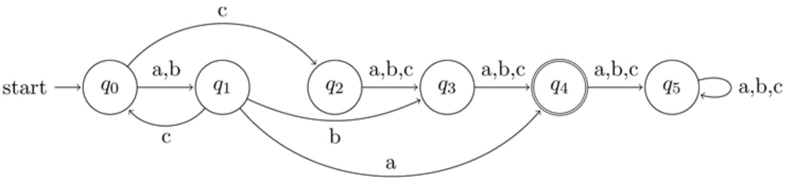
Efficiently Matching Multiple Regular Expressions

Nick Black

BetterCloud (Atlanta, GA)



December 6, 2013

Overview

We introduce techniques to match arbitrarily many POSIX Extended regular expressions, in an online fashion[1], in linear time and polynomial space.

These techniques—arising from automata theory, abstract algebra, and formal language theory—are employed by the BetterCloud Data Loss Prevention (DLP) Engine.

[1] I.e., the input can be provided piece by piece.

The **String** Problem

Given

- an alphabet Σ,
- a pattern p,
- and a text $t_0, t_1, \ldots, t_m$,

find and distinguish all matches.

The **String** Problem—Naïve solution

```
unsigned naive(const char *needle, const char *haystack){
        unsigned matches = 0;
        while(*haystack){
                const char *n;
                for(n = needle ; *n ; ++n){
                        if(haystack[n - needle] != *n){
                                break;
                        }
                }
                matches += !*n;
                ++haystack;
        }
        return matches;
}
```

$\Omega(n)/\mathcal{O}(mn)$ time[2], $\Theta(1)$ space.

[2] $m = |needle|$, $n = |haystack|$

Analysis of naïve solution to **String**

- State is independent of problem
- Performance worsens as the number and length of prefix matches increases
- Length of prefix matches are bounded by length of search term
- Worst case: Match at every character ($m * n$ ops)
  Search term: `AAAA`
  Search text: `AAAAAAAAAAAAAAAAAAAA`
- Best case: No prefix matches (n ops)
  Search term: `AAAA`
  Search text: `BBBBBBBBBBBBBBBBBBBB`

Can we tighten the upper bound?

The **String** Problem—Prefix skips

While verifying a match, we ought be able to eliminate other match candidates.

- Search for `ACGT`
  Search text: `ACGACGT`
  Fail 3, skip 3, win 4 (7 ops)
  Search text: `AAAAAAAAAAAAAAAAAAA`
  Fail 1, skip 1, fail 1, skip 1...(n ops)
- Search for `ATATAT`
  Search text: `ATATATATAT`
  Win 6, skip 4, win 2, skip 2, win 2 (10 ops)
  Search text: `ATATAATATAT`
  Fail 6, skip 5, win 6 (11 ops)

From this insight arises the **Knuth-Morris-Pratt** algorithm (1977).

The **String** Problem—KMP Algorithm (preprocessing)

Construct a tabular *failure function*:

```
void kmptable(const char *needle, int *t){
        int pos = 2, cnd = 0;
        t[0] = -1;
        t[1] = 0;
        while(pos < strlen(needle)){
                if(needle[pos - 1] == needle[cnd]){
                        t[pos++] = ++cnd;
                }else if(cnd){
                        cnd = t[cnd];
                }else{
                        t[pos++] = 0;
                }
        }
}
```

$\Theta(m)$ time, $\Theta(m)$ space.

The **String** Problem—KMP Algorithm (search)

Search using the precomputed table:

```
unsigned kmp(const char *needle, const char *haystack,
                                        const int *t){
        unsigned matches = 0, m = 0, i = 0;
        while(m + i < strlen(haystack){
                if(needle[i] == haystack[m + i]){
                        matches += (i == strlen(needle) - 1);
                        ++i;
                }else{
                        m = m + i - t[i];
                        i = t[i] > -1 ? t[i] : 0;
                }
        }
        return matches;
}
```

$2n \in \Theta(n)$ time, $\Theta(1)$ space. The full procedure is thus $\Theta(n + m)$ time and $\Theta(m)$ space. As T is independent of the text being searched, it can be reused, yielding an amortized time $\Theta(n)$.

The **String** Problem—Other solutions

KMP is hardly the last word in string matching!

- **Boyer-Moore** matches from the back, and can skip characters in some cases, achieving sublinear time. Its worst case does not improve on the naïve solution:
  $\Omega(\frac{n}{m})/\mathcal{O}(mn)$, average $\mathcal{O}(\frac{n \log_{|\Sigma|} m}{m})$ (random text)
- **Boyer-Moore-Galil** tightens the worst case to $\mathcal{O}(n)$
- **Horspool** reduces state and preprocessing
- **Backwards DAWG Match** (1994, suffix automaton)
- **Backwards Oracle Match** (2001, factor oracle)
- Bit-parallel approaches (**Shift-OR**, **BNDM**, ...)

The **Multistring** Problem

Given

- an alphabet Σ,
- a set of patterns $p_0, p_1, \ldots, p_n$,
- and a text $t_0, t_1, \ldots, t_m$,

find and distinguish all matches.

Note that multiple p_i might be matched at a given t_i.

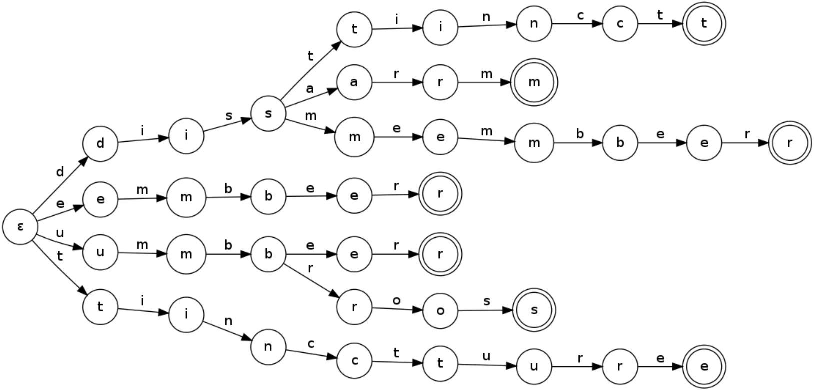
The **Multistring** Problem—Naïve solution

```
unsigned mnaive(const char **needles, const char *haystack){
        unsigned matches = 0;

        while(*needles){
                matches += naive(*needles,haystack);
                ++needles;
        }
        return matches;
}
```

Iterated application of [your favorite solution to **String** here]

Tries



$\mathcal{O}(n)$ lookup datastructure:

a $|\Sigma|$-ary rooted directed acyclic graph (a $|\Sigma|$-tree)

Basis of the **Aho-Corasick** algorithm (1975)

The **Multistring** Problem—Aho-Corasick

- Build the trie
- Augment each path with a *suffix link* to the longest **path** in the trie matching a suffix

  Most of these will typically be the root
- Augment each path with a *match link* to the longest **entry** in the trie matching a suffix

  Most of these will typically be null
- On each character of the searchtext, move through the trie. If there is no transition, traverse the suffix link chain until a transition is found, or the root has been checked.
- Following the transition, report matches for each element on the match link chain.

The **Multistring** Problem—Advanced Aho-Corasick

Trade space for time:

- Merge suffix link chain transitions directly into each node
- Collect match link chain as a set in each node

This solves **Multistring** in $\Theta(n)$ time, requiring space for $\Theta(P)$ nodes and $\mathcal{O}(P|\Sigma|)$ transitions ($P = \sum_{i=0}^{n} p_i$). The preprocessing can, like in KMP, be amortized over multiple search texts.

The **Multistring** Problem—`RegexEngine.java`

```
public List<T> match(char c){
        RegexNode next = node.getTransition(c);
        if(next == null){
                next = automaton.startMatch().getTransition(c);
                if(next == null){
                        next = automaton.startMatch();
                }
        }
        node = next;
        return node.getMatches();
}
```

The **MultiRE** Problem

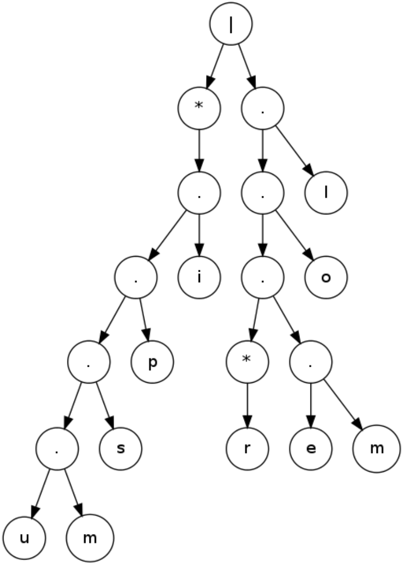
Given

- an alphabet Σ,
- a set of regular expressions $r_0, r_1, \ldots, r_n$,
- and a text $t_0, t_1, \ldots, t_m$,

find and distinguish all matches.

Note that multiple r_i might be matched at a given t_i.

Regular Expressions—Parse Tree



Result of parsing "(r*emol)|((umspi)*)"

GNFAs, NFAs, and DFAs

These classes of finite automata are characterized by

- A finite set of states S,
- A finite alphabet Σ,
- A start state $s_0 \in S$,
- A set of accepting states $S_a \subset S$,

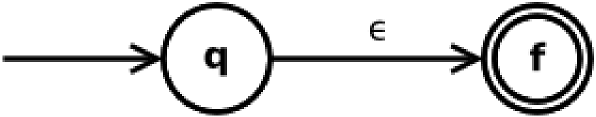
And a transition function $T(s \in S, i \in \Sigma) \rightarrow S_{next} \subset S$.

In a **GNFA**, the transitions are regular expressions on Σ.
In a **NFA**, the transitions are from Σ or ϵ.
In a **DFA**, the transitions are from Σ.

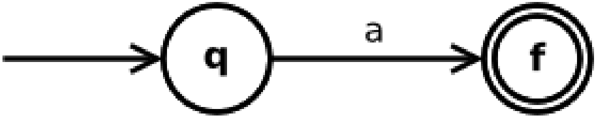
GNFAs are no more powerful than NFAs, which are no more powerful than DFAs!

The Thompson Construction—Part 1

Encoding the empty string:

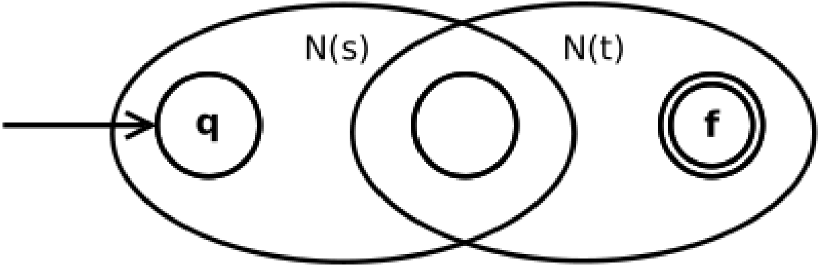


Encoding a symbol from Σ:



The Thompson Construction—Part 2

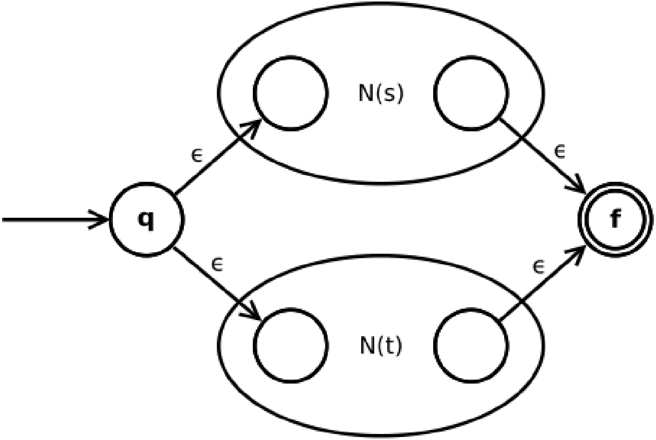
Concatenation of two NFAs $N(s)$ and $N(t)$:



Initial state of $N(s)$ becomes initial state of resulting NFA.
Final state of $N(t)$ becomes final state of resulting NFA.

The Thompson Construction—Part 3

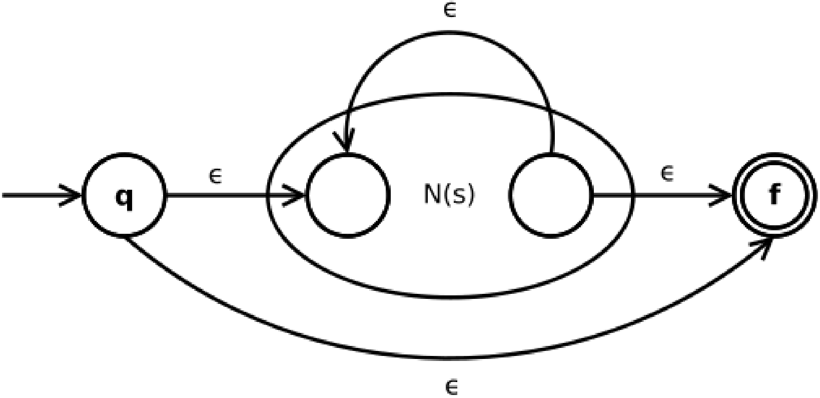
Union of two NFAs $N(s)$ and $N(t)$:



New initial state takes ϵ-transitions to initial states of $N(s)$ and $N(t)$. Final states of both take ϵ-transitions to a new final state.

The Thompson Construction—Part 4

Kleene closure over NFA $N(s)$:



New initial state takes ϵ-transitions to initial states of $N(s)$ and new final state. Final states of $N(s)$ take ϵ-transitions to new final state. Old final state takes ϵ-transition to old start.

NFA to DFA

Matching an NFA can take superlinear time, since at each step we must keep track of the current set of states, and evaluate a transition from each.

For any NFA, there exists an equivalent DFA—construct it!
Powerset construction (Rabin and Scott, 1959)

Minimize the DFA:
Coarsest common refimement + radix sort (Moore, 1956)
Inverted powerset (Brzozowski, 1963)
Partition refinement/Myhill-Nerode equivalence (Hopcroft, 1971)

We can now match arbitrary text against our multiple regular expressions, in linear time. Any questions?