CS170–Fall 2014 — Solutions to Homework 9

Kevin Hu, SID 23360403, `cs170-pj`

November 12, 2014

1. Main Idea

My algorithm finds the amount of characters that every read overlaps with the front and end of every other read. Then, it keeps merging reads to the beginning or end of the output string, with the next read to be merged being the read with the greatest amount of overlapping characters with the beginning or end of the output string.

Before finding the overlap amounts, it go through all of the reads and remove any read u that is a substring of another read v, since any read that overlaps with u will necessarily overlap with v as well.

For every read u, it keeps a map "front" that maps an integer i to a list of reads that overlap the beginning of u by i characters. For example, for a read v in the list mapped to i, u[0, i] = v[v.length-i, v.length]. Also, it keeps a map "back" that maps an integer j to a list of reads such that u[u.length-j, u.length] = v[0, j]. For example, given reads u, v, and w: u = [eeeeabcd], v = [abcdffffffff], w = [fffffee]. u.front = {2:[w]}, u.back = {4:[v]}; v.front = {4:[u]}, v.back = {5:[w]}; w.front = {5:[v]}, w.back = {2:[u]}. While finding the overlaps between reads, it keeps a global variable maxTotalOverlap that's set to the maximum amount of characters that any two reads overlap by. Also, it keeps track of the two reads maxFront and maxEnd where maxFront[maxFront.length-maxTotalOverlap, maxFront.length] = maxEnd[0, maxTotalOverlap].

After obtaining all the values of overlaps between every pair of reads, it merges maxFront and maxEnd and sets finalString to be equal to this merged string. Then, with a greedy algorithm approach, it merges the next read that overlaps the finalString by the most characters. To do this, we keep track of which read was most recently merged to the beginning and end of finalString with the variables currentFront and currentEnd, respectively. This is because after merging currentFront to the beginning of finalString, any read that overlaps the beginning of currentFront by i characters will necessarily overlap finalString by i characters as well; this applies for currentEnd and the end of finalString as well. Then, the next optimal read to merge is the read in currentFront.front or currentEnd.back mapped to the largest integer. After adding the next optimal read, the value of currentFront or currentEnd is updated accordingly, and we continue to merge the next optimal read until all reads have been added.

For an overlap amount i, there is a list of q reads. Assuming that the input DNA sequence is random and the short reads are created randomly, q should be equal to 1 most of the time except for small values of i, where reads may coincidentally overlap by i characters. If the greedy algorithm leads to a final string that does not include every read, which may be caused in the situation with small values of i, the algorithm simply adds any non-merged strings to the final string, since there can be no other optimal placement for the non-merged strings.

2. Pseudocode

input: list of n reads $s_0, s_1, ..., s_{n-1}$ of <= k length each
output: a final string that includes all reads

A. Remove all reads that are substrings of other strings:
1. for all pairs of reads u, v in reads:
2. if u == v, remove v from reads
3. else if u is a substring of v, remove u from reads
4. else if v is a substring of u, remove v from reads

B. For every pair of reads u and v, find the maximum number of characters that the end of u overlaps with the beginning of v and vice versa:
5. for all pairs of reads u, v in reads:
6. set uBackOverlap = vFrontOverlap = the maximum value of i such that
u[u.length-i, u.length] = v[0, i]
7. set vBackOverlap = uFrontOverlap = the maximum value of j such that
u[0, j] = v[v.length-j, v.length]
8. if uBackOverlap > 0:
9. put v in u.back.get(uBackOverlap)
10. put u in v.front.get(uBackOverlap)
11. if vBackOverlap > 0:
12. put u in v.back.get(vBackOverlap)
13. put v in u.front.get(vBackOverlap)

C. Merge all of the reads to get the final output:
14. alreadyMerged = set of reads that have already been merged into the final string
15. let totalMaxOverlap be the maximum value of uBackOverlap or vBackOverlap from part B, and reads totalMaxFront and totalMaxBack that overlap by totalMaxOverlap such that totalMaxFront[totalMaxFront.length - totalMaxOverlap] = totalMaxEnd[0, totalMaxOverlap]
16. set finalString = totalMaxFront + totaMaxEnd[totalMaxEnd.length - totalMaxOverlap, totalMaxEnd]
17. add totalMaxFront and totalMaxEnd to alreadyMerged
18. set currentFront = totalMaxFront and currentEnd = totalMaxEnd
19. while alreadyMerged.size < reads.size:
20. nextRead = read with the greatest overlap in currentFront.front and currentEnd.back
21. if no more optimal next reads exist:
add any reads that have not been merged to finalString and return
22. if nextRead was from currentFront.front:
merge nextRead to the front of finalString and set currentFront = nextRead
23. else if nextRead was from curretEnd.back:
merge nextRead to the end of finalString and set currentEnd = nextRead
24. return finalString

3. Runtime analysis

Part A takes n^2 time to observe every pair of reads, and takes at most k time per pair of reads to check if one is a substring of another. Therefore, part A takes $n^2 * k$ time.

Part B takes n^2 time to observe every pair of reads u, v. To get the maximum number of overlapping characters between u and v, we have to compare the substrings u[0, i] and v[v.length-i, v.length] for i:[1, length of the smaller string], where the upper bound of i is k in the worst case. Also, we have to compare v[0, j] and u[u.length-j, u.length] where the bounds of j are the same as i. Comparing the substrings requires comparing indivdual characters in the substrings, so we will compare a total of 2* $\sum_{i=0}^{k} i$ = 2*(1/2)*k*(k+1) = $k^2 + k$ characters in the worst case. Therefore, part B takes n^2k^2 time.

In part C, to find the optimal read, we have to find the max integer in the keysets of currentFront.front and currentEnd.back. There can be a maximum of n keys in either front or map back, so finding the max will take n time. We also go through the while loop n times. Checking if a read has been already merged can be done in constant time using a HashSet. Therefore, we find the max n number of times, so Part C takes n^2 time.

Therefore, the overall runtime is $n^2 + n^2k^2 + n^2 \in \theta(n^2k^2)$.