Fragment assembly: An implementation in Ocaml

Martin Kochan (mkochan@ksu.edu)

25 November, 2005

Abstract

This text serves as a brief description of methods used in the OCaml program it accompanies. It is descriptive rather than analytic; there is not much attempt at formalism. After reading it, the reader should be much better able to understand the code. In fact, reading the code will likely not uncover many more new significant ideas than those already contained here. The reader needs to have experience with programming in OCaml and/or SML and should understand the workings of the Smith–Waterman algorithm.

1 Problem statement

We are given an array *sa* of sequences of characters `A`, `G`, `C`, `T`. Let us call these sequences *reads*.

We are to produce a list of *contigs*, which represent parts of this target sequence.

2 Contigs

Our definition of contig is that it is a group of sequences that have high local similarities, ie. are partly overlapping. A contig is represented as a layout of these sequneces. In a layout, the strings are mutually aligned thus characters contributing to better local similarities are found in one column. [1]

In the OCaml code, a contig is represented as:

[1] We could also define the contig as the resulting *consensus sequence* stemming from the layout. But here please consider the above definition.

```
type contig = int * layout   (* contig id, and it's layout*)
and layout = layout_row list
and layout_row = {i : int; shift : int; chs : char list}
```

Besides from having some id, a contig is represented by it's layout, as explained above. A layout is simply a list of `layout_row`s, each of them shifted `shift` characters to the right in respect to a *reference row* — one row is guaranteed to have a shift of zero. The attribute `i` says whose read (i-th read) the row is a laid-out form of. The attribute `chs` then enumerates char-by-char the row, with gaps introduced at some positions.

3 The process

Fragment assembly is performed as sequence of these steps:

1. Read n input strings (or *reads*). Make each string be contained in a single-item contig, or *singlet*.
2. (Preprocessing.) Determine the pairs of reads with significant *local* similarity. That is, create a graph whose vertices are the reads and whose edges are the pairs with significant local similarity.
3. Sort these edges into a list in decreasing order of local similarity scores.
4. Begin and keep reading the list of edges until empty and perform the following with every newly read edge (i, j):
   - Determine which contig c_i contains the read i.
   - Determine which contig c_j contains the read j.
   - Merge c_i and c_j into a new contig c', **so that the reads i and j are laid out in optimal pairwise alignment.**
5. Finally we have a set of contigs, each of them containing one or more reads, and no read appearing in two distinct contigs.

3.1 Optimization for step 2.

It turns out that to produce the similarity graph rigorously would require $O(mn^2) \subseteq O(n^2)$ time (assuming some average read length m). This is impractical for large n (which is common), hence we use a heuristic which runs in $O(nm)$ time.

This heuristic makes use of so-called *n-mers*. [2] The regions of high

[2] This notion was first used in PHRAP, a popular fragment assembly program.

similarity in locally aligned sequences usually contain at their center an interval called *n-mer* with 100% identity and no gaps. Usually $n \geq 14$.

Hence our optimization goes as follows: We gradually identify all subsequences of reads of length 14 — called *14-mers* — and create a hash, say, H_{14_mers}, that assigns to any 14-character string s a list of it's occurences across all reads.

Thus all 14-mers in H_{14_mers} that appear in more that one read actually imply that **their "owner" reads will be mutually connected by edges in the similarity graph.**

3.2 Note on merging of contigs (in step 4.)

Arguably the most important stage of the process is the merging of contigs.

Imagine we have just read an edge $E = (i, j)$ from the list of edges in the similarity graph. We wish to merge two contigs, c_i and c_j that contain the reads i and j, into a "common" contig c'.

As a trivial case, if $c_i = c_j$, then $c' = c_i = c_j$ is already merged. The contigs are equivalent whenever their id's are equal.

What if $c_i \neq c_j$? First recall that contigs have id's. We can simply make the id of c' to be the id of c_i. That will assure that every contig has a unique name (remember that initially, every read has been assigned a unique contig — a singlet).

But the main question is that of merging the layouts — let's call them l_i and l_j, the resulting layout being l'. Say we identified the rows r_i and r_j that represent i and j, respectively. In c', we want these two rows to be well-aligned along regions of high local similarity. We do something like this:

1. Perform *gapped Smith–Waterman* [3] alignment of r_i and r_j:
   - Find best *local* pairwise alignment between r_i and r_j. Note that the original rows already may contain gaps at some places — those are to be considered as wildcards [4].
   - Connect the remaining lower-quality parts of r_i and r_j to the pairwise alignment.
2. If the similarity score in the pairwise alignment was below certain threshold, terminate this routine and disregard the edge (i, j)! [5]

[3] The "gapped" here is my extension of the well-known Smith–Waterman algorithm.
[4] This actually is my "gapped" extension.
[5] This has to be, since our preprocessing using *14-mers* was just a heuristic.

3. By adding new gaps to into the aligned rows, the remaining rows in both layouts contain their letters at positions inconsistent with those in the new rows! Thus we add the new gaps into proper columns in both layouts.

4. We shift the rows in l_j so that the above-obtained shift is followed in l' as well, and so that r_i becomes the reference row in l'. Thus the layouts become "synchronized".

5. Finally, concatenate the synchronized layouts.

Scoring local similarity. In gapped Smith–Waterman, the scores for gapped sequences r_i and r_j are determined using a function $s(x, y)$, where x and y are some characters of r_i and r_j, respectively:

$$s(x,y) = \begin{cases} 1 & \text{if } x = \text{y} \\ 1 & \text{if } x \text{ or } y \text{ is a gap} \\ -1 & \text{if } x \neq \text{y} \end{cases}$$

Thus a gap really works as a wildcard. *However, a prize for* adding *a gap into the alignment is -3!*