

18.417 Introduction to Computational Molecular Biology

Lecture 10: October 12, 2004

Lecturer: Ross Lippert

Scribe: Lele Yu

Editor: Mark Halsey

Exact Matching: Hash-tables and Automata

While edit distances and dynamic programming will give a solution to exact matches, they take a long time to return an answer. If n is the length of the text we are searching and m is the length of the string we are searching for, the obvious method takes $O(nm)$. The point of this lecture is to find an exact match in better than $O(nm)$ time.

History

The history of the study of exact matching has been longer than everything else we've been studying. More people than just biologists care about the subject. Examples of where it is used are:

- the world-wide web
- linguistic texts
- biologists

The result of so much interest in the subject is that we've gotten better results.

Notation

To formally define the problem, we use the following notation. The inputs to our exact matching algorithm are the following:

- q = query ($|q| = m$)
- T = text ($|T| = n$)

- A = alphabet

Query is the string we're searching for, T is the entire text and A is the alphabet. The alphabet indicates the set of all values each character of the text can take.

The outputs of our algorithm are:

- all occurrences of q in T .

A variation on this problem can be replacing q with a set of q_1, q_2, \dots, q_k so that finding any one of the queries is fine.

Other variations include:

1. q_i given in advance. We can preprocess all the queries.
2. T given in advance. We can preprocess T .

Text Preprocessing

To find occurrences of q in T

Formally, this can be written as:

$q_i \in A^l$ where A is the alphabet.

$H[]: A^l \rightarrow \{\text{location in } T\}$.

In other words, preprocessing the text means that for all possible queries of a certain length l find all the locations where such queries may occur. Obviously, this gets very expensive very quickly. However, this might be worth it if you are dealing with some huge T but the number of possible queries is limited.

For the alphabet we're interested in ($\{A, T, G, C\}$) it is very possible for H to be an array of 4^{16} elements. This is very expensive.

Instead, let H be a hashtable:

For an $H[]$ array of $2v$, we have a function $h: 4^l = 2v$.

So that, $H[h(q)] = H[h(q)] \cup \text{location}$ and each q is a query

Now, it could be that $h(q_i) = h(q_j)$ where q_i is not equal to q_j . When this is the case, a collision occurs. To do with that, let $H[]$ be a bunch of pointers.

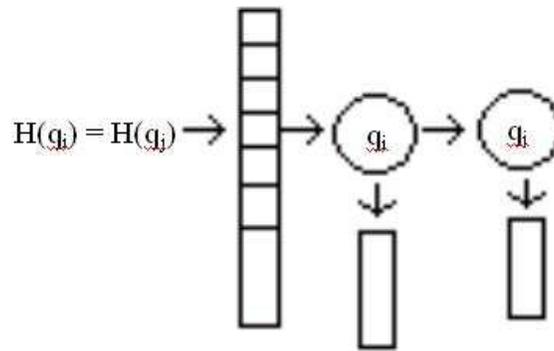


Figure 10.1: A Hashtable collision.

One can argue that the hashtable supports one pattern query in time proportional to query length.

Time = $O(|q|)$ (have to examine each character of your pattern once)

Space = $O(|T|)$

One drawback to hashing is that it is relatively inflexible because creating hash functions isn't easy. Ideally, you want random output in order to have the fewest collisions. However, it is difficult to come up with a function that produces random output. Hence, hash functions tend to be very specific. If you slightly change your query type (e.g. from L-mers to (L+1)-mers), the hash function might prove ineffective.

Query Preprocessing

Now let's try pre-processing the query (we just talked about pre-processing the text). Take an example of applying the slow algorithm:

$q = A C T A A C$

$T = A C A C T A A C$

In this case, you look at the first C and the second A twice, because you start all the way at the beginning. Traditionally, in order to find q in T, you start at the first A and continue to C. At the second A, the test fails and you start over at the first C. The test fails automatically. Finally you start at the second A and succeed. However, optimally, you shouldn't have to look at any character of the text more than once.

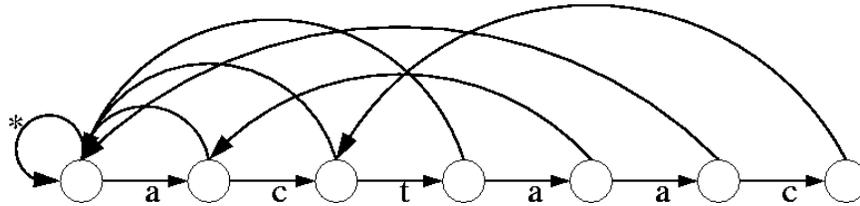


Figure 10.2: A FSM.

It is possible to look at every element the text only once because every letter in the text is absorbed by a FSM transition. In other words, scan time for T is $O(|T|)$. We can see this through amortized analysis.

The FSM fits into a small table:

i:	0	1	2	3	4	5	6
fail[i]:	-1	0	0	0	1	0	2
S[i]:	a	c	t	a	a	c	\$

Figure 10.3: A FSM table.

$$\text{fail}[i] = \max f : f < i, S[0:f-1] == S[i-f:i-1] \text{ and } S[f] \neq S[i]$$

$S[0:f-1]$ gives the longest prefix of S

$S[i-1]$ gives what's being matched now

$S[i-f:i-1]$ matches a suffix of the string we've matched so far

The code follows:

```

kmp-match(T,S)
  S= S+ '*'
  kmp-process(S,F)
  j= 0
  for 0<= i < length(T):
    j = kmp-next(T[i], j, S, F)
    if S[j] == '$':
      report matching ending at i
kmp-next(c, j, S, F)
  if j==-1 or S[j] == c:
    return j+1
  return kmp-next(c, F[j], S, F)
kmp-process(S, F)
  F[0] = -1
  f= 0
  for i in range( 1, length(S)):
    //loop inv: f = max f: f < i, S[0:f-1] == S[i-f:i-1]
    F[i] = f
    if S[F[i]] = S[i] and F[F[i]] != -1
      then F[i] = F[F[i]]
  f = kmp-next(S[i], f, S, F)

```

The method `kmp-process(S,F)` computes the failure array. It takes $O(|q|)$ time since there is only one loop and it iterates through the length of S (the string we're matching). The time to match is $O(|T|)$ since the loop in `kmp-match` iterates through all elements of the text T . Hence, the processing time is $O(|S|)$ and the matching time is $|T|$.

A common variation would be to build a $|A||S|$ lookup table for `kmp-next` for constant time improvement. This will avoid the recursive calls in `kmp-next`.

Multiple Queries

Now we will discuss multiple patterns.

Aho-Corasick

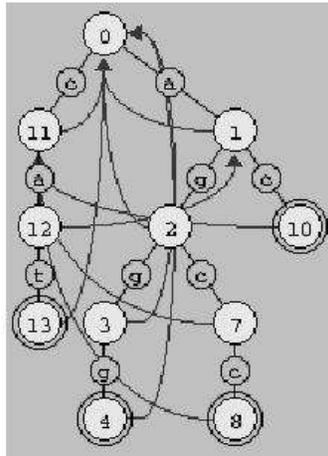


Figure 10.4: A multiple pattern FSM.

Implementation; you can again fit the FSM into one table. The first line in the table encodes the fact that you can branch to siblings.

The FSM fits into a small table:

i:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
fail[i]:	13	11	7	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
fail[i]:	13	11	7	0	0	-1	-1	0	14	14	-1	0	14	-1	0	1	0
S[i]:	a	g	g	g	\$	a	g	c	c	\$	a	c	\$	c	a	t	\$

Figure 10.5: A multiple pattern FSM table.

The failure row explains where to go if you failed to match. A failure moves you "up" in the BFS tree.

The Code:

```

ac-match(T, P1, P2, ...)
  S = P1 + '*' + ... + PN + '*'
  ac-process(S, F)
  then same as kmp-match
ac-next(c, j, S, F)
  same as kmp-next
ac-process(i, j, S, F)
  i = 0, j = 0
  while i < length(S):
    if S[i] == S[j]:
      if S[i] == '*':
        j = -1
      j = j+1
      i = i+1
    else
      if F[j] == -1:
        F[j] = i
      j = F[j]
  failures(0, -1, S, F)
failures(i, f, S, F)
  if F[i] == -1:
    F[i] = f
  else
    failures(F[i], f, S, F)
  if S[i] != '*':
    failures(i+1, ac-next(S[i], f, S, F), S, F)

```

This code is completely the same as kmp-match except for the ac-process method. The difference in the ac-process function is that:

1. failures do not occur immediately because they could still be branched off to siblings. Only after a series of soft failures do we have a hard failure.
2. The code incorporates a linked list.

In this case, the pre-processing takes $O(|A||Q|)$ (where Q is the sum of the lengths of the q_i) despite the fact that it looks quadratic. Running time of the the match process is then $O(|A||T|)$

We can also consider building a lookup table for these transitions. This will take $O(|A||Q|)$ space but then running time is $O(|T|)$. For next time: Suffix Trees.