Project 4 Writeup – Parker Nilson

1. Source Code

(See Appendix)

2. Time and Space Complexity

The time complexity of this algorithm is `O(MN)`, where `M` is the length of sequence 1, and `N` is the length of sequence 2, and in the case of the banded algorithm, `K` is the bandwidth (7). This is because in each case, once we have constructed the matrix of edit distances (where each entry builds off of previous entries in a constant fashion), we simply need to look at the final entry to get the answer and perform a linear traversal through the back-pointers to construct the alignment.

Time Complexity

The general steps of this algorithm are:

```
seq1_truncated = seq1[:align_length]
seq2_truncated = seq2[:align_length]
```

Figure 1: Truncate Strings

1. `O(M) + O(N)`: Truncate the strings to align_length In python, truncating a string is an O(N) operation since it creates a copy of the part of the string you are truncating to.

2. `O(1)`: Define some inline utility functions like `in_bounds` and `to_banded`, and perform an early return if the sequences are not compatible

```
# Initialize the matrix with first row and column
EDIT_DISTANCE = [
    [
        row * 5 if col == 0 else col * 5 if row == 0 else float("inf")
        for col in (
            range(row - BANDED_D, row + BANDED_D + 1)
            if banded
            else range(len(seq2_truncated) + 1)
        )
        # This will only include the cells in the banded region if it is banded
        if in_bounds(row, col)
    ]
    for row in range(len(seq1_truncated) + 1)
]
```

Figure 2: Initialize Edit Distances

3. `O(MN)` unrestricted or `O(MK)` banded: Initialize the edit distances

- **Unrestricted:** In the unrestricted case, this algorithm will create an array of size `M x N` which represents the optimal edit distance of the substring `(0,m) and (0,n) for m in M and n in N`. Therefore, in order to initialize each value of this array it must perform `O(MN)` operations.
- **Banded:** In the banded case, this algorithm will create an array of size `M x K` (or `M x 7`). This is because each row only contains at most 7 items, as this is the bandwidth. Therefore, in order to initialize each of the 7 items on M rows, the runtime of this operation is `O(MK)`.
- **Note:** For each value in the array, the method `in_bounds` is called, however this is a simple pair of if-statements and has runtime `O(1)`, therefore it does not affect the runtime efficiency.

```
# Initialize the previous cell pointers (including the back pointers for
# the first row and column)
PREV = {
    (row, col): (row - 1, col) if col == 0 else (row, col - 1)
    for (row, col) in [(i, 0) for i in range(1, len(seq1_truncated) + 1)]
    + [(0, j) for j in range(1, len(seq2_truncated) + 1)]
}
```

Figure 3: Initialize Back Pointers

4. `O(M) + O(N)`: Initialize the back pointers For each of the N items on the first row, and each of the M items on the first column, we add a tuple to a python dictionary to represent a pointer back to the previous character. The runtime efficiency of adding an item to a dictionary in python is `O(1)` thanks to its hashed keys. Therefore, the overall runtime of adding these initial back pointers to the dictionary is `O(M) + O(N)`.

```
for row in range(len(seq1_truncated) + 1):
    for col in (
        range(row - BANDED_D, row + BANDED_D + 1)
        if banded
        else range(len(seq2_truncated) + 1)
    ):
        if not in_bounds(row, col):
            continue

        # Map the row, col to the appropriate cell in the banded matrix
        (row_i, col_i) = to_banded(row, col) if banded == True else (row, col)
```

Figure 4: Construct Edit Distance Array and Back Pointers

5. `O(MN)` unrestricted or `O(MK)` banded: Construct the EDIT_DISTANCE array and the PREV pointers In this loop, we are either going through the `M x N` array from left to right, top to bottom, or in the banded case, we are only visiting the 7 items within the band on each row. The row and column numbers are then mapped onto the `M x K` array if necessary with the `to_banded` function (which is a simple pair of if-statements and runs in `O(1)` time).

Since the only operations performed within this for-loop are `if`, `to_banded`, and editing values in the `EDIT_DISTANCE` array and `PREV` dictionary (which has already been established to be a `O(1)` operation thanks to hashed keys), we know that the time complexity must be that of the size of the array being mapped over,

in this case `EDIT_DISTANCE`. Thus, in the unrestricted case, the time complexity of this step is `O(MN)`, and in the banded case, the time complexity is `O(MK)`.

6. `O(M + N)`: Backtrack through the PREV pointers to construct the optimal alignment Finally, this step must be, in the worst case, `O(M + N)`. This is because every pointer only ever points either to the left and/or up. Therefore, if we start at the bottom, right, the worst case scenario is that we must do `N` lookups to get all the way to the left, and then `M` lookups to get all the way to the top. Since looking up an item in a python dictionary uses hashed keys, it is `O(1)`, therefore we must perform at most `O(M + N)` operations. Then, we must reverse both strings, giving us another `O(M + N)` operation.

Conclusion Therefore, we have `O(M) + O(N) + O(1) + O(MN) + O(M) + O(N) + O(MN) + O(M + N)`, which simplifies to `O(MN)` when we drop non-dominating terms.

Space Complexity

This algorithm requires `O(MN)` space in the unrestricted case, and `O(MK)` space in the banded case. The following is a breakdown of the memory used by this algorithm: 1. `O(MN)` unrestricted, `O(MK)` banded: The `EDIT_DISTANCE` array. This array will always have `M` rows, and either `N` columns if unrestricted, or only `K` columns when it is confined to the band.

2. `O(MN)` unrestricted, `O(MK)` banded: The `PREV` dictionary. This dictionary can have at most `M x N` entries in the unrestricted case because it only contains `(row, col)` keys for rows and columns that exist in the `PREV_DISTANCE` array. In the banded case, it is at most size `M x K` for the same reason.
3. `O(N)`: Copies of the original sequences This algorithm includes multiple copies of the original sequences, none of these exceeding the original length. Therefore they account for `O(N)` space complexity.

Therefore, in total, only `O(MN)` space is used in the unrestricted case, and `O(MK)` in the banded case.

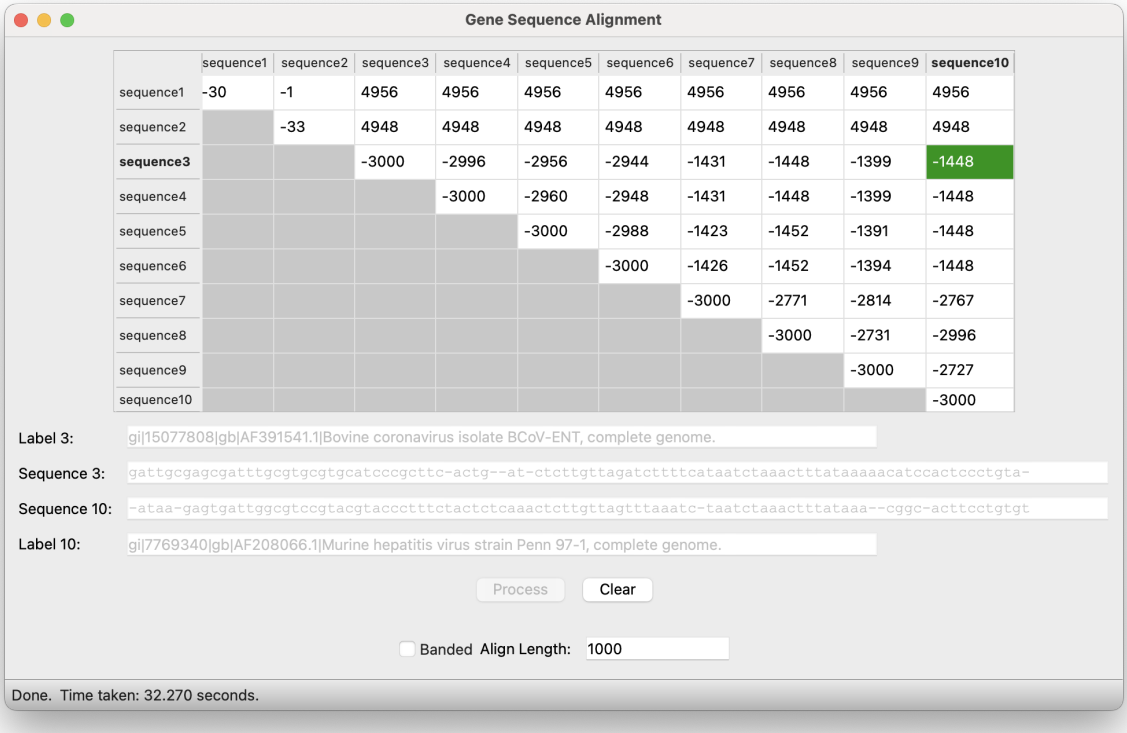

Figure 5: Unrestricted Algorithm with n = 1000

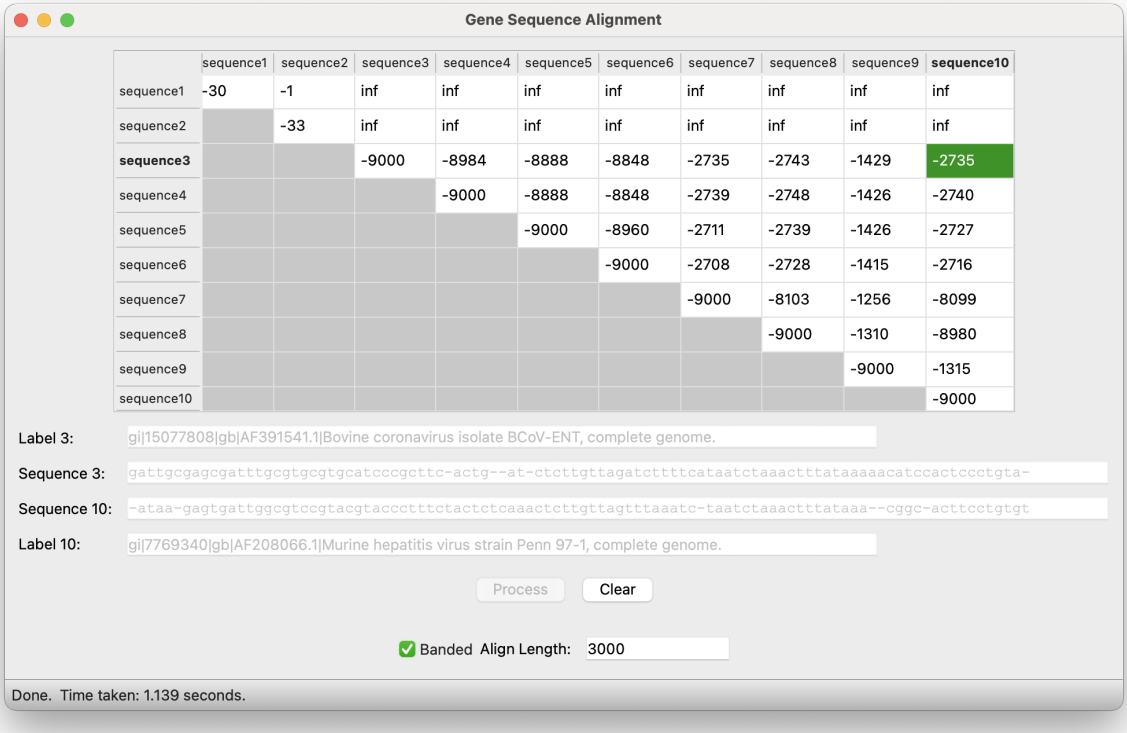

Figure 6: Banded Algorithm with n = 3000

Figure 7: Unrestricted Algorithm with n = 1000, sequences

Figure 8: Banded Algorithm with n = 3000, sequences

3. Screenshots

Unrestricted algorithm, 1000 align length (Completes in under 120 seconds)

Banded algorithm, 3000 align length (Completes in under 10 seconds)

4. Test Alignment

Sequence 3 x Sequence 10, First 100 characters, unrestricted n=1000

Sequence 3 x Sequence 10, First 100 characters, banded n=3000

Appendix

Source Code

```
# Used to implement Needleman-Wunsch scoring
MATCH = -3
INDEL = 5
SUB = 1

BANDED_D = 3      You, 2 weeks ago • Implement banded implementation


def get_in_bounds(seq1, seq2, banded):
    """
    Returns a function that checks if a given row, col is in bounds
    in comparison to the given sequences, and within the banded region
    (if banded is True)
    """
    def in_bounds(row, col):
        if row < 0 or row > len(seq1) or col < 0 or col > len(seq2):
            return False
        if banded == True:
            if col < row - BANDED_D or col > row + BANDED_D:
                return False
            if row < col - BANDED_D or row > row + BANDED_D:
                return False
        return True

    return in_bounds
```

```
def align(self, seq1, seq2, banded, align_length):
    self.banded = banded
    self.MaxCharactersToAlign = align_length

    seq1_truncated = seq1[:align_length]
    seq2_truncated = seq2[:align_length]

    in_bounds = get_in_bounds(seq1_truncated, seq2_truncated, banded)

    def to_banded(row, col):
        """
        if banded is True, maps the given row, col so that it points
```

```
# Initialize the matrix with first row and column
EDIT_DISTANCE = [
    [
        row * 5 if col == 0 else col * 5 if row == 0 else float("inf")
        for col in (
            range(row - BANDED_D, row + BANDED_D + 1)
            if banded
            else range(len(seq2_truncated) + 1)
        )
        # This will only include the cells in the banded region if it is banded
        if in_bounds(row, col)
    ]
    for row in range(len(seq1_truncated) + 1)
]

# Initialize the previous cell pointers (including the back pointers for
# the first row and column)
PREV = {
    (row, col): (row - 1, col) if col == 0 else (row, col - 1)
    for (row, col) in [(i, 0) for i in range(1, len(seq1_truncated) + 1)]
    + [(0, j) for j in range(1, len(seq2_truncated) + 1)]
}
```

```python
for row in range(len(seq1_truncated) + 1):
    for col in (
        range(row - BANDED_D, row + BANDED_D + 1)
        if banded
        else range(len(seq2_truncated) + 1)
    ):
        if not in_bounds(row, col):
            continue

        # Map the row, col to the appropriate cell in the banded matrix
        (row_i, col_i) = to_banded(row, col) if banded == True else (row, col)

        # prefer left, then top, then diagonal

        # Calculate diagonal distance
        if in_bounds(row - 1, col - 1):
            (prev_row_diag, prev_col_diag) = (
                to_banded(row - 1, col - 1)
                if banded == True
                else (row - 1, col - 1)
            )
            prev_diag_dist = EDIT_DISTANCE[prev_row_diag][prev_col_diag]
            potential_dist_diag = (
                prev_diag_dist - 3
                if seq1_truncated[row - 1] == seq2_truncated[col - 1]
                else prev_diag_dist + 1
            )
            EDIT_DISTANCE[row_i][col_i] = potential_dist_diag
            PREV[(row, col)] = (row - 1, col - 1)

        # Calculate top distance (and update if better)
        (prev_row_top, prev_col_top) = (
            to_banded(row - 1, col) if banded == True else (row - 1, col)
        )
        if in_bounds(row - 1, col):
            prev_top_dist = EDIT_DISTANCE[prev_row_top][prev_col_top]
            potential_dist_top = prev_top_dist + 5
            if potential_dist_top <= EDIT_DISTANCE[row_i][col_i]:
                EDIT_DISTANCE[row_i][col_i] = potential_dist_top
                PREV[(row, col)] = (row - 1, col)

        # Calculate the left distance (and update if better)
        (prev_row_left, prev_col_left) = (
            to_banded(row, col - 1) if banded == True else (row, col - 1)
        )
        if in_bounds(row, col - 1):
            prev_left_dist = EDIT_DISTANCE[prev_row_left][prev_col_left]
            potential_dist_left = prev_left_dist + 5
            if potential_dist_left <= EDIT_DISTANCE[row_i][col_i]:
                EDIT_DISTANCE[row_i][col_i] = potential_dist_left
                PREV[(row, col)] = (row, col - 1)
```

```python
# Backtrack to
seq_1_aligned_
seq_2_aligned_
(cur_row, cur_
while cur_row
    (prev_row,
    # if diag,
    if prev_ro
        seq_1_
        seq_2_
    # if top,
    elif prev_
        seq_1_
        seq_2_
    # if left,
    elif prev_
        seq_1_
        seq_2_
    (cur_row,

seq_1_aligned
seq_2_aligned

(end_row, end_
    to_banded(
    if banded
    else (len(
)
score = EDIT_D
alignment1 = s
alignment2 = s
```

```
return {
    "align_cost": score,
    "seqi_first100": alignment1[:100],
    "seqj_first100": alignment2[:100],
}
```