OBSOLETE

SEE TR-IC-01-05

A program for building contig scaffolds in double-barreled shotgun genome sequencing

João Carlos Setubal   *Renato F. Werneck*

Relatório Técnico IC–00-20

Dezembro de 2000

A program for building contig scaffolds in double-barreled shotgun genome sequencing

João Carlos Setubal[*]   Renato F. Werneck[†]

OBSOLETE - SEE TR-IC-01-05

Abstract

We describe a program that builds contig scaffolds from contig assemblies, to be used in a whole-genome sequencing project. Our program builds scaffolds based on forward/reverse pair information (both from small clones, such as plasmids, and from large clones, such as cosmids). The program assumes that a DNA assembly, preferably with large repeats masked, is available. A scaffold is a path in a weighted graph, and the main novelty of our approach is a careful weighting scheme for arcs in this graph, such that heavier paths represent more reliable scaffolds. This weighting scheme takes into account the presence of repeats, possible clone duplication, existence of different clone libraries, and hybrid (small clones mixed with large clones) links between contigs. The program provides two different algorithms for scaffold building: one that uses a simple greedy strategy, and one that produces scaffolds that correspond to paths of maximum weight. If $|N|$ is the number of contigs and $|L|$ is the number of F/R pairs, the complexities are $O(|N| + |L| \log |L|)$ (greedy) and $O(|N|^2 + |L| \log |L|)$ (maximum-weight path). This program has been successfully used in a bacterial genome project.

1 Introduction

There seems to be a general agreement that the most efficient way to sequence the whole genome of a prokariote is by doing shotgun sequencing. This technique was first effectively demonstrated in the genome of *Haemophilus influenzae* [4], and has been used many times since. We assume knowledge of this technique. The problem with using only shotgun reads is the possible presence of long repeats in the genome. When one uses a "standard" assembly software such as `phrap` [5] to assemble a genome using only shotgun reads, repeats can cause the following problems:

1. Some contigs may be misassembled.
2. Two distinct regions that are very similar to one another may be merged into one (this is a *collapsed repeat*).

[*]setubal@ic.unicamp.br. Institute of Computing, University of Campinas, 13083-970 Campinas, SP, Brazil. Research supported in part by FAPESP and CNPq.

[†]rwerneck@inf.puc-rio.br. Department of Informatics, Pontifícia Universidade Católica do Rio de Janeiro, Rio De janeiro, RJ, Brazil. Research supported in part by CAPES.

Example of misassembled contigs:

- result of incorrect assembly:
  - A–`rrr`–B . . . C–`rrr`–D

  where A–`rrr`–B and C–`rrr`–D are contigs, and `rrr` is a repeat.
- correct assembly: A–`rrr`–D . . . C–`rrr`–B.

Example of collapsed repeat:

- result of incorrect assembly:
  1. A–`rrr`–B–C–`sss`–D
  2. `sss`–E–F–`rrr`
- correct assembly: A–`rrr`–B–C–`sss`–E–F–`rrr`–B–C–`sss`–D (note that the actual repeat is `rrr`–B–C–`sss`).

Other examples of problems in assembling in the presence of repeats are presented by Myers [7].

To explain one way to deal with these problems we need the concept of a *forward-reverse pair* (F/R pair for short). A given clone can be sequenced from either end. One end results in the *forward read* and the other results in the *reverse read*. Given that sequencers can read about 800 bp of sequence, then if the clone is larger than, say, 2 kbp, the sequences derived from the end reads will not overlap. This information can be used to "virtually" link contigs when read F from a clone is in one contig and read R is in another contig, and each is "pointing" to the other (relying on F/R pairs in an assembly project is also known as *double-barreled sequencing*, and has been described in [8]). This information can also be used to span repeats, as follows. In addition to obtaining shotgun inserts cloned in plasmids (whose size is no more than 5 kbp) the target DNA is cut in larger pieces. These fragments can be cloned in cosmids (where average insert length is about 40 kbp) or in BACs (average length 100 kbp). This means that these clones are able to span any repeats that might reasonably be expected in prokariotes. One then sequences only the ends (but both of them) of these large inserts. One then could use an assembly software that would take into account the fact that certain reads belong to clones that are a certain distance apart (the F/R pairs), and therefore deal correctly with repeats.

There are many assembly programs available. Three popular ones are: Staden [1], `phrap` [5], and CAP3 [6]. Staden and `phrap` do not use information from F/R pairs in determining contigs. CAP3 does, but in a limited fashion. This paper describes a program that can be used in conjunction with any assembly program to build a *scaffold* of a genome based on F/R pair information. A scaffold in this paper is an ordered sequence of contigs given by links provided by F/R pairs. The use of F/R pair information is done in a much more careful way than in the CAP3 program and results in better assemblies. On the other hand, being an add-on to an assembly program our program relies on some manual intervention to be of any practical use in a real genome project. The program has been successfully used in the *Xanthomonas axonopodis* pv *citri* genome project [2] in the following way:

1. All reads are assembled using `phrap`.
2. repeats are identified by comparing contig sequences to one another (several tools are available for this; one is `cross_match` [5]). Even collapsed repeats can be identified, because usually differing flanking sequences will cause parts of the repeat to separate, as shown in the example above.
3. After the longer (≥ 400 bp, i.e. larger than the average length of a read) repeats have been identified, a new whole assembly should be done, but screening (masking) the reads for those repeats.
4. Our program is applied on the resulting contigs.

In the remainder of this paper we describe the model upon which our program is based and the scaffold construction algorithms. The appendix provides documentation for the program.

2 Overview of Model and Algorithms

In this section we provide an overview that will be detailed in the following sections.

We model the problem of building a scaffold by the problem of finding paths in a weighted directed graph. In this graph, nodes represent contigs and an arc exists between nodes u and v if there is at least one F/R link between the corresponding contigs. Given the assembly output, it is relatively straightforward to build such a graph, but one has to be careful with read and contig orientation. One has also to deal with the possible many F/R links between two nodes, and this is a crucial part of our program.

The novelty of our approach is in determining arc weights. The idea is that the weight of an arc (u, v) represents the degree of confidence that we have that its u and v contigs are indeed linked. Therefore, the program has a preprocessing step in which all F/R links for each pair of nodes are carefully analyzed. The result of this analysis is the weight of the arc, and is based on a simple scoring scheme.

Path finding can be done by two different algorithms, at the user's choice. In one of the algorithms, we determine maximum weight paths in the graph G. This algorithm assumes G is acyclic. In the presence of repeats or errors the kind of graphs we build would not necessarily be acyclic. We use our weighting scheme to throw away arcs that would make G cyclic. In the other algorithm we greedily build vertex-disjoint paths by selecting heavier edges first.

3 Arc List Construction

In this section we describe in detail how the list of arcs between nodes is constructed based upon the DNA assembly and F/R link information.

A basic aspect of this construction is that it relies on naming conventions for the reads used in the assembly. This means that it is crucial for our program that most (but not all) read names do correspond to the actual physical clones. It is well known that this practice is subject to a number of errors, but this problem is becoming less of a concern with the increased use of capillary sequencers. We require three pieces of information from a read name: the clone end which it came from (either forward or reverse), the clone library which it came from, and whether it is an `sclone` or an `lclone` (see below). Most genome projects nowadays include such information in read names.

Another basic aspect of our construction is that it makes a distinction between *small clones* and *large clones*. We define an `sclone` as an F/R pair derived from a small clone ($\leq$ 5 kbp); and an `lclone` as an F/R pair derived from a large clone (in the range 30–55 kbp). In practice `sclone`s correspond to plasmids and `lclone`s correspond to cosmids.

A third basic aspect of our construction has to do with the information we require from the DNA assembly (step 1 in the process described at the end of Section 1). We will base the description below on `phrap`, but other assembly programs should be able to provide the same kind of information. If reads have been named according to the rules above, `phrap` will list all F/R pairs that it has detected as part of its output. In particular it lists F/R pairs whose components are in different contigs. For each such pair (r_f, r_r), it will output (among other things) the following information, which is used by our program:

- The orientation of r_f and of r_r with respect to the contigs where they have been placed.
- Alignment position of the $5'$ end of each read.
- An estimate of the clone length (which is given by the sum of the distance between the $5'$ end of a read to the far end of the contig it has been placed in with the analogous distance for the other read); we refer to this estimate as $\mathcal{L}$.
- The *primary* and *secondary alignment scores* of each read. The primary score of a read is the alignment score that the read has with respect to the contig to which it belongs. The secondary score is defined by the `phrap` documentation [5] thus: "the highest score of a match of the read against some other read in a different contig or elsewhere in the same contig (so reads for which this number is non-zero are those which overlap a repeat, or an incorrectly or incompletely assembled region)."

A final basic aspect of arc list construction has to do with the definition of an arc. Contigs have an orientation with respect to each other. This means that each contig has two ends: the *left end* and the *right end*. Given a contig c_u we represent its left end by c_u^l and its right end by c_u^r. The paths we will look for in the graph will always have to enter a node (contig) by one of its ends (either left or right) and then leave by the other (right or left). Therefore, an arc is a link between contig *ends* and not between contigs as a whole. Given an F/R pair and their respective contigs, how do we know what ends to link? This depends on the orientation of each read in its contig. Assuming contigs c_u and c_v and an F/R pair (r_p, r_q), so that read r_p is in contig c_u and read r_q is in contig c_v, we have arcs as given by Table 1.

Table 1: How links between contig ends are determined. c_u and c_v are contigs, and r_p and r_q are an F/R pair, such that r_p is in c_u and r_q is in c_v. When arrows are in the same direction it means that the direct sequence of the read was aligned; when arrows have opposite orientation, it means that the reverse complement of the read was aligned.

	$\overrightarrow{c_v}\overrightarrow{r_q}$	$\overrightarrow{c_v}\overleftarrow{r_q}$
$\overrightarrow{c_u}\overrightarrow{r_p}$	$c_u^r \leftrightarrow c_v^r$	$c_u^r \leftrightarrow c_v^l$
$\overrightarrow{c_u}\overleftarrow{r_p}$	$c_u^l \leftrightarrow c_v^r$	$c_u^l \leftrightarrow c_v^l$

Now we finally come to actual arc list construction. This is done in two phases. In the first phase all F/R pairs whose components are in different contigs are scanned. F/R pairs derived from `lclones` are retained only if their $\mathcal{L}$ is $\leq$ LMAX (user-defined). F/R pairs derived from `sclones` are retained only if their $\mathcal{L}$ is $\leq$ SMAX (user-defined).

In the second phase all F/R links between the same pair of contig ends are analyzed, for each pair of contig ends, with the aim of assigning a weight to the arc that will represent this link. This is done by a simple scoring scheme in which a read can contribute at most 1.0 point to the total weight. When assigning points to links we look for events that confirm the links and that are independent as much as possible from each other. The details are as follows:

- First, F/R pairs in which one member of the pair (or both) have a secondary score that is deemed too high are discarded. Secondary scores, as mentioned above, indicate that the read probably overlaps a repeat, and we do not want repeats to interfere with correct scaffold construction. Rather than dealing with high values of secondary scores, we prefer to deal with what we call *uniqueness:*

$$\text{uniqueness} = 1 - \frac{\text{secondary score}}{\text{primary score}}$$

  With this definition, well-anchored reads (secondary score = 0) have uniqueness = 1. Reads for which the uniqueness is below a certain threshold (user-defined) are discarded.

- Putative identical F/R pairs are identified, and only one copy is retained. An F/R pair (r_x^p, r_y^q) is considered identical to another F/R pair (s_x^p, s_y^q) (where x and y are the contigs), if the alignment positions of r_x^p and s_x^p differ by at most 5 bp, and if the alignment positions of r_y^q and s_y^q differ by at most 5 bp. Note that this check does not take into account the name of the reads; i.e., two reads could have totally different names and still be considered copies of each other. The value 5 bp is empirically derived.

- The first F/R pair from a given library contributes 1.0 to the total weight; each additional F/R pair from the same library contributes 0.5. We regard reads from the same library as being events that are not as independent as reads from different libraries.

- If all links are just of one type (all `sclone`s or all `lclone`s), then the weight of the arc is determined by the above rules. If there are `lclone`s mixed with `sclone`s, then further processing is done as follows.

  First a test is done to check whether each `lclone` link yields a value for $\mathcal{L}$ that is not too small. Note that having links from `sclone`s already indicates that the contigs could be quite close to each other in the genome. This means that one or both ends of `lclone`s should be quite a distance away from the contig end. To be consistent with this an `lclone` link must have $\mathcal{L}$ larger than or equal to LMIN (user-defined). If all `lclone` links are consistent with small separation between contigs, then both sets of links are considered valid, and an additional 1.0 is awarded to the arc weight. This bonus point comes from the empirical observation that k consistent `sclone` and `lclone` links give more credence to the contig link than k links of just one type.

  If some `lclone`s are long enough but some are not, then the ones for which $\mathcal{L}$ is too small are discarded (and their points are not counted), but an additional 1.0 is still awarded to the arc weight.

  If all `lclone`s seem to be too short, then the arc weight is determined by either `sclone` links only or by `lclone` links only. The set that has larger weight determines the choice.

4 Algorithms

4.1 Description

This section describes the algorithms used to find scaffolds in the genome: *Greedy Path* (GP) and *Maximum Weight Path* (MWP).

They share a common basic structure. First, they read an input file describing the problem by enumerating contigs and F/R links. Then, a set A_0 of weighted arcs is built according to the procedure described in Section 3. Starting from a graph $G = (N, \emptyset)$, both algorithms try to add arcs to G in a greedy fashion, scanning A_0 in nondecreasing order by weight. To be actually inserted, each arc must satisfy a given set of conditions. Finally, both algorithms find "good" paths in $G = (N, A)$, the resulting graph.

The essential difference between the algorithms is the set of conditions an arc a must satisify in order to be inserted into G.

In the description below we use (c_x^p, c_y^q) to represent an arc, where c_x^p is end p (either left or right) of contig/node x, and c_y^q is end q (either left or right) of contig/node y.

4.1.1 Greedy Path Algorithm

Let $a = (c_x^p, c_y^q)$ (with $1 \leq x, y \leq |N|$) be an arc we want to insert into G. For a to be actually inserted, two conditions must hold: (1) c_x and c_y must belong to different connected components, and (2) both c_x^p and c_y^q must be free, i.e., there must be no arc incident to any of them in G.

Note that this strategy is very restrictive. Not only does it avoid cycles, but it also forbids parallel paths. By the end of the algorithm, the graph will become a collection of vertex-disjoint paths, all of which are output.

4.1.2 Maximum Weight Path Algorithm

An arc $a = (c_x^p, c_y^q)$ (with $1 \leq x, y \leq |N|$) will be inserted into G by MWP if (1) at least one of $\{c_x^p, c_y^q\}$ is free, (2) a's orientation does not conflict with previously inserted arcs, and (3) a does not create a cycle. Note that conditions (2) and (3) will be relevant only when c_x and c_y belong to the same connected component.

Once the graph is built, the algorithm outputs a number of paths of different classes. For each connected component, the procedure is the following. First, find the heaviest *main path* p_1 and output it. Second, report all *alternate paths* between contigs in p_1, i.e., paths that start and end in contigs that appear in p_1 but do not use any arc used by p_1. Finally, remove all arcs of p_1 from the graph and repeat the process, finding the second heaviest main path p_2 and the corresponding alternate paths. As an additional constraint, no contig in p_i may have appeared in p_j, $j < i$. In other words, all main paths are vertex-disjoint. The connected component will be fully processed only when every arc is either part of some main path p_i or has both ends in main paths (in this case, the arc is said to *link* the paths and is listed accordingly in the output file).

4.2 Data Structures and Running Times

The first step of both GP and MWP is to build the set A_0 of arcs from the set L of F/R links read from the input file. The links that constitute the arcs can be determined in $O(|L| \log |L|)$ total time: sort L according to link ends and build the arcs from consecutive elements in L. Since the weight of each arc a is heuristically determined in linear time w.r.t. the number of links in a, $O(|L| \log |L|)$ is indeed the complexity of building A_0. Sorting this set according to arc weights can be done in $O(|A_0| \log |A_0|)$ time. An empty graph G can be built in $O(|N|)$ time, $|N|$ being the number of contigs.

Up to this point, the overall complexity of both algorithms is $O(|N| + |L| \log |L|)$, since $|L| \geq |A_0|$. The following subsections analyze the remaining operations.

4.2.1 Greedy Path Algorithm

For each arc, two tests must be made for an arc a to be inserted. First, the ends of a must be in different connected components. Using a forest-based implementation of a union-find data structure, we can check all arcs in $O(|A_0| \cdot \alpha(|N| + |A_0|, |N|))$ total time [9]. Second, the ends of a must be free. Testing this is trivial and can be done in $O(1)$ time. The paths in the graph G build by GP can be found in $O(|N|)$ time, since G is actually a collection of vertex-disjoint paths. Therefore, the overall complexity of the algorithm is still $O(|N| + |L| \log |L|)$.

4.2.2 Maximum Weight Path Algorithm

Although MWP's conditions are not as strict as GP's, testing them is a more complex task. Since contigs in the same connected component may be linked by a new arc, a union-find data strucuture is not enough. Nevertheless, the algorithm does require a union-find data strucuture. Since we sometimes need to know the complete list of contigs that are part of a connected component, the list implementation [3, section 22.2] is the better choice in this case.

Let $a = (c_x^p, c_y^q)$ be the arc we want to insert. The easier condition to test is whether at least one of $\{c_x^p, c_y^q\}$ is free: this can be done in $O(1)$ time. If the arc passes this test and if its ends belong to different connected components (which can also be determined in $O(1)$ time using the list implementation of the union-find data structure), the insertion is performed.

On the other hand, if the contigs are already connected, their relative orientation must be consistent with that suggested by a. In order to make this test in $O(1)$ time for each arc, we keep, at all times, the relative orientation of all contigs. We start the algorithm assuming that every contig has the same orientation (whether it is `LR` or `RL` is irrelevant). This information is updated as needed, namely when we insert an arc joining different connected components. For simplicity, we will use an example to explain how this is done. Assume that arc $a = \{c_1^r, c_2^r\}$ is inserted between two different connected components and that both c_1 and c_2 have orientation `LR`. Arc a clearly shows that c_1 and c_2 cannot have the same orientation in the genome. Therefore, one of the contigs must be changed to `RL`. Actually, if it is already linked to other contigs, the whole connected component is flipped: `RL` contigs become `LR` and vice-versa. Although any of the connected components could be flipped, we achieve better performance by choosing the one with fewer contigs. This ensures that at most $O(|N| \log |N|)$ contig flips will be necessary during the algorithm. (The entire flipping procedure can be interpreted as a subroutine of the *union* operation of the list-based union-find data strucuture.)

The third and most costly condition to test is whether the new arc creates a cycle. Since it would link c_x^p to c_y^q, we have to check whether there is a path in G that starts in c_y^p and ends in c_x^q. This requires $O(|A|)$ time in the worst case.

Once the graph G is built, MWP finds and reports the paths. Since the resulting graph is directed and acyclic, maximum weight paths can be found in $O(|A|)$ time (classic computer science result; see for example [3, section 25.4]). There will be at most $O(|N|)$ main paths, since they are all vertex-disjoint. Therefore, all $O(|N|)$ main paths can be found in $O(|N||A|)$ time. The algorithm finds at most one alternate path starting at each contig, so $O(|N||A|)$ is enough to find all alternate paths. Links can be easily found in $O(|A|)$ time.

All steps considered, $O(|L| \log |L| + |A|^2 + |N||A| + |N| \log |N|))$ is the worst case running time of MWP. However, since each node in G has at most four arcs incident to it (two in each end), $|A| = O(|N|)$, no matter how large A_0 or even L are. Therefore, the complexity of MWP can be rewritten as $O(|N|^2 + |L| \log |L|)$.

5 Final Remarks

As already remarked, the program described has been successfully used in a genome project [2]. The values for the various parameters were as follows:

- LMAX = 55 kbp
- LMIN = 30 kbp
- SMAX = 5 kbp
- minimum uniqueness required: 0.8

The scaffold program takes a few seconds on a Compaq DS20 workstation, on a genome project with around 100 contigs. This is insignificant compared to the time spent in assembly (measured in hours). The program is available from the authors upon request.

References

[1] J. K. Bonfield, K. F. Smith, and R. Staden. A new DNA sequence assembly program. *Nucleic Acids Research*, 23:4992–4999, 1995.

[2] ONSA consortium. *Xanthomonas axonopodis* pv *citri* genome project. www.lbi.ic.unicamp.br.

[3] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* MIT Press/McGraw-Hill, 1990.

[4] R. D. Fleishmann et al. Whole-genome random sequencing and assembly of *Haemophilus influenzae* Rd. *Science*, 269:496–512, 1995.

[5] P. Green. Phrap documentation. www.phrap.org.

[6] X. Huang and A. Madan. CAP3: A DNA sequence assembly program. *Genome Research*, 9:868–877, 1999.

[7] E. W. Myers et al. A whole-genome assembly of *Drosophila*. *Science*, 287:2196–2204, 2000.

[8] J. C. Roach, C. Boysen, K. Wang, and L. Hood. Pairwise end sequencing: a unified approach to genomic mapping and sequencing. *Genomics*, 26:345–353, 1995.

[9] Robert E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the Association for Computing Machinery*, 2:212–225, 1975.

6 Program Documentation

6.1 Command Line Options

The program is called `genscaff`. The command line for `genscaff` has the following basic format:

```
genscaff <input file> <libfile> [-a <gp|mwp>]
```

The first two parameters are the file containing describing the F/R pairs (the "intermediary file", described in Section 6.2) and the file containing information about the libraries (Section 6.3). The optional parameter `-a` selects the algorithm to be executed, `mwp` or `gp` (default is `gp`).

There are some additional optional parameters:

- `-mrs %f`: minimum uniqueness a read must have to be considered (default: 0.8);
- `-ht %d`: hybrid threshold (in base pairs). Clones whose maximum lengths are smaller than this value will be considered `sclone`'s; clones whose minimum lengths are larger than the threshold will be `lclone`'s (default: 10,000).
- `-maxaltdif %d`: maximum percentual difference allowed between a consistent alternate path and a main path (see Section 6.4.9). The parameter must be an integer between 1 and 100.

6.2 Intermediary File

This file is in extended DIMACS format. There are three types of lines, identified by their first character. Lines beginning with `c` are reserved for comments and may be ignored. Lines with `v` represent contigs (the vertices of the graph). Each such line contains only two fields:

`v` $\langle$*contig label*$\rangle$ $\langle$*contig length*$\rangle$

Links are represented in `a` lines, since they will constitute the arcs of the graph. Each line is made up by 14 fields,

`a` $\langle c_1\rangle$ $\langle c_2\rangle$ $\langle r_1\rangle$ $\langle r_2\rangle$ $\langle \texttt{R}|\texttt{L}\rangle$ $\langle \texttt{U}|\texttt{C}\rangle$ $\langle \textit{length}\rangle$ $\langle d_1\rangle$ $\langle d_2\rangle$ $\langle p_1\rangle$ $\langle s_1\rangle$ $\langle p_2\rangle$ $\langle s_2\rangle$ $\langle \textit{lib}\rangle$,

meaning:

- $\langle c_1\rangle$ and $\langle c_2\rangle$: contig labels;
- $\langle r_x\rangle$: string representing the read aligned with c_x;
- $\langle \texttt{R}|\texttt{L}\rangle$ and $\langle \texttt{U}|\texttt{C}\rangle$: together, these fields define the relative orientation of c_1 and c_2 induced by the link (see Section 3);
- $\langle \textit{length}\rangle$: lower bound on the length (in base pairs) of the clone;

- $\langle d_x \rangle$: distance (in bp) between the first aligned base at the $5'$ end of r_x with respect to c_x and the far end of c_x;
- $\langle p_x \rangle$ and $\langle s_x \rangle$: primary and secondary scores of r_x;
- $\langle lib \rangle$: string representing the name of the library to which the clone belongs.

6.3 Library File

This file contains information about the libraries mentioned in the intermediary file. There are just two types of lines, each identified by its first character: `c` lines contain comments and may be ignored; `l` lines describe the libraries. Each `l` line contains the name of the library (an arbitrary string) and three integers representing clone sizes (in bp): minimum, mean and maximum. Minimum and maximum sizes are used to determine whether a clone is an `lclone` or an `sclone`. The mean size is used to estimate the length of the paths found by the algorithms. A typical `l` line looks like this:

```
l c01 30000 40000 55000
```

According to this line, library `c01` contains clones with sizes varying from 30 kbp to 55 kbp; the mean clone size in this library is 40 kbp.

6.4 Output File

Each line in the output file has its structure defined by the first character. Some types of lines appear in several sections:

- `f` (free line): doesn't have a specific format and should be ignored by parsers (free lines with no text are often used to format the output — there are no blank lines);
- `s` (section line): marks the beginning of a section (the output is divided into sections). Everything after `s` represents the name of the section, as in

```
s INPUT DATA
```

- `d` (data line): has a well-defined format, but it depends on the context (section) in which the line appears;
- `t` (tagged data line): has a well-defined format and contains a globally unique tag right after the letter `t` indicating what piece of information is represented. For instance,

```
t contigs 987
```

  indicates that there are 987 contigs in the graph. Global uniqueness is useful to build simples parsers.

Other types of lines only occur in section `SCAFFOLDS`, which reports the scaffolds themselves: `c` (connected component), `p` (main path), `q` (alternate path), `a` (arc) and `l` (link). Their formats will be described in Section 6.4.9.

In some sections, the output file reports information on individual arcs. As already mentioned, an arc is made up by one or more clones. We use the clone with the highest uniqueness, considering the minimum of both ends, to represent the arc (ties are broken lexicographically). In case of hybrid arcs, an asterisk is appended to the name: while `AOQH6313D02` represents a simple arc, `AOQH6313D02*` represents a hybrid one.

The remainder of this section describes the individual sections of the output file.

6.4.1 `INPUT DATA`

This section summarizes the input information (files and parameters) used to build the scaffolds. It contains the following `t` lines:

- `file`: name of the input file, as it appeared in the commmand line;
- `lclones`: number of large clones read;
- `sclones`: number of small clones read;
- `clones`: total number of clones read;
- `contigs`: number of contigs read;
- `algorithm`: either `mwp` or `gp`;
- `minrelscore`: minimum uniqueness for an arc to be considered;
- `maxaltdif`: minimum percentual difference between an alternate and a main path (see Section 6.4.9).

Reports, in `t` lines, the number of large clones (`lclones`), the number of small clones (`sclones`), the total number of clones (`clones`) and the number of contigs (`contigs`) read. There are `t` lines describing all the parameters used in the execution: `algorithm` (`mwp` or `gp`), `input` (name of the input file), `minrelscore` (minimum uniqueness), `maxaltdif` (maximum alternate path difference; see 6.4.9).

6.4.2 `CLONE SIZES`

This section contains the information read from the library file. There is a `d` line for each library. A line contains four fields: a string representing the name of the library followed by the minumum, the mean and the maximum sizes of its clones. For example,

```
d 07 4000 5000 7000
```

indicates that clones in library `07` have sizes ranging from 4 kbp to 7 kpb and their average length is 5 kbp.

6.4.3 UNRELIABLE CLONES

This section presents clones that should not be trusted upon, since they have primary scores that are smaller than their secondary scores.

The first line in this section is a `t` line reporting the total number of unreliable clones (`unreliable`). Then, each such clone is reported in a `d` line with four fields: the name of the clone, its primary score, its secondary score and the label of the contig to which the clone is unreliably linked.

A typical section would look like this:

```
s UNRELIABLE CLONES
t unreliable 3
d A0JJ-0DG12-LA00 344 347 510
d A0QR5508D04 515 516 494
d A0JJ-0IA02-LA00 158 260 543
```

6.4.4 INCOMPATIBLE LINKS

This section presents all pairs of contigs linked to each other in more than one way (i.e., with different pairs of ends involved). The first line in the section is a `t` line reporting the total number of contig pairs with incompatible links (`incompatiblelinks`). A series of `d` lines follows, each containing at least four fields and at most six. The first two are the labels of the contigs; the others represent all possible ways of linking the contigs found. Consider the following example:

```
d 310 396 LL RL
```

This line refers to contigs c_{310} and c_{396}. While at least one clone indicates that c^l_{310} is linked to c^l_{396}, there is at least another one in the input file stating that the link is between c^r_{310} and c^l_{396}. Clearly, they can't both be correct.

6.4.5 ARC WEIGHTS

This section shows some statistics about the arcs created (regardless of whether they are actually inserted in the graph or not). The first line is a `t` line with three fields (other than the tag itself, `arcs`): the number of arcs created, the weight of the lighter arc and the weight of the heavier arc. This is followed by a series of `d` lines, each with two fields: the first represents a weight and the second the number of arcs with such weight. A typical output looks like this:

```
s ARC WEIGHTS
t arcs 585 1.0 4.0
d 1.0 426
d 1.5 17
d 2.0 130
d 2.5 8
```

```
d 3.0 3
d 4.0 1
```

In this example, there are 426 arcs with weight 1.0, 17 with weight 1.5, and so on. Note that if there is no arc with a given weight in the range, the corresponding `d` line may be omitted (this is the case of weight 3.5 in the example).

6.4.6 HYBRID ARCS

This section reports the number of hybrid arcs, arcs that contain both `sclone`'s and `lclone`'s. Consistent and inconsistent hybrid arcs also have their number reported separately. The information is presented in `t` lines, with tags `hybridarcs`, `consistenthybrid` and `inconsistenthybrid`. A typical section is:

```
s HYBRID ARCS
t hybridarcs 18
t consistenthybrid 13
t inconsistenthybrid 5
```

6.4.7 ARCS NOT INSERTED

This section lists all arcs not inserted into the graph alongside with the reason why they were kept out. A typical line looks like this:

```
d A0JJ1388B03* 390 477 degree
```

There are four fields: the first is the name of the arc, the next two are the contigs it connects, and the last one is the reason why the arc could not be inserted. There are five possible reasons:

- `degree`: at least one end of the arc was already connected to other arcs;
- `score`: the uniqueness of the arc is lower than the mininum threshold (a fifth field presents the score of the rejected arc);
- `cycle`: the arc would create a cycle if inserted[1];
- `orientation`: the relative orientation of the ends induced by the arc is incompatible with the orientation induced by previously inserted arcs;
- `weight`: both ends of the arc were already connected to heavier arcs.

Only `score` and `cycle` can occur in both algorithms; `degree` applies exclusively to GP, while `orientation` and `weight` may happen only in MWP.

[1]Algorithm MWP adds an `f` line right after the `d` line to list the vertices of the cycle. GP doesn't do this, since it can detect cycles using a union-find data structure; finding the actual cycles would be too costly.

6.4.8 INSERTION STATISTICS

This section summarizes, in `t` lines, the information presented in the previous section. It reports how many arcs were not inserted due to each of the five types of problems mentioned. The total number of arcs inserted and not inserted is also shown. All quantities are expressed both as absolute values (first field after the tag) and as percentages (third field after the tag). A typical output would be:

```
s INSERTION STATISTICS
t inserted 218 29.1
t notinserted 531 70.9
t weight 0 0.0
t orientation 0 0.0
t score 131 17.5
t cycle 1 0.1
t degree 399 53.3
```

Note that all five types of problems are always listed, regardless of the algorithm.

6.4.9 SCAFFOLDS

This is the section that actually presents the scaffolds. It begins with a `t` line reporting the number of connected components in the graph (`components`). Then the components themselves are listed in nonincreasing order of size (number of vertices/contig ends). Components with no arcs (i.e., only one contig) are omitted.

The description of a single connected component can be rather lengthy. It starts, with a `c` line with three fields: the label of the connected component (a sequential number starting at one), the number of contigs, the number of arcs and the number of main paths built. An example:

```
c 2 117 151 31
```

This connected component (component 2) has 117 vertices (contig ends) and 151 arcs; 31 main paths were found by the algorithm (note that for GP the third field will always be 1).

This line is followed by a description of the paths in the connected component, listed in nondecreasing order by weight. A typical path looks like this:

```
p 2 6 41.0 330993
a 497 U A0UV5307G09* 8.5
a 431 U A0JJ-1BF10-LA00 2.0
a 432 U A0QH6313D02* 5.0
a 484 U A0AC6408A07 5.5
a 397 C A0AM1319C05 1.0
a 424 U A0JJ-1CC12-LA00 2.0
```

```
a 509 U A0QR6366E09* 11.0
a 532 U A0QR5323G03 4.0
a 313 C A0QR6330C07 2.0
a 401 C END 0.0
l 525 1 A0JJ-0IH08-LA00 497 6
l 397 6 A0JJ-0AB01-LA00 511 5
t 155068 155410 C
a 497 U A0JJ-1AH02-LA00 2.0
a 499 C A0JJ-0EG12-LA00 2.0
a 484 U END 0.0
```

Main path According to the `p` line, this is the sixth heaviest path in its connected component (component 2). Its weight is 41.0 and its estimated length is 330,993 bp.

The first field in each `a` line represents the label of a contig in the path. This particular one is made up by 10 contigs ($c_{497}, c_{431}, c_{432}, \ldots, c_{401}$, in this order). The second field is either `C` or `U`, depending on whether the contig is complemented or not in the path (in the example, c_{397}, c_{313}, and c_{401} are complemented). The name and the weight of the arc linking a pair of contigs are reported in the third and fourth fields, respectively. For instance, c_{432} and c_{484} are linked by arc `A0QH6313D02`, whose weight is 5.0. Note that the last `a` line is special; the word `END` replaces the arc name and marks the last contig of the path (to make parsing easier, a meaningless `0.0` is placed on the fourth field).

Links After the path itself is printed, its links to previously reported main paths are listed in `l` lines. In our example, two links are reported. Arc `A0JJ-0IH08-LA00` links c_{525}, which belongs to path 1, to c_{497}, which belongs to the current path (path 6). The other link is between c_{397} in path 6 and c_{511} in path 5. The order in which the contigs appear is important. Were c_{525} to be inserted in path 6, it would appear *before* c_{497}. On the other hand, c_{511} would come *after* c_{397} in this path.

Alternate paths Finally, alternate paths (if any) are reported. Each one is introduced by a `q` line, which has three fields. The first two are the length of the alternate path (ℓ_a) and the length of the corresponding portion of the original main path (ℓ_o). The last one is either `C` (consistent) or `I` (inconsistent), depending on how ℓ_o and ℓ_a relate to each other. If they differ by no more than 20% (ℓ_a being the base value), the path is said to be consistent; otherwise, the path will be inconsistent. (Actually, 20 is just the default percentage; the actual value, `maxaltdif`, is user-defined.) Each `t` line is followed by `a` lines describing the path (exactly like in the main path).

In our example, only one alternate path was found. Since its length is 155,068 bp and it corresponds to 155,410 bp in the original path, the path is consistent. The path contains three contigs, c_{497}, c_{499}, and c_{484}. Note that the first and the last contigs of an alternate path must also belong to the original path, while all others must not.