Bioinformatics

Antonio Vivace, revision `e68`

Introductory Concepts

Naive method

Given a pattern P of size n, a text T of size m and an alphabet Σ.

Find every occurency of P in T, every i for which `T[i]...T[i+m-1] = P`.

Performance is $\mathcal{O}$(nm) with lower bound $\mathcal{O}$(n + m).

Preprocessing approach

Definition Given a string S, its size m and a position $i > 1$, let $Z_i(S)$ be the *length* of the longest substring of S that *starts* at i and matches a prefix of S.

Definition For any position $i > 1$ where $Z_i > 0$, the *Z-box* at i is defined as the interval starting at i and ending at position i + Z_i - 1.

Definition For every i > 1, r_i is the right-most endpoint of the Z-boxes that begin at or before position i. Another way to state this is: r_i is the largest value of *j + Z_j - 1* over all *1 < j ≤ i* such that $Z_j > 0$.

Theorem All the $Z_i(S)$ values are computed in $\mathcal{O}$(|S|) time (see *fundamental preprocessing in linear time* for algorithm and proof).

The simplest linear-time exact matching algorithm

Let S = P$T be the string consisting of P of length m followed by the symbol $ followed by T of length n, where $ is a character not appearing in neither P or T and n ≤ m.

Compute $Z_i(S)$ for i from 2 to n + m + 1. $ does not appear in P or T so Z ≤ n for every $i > 1$.

Any value of i > n + 1 such that $Z_i(S) = n$ identifies an occurence of P in T starting at position i - (n + 1) fo T. Conversely, if P occurs in T starting at position j of T, then $Z_{(n+1)} + j$=n.

This approach identifies all occurences of P in T in $\mathcal{O}$(m) time since all the Z_i(S) values can be computed in $\mathcal{O}$(n + m) = $\mathcal{O}$(m) time.

In terms of space, it uses only $\mathcal{O}$(n) space (in addition to the one needed for pattern and text) *independent* of the size of the alphabet (this is an *alphabet-independent* linear-time method).

Shift-And

Dömölki / Baeza-Yates, Gonnet.

Given a pattern P of size n, a text T of size m and the computer word ω.

Definition Let M be an $n * m + 1$ *binary* valued array, with index i running from 1 to n and index j running from 1 to m. Entry *M(i,j)* is 1 if the first i characters of P exactly match the i characters of T ending at character j. Otherwise the entry is 0.

Computing the last row of M solves the exact matching problem.

For the algorithm to compute M it first constructs an n-length binary vector U(x) for each character x of the alphabet where U(x) is set to 1 for the positions in P where character x appears. If P = *abacdeab* then U(a) = 10100010.

Definition Define *Bit-Shift(j-1)* as the vector derived by shifting the vector for column j-1 down by one position and setting the first to 1. The previous bit in position n disappears.

Building array M

Array M is constructed column by column as follows:

- Column 1 of M is initialized to all zero entries but the first to 1 if $T(1) \neq P(1)$, otherwise the first is zero, too.
- Column j, with j > 1, is C(j) = *Bit-Shift(j-1)* **AND** *U(T(j))*.
- `C[j] = ((C[ j - 1] >> 1) | (1 << ( w - 1))) AND U[T[ j]]).`

Performance

$\mathcal{O}$(n) if m ≤ ω, otherwise $\mathcal{O}$(nm).

Pay attention to the case ω< m <2ω.

Karp-Rabin

Shift-And assumes that we can efficiently shift a vector of bits, and increment an integer by one (the generalized method). If we treat a bit vector as an integer number then a lef shift by one results in doubling the number. So it is not much of an extension to assume that we can also efficiently multiply an integer by two. With that added primitive operation we can turn the exact match problem into an arithmetic

Definition For a text string T, let T_r^n denote the n-length substring of T starting at character r. Usually, n is known by context, leaving us with T_r.

Definition For the binary pattern P, let

$$H(P) = \sum_{i=1}^{i=n} 2^{n-i} P(i)$$

Similarly, let

$$H(T_r) = \sum_{i=1}^{i=n} 2^{n-i} T(r+i-1)$$

Theorem There's an occurence of P starting at position r of T if and only if $H(P) = H(T_r)$.

This is an immediate conseguence of the fact that every integer can be written in a unique way as the som of positive powers of two.

The **problem** is now that the required powers of two used in the definition of H(P) and $H(T_r)$ grow too large rapidly: they violates the unit-time *random access machine* (RAM) model which requires that allowed numbers must be represented in O[log(n+m)] bits but 2^n requires n bits. Even worse, when the alphabet is not binary, but has t characters, then numbers as large as t^n are needed.

Fingerprints

The general idea is that, instead of workng with numbers as large as H(P) and $H(T_r)$, we will work with those numbers *reduced modulo*. The arithmetic will then be done on numbers requiring smaller numbers of bits (and so efficient). The attractive feature of this method is a proof that the probability of error can be small if p is chosen randomly in a certain range.

Definition For a positive integer p, $H_p(P) = H(P)$ mod p ($H_p(P) is the remainder of H(P)/p). Similarly, $H_p(T_r) = H(T_r)$ mod p. $H_p(P)$ and $H_p(T_r)$ are the *fingerprints* of P and T_r.

Every fingerprint remains in the range 0 to p-1, not violating the RAM model anymore. The same problem appears in computing H(P) and $H(T_r)$, before they can be reduced modulo p: the following generalization of Horner's rule holds:

Lemma No number exceeds 2p during the computation of $H_p(p)$.

So, not only the number of multiplications and additions required is linear, but the intermediate numbers are always kept small.

False matches

Possible errors are false matches(the occurence is not true) and false negatives(the occurence is not found).

The error probability is **P[#FM $\geq$ 1]$\leq$ O(nm/I)** if the prime p is chosen among all the primes $\leq$ I.

Where I is:

- I = $n^2m \rightarrow$ P[#FM $\geq$ 1]$\leq 2.54/n$
- I = $nm^2 \rightarrow$ P[#FM $\geq$ 1]$\in O(1/m)$

We can lower the probability of errors choosing k random primes (independent and without repetitions) and change p after every false match.

Theorem If a new prime is randomly chosen after the detection of an error, then for any pattern and text the probability of t errors is at most $(2.53/m)^t$

Why fingerprints?

The Karp-Rabin fingerprint method runs in linear worst-case time, with a nonzero(though extremely small) chance of error. It can be thought of as a method that never makes an error and whose expected running time is linear which is not exactly good compared to the other method which runs in linear worst-case time. The reasons of studying this are:

- The method can be extended to other problems, such as two-dimensional pattern matching with odd pattern shapes.
- The method is accompanied by concrete proofs, establishing significant properties of the method's performance.
- The method is based on very different ideas than the other linear-time methods that guarantee no error.

Suffix Trees

A suffix tree is a data structure that exposes the internal structure of a string in a deeper way than does the *fondamental preprocessing*. Suffix trees provide a bridge between exact matching problems and *inexact* matching problems.

The preprocessing takes time proportional to the length of the text ($\mathcal{O}$(n)), but thereafter, the search for S must be done in time proportinal only to S, independent of the length of T.

Definition The suffix tree for the string S of length n is defined as a tree such that:

- The tree has exactly n leaves numbered from 1 to n
- Except for the root, every internal node has at least two children.
- Each edge is labeled with a non-empty substring of S.
- No two edges starting out of a node can have string-labels beginning with the same character.
- The string obtained by concatenating all the string-labels found on the path from the root to leaf i spells out suffix S[i..n], for i from 1 to n.

Since such a tree does not exist for all strings, S is padded with a terminal symbol not seen in the string (usually denoted \$).

Definition The *label of a path* from the root that ends at a node is the concatenation, in order, of the substrings labeling the edges of that path. The *path-label of a node* is the label of the path from the root of T to that node.

Definition For any node v in a suffix tree, the *string-depth* of v is the number of characters in v's label.

Definition A path that ends in the middle of an edge *(u, v)* splits the label on *(u, v)* at a designated point. Define the label of such a path as the label of u concatenated with the characters on edge *(u, v)* down to the designated split point.

Definition We will call *generalized suffix tree* a suffix tree for a set of strings. When constructing such a tree, each string should be padded with a unique out-of-alphabet marker symbol (or string) to ensure no suffix is a substring of another, guaranteeing each suffix is represented by a unique leaf node.

Exact match problem with suffix trees

Given a pattern P of length n and a text T of length m, find all occurences of P in T in $\mathcal{O}$(n + m) time. Suffix trees provide another approach solving this problem:

- Build a suffix tree T for text T in $\mathcal{O}$(m) time
- Match the characters of P in the unique parth in T until (1) P is exhausted or (2) no more matches are possibile.
  - In case 1, P does not appear anywhere in T.
  - In case 2, every leaf in the subtree below the point of the last match is numbered with a starting location of P in T, and every starting location of P in T numbers such a leaf.

If P fully matches some path in the tree, the algorithm can find all the starting positions of P in T traversing the subtree below the end of the matching path, collecting positions numbers written at the leaves.

ALl occurences of P in T can therefore be found in $\mathcal{O}$(n + m) time. Suffix tree approach spends $\mathcal{O}$(m) in preprocessing time and then $\mathcal{O}$(n + k) search time, where k is the number of occurences of P in T.

To collect k starting positions of P, traverse the subtree at the end of the matching path using any linear-time traversal(i.e. depth-first), and note the leaf numbers encountered. The time for the traversal is $\mathcal{O}$(k) because the number of leaves encountered is proportional to the number of edges traversed (since every internal node has at least 2 children).

If a **single occurence** is required, and the preprocessing is extended a bit, the search time can be reduced to $\mathcal{O}$(n) time. The idea is to write at each node one number (say, the smallest) of a leaf in its subtree (can be achieved in $\mathcal{O}$(m) in the preprocessing stage). In the search stage, the number written on the node at or below the end of the match gives one starting position of P in T.

APL4: Longest common substring of two strings

Note that substring $\neq$ subsequence.

An efficient and conceptually simple way to find a longest common substring is to build a *generalized suffix tree* for S_1 and S_2. Each leaf represents either a suffix from one of the two strings or a suffix that occurs in both the strngs. Mark each internal node with 1, 2 or both. The path-label of any internal node marked both 1 and 2 is a substirng common to S_1 and S_2, and the longest such string is the longes common substring. The algorithm has only to find the node with the grates string-depth that is marked both 1 and 2.

Construction of the suffix tree can be done in linear time, and the node markings and calculations of string-depth can be done by standard linear-time tree traversal methods.

Theorem The *longest common substring* of two strings can be found in linear time using a generalized suffix tree.

APL6: Common substrings of more than two strings

Suppose we have K strings whose lengths sum to n.

Definition For each k between 2 and K, we define *l(k)* to be the length of the *longest substring common to al teast k of the strings.*

$\mathcal{O}$(Kn) Solution

First, build a generalized suffix tree T for the K strings. Each leaf of the tree represents a suffix from on of the K strings and is marked with one of the K unique string identifiers, 1 to K, to indicate which string the suffix is from. Each of the K strings is given a distinct termination symbol, so that identical suffixes appearing in more than one string end at distict leaves in the generalized suffix tree. Hence, each leaf in T has only one string identifier.

Definition For every internal node v of T, define C(v) to be the number of *distinct* string indetifiers that appear at the leaves in the subtree of v.

Once C(v) numbers are known, and the string-depth of every node is known, the desired *l(k)* values can be easily accumulated with a linear-time traversal of the tree:

The traversal build a vector V where, for each value of k, ranging from 2 to K, V(k) holds the string-depth (and location, *if desired*) of the deepest node v encountered with C(v) = k.

Essentialy, *V(k)* reports the length of the longest string that occurs *exactly* k times.

So, V(k) $\leq$ l(k). To find *l(k)*:

- Scan V from largest to smallest index, writing into each position the maximum *V(k)* value seen.
- If V(k) < V(k+1) then V(K) = V(k+1).

The resulting *V(k)* vector holds the desired *l(k)* values

In linear time, it is easy to compute the number of leaves in v's subtree for each internal node i. Two leaves in the subtree may have the same identifier. This repetition of identifiers is what makes it hard to compute *C(v)* in $\mathcal{O}$(n) time.

Therefore, it uses $\mathcal{O}$(Kn) time to explicity compute which identifiers are found below any node:

For each internal node v, a K-length bit vector is created that has a 1 in bit i if there is a leaf with identifier i in the subtree of v. *C(v)* is then just the number of 1-bits in that vector.

The vector for v is obtained by **OR**ing the vectors of the children of v. For l children, this taked lK time. Over the entire tree, with $\mathcal{O}$(n) edges, the time needed for the entire table is $\mathcal{O}$(Kn).

$\mathcal{O}$(n) Solution

Let

- *C(v)* (for every node v) is the number of distinct leaf identifiers in the subtree of v.
- *l(k)* can be computed in O(n) once C(v) is known.
- *S(v)* is the total number of leaves in the subtree of v and can be computed in O(n).

S(b) $\geq$ *C(v)*

Introduce *U(v)* as *correcting factor*, it counts "duplicate" suffixes from the same string that occurs in v's subtree. Then

C(v) = *S(v)* - *U(v)*

Computing U(v)

Begin

1. Build a generalized suffix tree T for the K strings.
2. Number the leaves of T as they encountered in a depth-first traversal of T.
3. For each string id *i*, extract the ordered list L_i of leaves with id *i*.
4. For each node w in T set *h(w)* = *0*.
5. For each id *i* compute the *lca* (*lowest common ancestor*) of each consecutive pair of leaves in L_i, and increment *h(w)* by one each time that w = *lca*.
6. With a bottom-up traversal of T, compute, for each node v, *S(v)* and *U(v)* = $\sum[h(w):$ w is in the subtree of $v]$.
7. Set *C(v)* = *S(v)* - *U(v)* for each node v.
8. Accumulate the table of *l(k)* in the same way of the O(Kn) method.

End

APL13: Suffix Arrays - more space reduction

Assuming we have sufficient space to build a suffix tree for T but it cannot be kept intact to be used in the (many) subsequent searches for patterns in T. Instead, we convert the suffix tree to the more efficient suffix array. A suffix array for T can be obtain from its suffix tree by performing a *lexical* depth-first traversal of the tree. Once the suffix array is built, the suffix tree is discarded.

Definition Define an edge *(v, u)* to be *lexically less* than an edge *(v, w)* if and only if the first character on *(v, u)* is lexically less than the first character on *v, w)*. \$ is lexically less than any other character. Since there are no two edges out of v have labels beginning with the same characters, there is a strict lexical ordering of the edges out of v.

Suffix array *Pos* is just the ordered list of suffix numbers encountered at the leaves of T during the lexical depth-first search.

Theorem The suffix array *Pos* for a string T of length m can be constructed in $\mathcal{O}$(m) time.

Example

```
T = tartar, Pos=(5, 2, 6, 3, 4, 1)
```

```
T   t a r t a r
    1 2 3 4 5 6
5           a r
2     a r t a r
6   t a r t a r
3       r t a r
4         t a r
1             r
```

LCP Array

LCP is an auxiliary data structure to the suffix-array. It stores the lengths of the longest common prefixes between all pairs of consecutive suffixes in a sorted suffix array.

Pattern matching with Suffix-Array

Locating every occurence of a substring P within S is equivalent to fining every suffix that begins with P. Recall that suffixes are lexicographically ordered:

- Binary Search (1) to find the starting position of the interval;
- Binary Search (2) to find the ending position of the interval.

Alignment

Definition A string over the alphabet I, D, R, M that describes a transformation of one string to another is called an edit transcript, or transcript for short, of the two strings.

Definition The edit distance between two strings is defined as the minimum number of edit operations - insertions, deletions, and substitutions - needed to transform the first string into the second. For emphasis, note that matches are not counted.

The **edit distance** problem is to compute the edit distance between two given strings, along with an optimal edit transcript that describes the transformation.

An edit transcript is a way to represent a particular transformation of one string to another. An alternate (and often preferred) way is by displaying an explicit **alignment** of the two strings.

Definition A (global) alignment of two strings Si and S2 is obtained by first inserting chosen spaces (or dashes), either into or at the ends of Si and S2, and then placing the two resulting strings one above the other so that every character or space in either string is opposite a unique character or a unique space in the other string.

Table of Contents

- Suffix array: pattern matching
- Ricerca della sottostringa comune più lunga di k stringhe (Gusfield, 7.4, 9.7)
- Allineamento di globale di 2 sequenze. Needleman-Wunsch. Allineamento globale come caso generale di distanza di edit e LCS. (Gusfield, 11.6, 11.6.[1-2], 14.1, 14.[5-6])
- Allineamento con gap
- Allineamento locale di 2 sequenze. Smith-Waterman
- Sequenziamento e grafi di de Brujin
- Applicazioni di suffix array
- Filogenesi basata su caratteri

Credits

- *Algorithms on Strings, Trees, and Sequences* - D. Gusfield