Suffix trees: How to do Google search in bioinformatics?

Curriculum module designer: Ananth Kalyanaraman (ananth@eecs.wsu.edu)

PROGRAMMING PROJECT COMPONENT:

(please refer this document alongside the slides in Lecture4_PatternMatching_Project.pptx)

I) PARALLEL ALGORITHMS FOR PATTERN MATCHING Pattern matching with error tolerance – Problem definition

Inputs:

- {G: Genome FASTA file containing a single genome sequence with N characters}
- {Q: Queries FASTA file containing m queries, each of possibly a different length.}
- {Error tolerance level expressed in % (of query length)}

Problem statement: To search for the input queries in the genome, and report the topmost hit for each query. The "*topmost hitfor each query"* is defined as the location in the genome which matchesto the query with the least number of mismatching characters (errors), and the number of errors should be no more than the error tolerance level indicated by the user.If there is no such location, then the code outputs a "query not found" message. If there is more than one such location, then output any one of those locations (chosen arbitrarily) as the output.

Possible extensions:

1) One possible extension could be to ask the students to provide all locations where a query is found (under the allowed error threshold).
2) Another extension could be to have students implement a more optimized version of their code assuming no errors (i.e., queries need to match exactly to be reported as found).
3) A much more involved extension would to assume there is no genome and instead the problem goal is that given only a set of queries, find all other queries in the set that match significantly with each of the queries. This will be equivalent of performing an all-against-all comparison.

II) Pattern matching with error tolerance – A Naïve algorithm

Implementation source folder: pmatch_naive

A Naïve serial algorithm:

Assumptions:

1) The queries are of roughly the same length (i.e., negligible standarddeviation in the average query length).
2) There is sufficient memory available (local RAM) to store the entire Genome input.

This is a naive implementation that does a brute force search for each query against the entire genome, by sliding a window of the length of the query along the genome and enumerating only those positions that correspond to the topmost hit.
Note: This version does *NOT* need to use suffix trees or any other sophisticated string data structures.

The algorithmic steps are outlined below:

1) Load the input genome into a string array in the memory (call it G[1..N]).
   As for queries, you can either read all the queries at once, or one at a time.
2) For each query $q \in Q$
   i) Set "maximum allowed errors" = error tolerance (%) x query length (|q}|)
   ii) slide a window of the length |q| along the entire length of the genome.
   (Note: There will be N-|q|+1 windows.)
   iii) for each window:
      a. compare q against the characters in that window
      b. if the number of mismatching positions is below the maximum allowed errors for this query, then check if it the best hit so far. If it is, then update best hit information and remember the genomic location (i.e., window).
   iv) At the end, output one best hit for the query

Computational complexity analysis:

Let m: number of queries
M: sum of the length of all the queries
N: length of the genome
l: average length of a query
(p: number of processors)

Time complexity:

O(lN) per query, implying a total of O(mlN) (=O(MN)) for all queries.

Space complexity:

Each processor stores the entire genome and queries can be read one at a time. So space complexity is O(N).

Parallel algorithm for MPI platforms:
(An MPI implementation is provided in the pmatch_naive folder.)
Assuming the lengths of the queries are all roughly equal to one another (i.e., negligible standard deviation), this algorithm is easy to parallelize.

1) Each processor loads the entire genome G.
2) Each processor loads O(m/p) queries. This can be done in a distributed manner without needing any communication. For instance, assuming the queries are in one file of size ~M, each processor computes the file size, and then processor rank i can read all sequences between the file byte offset intervals: [i*(filesize/p) to

(i+1)*(filesize/p)]. This can be done through parallel I/O without requiring any communication. An alternative approach is to have a dedicated processor (say, rank 0) broadcast all the starting and ending file offsets to each of the other processor ranks.

3) Each processor runs the serial algorithm for searching its local set of O(m/p) queries against the genome.

Let p be the number of processors. This algorithm will take O(MN/p) time and O(N+m/p) space.

Parallel algorithm for OpenMP platforms:

The above same approach and analysis will carry forward to a OpenMP multithreaded setting as well with some minor changes:

1) The master thread loads the entire genome G.
2) The master thread also all the queries into the memory.
3) The master thread spawns p worker threads.
4) In the next step, the master thread distributes the queries to the worker threads, and have them compute on their individual sets. This can be implemented as either a static or dynamic partitioning.
    a. Static partitioning: The master thread partitions the query set into p groups, such that thread rank i receives the ~m/p queries in the interval [i*m/p -> (i+1)m/p - 1]. The master thread simply informs each worker thread of this work assignment and the individual worker threads start working on their portion. This implementation does *not* need any locking.
    b. Dynamic partitioning: Each worker thread first picks an unclaimed query (next in queue) and runs the serial algorithm for searching it against the genome. As threads pick a query they also mark it in the query array so that other threads do not pick it. This is the only step that needs a lock/unlock.

    Step (4) can be used to introduce students to the idea of static vs. dynamic partitioning schemes. The advantage of static (over dynamic) partition is that it has no overhead attached to locking/unlocking. However, static partitioning can lead to potential load imbalances that the dynamic partitioning could potentially overcome. Empirical testing can help students identify these issues.

Parallel algorithm for MapReduce platforms:

One MapReduce algorithm is as follows:

Let us assume there are p_m mappers and p_r reducers.

1) Each mapper loads roughly $O(m/p_m)$ queries (using the input splitter function)
2) Each mapper loads the entire genome once.
3) Each mapper then searches each local query against the genome and outputs.
4) There is no need for reducers in this model.

Another variant of MapReduce algorithm could be that, each mapper reads roughly an equal portion of both the query set and the genome. Then each mapper compares the

local queries against the locally stored genome, and emits intermediate <key,value> pairs the form <query id, best local hits>. Then the MapReduce shuffle stage will gather all hits corresponding each query in a designated reducer, where that reducer can report the global best hit.

III) Pattern matching with error tolerance – A suffix tree based algorithm

Implementation source folder: pmatch_st

This is a suffix tree based implementation that does searches for each query against the suffix tree built for the entire genome.

Assumptions:

1) The queries are of roughly the same length (i.e., negligible standard deviation in the average query length).
2) There is sufficient memory available locally to each processor to store the entire Genome input, its suffix tree, and part of the queries.

Serial Algorithm:

In this suffix tree based algorithm, search for each query against the suffix tree built for the entire genome. The main algorithmic steps are as follows:

(One-time preprocessing step)

1) Load the input genome and build the suffix tree for the genome. This can be done in O(N) time using linear algorithms such as the McCreight algorithm. The code provided in pmatch_st does this.

(Query matching)

2) Load the queries (incrementally or all at once).
3) For each query $q \in Q$
   i) Set "maximum allowed errors" = error tolerance (%) x query length (|q}|)
   ii) Search q in the suffix tree of G as follows:

Search algorithm:

A) FIND THE LONGEST SUBSTRING MATCHING POSITIONS

i) Start at the root, walk down the tree by comparing one character at a time with the query vs. tree path until one of the following happens:

a) there is mismatch:

Action:

Update longest path if this is the longest;
Keep track of the matching locations
(by querying the internal node immediately below the path);
If the number of mismatches is greater than the cutoff then
quit search;

or, b) the query's length has been successfully exhausted:

Action: select the longest matching path among the paths visited so far for extension

B) EXTEND AND EVALUATE TO FIND WHICH THE BEST HIT (ACCOUNTING FOR ERRORS BELOW CUTOFF)

i) extract the window of |q| characters from the genome and perform a simple comparison

ii) score each window by the number of errors

iii) output and report the window with the least number of errors.

This logic is implemented in the function st_search() of pmatch_st.cpp. This search function requires API calls to the public methods in the SuffixTree class (in SuffixTree.h).

Parallelization:

All of the above parallelization methods described for the Naïve approach can also be used here. Of course a more interesting extension (and a lot more challenging one) would be to build the suffix tree itself in parallel. There are some fairly complex solutions available in theliterature which may not be best suited for an undergraduate course curriculum. This could however provide a great discussion point in class regarding the challenges in parallel algorithm design.

Computational complexity analysis:

Time complexity:

Preprocessing time (one time):

O(N) to build the suffix tree in each processor for the entire genome

Query time:

O(l) per query, implying a total of O(ml) (=O(M)) for all queries.

In parallel, assuming linear scaling, this would imply O(M/p) time.

Space complexity:

Each processor stores the entire genome and O(m/p) queries.

Per processor peak space complexity = O(N + M/p).

The constant of proportionality of the suffix tree is roughly 150 in the current implementation. That is, for every input byte the suffix tree takes 150 bytes.

IV) Pattern matching with error tolerance – Naïve vs. suffix tree based approach: An evaluation

The Naïve and Suffix tree-based algorithms each have their individual strengths and weaknesses.

The Naïve approach is certainly simple and straightforward to code. In fact most of my coding effort on that component was consumed in implementing the Input/Output operations (e.g., FASTA file reading) that are necessary to load the inputs. The solution is also really simple to analyze in terms of time & memory complexities. The only disadvantage is that for *each* query the algorithm scans through the entire genome input. Therefore, the this approach cannot be considered scalable for very large genome sizes. This is an issue in practice, because the DNA databases are expanding in size and it won't be desirable to have a method whose time for searching a single query depends on the size of the database on that particular day! Perhaps one of the experiments that the students can do is to determine an approximate threshold genome size value for which this implementation still finishes in a practical amount of time (i.e., at most a few seconds per query).

The Suffix Tree approach overcomes the above disadvantage by performing a query search in time proportional to the query's length (and is independent of the genome size). The suffix tree construction is a one-time preprocessing activity. In the real world, if the genomic database grows then the tree has to be re-built occasionally (but never frequently). Therefore, under a setting where the genome database does not change frequently and there are huge number of queries that appear over time (something like a Google search setting), then suffix tree based approach can be highly desirable.

It is also not way more complex in terms of coding the pattern matching component using the suffix tree. Coding the pattern matching routine using suffix trees should not be much more complex than the naïve approach (since the code for suffix tree construction is already provided).The programmer only has to know how to navigate the tree top down from the root and for that needs to understand the API of the Suffix Tree class (which has only a handful of public methods). In fact I was able to reuse most of my code from the naïve implementation in my suffix tree based pattern matching code (compare the st_search in pmatch_st.cpp with the main function in pmatch_naive.cpp).

The only disadvantage of the suffix tree based approach is its space requirement. Although the storage requirements vary linearly with the length of the genome (O(N)), the constant of proportionality tends to be big in practice. In our current implementation it is well over 100, which makes it harder to conduct searches using serial computers with limited RAM. There are more space efficient data structures such as suffix arrays, but those could be difficult to cover as part of an undergraduate curriculum. Nevertheless this project could be a great introduction to the value of string data structures in general.