The Mammalian Sibelia Project

Ilia Minkin

1. Problem Formulation

Input: S: Set of m strings of equal length n and two positive integers k and c.

Output: G: De Bruijn graph of S constructed for substrings of length k with all bulges of size $\leq c$ collapsed into paths. Each edge of G also has a list of intervals of S associated with it.

Question: Come up with an algorithm with a low time/space complexity, in both m and n.

A note: while current version of Sibelia consequently constructs and collapses several graphs for different values of k and c, for sake of simplicity at this stage we assume we are given a single k and a c.

2. Analysis

The problem essentially consists of two parts: construction of the compressed de Bruijn graph out of S and its modification. While the graph construction can be well defined, its simplification is trickier.

2.1. Graph Construction Description

The problem of graph construction is as follows:

Input: S: Set of m strings of equal length n and a positive integers k.

Output: $G = (V, E)$: A compressed de Bruijn graph of S constructed for substrings of length k.

Once the vertex set V of the compressed graph is identified, construction of edge set E is straightforward. Suppose that there is a function $V_{id}(s)$ that returns number of vertex in G labeled with the k-mer s or -1 if there is no such vertex. Given V_{id} we construct edge set using the following algorithm:

Algorithm 1 Vertex Set of the Compressed Graph

Input: A collection of string $S = \{s_1, \ldots, s_m\}$, an integer k and function V_{id}
Output: Edge set E of G

$G \leftarrow \emptyset$
for $s \in S$ **do**
    $u \leftarrow V_{id}(s[1:k])$
    for $i \leftarrow 2$ **to** $|s|$ **do**
        $v \leftarrow V_{id}(s[i:i+k-1])$
        if $v \neq -1$ **then**
            $G \leftarrow G \cup (u, v)$
            $u \leftarrow v$

One of the possible ways to construct V_{id} efficiently is to scan the input strings two times:

1) At the first pass, store all $(k+1)$-mers in a set T using e.g. a Bloom filter
2) At the second pass, construct the dictionary of vertices by testing at each position if there are different edges leaving or entering the current k-mer

So the algorithm is the following:

Algorithm 2 Edge Set of the Compressed Graph

Input: A collection of string $S = \{s_1, \ldots, s_m\}$, an integer k
Output: Function V_{id}

$T \leftarrow \emptyset$
for $s \in S$ **do**
    for $i \leftarrow 1$ **to** $|s| - k$ **do**
        $T \leftarrow T \cup \{s[i:i+k]\}$
$V_{id} \leftarrow$ an empty dictionary
for $s \in S$ **do**
    for $i \leftarrow 1$ **to** $|s| - k + 1$ **do**
        $kmer \leftarrow s[i:i+k-1]$
        if $kmer \in V_{id}$ **then**
            continue
        $enter \leftarrow 0$ ▷ Number of entering edges
        $leave \leftarrow 0$ ▷ Number of leaving edges
        for $c \in \{A, C, G, T\}$ **do**
            if $kmer + c \in T$ **then**
                $leave \leftarrow leave + 1$
            if $c + kmer \in T$ **then**
                $enter \leftarrow enter + 1$
        if $enter > 1$ **or** $leave > 1$ **then**
            $V_{id}(kmer) \leftarrow |V_{id}|$

A note: the algorithms above consider only positive strands of the input strings, extending them to handle both strands is straightforward.

2.2. Graph Construction Complexity

Assuming that we implement the set T as a Bloom filter, we have following for the first stage. We have q hash functions and a bit array A of size w. Storing all $(k+1)$-mers requires time $O(nmq)$ and space $O(w+q)$: at each step we update values of q hash functions and modify A. A note: we assume that updating value of each hash function takes constant time (i.e. rolling hash).

The naive implementation of the second stage uses a hash table for V_{id}. Each testing for an edge takes $O(q)$ time since we have to update value of all hash functions to query T and we perform 8 testings for each position. Insertion to the hash table takes time $O(k)$ and we perform exactly $|V|$ such insertions.

Combining all together we have $O(nmq + k|V|)$ time and $O(|V| + w + q)$ space for both stages. It is worth to note that the space complexity depends on the size of output $|V|$ plus size of the bit array A for the Bloom filter. Size of the bit array only depends on desired false positive rate and can be tuned. It is important to say that false positives only hurt in the way of adding non-bifurcation vertices to the graph and do not generate any loss of information. Hence, the trade off between the false positive rate and the filter size could be quite efficient.

Assuming that G is sparse, it takes smaller space than the input itself and is better than using a suffix array (linear space respective to the input) or indexing the whole input in a hash table to store ordinary de Bruijn graph and compress it (a totally naive approach).

An unpleasant fact is that the complexity depends on k, unlike the suffix array implementation. But I believe that we can do much better.

2.3. Graph Simplification

A note: during the graph simplification we consider only paths corresponding to some substrings of the input genome. There is an issue that makes complexity analysis of the simplification tricky: it obviously depends on the structure of the graph, i.e. number of bulges. One may make an interesting observation: while collapsing a bulge obviously removes it from the graph, it may create other bulges. So the question is whether this process converges, i.e. do we remove more bulges from the graph than we create.

In practice it seems to converge, but it is possible to draw an example where it will oscillate forever: i.e. removing a bulge b_1 creates a bulge b_2 and removal of b_2 recreates b_1 and so on. It is a very special case that requires presence of at least two copies of substring s such that $s =$ reverse complement of s.

Another issue is that there is more than one way of simplification of the graph and they yield different graphs. A possible solution to both of this issues is to propose a concrete strategy of simplification and show that under this strategy:

- The simplification converges within reasonable time so we can estimate its complexity
- The resulting graph structure is "nice" in some way of defining niceness

References