Algorithms in Genome Analysis, Spring 2023

Veli Mäkinen

Week 6

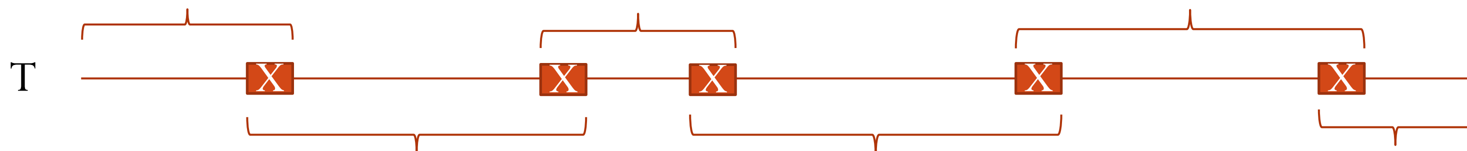
Prefix-free parsing and r-index

Motivation

- Consider a collection of d similar sequences
- One can build any BWT index on the collection to support fast read alignment, but the size of the index is d times the size of a single sequence
- If d-1 sequences differ from reference T[1..n] in s positions, one would wish to have an index that takes O(n+d+s) space
- Lempel-Ziv (LZ) compression achieves such bound, but index structures based on LZ are slow in practice
- BWT of such collection has long runs of identical characters, in fact, $r = O(z \log^2 n)$, where r is the number of runs and z is the length of the LZ parsing
- Can we build run-length (RL) encoded BWT directly and can we use it as an index?
- Yes, direct construction is possible e.g. using prefix-free parsing and r-index operates directly on the RL-encoded BWT

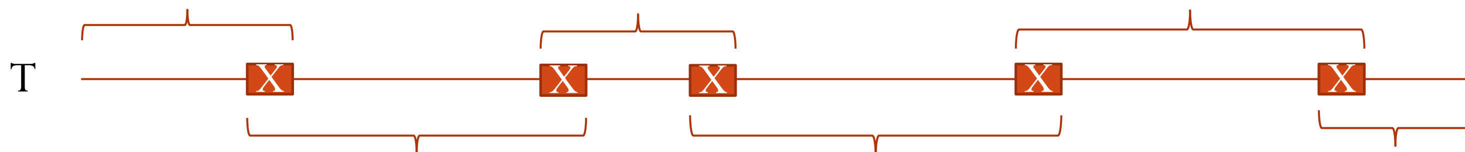
Prefix-free parsing 1/2

- Fix k and a hash-function h() on k-mers of a collection of sequences concatenated into a long string T[1..n].
- Let also h() be such that h(T[1..k])≠0
- Assume there is unique substring X of T with h(X)=0 (this is for simplicity of exposition, and can be relaxed)
- The first phrase of T is T[1..i], where T[i-k+1..i]=X is the first occurrence of X. The second phrase of T is T[i-k+1..j], where T[j-k+1..j]=X is the second occurrence of X, and so on until the second last phrase. The last phrase is T[p..n]#, where T[p..p+k-1]=X is the last occurrence of X



Prefix-free parsing 2/2

- Note that the phrases form a prefix-free set, hence the name
- Identical parts in the collection will have identical parsing
- We can collect the phrases into a set, considering occurrences of a phrase as one entity
- It turns out one can sort the suffixes of the set of phrases to infer the sorted order of suffixes of the collection
- Furthermore, in many cases a range of suffixes is inferred at a time so we can output directly a run of BWT (for details see the 2nd edition of the course book)



Example

k=2, h(si)=0

parsing

mississippimississippi# → 1,4,3,4,2

sort suffixes of the parsing

Phrases	Lex rank
missi	1
sissi	4
sippimissi	3
sippi#	2

Sort suffixes of the concatenation of phrases

Combine the information to get the final sorted order of suffixes of T

Where do we get the runs?

The lex order of suffixes close to the start of a phrase may be fully determined within the phrase

→ Phrase occurs x times → BWT run of length x

r-index

- Let T[1..n] be a collection of sequences concatenated
- Let r be the number of runs in the BWT of T. E.g. if BWT=TTTAACCCC#AATT, r=6.
- To turn our BWT indexes to use space sub-linear in n, we need to replace the rank and wavelet tree data structures, e.g., with balanced binary trees (BSTs).

Run-length rank data structure

- For backward search, we need to support $rank_c(L,i)$.
- We store the start of the runs as keys in a BST. As values we add the rank of each run.
  - Assume L=TTTAACCCC#AATT.
  - We add (key,value) pairs (1,1),(4,2),(6,3),(10,4), (11,5),(13,6) to BST
- We store the rank answers preceding each run for the character of each run
  - Pre-rank[1..r]=0,0,0,0,2,3
- We also build the wavelet tree of L', where L' is the sequence of characters of the runs.

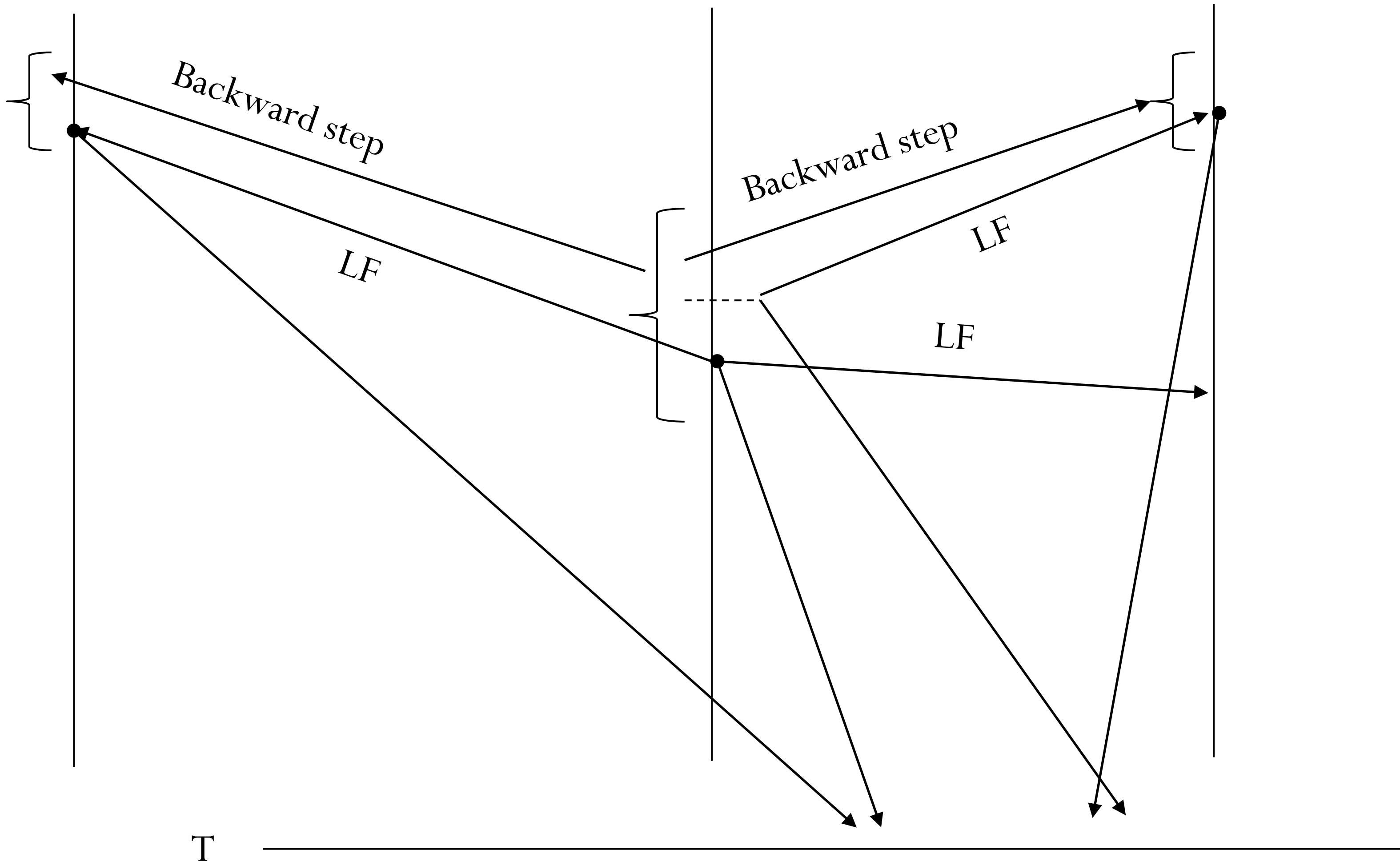
Run-length rank query

- Consider $rank_c(L,i)$.
- We search the largest key i' not larger than i in BST. Let it be associated with value p.
- If c=L'[p], $rank_c(L,i)$=pre-rank[p]+i-i'+1.
- If $c \neq$ L'[p], $rank_c(L,i)$=pre-rank[p']+$d^{p'}$, where p'=$select_c$(L',$rank_c$(L',p)) and $d^{p'}$ is the length of the p'-th run.
- Since rank takes O(log n) time, we can do backward search in O(m log n) time for a query of length m.

Locating the occurrences

- Our previous sampling scheme fails as in O(r) space each locate needs O(n/r) steps.
- It turns our that there is a different samping scheme taking O(r) space and allowing to locate each occurrence in O(log n) time.
  - The idea is to sample the beginning and end of runs.
  - During the backward search one maintains one sampled location in the interval (next slide).
  - In the end of the process, the neighboring occurrences can be revealed using a tunneling property of BWT (following slide).

Maintaining one occurrence



Retrieving neighboring occurrences