Regular expression matching

Eirik Aung

2022 nov

Motivation

- We want to search text using regular expressions, i.e. given a regular expression, find all substrings that match it
- `grep` does this:

```
in3120-2022/data grep --line-number 'data\(språk\|problem\)' no.txt
3078:Erlandsen tør ikke å si noe om hvor lenge dataproblemene vil vare.
7870:På dataspråket kalles dette for «vannhullsangrep».
```

Outline

- What is a regular expression
- What is an NFA
- From regular expression to NFA
- Matching in text

Regular expressions

- An *alphabet* is a set of symbols, e.g.

$$\Sigma = \{a, b, c, \ldots, x, y, z\}$$

- A *language* (over an alphabet) is a set of strings, e.g.

$$A = \{cat, dog, sloth\}$$

- ε is used to denote the empty string (think "" in python)
- $\emptyset$ is used to denote the empty language

Regular expressions

- Definition (1.52 Sipser): R is a *regular expression* if R is
  1. some symbol in the alphabet Σ,
  2. ε,
  3. $\emptyset$,
  4. $(R_1|R_2)$, where R_1 and R_2 are regular expressions,
  5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions, or
  6. (R_1^*), where R_1 is a regular expression.

  - $|$ is the *alternation* operator (or)
  - $\circ$ is the *concatenation* operator (join)
  - * is the *Kleene star* operator (zero or more repetitions)
- Sometimes we drop the $\circ$ operator and write *cat* instead of $c \circ a \circ t$
- If e is a regular expression we write $L(e)$ to denote the language *generated* by e

Examples

- $L(he(y|llo)) = \{hey, hello\}$
- $L(r(e^*)d) = \{rd, red, reed, reeed, \ldots\}$

Regular expression

To define the language generated by a regular expression formally, then for languages A, B we define

$$A \circ B = \{xy : x \in A, y \in B\},$$
$$A^* = \{x_1 x_2 \cdots x_k : k \geq 0, x_i \in A\}.$$

Given alphabet Σ we define

1. $L(\emptyset) = \emptyset$,
2. $L(a) = \{a\}$ if $a \in \Sigma \cup \{\varepsilon\}$,
3. $L(R_1 | R_2) = L(R_1) \cup L(R_2)$,
4. $L(R_1 \circ R_2) = L(R_1) \circ L(R_2)$,
5. $L(R_1^*) = L(R_1)^*$.

Regular expression

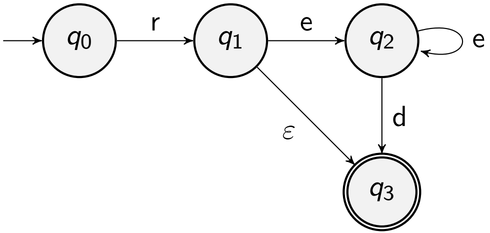
- Reverse polish notation will make our lives easier
- $(5 + 3) * (2 - 3)$ written as $5\,3\ +\ 2\,3\ -\ *$
- $(a|b) \circ (c^*)$ written as $a\,b\,|\,c\,^*\,\circ$
- Keep elements on stack, if we see operator then pop, eval, push
- Will be useful when generating NFA!
- Dont have to worry about precedence

Nondeterministic finite automata

- A machine that takes a string as input, and accepts/rejects

Example

q_0 is the initial state, q_3 is the accepting state



- Nondeterminism: can be in multiple states at same time
- Accept if one of the current states are accepting after reading entire input
- The set of strings the NFA accepts is called the language of the NFA

Nondeterministic finite automata

Definition (1.37 Sipser) A *nondeterministic finite automaton* is a 5-tuple $N = (Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set of states,
2. Σ is a finite alphabet,
3. $\delta\colon Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$ is the transition function,
4. $q_0 \in Q$ is the start state, and
5. $F \subseteq Q$ is the set of accepting states.

We say that N *accepts* a string w if $w = y_1 y_2 \cdots y_m$ where $y_i \in \Sigma \cup \{\varepsilon\}$, and there exists a sequence of states $r_0, r_1, \ldots, r_m$ where $r_i \in Q$ such that

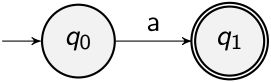
1. $r_0 = q_0$,
2. $r_{i+1} \in \delta(r_i, y_{i+1})$ for $0 \leq i < m$, and
3. $r_m \in F$.

From regular expression to NFA

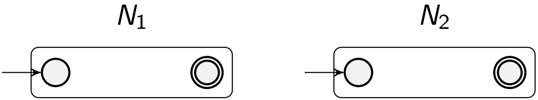
- Given a regular expression R we want to generate an NFA N such that $L(R)$ is exactly the language accepted by N.
- We ignore regular expressions containing $\emptyset$ as no substring will match it.
- It will be practical for our construction if every NFA has exactly one accepting state, and every state has maximum two out edges.

From regular expression to NFA

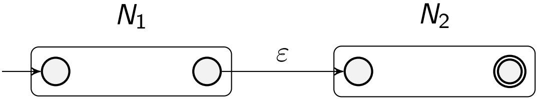
- If the regular expression is simply a for $a \in \Sigma \cup \{\varepsilon\}$, we construct NFA



- If for regular expressions R_1, R_2 we have constructed NFAs N_1, N_2

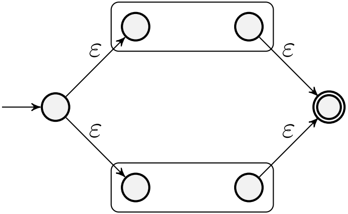


we construct the NFA for $R_1 \circ R_2$ by



From regular expression to NFA

- We do $R_1|R_2$ by



- How to do (R_1^*)? Exercise

From regular expression to NFA

- Given regular expression in reverse polish notation, generate the NFA using a stack
- How will the NFA for these examples be constructed?

Examples

- $(a|b)c$ same as $a\,b\,|\,c\,\circ$
- $he(y|ll)$ same as $h\,e\circ y\,l\,l\circ|\circ$

Matching in text

- How can we know if a string is accepted by the NFA?
    1. Keep track of current states with list, starting with just the initial state.
    2. For every input symbol read, the next list of current states will be generated by by looking at the transitions of the previous list of states.
    3. ε transitions have to be taken care of (can do recursively).
    4. After entire input is read, accept if accepting state in list of current states.

Matching in text

```
states = [nfa.initial]
for c in s:
    new_states = []
    for state in states:
        add(state.outa, c, new_states)
        add(state.outb, c, new_states)
        # adds to new_states if transition label
        # and character correspond.
        # also takes care of epsilon-transitions,
        # and makes sure duplicates arent added to new_states,
    states = new_states

for state in states:
    if state.accept:
        return True
return False
```

Matching in text

- We can now see if a regular expression matches a string.
- But we are interested in matching substrings in a larger text.
- Problems: overlapping matches and matching multiple at once. What do we return to user?

Examples

- `hello` on regular expression $hello|llo$
- `giraffe` on $gira|raffe$
- `aaaaa` on a^*
  - We will do non-overlapping shortest match.

Matching in text

1. General idea same as before, we "start" the NFA for every substring.
2. Keep only one list of current states to maintain low complexity.
3. Each state maintains an index to maintain which substring we started on to reach this state.
4. If matching state found, use index to find start of matching substring.

Matching in text

- Time complexity $\mathcal{O}(mn)$ where n is size of text we are searching through, and m is number of states in NFA (which is a constant times the size of the regular expression).
- n typically very large, m typically very small.

Matching in text

- Demos: `matcher.py`, `matcher.c`
- `no.txt`: 10,000 lines, >1M characters

```
time python matcher.py 'da.t.a.sp.r.å.k.pr.o.b.l.e.m.|.' no.txt
3078:Erlandsen tør ikke å si noe om hvor lenge dataproblemene vil vare.
7870:På dataspråket kalles dette for ≪vannhullsangrep≫.

real    0m0.605s
user    0m0.595s
sys 0m0.010s
```

- `matcher.c` is a modification to an NFA program by Russ Cox, same query in 0.006s
- grep does same query in 0.004s (but scales way better than our naive `matcher.c`)

Matching in text

- How does `grep` do it? "GNU grep is based on a fast lazy-state deterministic matcher hybridized with Boyer-Moore and Aho-Corasick"
- Deterministic matcher: deterministic version of NFA (aka DFA), may have exponentially many more states.
- Lazy: can't keep track of all states at once, construct on the fly and use cache
- Russ Cox' series of articles is great if you want to learn more about implementations

References

- R. Cox. *Regular Expression Matching Can Be Simple And Fast*, https://swtch.com/~rsc/regexp/regexp1.html
- M. Sipser. *Introduction to the Theory of Computation*. 3rd edition (international).