Homework

July 31, 2021

1 Programming Homework 2 Instructions (Read First)

In a practical, we saw Python code implementing the Boyer-Moore algorithm. Some of the code is for preprocessing the pattern P into the tables needed to execute the bad character and good suffix rules — we did not discuss that code. But we did discuss the code that performs the algorithm given those tables:

[63]:
```
def boyer_moore(p, p_bm, t):
    """ Do Boyer-Moore matching. p=pattern, t=text,
        p_bm=BoyerMoore object for p """
    i = 0
    occurrences = []
    while i < len(t) - len(p) + 1:
        shift = 1
        mismatched = False
        for j in range(len(p)-1, -1, -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
```

Measuring Boyer-Moore's benefit First, download the Python module for Boyer-Moore preprocessing:

http://d28rh4a8wq0iu5.cloudfront.net/ads1/code/bm_preproc.py

This module provides the BoyerMoore class, which encapsulates the preprocessing info used by the boyer_moore function above. Second, download the provided excerpt of human chromosome 1:

http://d28rh4a8wq0iu5.cloudfront.net/ads1/data/chr1.GRCh38.excerpt.fasta

Third, implement versions of the naive exact matching and Boyer-Moore algorithms that addition-

ally 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.

For a few examples to help you test if your enhanced versions of the naive exact matching and Boyer-Moore algorithms are working properly, see these notebooks:

- Naive
- Boyer-Moore

[2]:
```
!wget http://d28rh4a8wq0iu5.cloudfront.net/ads1/code/bm_preproc.py
```

```
--2021-07-31 11:34:49--
http://d28rh4a8wq0iu5.cloudfront.net/ads1/code/bm_preproc.py
Resolviendo d28rh4a8wq0iu5.cloudfront.net (d28rh4a8wq0iu5.cloudfront.net)…
65.9.114.31, 65.9.114.155, 65.9.114.156, …
Conectando con d28rh4a8wq0iu5.cloudfront.net
(d28rh4a8wq0iu5.cloudfront.net)[65.9.114.31]:80… conectado.
Petición HTTP enviada, esperando respuesta… 200 OK
Longitud: 9400 (9.2K) [application/octet-stream]
Grabando a: "bm_preproc.py.1"

bm_preproc.py.1     100%[===================>]   9.18K  --.-KB/s    en 0.001s

2021-07-31 11:34:50 (11.5 MB/s) - "bm_preproc.py.1" guardado [9400/9400]
```

[3]:
```
!wget http://d28rh4a8wq0iu5.cloudfront.net/ads1/data/chr1.GRCh38.excerpt.fasta
```

```
--2021-07-31 11:35:04--
http://d28rh4a8wq0iu5.cloudfront.net/ads1/data/chr1.GRCh38.excerpt.fasta
Resolviendo d28rh4a8wq0iu5.cloudfront.net (d28rh4a8wq0iu5.cloudfront.net)…
65.9.114.156, 65.9.114.182, 65.9.114.31, …
Conectando con d28rh4a8wq0iu5.cloudfront.net
(d28rh4a8wq0iu5.cloudfront.net)[65.9.114.156]:80… conectado.
Petición HTTP enviada, esperando respuesta… 200 OK
Longitud: 810105 (791K) [application/octet-stream]
Grabando a: "chr1.GRCh38.excerpt.fasta"

chr1.GRCh38.excerpt 100%[===================>] 791.12K  2.64MB/s    en 0.3s

2021-07-31 11:35:05 (2.64 MB/s) - "chr1.GRCh38.excerpt.fasta" guardado
[810105/810105]
```

[28]:
```
from Bio import Seq, SeqIO
```

1.0.1 Questions 1 and 2

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? (Don't consider reverse complements.)

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? (Don't consider reverse complements.)

[31]:
```
my_dna = ""
string = "GGCGCGGTGGCTCACGCCTGTAATCCCAGCACTTTGGGAGGCCGAGG"

for read in SeqIO.parse("chr1.GRCh38.excerpt.fasta","fasta"):
    my_dna = str(read.seq)
```

[32]:
```
from collections import Counter
```

[33]:
```
Counter(my_dna)
```

[33]:
```
Counter({'T': 259344, 'G': 144991, 'A': 254581, 'C': 141084})
```

[72]:
```
def naive_with_counts(p, t):
    occurrences = []
    num_alignments = 0
    num_character_comparisons = 0
    for i in range(len(t) - len(p) + 1):  # loop over alignments
        match = True
        num_alignments += 1
        for j in range(len(p)):  # loop over characters
            num_character_comparisons += 1
            if t[i+j] != p[j]:  # compare characters
                match = False
                break
        if match:
            occurrences.append(i)  # all chars matched; record
    return occurrences,num_alignments,num_character_comparisons
```

[73]:
```
sequences = naive_with_counts(string, my_dna)
```

[74]:
```
sequences
```

[74]:
```
([56922], 799954, 984143)
```

[37]:
```
my_dna.count(string)
```

[37]:
```
1
```

1.0.2 Question 3

How many alignments does Boyer-Moore try when matching the string GGCGCGGTGGCTCACGCCTGTAATCCCAGCACTTTGGGAGGCCGAGG(derived from human Alu sequences) to the excerpt of human chromosome 1? (Don't consider reverse complements.)

[39]:
```
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]
```

[47]:
```
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 = []
    num_alignments = 0
    num_character_comparisons = 0
    while i < len(t) - len(p) + 1:
        shift = 1
        mismatched = False
        num_alignments += 1
        for j in range(len(p)-1, -1, -1):
            num_character_comparisons += 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, num_alignments, num_character_comparisons
```

[43]:
```
p = "GGCGCGGTGGCTCACGCCTGTAATCCCAGCACTTTGGGAGGCCGAGG"
p_bm = BoyerMoore(p)
```

[48]:
```
boyer_moore_with_counts(p,p_bm,my_dna)
```

[48]:
```
([56922], 127974, 165191)
```

1.0.3 Question 4

Index-assisted approximate matching. In practicals, we built a Python class called Index implementing an ordered-list version of the k-mer index. The Index class is copied below.

[82]:
```
class Index(object):
    def __init__(self, t, k):
        ''' Create index from all substrings of size 'length' '''
        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
    def genome_index(self):
         return self.index
```

We also implemented the pigeonhole principle using Boyer-Moore as our exact matching algorithm.

Implement the pigeonhole principle using `IndexIndex` to find exact matches for the partitions. Assume P always has length 24, and that we are looking for approximate matches with up to 2 mismatches (substitutions). We will use an 8-mer index.

Download the Python module for building a k-mer index.

https://d28rh4a8wq0iu5.cloudfront.net/ads1/code/kmer_index.py

Write a function that, given a length-24 pattern P and given an `IndexIndex` 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.

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? (Don't consider reverse complements here.)

Hint 1: Multiple index hits might direct you to the same match multiple times, but be careful not to count a match more than once.

Hint 2: You can check your work by comparing the output of your new function to that of the naive_2mm function implemented in the previous module.

[83]:
```
def naive_2mm(p, t):
    occurrences = []
    for i in range(len(t) - len(p) + 1):  # loop over alignments
        mismatches = 0
        match = True
        for j in range(len(p)):  # loop over characters
            if t[i+j] != p[j]:  # compare characters
                if mismatches < 2:
                    mismatches += 1
                else:
                    match = False
                    break
        if match:
            occurrences.append(i)  # all chars matched; record
    return occurrences
```

[84]:
```
p = "GGCGCGGTGGCTCACGCCTGTAAT"
len(p)
```

[84]: 24

[85]:
```
len(naive_2mm(p, my_dna))
```

[85]: 19

1.0.4 Question 5

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?

(Don't consider reverse complements.)

Hint: You should be able to use the boyer_moore function (or the slower naive function) to double-check your answer.

[91]:
```
import bisect

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
```

[113]:
```
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
```

[124]:
```
p = 'GGCGCGGTGGCTCACGCCTGTAAT'
```

[125]:
```
x,y = approximate_match(p, my_dna, 2)
```

[126]:
```
y
```

[126]: 90

1.0.5 Question 6

Let's examine whether there is a benefit to using an index built using subsequences of T rather than substrings, as we discussed in the "Variations on k-mer indexes" video. We'll consider subsequences involving every N characters. For example, if we split ATATAT into two substring partitions, we would get partitions ATA (the first half) and TAT (second half). But if we split ATATAT into two subsequences by taking every other character, we would get AAA (first, third and fifth characters) and TTT (second, fourth and sixth).

Another way to visualize this is using numbers to show how each character of P is allocated to a partition. Splitting a length-6 pattern into two substrings could be represented as 111222, and splitting into two subsequences of every other character could be represented as 121212.

For example, if we do:

[123]:
```
ind = SubseqIndex('ATATAT', 3, 2)
print(ind.index)
```

```
[('AAA', 0), ('TTT', 1)]
```

And if we query this index:

[103]:
```
p = 'TTATAT'
print(ind.query(p[0:]))
```

```
[]
```

because the subsequence TAA is not in the index. But if we query with the second subsequence:

[104]:
```
print(ind.query(p[1:]))
```

```
[1]
```

because the second subsequence TTT is in the index.

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

When using this function, how many total index hits are there when searching for GGCGCG-GTGGCTCACGCCTGTAAT with up to 2 substitutions in the excerpt of human chromosome 1? (Again, don't consider reverse complements.)

Hint: See this notebook for a few examples you can use to test your function.

https://nbviewer.jupyter.org/github/BenLangmead/ads1-hw-examples/blob/master/hw2_query_subseq_index.i

[120]:
```python
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 idx_hits
```

[121]:
```python
approximate_match_subseq(p, my_dna, 2, 3)
```

[121]:
```
79
```

[ ]:
```
```