Nathaniel Evans

Christina Zheng & Shannon Mcweeney

BMI650

10/23/18

HW#4

**Part 1 (20pts):** Please answer the following questions for either a suffix array **OR** a burrows wheeler transform:

1. **Name of approach.**
   1. Burrows Wheeler
2. **What question/problem does this approach answer?**
   1. Efficient suffix string search to answer the questions and memory compression:
      1. Does the suffix exist?
      2. If so, what is the index/offset? I.e. where does the suffix start in the sequence?
3. **Describe/outline a naive/brute force/strawperson approach to the problem above.**
   1. To solve the suffix search problem, the brute force approach can be imagined as:

For **EVERY** match string given:

Generate all subsequence permutations of expected length (of given string to match) record subsequence indexes

For each subsequence

Compare subsequence to string to match

If none are found, suffix doesn’t exist

If one or more are found, return the index/offsets

Let n = sequence length, k = k-mer length (substring match length)

Time complexity O(n) ~ n-k

Memory complexity O(n) ~ (n-k)\*k

1. **Describe/outline the steps to the approach named in question 1 (Include assumptions, inputs, outputs)**
   1. **Create BW transform**

Add a terminal character to end of sequence

Create set of all permutations of terminal character index

Lexicographically sort permutations

Keep only the last column – Memory complexity O(n)

* 1. **To search BW transform for suffix ‘abc…’**

Create first column by sorting the last column lexicographically

Create B rank array for first and last cols, number each unique letter by order it appears (top to bottom)

In FC, for each occurrence of ‘a’ (first char) in FC

find the b-ranked matched occurrence of ‘a’ in in LC, then look at FC value in that row, this is the second value of our string, if it matches ‘b’ continue, else break.

Look for the b-ranked matched occurrence of our second value in the LC, look at FC value, this is our third char of the string, if it matches ‘c’ continue, else break.

… continue until

you reach a terminal character, return this value

For each matched substring, subtract length from sequence length, this is the substring offset.

If no matches are found, return None

Additional time complexity optimization at the detriment of increased memory requirements include precomputing suffix array to speed up indexing and creating a FM matrix (FC/LC b-rank array) to speed up rebuilding the BW matrix. Also, I can envision a method of searching for ANY substring in the sequence by checking for a match before reaching a terminal character. In this way, prefix, mid-string, or suffixes could be searched for. The offset determination mechanism would need to be altered. Memory compression can be employed by compressing the LC repeated characters.

**Describe at least 2 advantages this approach has over the naive/brute force approach.**

1. Decreased time complexity (after the precomputed BW transform) during search. Easy to search only suffixes that start with the desired character(s). Narrows the search space significantly.
2. Memory compression, by represented repeated characters as <char><num\_repetitions> the BW transform can be stored in less memory than the original string.
3. **Describe at least 1 limitation of this approach and a potential extension/solution for this limitation.**
   1. As the BW matrix is rebuilt (during search) the b-ranks for the F/L col have to be calculated and the offset has to be calculated after a string is found. To remedy this, the FM and suffix array (minus the suffix strings) can be pre-calculated at the detriment of memory space. To create a happy medium, *some* of the FM and suffix array can be calculated, every n rows. So that the entire array doesn’t need to be calculated for each string found.

FOR STRING: “ACTGCTCGGCT”

See evans\_hw4.py for code that produce these.

BW Matrix Last Col- V

[['$' 'A' 'C' 'T' 'G' 'C' 'T' 'C' 'G' 'G' 'C' '**T**']

['A' 'C' 'T' 'G' 'C' 'T' 'C' 'G' 'G' 'C' 'T' '**$**']

['C' 'G' 'G' 'C' 'T' '$' 'A' 'C' 'T' 'G' 'C' '**T**']

['C' 'T' '$' 'A' 'C' 'T' 'G' 'C' 'T' 'C' 'G' '**G**']

['C' 'T' 'C' 'G' 'G' 'C' 'T' '$' 'A' 'C' 'T' '**G**']

['C' 'T' 'G' 'C' 'T' 'C' 'G' 'G' 'C' 'T' '$' '**A**']

['G' 'C' 'T' '$' 'A' 'C' 'T' 'G' 'C' 'T' 'C' '**G**']

['G' 'C' 'T' 'C' 'G' 'G' 'C' 'T' '$' 'A' 'C' '**T**']

['G' 'G' 'C' 'T' '$' 'A' 'C' 'T' 'G' 'C' 'T' '**C**']

['T' '$' 'A' 'C' 'T' 'G' 'C' 'T' 'C' 'G' 'G' '**C**']

['T' 'C' 'G' 'G' 'C' 'T' '$' 'A' 'C' 'T' 'G' '**C**']

['T' 'G' 'C' 'T' 'C' 'G' 'G' 'C' 'T' '$' 'A' '**C**']]

Suffix Array

01 || ACTGCTCGGCT

07 || CGGCT

10 || CT

05 || CTCGGCT

02 || CTGCTCGGCT

09 || GCT

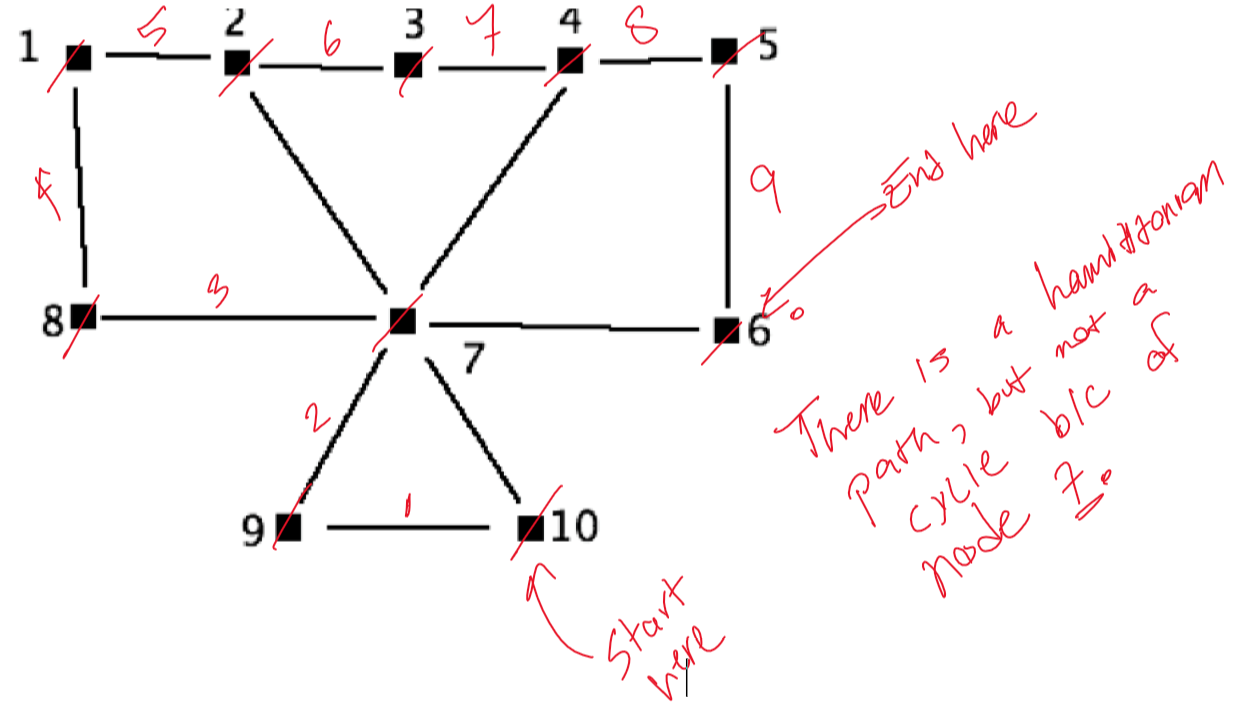
04 || GCTCGGCT

08 || GGCT

11 || T

06 || TCGGCT

03 || TGCTCGGCT



A Hamiltonian cycle is one that traverses each node of a given graph, and cycle indicates that the traversal ends at the same node that you started. We can traverse the graph such that we visit each node once, but we do not start and end at the same node, so it does have a Hamiltonian path but not a Hamiltonian cycle.

Additionally, it does not have a eulerian cycle because there is no way to balance the graph as some nodes have an odd number of edges (2 & 4).