String Algorithms

CITS3001 Algorithms, Agents and Artificial Intelligence

CLRS Chapter 32

2019, Semester 2

Tim French
Department of Computer Science and Software Engineering
The University of Western Australia

Summary

- String a sequences of characters and symbols. Many different applications require fast processing of such data:
  - Search and regular expression matching
  - Bioinformatics and gene sequencing
  - Data compression
  - Plagiarism detection
  - …
- We will look at two of the most common string operations
  - Four algorithms for pattern-matching
  - Two algorithms for the longest common subsequence problem
- We will look at the design, the correctness, and the complexity of each algorithm

Pattern matching

- Consider two strings T (the *text*) and P (the *pattern*) over a finite alphabet Σ, respectively with lengths *n* and *m*
- The *pattern-matching problem* is to find occurrences of P in T
  - Either all occurrences, or just the first occurrence
- Pattern-matching has many important applications, e.g.
  - Text-editing programs
  - DNA processing
  - Searching bitmaps and other types of files

- We shall use a running example where

T = abaaabacccaabbaccaababacaababaac

P = aab

- We describe a match by giving the number of characters *s* that the pattern must be moved along the text to give a *valid shift*
  - $s \in \{3, 10, 17, 24\}$ are the valid matches for P in T

The Naïve Method

- We could simply consider each possible shift in turn
- e.g. when *s* = 0 we compare

/abaaabacccaabbaccaababacaababaac

aab

  - Which fails at the second character

- When *s* = 1 we compare

a/baaabacccaabbaccaababacaababaac

aab

  - Which fails at the first character
  -

- *s* = 3 succeeds

```
procedure naive(T, P):
result = { }
for s = 0 to n – m      // for each possible shift
    match = true
    for j = 0 to m – 1   // check each item in P
        if T[s+j] ≠ P[j]
            match = false
    if match
        result = result + {s}
```

Analysis of the naïve method

- There are $n–m+1$ possible shifts
- In the worst case, each possible shift might fail at the last (the m^{th}) character
- Thus the worst case involves $O(m(n – m + 1))$ time
- The naïve string matcher is inefficient in two ways

In the shifting process: when it checks the shift *s*, it ignores whatever information it has learned while checking earlier shifts $s' < s$ e.g. given:

000000100000001000000100000001
0000000
$s = 0$ fails at the 7^{th} item.That item is also involved in checking $s = 1, 2, \ldots, 6$, so we should be able to re-use this information
Knuth-Morris-Pratt and Boyer-Moore exploit this inefficiency

In the comparison process: it compares the pattern and the text item-by-item

Surely there's a better way!

Rabin-Karp exploits this inefficiency

Rabin-Karp Algorithm

THE UNIVERSITY OF WESTERN AUSTRALIA

- Rabin-Karp tries to replace the innermost loop of the naïve method with a single comparison, wherever possible

e.g. if the alphabet is decimal digits, and given

1229384912817608213081762831O1

176

- We can represent the pattern as a single (multi-digit) integer; then at each shift we just need to perform a single comparison
  - e.g. at s = 0, instead of comparing the sequences “176” and “122”, we compare numbers 176 and 122

- We can calculate all of the numbers to compare with in (amortised) O(n) time
- Calculate the first number z = 122 in O(m) time
- Calculate the next number in O(1) time by

  $$z \bmod 10^{m-1} * 10 + T[m] = 229$$

- Total time = O(m + 1 * (n – m)) = O(n)

Pseudo-Code

```
procedure rabinkarp(T, P):
p′ = 0
for j = 0 to m − 1                        //
turn P into a number
     p′ = p′ * 10 + P[j]
z = 0
for j = 0 to m − 2                        //
get the first number in T
     z = z * 10 + T[j]
result = { }
for s = 0 to n − m                        //
check each possible shift
     z = z mod 10^(m-1) * 10 + T[s+m−1]
            // update z
     if z == p′
          result = result + {s}
```

- If | Σ | = d, we can use d-ary numbers instead of decimal
  - But still this version assumes that the calculated values can be stored in one word, and hence can be compared in a single operation
- The complexity of Rabin-Karp is
  - O(m) for the pre-processing
  - O($n - m$) for the main loop

What if P doesn't fit in one word?

- Sometimes the combination of *m* and *d* means that the comparisons can't be done in one word
  - The biggest possible number is $d^m - 1$, which might be huge
- We can still use a *filter value* derived from P to speed up the pattern-matching process
- Choose a prime number *q* such that *dq* fits into one word
  - Best to make *q* as large as possible
- Calculate p'' = p' mod *q*, and z' = z mod *q*
  - It should be clear that z' can be updated in O(1) time
- At each iteration, compare p'' and z'
  - If they are different, then p' and z are different
  - If they are the same, compare P and the relevant characters in T
- In the worst case where p'' and z' match often, this is $O(m(n - m + 1))$
- In the more common case where there are few matches, it is a lot faster

Example

Consider this example

T = 54142135621414

P = 414

Assume q = 13, so p'' = 414 mod 13 = 11

What values does z' take, associated with T?

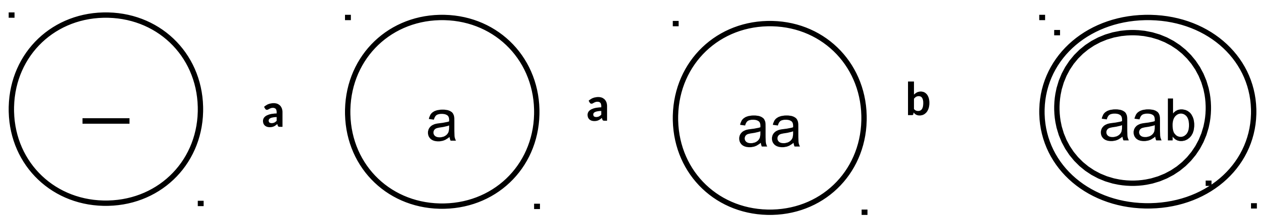
A bigger q means a bigger range of values for z', which (usually) means fewer spurious hits

```
T  = (541)42135621414
z' =    8
              filter!
T  = 5(414)2135621414
z' =     11
        check: valid!
T  = 54(142)135621414
z' =     12
        filter!
…
T  = 5414213562(141)4
z' =             11
        check: spurious!
```

Pattern-matching with a FSM

Suppose we want to build an FSA that recognises any string ending with aab

We can start by building the backbone of the machine



Clearly this will recognise the string aab

- But what about other strings?
- And what about longer strings?
- i.e. what about the other five arrows?

In each case, we need to go to the state that captures the *longest prefix of* aab *we have seen so far*

e.g. if State aab gets an a, that means the last four characters were aaba

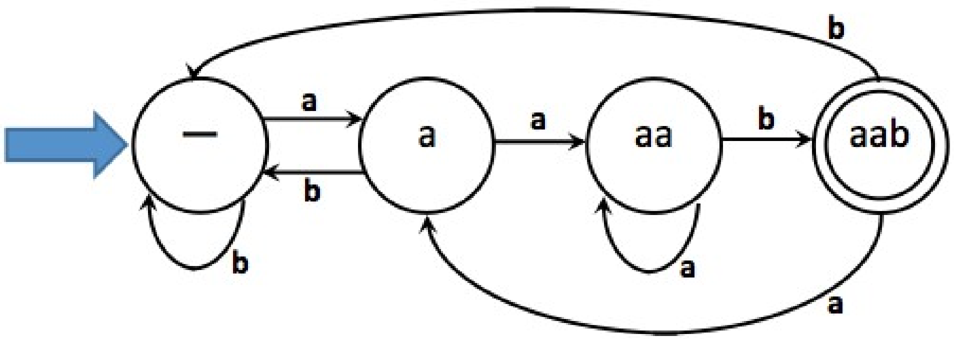
- We should go to State a

e.g. if State aa gets an a, that means the last three characters were aaa

- We should stay in the same state

Building a FSM

Thus the complete machine will be



Make sure you understand all of these arrows
The machine basically encodes the pattern P
Once we have this machine, we can do pattern-matching just by executing it for the text T and counting the characters as we go
e.g. run the machine for

T = abaaabacccaabbaccaababacaababaac

Pattern-matching in O(n) time!

How long does it take to build the FSA?
It has $m+1$ states, each of which needs $|\Sigma|$ arrows
With clever implementation, we can find these arrows in $O(m|\Sigma|)$ time
Thus the overall complexity is $O(n+m|\Sigma|)$
But we can do even better…

Knuth-Morris-Pratt algorithm

Knuth-Morris-Pratt uses the same principle as the FSA

- When we find a character x that doesn't match, go to a state that recognises what we have already seen

Consider P = abbabaa and T = abbabxyz

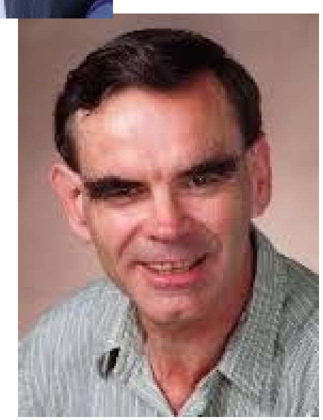
In the FSA, there is a different arrow for each possible value of x

- We move to the state where abbabx matches the longest prefix of P
- Hence quite a complex machine, for big alphabets

In KMP, we *ignore* x

- We move to the state where abbab matches the longest prefix of P, and we inspect x again
- Hence each state needs only two arrows: one for a match, and one for a non-match

We pre-compute a *prefix function* that returns, for each index q into P, the largest $k < q$ such that P_k is a suffix of P_q



The prefix function

We need to consider each prefix of P = abbabaa

Thus the prefix function for P is {(1,0), (2,0), (3,0), (4,1), (5,2), (6,1), (7,1)}

Because the prefix function depends only on P, it can be derived in O(*m*) time

Much faster for big alphabets

q	P_q	proper prefixes to consider	k
1	a	none	0
2	ab	a	0
3	abb	ab, a	0
4	abba	abb, ab, a	1
5	abbab	abba, abb, ab, a	2
6	abbaba	abbab, abba, abb, ab, a	1
7	abbabaa	abbaba, abbab, abba, abb, ab, a	1

In each row of the table:

- q is an index into P
- P_q is the first q characters of P
- The third column lists all proper prefixes of P_q
- k is the length of the longest sequence from the third column which is a suffix of P_q

Knuth-Morris-Pratt algorithm

THE UNIVERSITY OF WESTERN AUSTRALIA

```
procedure kmp(T, P):
π = prefix-function(P)
q = 0                       // stores the no. of
items matched
result = { }
for s = 0 to n - 1          // for each possible
shift
     while q > 0 and P[q] ≠ T[s]
          q = π[q]
     if P[q] == T[s]
          q = q + 1
     if q == m
          result = result + {s-m+1}
          q = π[q]
```

The computation of the prefix function is broadly similar to this pseudo-code for the main algorithm

Ps. 1005–6, CLRS

Notice that KMP has nested loops – yet it is a linear algorithm. How can this be?

Heuristics

- Knuth-Morris-Pratt is O(m+n), which is clearly the optimal complexity in the worst case
  - In the worst case, we have to examine every character in both strings
  - But there are heuristic approaches that can perform better for some common cases

- A *heuristic* is a strategy that is used to guide a process or algorithm
  - e.g. if you lose your keys, look first in the place where you last remember seeing them
  - Heuristics can come from maths, logic, experience, common sense, …

- A well-designed heuristic helps in common and/or important cases, but not necessarily in all cases
  - If it worked well in all cases, it would be a rule!

- Two pattern-matching heuristics in particular are very effective with large alphabets and/or long patterns
  - The heuristics are incorporated into the Boyer-Moore algorithm

Basic Boyer-Moore algorithm

The basic Boyer-Moore algorithm is the same as the naïve algorithm, with only one change:

> The characters in the pattern are checked *right-to-left* instead of left-to-right

If a mismatch is found, the shift is invalid, and we try the next possible shift: s = s + 1

Both heuristics operate by providing a number other than 1 by which s can be incremented

- Hopefully usually bigger than 1!
- Effectively we can avoid checking some shifts
- Thus reducing the run-time of the algorithm!

The bad-character heuristic

Consider this example

```
T = once_we_noticed_that
P =             imbalance
```

Matching right-to-left, we fail at the i in T: This is known as a *bad character*

We know the i in T can only match an i in P. Therefore we can shift by 6 immediately

- No smaller shift can possibly work, Several shifts are never checked at all!

We pre-compute a function $\lambda\colon \Sigma \rightarrow \{0,1, \ldots, m\}$ such that for each character $c \in \Sigma$, $\lambda(c)$ is the rightmost position where c occurs in P

- Or 0 if $c \notin P$

Then when a mismatch occurs while looking at the j^{th} character in P, the bad-character heuristic suggests the shift-advance

- $s = s + (j - \lambda(T[s+j]))$

The bad-character heuristic sometimes suggests a negative shift, so it cannot be used alone

- Consider T = brabham and P = drab
- At s = 0, when comparing the d with the first b in T, the suggested shift will be $1 - 4 = -3$

The good-suffix heuristic

Consider this example

```
T = the_late_edition_of
P =          edited
```

Matching right-to-left, the *good suffix* comprises the characters that match at a given shift, i.e. here ed

We know the ed in T can only match an ed in P

Therefore we can shift by 4 immediately

- Again, no smaller shift can possibly work

We pre-compute a function $\gamma:\{0,1,\ldots,m\} \to \{0,1,\ldots,m\}$ such that $\gamma(j)$ is the smallest positive shift where P matches all of the characters that it still overlaps

Then when a mismatch occurs while looking at the j^{th} character in P, the good-suffix heuristic suggests the shift-advance

$$s = s + \gamma(j)$$

Example heurstic calculations

Consider the pattern

`P = one_shone_the_one_phone`

λ is easy to work out: simply the index of the rightmost instance of each character in Σ

e.g. $\lambda(e) = 23$, $\lambda(h) = 20$, $\lambda(a) = 0$

γ is a tad more complicated

j	matched	smallest shift to same fragment
22	e	6
21	ne	6
20	one	6
19	hone	14
18	phone	20

Given that $\gamma(18) \geq 18$, we can infer that $\gamma(j) = 20$, $j \leq 18$

Boyer-Moore just applies both heuristics and uses the better value

Improving good suffix

The good-suffix heuristic can be improved further, by not re-testing characters that we know are wrong

```
P = one_shone_the_one_phone
```

If we match the e on the end then fail at the n, there's actually no point doing a shift of 6

- We would be looking for an n again!

So the biggest safe shift is to the next occurrence of the good suffix which is preceded by a different character

- e.g. for the e we can do a shift of 10

The new table would be

j	matched	biggest shift to safe same fragment
22	e	10
21	ne	23
20	one	6
19	hone	14
18	phone	20

The pre-processing is now a bit more expensive

- We have lost monotonicity in the table
- But very likely it's worth the extra work

Longest common subsequence

Finally, we'll consider the problem of comparing string to see how similar they are…

Given two sequences X and Y, what is their longest common subsequence?

A subsequence of X is X with zero or more items omitted

- e.g. ABC has seven subsequences: ABC, AB, AC, BC, A, B, C
- A sequence with n items has 2^n-1 subsequences

So e.g. the LCSs of ABCBDAB and BDCABA are

BCBA

BCAB

BDAB

We can solve this problem efficiently using *dynamic programming*

A recursive relationship

The first step is to find a recursive rule whereby the main problem can be solved by solving smaller sub-problems

Assume that

$$X = x_1 \ x_2 \ \ldots \ x_m$$

$$Y = y_1 \ y_2 \ \ldots \ y_n$$

and that they have an LCS

$$Z = z_1 \ z_2 \ \ldots \ z_k$$

Clearly either $x_m = y_n$, or not

If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}

e.g. X = abcd, Y = paqd

If $x_m \neq y_n$, then at least one of x_m and y_n was discarded, and Z_k is an LCS of either X_{m-1} and Y, or X and Y_{n-1}

For this case there are three possibilities

e.g. X = abcd, Y = padq (q discarded)

e.g. X = abcd, Y = apqc (d discarded)

e.g. X = abcd, Y = apbr (both d and r discarded)

The recursive relationship

This allows us to define a recursive relationship:

$$
\begin{aligned}
\mathrm{LCS}(X_i, Y_j) &= [\ ], && \text{if } ij = 0 \\
&= \mathrm{LCS}(X_{i-1}, Y_{j-1}) + [x_i], && \text{if } x_i = y_j \\
&= \mathrm{longer}(\mathrm{LCS}(X_{i-1}, Y_j), \mathrm{LCS}(X_i, Y_{j-1})), && \text{if } x_i \neq y_j
\end{aligned}
$$

And a corresponding relationship on lengths:

$$
\begin{aligned}
\mathrm{len}(X_i, Y_j) &= 0, && \text{if } ij = 0 \\
&= \mathrm{len}(X_{i-1}, Y_{j-1}) + 1, && \text{if } x_i = y_j \\
&= \max(\mathrm{len}(X_{i-1}, Y_j), \mathrm{len}(X_i, Y_{j-1})), && \text{if } x_i \neq y_j
\end{aligned}
$$

Applying this rule directly as an algorithm requires the calculation of LCS(X_i, Y_j) for all $0 \leq i \leq m$ and $0 \leq j \leq n$

It should also be clear that it requires the repeated calculation of some sub-expressions

e.g. LCS(abcd, pqr)

= longer(LCS(abc, pqr), LCS(abcd, pq))

= longer(longer(LCS(ab, pqr), LCS(abc, pq)),

Dynamic Programming

These are the two essential features which tell us that dynamic programming can be applied

- Recursive sub-structure
- Overlapping sub-problems

The basic principle is known as *memoisation* and is very simple:

- When we evaluate an application, remember the result so that we don't have to evaluate it again

We maintain a table of applications that have been evaluated. For every new application, we check first to see if we have done it previously

- If we have, just look up the result
- If we haven't, do the evaluation and store the result

Dynamic programming applies this principle in a slightly cleverer way

- We know what applications have to be evaluated, so we order them in such a way that repetition is avoided and no checking is needed

Recall: Fibaonacci numbers

The recursive program

```
fib(k)  = fib(k-1) + fib(k-2), if k > 1
        = 1, if k <= 1
```

Recursive sub-structure and
overlapping sub-problems

The “dynamic programming” version

```
fib(k)              = fib’(k, 1, 1)
fib’(k, x, y) = x, if k == 0
              = fib’(k-1, y, x+y), if k > 0
```

Checking to see if an application has been evaluated previously is avoided, because we have ordered the evaluation so that at every point we know we have already done what’s needed

With LCS, the information that needs to be stored is more complicated…

Dynamic Programming for LCS

- Given X of length *m* and Y of length *n*, we need to build a table of applications LCS(X_i, Yj),
  for all 0 ≤ i ≤ *m* and 0 ≤ j ≤ *n*
- The (i, j) entry holds two pieces of information:
  - the length of LCS(X_i, Yj)
  - an arrow denoting the rule used for that entry
- Consider X = 01101001 and Y = 110110
- We know all of the boundary cases
  - Whenever either string is empty, the LCS is empty
  - So the initial table looks like this
  - The yellow entry will hold the info for LCS(01101, 110)

X \ Y	—	1	1	0	1	1	0
—	0	0	0	0	0	0	0
0	0						
1	0						
1	0						
0	0						
1	0						
0	0						
0	0						
1	0						

Populating the table

- Each entry depends on three other entries:
  - The one to the left of it
  - The one above it
  - The one diagonally above-left of it
- Therefore we can fill in the table one row at a time, working down the rows and going right along each row
  - (Clearly we could do it column-wise instead)
  - For each entry, use the LCS recurrence relation

X \ Y	—	1	1	0	1	1	0
—	0	0	0	0	0	0	0
0	0	↑ 0	↑ 0	\ 1	← 1	← 1	\ 1
1	0						
1	0						
0	0						
1	0						
0	0						
0	0						
1	0						

$$
\begin{aligned}
\mathtt{len}(X_i, Y_j) &= 0, \text{ if } ij = 0 \\
&= \mathtt{len}(X_{i-1}, Y_{j-1}) + 1, \text{ if } x_i = y_j \\
&= \mathtt{len}(X_{i-1}, Y_j), \text{ if } x_i \neq y_j \ \&\ \mathtt{len}(X_{i-1}, Y_j) \geq \mathtt{len}(X_i, Y_{j-1}) \\
&= \mathtt{len}(X_i, Y_{j-1}), \text{ if } x_i \neq y_j \ \&\ \mathtt{len}(X_{i-1}, Y_j) \leq \mathtt{len}(X_i, Y_{j-1})
\end{aligned}
$$

The final table

- The final table looks like this
  - When different rules would give the same number, we can use any of them
- The length of LCS(01101001, 110110) is given by
  the last (bottom-right) entry
- To extract the LCS, follow the arrows up to the top-left
  - Wherever we encounter a \, add that character
  - Thus the LCS is 11010, denoted by yellow entries
- Note that if we had chosen different arrows (when we could), we may have gotten a different LCS
  - e.g. if (8,6) was "← 5", what would the LCS be?
- The complexity now is O(*mn*)!

X \ Y	—	1	1	0	1	1	0
—	0	0	0	0	0	0	0
0	0	↑ 0	↑ 0	\ 1	← 1	← 1	\ 1
1	0	**\ 1**	\ 1	↑ 1	\ 2	\ 2	← 2
1	0	\ 1	**\ 2**	← 2	\ 2	\ 3	← 3
0	0	↑ 1	↑ 2	**\ 3**	← 3	↑ 3	\ 4
1	0	\ 1	\ 2	↑ 3	\ 4	**\ 4**	↑ 4
0	0	↑ 1	↑ 2	\ 3	↑ 4	↑ 4	\ 5
0	0	↑ 1	↑ 2	\ 3	↑ 4	↑ 4	**\ 5**
1	0	\ 1	\ 2	↑ 3	\ 4	\ 5	↑ 5

Next time: Optimisation!