Genomics Mid1

tyang27

September 2019

1 Notation

Assume ranges follow normal CS conventions.

2 Intro

2.1 Sequencing technologies

- Sanger
- DNA Microarrays
- 2nd Generation (our focus)
- 3rd Generation/Single Molecule

2.1.1 2nd Generation Sequencing

- Cut DNA into snippets.
- Deposit snippets onto slide (Massively parallel).
- Create cluster of clones.
- Use DNAP to add complement with special bases with terminator (keep it in sync) and radioactive label (to take snapshots).
- Remove terminator cap.
- Go back to DNAP step.

Some sequences might get ahead of schedule bc the terminator fell off, and as the reads go on, these mistakes accumulate. Let p be the probability incorrect. 1 in $10^{Q/10}$ incorrect. For ascii encoding, need to add 33 for it to be a visible character.

$$Q = -10 \log p$$

$$Phred = chr(round(Q) + 33)$$

3 Alignment problem

Match reads to a reference.

3.1 Online versus Offline

Offline iff processes T. Otherwise, online.

Naive exact	online
Boyer Moore	online
Indexing	offline

3.2 Naive Exact

Let $n = \|P\|$, $m = \|T\|$. Then there are $\|T\| - \|P\| + 1 = m - n + 1$ alignments. For each alignment, at most n comparisons.

$$O(n(m - n + 1))$$

```
function NAIVE(P, T)
    Let matches be an array
    for i = 0 ... ||T|| - ||P|| + 1 do
        match = True;
        for j = 0 ... ||P|| do
            if T[i + j] ≠ P[j] then
                matches = False; break;
            end if
        end for
        if match then
            matches.append(i);
        end if
    end for
end function
```

3.3 Boyer Moore

Preprocess pattern (amortized), so that we can skip character comparisons based on the following insights:

- Compare characters backwards (allows us to skip more, it gives insight about location).
- Bad character rule - Let mm represent the index of mismatch. Say that we slide the pattern to a match such that the $T[mm] \neq P[i]$ where $i < mm_P$. Contradiction by definition of a match. Therefore, we can at least slide it until either a character in the pattern matches $T[mm]$, or past the mismatch if $T[mm]$ does not occur in the pattern.
- Good suffix rule - Intuition for this is that patterns might be repetitive, so if we match some part of the suffix and then hit a mismatch, that might actually be the middle of the pattern where the same pattern occurs again. Let $t = T[mm + 1 :]$ be the suffix correctly matched. If t occurs to the left, we can align those matches, line up so that t matches. If a prefix of P matches a suffix of t, we can also align those matches.
- Good suffix rule and bad character give lower bounds for how much we can skip, so take the max.

3.3.1 Disadvantages

- Worse case when P matches many times, since we have to fully check each one $O(\|P\|)$.
- Worse case when alphabet is small, since subpattern shows up more common, causing more comparisons and shorter skips. E.g. if z shows up really infrequently, so if our long pattern does not contain z's, we can skip a lot.

3.4 Multimap indexing

Sliding window of k-mers across T with length k, say $t = T[i : i + k]$. Map t to the indices where offsets of t occur.

3.4.1 Exact

Just look up pattern/subpatterns in multimap.

3.4.2 Approximation

Hamming distance - Hamming distance of 2 strings, X, Y. Assume that $\|X\| = \|Y\|$. Minimum number of subtitutions needed to turn one into another.

$$\sum_{i=0...\|X\|} \mathbb{I}_{X[i] \neq Y[i]}$$

Pigeonhole - If k mismatches in text T, and create $k+1$ partitions, at least one partition must have no edits.

- We can easily combine pigeonhole with any exact matching algorithm, and hamming distance.
- However, grows with number of edits (large k yields small partitions, $k+1$ exact matching problems).

3.5 Suffix Tries - $O(m^2)$ construction, $O(m^2)$ space

Merges parts of like suffixes. Can use to create indexing multimap. Can also use to contain all suffixes of a text T. Let \$ be a new character with lexicographical order less than letters. Terminating nodes iff leaves.

3.5.1 Disadvantages

Worst case, we have lots of repetition, so we have $O(m^2/2) = O(m^2)/2$.
Not sure if this recurrence is right, but n represents the number of nodes added, and $T(n-1)$ is the next suffix.

$$T(n) = n + T(n-1)$$

3.6 Suffix Trees - $O(m^2)$ construction, $O(m)$ space

- Make it smaller by collapsing nonbranching paths into a single label. Internal nodes at least 1 child. $O(m)$ nodes, since m leaves and $\leq m-1$ internal, but labels are still $O(m)$
- Store T, and rather than copying substrings over and over again, just store $(offset, length)$. We can store offsets in leaves.
- Node depth is the number of edges from root to node. Label length is sum of labels.

3.6.1 Disadvantages

Large constants, since we are putting it into a tree.

3.7 Operations

- Substring match - Every substring is a prefix of some suffix. Go down the tree. True if don't fall off the tree, else false.
- Exact suffix match - A special case of substring match. Go down the tree. True if don't fall off the tree and reach a terminating node, else false.
- Total substring matches - DFS to count terminating nodes after substring match. If every substring is a prefix of some suffix, and there are multiple terminating nodes, then this substring must be a prefix of multiple suffixes, so it must appear multiple times in the string.
- Longest repeated substring - Deepest node with more than one child. Similar reasoning as above. WLOG, shorter one must be a substring of the longer (Let there be a block string $AXBXC$, where X is repeated substring. XBXC\$ and XC\$ must be in the suffix tree bc they are suffixes. If the shorter is not in the longer, then we must have XB\$ in the tree, but this is not necessarily true).
- Longest common substring - Create string $X\#Y\$$ where $\#$ is a new terminal symbol, put into suffix tree. Deepest node (label depth) that belongs to both X and Y.

3.8 Suffix Array

Store offsets of all suffixes lexicographically (don't need to store suffixes, since we can easily reconstruct this from the text, but in diagrams just to make life easy). This lowers constants.

3.8.1 Disadvantages

For searching, need to use binary search.

3.9 Burrows-Wheeler Transform

All rotations of a string, sort (BW Matrix), take the last column. Right context refers to everything left of the character in the last column, this might seem weird, but when rotated, the last character is left of the right context.

- Compression - strings sorted by right-context, so patterns may occur that cause repetition, allowing us to replace long repeated substrings of one character with a character and a number.
- BWM is equivalent to the strings in suffix arrays.
- Reversible using LF Mapping - We can impose rankings on strings, e.g. T ranking. B ranking (occurences up and until that character) shows us that rank order is same between Last and First columns.

Using the LF Mapping/B index, we want to make it easy to search for ranges that represent suffix nodes, s.t. these ranges represent the previous character of the suffix. If we know the rank of the character we are searching for, we can easily jump to the index in the lexicographically sorted F column. Note, remember to add 1 for the $ row.

3.10 FM Index

Rather than doing binary search, we can precalculate fast B-ranks. This allows us to query ranges in $O(1 * 2) = O(1)$. Naive space is $O(m * \|\Sigma\|)$, but we can improve this by saving checkpoints s.t. they are separated by a constant distance. Then, our lookups are $O(1)$. Still $O(m)$ space, but $O(m/chkpts * \|\Sigma\|)$.

Sample.

3.11 Operations

-