Index-assisted approximate matching

Ben Langmead

You are free to use these slides. If you do, please sign the guestbook (www.langmead-lab.org/teaching-materials), or email me (ben.langmead@gmail.com) and tell me briefly how you’re using them. For original Keynote files, email me.
Dynamic programming summary

With these algorithms, we have a very powerful set of tools

- DP framework deals naturally with both mismatches and gaps
- DP scoring can be adjusted to capture variation, sequencing error, etc

And along the way we came up an algorithm we might use for read alignment:

Finding approximate occurrences of \( P \) in \( T \)

...but no faster than \( O(mn) \) and \( m \) is big!!!
A de-motivating example

\[ d = 6 \times 10^9 \text{ reads} \]
\[ n = 100 \text{ nt} \]
\[ m = 3 \times 10^9 \text{ nt} \approx \text{human} \]

\[ \{ \} \approx 1 \text{ week-long run of} \]

\[ \approx 1 \text{ week-long run of} \]

Say we have 1,000 processors, each clocked at 3 GHz, each capable of completing 8 dynamic programming cell updates per clock cycle

(We’re being optimistic)

Total of \[ d \times m \times n = 2 \times 10^{21} \text{ cell updates} \]

Takes \( > 2 \) years
A de-motivating example

\[
\begin{align*}
d &= 6 \times 10^9 \text{ reads} \\
 n &= 100 \text{ nt} \\
m &= 3 \times 10^9 \text{ nt} \approx \text{human}
\end{align*}
\]

\{ \approx 1 \text{ week-long run of Illumina HiSeq 2000} \}

Problem: our dynamic programming approach is \(O(dmn)\)

We’ll now consider two ideas for how to maintain the power of dynamic programming while diminishing effect of \(m\)
Index-assisted approximate matching

Idea 1: Use index for exact-matching subproblems, follow up with DP

Partition $P$, like for pigeonhole

Index finds exact partition matches (hits)

Use DP in vicinity of exact matches
Index-assisted approximate matching

Index-assisted function for finding occurrences of $P$ in $T$ with up to $k$ edits:

```python
def queryIndexEdit(p, t, k, index):
    ''' Look for occurrences of $p$ in $t$ with up to $k$ edits using an
    index combined with dynamic-programming alignment. '''
    l = index.in
    occurrences = []
    seen = set()  # for avoiding reporting same hit twice
    for part, poff in partition(p, k+1):
        for hit in index.occurrences(part):  # query index w/ partition
            # left edge of $T$ to include in DP matrix
            lf = max(0, hit - poff - k)
            # right edge of $T$ to include in DP matrix
            rt = min(len(t), hit - poff + len(p) + k)
            mn, off, xcript = kEditDp(p, t[lf:rt])
            off += lf
            if mn <= k and (mn, off) not in seen:
                occurrences.append((mn, off, xcript))
                seen.add((mn, off))
    return occurrences
```

Python example: [http://nbviewer.ipython.org/7011945](http://nbviewer.ipython.org/7011945)
Index-assisted approximate matching

Think in terms of the full $P$-to-$T$ dynamic programming matrix

Index is identifying diagonal stretches of matches
These are likely to be part of a high-scoring alignment
Many stretches within a few diagonals of each other are even more likely to be part of a high-scoring alignment
Neighborhood search

Idea 2: Use index to find exact occurrences of strings in P’s neighborhood

Neighborhood = set of strings within some Hamming / edit distance

The 1-edit neighborhood of cat, using DNA alphabet:

- cat, aat, gat, tat, cct, cg, ct, ca, ac

The 2-mismatch neighborhood of cat:

- cat, aat, gat, tat, cct, cg, ct, ca, ac

All ways to add 1 mismatch

All ways to add 2nd mismatch to aat

All ways to add 2nd mismatch to gat

All ways to delete 1 char

All ways to insert 1 char

...
Neighborhood search

Idea 2: Use index to find occurrences of strings in $P$'s “neighborhood”

Is the neighborhood huge? Can we bound it?

If $|P| = n$, and $|\Sigma| = a$, how many strings are within Hamming distance 1?

$$1 + n(a - 1)$$

$P$ itself  $a - 1$ ways to replace each of $P$'s $n$ chars

How many strings are within edit distance 1?

$$1 + n(a - 1) + n + (n + 1)a$$

Delete each char in $P$  $n + 1$ positions where we can insert any of the $a$ characters

In both cases, $O(an)$ strings in the neighborhood

* Some insertions are equivalent. E.g. there are two equivalent insertions of ‘a’ into ‘cat’. Likewise deletions (‘caat’).
Neighborhood search

How about within Hamming or edit distance 2?

\[O(an)\] strings within Hamming or edit distance 1, each with \(O(an)\) neighbors within distance 1, so \(O(a^2n^2)\)

Within distance \(k\)?

\[O(a^k n^k)\]

How much work to query suffix tree with all strings within distance \(k\)?

\[O(n + \#\,\text{occurrences})\] for each of the \(O(a^k n^k)\) strings, so roughly \(O(a^k n^{k+1})\)

Compare to \(O(a^k n^{k+1})\) to \(O(mn)\) for full dynamic programming

Good news: no \(m\). Bad news: exponential in \(k\).
Neighborhood search

Organize neighborhood of $P$ into a trie

Neighbors of $P = CAA$, within hamming distance 1:

# leaves = # neighbors = $1 + n(a - 1) = 1 + 3(4 - 1) = 10$
Neighborhood search

Navigating and/or building neighborhood trie is simple with recursion

Assume Hamming distance for now: Move left-to-right across $P$ and start with “budget” of $k$ mismatches

At each step, for each alphabet character $c$:

- If $c$ matches current character in $P$, recursively build subtree starting at next position of $P$ with same budget

- If $c$ mismatches current character in $P$ and budget > 0, recursively build subtree starting at next position of $P$ with 1 subtracted from budget.

Otherwise if budget = 0, move on.
def stringNeighbors(st, alph, edits=1, gaps=True):
    """ Given a string, an alphabet, and a maximum edit or Hamming
distance, return all strings within that distance. """
    ret = []
    def __editNeighborsHelp(st, edits, ii):
        for i in xrange(ii, len(st)):
            if edits > 0:
                if gaps:
                    # Insertion just before position i
                    for a in alph:
                        newst = st[:i] + a + st[i:]
                        __editNeighborsHelp(newst, edits - 1, ii)
                    # Deletion of position i
                    newst = st[:i] + st[i+1:]
                    __editNeighborsHelp(newst, edits - 1, ii+1)
                # Mismatch at position i
                for a in alph:
                    if a != st[i]:
                        newst = st[:i] + a + st[i+1:]
                        __editNeighborsHelp(newst, edits - 1, ii+1)
            if gaps and edits > 0:
                # Insertion just after last position
                for a in alph: ret.append(st + a)
                ret.append(st)
    __editNeighborsHelp(st, edits, 0)
    return ret
Neighborhood search

```python
>>> stringNeighbors("cat", "acgt", edits=1, gaps=False)
['aat', 'gat', 'tat', 'cct', 'cgt', 'ctt', 'caa', 'cac', 'cag', 'cat']

>>> stringNeighbors("cat", "acgt", edits=1, gaps=True)
```

Python example: [http://nbviewer.ipython.org/7012233](http://nbviewer.ipython.org/7012233)
Co-traversal

Say index is a suffix trie. Imagine querying it with each neighbor.

Suffix trie of $T = ACATAG$

Trie for neighborhood within 1 mismatch of $P = CAA$
Co-traversal

Say index is a suffix trie. Imagine querying it with each neighbor.

Suffix trie of $T = \text{ACATAG}$

Trie for neighborhood within 1 mismatch of $P = \text{CAA}$

Try AAA: we fall off suffix trie after A, before AA
Co-traversal

Say index is a suffix trie. Imagine querying it with each neighbor.

Suffix trie of $T = ACATAG$

Trie for neighborhood within 1 mismatch of $P = CAA$

Next try CAA: we fall off suffix trie after CA, before CAA
Co-traversal

Say index is a suffix trie. Imagine querying it with each neighbor.

Suffix trie of $T = \text{ACATAG}$

Trie for neighborhood within 1 mismatch of $P = \text{CAA}$

CAC and CAG also fail. Next try CAT: success.
Co-traversal

Say index is a suffix trie. Imagine querying it with each neighbor.

Suffix trie of $T = ACATAG$

Trie for neighborhood within 1 mismatch of $P = CAA$

Common path ending in a neighbor leaf corresponds to an alignment of a neighbor of $P$ to a substring of $T$
Co-traversal

We can find all such alignments with *co-traversal*: explore all paths that are present in both trees and end in a neighbor leaf.

Lexicographical depth-first co-traversal visits node pairs in this order:

- Suffix trie of $T = ACATAG$
- Trie for neighborhood within 1 mismatch of $P = CAA$

Only visited 8 nodes in these 20- and 22-node tries.
Co-traversal

We can also conduct best-first search, visiting paths with fewer edits before paths with more edits:

Light blue nodes match $P$ exactly. Others have 1 mismatch.

At this point we know there are no exact matches, so we widen our net to include alignments with at least one mismatch.

Suffix trie of $T = ACATAG$

Trie for neighborhood within 1 mismatch of $P = CAA$
Co-traversal

We can also conduct best-first search, visiting paths with fewer edits before paths with more edits:

Light blue nodes match $P$ exactly. Others have 1 mismatch.

Suffix trie of $T = ACATAG$

Trie for neighborhood within 1 mismatch of $P = CAA$
Co-traversal: pruning

We can think of the tree we’re exploring as being the *intersection* of these two trees...
Co-traversal: pruning

Suffix trie of $T = ACATAG \bigcap$ Trie for neighborhood within 1 mismatch of $P = CAA$
Co-traversal: indexing

Co-traversal uses the shape of the suffix trie, but we don’t want to actually build it. It’s $O(m^2)$ space. What’s an alternative?
Co-traversal: indexing

Alternative 1: Replace suffix trie with suffix tree

Suffix tree of $T = ACATAG\$
Co-traversal: indexing

Alternative 2: Replace suffix trie with suffix array

If we know range of SA elements with A as a prefix, additional binary searching gives range with AC as a prefix.

Even if we don’t build suffix trie/tree, suffix array allows us to traverse it.
Co-traversal: indexing

Alternative 3: Replace suffix trie with FM Index

Similar argument as suffix array, using LF Mapping instead of binary search

To traverse suffix trie, we need to build FM Index of $T' = \text{reverse}(T)$

Why?

Typical FM Index matches successively longer suffixes of $P$. If we want to match successively longer prefixes, we have to reverse $T$ before building FM Index.
Alignment summary

Exact matching with naive algorithm and Boyer-Moore
Online versus offline
Inverted indexes using substrings
Approximate matching: pigeonhole, q-gram lemma
Suffix indexes:

  Suffix Trie & Tree: querying, naive building
  Suffix Array: querying with binary search, accelerants
  FM Index: querying with LF mapping

Dynamic programming: edit distance, global alignment, local alignment
Combining dynamic-programming alignment with indexes; co-traversal