class(vf_0)	
2	INC
3	vf_0=3, vf_1=3
4	vf_0=4, vf_1=3

Table 11
Possible results of a topological scan for final case S11.1.1

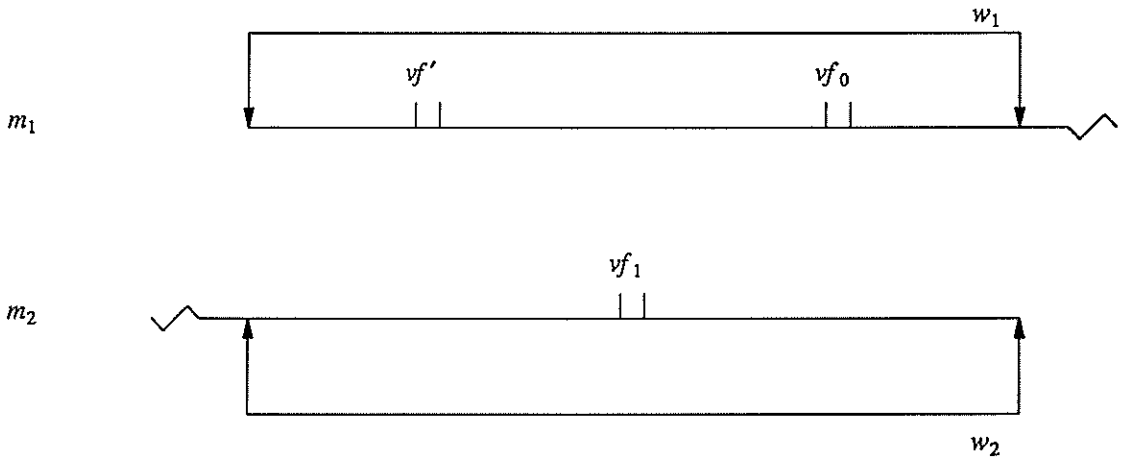

Figure 58: A sketch of final case S11.1.2

class(vf_0) → class(vf') ↓	2	3	4
2	IMP	vf_0=3, vf_1=2, vf'=2	INC
3	vf_0=2, vf_1=2, vf'=3	vf_0=3, vf_1=3, vf'=3	vf_0=4, vf_1=3, vf'=3
4	INC	vf_0=3, vf_1=3, vf'=4	IMP

Table 12
Possible results of a topological scan for final case S11.1.2

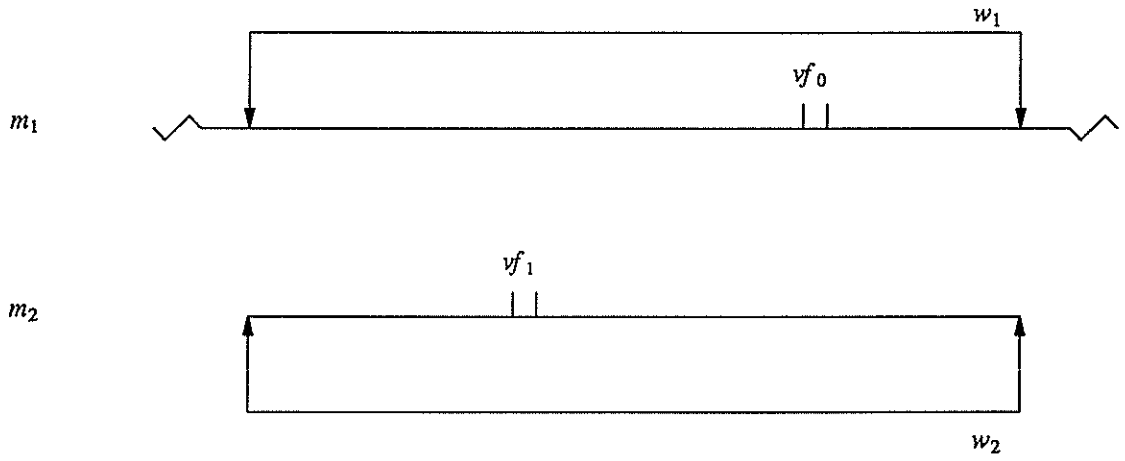

Figure 59: A sketch of final case S11.2.1

$|ml| = |w_2|$

class(vf_0)	
2	INC
3	vf_0=3, vf_1=3
4	vf_0=4, vf_1=3

Table 13
Possible results of a topological scan for final case S11.2.1

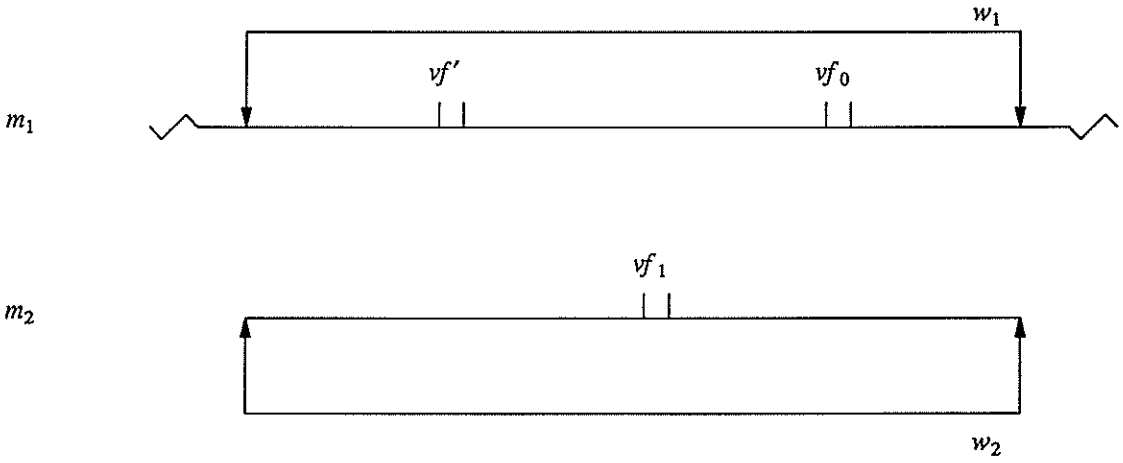

Figure 60: A sketch of final case S11.2.2

$|ml| = |w_2|$

class(vf_0) → class(vf') ↓	2	3	4
2	IMP	vf_0=3, vf_1=2, vf'=2	INC
3	vf_0=2, vf_1=2, vf'=3	vf_0=3, vf_1=3, vf'=3	vf_0=4, vf_1=3, vf'=3
4	INC	vf_0=3, vf_1=3, vf'=4	vf_0=4, vf_1=3, vf'=4

Table 14
Possible results of a topological scan for final case S11.2.2

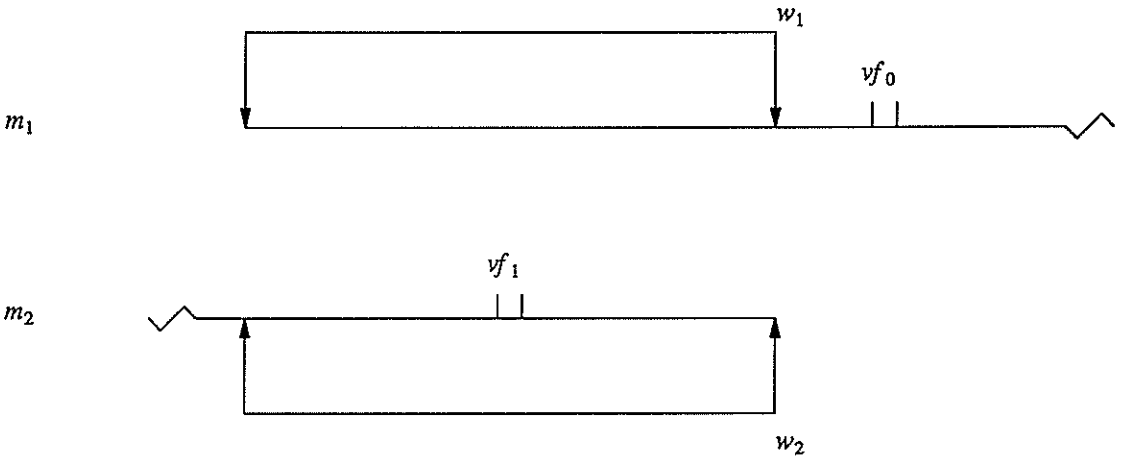

Figure 61: A sketch of final case S12.1.1

class(vf_0)	
1	vf_0=1, vf_1=3

Table 15
Possible results of a topological scan for final case S12.1.1

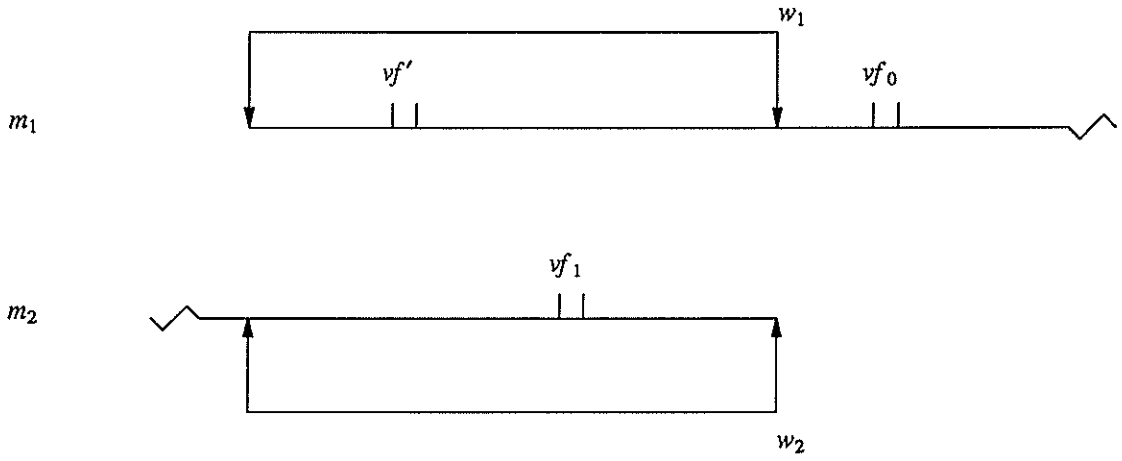

Figure 62: A sketch of final case S12.1.2

class(vf')	
2	INC
3	vf_0=1, vf_1=3, vf'=3
4	vf_0=1, vf_1=3, vf'=4

Table 16
Possible results of a topological scan for final case S12.1.2

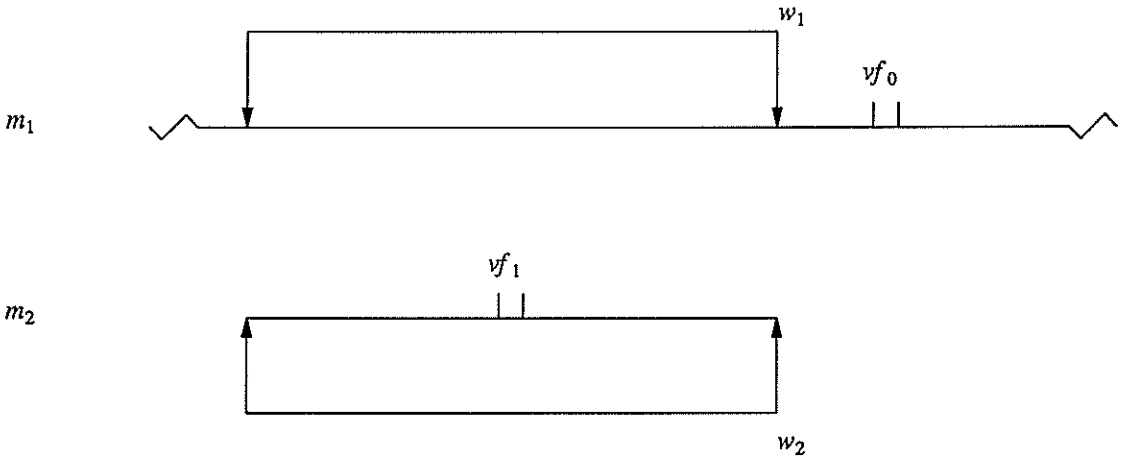

Figure 63: A sketch of final case S12.2.1

$|ml| = |w_2| - 1$

class(vf_0)	
1	vf_0=1, vf_1=3

Table 17
Possible results of a topological scan for final case S12.2.1

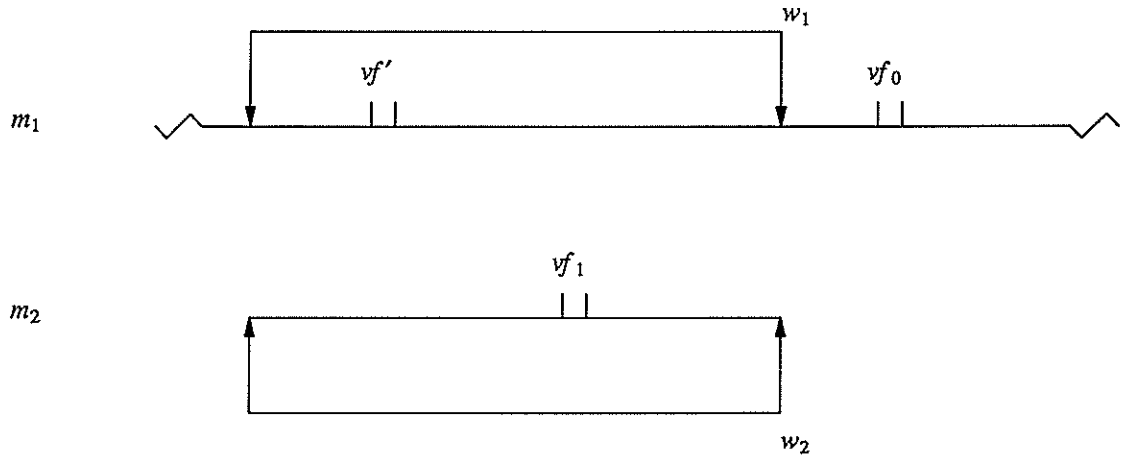

Figure 64: A sketch of final case S12.2.2

$|ml| = |w_2|$

class(vf')	
2	INC
3	vf_0=1, vf_1=3, vf'=3
4	vf_0=1, vf_1=3, vf'=4

Table 18
Possible results of a topological scan for final case S12.2.2

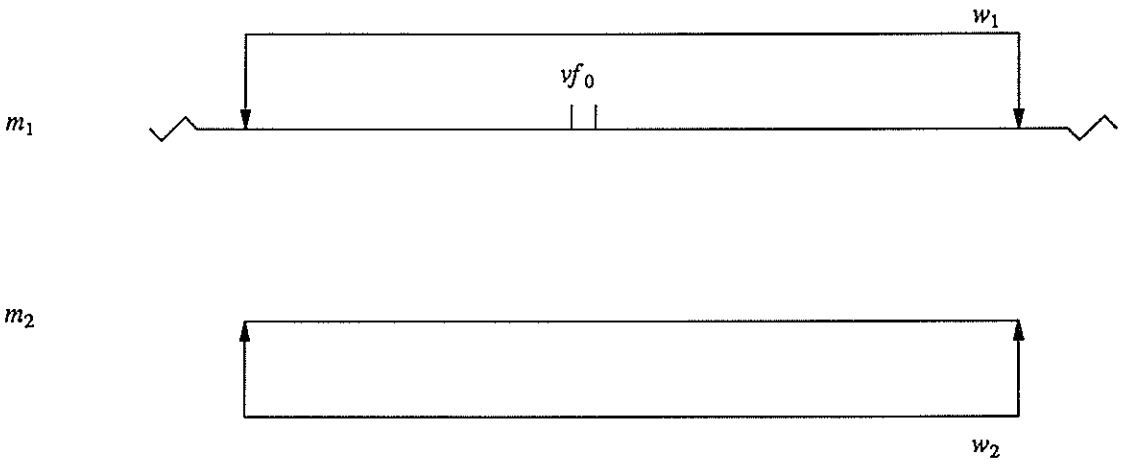

Figure 65: A sketch of final case S31.1.1

$|ml| = |w_2|$

class(vf_0)	
2	IMP
3	vf_0=3
4	INC

Table 19
Possible results of a topological scan for final case S31.1.1

The next section uses the information contained in the tables just presented to construct a larger table useful to FIX.

3.5. Constructing the Split Table

The tables presented in the previous section can be used to construct another table (called the split table) where one can use the observable properties of two maps and determine the class and length of a virtual fragment that is likely to contain a fragment matching mistake.

The construction of the split table begins with an empty spilt table. The table corresponding to each final case z in the limited enumeration (i.e., the tables in §3.4) is examined. The entries in the table for z are grouped according to the number of class 3 fragments in each map. For each group of entries, the class y (or set of classes y) of vf_0 (the fragment containing the mistake) is determined. Then an entry in the split table is added, which indicates that if the visible characteristics of two maps matches those for this group of entries, then z is a possible underlying reality. Thus, each class y fragment (or each fragment of a class in the set y) might contain the fragment matching mistake and thus is a candidate for splitting. When this is done for each case in the limited enumeration, the split table is complete.

Final case S11.2.2 (see Table 14) is used to illustrate the construction of the split table. Entries (1,1), (1,3) and (3,1) are either impossible or result in incorporation and thus are not used in the construction of the split table. Because class 3 fragments are important, the remaining entries are grouped based upon the number of class 3 virtual fragments in each map. Entries (1,2) and (2,1) form the first group of interest because each contains one class 3 fragment in m_1 and none in m_2. Entries (2,3) and (3,2) form the second group of interest because each contains one class 3 fragment in m_1 and m_2. Entry (2,2) forms the third group of interest because it is the only entry with two class 3 virtual fragments in m_1 and one in m_2. Entry (3,3) forms the fourth and final group of interest because it is the only entry with no class 3 fragments in m_1 and one in m_2. (Note that the order in which these groups are presented is irrelevant.)

Consider the first group, which contains entries (1,2) and (2,1). In entry (1,2), vf_0 is a class 3 fragment. In entry (2,1), vf_0 is a class 2 fragment. So if one is given two maps m_1 and m_2 and the results of a topological scan and observes that

(1) m_2 nearly assimilates into m_1,

(2) there is exactly one class 3 fragment in m_1,

(3) there are no class 3 fragments in m_2 and

(4) the size of a maximum size non-topological matchlist between w_1 and w_2 is equal to the number of fragments in w_2,

then it is possible that the underlying reality corresponds to entry (1,2) or (2,1) of the table of case S11.2.2. If this is true, the fragment matching mistake can be fixed by splitting (1) the only class 3 fragment in w_1 or (2) all class 2 fragments in w_1 of length similar to the class 3 fragment.

The entry in the split table corresponding to the first group is illustrated in Figure 66. The split table is read by starting at the outside and moving into the parts of the table that match the observable characteristics of the two maps and the topological scan results. At some point, a final case is reached and a set of fragments to split is determined, usually by specifying a class and a length. All virtual fragments of that class that are within range of the specified length belong in the set.

Next consider the second group of entries, which contains entries (2,3) and (3,2). In entry (2,3), vf_0 is a class 4 fragment. In entry (3,2), vf_0 is a class 3 fragment. So if one is given two maps m_1 and m_2 and the results of a topological scan and observes that

(1) m_2 nearly assimilates into m_1,

(2) there is exactly one class 3 fragment in m_1,

(3) there is exactly one class 3 fragment in m_2 and

(4) the size of a maximum size non-topological matchlist between w_1 and w_2 is equal to the number of fragments in w_2,

then it is possible that the underlying reality is entry (2,3) or (3,2) of the table of case S11.2.2. If this is true, the fragment matching mistake can be fixed by splitting (1) the only class 3 fragment in m_1 or (2) all class 4 fragments in w_1 of similar length as the class 3 fragment.

The split table with the addition of an entry for the second group is illustrated in Figure 67. The third and fourth group of entries are examined in a similar manner. The split table after examining all four groups of entries is illustrated in Figure 68.

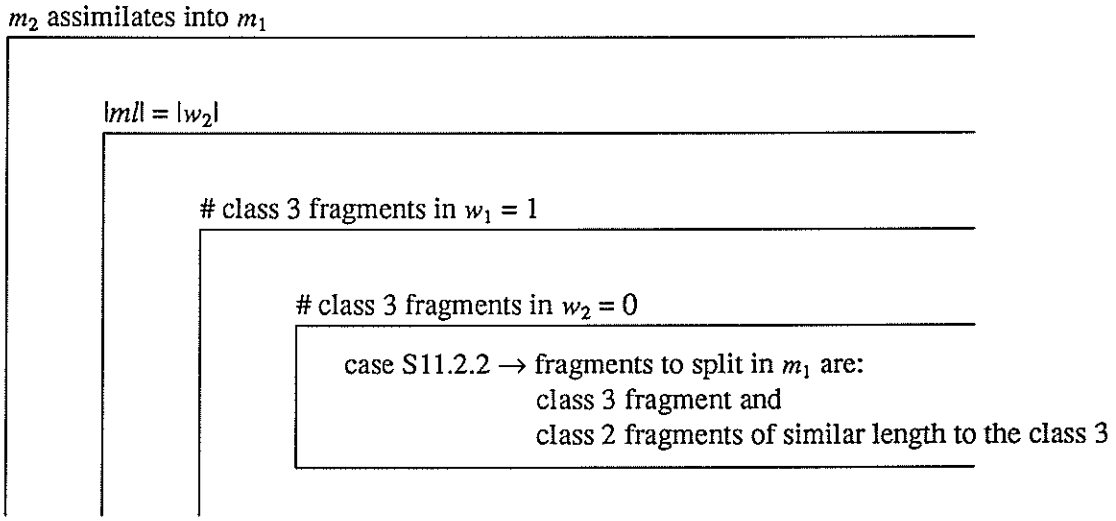

Figure 66: The split table after examining the first group of entries

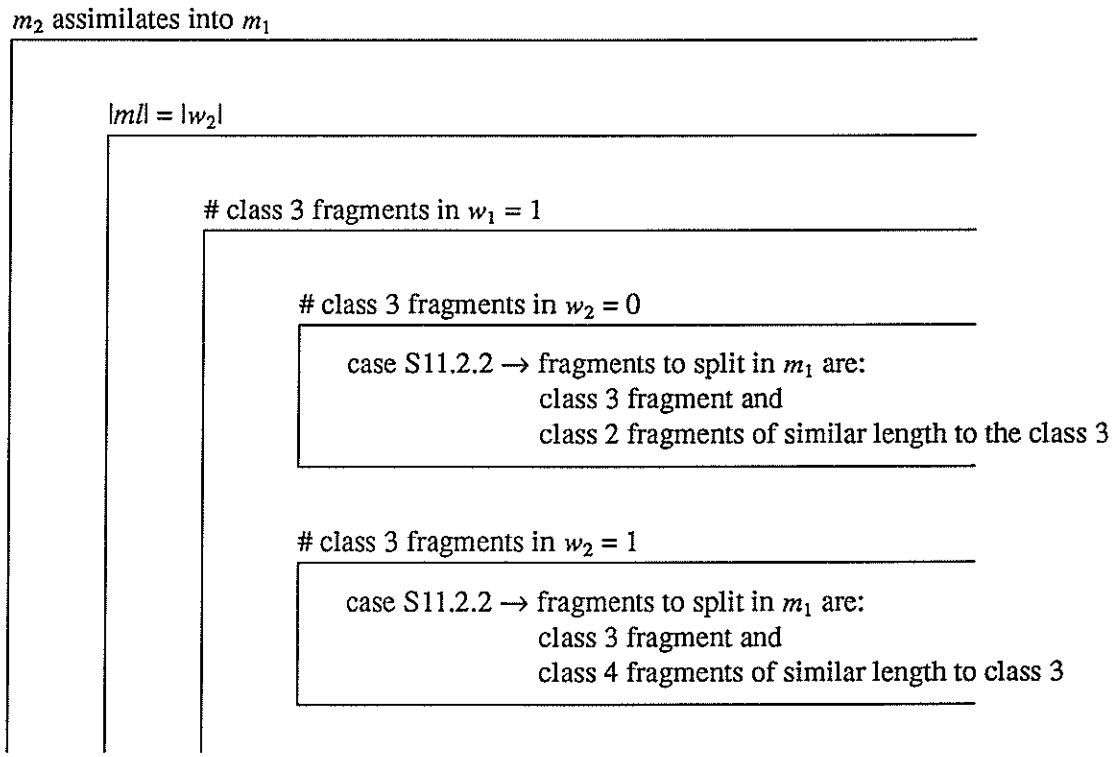

Figure 67: The split table after examining the second group of entries

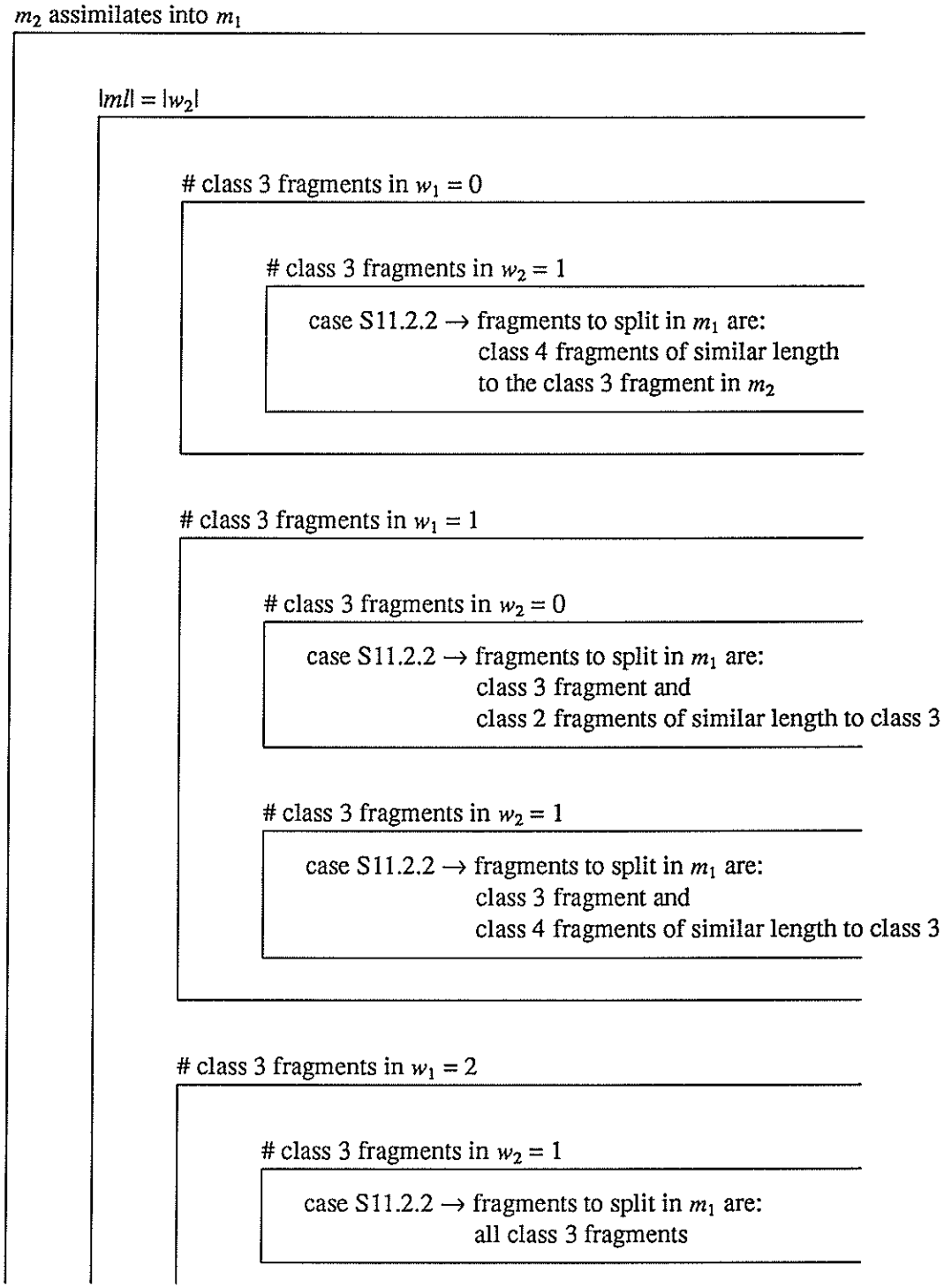

Figure 68: The split table after examining final case S11.2.2

After each final case in the limited enumeration is examined in this way, a complete version of the split table (which is not presented in this report) has been constructed. The conceptual approach that has been applied here is to enumerate all likely underlying realities and determine what would be observed given each of these realities. Then given a specific observation, we back-infer what the possible underlying

realities might have been, given what was observed. Since several different realities may produce the same observable results, there may be ambiguity about the corresponding underlying reality, given a specific observed result. The ramification of this is that a number of cases may end up in the same location in the split table. If this occurs, then the set of fragments to split is the union of the sets for each case.

In §4, another analysis, similar to the one presented in this section, is given. However, the analysis of §4 focuses on finding the location of fragment matching mistakes that result from the separation of real fragments that in reality should be matched, and thus must be fixed with a combine.

4. Detecting Fragments to Combine

In §1.3, the manner in which two fragments that do not correspond to the same stretch of DNA in the genome can be incorrectly matched was described. It is also possible that two fragment that do correspond to the same stretch of DNA in the genome are not matched, even though they should be. This usually happens as a result of a nearby fragment matching mistake of the more common variety creating pushed fragments that dislocate fragments that would otherwise match. Fragment matching mistakes of this second variety require the combining of virtual fragments in order to repair the map.

In this section, another component of FIX, very similar to the component described in §3, is described. As in §3, this component uses information that can be obtained from the maps and information about how a fragment matching mistake affects a map to make educated guesses about which virtual fragments may be part of a fragment matching mistake. The difference is that this component is concerned with fragment matching mistakes that require a combine to repair. The description of this component will be brief, presenting only the differences from the previous component and the results of the case analysis.

4.1. Differences in the Limited Enumeration of Possible Underlying Realities

As in §3, it would be ideal to completely enumerate and describe each underlying reality that leads to a fragment matching mistake (in this case, one that could be corrected by a combine). But again, this is intractable. So as in §3, a limited enumeration of the possible underlying realities is performed.

This limited enumeration is based upon the assumption that the underlying reality contains a single fragment matching mistake, one which requires a combine to repair. More precisely, the following conditions (illustrated in Figure 69) are assumed to exist.

(1) There is a genomic fragment *gf*.

(2) There is a clone c_1 containing a real fragment rf_1 that corresponds to *gf*.

(3) There is a clone c_2 containing a real fragment rf_2 that corresponds to *gf*.

(4) There is a map m_1, containing c_1 and c_2, which is correct except that rf_1 and rf_2 are not matched and are part of virtual fragments vf_1 and vf_2, respectively.

(5) There is a map m_2 which is correct and may contain a virtual fragment vf_0, which corresponds to *gf*.

Note that these conditions are different than the conditions used in §3. Given these assumptions, m_1 and m_2 are likely to have significant overlap. However, they will not incorporate because vf_0 cannot pair with both vf_1 and vf_2. If topological scanning were applied to m_1 and m_2, it is likely that an ATVML of

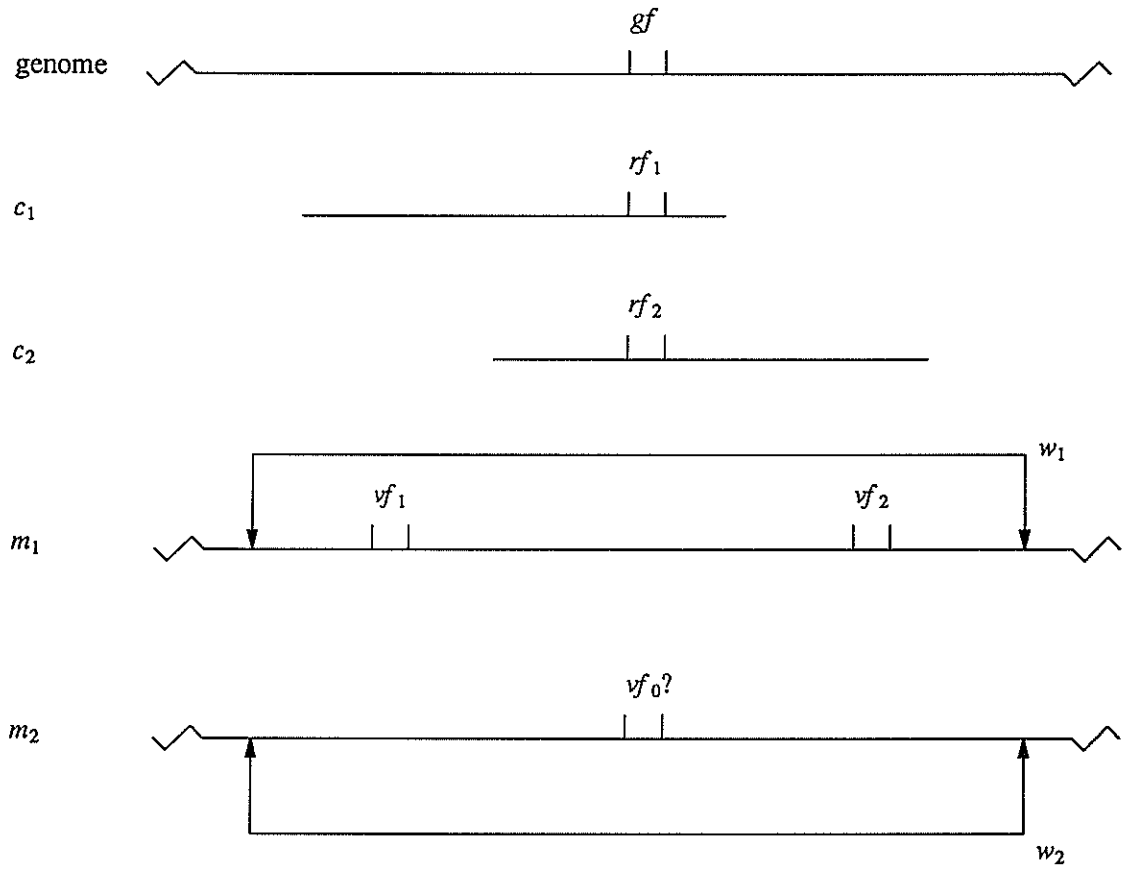

Figure 69: Illustration of the assumptions for the limited enumeration

significant size would be constructed. Let w_1 be the minimum size window of m_1 with respect to that ATVML. Let w_2 be the minimum size window of m_2 with respect to that ATVML.

As in §3, the details important to the enumeration are broken into several levels; in this case there are two levels. The first level is concerned with the position of m_2, w_1 and w_2 with respect to gf, vf_0, vf_1 and vf_2. More specifically, the following conditions are important.

(kk_1) m_2 contains vf_0.

(kk_2) w_2 contains vf_0.

(kk_3) w_1 contains vf_1.

(kk_4) w_1 contains vf_2.

Each condition may be true or false. Thus, there are 16 possible combinations of kk_1 through kk_4. As in §3, many of these combinations are impossible, very unlikely, of no interest or are symmetric to other cases.

In the second level of enumeration, each first-level case that is still of interest is broken into subcases, based upon the assumed overlap relationship of m_1 and m_2.

4.2. Limited Enumeration of Possible Underlying Realities

4.2.1. The First Level of Enumeration

The 16 first-level cases of the limited enumeration are presented in Table 20. In §3, some theorems and observations eliminated many of the first-level cases. Likewise, many of the first-level cases in Table 20 can be eliminated using the theorems below, which are given without proof. (The proofs are very similar to those in §3.)

Theorem 7: $kk_2 \rightarrow kk_1$

Theorem 8: $kk_1 \rightarrow kk_2$ is highly likely

Theorem 9: kk_3 and $kk_4 \rightarrow kk_1$ is highly likely

Theorem 7 eliminates cases C9-C12. Theorem 8 eliminates cases C5-C8. Theorem 9 eliminates cases C9 and C13. Thus, the first level of enumeration has four major cases (C1, C2, C4 and C14) which are of interest in the second level of enumeration.

Case	kk_1	kk_2	kk_3	kk_4	Remark
C1	T	T	T	T	OK
C2	T	T	T	F	OK
C3	T	T	F	T	Symmetric to C2
C4	T	T	F	F	OK
C5	T	F	T	T	Unlikely
C6	T	F	T	F	Unlikely
C7	T	F	F	T	Unlikely
C8	T	F	F	F	Unlikely
C9	F	T	T	T	Impossible
C10	F	T	T	F	Impossible
C11	F	T	F	T	Impossible
C12	F	T	F	F	Impossible
C13	F	F	T	T	Unlikely
C14	F	F	T	F	OK
C15	F	F	F	T	Symmetric to C14
C16	F	F	F	F	Of No Interest

Table 20
The first-level cases for Combine

4.2.2. The Second Level of Enumeration

As in §3, the second level of enumeration involves breaking down the remaining first-level cases based upon the assumed overlap relationship of m_1 and m_2. Also as in §3, not all overlap relationships are possible for certain first-level cases. Table 21 summarizes the status of the second-level cases.

Unlike §3, there is no third level of enumeration to perform. The tree illustrated in Figure 70 summarizes the limited enumeration. Again, each final case is given a label for future reference.

4.3. Results of the Analysis of All Cases

This section summarizes the results of performing an analysis like the one in §3.3 for each possible underlying reality identified by the limited enumeration. The format of the summary is the same as that used in §3.4.

Overlap Relationship → Case ↓	or_1	or_2	or_3	or_4
C1	OK	SYM	OK	OK
C2	OK	IMP	IMP	OK
C4	IMP	IMP	IMP	OK
C14	OK	IMP	IMP	OK

Table 21
The second-level cases for Combine

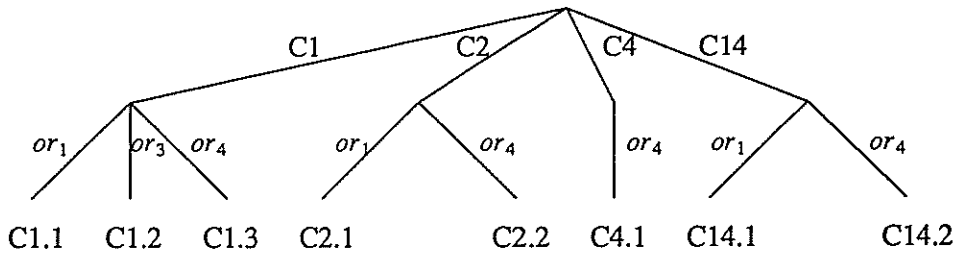

Figure 70: Summary of the limited enumeration

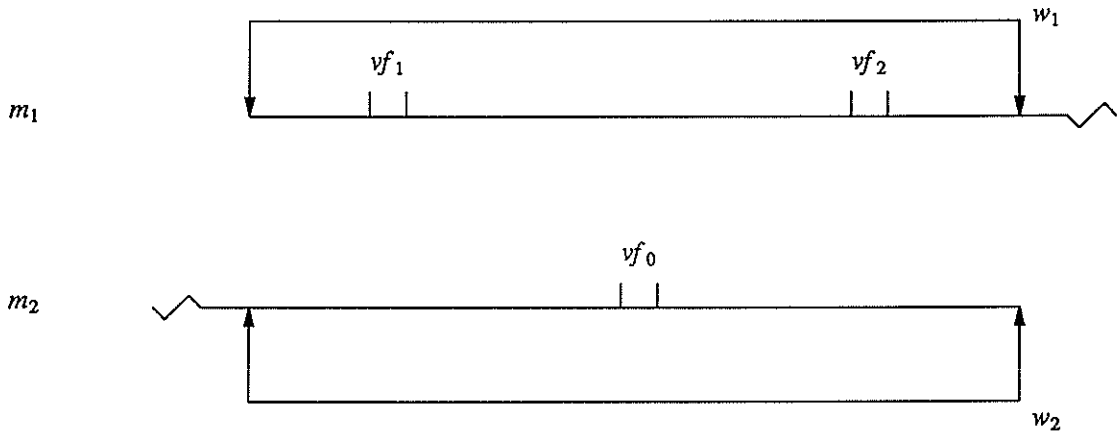

Figure 71: A sketch of final case C1.1

class(vf_2) → class(vf_1) ↓	2	3	4
2	IMP	vf_0=2, vf_1=2, vf_2=3	INC
3	vf_0=2, vf_1=3, vf_2=2	vf_0=3, vf_1=3, vf_2=3	vf_0=3, vf_1=3, vf_2=4
4	INC	vf_0=3, vf_1=4, vf_2=3	IMP

Table 22
Possible results of a topological scan for final case C1.1

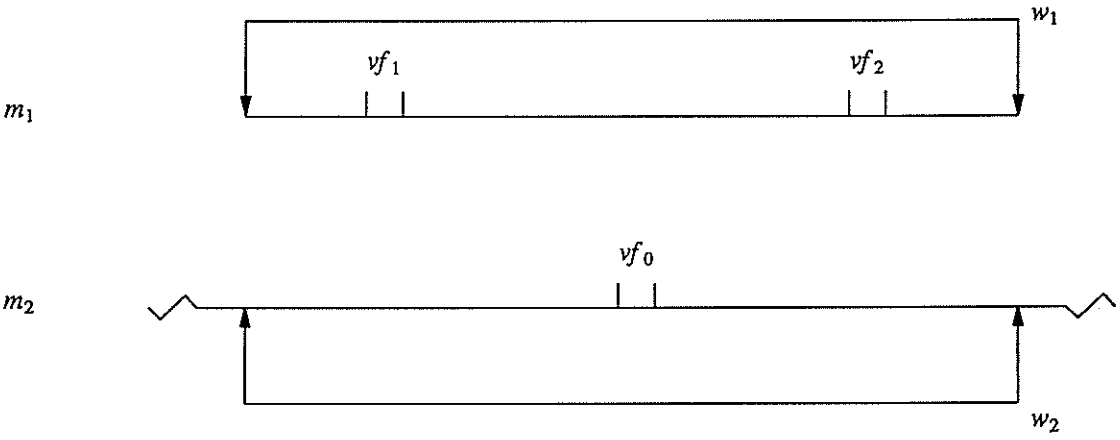

Figure 72: A sketch of final case C1.2

class(vf_2) → class(vf_1) ↓	2	3	4
2	IMP	vf_0=2, vf_1=2, vf_2=3	IMP
3	vf_0=2, vf_1=3, vf_2=2	vf_0=3, vf_1=3, vf_2=3	IMP
4	IMP	IMP	IMP

Table 23
Possible results of a topological scan for final case C1.2

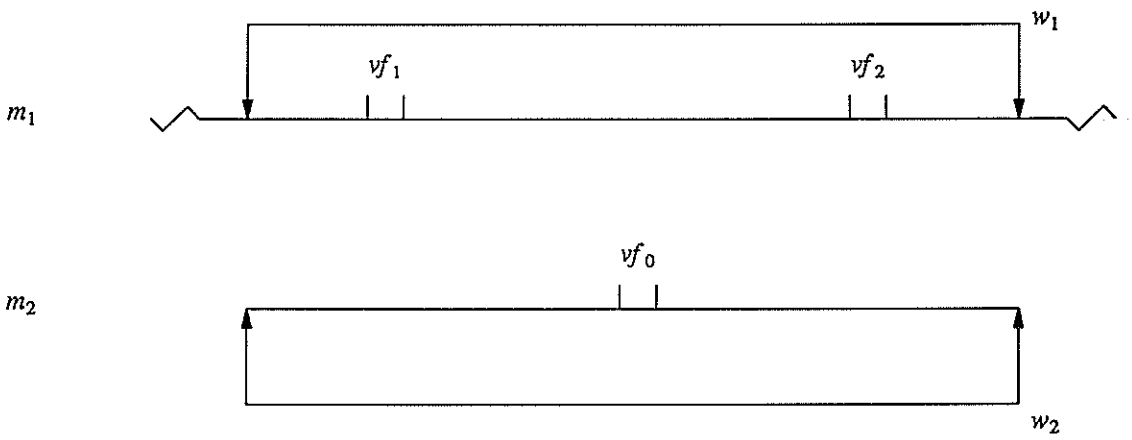

Figure 73: A sketch of final case C1.3

class(vf_2) → class(vf_1) ↓	2	3	4
2	IMP	vf_0=2, vf_1=2, vf_2=3	INC
3	vf_0=2, vf_1=3, vf_2=2	vf_0=3, vf_1=3, vf_2=3	vf_0=3, vf_1=3, vf_2=4
4	INC	vf_0=3, vf_1=4, vf_2=3	vf_0=3, vf_1=4, vf_2=4

Table 24
Possible results of a topological scan for final case C1.3

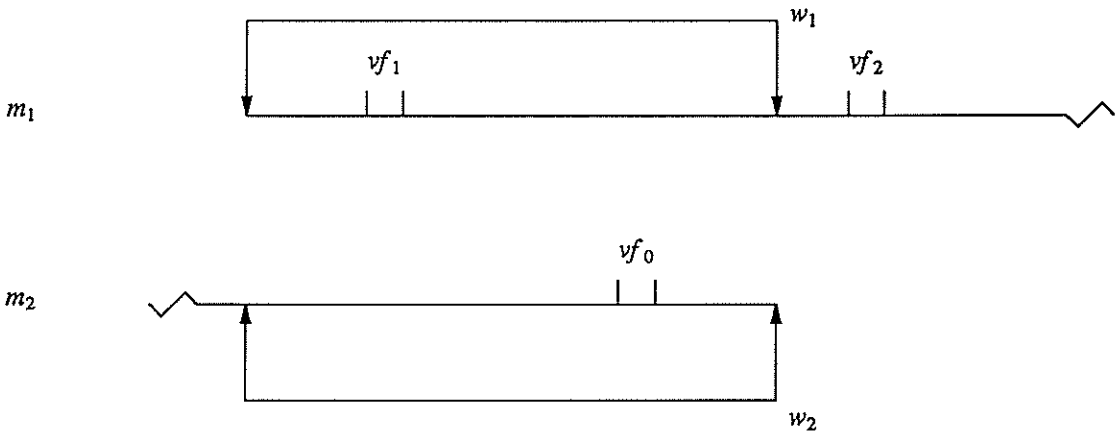

Figure 74: A sketch of final case C2.1

class(vf_1)	
2	INC
3	vf_0=3, vf_1=3
4	vf_0=3, vf_1=4

Table 25
Possible results of a topological scan for final case C2.1

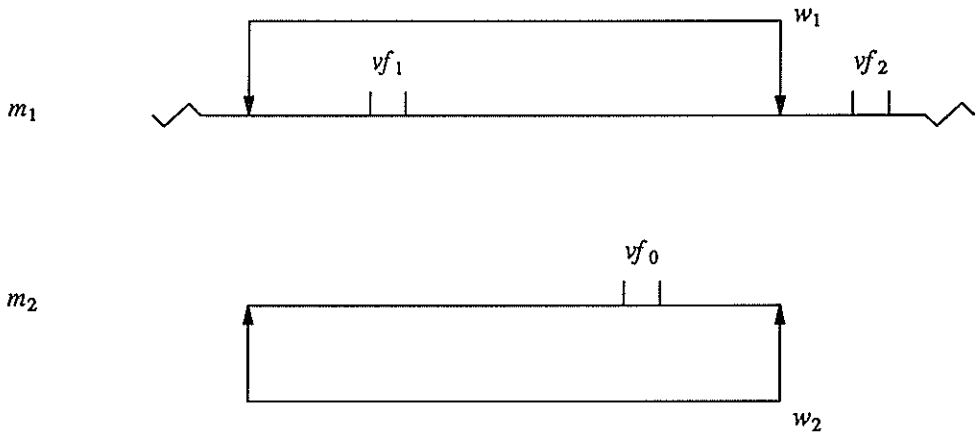

Figure 75: A sketch of final case C2.2

class(vf_1)	
2	INC
3	vf_0=3, vf_1=3
4	vf_0=3, vf_1=4

Table 26
Possible results of a topological scan for final case C2.2

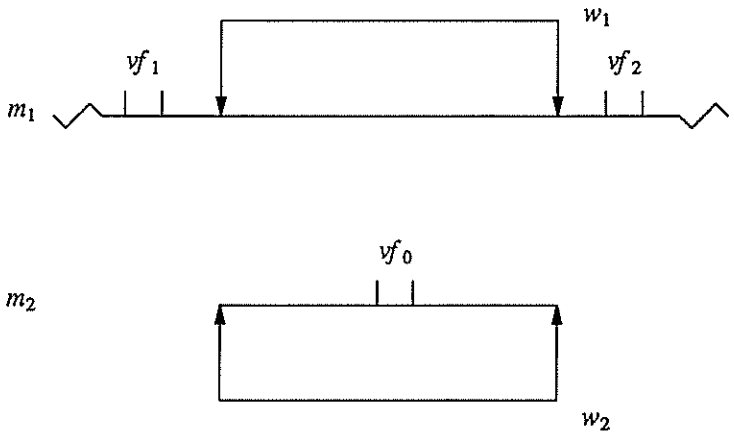

Figure 76: A sketch of final case C4.1

class(vf_0)	
2	IMP
3	vf_0=3
4	INC

Table 27
Possible results of a topological scan for final case C4.1

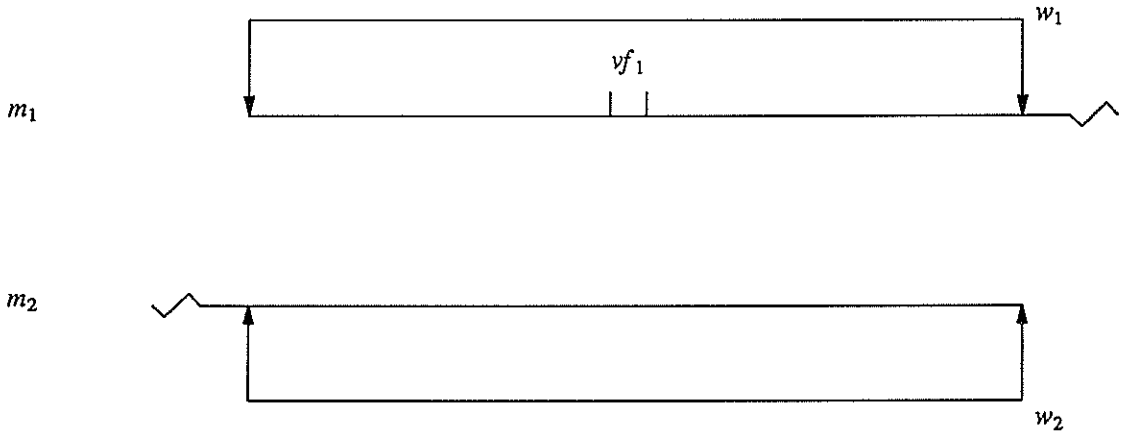

Figure 77: A sketch of final case C14.1

class(vf_1)	
2	IMP
3	vf_1=3
4	INC

Table 28
Possible results of a topological scan for final case C14.1

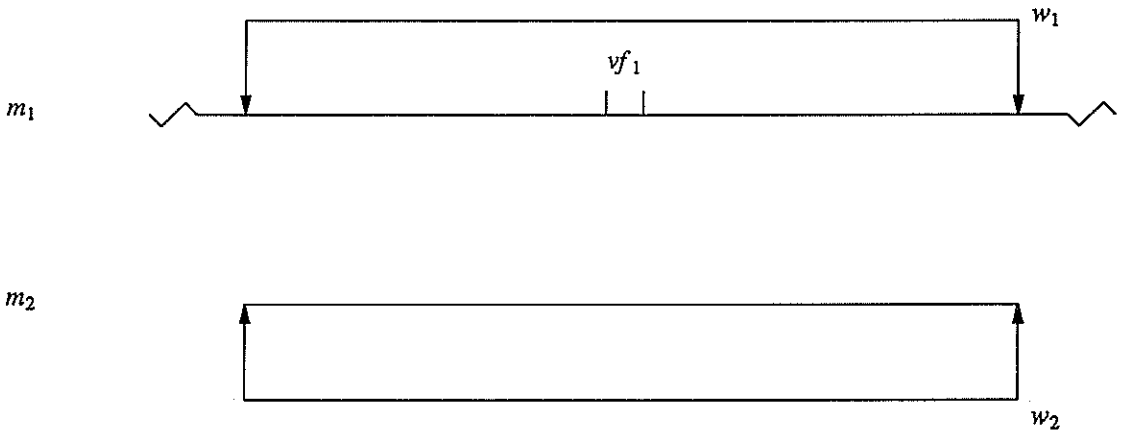

Figure 78: A sketch of final case C14.2

class(vf_1)	
2	IMP
3	vf_1=3
4	INC

Table 29
Possible results of a topological scan for final case C14.2

4.4. Constructing the Combine Table

The split table constructed in §3.5 organizes the results of the case analysis performed in a manner useful to FIX. A similar table, the **combine table**, can be constructed from the results of the analysis presented in §4.3 in a similar manner. However, there are a few differences between the split table and the combine table.

From the split table, one could determine the class of a fragment to split. However, performing a combine requires two fragments, which may or may not be of the same class. Thus, the combine table must determine *two* classes x_1 and x_2, which indicates that one should combine a class x_1 fragment with a class x_2 fragment.

Figure 79 illustrates the entries in the combine table corresponding to case C1.2 (see Table23). The first entry corresponds to entries (1,2) and (2,1) of Table 23. The fragments that must be combined are vf_1 and vf_2. In these entries, vf_1 is a class 2 fragment and vf_2 is a class 3 fragment, or vice-versa. Thus, one must attempt to combine the class 3 fragment with each class 2 fragment of similar length in order to determine which combinations might fix the map. The second entry of the combine table for case C1.2 corresponds to entry (2,2) of Table 23. Here vf_1 and vf_2 are both class 3. (Recall that there can be no more than two class 3 fragments in a map.) Thus, the class 3 fragments of w_1 should be combined in order to fix the map.

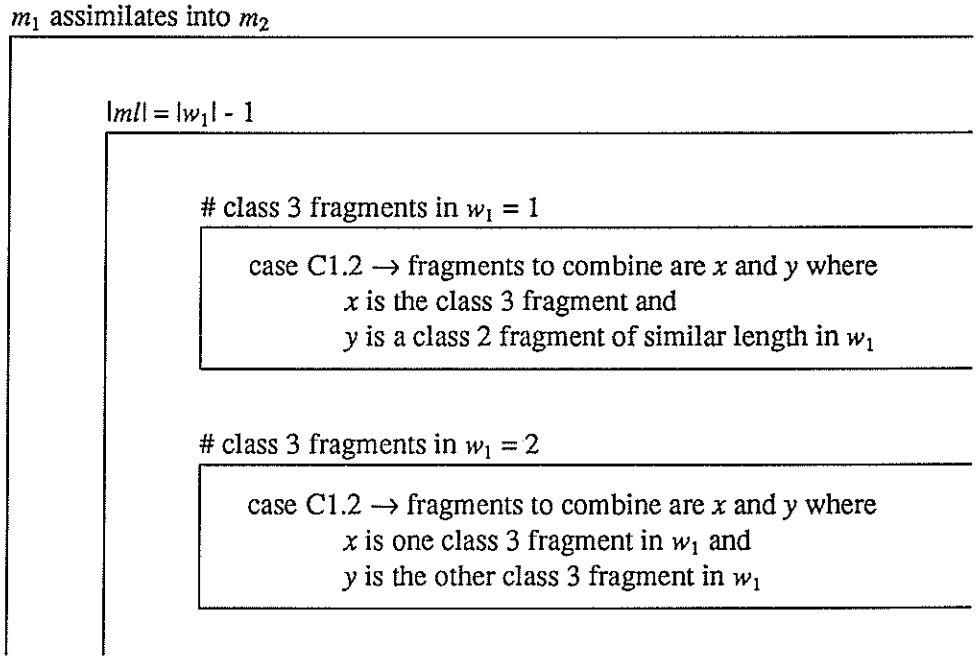

Figure 79: A portion of the combine table

5. Correcting Two Maps

This section describes how the components described in the previous sections are put together to form the FIX algorithm. FIX combines topological scanning (described in §2), the split and combine tables (described in §3 and §4) and the RSA. The pseudocode for the FIX algorithm and a brief description of each function is given in the remainder of this section.

5.1. The FIX Algorithm

The top level function of the FIX algorithm is fix, whose pseudocode is presented in Figure 80. fix takes as input two windows w_1 and w_2. Let m_1 be the map on which the window w_1 is defined. Let m_2 be defined similarly for w_2. fix returns a set of triples of the form (m_1', m_2', I), where m_1' is either m_1 or a result of applying the RSA to m_1, m_2' is either m_2 or a result of applying the RSA to m_2 and I is the set of all incorporations of m_1' and m_2' with windows similar to w_1 and w_2.

First, fix determines if w_1 and w_2 incorporate. If they do, then fix returns a set containing a single triple, (m_1, m_2, I), where I is the set of all incorporations of w_1 and w_2. If this is the case, then either (1) there was no fragment matching mistake to begin with or (2) the fragment matching mistake occurred in a location where it does not prevent incorporation. In the first case, fix is correct in not attempting to repair either map, because there is no mistake to repair. In the second case, fix is incorrect because the fragment matching mistake is not detected; this is simply a case where there is not enough evidence for fix to use.

If w_1 and w_2 do not incorporate, then fix tries to repair the maps by splitting fragments. fix calls the function find_all_scans, which performs a topological scan of w_1 and w_2. (This process was described in great detail in §2.) Then fix calls the function fix_by_split (see Figure 81). fix_by_split returns a set of triples of the same form as that returned by fix. The incorporations returned by fix_by_split are those that

```
SET
fix(w1,w2)
    WINDOW          w1,w2;
{
    SET             ans,scans,incorps;
    MAP             m1,m2;

    incorps ← incorporate(w1,w2);

    if (incorps = ∅)
        then
            scans ← find_all_scans(w1,w2);
            ans ← fix_by_split(w1,w2,scans);

            if (ans = ∅)
                then ans ← fix_by_combine(w1,w2,scans);
            fi
        else
            m1 ← map_of_window(w1);
            m2 ← map_of_window(w2);
            ans ← {(m1,m2,incorps)};
    fi

    return(ans);
}
```

Figure 80: Pseudocode for fix

```
SET
fix_by_split(w1,w2,scans)
    WINDOW          w1,w2;
    SET             scans;
{
    PAIR p1,p2;
    MAP m1,m2;

    m1 ← map_of_window(w1);
    m2 ← map_of_window(w2);

    p1 ← find_vfrags_to_split(m1,m2,scans);
    p2 ← perform_splits(m1,m2,p1);
    ans ← find_all_incorporations(w1,w2,p2);

    return(ans);
}
```

Figure 81: Pseudocode for fix_by_split

result when one of the maps has one of its fragments split by the RSA. The fragments to split are determined using the results of the topological scan and the split table. If the set returned by fix_by_split is non-empty, then this set is returned by fix.

If fix_by_split returns ∅, (i.e., no split resulted in maps that incorporated) then fix calls the function fix_by_combine (see Figure 82). fix_by_combine also returns a set of triples of the same form as that returned by fix. However, the incorporations returned by fix_by_combine are those that result when one of the maps has two of its fragments combined by the RSA. The fragments to combine are determined

```
SET
fix_by_combine(w1,w2,scans)
    WINDOW          w1,w2;
    SET             scans;
{
    PAIR p1,p2;
    MAP m1,m2;

    m1 ← map_of_window(w1);
    m2 ← map_of_window(w2);

    p1 ← find_vfrags_to_combine(m1,m2,scans);
    p2 ← perform_combines(m1,m2,p1);
    ans ← find_all_incorporations(w1,w2,p2);

    return(ans);
}
```

Figure 82: Pseudocode for fix_by_combine

using the results of the topological scan and the combine table. The set returned by fix_by_combine is returned by fix. (Note that this set might be empty.)

The fact that fix_by_combine is called only if fix_by_split fails to find an answer (i.e., returns ∅) implies a preference towards repairing fragment mistakes by splits over combines. The preference is based upon the belief that it is more common that real fragments of similar length are incorrectly matched than it is that real fragments that should be matched are not. Of course, this preference is sometimes incorrect and introduces the possibility that fix returns incorrect results. The reason for having a preference is that one is more likely to get a single incorporation from fix.

Now fix_by_split is examined in more detail. fix_by_split extracts the maps m_1 and m_2 corresponding to the windows w_1 and w_2. Then it calls the function find_vfrags_to_split, which returns a pair of the form (vfs_1,vfs_2), where vfs_1 and vfs_2 are sets of virtual fragments. vfs_1 is the set of virtual fragments in m_1 whose splitting may repair a fragment matching mistake in m_1. These fragments are determined by using the results of the topological scan and the split table (as described in §3). vfs_2 is computed similarly for m_2.

Then fix_by_split calls the function perform_splits with $p = (vfs_1,vfs_2)$ (see Figure 83). perform_splits is very simple, returning a pair of the form (ms_1,ms_2), where ms_1 is the set of all maps that result from splitting each virtual fragment in vfs_1 and ms_2 is computed similarly using vfs_2.

At this point, fix_by_split has new versions of m_1 and m_2, at most one of which may correspond to the underlying reality. fix_by_split must try to determine which of these (if any) might correspond to the underlying reality. For this purpose, fix_by_split calls the function find_all_incorporations (see Figure 84).

find_all_incorporations is based upon the following premise. If m_1 is the map containing the fragment matching mistake and a new version of m_1 corresponds to the underlying reality, then this new version must successfully incorporate with w_2 using a window similar to w_1. (A similar statement can be made for new versions of m_2.) find_all_incorporations takes the windows w_1 and w_2 and the pair (ms_1,ms_2) returned by perform_splits and returns a set of triples of the same form as that returned by fix.

```
PAIR
perform_splits(m1,m2,p)
    MAP             m1,m2;
    PAIR            p;
{
    PAIR            ans;
    SET             vfs1,vfs2,ms1,ms2;

    (vfs1,vfs2) ← p;

    ms1 ← ∅;
    for vf ∈ vfs1 do ms1 ← ms1 ∪ top_level_split(m1,vf); rof

    ms2 ← ∅;
    for vf ∈ vfs2 do ms2 ← ms2 ∪ top_level_split(m2,vf); rof

    ans ← (ms1,ms2);
    return(ans);
}
```

Figure 83: Pseudocode for perform_splits

```
SET
find_all_incorporations(w1,w2,p)
    WINDOW          w1,w2;
    PAIR            p;
{
    MAP             m1,m2,m3;
    WINDOW          w1;
    SET             ms1,ms2,ans;

    (ms1,ms2) ← p;

    for m3 ∈ ms1 do
        w3 ← find_similar_window(m3,w1);
        ms3 ← incorporate(w3,w2);
        if (ms3 ≠ ∅)
            then ans ← ans ∪ {(m3,m2,ms3)};
        fi
    rof

    for m3 ∈ ms2 do
        w3 ← find_similar_window(m3,w2);
        ms3 ← incorporate(w1,w3);
        if (ms3 ≠ ∅)
            ans ← ans ∪ {(m1,m3,ms3)};
        fi
    rof

    return(ans);
}
```

Figure 84: Pseudocode for find_all_incorporations

find_all_incorporations takes each map m_3 in the set ms_1 (thus m_3 is a version of m_1), finds a window for m_3 in a similar position as w_1 and attempts to incorporate it with w_2. If a non-empty set of maps ms_3 results from the incorporation, then the triple (m_3,m_2,ms_3) is placed in the set to be returned. This triple represents one possible way of fixing a fragment matching mistake in m_1 that leads to successful

incorporation, which is a solution to the problem FIX is attempting to solve. Next, find_all_incorporations takes each map m_3 in the set ms_2 (so m_3 is now a version of m_2), finds a window for m_3 in a similar position as w_2 and attempts to incorporate it with w_1. Again, if a non-empty set of maps ms_3 results from the incorporation, then the triple (m_1,m_3,ms_3) is placed in the set to be returned. Finally, find_all_incorporations returns the set of triples it has constructed, which is then returned by fix_by_split.

Now fix_by_combine is examined in more detail. fix_by_combine is very similar to fix_by_split. It extracts the maps m_1 and m_2 corresponding to the windows w_1 and w_2. Then it calls the function find_vfrags_to_combine, which returns a pair of the form $(vfps_1,vfps_2)$, where $vfps_1$ and $vfps_2$ are sets of *pairs* of virtual fragments. $vfps_1$ is the set of virtual fragment pairs in m_1 whose combining may repair a fragment matching mistake in m_1. These fragments are determined by using the results of the topological scan and the combine table (as described in §4). $vfps_2$ is computed similarly for m_2.

fix_by_combine calls the function perform_combines with $vfps_1$ and $vfps_2$ (see Figure 85). This function is very similar to perform_splits, except that perform_combines combines fragments rather than splitting them. It returns a pair of the form (ms_1,ms_2), where ms_1 is the set of all maps that result from combining each virtual fragment pair in $vfps_1$ and ms_2 is computed similarly using $vfps_2$. Finally, fix_by_combine calls find_all_incorporations in order to determine which maps resulting from combines actually allow incorporation. The set returned by find_all_incorporations is returned by fix_by_combine, which is in turn returned by fix.

This concludes the section describing the highest level logic of the FIX algorithm. The reader should now understand how the components described in the previous sections work together to locate and repair fragment matching mistakes.

```
PAIR
perform_combines(m1,m2,p)
    MAP             m1,m2;
    PAIR            p;
{
    PAIR                    ans;
    SET                     vfps1,vfps2,ms1,ms2;
    VIRTUAL_FRAGMENT        vf1,vf2;

    (vfps1,vfps2) ← p;

    ms1 ← ∅;
    for (vf1,vf2) ∈ vfps1 do ms1 ← ms1 ∪ top_level_combine(m1,vf1,vf2); rof

    ms2 ← ∅;
    for (vf1,vf2) ∈ vfps2 do ms2 ← ms2 ∪ top_level_combine(m2,vf1,vf2); rof

    ans ← (ms1,ms2);
    return(ans);
}
```

Figure 85: Pseudocode for perform_combines

6. Conclusion

The algorithm described in this report, the FIX algorithm, is useful for finding and repairing maps that contain a fragment matching mistake. FIX uses the technique of topological scanning to identify fragments which prevent the incorporation of two maps of interest. The results of the topological scan and a detailed analysis of how a fragment matching mistake affects a map are used to determine which fragments should be split or combined. Then the RSA is used to actually split and combine those fragments. Results from the RSA which enable the incorporation of the two maps of interest are returned by FIX as the final answers.

There are certainly aspects of FIX which could be improved. The most significant weakness of FIX is its dependence upon the assumption that topological scanning will find at most two class 3 fragments in a window. This assumption simplifies the analyses in §3 and §4. In addition, the discarding of certain cases which are deemed unlikely also simplifies these analyses. However, these simplifications limit the complexity of the fragment matching mistakes that FIX can identify and repair. If a fragment matching mistake causes too much deviation from the underlying reality in a map, it is unlikely that FIX will be able to repair the map.

Most importantly, the topological scanning technique has the potential to be useful outside of FIX. Topological scanning identifies fragments that are "out of place" independent of why those fragments are out of place. Thus, for instance, one might be able use topological scanning to form an algorithm which repairs maps that are incorrect due to extra or missing real fragments in a clone. The application of topological scanning to other areas of DNA mapping seems to be the most promising avenue for future work based upon the work described in this report.

APPENDIX A

Description of Functions not Defined by Pseudocode

This appendix describes the functionality of operators used in the pseudocode as primitives, for which no pseudocode was given.

```
SET
find_all_valid_configurations(w1,w2)
    WINDOW w1,w2;
```

This function returns a set of configurations that are consistent with the windows w_1 and w_2. It follows the rules illustrated in Table 1.

```
SET
find_best_matchlists(s1,s2)
    SET s1,s2;
```

This function returns a set containing the maximum size matchlists between the sets of virtual fragments s_1 and s_2.

```
WINDOW
find_similar_window(m,w)
    MAP m;
    WINDOW w;
```

This function returns a minimum size window for the map m that spans the same area as the window w. (I.e., The returned window contains all clone ends in w that are in m as well.)

```
PAIR
find_vfrags_to_combine(m1,m2,s)
    MAP m1,m2;
    SET s;
```

This function takes two maps m_1 and m_2 and a set of SCANs s from the topological scan of m_1 and m_2. It returns a pair of sets (s_1,s_2), where s_1 is a set of pairs of virtual fragments in m_1 that should be combined. s_2 is defined similarly for m_2. The virtual fragments to combine are determined using the combine table (described in §4.4).

```
PAIR
find_vfrags_to_split(m1,m2,s)
    MAP m1,m2;
    SET s;
```

This function takes two maps m_1 and m_2 and a set of SCANs s from the topological scan of m_1 and m_2. It returns a pair of sets (s_1,s_2), where s_1 is a set of virtual fragments in m_1 that should be split. s_2 is defined similarly for m_2. The virtual fragments to split are determined using the split table (described in §3.5).

SET
incorporate(w_1,w_2)
    WINDOW w_1,w_2;

This function returns a set containing all possible incorporations of the windows w_1 and w_2.

SET
left_elements_of(l)
    LIST l;

The list l must contain only pairs. This function returns a set containing the left element of each pair in l.

LIST
list_reverse(l)
    LIST l;

This function returns a list that is the list l with the order of its elements reversed.

MAP
map_of_window(w)
    WINDOW w;

This function returns the map corresponding to the window w.

SET
right_elements_of(l)
    LIST l;

The list l must contain only pairs. This function returns a set containing the right element of each pair in l.

anytype
stack_pop(s)
    STACK s;

This function returns the object on top of the stack s and removes that object from s.

STACK
stack_push(s,o)
    STACK s;
    anytype o;

This function returns a stack that is the stack s with the object o pushed on top.

SET
top_level_combine(m,vf_1,vf_2)
    MAP m;
    VIRTUAL_FRAGMENT vf_1,vf_2;

This function returns a set of maps that are the result of combining the virtual fragments vf_1 and vf_2 in the map m.

```
SET
top_level_split(m,vf)
    MAP m;
    VIRTUAL_FRAGMENT vf;
```

This function returns a set of maps that are the result of splitting the virtual fragment vf in the map m.

```
LIST
window_find_vfrag_setseq(w)
    WINDOW w;
```

This function returns the VFSS (defined in §2.3.2) of the window w.

APPENDIX B

Notational Conventions in the Pseudocode

This appendix defines the symbolic notation used in the body of the report.

∅	the empty set
|s|	the cardinality of a set s
[]	the empty list
|x|	the cardinality of a list x
x[i]	the ith element of a list x
x[i,...]	the sublist of a list x formed by the ith through last elements
||	the list concatenation operator
$\|f_1=v_1, f_2=v_2, \ldots, f_N=v_N\|$	a record-like structure with N fields where f_i is the *i*th field name and v_i is the value of the *i*th field.
empty_stack	the empty stack

References

1. Maynard V. Olson, James E. Dutchik, Madge Y. Graham, Garret M. Brodeur, Cynthia Helms, Mark Frank, Mia MacCollin, Robert Scheinman, and Thomas Frank, "Random-Clone Strategy for Genomic Restriction Mapping in Yeast," *Proc. Natl. Acad. Sci. (Genetics), Vol. 83*, pp. 7826-7830 (October, 1986).

2. Linda Riles, James E. Dutchik, Amara Baktha, Brigid K. McCauley, Edward C. Thayer, Mary P. Leckie, Valerie V. Braden, Julie E. Depke, and Maynard V. Olson, "Physical maps of the six smallest chromosomes of *Saccharomyces cerevisiae* at a resolution of 2.6 kilobase pairs," *Genetics, Vol. 134*, pp. 81-150 (May 1993).

3. Will Gillett, *DNA Mapping Algorithms: Strategies for Single Restriction Enzyme and Multiple Restriction Enzyme Mapping*, Washington University, Dept. of Computer Science, Technical Report WUCS-92-29, August 1992.

4. Eric S. Lander and Michael S. Waterman, "Genomic Mapping By Fingerprinting Random Clones: A Mathematical Analysis," *Genomics, Vol. 2*, pp. 231-239 (1988).

5. Frank Michiels, Alister G Craig, Guenther Zehetner, George P. Smith, and Hans Lehrach, "Molecular approaches to genome analysis: a strategy for construction of ordered overlapping clone libraries," *CABIOS, Vol. 3, No. 3*, pp. 203-210 (1987).

6. A. V. Carrano, J. Lamerdin, L. K. Ashworth, B. Watkins, E. Branscomb, T. Slezak, M. Raff, P. J. De Jong, D. Keith, L. McBride, S. Meister, and M. Kronick, "A High-resolution, Fluorescence-Based, Semiautomated Method for DNA Fingerprinting," *Genomics, Vol. 4*, pp. 129-136 (1989).

7. John Sulston, Frank Mallett, Roger Staden, Richard Durbin, Terry Horsnell, and Alan Coulson, "Software for genome mapping by fingerprinting techniques," *CABIOS, Vol. 4, No. 1*, pp. 125-132 (1988).

8. Alan Coulson, John Sulston, Sydney Brenner, and Jonathan Karn, "Toward a physical map of the genome of the nematode *Caenorhabditis elegans*," *Proceeding National Academy of Science: Genetics, Vol 83*, pp. 7821-7825 (October 1986).

9. Larry Goldstein and Michael S. Waterman, "Mapping DNA by Stochastic Relaxation," *Advances in Applied Mathematics, Vol. 8*, pp. 194-207 (1987).

10. Yuji Kohara, Kiyotaka Akiyama, and Katsumi Isono, "The Physical Map of the Whole E. coli Chromosome: Application of a New Strategy for Rapid Analysis and Sorting of a Large Genomic Library," *Cell, Vol. 50*, pp. 495-508 (July, 1987).

11. David C. Torney, Karen R. Schenk, Clive C. Whittaker, and Steven W. White, "Computational Methods for Physical Mapping of Chromosomes," *Proceedings of the First International Conference on Electrophoresis, Supercomputing, and the Human Genome*, pp. 268-278 (April, 1990).

12. James Daues and Will Gillett, *DNA Mapping Algorithms: Fragment Splitting and Combining*, Washington University, Dept. of Computer Science, Technical Report WUCS-91-50, October 1991.