Project2 Boyer Moore

Shengyuan Wang
Mar 31, 2020

Implement versions of the naive exact matching and Boyer-Moore algorithms that additionally count and return (a) the number of character comparisons performed and (b) the number of alignments tried. Roughly speaking, these measure how much work the two different algorithms are doing.

Q1. How many alignments does the naive exact matching algorithm try when matching the string GGCGCGGTGGCTCACGCCTGTAATCCCAGCACTTTGGGAGGCCGAGG (derived from human Alu sequences) to the excerpt of human chromosome 1?

Q2. How many character comparisons does the naive exact matching algorithm try when matching the string GGCGCGGTGGCTCACGCCTGTAATCCCAGCACTTTGGGAGGCCGAGG (derived from human Alu sequences) to the excerpt of human chromosome 1?

Q3. How many alignments does Boyer-Moore try when matching the string GGCGCGGTGGCTCACGCCTGTAATCCCAGCACTTTGGGAGGCCGAGG (derived from human Alu sequences) to the excerpt of human chromosome 1?

In [26]:
```python
def readGenome(filename):
    genome = ''
    with open(filename, 'r') as f:
        for line in f:
            # ignore header line with genome information
            if not line[0] == '>':
                genome += line.rstrip()
    return genome


# class Boyer_Moore is provided by instructor Ben Langmead
def z_array(s):
    """ Use Z algorithm (Gusfield theorem 1.4.1) to preprocess s """
    assert len(s) > 1
    z = [len(s)] + [0] * (len(s)-1)

    # Initial comparison of s[1:] with prefix
    for i in range(1, len(s)):
        if s[i] == s[i-1]:
            z[1] += 1
        else:
            break

    r, l = 0, 0
    if z[1] > 0:
        r, l = z[1], 1

    for k in range(2, len(s)):
        assert z[k] == 0
        if k > r:
            # Case 1
            for i in range(k, len(s)):
                if s[i] == s[i-k]:
                    z[k] += 1
                else:
                    break
            r, l = k + z[k] - 1, k
        else:
            # Case 2
            # Calculate length of beta
            nbeta = r - k + 1
            zkp = z[k - l]
            if nbeta > zkp:
                # Case 2a: zkp wins
                z[k] = zkp
            else:
                # Case 2b: Compare characters just past r
                nmatch = 0
                for i in range(r+1, len(s)):
                    if s[i] == s[i - k]:
                        nmatch += 1
                    else:
                        break
                l, r = k, r + nmatch
                z[k] = r - k + 1
    return z


def n_array(s):
    """ Compile the N array (Gusfield theorem 2.2.2) from the Z array """
    return z_array(s[::-1])[::-1]


def big_l_prime_array(p, n):
    """ Compile L' array (Gusfield theorem 2.2.2) using p and N array.
        L'[i] = largest index j less than n such that N[j] = |P[i:]| """
    lp = [0] * len(p)
    for j in range(len(p)-1):
        i = len(p) - n[j]
        if i < len(p):
            lp[i] = j + 1
    return lp


def big_l_array(p, lp):
    """ Compile L array (Gusfield theorem 2.2.2) using p and L' array.
        L[i] = largest index j less than n such that N[j] >= |P[i:]| """
    l = [0] * len(p)
    l[1] = lp[1]
    for i in range(2, len(p)):
        l[i] = max(l[i-1], lp[i])
    return l


def small_l_prime_array(n):
    """ Compile lp' array (Gusfield theorem 2.2.4) using N array. """
    small_lp = [0] * len(n)
    for i in range(len(n)):
        if n[i] == i+1:  # prefix matching a suffix
            small_lp[len(n)-i-1] = i+1
    for i in range(len(n)-2, -1, -1):  # "smear" them out to the left
        if small_lp[i] == 0:
            small_lp[i] = small_lp[i+1]
    return small_lp


def good_suffix_table(p):
    """ Return tables needed to apply good suffix rule. """
    n = n_array(p)
    lp = big_l_prime_array(p, n)
    return lp, big_l_array(p, lp), small_l_prime_array(n)


def good_suffix_mismatch(i, big_l_prime, small_l_prime):
    """ Given a mismatch at offset i, and given L/L' and l' arrays,
        return amount to shift as determined by good suffix rule. """
    length = len(big_l_prime)
    assert i < length
    if i == length - 1:
        return 0
    i += 1  # i points to leftmost matching position of P
    if big_l_prime[i] > 0:
        return length - big_l_prime[i]
    return length - small_l_prime[i]


def good_suffix_match(small_l_prime):
    """ Given a full match of P to T, return amount to shift as
        determined by good suffix rule. """
    return len(small_l_prime) - small_l_prime[1]


def dense_bad_char_tab(p, amap):
    """ Given pattern string and list with ordered alphabet characters, create
        and return a dense bad character table.  Table is indexed by offset
        then by character. """
    tab = []
    nxt = [0] * len(amap)
    for i in range(0, len(p)):
        c = p[i]
        assert c in amap
        tab.append(nxt[:])
        nxt[amap[c]] = i+1
    return tab


class BoyerMoore(object):
    """ Encapsulates pattern and associated Boyer-Moore preprocessing. """
    def __init__(self, p, alphabet='ACGT'):
        # Create map from alphabet characters to integers
        self.amap = {alphabet[i]: i for i in range(len(alphabet))}
        # Make bad character rule table
        self.bad_char = dense_bad_char_tab(p, self.amap)
        # Create good suffix rule table
        _, self.big_l, self.small_l_prime = good_suffix_table(p)

    def bad_character_rule(self, i, c):
        """ Return # skips given by bad character rule at offset i """
        assert c in self.amap
        assert i < len(self.bad_char)
        ci = self.amap[c]
        return i - (self.bad_char[i][ci]-1)

    def good_suffix_rule(self, i):
        """ Given a mismatch at offset i, return amount to shift
            as determined by (weak) good suffix rule. """
        length = len(self.big_l)
        assert i < length
        if i == length - 1:
            return 0
        i += 1  # i points to leftmost matching position of P
        if self.big_l[i] > 0:
            return length - self.big_l[i]
        return length - self.small_l_prime[i]

    def match_skip(self):
        """ Return amount to shift in case where P matches T """
        return len(self.small_l_prime) - self.small_l_prime[1]


def naive_with_counts(p, t):
    occurrences = []
    count_chr = 0
    count_align = 0
    for i in range(len(t) - len(p) + 1):
        count_align += 1
        match = True
        for j in range(len(p)):
            count_chr += 1
            if t[i+j] != p[j]:
                match = False
                break
        if match:
            occurrences.append(i)
    return occurrences, count_align, count_chr


def boyer_moore_with_counts(p, p_bm, t):
    """ Do Boyer-Moore matching. p=pattern, t=text,
        p_bm=BoyerMoore object for p """
    i = 0
    occurrences = []
    count_chr = 0
    count_align = 0
    while i < len(t) - len(p) + 1:
        count_align += 1
        shift = 1
        mismatched = False
        for j in range(len(p)-1, -1, -1):
            count_chr += 1
            if p[j] != t[i+j]:
                skip_bc = p_bm.bad_character_rule(j, t[i+j])
                skip_gs = p_bm.good_suffix_rule(j)
                shift = max(shift, skip_bc, skip_gs)
                mismatched = True
                break
        if not mismatched:
            occurrences.append(i)
            skip_gs = p_bm.match_skip()
            shift = max(shift, skip_gs)
        i += shift
    return occurrences, count_align, count_chr
```

In [15]:
```
genome_file = 'chr1.GRCh38.excerpt.fasta'

t = readGenome(genome_file)

# Question1, 2, 3
p = 'GGCGCGGTGGCTCACGCCTGTAATCCCAGCACTTTGGGAGGCCGAGG'

occurrences, num_alignments, num_character_comparisons = naive_with_counts(p, t)
print("Naive Exact occurrences, number of alignments try, number of charcter comparisons
 try:\n", occurrences[0], num_alignments, num_character_comparisons)
```

```
Naive Exact occurrences, number of alignments try, number of charcter comparisons try:
 56922 799954 984143
```

In [16]:
```
p_bm = BoyerMoore(p, alphabet='ACGT')
occurrences, num_alignments, num_character_comparisons = boyer_moore_with_counts(p, p_bm,
t)
print("BoyerMoore occurrences, number of alignments try, number of charcter comparisons t
ry:\n", occurrences[0], num_alignments, num_character_comparisons)
```

```
BoyerMoore occurrences, number of alignments try, number of charcter comparisons try:
 56922 127974 165191
```

Write a function that, given a length-24 pattern P and given an Index object built on 8-mers, finds all approximate occurrences of P within T with up to 2 mismatches. Insertions and deletions are not allowed. Don't consider any reverse complements.

Q4. How many times does the string GGCGCGGTGGCTCACGCCTGTAAT, which is derived from a human Alu sequence, occur with up to 2 substitutions in the excerpt of human chromosome 1?

Q5. Using the instructions given in Question 4, how many total index hits are there when searching for occurrences of GGCGCGGTGGCTCACGCCTGTAAT with up to 2 substitutions in the excerpt of human chromosome 1?

Write a function that, given a length-24 pattern P and given a \verb|SubseqIndex|SubseqIndex object built with k = 8 and ival = 3, finds all approximate occurrences of P within T with up to 2 mismatches.

Q6. When using this function, how many total index hits are there when searching for GGCGCGGTGGCTCACGCCTGTAAT with up to 2 substitutions in the excerpt of human chromosome 1?

In [19]:
```python
import bisect


def naive_2mm(p, t):
    occurrences = []
    for i in range(len(t) - len(p) + 1):  # loop over alignments
        count_mismatch = 0
        for j in range(len(p)):  # loop over characters
            if t[i+j] != p[j]:  # compare characters
                count_mismatch += 1
        if count_mismatch <= 2:
            occurrences.append(i)  # all chars matched; record
    return occurrences


def approximate_match(p, t, n):
    segment_length = int(round(len(p) / (n + 1)))
    all_matches = set()
    p_idx = Index(t, segment_length)
    idx_hits = 0
    for i in range(n + 1):
        start = i * segment_length
        end = min((i + 1) * segment_length, len(p))
        matches = p_idx.query(p[start:end])

        # Extend matching segments to see if whole p matches
        for m in matches:
            idx_hits += 1
            if m < start or m - start + len(p) > len(t):
                continue

            mismatches = 0

            for j in range(0, start):
                if not p[j] == t[m - start + j]:
                    mismatches += 1
                    if mismatches > n:
                        break
            for j in range(end, len(p)):
                if not p[j] == t[m - start + j]:
                    mismatches += 1
                    if mismatches > n:
                        break

            if mismatches <= n:
                all_matches.add(m - start)
    return list(all_matches), idx_hits


def approximate_match_subseq(p, t, n, ival):
    segment_length = int(round(len(p) / (n + 1)))
    all_matches = set()
    p_idx = SubseqIndex(t, segment_length, ival)
    idx_hits = 0
    for i in range(n + 1):
        start = i
        matches = p_idx.query(p[start:])

        # Extend matching segments to see if whole p matches
        for m in matches:
            idx_hits += 1
            if m < start or m - start + len(p) > len(t):
                continue

            mismatches = 0

            for j in range(0, len(p)):
                if not p[j] == t[m - start + j]:
                    mismatches += 1
                    if mismatches > n:
                        break

                if mismatches <= n:
                    all_matches.add(m - start)
    return list(all_matches), idx_hits


# class Index is provided by instructor Ben Langmead
class Index(object):
    """ Holds a substring index for a text T """

    def __init__(self, t, k):
        """ Create index from all substrings of t of length k """
        self.k = k  # k-mer length (k)
        self.index = []
        for i in range(len(t) - k + 1):  # for each k-mer
            self.index.append((t[i:i+k], i))  # add (k-mer, offset) pair
        self.index.sort()  # alphabetize by k-mer

    def query(self, p):
        """ Return index hits for first k-mer of p """
        kmer = p[:self.k]  # query with first k-mer
        i = bisect.bisect_left(self.index, (kmer, -1))  # binary search
        hits = []
        while i < len(self.index):  # collect matching index entries
            if self.index[i][0] != kmer:
                break
            hits.append(self.index[i][1])
            i += 1
        return hits


class SubseqIndex(object):
    """ Holds a subsequence index for a text T """

    def __init__(self, t, k, ival):
        """ Create index from all subsequences consisting of k characters
            spaced ival positions apart.  E.g., SubseqIndex("ATAT", 2, 2)
            extracts ("AA", 0) and ("TT", 1). """
        self.k = k  # num characters per subsequence extracted
        self.ival = ival  # space between them; 1=adjacent, 2=every other, etc
        self.index = []
        self.span = 1 + ival * (k - 1)
        for i in range(len(t) - self.span + 1):  # for each subseq
            self.index.append((t[i:i + self.span:ival], i))  # add (subseq, offset)
        self.index.sort()  # alphabetize by subseq

    def query(self, p):
        """ Return index hits for first subseq of p """
        subseq = p[:self.span:self.ival]  # query with first subseq
        i = bisect.bisect_left(self.index, (subseq, -1))  # binary search
        hits = []
        while i < len(self.index):  # collect matching index entries
            if self.index[i][0] != subseq:
                break
            hits.append(self.index[i][1])
            i += 1
        return hits
```

In [24]:
```
p = 'GGCGCGGTGGCTCACGCCTGTAAT'

# Using naive_2mm to check the result.
print("Occurence with up to 2 substitutions using Naive Exact:", len(naive_2mm(p, t)))
```

```
Occurence with up to 2 substitutions using Naive Exact: 19
```

In [25]:
```
matches, hit_num = approximate_match(p, t, 2)

# Question 4
print("Occurence with up to 2 substitutions using Boyer Moore:", len(matches))
```

```
Occurence with up to 2 substitutions using Boyer Moore: 19
```

In [22]:
```
# Question 5
print("Total index hits with up to 2 substitutions:", hit_num)
```

```
Total index hits with up to 2 substitutions: 90
```

In [23]:
```
# Question 6
print("Total index hits with up to 2 substitutions, using subsequences:", approximate_mat
ch_subseq(p, t, 2, 3)[1])
```

```
Total index hits with up to 2 substitutions, using subsequences: 79
```