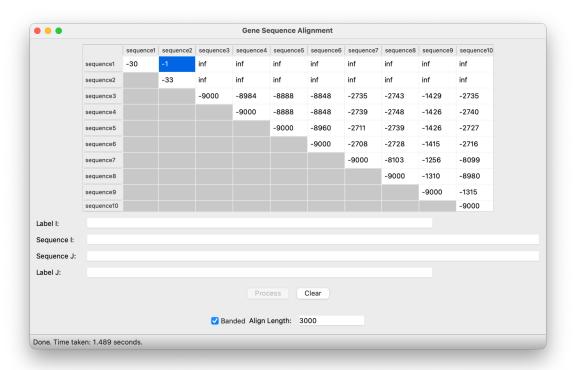
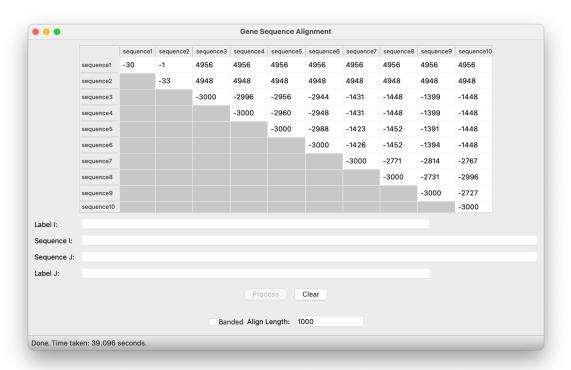
Project 4 - Gene Sequence Alignment

Austin McKamey CS312 Nov 1, 2022

- 1. Commented Code in #5
 - a. Unrestricted:
 - i. Time Complexity: O(mn)
 unrestricted() O(mn) nested for loops of length m and n
 find_min() O(1) all comparisons and assignments
 make_strings() O(n) where n is the length of the alignment
 - ii. Space Complexity: O(mn) unrestricted() O(mn) makes table of size mn find_min() O(5) \sim O(1) temporary values make_strings() O(2n + 3) \sim O(n) where n is the length of the alignment
 - b. Banded:
 - Time Complexity: O(kn)
 restricted() O(kn) nested for loops of length k and n
 find_min() O(1) all comparisons and assignments
 make strings() O(n) where n is the length of the alignment
 - ii. Space Complexity: O(kn) restricted() O(kn) makes table of size kn find_min() $O(8) \sim O(1)$ temporary values make_strings() $O(2n + 3) \sim O(n)$ where n is the length of the alignment
- 2. For both the unrestricted and banded algorithms, my program first creates a table of tuples based off of the Needleman-Wunsch alignment values. In both algorithms, there are base cases that are automatically filled in with no computation on the top row and left column of the table. The rest of the values are then computed by the minimum, and the direction of the box used to get the minimum is stored in the second value of the tuple. In the banded algorithm, the table values are three tuples, and the third value is the index of the letter in the sequence, because that is not preserved by the width of the table. The backtrace then begins with the score, or minimum cost of the full alignment. Using the "directions" found in the second value of the tuples, it moves to that box in the table and repeats until it reaches the upper left corner of the table, which is the beginning, concatenating the alignments along the way. Finally, these strings are reversed and sliced to the proper length.





Sequence #3 & Sequence #10 - unrestricted

attgcgagcgatttgcgtgcgtgcatcccgcttc-actg--at-ctcttgttagatcttttcataatctaaactttataaaaacatccactcctgta-gataa-gagtgattggcgtccgtacgtaccctttctactctcaaactcttgttagtttaaatc-taatctaaactttataaa--cggc-acttcctgtgtg

Sequence #3 & Sequence #10 - banded

5.

```
def align(self, seq1, seq2, banded, align_length):
  self.banded = banded
  self.MaxCharactersToAlign = align length
  # slice sequences to the align length given, normally 1000
  if len(seq1) > align length:
     seq1 = seq1[0: align length]
  self.sq1 = seq1
  if len(seq2) > align length:
     seq2 = seq2[0: align_length]
  self.sq2 = seq2
  if banded:
     self.restricted()
     ans1, ans2 = self.make strings restricted()
     if len(self.sq1) == len(self.sq2): # score is stored at table[len][MAXINDELS]
when lengths are equal
        score = self.table[len(seq1)][MAXINDELS][0]
        # slice alignments to the first 100 characters if the alignment is possible
        ans1 = ans1[0: 100]
        ans2 = ans2[0: 100]
     elif (len(self.sq2) - len(self.sq1)) > 2: # score is infinite when
differences in length are too great
        score = math.inf
        ans1 = "No Alignment Possible"
        ans2 = "No Alignment Possible"
     else: # score is stored at table[len][MAXINDELS + 1] when lengths are
different
        score = self.table[len(seq1)][MAXINDELS + 1][0]
        ans1 = ans1[0: 100]
        ans2 = ans2[0: 100]
 else:
     self.unrestricted()
     ans1, ans2 = self.make_strings_unrestricted()
     score = self.table[len(seq1)][len(seq2)][0] # score is stored at bottom right
hand corner of table
    ans1 = ans1[0: 100]
     ans2 = ans2[0: 100]
  alignment1 = (ans1 + ' DEBUG:({} chars,align len={}{})').format(
```

```
len(seq1), align_length, ',BANDED' if banded else '')
  alignment2 = (ans2 + ' DEBUG:({} chars,align_len={}{})').format(
    len(seq2), align_length, ',BANDED' if banded else '')
  return {'align_cost': score, 'seqi_first100': alignment1, 'seqj_first100':
alignment2}
# general banded algorithm
def restricted(self):
 height = len(self.sq1) + 1
 length = len(self.sq2) + 1
 self.table = []
 t = -3 # preserves starting index of j in the sequence in the next "band" of the
  for i in range(height): # O(kn) where k is BANDWIDTH and n is height (time &
space)
    row = []
    for j in range(t, length):
       if j == t + BANDWIDTH: # breaks from creating row when it is the correct
width
          break
       edge = 'false'
       k = -t # helps preserve index in row rather than index in sequence
       # checks position in row to account for missing values in later
computations
       if j == t + BANDWIDTH - 1:
          edge = 'back'
       elif j == t:
          edge = 'front'
       # inserts null values at the beginning of the table to account for bands
off table
       if j < 0:
           row.append((None, None))
       elif i == 0: # base cases
           row.append((j * 5, 'l', j))
       elif i <= MAXINDELS and i == 0: # base cases
           row.append((i * 5, 't', j))
       else: # computes min from surrounding available values
           row.append((self.find_min_restricted(i, j, k + j, row, edge)))
     t += 1
     self.table.append(row)
# computes current min value from surrounding available values
def find_min_restricted(self, i, j, k, row, edge): # 0(1) only performing
comparisons, conditionals, and assignments (time)
 mini = math.inf
 c = 'x'
                                           # O(1) single temporary values created
and discarded (space)
```

```
# always checks diagonal values first
 if self.sq1[i - 1] == self.sq2[j - 1]:
    temp = MATCH + self.table[i - 1][k % BANDWIDTH][0]
     temp = SUB + self.table[i - 1][k % BANDWIDTH][0]
 if temp <= mini:</pre>
    mini = temp
    c = 'd'
 # only checks to the left or top if that value exists
 if edge == 'front':
    temp = INDEL + self.table[i - 1][k % BANDWIDTH + 1][0]
    if temp <= mini:</pre>
        mini = temp
        c = 't'
 elif edge == 'back':
    temp = INDEL + row[len(row) - 1][0]
    if temp <= mini:</pre>
        mini = temp
       c = '1'
 else: # both values exist
    temp = INDEL + self.table[i - 1][k % BANDWIDTH + 1][0]
    if temp <= mini:</pre>
        mini = temp
        c = 't'
    temp = INDEL + row[len(row) - 1][0]
    if temp <= mini:</pre>
        mini = temp
        c = '1'
 return mini, c, j
# constructs alignment strings
def make_strings_restricted(self):
 i = len(self.sq1)
 # location of score varies based on length discrepancies
 if len(self.sq1) == len(self.sq2):
     j = MAXINDELS
 else:
    j = MAXINDELS + 1
 answer1 = ''
 answer2 = ''
 curr = self.table[i][j]
 # concatenates to string until it reaches the beginning of the table
 while i != 0 or j != 3: # O(n) where n is the length of the alignment (time &
space)
    # adds letters or dashes and increments or decrements indices as necessary
    if curr[1] == 'd':
        answer1 = answer1 + self.sq1[i - 1]
```

```
answer2 = answer2 + self.sq2[curr[2] - 1]
       i -= 1
     elif curr[1] == 't':
       answer1 = answer1 + self.sq1[i - 1]
       answer2 = answer2 + '-'
       i -= 1
       j += 1
    else:
       answer1 = answer1 + '-'
       answer2 = answer2 + self.sq2[curr[2] - 1]
       j -= 1
    curr = self.table[i][j]
 return answer1[::-1], answer2[::-1] # reverses strings, as they were constructed
backwards
# general unrestricted algorithm
def unrestricted(self):
 height = len(self.sq1) + 1
 width = len(self.sq2) + 1
 self.table = []
 for i in range(height): # O(mn) where m is height and n is width (time & space)
    row = []
    for j in range(width):
       if i is 0: # base cases
           row.append((j*5, '1'))
       elif j is 0: # base cases
           row.append((i*5, 't'))
       else: # computes min from surrounding values
           row.append((self.find_min_unrestricted(i, j, row)))
     self.table.append(row)
# computes current min value from surrounding values
def find min unrestricted(self, i, j, row): # 0(1) only performing comparisons,
conditionals, and assignments (time)
 mini = math.inf
 c = 'x'
                                      # O(1) single temporary values created and
discarded (space)
 # checks all directions to get min value
 if self.sq1[i - 1] is self.sq2[j - 1]:
    temp = MATCH + self.table[i - 1][j - 1][0]
 else:
    temp = SUB + self.table[i - 1][j - 1][0]
 if temp <= mini:</pre>
    mini = temp
     c = 'd'
 temp = INDEL + self.table[i - 1][j][0]
 if temp <= mini:</pre>
```

```
mini = temp
    c = 't'
 temp = INDEL + row[j - 1][0]
 if temp <= mini:</pre>
    mini = temp
     c = '1'
 return mini, c
# constructs alignment strings
def make_strings_unrestricted(self):
 i = len(self.sq1)
 j = len(self.sq2)
 answer1 = ''
 answer2 = ''
 # concatenates to string until it reaches the beginning of the table
 while i is not 0 and j is not 0: # O(n) where n is the length of the alignment
(time & space)
    curr = self.table[i][j]
    # adds letters or dashes and increments or decrements indices as necessary
    if curr[1] is 'd':
       answer1 = answer1 + self.sq1[i-1]
       answer2 = answer2 + self.sq2[j-1]
       i -= 1
       j -= 1
    elif curr[1] is 't':
       answer1 = answer1 + self.sq1[i-1]
       answer2 = answer2 + '-'
       i -= 1
    else:
       answer1 = answer1 + '-'
       answer2 = answer2 + self.sq2[j-1]
 return answer1[::-1], answer2[::-1] # reverses strings, as they were constructed
backwards
```