CS 482: Computational Techniques in Biological Sequence Analysis Homework #1

Eric Haoran Huang[1]

[1]e48huang@uwaterloo.ca, 20880126, e48huang

Abstract

This assignment here is meant to analyze and showcase the power of dynamic programming and the relevant sequence alignment programs.

Setup

I used Python 3.8.10 in this run without any external dependencies besides the regular Python library dependencies.

Part 1: Edit Distance

CLI

Run this section using

```
> python edit_distance.py --input <FASTA file> --debug
```

where the debug flag is optional, for output such as the alignment and distance array.

We have the FASTA file is formatted as:

```
>seq1
ACGT
>seq2
ACTG
```

See q1_examples/ for more examples.

Implementation and Runtime

Let n be the length of v and m be the length of w.

The implementation is pretty simple, we followed the classic dynamic programming approach of tabulating data and doing a bottom-up approach by filling out the table. We can accomplish this simply by creating a matrix of size $n \times m$ where n is the length of one nucleotide string, and m is the length of the other nucleotide string.

Then, we can describe as in class, the set of all possible sequences through a $n \times m$ table, where we fill it out, with each step taking $O(1)$ time (just three different comparisons with some other linear operators). We calculate the edit distance as the minimum of either mismatching, deleting or inserting.

Because we're filling out a $n \times m$ table, we're running in $O(nm)$ time.

Part 2: Fitting Alignment Problem

CLI

Run this section using

```
> python fit_alignment.py --input <FASTA file> --debug
```

where the debug flag is optional, for output such as the alignment and distance array.

We have the FASTA file is formatted as:

```
>seq1
ACGT
>seq2
ACTG
```

See q2_examples/ for more examples.

Implementation and Runtime

Again, we follow a dynamic programming approach. This time however, it's important to note that because we want to find any substring of v, but match the entirety of w, if we have the rows representing v, and the columns representing w, then we have that our final sequence needs to start in the first column and end in the last column.

Therefore, the implementation is set such that the base case is all of the first column's score is set to 0, and we take the best score out of the last column. Then, the rest of the scoring is set normally as we don't want any other skips besides where we start and where we end. The number of rows travelled represents the length of the substring of v.

Again, this is $O(|v||w|) = O(nm)$ for the tabular bottom-up approach plus the backtracking algorithm portion. The backtracking spends $O(|v|)$ time to find the best score, while it takes at most $O(|v| + |w|) = O(n + m)$ time to finish the path (as the longest path is from $(0, 0)$ to (n, m)).

Because $O(n + m)$ is slower than $O(nm)$, then we have that the entire algorithm runs in $O(nm)$.

Part 3: Identifying Viral Strands

As a primer, I decided to attempt to solve this problem in many different ways to try to reduce the running time as much as possible. This primarily includes using k-mer matching heuristics to either reduce the number of local alignments needed, to reduce the input to local alignments or getting rid of local alignments completely.

CLI

This section here has many different settings to tune and play around with. I will describe all of them here.

```
python search_variant.py --db <FASTA file>
--query <FASTA file> --debug --verbose --k <int> --top_k <int>
--blast_scoring --gen_close_match --local_align --k_multiplier <int>
--use_hsp_gapped
```

We have the FASTA file is formatted as:

```
>seq1
ACGT
>seq2
ACTG
```

Here is a more detailed description of all the flags.

1. db. This flag here specifies the database of sequences to compare against.
2. query. This flag here specifies the sequence we want to search within the database for, and find the best alignment with.

3. debug, verbose. Debugging flags used to print out. Verbose is extremely verbose, would not suggest using. Default is off for both, would suggest turning on debug for timing information.

4. k. This determines the size of your k-mers and is defaulted to 10.

5. top_k. This determines how many of the database sequences that were hit we should investigate. Default is 10.

6. blast_scoring. Determines whether to use the blast scoring method of reducing the penalty of gaps. Default is off.

7. gen_close_match. Determines whether we should generate all close k-mers in our inverse index. Default is off. Note: this increases our memory consumption exponentially, would not suggest using.

8. local_align. Determines our search algorithm. If turned on, then we simply run local alignment directly on the entirety of the database sequence. If turned off, we run HSP formatting, with it being dependent on whether the flag use_hsp_gapped is set to true. If the HSP gapped flag is turned on, it does not do local alignment on the range with the most HSPs, rather it simply replaces missing nucleotides with gaps. Default is off.

9. k_multiplier. Determines the range of which we run local alignment with. If we increase the k_multiplier, we look at larger ranges of nucleotides to local align against. Default is 2.

10. use_hsp_gapped. Only matters if local_align is set to false, as this affects our HSP matching. If used, then we will run gapped alignments on a subset of the database sequences with the range of the most HSPs, and avoid incurring the cost of local alignments. **Note: This is currently broken and not functionally performing as wanted due to time constraints.**

See q3_examples/ and extra_examples/ for more examples.

Implementation and Runtime

Before the alignment method, we follow these steps:

1. Build an inverse index by generating all k-mers of the database and create a mapping from k-mers to their database sequence id and position of where the k-mer starts within that sequence. Let v be the expected value of the viral sequence length. Then, we have that this step runs in $O(nv)$ time where there are n viral sequences. This is because the number of k-mers is linear (as there are $v - k + 1$ k-mers, which with k constant is $O(v)$) and with repeating this n times, we get $O(nv)$.

2. Generate the query k-mers. Given w is the size of the query, then this is $O(w)$ time.

3. Get the hits from the inverse index of the query k-mers. At most we'll be doing $O(w)$ queries with the sum of these queries returning all the possible viral sequences k-mers, so this is still $O(nv + w)$ time.

4. Once we have all our hits, we aggergate them and choose the database sequences with the highest hit frequency. This is a really good heuristic in knowing what the best local alignment will be, as the more hits there are, the more matches there are. Again, since we are only dealing with $O(nv + w)$ data and doing linear operations, we have this part running in $O(nv + w)$ time.

Then, once we get the top hits, we then align them according to three different strategies described below. All these strategies are at most $O(nvw)$ time, satisfying the requirements.

Full Local Alignment

By running:

```
python search_variant.py --db <FASTA file>
--query <FASTA file> --top_k 5
--local_align
```

We limit top-k because otherwise, there would be too much time taken.

The local alignment is similarly implemented as part b with the exception that we take the maximum of all the values and 0, i.e. we can start and stop anywhere. This doesn't change our runtime however, as it still lies in $O(vw)$ time.

In this case here, we run each local alignment k times according to the top-k we provide. We can not limit the top-k and at worst, this runs at $O(nvw)$ time as required, or realistically, $O(kvw)$ time. However, one of our other heuristics can make this even better.

Partial Local Alignment

By running:

```
python search_variant.py --db <FASTA file>
--query <FASTA file> --top_k 10
```

We limit top-k because otherwise, there would be too much time taken.

Just as the previous section we are using local alignment, however, we first take all our hits per single database sequence and build another heuristic to make our local alignment search smaller and in $O(w)$ time of the query. This is done by taking $O(nv)$ time to create ranges where we merge seeds if they are within a certain distance away, which we can control through the k_multiplier. By default, if it is 2 k-mers away, we say it's worth merging with. This technique was shown in class to be the two-hit technique. Then, by choosing the largest range and extending to match the sequence length of the query, we can then do a local alignment these two strings to generate the best score.

Note that the local alignment now is performing on two strings of the size of the query, otherwise known as performing on $O(w^2)$. We repeat this process k times, and so we run $O(k(nv + w^2))$, which if k is n, then this would be $O(n^2v + nw^2)$. Since n is significantly smaller than genome sequences, and w is usually smaller than v, this is a lot faster in practice.

Gapped Local Alignment

By running:

```
python search_variant.py --db <FASTA file>
--query <FASTA file> --top_k 10 --use_hsp_gapped
```

We limit top-k because otherwise, there would be too much time taken.

Note, this section here is describing what my next steps would be to improve upon the database querying. Just as the previous section, we do indeed construct ranges. This time, we construct ranges by filling in missing gaps where needed to best align the matching sections together. Then, this would run in $O(w)$ for score calculation and $O(nv)$ time for merging the seeds together. The total time would be $O(k(nv + w))$ in this case here, or over the entirety of the dataset, would be $O(n^2v + nw)$.