldl*:
foldl (*ac-step pats*) [] *xs* = *ac-state pats xs*
⟨*proof*⟩

theorem *ac-state-foldl-take*:
foldl (*ac-step pats*) [] (*take i text*) = *ac-state pats* (*take i text*)
⟨*proof*⟩

1.4 Failure-link transition refinement

The failure link of a state is the longest proper suffix that is also a pattern-prefix state. The recursive failure-link transition follows these links until it finds a state that can consume the next symbol, or returns to the empty state. The main theorem of this subsection proves that this recognizable Aho-Corasick transition refines the canonical transition above.

definition *proper-suffix-state-candidates* :: *'a list list* ⇒ *'a list* ⇒ *'a list list* **where**
proper-suffix-state-candidates pats q =
filter (λ*r*. *suffix r q* ∧ *r* ≠ *q*) (*ac-states pats*)

definition *ac-fail* :: *'a list list* ⇒ *'a list* ⇒ *'a list* **where**
ac-fail pats q = *longest-list* (*proper-suffix-state-candidates pats q*)

lemma *proper-suffix-state-candidates-nonempty*:
assumes *q* ≠ []
shows *proper-suffix-state-candidates pats q* ≠ []
⟨*proof*⟩

lemma *ac-fail-in-states*:
assumes $q \neq []$
shows *ac-fail pats q* $\in$ *set* (*ac-states pats*)
⟨*proof*⟩

lemma *ac-fail-suffix*:
assumes $q \neq []$
shows *suffix* (*ac-fail pats q*) *q*
⟨*proof*⟩

lemma *ac-fail-proper*:
assumes $q \neq []$
shows *ac-fail pats q* $\neq$ *q*
⟨*proof*⟩

lemma *length-ac-fail-less*:
assumes $q \neq []$
shows *length* (*ac-fail pats q*) $<$ *length q*
⟨*proof*⟩

lemma *ac-fail-longest*:
assumes $r \in$ *set* (*ac-states pats*)
assumes *suffix r q*
assumes $r \neq q$
shows *length r* $\leq$ *length* (*ac-fail pats q*)
⟨*proof*⟩

lemma *ac-state-self-if-state*:
assumes $q \in$ *set* (*ac-states pats*)
shows *ac-state pats q* = *q*
⟨*proof*⟩

lemma *ac-state-singleton-if-no-state*:
assumes $[x] \notin$ *set* (*ac-states pats*)
shows *ac-state pats* $[x] = []$
⟨*proof*⟩

lemma *state-suffix-fail-next*:
assumes *q-ne*: $q \neq []$
assumes *qx-not-state*: q @ $[x] \notin$ *set* (*ac-states pats*)
assumes *r-state*: $r \in$ *set* (*ac-states pats*)
assumes *r-suffix*: *suffix r* (q @ $[x]$)
shows *suffix r* (*ac-fail pats q* @ $[x]$)
⟨*proof*⟩

lemma *ac-step-fail-eq*:
assumes *q-state*: $q \in$ *set* (*ac-states pats*)
assumes *q-ne*: $q \neq []$

assumes *qx-not-state*: q @ $[x] \notin$ *set* (*ac-states pats*)
shows *ac-step pats* (*ac-fail pats q*) x = *ac-step pats q x*
⟨*proof*⟩

function *ac-fail-step* :: *'a list list* ⇒ *'a list* ⇒ *'a* ⇒ *'a list* **where**
ac-fail-step pats q x =
(*if q* @ $[x] \in$ *set* (*ac-states pats*) *then q* @ $[x]$
else if q = [] *then* []
else ac-fail-step pats (*ac-fail pats q*) *x*)
⟨*proof*⟩
termination
⟨*proof*⟩

theorem *ac-fail-step-refines*:
assumes $q \in$ *set* (*ac-states pats*)
shows *ac-fail-step pats q x* = *ac-step pats q x*
⟨*proof*⟩

theorem *ac-fail-state-foldl-from*:
assumes q = *ac-state pats w*
shows *foldl* (*ac-fail-step pats*) *q xs* = *ac-state pats* (w @ *xs*)
⟨*proof*⟩

theorem *ac-fail-state-foldl*:
foldl (*ac-fail-step pats*) [] *xs* = *ac-state pats xs*
⟨*proof*⟩

theorem *ac-fail-state-foldl-take*:
foldl (*ac-fail-step pats*) [] (*take i text*) = *ac-state pats* (*take i text*)
⟨*proof*⟩

1.5 Search and correctness

The output function filters the dictionary to the patterns that are suffixes of the current state. By the canonical-state invariant, this is equivalent to filtering the dictionary to the patterns that occur ending at the current text position.

definition *ac-output* :: *'a list list* ⇒ *'a list* ⇒ *'a list list* **where**
ac-output pats q = *filter* (λ*p*. *suffix p q*) *pats*

theorem *ac-output-state*:
set (*ac-output pats* (*ac-state pats w*)) =
{*p*. $p \in$ *set pats* ∧ *suffix p w*}
⟨*proof*⟩

fun *ac-search-from* ::
'a list list ⇒ *'a list* ⇒ *nat* ⇒ *'a list* ⇒ (*'a list* × *nat*) *list*
where
ac-search-from pats w i [] = []

```
| ac-search-from pats w i (x # xs) =
    (let w' = w @ [x];
         q = ac-state pats w'
     in map (λp. (p, i)) (ac-output pats q) @
        ac-search-from pats w' (Suc i) xs)

definition ac-search :: 'a list list ⇒ 'a list ⇒ ('a list × nat) list where
  ac-search pats text = ac-search-from pats [] 0 text

fun ac-search-states-from ::
  'a list list ⇒ 'a list ⇒ nat ⇒ 'a list ⇒ ('a list × nat) list
where
  ac-search-states-from pats q i [] = []
| ac-search-states-from pats q i (x # xs) =
    (let q' = ac-step pats q x in
       map (λp. (p, i)) (ac-output pats q') @
       ac-search-states-from pats q' (Suc i) xs)

definition ac-search-states :: 'a list list ⇒ 'a list ⇒ ('a list × nat) list where
  ac-search-states pats text = ac-search-states-from pats [] 0 text

fun ac-search-fail-from ::
  'a list list ⇒ 'a list ⇒ nat ⇒ 'a list ⇒ ('a list × nat) list
where
  ac-search-fail-from pats q i [] = []
| ac-search-fail-from pats q i (x # xs) =
    (let q' = ac-fail-step pats q x in
       map (λp. (p, i)) (ac-output pats q') @
       ac-search-fail-from pats q' (Suc i) xs)

definition ac-search-fail :: 'a list list ⇒ 'a list ⇒ ('a list × nat) list where
  ac-search-fail pats text = ac-search-fail-from pats [] 0 text

lemma ac-search-states-from-eq:
  assumes q = ac-state pats w
  shows ac-search-states-from pats q i xs = ac-search-from pats w i xs
  ⟨proof⟩

theorem ac-search-states-eq:
  ac-search-states pats text = ac-search pats text
  ⟨proof⟩

lemma ac-search-fail-from-eq:
  assumes q = ac-state pats w
  shows ac-search-fail-from pats q i xs = ac-search-from pats w i xs
  ⟨proof⟩

theorem ac-search-fail-eq:
  ac-search-fail pats text = ac-search pats text
```

⟨*proof*⟩

1.6 Executable example

The following concrete input exercises overlapping dictionary patterns for both executable search procedures. The *eval* proof method compiles the definitions to ML, evaluates them, and checks the resulting equations.

lemma *ac-search-states-example*:
  ac-search-states [[*0*], [*0*,*1*], [*1*,*0*,*1*], [*1*,*2*], [*1*,*2*,*0*], [*2*], [*2*,*0*,*0*]]
    [*0*,*1*,*2*,*2*,*0*,*1*::*nat*] =
  [([*0*], *0*), ([*0*, *1*], *1*), ([*1*, *2*], *2*), ([*2*], *2*), ([*2*], *3*), ([*0*], *4*), ([*0*, *1*], *5*)]
  ⟨*proof*⟩

lemma *ac-search-fail-example*:
  ac-search-fail [[*0*], [*0*,*1*], [*1*,*0*,*1*], [*1*,*2*], [*1*,*2*,*0*], [*2*], [*2*,*0*,*0*]]
    [*0*,*1*,*2*,*2*,*0*,*1*::*nat*] =
  [([*0*], *0*), ([*0*, *1*], *1*), ([*1*, *2*], *2*), ([*2*], *2*), ([*2*], *3*), ([*0*], *4*), ([*0*, *1*], *5*)]
  ⟨*proof*⟩

definition *occurs-ending-at* :: *'a list* ⇒ *'a list* ⇒ *nat* ⇒ *bool* **where**
  occurs-ending-at pat text i ⟷
    i < *length text* ∧ *suffix pat* (*take* (*Suc i*) *text*)

lemma *in-ac-search-from-iff*:
  (*p*, *k*) ∈ *set* (*ac-search-from pats w i xs*) ⟷
    (∃ *j* < *length xs*.
      k = *i* + *j* ∧
      p ∈ *set pats* ∧
      suffix p (*ac-state pats* (*w* @ *take* (*Suc j*) *xs*)))
⟨*proof*⟩

theorem *ac-search-correct*:
  set (*ac-search pats text*) =
    {(*p*, *i*). *p* ∈ *set pats* ∧ *occurs-ending-at p text i*}
⟨*proof*⟩

theorem *ac-search-states-correct*:
  set (*ac-search-states pats text*) =
    {(*p*, *i*). *p* ∈ *set pats* ∧ *occurs-ending-at p text i*}
  ⟨*proof*⟩

theorem *ac-search-fail-correct*:
  set (*ac-search-fail pats text*) =
    {(*p*, *i*). *p* ∈ *set pats* ∧ *occurs-ending-at p text i*}
  ⟨*proof*⟩

end

References

[1] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975. DOI: 10.1145/360825.360855.