

CDM

Pattern Matching

Klaus Sutner
Carnegie Mellon University
www.cs.cmu.edu/~sutner

Battleplan

- General Pattern Matching
- Conversion to Regular Expressions
- Conversion to Machine
- Realistic Regular Expressions
- Limitations of Regular Languages

General Pattern Matching

Warm Up

We already know how to search for strings or even for a finite collection of strings in a text.

Alas, often more complicated searches are required. For example, we might be interested in finding all blocks in a file that can be interpreted (at least in principle) as words in the English language (as opposed to binary data, numbers and such like). This can be very useful in examining an executable, finding comments in image file and so on. The old Unix utility `strings` does exactly that.

Converting a complete English dictionary into a string matcher, say, via Aho-Corasick, is a bad idea, we want a simple, fast tool. How do we describe a potential English word?

A similar problem would be to extract all URLs from an html-file or all numbers in scientific notation from a data-file.

Some Criteria

Here is a fairly straightforward list of search criteria for the "English word" problem.

- It has to be a block of upper-case or lower-case letters.
- It must be delimited by a space or one of a few other special characters (commata, periods, hyphens, ...).
- It should have length at least 4 (this is purely a heuristic to rule out false hits).
- It should have length at most 20 (again, a heuristic).

For the record: the classical `strings` program does not conform to these rules; still, they seem reasonable.

Also, we are just looking for potential words, we do not check against an actual dictionary.

Pattern Matching

How would we go about searching for a pattern, described by these or similar rules?

It is tempting to use a finite state machine based approach, generalizing the methods for single/finitely-many word searches. So we would like to construct some type of finite state machine that, given an input stream, will check if the input matches the pattern.

As always, there are many variants: we might want to stop at the first match, we might want to enumerate all matches, we might simply want to count the matches, and so forth.

Algorithmic Issues

- How exactly does one specify the patterns?
We need some simple notation system; e.g., we would like to be able to type in a pattern at the commandline.
- How does one generate the corresponding FSM from the pattern?
In particular, what type of machine should be generated? E.g., DFAs are faster for matching but are slower to build.

As always, computational hardness is lurking right around the corner, so one cannot be too ambitious about the admissible patterns. However, the regular expressions that we will introduce in a moment work extremely well in practice. This is one of the shining success stories in CS. Pattern matching based on regular expressions (and various generalizations) is used in many classical tools:

grep, sed, awk, perl, vi, emacs, agrep, Python, ...

Kleene's Theorem

Our notation system is based on Kleene's classical result.

Theorem 1. Kleene

Every regular language over Σ can be constructed from \emptyset and $\{a\}$, $a \in \Sigma$, using only the operations union, concatenation and Kleene star.

Regular languages are closed under other operations such as intersection and complement, but these are not needed to construct a regular language from the basic ones.

Before we sketch the proof, let us introduce the notation system suggested by Kleene's result.

Regular Expressions

Definition 1. A **regular expression** is a term constructed as follows:

- *Basic expressions:* \emptyset , a for all $a \in \Sigma$
- *Operators:* $E_1 + E_2$, $E_1 \cdot E_2$, E^* .

Since we are using infix notation we allow parentheses for grouping.

The dot for concatenation is usually not written.

One usually allows ε as a primitive denoting the empty word, though that is technically redundant since $\emptyset^* = \{\varepsilon\}$.

Regex Example

Example 1.

All words containing bab : $(a + b)^*bab(a + b)^*$.

All words containing 3 a 's: $b^*ab^*ab^*ab^*$

All words not containing aaa : $(\varepsilon + a + aa)(b + ba + baa)^*$

Exercise 1. Construct a regex for all words with an even number of a 's and b 's.

Exercise 2. Construct a regex for all words with an odd number of a 's and b 's.

Proof Sketch Kleene

Suppose we have an NFA that accepts L . Assume $Q = [n]$. For p, q in Q define

$$L_{p,q} = \mathcal{L}(\langle Q, \Sigma, \delta, \{p\}, \{q\} \rangle)$$

Then $L = \bigcup_{p \in I, q \in F} L_{p,q}$ and it suffices to construct regular expressions for the $L_{p,q}$.

In order to obtain an inductive argument, define a run from state p to state q to be **k -bounded** if all intermediate states are no greater than k . Note that p and q themselves are not required to be bounded by k .

Now consider the approximation languages:

$$L_{p,q}^k = \{x \in \Sigma^* \mid \text{there is a } k\text{-bounded run } p \xrightarrow{x} q\}.$$

Note that $L_{p,q}^n = L_{p,q}$.

Proof Sketch, contd.

One can build expressions for $L_{p,q}^k$ by induction on k .

For $k = 0$ the expressions are easy:

$$L_{p,q}^0 = \begin{cases} \sum_{\tau(p,a,q)} a & \text{if } q \neq p, \\ \sum_{\tau(p,a,p)} a + \varepsilon & \text{otherwise.} \end{cases}$$

So suppose $k > 0$. The key idea is to use the equality

$$L_{p,q}^k = L_{p,k}^{k-1} \cdot (L_{k,k}^{k-1})^* \cdot L_{k,q}^{k-1} + L_{p,q}^{k-1}.$$

Done by IH.

Conversion To Regular Expression

Machine to Regex

There are two basic approaches to converting a finite state machine to a regular expression:

- Linear systems of equations
The machine is converted into a system of linear equations over the language semiring. The system can then be solved using Arden's Lemma.
- Kleene's State elimination method
The proof of Kleene's theorem provides a dynamic programming algorithm.

Converting a finite state machine to a regular expression is a bit of an academic exercise.

The key problem is that for both approaches to yield manageable expressions one needs to simplify the intermediate results. There are heuristics to do that, but in general simplification is computationally hard.

Arden's Lemma

Consider a linear equation of the form

$$X = A \cdot X + B.$$

where $A, B \subseteq \Sigma^*$ are arbitrary languages, though we will be mostly interested in the case where A and B are finite or perhaps regular. We are looking for a solution $X_0 \subseteq \Sigma^*$.

As it turns out, linear equations are rather easy to solve.

Lemma 1. Arden's Lemma

Let A and B be languages over Σ . Then the equation $X = A \cdot X + B$ has a solution $X_0 = A^*B$. Moreover, if $\epsilon \notin A$, then this solution is unique. In any case, X_0 is the smallest solution (with respect to set-theoretic inclusion).

Proof

To see that X_0 is a solution note that

$$AX_0 + B = AA^*B + B = (A^+ + \epsilon)B = A^*B = X_0.$$

Now let Z be any solution, so $Z = AZ + B$. Then for all $k \geq 0$:

$$Z = A^{k+1}Z + (A^k + \dots + \epsilon)B.$$

Hence $X_0 \subseteq Z$.

Lastly suppose $\epsilon \notin A$ and let $x \in Z$; set $k := |x|$. Then $x \notin A^{k+1}Z$, whence necessarily $x \in (A^k + \dots + \epsilon)B \subseteq X_0$ and we are done. \square

In the applications of interest to us $\epsilon \notin A$ so that the solution is unique.

Regularity

Note that Arden's Lemma implies that equation $X = A \cdot X + B$ has a regular solution $X_0 = A^*B$ whenever A and B are regular. Moreover, if A does not contain ϵ this is the only solution. If A and B are given as rational expressions we obtain a rational expression for the solution.

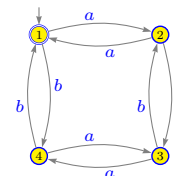
Example 2.

The equation $X = bX + a$ has b^*a as its unique solution.

By contrast, the equation $X = b^*X + a$ has infinitely many solutions $X_k = b^*(a + a^2 + \dots + a^k)$ and $X_1 = b^*a$ is the least solution. Indeed, there are uncountably many solutions $b^* \cdot L$ where $a \in L \subseteq a^*$ is arbitrary.

Even/Even Example

It is not hard to construct a DFA for the even/even language (and for all kinds of generalizations with different moduli). How about the corresponding regular expression?



We convert the DFA into a system of equations.

Corresponding System

$$X_1 = \varepsilon + aX_2 + bX_4 \quad (1)$$

$$X_2 = aX_1 + bX_3 \quad (2)$$

$$X_3 = aX_4 + bX_2 \quad (3)$$

$$X_4 = aX_3 + bX_1 \quad (4)$$

Substituting (2) and (4) into (1) and (3) we get

$$X_1 = \varepsilon + (aa + bb)X_1 + (ab + ba)X_3 \quad (5)$$

$$X_3 = (aa + bb)X_3 + (ab + ba)X_1 \quad (6)$$

Applying Arden's lemma to (6) we get

$$X_3 = (aa + bb)^*(ab + ba)X_1 \quad (7)$$

The Solution

Substituting (7) into (5)

$$X_1 = \varepsilon + ((aa + bb) + (ab + ba)(aa + bb)^*(ab + ba))X_1$$

Applying Arden's lemma one more time we get

$$X_1 = (aa + bb + (ab + ba)(aa + bb)^*(ab + ba))^*$$

which solution makes intuitive sense.

It is important to note, though, that we tacitly did quite a bit of cleanup along the way, writing the expressions in a particular appealing way.

Kleene's Method

A direct application of Kleene's algorithm with minimal simplifications during the computation produces the following monstrous regular expression for the even/even language.

$$\begin{aligned} &\varepsilon + b(bb)^*b + (a + b(bb)^*ba) (aa + ab(bb)^*ba)^* (a + ab(bb)^*b) + \\ &\quad \left(b(bb)^*a + (a + b(bb)^*ba) (aa + ab(bb)^*ba)^* (b + ab(bb)^*a) \right) \\ &\quad \left(a(bb)^*a + (b + a(bb)^*ba) (aa + ab(bb)^*ba)^* (b + ab(bb)^*a) \right)^* \\ &\quad \left(a(bb)^*b + (b + a(bb)^*ba) (aa + ab(bb)^*ba)^* (a + ab(bb)^*b) \right) \end{aligned}$$

It takes quite a bit of effort to check that this expression is correct, try it.

Another Example

Needless to say, solving systems of equations does not always produce a neat solution either. Consider the system

$$X = aX + bY$$

$$Y = \varepsilon + aX + ba^*$$

$$Z = aZ + b$$

Z can immediately be replaced by a^*b . If we solve for X in the first equation, $X = a^*bY$, and substitute in the second and then solve for Y and resubstitute we get

$$X = a^*b(a^+b)^*(\varepsilon + ba^*)$$

$$Y = (a^+b)^*(\varepsilon + ba^*)$$

So far, so good.

Different Approach

However, we could also first substitute the second equation into the first and solve for X (after getting rid of Z).

This leads to solutions

$$X' = (a + b(ab)^*aa)^*b(ab)^*(\varepsilon + ba^*)$$

$$Y' = (ab)^* (aa(a + b(ab)^*aa)^*b(ab)^*(\varepsilon + ba^*) + \varepsilon + ba^*)$$

Unless we made a mistake, these expressions must be equivalent to X and Y , but this is certainly not obvious from looking at them.

What is sorely missing here is a simplification algorithm that brings a regular expression into a normal form. We can check for equivalence by converting to finite state machines and then testing these for equivalence.

Conversion To Machine

Regex To Machine

We will ignore the issue of parsing a regular expression and simply worry about how to construct the finite state machine.

It is not hard to guess that we will build the machine by induction on the structure of the regular expression, but the question is what kind of machine we should be build.

DFAs are certainly a bad choice since we have to deal with concatenation and Kleene star. It is a fair guess that some kind of NFAE is the right target architecture.

As it turns out, the construction becomes quite a bit easier if insist on very special NFAEs. Of course, the right choice requires a bit of experimentation.

Begin/Exit Automata

Definition 2. A *begin/exit automaton (BEFA)* is an NFAE that has exactly one initial state b (the *begin*), exactly one final state e (the *exit*). Furthermore, no transitions end at b and no transitions start at e .

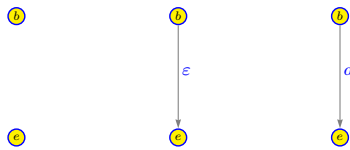
If we assume the state set $[n]$ a BEFA is essentially just a list of transitions: by renumbering we can make sure that $b = 1$ and $e = n$.

Hence the data structure representing a BEFA is particularly simple.

There are other ways to go about the conversion, but BEFAs are probably the most intuitive choice.

The Basic machines

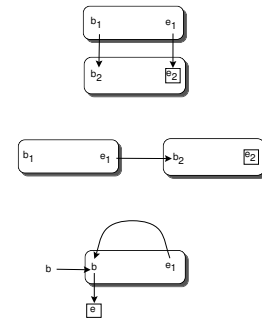
For expressions \emptyset , a and ϵ the corresponding BEFAs are as follows.



In practice, the BEFA for \emptyset is never used.

The Operations

Sum, concatenation and Kleene star are handled as follows.



All new transitions are ϵ -transitions.

Correctness

Note that the conditions on transitions to and from begin/exit are crucial for correctness, the constructions would not work for arbitrary NFAEs with unique initial and final state.

For example, in the Kleene star construction we cannot simply maintain e_1 as the exit since there is now a transition leaving that state.

Theorem 2. A regular expression can be converted in linear time into an equivalent begin/exit automaton.

Proof. The number of states is

$$2\alpha + 2\sigma$$

where α is the number of atomic symbols in the expression, and σ the number of Kleene stars.

□

BEFA Example

Expression $(\epsilon + a + aa)(b + ba + baa)^*$ produces a 6-state BEFA with transitions

1	1	1	2	3	4	4	5
a	a	ϵ	a	ϵ	b	ϵ	ϵ
2	3	3	3	4	5	6	4

ϵ -elimination produces an NFA with initial states $\{1, 3, 4, 6\}$ and final states $\{6\}$.

1	1	1	1	2	2	2	4	4	4
a	a	a	a	a	a	a	b	b	b
2	3	4	6	3	4	6	4	5	6

Deterministic Machines

The DFA obtained by deterministic simulation has 5 states (initial state 1 and final states {1, 2, 3, 4}).

	1	2	3	4	5
a	2	4	5	5	5
b	3	3	3	3	5

The corresponding minimal DFA has 4 states (initial state 1 and final states {1, 2, 3}).

	1	2	3	4
a	2	3	4	4
b	3	3	3	4

Blow-Up

The regular expression

$$(a + b)^*a(a + b)(a + b)(a + b)(a + b)(a + b)$$

for $L(a, -6)$ produces a BEFA with 10 states

1	2	2	3	4	5	5	6	6	7	7	8	8	9	9	
ϵ	a	b	ϵ	ϵ	a	a	b	a	b	a	b	a	b	a	b
2	3	3	4	2	5	6	6	7	7	8	8	9	9	10	10

... but the DFA obtained from conversion has 65 states; the minimal DFA has 64 states.

Realistic Regular Expressions

Extended Regex

Our definition of regular expressions directly mimics Kleene's theorem: all regular languages can be built from trivial components by union, concatenation and Kleene star.

Sometimes it is convenient to enhance regular expressions a bit. There are two types of enhancements:

- More compact ways to describe regular languages.
- Describing non-regular languages.

Type 1 comes down to notational convenience, but may have radical effects on the running time of the conversion algorithm. For example, we know that regular languages are closed under intersection and complement. If we were to add corresponding regular expressions for these operations we would still describe regular languages. However, the conversion process based on BEFAs now fails: the only way we can perform, say, complementation is by converting to a DFA first. This conversion may carry an exponential cost.

Type 2 requires a redesign of the acceptance testing algorithm: finite state machines are no longer sufficient, though the modifications may turn out to be fairly easy to do from an algorithmic point of view.

Intersection

As an example, consider the addition of a new operation symbol \cap for intersection to regular expressions.

What has to change in the conversion algorithm? We need to add a product automata construction.

From the implementation perspective this is not too bad, but it breaks polynomial bounds on the size of the machine constructed from a regex. The following theorem shows that there is little hope for a simple remedy.

Theorem 3. Suppose M_1, \dots, M_n is a list of DFAs (over the same alphabet). It is PSPACE-hard to check whether there is a string that is accepted by all the M_i .

Complement

Similar problems arise when a complementation operation $-$ is added. In the Real WorldTM complementation can often be kludged by piping.

```
fgrep foo | fgrep -v foobag
```

But to do this in general we have to construct a machine for the complement of a regular language – and that requires to build a DFA first, at a potentially exponential cost.

Also note that complement together with union automatically produces intersection, so the hardness result from above applies.

Iteration

A very handy feature in most regular expression matchers generalizes concatenation and Kleene star.

notation	number of matches
*	≥ 0
+	≥ 1
?	$= 0, 1$
{n}	$= n$
{n,}	$\geq n$
{n,m}	$\geq n, \leq m$

So one can write things like

```
egrep -e '0\.[0-9]{5}'
```

to find all decimal numbers starting with 0 with exactly 5 digits after the decimal point.

Limitations of Regular Languages

Limitations of Regular Languages

A brute-force cardinality argument that most languages fail to be regular. More importantly, relative simple and practically relevant languages such as

$$L = \{ a^i b^i \mid i \geq 0 \}$$

turn out to be non-regular.

It would be nice to have a reasonably simple test that can identify non-regular languages as such.

Using Quotients

One approach is to count quotients.

Proposition 1. *A language fails to be regular if, and only if, it has infinitely many left quotients.*

Example 3. $L = \{ a^i b^i \mid i \geq 0 \}$ fails to be regular.

For consider the quotients $K_i = (a^i)^{-1} L$. The shortest word in K_i is b^i , so they are all distinct.

Alas, making sure that there are indeed infinitely many quotients can be difficult, we really need a less challenging test is needed.

Pumping Lemma

Lemma 2. *Pumping Lemma*

For every regular language L there is a constant n such that for all words $x \in L$ with $|x| \geq n$ we have $x = uvw$ where $v \neq \varepsilon$, $|uv| \leq n$ and $uv^t w \in L$ for all $t \geq 0$.

Proof.

Consider the minimal DFA M for L and let n be the number of states of M . Then any word in L of length at least n must trace a loop in the diagram of M . The claim follows. \square

The Pumping Lemma is useless to establish regularity but often the weapon of choice to refute it.

PL Example 1

$L = \{ a^i b^i \mid i \geq 0 \}$ fails to be regular.

Assume otherwise.

Let n be as in the PL and consider $x = a^n b^n \in L$.

Then $x = uvw$ and $v = a^i$ for some $i > 0$.

But then $uv^t w \notin L$ for all $t \neq 1$, contradiction.

It follows that the language P of balanced parentheses is also non-regular.

For otherwise $P \cap a^* b^* = L$ would also be regular, which assertion we already know to be false.

PL Example 2

$L = \{zz \mid z \in \Sigma^*\}$ fails to be regular.

Assume otherwise.

Let n be as in the PL and consider $x = ab^n ab^n \in L$.

Then $x = uvw$ and $|uv| \leq n$. If $v = a$ we get a contradiction with $t = 0$. If $v = b^i$ for some $i > 0$ we also get a contradiction with $t = 0$.

The problem here really is that a DFA cannot remember an arbitrarily long prefix z which is needed to check the remainder of the input.

PL Example 3

$L = \{zz^r \mid z \in \Sigma^*\}$ fails to be regular.

Assume otherwise.

Let n be as in the PL and consider $x = (ab)^n (ba)^n \in L$.

Then $x = uvw$ and $|uv| \leq n$.

A straightforward but tedious argument shows that $uv^t w$ cannot be a palindrome for any $t \neq 1$.

As in the last example, a DFA cannot remember an arbitrarily long prefix z which is needed to check the remainder of the input.

Complexity of Languages

Incidentally, between

$$L_1 = \{zz \mid z \in \Sigma^*\}$$

and

$$L_2 = \{zz^r \mid z \in \Sigma^*\}$$

the second one (even length palindromes) is much simpler:

To recognize these words one only needs to attach a stack to a FSM (context free language).

For L_1 , a simple stack is not sufficient (context sensitive language).

Non-Regular Extensions

Many pattern matchers allow the user to specify repetitions of previously matched parts of a string.

The standard `egrep` for example allows

```
egrep -e '[a-z]*\1'
```

which will match words of the form ww . It is not hard to see that these words form a non-regular language.

The `\1` is a so-called back-reference and matches whatever string has already matched the expression in parens.

Back-References

In general, `\n` refers to the string that has already matched the n th paren pair, which has to occur before the back-reference.

Also note that one actually has to understand the matching mechanism in greater detail (greedy versus lazy).

Usually the longest match possible is chosen.

For example, the expression

```
((a|b)*c\2)*
```

matches all words in

$$\{xcx \mid x \in \{a,b\}^*\}$$

Warning

Careful, though, the fancy stuff sometimes does not work right.

```
egrep -e '(.)?(.?)?(.?)?(.?)\4\3\2\1'
```

is supposed to match all palindromes of even length up to 8, but only works on lengths 6 and 8.

The `.` is a special symbol that matches a single character.

On `grep-2.5.1-4` some of the counting extensions also do not work reliably.