Define:
n = len(reference sequence)
m = len(query sequence)

Sapling

Learned index on suffix array.

Input
- fixed length string

Output
- predicted residual
- then derive the index in the reference sequence with an error bound E

Model:
- Fully connected neural network
- piecewise linear architecture

Basic idea:
Instead of traditional binary search starting with the whole reference sequence, it predicts the index p(x) of input sequence x with an error bound E. E means it's guaranteed that SA[p(x) - E, p(x) + E] contains suffixes starting with x.

Benefits:
The traditional time complexity of a query is O(mlogn). The learned version reduces logn term. The difference between logn is actually small. The benefits seem to be more from cache misses. Since traditionally it does binary search on a large interval so there will be a lot of cache misses.

Problem:
It's really surprising that they can guarantee the error bound E, because in ML there is no guarantee of the error of an arbitrary input. In this scenario it's possible given a particular input, E is as large as the length of the reference. However, they claimed so because they performed query on all k-mers, i.e., all possible input to the model, and obtain the global maximum E. I don't think this idea is practical because for large k and long reference, it's impractical to train the model on all k-mers. Also, there is no guarantee that E will be small for any reference sequence.

Lisa

Learned index on modified FM-index

Input
- fixed length string

Output
- the position in their IPBWT with a range where the key is guaranteed to occur

Model:
- RMI (recursive model index)
- …

Basic idea:
First modify FM-index by adding IPBWT. The basic idea of IPBWT is to process multiple characters at a time. Originally in each backward iteration, it prepends one new character. In the modified version it prepends k characters (maybe not true).

The challenge here is to quickly locate the prepened k characters in the first row. Originally this is easy because there are only 4 characters. It's easy to keep a table of the number of occurrences of characters. But in the modified version it needs to keep 4^k entries, which is too big.

So the idea is, since IPBWT is sorted, we can do binary search in IPBWT to find the next one. But in total it takes O(m/k*logn). And here is where the model comes in. Similar to Sapling it predicts a position with guaranteed interval.

Benefits:
O(m/k*log(predicted interval length)). Certainly speed up by a factor of k, but also depend on how big is log(predicted interval length). I didn't see where average time of prediction and the average size of predicted interval length

Problem:
Again I'm not sure how it obtains a guaranteed error bound.

BWA-MEME

Learned index on suffix array.

Input
- variable-length string

Output
- index in the reference sequence with an error bound E

Model:
- P-RMI (partial recursive model index)

Basic idea:
Sapling and Lisa only work on fixed-length input. This work proposed a tokenization to encode a variable length string into a numerical key. Moreover, the traditional RMI suffers from data imbalance. This work introduces P-RMI to mitigate this issue.

Benefits:
variable-length input. Better model for prediction

Problem:
I'm not convinced by tokenization. It sets the input length of the model to be 32. For input longer than 32 it simply takes the first 32 characters and ignores the rest. Although in the "last mile" binary search, it uses the full input, the tokenization seems too simple for the model.

It answers why it can guarantee an error bound in Supplementary data

memory is huge

use fmd index and learned index on the last mile search

A representation of a compressed de Bruijn graph for pan-genome analysis that enables search

Notations:
- **Σ**: alphabet set including $, #
- σ: alphabet size
- S: In pan-genome analysis, S is the concatenation of multiple genomic sequences, separated by #. S ends with $
- S[i]: character at i in S
- S[i..j]: substring of S
- Si: ith suffix S[i..n] of S
- SA: suffix array (sorted)
- SA[i]: starting index in S of the ith smallest suffix.

- $S_{SA[i]}$: the actual suffix corresponding to the index of SA
- BWM: Burrows-Wheeler Matrix
- BWT: Last column in BWM
- **ω**: substring of S
- **ω**-interval: the suffix array interval where **ω** is a prefix of all suffix in this interval.
- c: a character in **Σ**
- c**ω**: string by prepending c to **ω**
- c**ω**-interval: the suffix array interval where c**ω** is a prefix of all suffix in this interval. Can be found by one step of backwardSearch(c, [lb..rb]), given c and **ω**-interval [lb..rb]
- LF-mapping: i is an index of SA. LF(i) returns j, which is an index of SA such that the suffix corresponding to j is the previous one of the suffix corresponding to i in S. Another way to say this: is SA[i] = k, then LF(i) = j such that SA[j] = k - 1
- **Ψ**: inverse of LF-mapping. So it returns the index next suffix of i.
- LCP-array: LCP[i] is ihe length of the longest common prefix between i and i-1 suffix in SA
- left/right-maximal repeat: for a repeated substring, if the left context of all occurrences are the same character, then it's not left-maximal. Otherwise it is left maximal.
- Bl: Indicates left-maximal k-mer. It is 1 for cw where the first k chars of w is a left-maximal k-mer but not right-maximal k-mer. For same c's, it is 1 only for the last one.
- Br: Indicates right-maximal k-mer. It is 1 at the start and the end of w-interval where the first k chars of w is a right-maximal k-mer.
- C: C[c] is the number chars strictly smaller than c in BWT. It's precomputed, and can change in Algorithm1
- getIntervals([lb..rb]): returns the list [(c, [i..j]) | cω is a substring of S and [i..j] is the cω-interval]
- G: the compacted de Bruijn Graph.
- N: the number of nodes in G
- d: number of concatenated strings. $S = S^1\#S^2\#...\#S^d\$$
- implicit G[id]: a node in G. id ∈ {1, ..., N}. G[id] jas the fpr, (len, lb, size, suffix_lb)
    - len: the length of the string w of that node
    - lb: start point of the w-interval
    - size: the size of the w-interval
    - suffix_lb: start point of the interval of last k chars of w. These two interval have the same size
    - G is built by first adding right-maximal k-mers to
- explicit G[id]
    - len: the length of the string w of that node

- posList: the positions (sorted) where w occurs in the original S. i.e., the values in SA[i..j], where [i..j] is the w-interval
- adjList: successors of the node in dBG

Algorithm 1:

Purpose:
Compute right-maximal and left-maximal and store info in Br, Bl

Basic idea:
Go through BWT one by one. If the current one has a common prefix longer than k as the previous one, note down this is the start of a repetition by not changing lb anymore. If the common prefix length is exactly k, note down this is the start of a repeated k-mer.

Then when it reaches the last position of the repetition (LCP no longer greater than k), check if the repetition is length k by checking kIdex > lb. If not then it means the k-mer can be extended so it's not a right-maximal. If so, set Br[lb] to 1 to track the starting point of this repetition, and set Br[i-1] to track the ending point of this repetition.

For left-maximal, track the last different char. If at the end of a repetition, lastdiff > lb, it means there are more than 1 different left context (except # $), meaning it is a right maximal. Set the last occurrence of its left context chars to have Bl = 1.

Finally, since if a repetition is already right-maximal, we don't care about if it is a left-maximal. Go through Br and Bl to set Bl = 0 where Br = 1

Additionally, at the end of a right maximal-repetition, add the w-interval of this repetition to G, which will be used in algorithm 2 to construct implicit dBG.

Algorithm 2:

Purpose:
Construct implicit dBG G.

Basic idea:
Start from right-maximals added to G in Algorithm 1. Also add stop nodes for all d sequences (line 7-11). Then go through each added node in G and try to extend them.

For each node, check one more char to its left (line 18). If there is only 1 char, then it means this node can be extended by 1 character to the left. If there is more than 1 char, then it means this is a left-maximal repetition, so a split happens according to fig 5. New nodes are added for each new char.

Algorithm 4:

Purpose:
Search for a pattern P in dBG. Return a list of node that the pattern covers

Basic idea:
First locate the last k chars in P in FM index. If the k-mer also happens to be at the end of w of a node, then we can directly locate the node. Otherwise we do reverse backward search to go backward 1 char by 1 char to find the last k-mer in w. Once we find it, we locate the last node P covers.

Then we go forward (second while loop) to find other nodes that P covers

office hour:

- on the representation of dbg
- given a pattern, tell me the unitig and the offset in that unitig
- pufflefish only works for fixed length k because hash based
- sshash, dbg, puflefish unitig, dbgfm
- size of the index, distribution
- a lit bit bigger than fm, not as big as hash, and faster than fm
- global position -> unitig index
- store the end points of each unitig

Plan:

-

- presentation:
    - tell the class the idea and what we did
    - high-level result
- report
    - light-weight academic paper
    - intro, why interesting

- related work
- method
    - combining learned index with compacted dbg
- result
    - size
    - time
- 3-6 pages

FM Index

Components of the FM Index
First column (F): |**Σ**| integers
Last column (L): n characters
sampled SA: n * a integers
checkpoints for character ranks of last column: n * |**Σ**| * b integers

Query time: O(n + k), O(n * “checkpoint sample factor” + k * “SA sample factor”)
Construction time: O(n)