Washington University in St. Louis
Washington University Open Scholarship

All Computer Science and Engineering Research
Computer Science and Engineering

Report Number: WUCS-93-50

1993

DNA Mapping Algorithms: Fragment Matching Mistake Detection and Correction

Jim Daues and Will Gillett

When using random clone overlap based methods to make DNA maps, fragment matching mistakes, the incorrect matching of similar length restriction fragments, are a common problem that produces incorrect maps. Previous work presented the Restricted Splitting Algorithm (or RSA), which is useful for repairing a map containing a fragment mistake when the location of the mistake is known. This work presents an algorithm, called FIX, which attempts to identify the location of the fragment matching mistake and then uses the RSA to repair the map containing the mistake. In essense, the two techniques combined constitute a hypothesis formulation/hypothesis verification... **Read complete abstract on page 2.**

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

Recommended Citation
Daues, Jim and Gillett, Will, "DNA Mapping Algorithms: Fragment Matching Mistake Detection and Correction" Report Number: WUCS-93-50 (1993). *All Computer Science and Engineering Research.*
https://openscholarship.wustl.edu/cse_research/545

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

This technical report is available at Washington University Open Scholarship: https://openscholarship.wustl.edu/cse_research/545

DNA Mapping Algorithms: Fragment Matching Mistake Detection and Correction

Jim Daues and Will Gillett

Complete Abstract:

When using random clone overlap based methods to make DNA maps, fragment matching mistakes, the incorrect matching of similar length restriction fragments, are a common problem that produces incorrect maps. Previous work presented the Restricted Splitting Algorithm (or RSA), which is useful for repairing a map containing a fragment mistake when the location of the mistake is known. This work presents an algorithm, called FIX, which attempts to identify the location of the fragment matching mistake and then uses the RSA to repair the map containing the mistake. In essense, the two techniques combined constitute a hypothesis formulation/hypothesis verification paradigm for correcting restriction maps that contain fragment matching mistakes.

DNA Mapping Algorithms: Fragment Matching Mistake Detection and Correction

Jim Daues and Will Gillett

WUCS-93-50

October 1993

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
St. Louis MO 63130-4899

This work was supported by the James S. McDonnell Foundation under Grant 87-24 and NIH under Grant R01 HG00180.

ABSTRACT

When using random clone overlap based methods to make DNA maps, *fragment matching mistakes*, the incorrect matching of similar length restriction fragments, are a common problem that produces incorrect maps. Previous work presented the Restricted Splitting Algorithm (or RSA), which is useful for repairing a map containing a fragment mistake when the location of the mistake is known. This work presents an algorithm, called FIX, which attempts to identify the location of the fragment matching mistake and then uses the RSA to repair the map containing the mistake. In essence, the two techniques combined constitute a hypothesis formulation/hypothesis verification paradigm for correcting restriction maps that contain fragment matching mistakes.

TABLE OF CONTENTS

1. Introduction 1
    1.1. An Overview of DNA Mapping 1
    1.2. Some Details of DNA Mapping 2
        1.2.1. Data Collection 2
        1.2.2. Mapping Two Clones Together 4
        1.2.3. Mapping a Set of Clones 5
    1.3. The Fragment Matching Mistake and Fragment Splitting 12
    1.4. Overview of Fragment Matching Mistake Detection and Correction 14

2. Topological Scanning 19
    2.1. Quick Overview 19
    2.2. Finding All Valid Configurations 22
    2.3. Finding ATVMLs for a Particular Configuration 24
        2.3.1. An Informal Example 24
        2.3.2. More Formal Discussion and Example 25

3. Detecting Fragments to Split 45
    3.1. Overview 45
        3.1.1. Heuristic Basis for Detecting Fragments to Split 45
        3.1.2. Limited Enumeration of Possible Underlying Realities 46
        3.1.3. Determining the Important Properties of a Possible Underlying Reality 48
        3.1.4. Constructing the Split Table 49
    3.2. Limited Enumeration of Possible Underlying Realities 50
        3.2.1. The First Level of the Enumeration 50
        3.2.2. The Second Level of the Enumeration 52
        3.2.3. The Third Level of the Enumeration 52
    3.3. Detailed Analysis of a Particular Case 54
    3.4. Results of the Analysis of All Cases 56
    3.5. Constructing the Split Table 66

4. Detecting Fragments to Combine 70
    4.1. Differences in the Limited Enumeration of Possible Underlying Realities 70
    4.2. Limited Enumeration of Possible Underlying Realities 72
        4.2.1. The First Level of Enumeration 72
        4.2.2. The Second Level of Enumeration 73
    4.3. Results of the Analysis of All Cases 73
    4.4. Constructing the Combine Table 78

5. Correcting Two Maps 79
    5.1. The FIX Algorithm 79

6. Conclusion 84

Appendix A: Description of Functions not Defined by Pseudocode 85

Appendix B: Notational Conventions in the Pseudocode 88

TABLE OF FIGURES

Figure 1: Random clone inserts in context 2
Figure 2: Making a clone from a clone insert 3
Figure 3: Set of clones to map 5
Figure 4: Map unit produced from mapping Clone 1 and Clone 3 7
Figure 5: Clones 1, 3, and 4 7
Figure 6: Clones 1, 2, 3, and 4 8
Figure 7: Completed mapping of clone set 8
Figure 8: An ambiguous assimilation 10
Figure 9: Subset ambiguity 11
Figure 10: Two overlapping clones 12
Figure 11: The fingerprints of the clones in Figure 10 13
Figure 12: Map built from the fingerprints in Figure 11 13
Figure 13: Map produced by the RSA 14
Figure 14: Maps M_1 and M_2 15
Figure 15: Maps M_1 and M_2 and an ATVML x 17
Figure 16: Maps M_1', M_2, M_I and a matchlist 18
Figure 17: Examples of the virtual fragment classes 20
Figure 18: Pseudocode for find_all_scans 21
Figure 19: Pseudocode for topological_scan_fixed_configuration 21
Figure 20: Result of the scan in the right direction 26
Figure 21: Result of the scan in the left direction 27
Figure 22: Possible starting groups for an assimilation 28
Figure 23: Possible starting groups for an extension 29
Figure 24: Pseudocode for topological_scan_fixed_direction 30
Figure 25: Pseudocode for topological_scan_fixed_start 31
Figure 26: Pseudocode for scan_right 32
Figure 27: Pseudocode for fix_by_ignore 33
Figure 28: Pseudocode for fix_by_discard 34
Figure 29: Pseudocode for find_initial_partial_scan 35
Figure 30: The PARTIAL_SCAN ps_0 36
Figure 31: The PARTIAL_SCAN ps_1 37
Figure 32: The PARTIAL_SCAN ps_2 38
Figure 33: The PARTIAL_SCAN ps_3 38
Figure 34: Pseudocode for best_match_ahead 40
Figure 35: Looking ahead for a matchlist 41
Figure 36: The PARTIAL_SCAN ps_4 42
Figure 37: The PARTIAL_SCAN ps_5 42

Figure 38: The PARTIAL_SCAN ps_6 42
Figure 39: The PARTIAL_SCAN ps_7 43
Figure 40: The PARTIAL_SCAN ps_8 43
Figure 41: The PARTIAL_SCAN ps_9 43
Figure 42: The PARTIAL_SCAN ps_{10} 43
Figure 43: The PARTIAL_SCAN ps_{11} 44
Figure 44: The SCAN s_1 44
Figure 45: The SCAN s_2 44
Figure 46: Illustration of the assumptions for the limited enumeration 47
Figure 47: A specific final case 48
Figure 48: Pushed virtual fragments 53
Figure 49: Summary of the limited enumeration 54
Figure 50: Final case S11.2.2 54
Figure 51: A sketch of final case S1.1.1 56
Figure 52: A sketch of final case S1.1.2 57
Figure 53: A sketch of final case S1.2.1 58
Figure 54: A sketch of final case S1.2.2 58
Figure 55: A sketch of final case S1.3.1 59
Figure 56: A sketch of final case S1.3.2 60
Figure 57: A sketch of final case S11.1.1 60
Figure 58: A sketch of final case S11.1.2 61
Figure 59: A sketch of final case S11.2.1 62
Figure 60: A sketch of final case S11.2.2 62
Figure 61: A sketch of final case S12.1.1 63
Figure 62: A sketch of final case S12.1.2 64
Figure 63: A sketch of final case S12.2.1 64
Figure 64: A sketch of final case S12.2.2 65
Figure 65: A sketch of final case S31.1.1 65
Figure 66: The split table after examining the first group of entries 67
Figure 67: The split table after examining the second group of entries 68
Figure 68: The split table after examining final case S11.2.2 69
Figure 69: Illustration of the assumptions for the limited enumeration 71
Figure 70: Summary of the limited enumeration 73
Figure 71: A sketch of final case C1.1 74
Figure 72: A sketch of final case C1.2 74
Figure 73: A sketch of final case C1.3 75
Figure 74: A sketch of final case C2.1 76
Figure 75: A sketch of final case C2.2 76
Figure 76: A sketch of final case C4.1 77
Figure 77: A sketch of final case C14.1 77
Figure 78: A sketch of final case C14.2 78
Figure 79: A portion of the combine table 79
Figure 80: Pseudocode for fix 80
Figure 81: Pseudocode for fix_by_split 80

Figure 82: Pseudocode for fix_by_combine 81
Figure 83: Pseudocode for perform_splits 82
Figure 84: Pseudocode for find_all_incorporations 82
Figure 85: Pseudocode for perform_combines 83

TABLE OF TABLES

Table 1: Valid configurations for types of windows 23
Table 2: Possible results of a topological scan 49
Table 3: The first-level cases for Split 51
Table 4: The second-level cases for Split 52
Table 5: Possible results of a topological scan for final case S1.1.1 57
Table 6: Possible results of a topological scan for final case S1.1.2 57
Table 7: Possible results of a topological scan for final case S1.2.1 58
Table 8: Possible results of a topological scan for final case S1.2.2 59
Table 9: Possible results of a topological scan for final case S1.3.1 59
Table 10: Possible results of a topological scan for final case S1.3.2 60
Table 11: Possible results of a topological scan for final case S11.1.1 61
Table 12: Possible results of a topological scan for final case S11.1.2 61
Table 13: Possible results of a topological scan for final case S11.2.1 62
Table 14: Possible results of a topological scan for final case S11.2.2 63
Table 15: Possible results of a topological scan for final case S12.1.1 63
Table 16: Possible results of a topological scan for final case S12.1.2 64
Table 17: Possible results of a topological scan for final case S12.2.1 64
Table 18: Possible results of a topological scan for final case S12.2.2 65
Table 19: Possible results of a topological scan for final case S31.1.1 66
Table 20: The first-level cases for Combine 72
Table 21: The second-level cases for Combine 73
Table 22: Possible results of a topological scan for final case C1.1 74
Table 23: Possible results of a topological scan for final case C1.2 75
Table 24: Possible results of a topological scan for final case C1.3 75
Table 25: Possible results of a topological scan for final case C2.1 76
Table 26: Possible results of a topological scan for final case C2.2 76
Table 27: Possible results of a topological scan for final case C4.1 77
Table 28: Possible results of a topological scan for final case C14.1 77
Table 29: Possible results of a topological scan for final case C14.2 78

1. Introduction

1.1. An Overview of DNA Mapping

DNA is the genetic material which supplies the blueprint for an organism's development. A DNA molecule is composed of **nucleotides**, with each nucleotide consisting of a sugar, a phosphate, and one of the four bases: A (Adenine), T (Thymine), C (Cytosine), and G (Guanine). Nucleotides are distinguished by the base they contain. Sugar-phosphate bonds bind the nucleotides into strands, and a base on one strand can form hydrogen bonds with a base on another strand. However, only certain base pairings are allowed: A bonds with T, and C bonds with G. Thus, A and T are known as **complementary bases**, as are C and G. A double-stranded DNA molecule is made of two complementary DNA nucleotide strands bound together by base pairing, the base sequence on one strand determining the complementary sequence on the other strand.

DNA restriction mapping [1-10] deals with determining the positions of specific sites along a given segment of DNA, which we will here refer to as a **genome**. The sites of interest are called **restriction sites**, and consist of a specific subsequence of DNA, often six nucleotides long. These restriction sites are recognized by specific enzymes, known as **restriction enzymes**; a restriction enzyme cleaves (or cuts) DNA that it encounters at these restriction sites. Thus, given appropriate conditions, a restriction enzyme reacting with a strand of DNA will produce fragments of DNA whose lengths are the distances between two successive restriction sites along the original DNA. The process of **agarose gel electrophoresis** can be used to measure the approximate lengths of these fragments, which are known as **restriction fragments**. If it were possible to (a) identify each restriction fragment present in the genome, (b) determine the length of each restriction fragment, and (c) determine the order of the restriction fragments in the genome, then it would be possible to construct the map of the restriction sites.

The mechanism for obtaining this information is somewhat indirect. Ordering of the restriction fragments is achieved by cleaving multiple copies of the original DNA at random positions to produce randomly overlapping strands of DNA, known as **clones**. Each clone is **digested** by the restriction enzyme of interest, and electrophoresis is used to determine the lengths of the resulting restriction fragments. This list of restriction-fragment lengths is known as the **fingerprint** of the clone. Overlap between the clones is inferred based on the similarity of the fingerprints of the restriction-fragment lengths, and the order of the clones is inferred based on multiple-clone overlap. As overlap between the clones is inferred from a significant number of restriction fragments of similar (within measurement error bounds) lengths, the exact order of the restriction fragments within each clone may remain unknown; only the relative (partial) ordering of large groups of fragments may be inferrable. As more clones are found to overlap a specific region of the original genome, the positions of the clone ends are used to refine the original partial order of the restriction fragments by reducing the size of the groups for which the fragment order is unknown.

This process of DNA restriction mapping is analogous to solving a large jigsaw puzzle. However, the uncertainty of where a clone should be placed can be significant, due to measurement error (produced during electrophoresis), experimental error (produced during cloning or digestion with the restriction enzyme), and certain biological properties of the DNA being mapped (e.g., two fragments of the same length do not necessarily contain the same sequence of nucleotides). When putting together a jigsaw puzzle, the pieces of the puzzle have several cues (shape, color, pattern on the surface) which can be used to guide their ultimate positioning in the final solution. In DNA restriction mapping, the clones have no shape or color, but the fingerprint information can be viewed as a pattern to be matched against potentially overlapping clones. The objective is to find a consistent positioning of clones with respect to one another in which fragments in different clones can be identified with one another while all fragments of each clone remain contiguous and no gaps or unpaired fragments are present internally. There may be multiple solutions to this restriction-map puzzle, and the one (or ones) which is (are) most compact is (are) preferred.

1.2. Some Details of DNA Mapping

This section presents a more in-depth discussion of the process described in the previous section.

1.2.1. Data Collection

The type of data considered in DNA mapping is clone fingerprint data. Prior to any mapping, the genome to be mapped is duplicated using traditional biological means. Then, the DNA is randomly cleaved into smaller sections by partially digesting it with a **cloning restriction enzyme**; this produces random **clone inserts**. The partial digestion process causes different copies of the DNA to be cleaved at some but not at all of the cloning restriction sites. This tends to produce clone inserts which have random overlap with one another. This is depicted in Figure 1, where four clone inserts (which will be used later in a running example) are shown in their positional context within the genome. The ends of these clones correspond to sites randomly cleaved by the cloning restriction enzyme during the partial digestion; other sites may be present within the clones at which cleavage did not occur.

The cloning restriction enzyme is usually selected to be different from any of the restriction enzymes being mapped; in the protocol described here, it is *assumed* that they are different. This implies that clone-end sites do not coincide with the sites of any of the restriction enzymes being mapped. Clone inserts are inserted into a biological organism known as the λ **phage**, which is a virus used as a cloning vector (*i.e.*, a mechanism used to reproduce many copies of a clone insert). This is depicted in Figure 2. The body of the λ phage is removed (leaving a left and a right λ arm) and is replaced by a clone insert. The λ arms are engineered so that they do not contain restriction sites corresponding to any of the restriction enzymes being mapped. Since the site at the end of the clone insert does not correspond to a site of any restriction enzyme being mapped, there is always a **partial fragment** at the end of each clone insert which remains attached to the λ arm as the clone is digested with a restriction enzyme being mapped. The λ arms (with the partial fragment attached) are large, and are thus easily identified during subsequent processing. Only the **complete fragments** (*i.e.*, those for which there are two delimiting restriction sites within the clone insert itself) are selected for inclusion in the mapping activity.

The size of the clone inserts is limited by the packaging mechanism of the λ phage. This size range lies roughly between 10,000 and 25,000 base pairs (bp); The combination of the λ phage and the inserted DNA is known as a **clone**, because the λ phage will be used to reproduce multiple copies of the clone

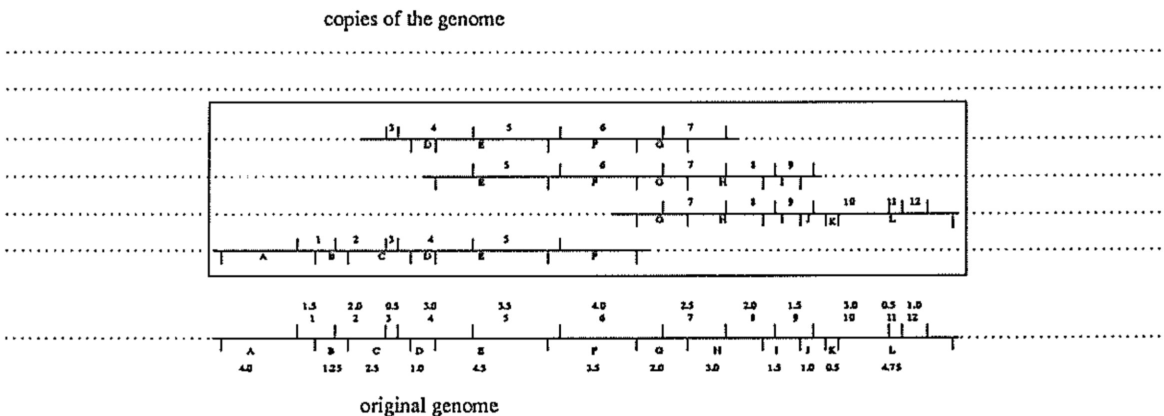
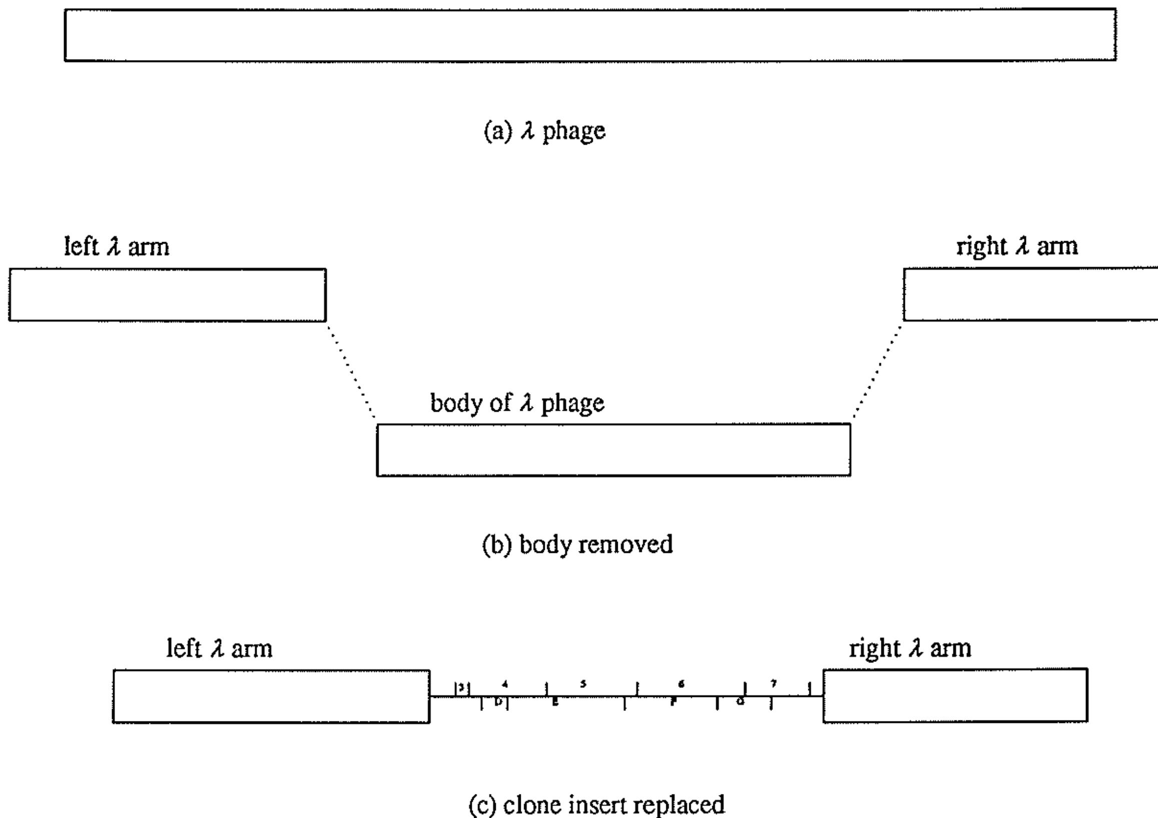


Figure 1: Random clone inserts in context



(a) λ phage

(b) body removed

(c) clone insert replaced

Figure 2: Making a clone from a clone insert

insert.

Enough independent λ clones must be produced so that randomly selected clones will cover all (or almost all) of the genome. Overlapping clones can be viewed as redundant copies of the underlying genome. The **redundancy factor** of a set of clones with respect to the underlying genome is the average (over all nucleotides) of the number of times the underlying genome is duplicated. The higher the redundancy factor, the higher the probability of covering the entire genome. A redundancy factor of between five and ten is usual. This implies that any region of DNA from the original genome is likely to appear in five to ten clones, on the average.

Since the inserts of DNA are the result of random cleavings, each insert may or may not contain some overlap with another insert from roughly the same region. This partial overlap can mean that each of two inserts contains DNA in addition to the region of overlap or that one insert is a subsection of another. The success of DNA mapping depends on the fact that the clones contain these overlapping regions of DNA. It is this overlap which will allow the clones to be aligned in the order in which they existed in the original genome.

After the clones are made, they are isolated and *in vivo* DNA reproduction is employed to obtain enough DNA for subsequent processing. For each clone, the DNA extracted from the amplification process is completely digested by a restriction enzyme (the restriction enzyme being mapped), producing fragments of DNA called restriction fragments. The lengths of these fragments are then measured using **agarose gel electrophoresis** technology. When an electric current is passed through an agarose gel in which DNA

fragments have been placed, the fragments will migrate down the gel. It is easier for smaller fragments to move through the gel than it is for larger ones, so the fragments arrange themselves in order of decreasing length. This creates **lanes** of DNA fragments in which **bands** of DNA of the same length have migrated to the same position in the gel. After the gel has been stained, these bands can be detected and their positions in the gel determined. **Reference lanes**, containing DNA fragments of known length, are also present in the gel. Using the positions of the bands in these reference lanes and the process of interpolation, it is possible to estimate the lengths of restriction fragments in the data lanes. Unfortunately, standard agarose gel electrophoresis technology is limited to measuring accurately fragments in a particular size range; here the range is approximately 400 bp to 15 kilobase pairs (kb). Restriction enzymes can be chosen to assure that most of the restriction-enzyme fragment-length data fall in this range.

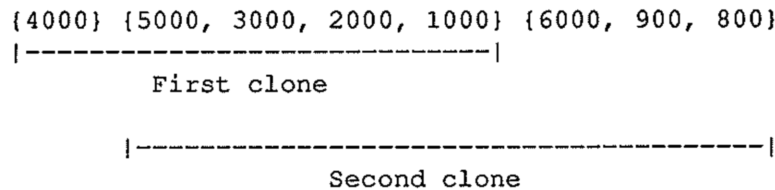
There are (at least) two significant sources of error which create uncertainty about the data produced by electrophoresis. The first is the classical problem of measurement error. From experimental evidence[11], it is known that the measured lengths of the same fragment measured multiple times (either as it occurs in different clone inserts or when the same insert is measured multiple times) are normally distributed about the true length of the fragment. This normal distribution is often characterized by giving an error window (a percentage difference around the true length) into which almost all measured lengths of the fragment will fall. Under some circumstances, it is possible to obtain a 3% error window around the true length of the fragment, 1.5% on either side of the actual length. Thus, a fragment which is actually 1000 bp in length may be measured as anywhere from 985 bp to 1015 bp. The second deals with determining the multiplicity of different genomic fragments of similar length; these are referred to as comigrating fragments. Two fragments of identical (or nearly identical) length will comigrate to the same location in the gel. Thus, it is possible for two (or more) fragments to be in the same band when the gel is stained. If this is not taken into account, the set of fragment lengths will not accurately reflect the number of fragments present in the clone. It is possible but difficult to identify multiple comigration fragments. The intensity of the stained DNA bands should decrease along the expanse of the gel, because there is less DNA to stain in smaller fragments. Deviation from this expected intensity distribution can be used to estimate the number of multiple restriction fragments present in a band.

1.2.2. Mapping Two Clones Together

The reason that clone data can be used to create a map of a genome is the fact that fragments which come from a single clone must be contiguous in the original DNA sequence. Given just one clone, it is impossible to know the ordering of the fragments within it; it is simply known that they *are* contiguous in a certain region of the original DNA. A more refined view of that area can be created by considering other clones which are suspected to overlap the same region. Consider one clone with fragment lengths {5000, 4000, 3000, 2000, 1000} and another with fragment lengths {6000, 5000, 3000, 2000, 1000, 900, 800}. Since these two clones share four fragments of the same lengths (5000, 3000, 2000, and 1000), it is highly probable that they are partially overlapping clones from the same general region of the original DNA. However, it is impossible to be sure these two actually do overlap without doing more biological work. Simply containing four fragments of the same lengths is no guarantee that two clones *actually* overlap, since two fragments of the same (apparent) length are not necessarily the same fragment. One of the ways that this is taken into account while mapping is to require **apparent overlap** of multiple fragments before assuming an **actual overlap** is present. Often, the minimum number of fragments which must seem to overlap (before actual overlap is inferred) is taken to be four. This increases the probability that the two clones actually share some region of the underlying genome.

Returning to the example, it is known that the five fragments in the first clone are contiguous (in some order). Similarly, the seven fragments of the second clone must be contiguous. This is all that can be determined from examining the clones independently of each other. However, more information can be extracted by examining the two clones in concert.

The four fragments which overlap also must be contiguous. This means that each clone can be divided into two sets, one set containing the fragments which overlap and the other set containing all the remaining fragments in the clone. In the first clone, these two sets are {4000} {5000, 3000, 2000, 1000}, while in the second clone they are {5000, 3000, 2000, 1000} {6000, 900, 800}. Since each of the two clones contains a region overlapping with the other clone, it is possible to fit the two back together into one partial sequence. This sequence is:



This ordering contains more information than either of the original two clones provided. Namely, it is now known that there is a restriction site 4000 bp in from one end of the first clone. Similarly, there is a restriction site 7,700 (6000 + 900 + 800) bp in from the other end of the of the second clone. The information about this particular region of the genome is still relatively unrefined. It is known that there are three sets of fragments, with one fragment in the first set, four fragments in the second set, and three in the last set. These subdivisions, or sets, will be referred to as **groups**. It is known how the three groups are positioned in relation to each other. It is not known, however, what the exact ordering of the fragments is in the second two groups. To gain a higher level of refinement, more clones would need to be added to the map.

The previous example is a trivial one. It ignored many of the problems which can occur while mapping, but its intent was to provide a first level of understanding about the basic process. With that understanding, it is possible to approach the mapping of a more complex, more realistic example.

1.2.3. Mapping a Set of Clones

Figure 3 presents a set of clones suspected of coming from the same region of the genome. The fragment lengths of each clone are sorted from longest to shortest, but this is for convenience only. Prior to mapping, nothing is known about the ordering of the fragments in any of the clones.

#1	#2	#3	#4	#5
6198	8567	6109	8644	4087
4082	7605	4087	6110	1085
1614	1605	1139	1600	529
1592	1586	1078	1573	517
1150	1139	630	1146	406
1092	623	527	632	
637		515		
513				

Figure 3: Set of clones to map

The first consideration is to determine which two clones should be mapped together initially. This is one area where intuition and experience are useful. A poor choice will result in problems with mapping later clones. Although intuition plays a large role in this initial choice, there are some guidelines which may be followed. One of the easiest is to make the initial choice based on the number of fragments in the clones, starting with the two clones which have the most fragments. In this case, these are Clones #1 and #3.

One way to approach clone-clone mapping is to scan through the fragments of each clone searching for a **match** (*i.e.*, two fragments whose lengths are within 3% of each other), starting with the largest fragments. Using this approach, the first match discovered between Clones #1 and #3 would be 6198—6109. (Although not the same length, the two fragment lengths are within the 3% error window.) After creating a match with two fragments, neither fragment is available for subsequent matches. Having paired 6198 with 6109, the process of scanning for matches continues in the two lists of fragment lengths. 4082 and 4087 are within 3%, so they are matched. Next, although there is a fragment of length 1614 in Clone #1, there is no corresponding fragment in Clone #3. Thus, 1614 does not match with anything. It is possible to use the ordering of the fragments by size to cut down on the amount of work performed in finding a match. If 1614 is under consideration, as soon as a fragment smaller than 1614 is found in the second clone (keeping in mind that "smaller than" must take into account the 3% window), no further searching for a match to this fragment is required. In this example, the search for a match for 1614 can stop as soon as the fragment 1139 is seen in Clone #3.

As with 1614, 1592 is unable to match with anything in Clone #3. This means that the next match that does occur is fragment length 1150 with fragment length 1139. This is followed by matching 1092 with 1078, and 637 with 630. There is now only one fragment left to examine in Clone #1 and two left to consider in Clone #3. The fragment with length 513 is the only unexamined one in Clone #1. The problem with matching it is that there are two possible matches. It might match with 527, or it might match with 515. Both are within the 3% error window. (A dual match like this is referred to as a **similar match.**) The 513—515 match might be considered better since there is just a two base pair difference in these lengths, whereas there is a fourteen base pair length difference between 513 and 527. Consequently, 513 is chosen to match with 515, and 527 remains unmatched.

Since there are no more fragments to consider, the mapping of Clone #1 with Clone #3 is complete. There is now a **fragment matching or fragmat** (*i.e.*, 6198—6109, 4082—4087, 1150—1139, 1092—1078, 637—630, 513—515) which describes the matches which exist between the two clones. It is also known which fragments in each clone did not pair. Using these data, the two clones can be put together as shown in Figure 4. It is no longer proper to call this finished structure a clone, since it is not that anymore. The term **map unit** is used to refer to the result of a mapping, such as this one. Map units are formed by mapping (or **fusing**) any two structures together: two clones, a clone with an existing map unit, or two map units. Map units generally contain more structure than the objects used to produce them. Note that it is always possible to identify a contiguous sequence of groups in a map unit which corresponds to an individual clone, as Figure 4 illustrates, because the fragments present in a clone must always remain contiguous.

In a map unit, some of the fragment lengths are not the lengths of the original fragments present in the clones. Instead, they are the average lengths of the fragments which matched. To emphasize this distinction, the term **virtual fragment** is used to describe a fragment which is the result of some matching. This is in contrast to **real fragments**, which are the actual fragments in the clones. The distinction often is irrelevant, and the blanket term fragment is used in most cases. The notation $vf<rf_1,\ldots,rf_n>$ is used to denote a virtual fragment composed of the real fragments $rf_1,\ldots,rf_n$. The **active clone set** (or **ACS**) of a virtual fragment is the set of clones from which the real fragments composing the virtual fragment come. Given two different map units constructed (in different ways) from the same set of clones but containing a different number of fragments, the map unit containing the smaller number of fragments will be considered more **compact** than the map unit containing the larger number of fragments.

```
--------          ---
 1614             |
 1592             |
--------          |          ---
 6154             |           |
 4085             |  clone #1 |
 1145             |           |
 1085             |           |
  634             |           |  clone #3
  514             |           |
--------          ---         |
  527                         |
--------                     ---
```

Figure 4: Map unit produced from mapping Clone 1 and Clone 3

Now that the first two clones are fused, it is time to map the remaining ones. Both #2 and #4 have the same number of fragments, so either could be considered next. In this example, Clone #4 is chosen to continue the mapping process with the map unit just produced (Figure 4). This mapping will not be presented in as much detail, but the same ideas as previously presented are being followed. The 1614 and 1600 match, as do the 1592 and 1573. Continuing, the 6154 and 6110 match, as do the 1145 and 1146, and the 634 and 632. The map unit, as it now stands, is shown in Figure 5.

Clone #2 is chosen as the next one to add. The fragments with lengths 8644 and 8567 match. Other matches are 1607—1605, 1583—1586, 1145—1139, and 633—623. Figure 6 shows the current state of the map unit being produced from the original set of data.

Finally, the last clone (#5) is added to the map. The 4085 and 4087 match, the 1085 and 1085 match, the 514 and 517 match, and the 527 and 529 match. The final completed mapping of these five clones is

```
--------                   ---
 8644                       |
--------    ---             |
 1607        |              |
 1583        |              |
--------     |      ---     | #4
 6139        |       |      |
 1145        | #1    |      |
  633        |       |      |
--------     |       |     ---
 4085        |       | #3
 1085        |       |
  514        |       |
--------    ---      |
  527                |
--------            ---
```

Figure 5: Clones 1, 3, and 4

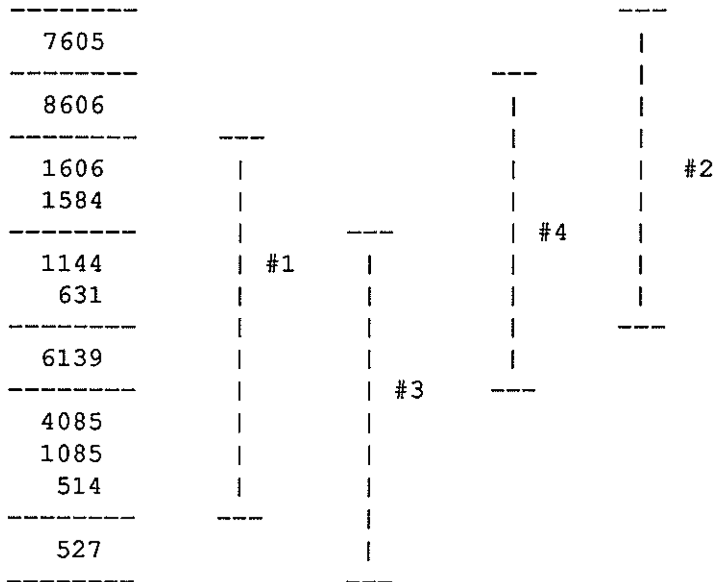


Figure 6: Clones 1, 2, 3, and 4

shown in Figure 7.

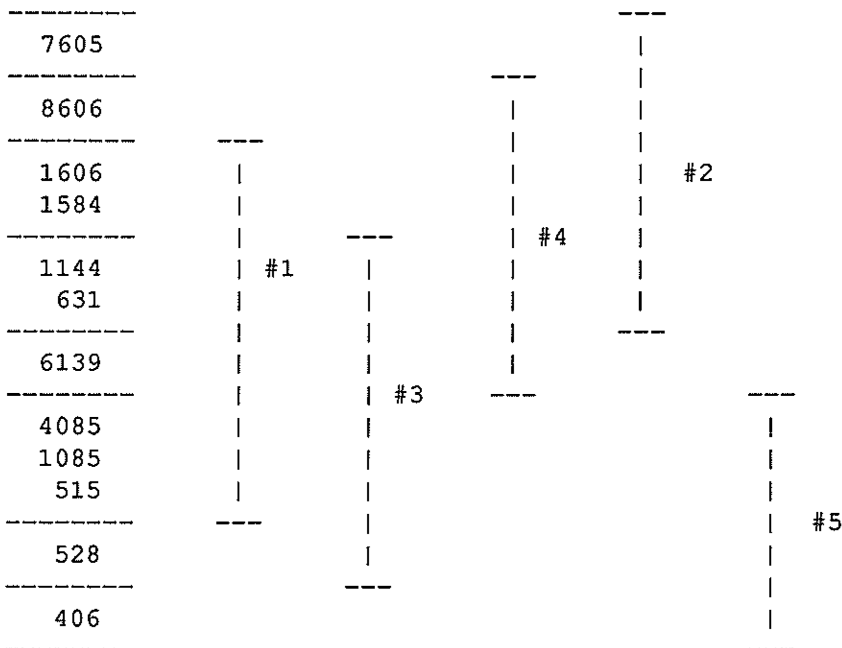


Figure 7: Completed mapping of clone set

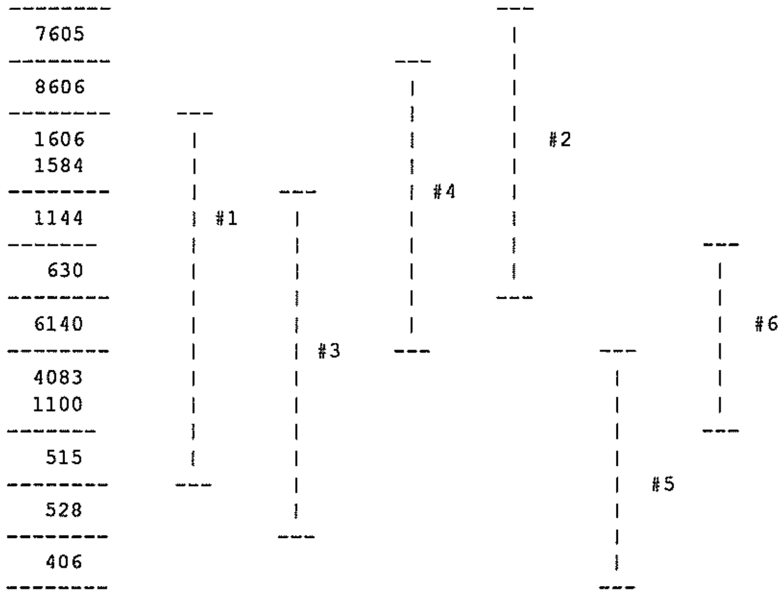
New clones can be incorporated into a map unit in two ways, *i.e.*, by **extension** or by **assimilation.** An **extension** has occurred if the number of fragments in the resulting map unit is greater than the number of fragments in the previous map unit, *i.e.*, some fragment of the clone extends beyond the boundaries of the original map unit. Each one of the clone incorporations performed in the previous example was an extension. An **assimilation** has occurred if the number of fragments in the resulting map unit is equal to the number of fragments in the previous map unit, *i.e.*, every fragment in the clone matched with an already existing fragment in the original map unit. In the previous example, the sequence of clone incorporations was <#1, #3, #4, #2, #5>. If instead, the order had been <#1, #3, #2, #4, #5>, then clone #4 would have been incorporated as an assimilation instead of as an extension.

It might be possible to incorporate a clone into a map unit in more than one way. Such a situation is referred to as **ambiguous.** As an example, assume that there is a sixth clone, Clone #6 with fragment set {6142, 4081, 1115, 629}, which is suspected to be from the same region of the genome as Clones #1 through #5. This new clone can be assimilated into the map unit shown in Figure 7. In fact, it can be assimilated in two different ways. The two possible fragmats are (631—629, 6139—6142, 4085—4081, 1085—1115) and (1144—1115, 631—629, 6139—6142, 4085—4081). Both of these are topologically feasible, and the corresponding map units are shown in Figure 8.

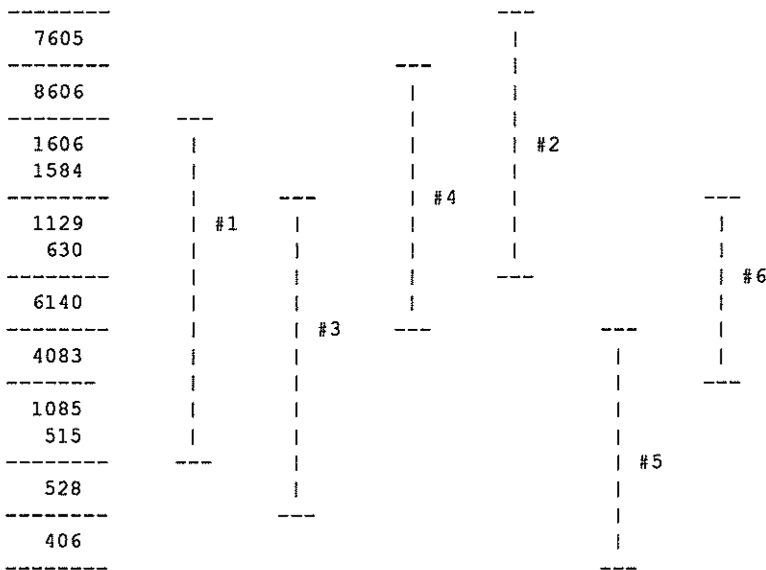
The structure of these two map units is significantly different. For instance, the map unit in Figure 8(a) contains one more group than the map unit in Figure 8(b). Also the map unit in Figure 8(a) restricts fragment 1144 to be adjacent to the group containing fragments 1584 and 1606, whereas the map unit in Figure 8(a) does not. Making a decision *now* about where a fragment *must* reside (when the decision is clearly in question) can have significant ramifications for the incorporation of subsequent clones not yet introduced. In such a case of ambiguous incorporation, a conservative approach is taken. That approach is to defer the incorporation of the clone, putting it aside to be addressed later. It is possible that the subsequent incorporation of other clones may add enough structure to the map unit that the deferred clone can be unambiguously incorporated later.

There are several forms of ambiguity. **External ambiguity** occurs when it is possible to incorporate a clone into a map unit in distinctly different regions (*i.e.*, sequence of groups) of the map unit. **Internal ambiguity** occurs when the clone can be incorporated in the same region in a number of different ways. There are two forms of internal ambiguity. The first is **similar match ambiguity**, which occurs when multiple fragmats allow map units of different structure to be constructed. This is illustrated by the example associated with Figure 8. The second is **subset ambiguity,** which occurs during assimilation when a clone is assimilated into a single group of a map unit and the fragments of the clone are a proper subset of the fragments of the group. As an example of this type of ambiguity, assume the state of clone mapping is as depicted in Figure 4, *i.e.*, only Clones #1 and #3 have been incorporated. Consider attempting to incorporate a new Clone #7, having fragment lengths {6158, 1151, 1079, 638}. This clone assimilates within the middle group of the map unit with fragmat (6154—6158, 1145—1151, 1085—1079, 634—638). Even though there is no fragment confusion involved, there are four different map units which can be constructed, as shown in Figure 9. Each of these map units represents a significantly different set of underlying realities, and none of the map units is compatible with any other.

This example was complex enough to demonstrate the general nature of the DNA mapping process. At first glance, DNA mapping may not appear to be a complex problem. However, the uncertainty about the validity of the fragment length-data along with the problem of determining the order in which a set of clones should be mapped together make the procedure a difficult one to automate effectively.



(a)
fragmat (631–629, 6139–6142, 4085–4081, 1085–1115)



(b)
fragmat (1144–1115, 631–629, 6139–6142, 4085–4081)

Figure 8: An ambiguous assimilation

```
--------   ---
  1614      |
  1592      |
--------    |      ---      ---
  6156      |       |        |
  1148      | #1    |        | #7
  1082      |       |        |
   636      |       |        |
--------    |       | #3    ---
  4085      |       |
   514      |       |
--------   ---      |
   527              |
--------           ---
```

(a)

```
--------   ---
  1614      |
  1592      |
--------    |      ---
  4085      |       |
--------    |       |       ---
  6156      |       |        |
  1148      | #1    |        | #7
  1082      |       |        |
   636      |       |        |
--------    |       | #3    ---
   514      |       |
--------   ---      |
   527              |
--------           ---
```

(b)

```
--------   ---
  1614      |
  1592      |
--------    |      ---
   514      |       |
--------    |       |       ---
  6156      |       |        |
  1148      | #1    |        | #7
  1082      |       |        |
   636      |       |        |
--------    |       | #3    ---
  4085      |       |
--------   ---      |
   527              |
--------           ---
```

(c)

```
--------   ---
  1614      |
  1592      |
--------    |      ---
  4085      |       |
   514      |       |
--------    |       |       ---
  6156      |       |        |
  1148      | #1    | #3     | #7
  1082      |       |        |
   636      |       |        |
--------   ---      |       ---
   527              |
--------           ---
```

(d)

Figure 9: Subset ambiguity

1.3. The Fragment Matching Mistake and Fragment Splitting

This section attempts to expose, by example, the fundamental fragment confusion error that often occurs during mapping, for which a fragment matching mistake detection and correction algorithm is needed to resolve. Consider the simple case in which two clones are being considered for possible incorporation. In this situation, the primary criterion for determining whether the two clones actually overlap is the maximum number of real fragment matches that can be constructed from the fingerprints of the two clones, i.e., the **apparent** overlap between the two clones. If this number is sufficiently high, then actual overlap is declared and the real fragment matches are used to produce virtual fragments within the resulting map.

It is natural to use the matchlist containing the most matches as the best approximation of the true overlap relationship between the clones. In this situation there is usually no reason to believe that a particular match is incorrect, since the desire is to produce the most compact map possible, given the data. However, just because two real fragments match (i.e., their measured lengths are within 3%) does not imply guarantee that they correspond to the same genomic fragment. A **fragment matching mistake** can occur when two real fragments of similar length occur in the genome roughly one clone length apart. This is also referred to as the **collapsed fragment error** (e.g., in Gillett)[3] to emphasize the fact that two real fragments from the underlying genome have been incorrectly collapsed into one fragment in the map being constructed. Consider the example illustrated in Figure 10.

The top horizontal line in Figure 10 represents a section of a genome. The middle horizontal line indicates the portion of the genome that Clone c_1 spans, and the bottom horizontal line indicates the portion of the genome that Clone c_2 spans. The small vertical lines on all of these horizontal lines represent restriction sites for the restriction enzyme being mapped. A number between two restriction sites indicates the number of base pairs between those two restriction sites, i.e., the length of the restriction fragment.

Given that Figure 10 represents reality, Figure 11 gives two fingerprints that might be obtained from gel electrophoresis applied to Clones c_1 and c_2. (Some random measurement error has been introduced.) Note that there are two *different* fragments of length 700 roughly one clone length apart which have the potential to be confused with each other. The two fragments that are of primary concern are denoted f_1 and f_2.

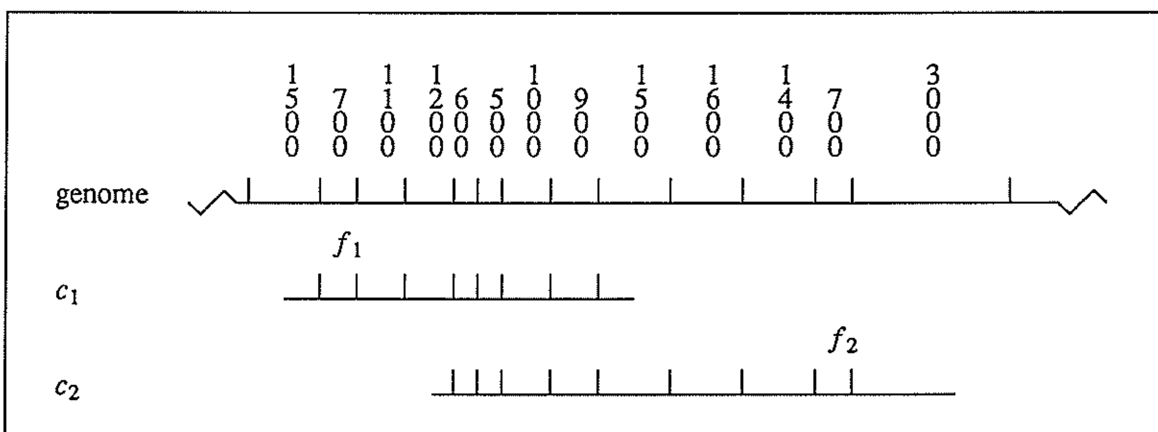


Figure 10: Two overlapping clones

c_1	c_2
505	497
610	601
705 (f_1)	692 (f_2)
912	902
1021	1002
1111	1411
1223	1523
	1630

Figure 11: The fingerprints of the clones in Figure 10

It is the data in Figure 11 that is used to determine the overlap between Clones c_1 and c_2. The maximum size matchlist is: [(505,497),(610,601),(705,692),(912,902),(1021,1002)]. (In this example the maximum size matchlist is unique, but in general there may be more than one.) There is no information available to indicate that real fragments f_1 and f_2 do not correspond to the same genomic fragment, and thus should not be matched. Thus this maximum matching would be used to form the Map *m* in Figure 12, where f_1 and f_2 are incorrectly matched together to form a virtual fragment of length 699.

Suppose that the mapper suspects that there is something wrong with *m*, and in particular with the virtual fragment $vf<f_1,f_2>$. If the mapper suspects that $vf<f_1,f_2>$ is the result of an incorrect match, he or she would want to decompose, or **split**, it into two virtual fragments, in order to undo the effects of the incorrect match. The Restricted Splitting Algorithm, or the RSA (described in Daues)[12], creates a new map by splitting a virtual fragment and then attempting to incorporate the subfragments that result from the split into the map. (The RSA can also create new maps by **combining** virtual fragments and then attempting to place the fragment that results into the map.) Given *m* and $vf<f_1,f_2>$, the RSA produces the map given in Figure 13. Note that this map corresponds to the underlying reality.

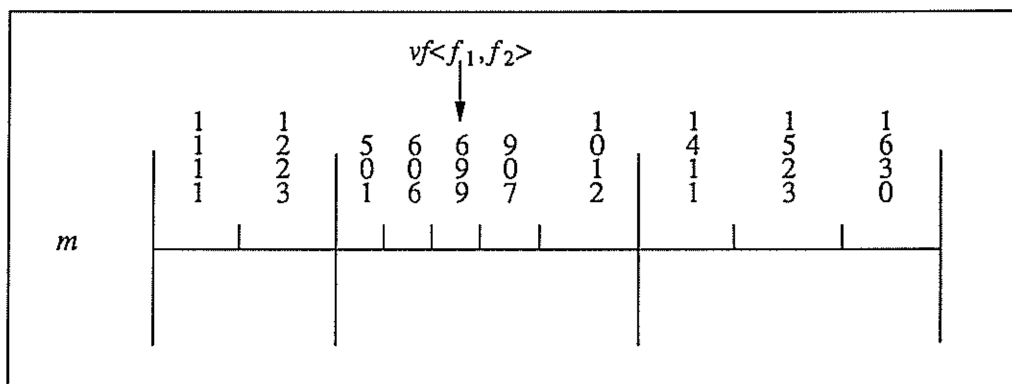


Figure 12: Map built from the fingerprints in Figure 11

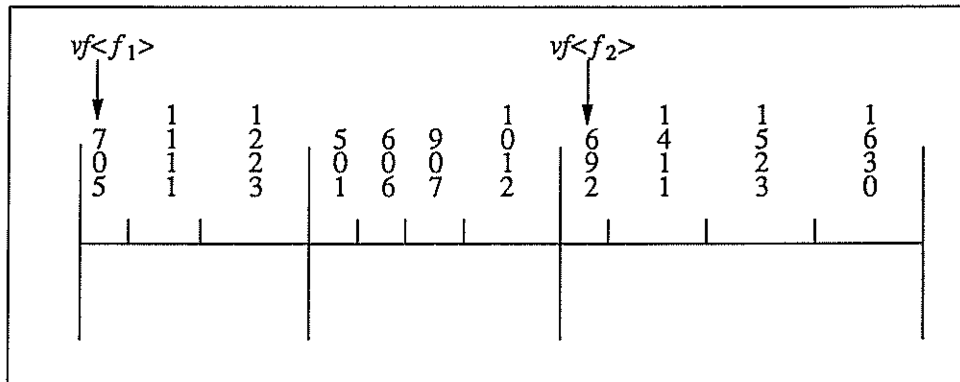


Figure 13: Map produced by the RSA

1.4. Overview of Fragment Matching Mistake Detection and Correction

Suppose m_1 and m_2 are two maps that should incorporate, but do not because one contains a fragment matching mistake. A common indicator of such a situation is when a large matchlist can be constructed between m_1 and m_2, but a small number of topological constraints are violated by the matchlist. In order to incorporate m_1 and m_2 one must (1) locate the fragment matching mistake and (2) repair the map containing the fragment matching mistake.

The RSA is designed to repair a map containing a fragment matching mistake given that the virtual fragment containing the mistake has been identified. However, the RSA cannot locate the fragment matching mistake. This report supplements the RSA by presenting a technique intended to locate a fragment matching mistake. In this section, a general overview of this technique is presented.

The reader is advised to read Daues[12] before attempting to understand the details embodied in this report. Many supporting concepts are presented and explained there. These concepts and their ramifications will be used here without any further discussion.

The focus of this report is an algorithm which will be referred to as FIX. FIX takes two windows, w_1 and w_2, as input. Let m_1 be the map associated with w_1 and let m_2 be the map associated with w_2. It is assumed that either m_1 or m_2 (but not both) contain a single fragment matching mistake. The output of FIX is a set of triples of the form (m_1', m_2', I). Map m_1' is either m_1 or a result of applying the RSA to m_1. Map m_2' is either m_2 or a result of applying the RSA to m_2. I is the set of all possible incorporations of m_1' and m_2' with windows similar to w_1 and w_2. FIX essentially returns ways that m_1 or m_2 can be repaired by one application of the RSA, so that the maps incorporate within w_1 and w_2.

FIX is a heuristic algorithm because several heuristics (including some within the RSA) are used to reduce the runtime and number of solutions returned. As with most heuristic algorithms, the correctness of FIX is not guaranteed. It is possible that FIX fails to find the solution that corresponds to reality.

FIX performs the function that the RSA does not, namely, the identification of which virtual fragment in the maps contains the fragment matching mistake. FIX locates the virtual fragment by performing a **topological scan** of m_1 and m_2. A topological scan produces a set of **almost topologically valid matchlists** (or ATVMLs) between m_1 and m_2. An ATVML is a matchlist that very nearly, but not quite, satisfies the topological constraints required to incorporate the two maps. After finding the ATVMLs, FIX determines which virtual fragments are probably responsible for the violation of topological constraints.

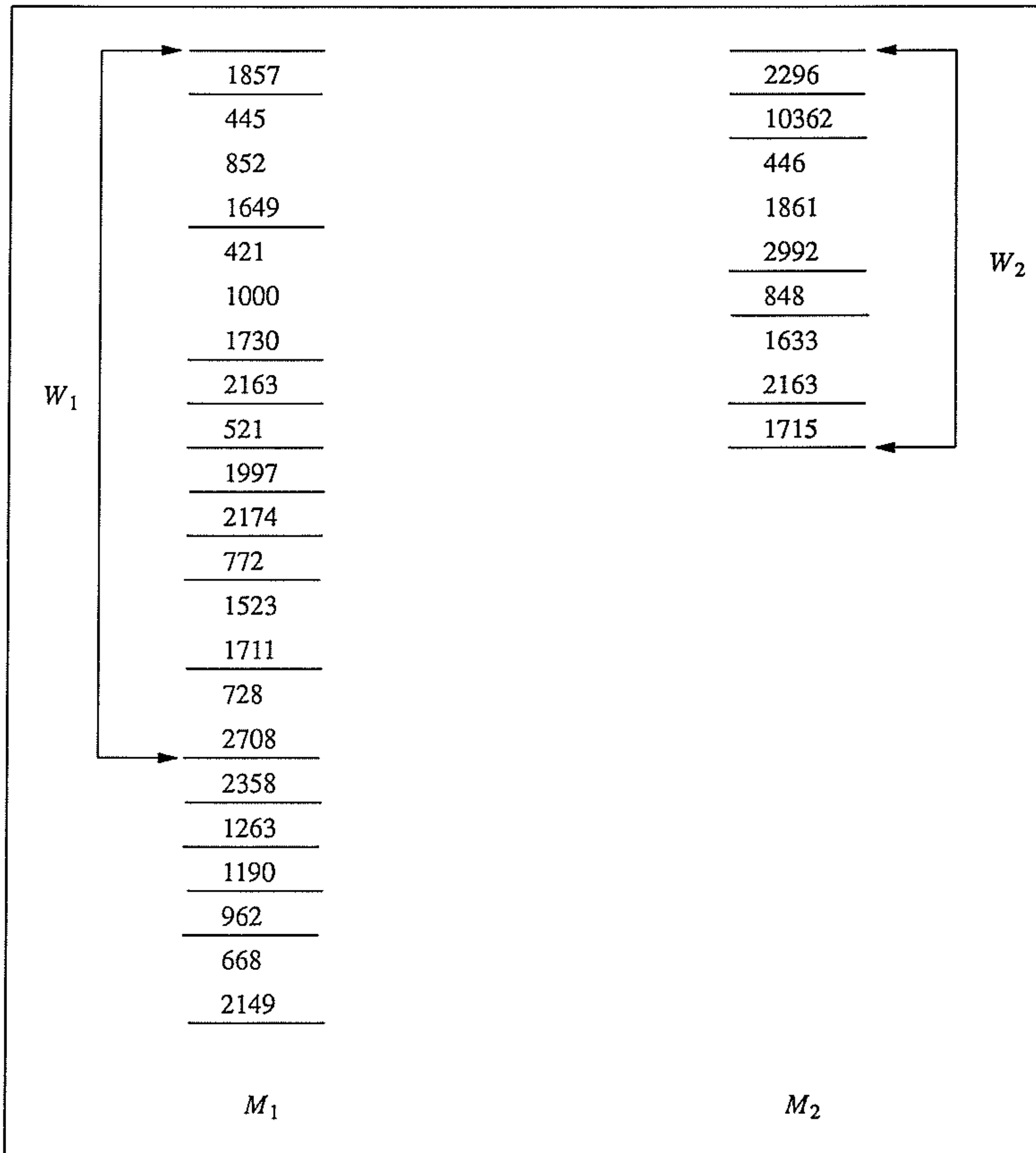


Figure 14: Maps M_1 and M_2

Once these virtual fragments are identified, FIX determines which virtual fragments are likely to contain the fragment matching mistake. This is accomplished through a detailed analysis of the effects that a fragment matching mistake has upon a map. Then the RSA can be applied to each of these virtual fragments. Finally, the results of the applying the RSA are collected and those results that allow incorporation are returned by FIX.

A running example is used throughout this report to help illustrate the concepts of FIX. Figure 14 illustrates two maps M_1 and M_2, with windows W_1 and W_2, respectively. W_1 and W_2 do not incorporate, although it appears there is a significant amount of fragment overlap between them. (Two windows

incorporate when their respective maps incorporate using only fragments found inside the windows.) It is possible that a fragment matching mistake in either M_1 or M_2 is preventing incorporation. Thus, it is appropriate to apply FIX to W_1 and W_2. The remainder of this section is devoted to a very general look at how FIX operates on W_1 and W_2.

In this report, many maps will be drawn vertically, instead of horizontally. Even so, the terms "left" and "right" are retained. In this context "left" refers to the top of the map and "right" refers to the bottom of the map. (I.e., when a map is drawn vertically, one starts listing the fragments at the left end of the map.)

By looking at the position of W_1 and W_2 within M_1 and M_2, it can be determined that there are four ways M_1 and M_2 might incorporate: (c_1) M_1 might extend to the right of M_2, (c_2) M_1 might extend to the right of M_2 with M_2 reversed, (c_3) M_2 might assimilate into M_1 or (c_4) M_2 might assimilate into M_1 with M_2 reversed. FIX attempts to construct ATVMLs for each way that M_1 and M_2 might incorporate. In this example, no ATVMLs are found using c_2, c_3 and c_4. However, an ATVML x is found using c_1 and is illustrated in Figure 15. Upon examining x, it is obvious that the virtual fragment of length 2163 in M_2 (denoted vf_1) is the source of the topological violation that prevents M_1 and M_2 from incorporating. No virtual fragment in M_1 matches with vf_1.

There are two approaches that can be used at this point. The first approach is to use the RSA to split vf_1. The subfragments of vf_1 might end up in locations where they do not interfere with incorporation in the way that vf_1 does. When the RSA is used to split vf_1, no maps are returned. Thus, the first approach fails for this example.

The second approach is to use the RSA to split some virtual fragment in M_1 that is within range of vf_1, with the hope that one of its subfragments will appear in a location where it matches with vf_1. The first step in this second approach is to find all virtual fragments in M_1 that are within range of vf_1. The virtual fragments of length 2163, 2174 and 2149 (denoted vf_2, vf_3 and vf_4, respectively in Figure 15) are within range of vf_1. No maps are returned when the RSA is used to split vf_2 and vf_4. However, three maps are returned when the RSA is used to split vf_3. Thus, FIX has three new versions of M_1 and determines which will incorporate with M_2. It turns out that only one of these three maps incorporates with M_2. This new map, M_1', is illustrated in Figure 16, along with the matchlist that allows incorporation and the map M_I that is the result of the incorporation. Thus, FIX returns $\{(M_1', M_2, \{M_I\})\}$.

In order to simplify the presentation of maps, clone ends are not shown. Without knowing the clone boundaries, it is impossible for the reader to verify that M_1' is the result of splitting vf_3 in M_1. The importance of clone boundaries and details about how fragments are split can be found in Daues[12]. Suffice it to say that as vf_3 was split, one component of length 2183 (denoted by a "*" in Figure 16) was added to the second group, previously containing {445, 852, 1649}, and the other component of length 2170 (denoted by a "+" in Figure 16) was added to the group previously containing {728, 2708}. Since the group in M_1 containing vf_3 contained only one fragment, it is no longer present in M_1'. The addition of a fragment to the second group, which is within range of vf_1 eliminates the topological barrier which originally caused the incorporation to fail.

The remainder of this report is structured in the following manner. In §2, the construction of ATVMLs (topological scanning) is discussed. In §3, the analysis of the effects of a fragment matching mistake that requires the splitting of a virtual fragment to repair is presented. In §4, the analysis of the effects of a fragment matching mistake that requires the combining of virtual fragments to repair is presented. In §5, the overall strategy of FIX is presented. In most sections, examples are used to clarify the concepts presented.

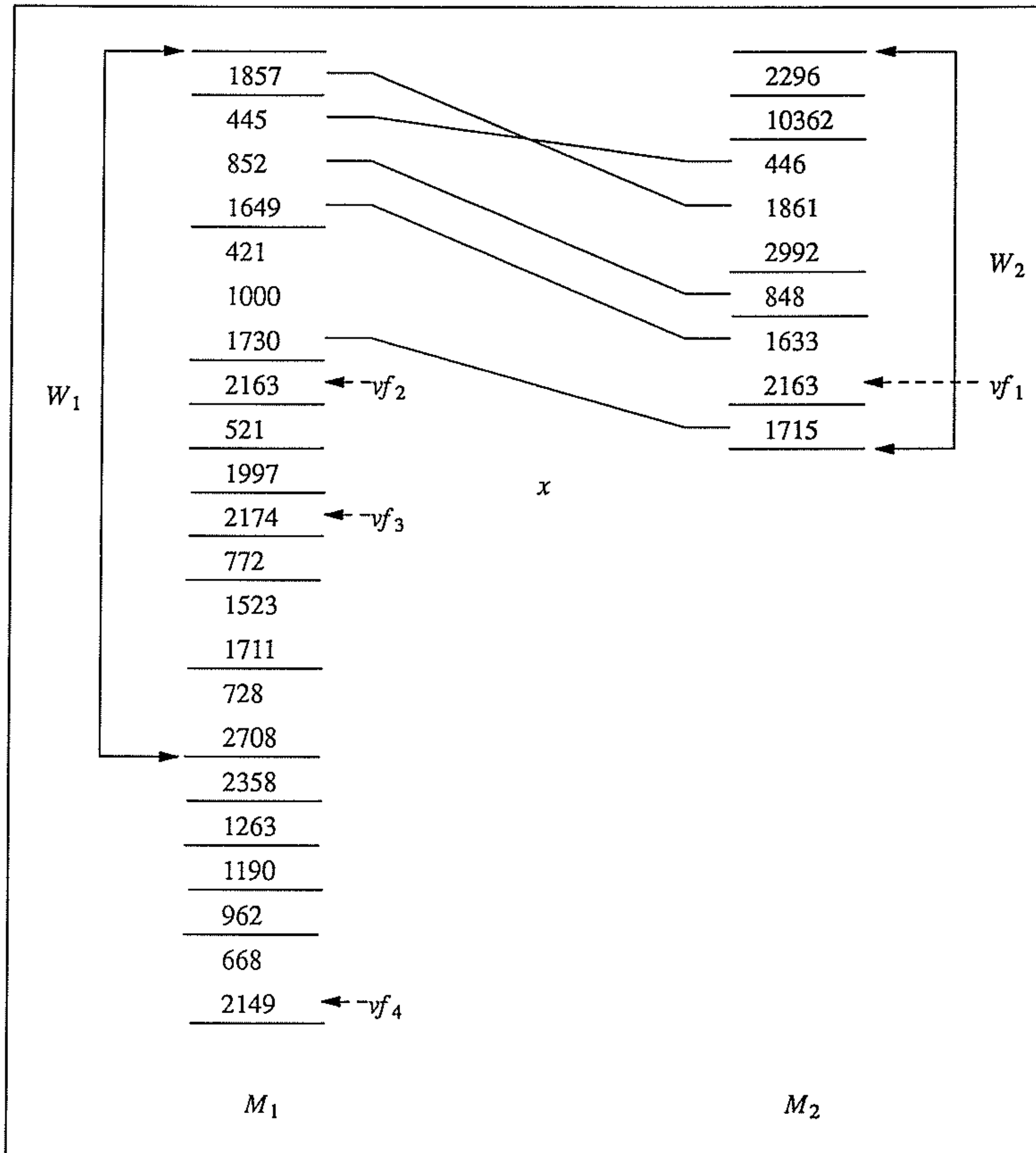


Figure 15: Maps M_1 and M_2 and an ATVML x

M_1'	M_2	M_I
1857	2296	2296
445	10362	10362
852	446	2992
1649	1861	1859
2183 *	2992	446
421	848	850
1000	1633	1641
1730	2163	2173
2163	1715	1723
521		421
1997		1000
772		2163
1523		521
1711		1997
728		772
2170 +		1523
2708		1711
2358		728
1263		2170
1190		2708
962		2358
668		1263
2149		1190
		962
		668
		2149

Figure 16: Maps M_1', M_2, M_I and a matchlist

2. Topological Scanning

2.1. Quick Overview

The purpose of a topological scan is to locate portions of the two maps that nearly incorporate and determine which virtual fragments prevent incorporation. This is accomplished by constructing ATVMLs. Before presenting the overview, a number of definitions are given.

The **orientation** of a map is either **asis** or **swap**. That is, one can use a map as it appears (asis) or one can reverse the order of the groups in the map (swap). The **overlap relationship** of maps m_1 and m_2 is the manner in which m_1 and m_2 might incorporate, given that the orientation of m_1 and m_2 are fixed. An overlap relationship has four possible states: (or_1) m_1 extends to the right of m_2, (or_2) m_2 extends to the right of m_1, (or_3) m_1 assimilates into m_2 and (or_4) m_2 assimilates into m_1. Finally, a **configuration** of m_1 and m_2 is the aggregate of the orientations of m_1 and m_2 and their overlap relationship. The configuration and overlap relationship of two windows is simply the configuration and overlap relationship of the maps associated with the windows.

Given two windows and a configuration, one can attempt to construct ATVMLs. There are two major subphases of topological scanning. First, all possible configurations of the given windows are found. (This is discussed in §2.2.) Then, ATVMLs are constructed for each configuration. (This is discussed in §2.3.) More definitions will be useful in order to define the output of the topological scan.

Given an ATVML, x, the virtual fragments in the two maps can be partitioned into four **classes**. Let w be the minimum size window in the map m that contains all virtual fragments in x. A virtual fragment vf in m is a **class 1** virtual fragment iff vf is not within w. A virtual fragment vf in m is a **class 2** virtual fragment iff vf is in x. A virtual fragment vf in m is a **class 3** virtual fragment iff vf is within w but not in x, and would have to be removed from m in order for x to be topologically valid. A virtual fragment vf in m is a **class 4** virtual fragment iff vf is within w but not in x, and does not have to be removed from m in order for x to be topologically valid. Class 3 virtual fragments are also called **discarded** virtual fragments. Class 4 virtual fragments are also called **ignored** virtual fragments. The class of a virtual fragment is very important when FIX determines which virtual fragments should be split or combined. (This is discussed in more detail in §3 and §4.) In Figure 17, the ATVML of the running example first presented in §1.4 is illustrated, along with the minimum size windows and an example of each virtual fragment class.

Topological scanning returns a set of tuples of the form (cf,x,d_1,d_2). The tuple element cf is a configuration, x is an ATVML, d_1 is the set of discarded virtual fragments (class 3) in the first map and d_2 is the set of discarded virtual fragments in the second map. Note that the class 1, class 2 and class 4 virtual fragments can be computed given cf, x, d_1 and d_2.

For the running example, topological scanning returns a single tuple, where the configuration has M_1 and M_2 asis with M_1 extending to the right of M_2, the ATVML is [(445,446),(852,848),(1649,1633),(1730,1715),(1857,1861)], $d_1 = \varnothing$ and $d_2 = \{2163\}$.

The pseudocode for the top level function in topological scanning, find_all_scans, is given in Figure 18. find_all_scans takes two windows as input and returns a set of objects of type SCAN. SCAN is a record-like structure with four fields: configuration, atvml, discards_1 and discards_2. The field configuration is an object of type CONFIGURATION, which is a record-like structure as well. CONFIGURATION contains three fields: orientation_1, orientation_2 and orelationship. The field orientation_1 is the orientation of map associated with the first window and has one of two values: ASIS or SWAP. The field orientation_2 is defined similarly for the map associated with the second window. The field orelationship is the overlap relationship of the two maps. The field atvml in the type SCAN is a list of pairs of virtual fragments, and discards_1 and discards_2 are sets of virtual fragments.

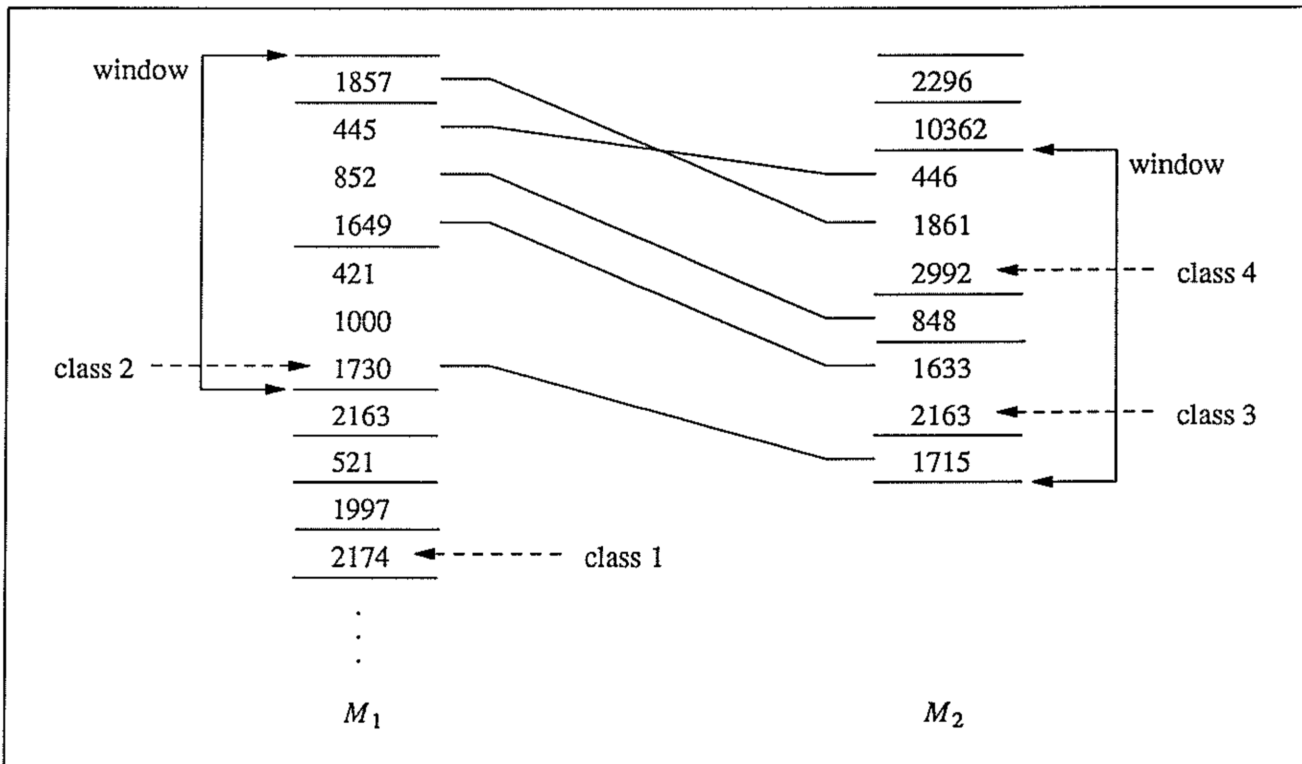


Figure 17: Examples of the virtual fragment classes

```
SET
find_all_scans(w₁,w₂)
    WINDOW          w₁,w₂;
{
    SCAN            s;
    PARTIAL_SCAN    ps;
    CONFIGURATION   c;
    SET             ans,pss,cs;

    ans ← ∅;
    cs ← find_all_valid_configurations(w₁,w₂);

    for c ∈ cs do
        pss ← topological_scan_fixed_configuration(w₁,w₂,c);

        for ps ∈ pss do
            s.configuration ← c;
            s.atvml ← ps.atvml;
            s.discards₁ ← ps.discards₁;
            s.discards₂ ← ps.discards₂;
            ans ← ans ∪ {s};
        rof
    rof

    return(ans);
}
```

Figure 18: Pseudocode for find_all_scans

find_all_scans first finds all valid configurations for the two windows by calling the function find_all_valid_configurations. (The operation of find_all_valid_configurations is examined in the next section.) Then for each valid configuration, it calls the function topological_scan_fixed_configuration (see Figure 19). topological_scan_fixed_configuration returns a set of PARTIAL_SCAN type objects (exact structure given in §2.3.2) which represents the result of a topological scan of the two windows for a particular configuration. However, the PARTIAL_SCANs contain information that is no longer necessary once the scan is complete. Thus, after each call to topological_scan_fixed_configuration, find_all_scans extracts the important information from each PARTIAL_SCAN to create a set of SCANs, which it then returns.

```
SET
topological_scan_fixed_configuration(w₁,w₂,c)
    WINDOW          w₁,w₂;
    CONFIGURATION   c;
{
    SET             ans,scans_LEFT,scans_RIGHT;

    scans_LEFT ← topological_scan_fixed_direction(w₁,w₂,c,LEFT);
    scans_RIGHT ← topological_scan_fixed_direction(w₁,w₂,c,RIGHT);

    ans ← scans_LEFT ∪ scans_RIGHT;
    return(ans);
}
```

Figure 19: Pseudocode for topological_scan_fixed_configuration

The next two sections concentrate on the two components of find_all_scans. In §2.2, the computation of all valid configurations is examined. In §2.3, the topological scanning of two windows for a particular configuration is examined.

2.2. Finding All Valid Configurations

In this section, the algorithm for taking two windows and determining all valid configurations for those windows is presented. Depending on the nature of the windows, there are certain restrictions on the possible configurations. This algorithm looks at the position of each window with respect to its map and then determines the valid configurations for those two windows.

A window w is an **umbrella** window if the leftmost group in w is the leftmost group in the map and the rightmost group in w is the rightmost group in the map. A window w is a **left end** window if the leftmost group in w is the leftmost group in the map and the rightmost group in w is not the rightmost group in the map. A window w is a **right end** window if the leftmost group in w is not the leftmost group in the map and the rightmost group in w is the rightmost group in the map. A window w is a **middle** window if the leftmost group in w is not the leftmost group in the map and the rightmost group in w is not the rightmost group in the map. Any window must be precisely one of the above types.

Thus, for two windows w_1 and w_2, there are sixteen possible combinations of window types. Some combinations are invalid (i.e., there is no configuration that allows the windows to incorporate). Theorems 1, 2 and 3 prove the invalidity of five particular combinations.

Theorem 1: If w_1 and w_2 are middle windows, then w_1 and w_2 cannot incorporate.

Proof: Let w_1 and w_2 be middle windows with a fixed configuration. Let g_1 be the group immediately to the left of w_1. Let g_2 be the group immediately to the left of w_2. (Both g_1 and g_2 must exist because w_1 and w_2 are middle windows.) In order for w_1 and w_2 to incorporate, g_1 and g_2 must have some overlap. However, since neither is in a window, that is impossible. Thus, w_1 and w_2 cannot incorporate. □

Theorem 2: If w_1 is a right end window and w_2 is a middle window, then w_1 and w_2 cannot incorporate.

Proof: Let w_1 be a right end window and let w_2 be a middle window with a fixed configuration. Let g_1 be the group immediately to the left of w_1. Let g_2 be the group immediately to the left of w_2. (Both g_1 and g_2 must exist.) In order for w_1 and w_2 to incorporate, g_1 and g_2 must have some overlap. However, since neither is in a window, that is impossible. Thus, w_1 and w_2 cannot incorporate. □

Theorem 3: If w_1 is a left end window and w_2 is a middle window, then w_1 and w_2 cannot incorporate.

Proof: Similar to Theorem 2. □

In addition to these combinations being invalid, other combinations restrict the possible configurations of the two windows. Theorem 4 proves the existence of restrictions in one particular combination.

Theorem 4: If w_1 is a middle window and w_2 is an umbrella window, then w_1 cannot extend w_2.

Proof: Let w_1 be a middle window and let w_2 be an umbrella window. Assume w_1 extends to the

right of w_2. Then there must be overlap between w_2 and the leftmost group g in the map associated with w_1. However, g is not in w_1. Thus, incorporation is impossible. A similar argument holds if w_1 extends to the left of w_2. □

The results of these theorems are summarized in Table 1. Entries in the table marked "INV" indicate that the particular combination of window types is invalid. Otherwise, the entry contains triples, one for each valid configuration, of the form (orientation$_1$,orientation$_2$,orelationship).

The function find_all_valid_configurations takes two windows and returns a set of CONFIGURATION type objects, which represents the set of valid configurations for the given windows.

The computation of find_all_valid_configurations for the running example is now examined. W_1 is a left end window and W_2 is an umbrella window. Thus, W_2 could assimilate into W_1, or W_1 could extend right of W_2. In either case, the orientation of W_2 could be asis or swap. (Note that one could vary the orientation of W_1 also, but in this case it has been arbitrarily decided that the orientation of W_2 is varied.) Thus, there are four valid configurations, and the set returned by find_all_valid_configurations is $\{cf_1,cf_2,cf_3,cf_4\}$, where

$cf_1 \equiv$ lorientation$_1$ = ASIS, orientation$_2$ = ASIS, orelationship = OR1l,
$cf_2 \equiv$ lorientation$_1$ = ASIS, orientation$_2$ = SWAP, orelationship = OR1l,
$cf_3 \equiv$ lorientation$_1$ = ASIS, orientation$_2$ = ASIS, orelationship = OR4l,
$cf_4 \equiv$ lorientation$_1$ = ASIS, orientation$_2$ = SWAP, orelationship = OR4l.

$w_1 \rightarrow$ $w_2 \downarrow$	left end	right end	umbrella	middle
left end	INV	(asis,asis,or_2)	(asis,asis,or_2) (swap,asis,or_2) (asis,asis,or_3) (swap,asis,or_3)	INV
right end	(asis,asis,or_1)	INV	(asis,asis,or_1) (swap,asis,or_1) (asis,asis,or_3) (swap,asis,or_3)	INV
umbrella	(asis,asis,or_1) (asis,swap,or_1) (asis,asis,or_4) (asis,swap,or_4)	(asis,asis,or_2) (asis,swap,or_2) (asis,asis,or_4) (asis,swap,or_4)	(asis,*any*,*any*)	(asis,asis,or_4) (asis,swap,or_4)
middle	INV	INV	(asis,asis,or_3) (swap,asis,or_3)	INV

Table 1
Valid configurations for types of windows

2.3. Finding ATVMLs for a Particular Configuration

The details of computing ATVMLs for a fixed configuration are presented in this section. However, it is helpful to examine the motivation and flavor of the algorithm with an informal example of the computation of an ATVML before the details and pseudocode are presented. This example is presented in §2.3.1. The pseudocode and a more formal example are given in §2.3.2.

2.3.1. An Informal Example

Consider the running example once again. The traditional method for finding matchlists between two windows is to aggregate the virtual fragments from each window into a single set, and then perform a large number of pairwise comparisons between a fragments from one set and fragments from the other set. For example, the set of virtual fragments from W_1 is

{421,445,521,728,772,852,1000,1523,1649,1711,1730,1857,1997,2163,2174,2708}

and the set of virtual fragments from W_2 is

{446,848,1633,1715,1861,2163,2296,2992,10362}.

Then there are four maximum size matchlists that can be formed from these two sets, one of which is

[(445,446),(852,848),(1649,1633),(1730,1715),(1857,1861),(2163,2163)].

The problem is that although it is easy to determine that this matchlist is not topologically valid, it is difficult to determine why it is not topologically valid. When the virtual fragments from different groups are aggregated into the same set, much topological constraint information is lost. This information cannot be regained from the matchlist. If the matchlist is constructed in a way such that this information is not lost, then perhaps the reasons for topological constraint violations will be clearer.

One way to accomplish this is to bring matchlist construction down to the group level. The idea is to pick a starting group in each window and a direction, and then move in the selected direction group by group, building an overall matchlist from the best matchlists that can be constructed between the current groups. If a point is reached where matchlist construction cannot continue due to a topological constraint violation caused by a single virtual fragment, then that virtual fragment is discarded (conceptually removed from the map). Then matchlist construction continues. If a matchlist of significant size can be constructed by discarding a small number of virtual fragments, then this matchlist is an ATVML. If a large number of virtual fragments must be discarded in order to eliminate topological violations before a matchlist of significant size is constructed, then no ATVML is constructed.

Consider the configuration cf_1, the starting group {1857} in W_1, the starting group {446,1861,2992} in W_2, and the right scan direction. A maximum size matchlist that can be constructed between the two groups is [(1857,1861)]. The virtual fragments that did not match are examined to see if they violate topological constraints. All virtual fragments in {1857} are part of the matchlist, but there are two virtual fragments in {446,1861,2992} that are not. However, these two virtual fragments could lie to the left of W_1. Thus, they do not violate topological constraints. Since {1857} is used entirely by the matchlist, the group immediately to the right, {445,852,1649}, is considered. Now a maximum size matchlist is constructed between {445,852,1649} and {446,2992}, the set of virtual fragments in the current group of W_2 that have not matched yet. The matchlist constructed is [(445,446)], which can be appended to the previously constructed matchlist to form [(1857,1861),(445,446)].

At this point, there are virtual fragments in both windows that have not matched and the matchlist constructed is too small to be considered significant. However, the virtual fragment not matched in W_2, namely 2992, could lie to the left of W_1 (i.e., it can be ignored). Assuming this is true, matchlist construction can continue by moving to the group immediately to the right of {446,1861,2992}, namely {848}.

The maximum matchlist constructed between {852,1649} and {848} is [(852,848)]. Now the overall matchlist is [(1857,1861),(445,446),(852,848)]. There are no unmatched virtual fragments in W_2, so the group immediately right of {848} is considered, namely {1633,2163}. The maximum matchlist constructed between {1649} and {1633,2163} is [(1649,1633)]. Now the overall matchlist is [(1857,1861),(445,446),(852,848),(1649,1633)]. There are no unmatched virtual fragments in W_1, so the group immediately right of {445,852,1649} is considered, namely {421,1000,1730}. No non-empty matchlists exist between {421,1000,1730} and {2163}.

Unlike the previous instance where unmatched virtual fragments existed in both windows, further matchlist construction is hindered by a topological constraint at this point. 2163 (in W_2) cannot lie in the stretch of genome outside of W_1, nor can 421, 1000 or 1730 (in W_1) lie outside of W_2. If one scans W_2 to the right a bit, the virtual fragment 1715 is encountered. This could match with 1730. Thus, if 2163 is discarded, matchlist construction could continue. A similar situation exists in the other map as well. If 421, 1000 and 1730 are discarded, then 2163 in W_2 could match with 2163 in W_1. However, it is desirable to discard as few virtual fragments as possible. Thus, the preferred choice at this point is to discard 2163 (in W_2) and move on to the next group in W_2. It is important to note how looking ahead helps decide the best course of action. The importance of this will become clear in §2.3.2.

Then the maximum matchlist between {421,1000,1730} and {1715} is [(1730,1715)], and the overall matchlist is [(1857,1861),(445,446),(852,848),(1649,1633),(1730,1715)]. At this point, W_2 ends. The overall matchlist is of significant size. Thus, it is an ATVML and 2163 in W_2 is the only discarded virtual fragment. (This ATVML corresponds to the one illustrated in Figure 17.)

This quick example exposes the flavor of the topological scanning algorithm, but it illustrates only a small part of the computation necessary. For example, other ATVMLs might be found if (1) different starting groups are selected, (2) a different scan direction is selected and (3) more than one matchlist is found between two groups. In addition, there are many details left unspecified, such as (1) the exact method for deciding what virtual fragments are discarded or ignored, (2) the maximum number of discarded virtual fragments allowed and (3) how to handle multiple maximum size matchlists. These issues are discussed in more detail in the next section.

2.3.2. More Formal Discussion and Example

Given two windows and a configuration, there are still two choices to be made before ATVMLs can be constructed. The first is to select a scan direction and the second is to select starting groups. First, the scan direction is discussed.

It may seem that it is not necessary for a map to be scanned in both directions. However, it is necessary to scan in both directions. To understand why, consider Figures 20 and 21. In both figures, two windows and an ATVML are illustrated. Each virtual fragment in the windows is denoted by a letter, such that two virtual fragments are within range if and only if they are denoted by the same letter. Figure 20 illustrates an ATVML constructed by starting at the leftmost group in each window ($\{A\}$) and scanning the maps in the right direction. Figure 21 illustrates an ATVML constructed by starting at the rightmost group in each window ($\{E\}$) and scanning the maps in the left direction. The ATVMLs are slightly different, depending on the direction of the topological scan. In addition, the virtual fragments which are discarded are different. Thus, the direction of the topological scan can affect the result of the scan. Therefore, it is necessary to scan in both directions.

Now the issue of selecting starting groups is discussed. Assuming a scan direction has been chosen, one must still select the group in each window at which to begin the scan. In the examples in Figures 20 and 21, the groups at the end of the window were selected as the starting groups. However, in §2.3.1, the starting group in W_2 was not the group at the end of W_2. In general, it cannot be known beforehand which

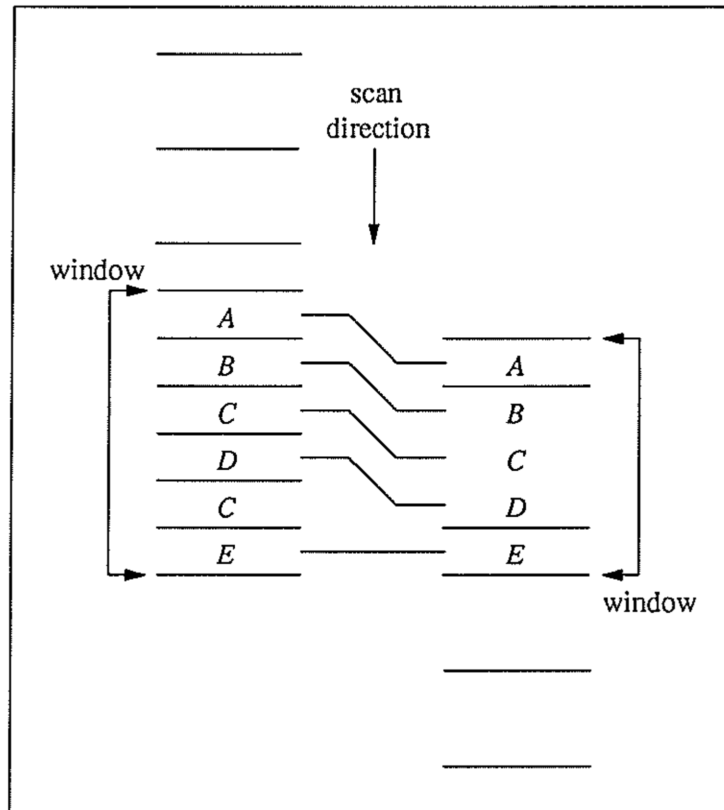


Figure 20: Result of the scan in the right direction

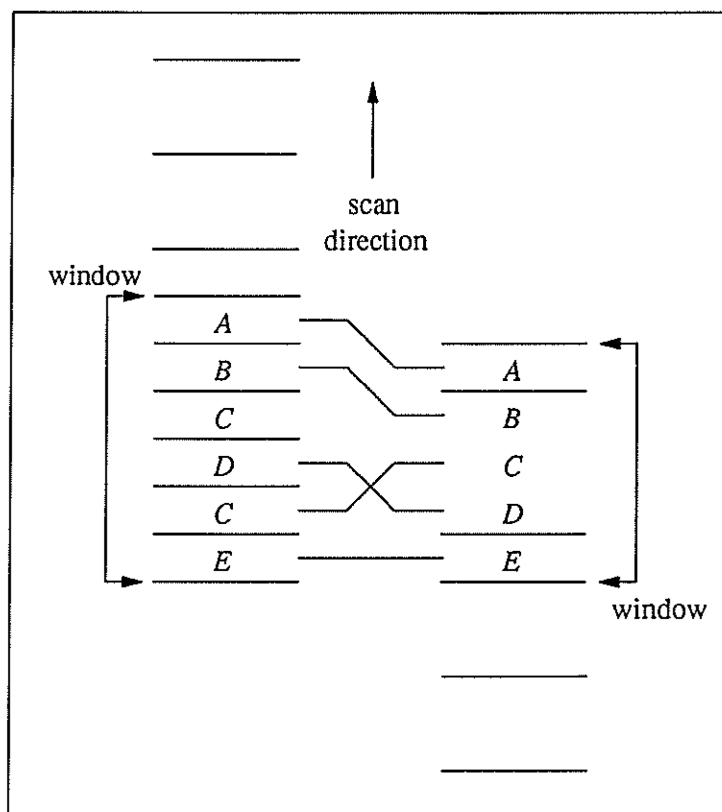


Figure 21: Result of the scan in the left direction

starting groups will lead to ATVMLs. Thus, more than one possibility must be considered. The overlap relationship of the two windows determines which possibilities must be attempted.

Suppose that the overlap relationship between windows w_1 and w_2 is that w_2 assimilates into w_1. This overlap relationship places certain constraints on the virtual fragments in the windows. For instance, the leftmost virtual fragment of w_2 cannot be to the left of the leftmost virtual fragment of w_1. Likewise, the rightmost virtual fragment of w_2 cannot be to the right of the rightmost virtual fragment of w_1. It is said that w_1 **starts before** (i.e., is more to the left than) w_2 and **ends after** (is more to the right than) w_2. In an assimilation, all virtual fragments in the smaller window must match with some other virtual fragment. So unless a virtual fragment in w_2 is discarded, it must be in the ATVML. This implies that virtual fragments in w_2 cannot be ignored.

It is known that the leftmost group of w_2 overlaps with some group in w_1, but which group is not known. In addition, there is no heuristic for determining a group that is preferred for w_2 to overlap. Thus, the starting group of w_2 is always the leftmost group and the starting group of w_1 could be any group in w_1, as long as w_1 still ends after w_2. (This is illustrated in Figure 22.) The dashed lines indicate the groups with which the leftmost group in w_2 must be paired as the starting group in w_1.

Suppose that the overlap relationship between w_1 and w_2 is that w_2 extends to the right of w_1. This implies that w_1 starts before w_2 and w_2 ends after w_1. It is known that the leftmost group of w_2 overlaps with some group in w_1, but which group is not known. In this case, there is a heuristic for determining a group that is preferred for w_2 to overlap. Similar to before, the starting group of w_2 is always the leftmost group and the starting group of w_1 could be any group in w_1, as long as w_2 ends after w_1. (This is

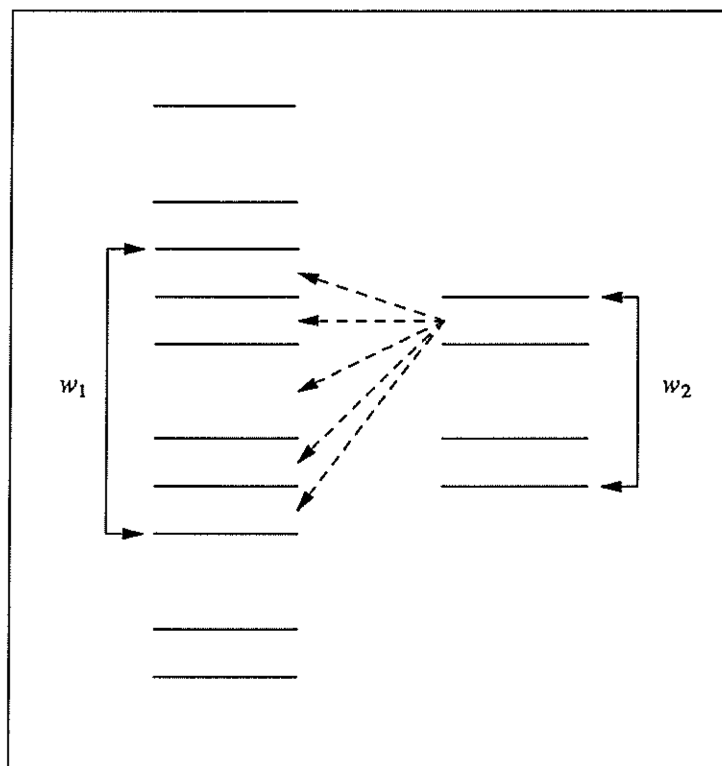


Figure 22: Possible starting groups for an assimilation

illustrated in Figure 23.) The difference is that given a choice between different ways of extending w_1 with w_2, the one that produces the greatest overlap is preferred. Thus, one first attempts to construct an ATVML starting with the leftmost group in w_1. If an ATVML is constructed, then the topological scan for this direction is terminated. If no ATVML is constructed, then the next to leftmost group in w_1 is used as the starting group. This step is repeated until an ATVML is constructed or until all groups within w_1 have been used as the starting group. With this technique, the greatest possible overlap between w_1 and w_2 is obtained.

Thus, in order to be sure that most possibilities for constructing ATVMLs have been explored, the topological scan must be performed in both directions (left and right) and with all possible starting groups that are applicable for the given configuration. This is reflected in the pseudocode for the function topological_scan_fixed_configuration (see Figure 19) and the function topological_scan_fixed_direction (see Figure 24).

The function topological_scan_fixed_configuration takes two windows and a configuration as input and returns a set of PARTIAL_SCAN type objects, which represents the result of all topological scans for this given configuration. It accomplishes this by calling topological_scan_fixed_direction twice. It specifies the scan direction as left in the first call and as right in the second call. The set of PARTIAL_SCAN type objects that are returned by each call to topological_scan_fixed_direction are unioned and returned.

The function topological_scan_fixed_direction takes two windows, a configuration and a direction as input and returns a set of PARTIAL_SCAN type objects, which represents the result of all topological scans for the given configuration and direction. It is in topological_scan_fixed_direction that the

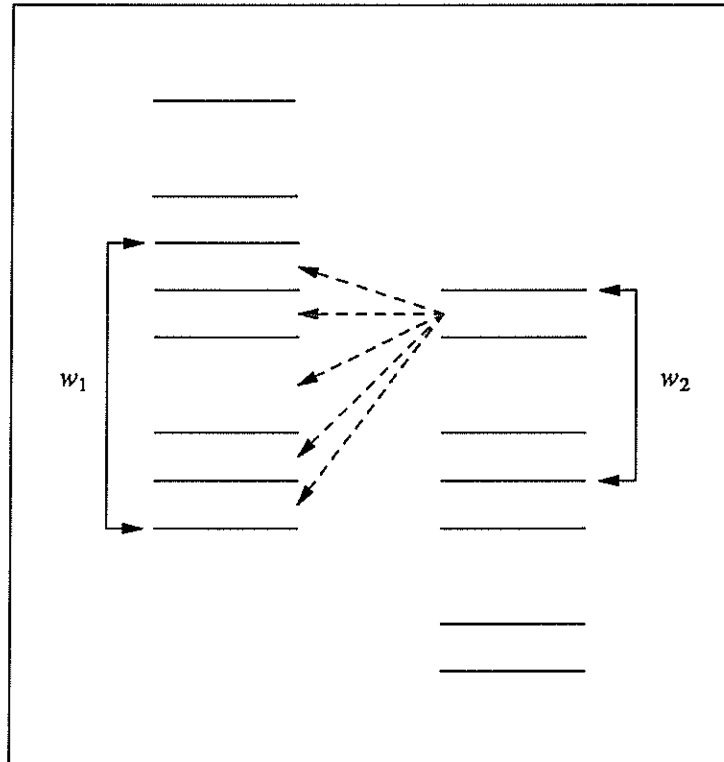


Figure 23: Possible starting groups for an extension

```
SET
topological_scan_fixed_direction(w1,w2,c,dir)
    WINDOW          w1,w2;
    CONFIGURATION   c;
    DIRECTION       dir;
{
    INTEGER         starts,ends;
    SET             ans;
    LIST            ss1,ss2;

    ans ← Ø;
    ss1 ← window_find_vfrag_setseq(w1); ss2 ← window_find_vfrag_setseq(w2);
    if (c.orientation1 = SWAP xor dir = LEFT) then ss1 ← list_reverse(ss1); fi
    if (c.orientation2 = SWAP xor dir = LEFT) then ss2 ← list_reverse(ss2); fi

    case
        subcase (c.orelationship = OR3)
            starts ← 2; ends ← 2;
        subcase (c.orelationship = OR4)
            starts ← 1; ends ← 1;
        subcase ((c.orelationship = OR1 and dir = RIGHT) or
                 (c.orelationship = OR2 and dir = LEFT))
            starts ← 2; ends ← 1;
        subcase ((c.orelationship = OR1 and dir = LEFT) or
                 (c.orelationship = OR2 and dir = RIGHT))
            starts ← 1; ends ← 2;
    esac

    if (c.orelationship = OR3 or c.orelationship = OR4)
        then
            while (ss1 ≠ [] and ss2 ≠ []) do
                ans ← ans ∪ topological_scan_fixed_start(ss1,ss2,starts,ends);
                ss_starts ← ss_starts[2,...];
            od
        else
            while (ss1 ≠ [] and ss2 ≠ [] and ans = Ø) do
                ans ← topological_scan_fixed_start(ss1,ss2,starts,ends);
                ss_starts ← ss_starts[2,...];
            od
    fi

    return(ans);
}
```

Figure 24: Pseudocode for topological_scan_fixed_direction

different starting groups are addressed. By examining the overlap relationship and the scan direction, topological_scan_fixed_direction determines which window should start first and which window should end later. In addition, topological_scan_fixed_direction isolates the virtual fragments of interest by extracting the **virtual fragment set sequence** (or **VFSS**) of each window. The VFSS of a window w is a list of sets of virtual fragments that corresponds to the partial order of the virtual fragments within the window. For example, the VFSS of W_2 is

$$[\{2296\},\{10362\},\{446,1861,2992\},\{848\},\{1633,2163\},\{1715\}].$$

The VFSS is used at this point (1) to make the virtual fragment information easier to manipulate and (2) to allow such manipulation without modifying the original windows.

Then topological_scan_fixed_direction calls topological_scan_fixed_start (see Figure 25) with the proper VFSSs and starting positions until each starting group has been used (if the overlap relationship indicates an assimilation) or until a scan succeeds in constructing at least one ATVML (if the overlap relationship indicates an extension).

```
SET
topological_scan_fixed_start(ss1,ss2,starts,ends)
    LIST            ss1,ss2;
    INTEGER         starts,ends;
{
    INTEGER         not_ends;
    LIST            ml;
    STACK           s;
    PARTIAL_SCAN    ps;
    SET             ans,mls,lfs1,lfs2;

    ans ← ∅;
    if (ends = 1) then not_ends ← 2; else not_ends ← 1; fi
    ps ← find_initial_partial_scan(ss1,ss2);
    s ← empty_stack;
    stack_push(s,ps);

    while (s ≠ empty_stack) do
        ps ← stack_pop(s);

        if (ps.cfs1 = NULL or ps.cfs2 = NULL)
            then
                if (|ps.atvml| ≥ MIN_OVERLAP and ps.ss_not_ends = [])
                    then ans ← ans ∪ {ps};
                fi
            else
                mls ← find_best_matchlists(ps.cfs1,ps.cfs2);

                for ml ∈ mls do
                    lfs1 ← ps.cfs1 — left_elements_of(ml);
                    lfs2 ← ps.cfs2 — right_elements_of(ml);

                    if (lfs1 = ∅ or lfs2 = ∅)
                        then s ← scan_right(ml,ps,s,lfs1,lfs2,starts);
                        else
                            if (ps.first_set)
                                then s ← fix_by_ignore(ml,ps,s,starts);
                                else s ← fix_by_discard(ml,ps,s,lfs1,lfs2);
                            fi
                    fi
                rof
        fi
    od

    return(ans);
}
```

Figure 25: Pseudocode for topological_scan_fixed_start

Before examining topological_scan_fixed_start in detail, a number of definitions are useful. Recall that in the example of §2.3.1, the maximum size matchlists between a set of virtual fragments from each window were computed. The set of virtual fragments from a window that maximum size matchlists are computed from at a particular time is called the **current virtual fragment set** (or **current fragment set** or **CFS**) of the window. Matchlists computed between CFSs are called **local** matchlists. Also recall that after a maximum size matchlist was computed, the virtual fragments that did not appear in a local matchlist were of importance. The set of virtual fragments in a current fragment set s that do not appear in a local matchlist x is the **leftover virtual fragment set** (or **leftover fragment set** or **LFS**) of s and x. The CFS and LFS are used to keep track of where the scan is and how it should continue.

The scheme used in topological_scan_fixed_start is as follows. Initially, there is one ATVML which is empty and the CFS of each VFSS is the set of virtual fragments in the starting group. All maximum size local matchlists between the CFSs of each window is computed. Each local matchlist is appended to the previous ATVML to form a new and distinct ATVML and the LFS for each window with respect to each local matchlist is computed. The LFS corresponding to each new ATVML is examined. Let *ml* be the ATVML corresponding to some LFS. If at least one of the LFSs is empty, then there are no topological problems introduced by *ml*. New CFSs are computed and the topological scan continues. (This processing is performed by the function scan_right, whose pseudocode is given in Figure 26.)

If neither LFS is empty, then there is a topological problem introduced by *ml*. If this problem can be avoided by ignoring some virtual fragments, then those virtual fragments are ignored, new CFSs are computed and the topological scan continues. (This processing is performed by the function fix_by_ignore, whose pseudocode is given in Figure 27.) Otherwise, if this problem can be avoided by discarding virtual fragments, then those virtual fragments are discarded, new CFSs are computed and the topological scan continues. (This processing is performed by the function fix_by_discard, whose pseudocode is given in Figure 28.) If the problem cannot be avoided by ignoring or discarding virtual fragments, then *ml* is *not* part of an ATVML and the scan with *ml* is terminated.

```
STACK
scan_right(ml,ps,s,lfs1,lfs2,starts)
    LIST            ml;
    PARTIAL_SCAN    ps;
    STACK           s;
    SET             lfs1,lfs2;
    INTEGER         starts;
{
    STACK           ans;
    PARTIAL_SCAN    ps';

    ps' ← ps;
    ps'.atvml ← ps'.atvml || ml;

    if (lfs1 = ∅)
        then
            ps'.cfs1 ← ps'.ss1[1];
            ps'.ss1 ← ps'.ss1[2,...];
            if (starts = 1) then ps'.first_set ← FALSE; fi
        else
            ps'.cfs1 ← lfs1;
    fi

    if (lfs2 = ∅)
        then
            ps'.cfs2 ← ps'.ss2[1];
            ps'.ss2 ← ps'.ss2[2,...];
            if (starts = 2) then ps'.first_set ← FALSE; fi
        else
            ps'.cfs2 ← lfs2;
    fi

    ans ← s; stack_push(ans,ps');
    returns(ans);
}
```

Figure 26: Pseudocode for scan_right

```
STACK
fix_by_ignore(ml,ps,s,starts)
     LIST              ml;
     PARTIAL_SCAN      ps;
     STACK             s;
     INTEGER           starts;
{
     STACK             ans;
     PARTIAL_SCAN      ps';

     ans ← s;

     if (ml ≠ [] or ps.atvml ≠ [])
          then
               ps' ← ps;
               ps'.atvml ← ps'.atvml || ml;
               ps'.cfs_starts ← ps'.ss_starts[1];
               ps'.ss_starts ← ps'.ss_starts[2,...];
               ps'.first_set ← FALSE;
               stack_push(ans,ps');
     fi

     return(ans);
}
```

Figure 27: Pseudocode for fix_by_ignore

```
STACK
fix_by_discard(ml,ps,s,lfs1,lfs2)
    LIST            ml;
    PARTIAL_SCAN    ps;
    STACK           s;
    SET             lfs1,lfs2;
{
    INTEGER         i,j,dr1,dr2;
    BOOLEAN         discard1,discard2,discard1+2;
    MATCH_TYPE      mt1,mt2;
    STACK           ans;
    PARTIAL_SCAN    ps';

    discard1 ← FALSE; discard2 ← FALSE; discard1+2 ← FALSE;
    ans ← s; dr1 ← MAX_DISCARDS - |ps.discards1|; dr2 ← MAX_DISCARDS - |ps.discards2|;
    mt1 ← best_match_ahead(lfs1,ps.ss2); mt2 ← best_match_ahead(lfs2,ps.ss1);

    if ((mt1 = MATCHED and mt2 = MATCHED) or (mt1 = RAN_OUT and mt2 = RAN_OUT))
        if (|lfs1| = 1) then discard1 ← TRUE; fi
        if (|lfs2| = 1) then discard2 ← TRUE; fi
    fi

    if ((mt1 = NOT_MATCHED and mt2 = MATCHED) or (mt1 = RAN_OUT and mt2 = MATCHED) or
      (mt1 = NOT_MATCHED and mt2 = RAN_OUT))
        if (|lfs1| = 1) then discard1 ← TRUE; else if (|lfs2| = 1) then discard2 ← TRUE; fi fi
    fi

    if ((mt1 = MATCHED and mt2 = NOT_MATCHED) or (mt1 = MATCHED and mt2 = RAN_OUT) or
      (mt1 = RAN_OUT and mt2 = NOT_MATCHED))
        if (|lfs2| = 1) then discard2 ← TRUE; else if (|lfs1| = 1) then discard1 ← TRUE; fi fi
    fi

    if (|lfs1| = 1 and |lfs2| = 1 and mt1 = NOT_MATCHED and mt2 = NOT_MATCHED)
        then discard1+2 ← TRUE;
    fi

    if (discard1+2 and dr1 > 0 and dr2 > 0)
        then
            ps' ← ps;
            ps'.atvml ← ps'.atvml || ml;
            ps'.discards1 ← ps'.discards1 ∪ lfs1; ps'.discards2 ← ps'.discards2 ∪ lfs2;
            ps'.cfs1 ← ps'.ss1[1]; ps'.ss1 ← ps'.ss1[2,...];
            ps'.cfs2 ← ps'.ss2[1]; ps'.ss2 ← ps'.ss2[2,...];
            ps'.first_set ← FALSE;
            stack_push(ans,ps');
    fi

    for i ∈ {1,2} do
        if (discardi and dri > 0)
            then
                if (i = 1) then j ← 2; else j ← 1; fi
                ps' ← ps;
                ps'.atvml ← ps'.atvml || ml;
                ps'.discardsi ← ps'.discardsi ∪ lfsi;
                ps'.cfsi ← ps'.ssi[1]; ps'.ssi ← ps'.ssi[2,...];
                ps'.cfsj ← lfsj;
                if (starts = i) then ps'.first_set ← FALSE; fi
                stack_push(ans,ps');
        fi
    rof

    return(ans);
}
```

Figure 28: Pseudocode for fix_by_discard

If either of the new CFSs is NULL, then the scan cannot continue. At this point, the ATVML corresponding to these CFSs is examined. If the size of the ATVML is greater than or equal to the minimum overlap requirement for incorporation (see §1.2.2) and the VFSS corresponding to the map that ends before the other is completely scanned, then this ATVML (and other related information) is placed in the set to be returned by topological_scan_fixed_start.

The key data structure in topological_scan_fixed_start is PARTIAL_SCAN, which represents the current state of the topological scan for a particular ATVML. PARTIAL_SCAN is a record-like structure containing eight fields: ss_1, ss_2, cfs_1, cfs_2, atvml, $discards_1$, $discards_2$ and first_set. The fields ss_1 and ss_2 are VFSSs, cfs_1 and cfs_2 are CFSs, atvml is an ATVML, $discards_1$ and $discards_2$ are sets of discarded virtual fragments and first_set is a Boolean that indicates whether any of the virtual fragments in the leftmost set of the original VFSS of the map that starts first are still in the CFS. (This is necessary for determining if virtual fragments can be ignored.) A stack of PARTIAL_SCANs is used to keep track of the different possible ATVMLs which can be constructed.

The running example is now used to present the remaining details of topological scanning, in which the trace of the call find_all_scans(W_1,W_2) will be presented. The set ans is set to ∅ and the function find_all_valid_configurations is called. This call has already been examined in §2.2 and returns $\{cf_1,cf_2,cf_3,cf_4\}$. Then find_all_scans calls topological_scan_fixed_configuration four times. The first call (which uses cf_1) will be examined in detail and the other calls will not.

In the first call, topological_scan_fixed_configuration calls topological_scan_fixed_direction (W_1,W_2,cf_1,LEFT). Then topological_scan_fixed_direction computes the VFSSs of W_1 and W_2 by calling window_find_vfrag_setseq twice. Because cf_1.$orientation_1$ = ASIS, cf_1.$orientation_2$ = ASIS and dir = LEFT, both VFSSs are reversed. As a result, ss_1 = [{728,2708},{1523,1711},{772},{2174},{1997},{521},{2163},{421,1000,1730},{445,852,1649},{1857}] and ss_2 = [{1715},{1633,2163},{848},{446,1861,2992},{10362},{2296}]. Because cf_1.orelationship = OR1, starts = 1 and ends = 2. Then a loop is entered where topological_scan_fixed_start is called repeatedly until it returns a non-empty set or until one of the VFSSs is exhausted.

The first call to topological_scan_fixed_start is now examined. The set ans is set to ∅. Any PARTIAL_SCAN which includes an ATVML of significant size will be placed in ans. A call to find_initial_partial_scan creates a PARTIAL_SCAN ps_0 that represents the initial state of the search for ATVMLs. (The pseudocode for find_initial_partial_scan is given in Figure 29, and the values of the

```
PARTIAL_SCAN
find_initial_partial_scan(ss1,ss2)
    LIST                ss1,ss2;
{
    PARTIAL_SCAN    ans;

    ans.cfs1 ← ss1[1]; ans.ss1 ← ss1[2,...];
    ans.cfs2 ← ss2[1]; ans.ss2 ← ss2[2,...];
    ans.atvml ← [];
    ans.discards1 ← ∅; ans.discards2 ← ∅;
    ans.first_set ← TRUE;

    return(ans);
}
```

Figure 29: Pseudocode for find_initial_partial_scan

various fields of ps_0 are given in Figure 30.) Then ps_0 is pushed onto an empty stack s and the while loop is entered. The object on top of the stack, ps_0 at this point, is popped off of the stack, leaving it empty. ps_0 is now the current PARTIAL_SCAN of interest. Neither CFS of ps_0 is NULL, and thus the set of maximum size local matchlists between the CFSs ({728,2708} and {1715}) is computed. In this case, the empty matchlist is the only local matchlist. Next the LFSs with respect to the empty list are computed, resulting in lfs_1 = {728,2708} and lfs_2 = {1715}. Since neither lfs_1 nor lfs_2 is the empty set, there must be a topological violation. Recall that there are two ways to handle such topological violations: ignore or discard virtual fragments.

Since ps_0.first_set is true, the virtual fragments in lfs_1 are from the starting group of W_1. This implies that it is possible that these virtual fragments can be ignored. Thus, the function fix_by_ignore is called. fix_by_ignore takes as input a local matchlist ml, a PARTIAL_SCAN ps, a stack s of PARTIAL_SCANs and an integer starts (which in this call are [], ps_0, an empty stack, and 1, respectively). fix_by_ignore determines if the members of the LFS of the starting window can be ignored. The basic premise behind fix_by_ignore is to create a new PARTIAL_SCAN object for each possible way of ignoring virtual fragments, push these PARTIAL_SCANs on top of the given stack and return the stack. (fix_by_discard and scan_right use the creation of new PARTIAL_SCANs and the stack in a similar manner.)

In the running example, it is known that the members of the LFS of the starting window are from the starting group of that window, and thus it is possible that these virtual fragments actually lie to the left of virtual fragments in the ATVML that is being constructed. Thus, it would certainly be a valid course of action if these virtual fragments were ignored. However, in order to make the scanning algorithm more efficient, these virtual fragments will not be ignored in this particular case. To examine why, consider the following possibility.

Let g_1 be the starting group of a starting window and let g_2 be the group immediately to the right of g_1. Suppose that all the virtual fragments in g_1 are in that window's LFS. If one then ignores the virtual fragments in that LFS, any ATVMLs that are constructed after that point will actually begin in g_2. Those same ATVMLs will be constructed when g_2 is used as the starting group. This means some computation is unnecessarily duplicated. By not allowing the virtual fragments in an LFS that contains all the virtual fragments of the starting group, such duplication can be avoided. To explicitly check for this could be time consuming, but fortunately the following two, very simple, conditions act as indicators of this situation.

(xx_1) The local matchlist is empty.

(xx_2) The ATVML of the PARTIAL_SCAN is empty.

So if xx_1 and xx_2 both hold, fix_by_ignore does not allow virtual fragments to be ignored and simply returns the stack s. If either xx_1 or xx_2 fails to hold, then fix_by_ignore allows the virtual fragments in the LFS to be ignored. It accomplishes this by creating a new PARTIAL_SCAN where

(1) the ATVML is the ATVML of the original PARTIAL_SCAN with the local matchlist appended and

atvml = [] cfs_1 = {728,2708} cfs_2 = {1715}
ss_1 = [{1523,1711},{772},...] ss_2 = [{1633,2163},{848},...]
$discards_1$ = ∅ $discards_2$ = ∅ first_set = TRUE

Figure 30: The PARTIAL_SCAN ps_0

(2) the leftmost set of virtual fragments in the VFSS of the starting window is extracted and becomes the CFS of that window.

The new PARTIAL_SCAN is pushed onto s, and s is returned.

In our example, conditions xx_1 and xx_2 do hold and thus, fix_by_ignore does not ignore any virtual fragments. It simply returns the stack s, which is empty.

Control returns to topological_scan_fixed_start, which then sets s to be the empty stack. The while loop is exited and topological_scan_fixed_start returns ∅. This means topological_scan_fixed_start could not construct any ATVMLs using {728,2708} as the starting group of W_1.

Now control returns to topological_scan_fixed_direction. The leftmost member of ss_1 is extracted resulting in ss_1 =
[{1523,1711},{772},{2174},{1997},{521},{2163},{421,1000,1730},{445,852,1649},{1857}].
This effectively causes the new starting group to be the next group immediately to the right of the previous starting group. Then topological_scan_fixed_start is called a second time.

A call to find_initial_partial_scan creates a PARTIAL_SCAN ps_1, which represents the initial state of the search for ATVMLs with this new starting group (and is illustrated in Figure 31). ps_1 is pushed onto the stack s and the while loop is entered. ps_1 is immediately popped off of the stack and becomes the current PARTIAL_SCAN. Neither CFS of ps_1 is empty, so the maximum size local matchlists between them ({1523,1711} and {1715}) are computed. One matchlist is found, [(1711,1715)]. Now the LFS of each CFS with respect to this matchlist is computed, resulting in lfs_1 = {1523} and lfs_2 = ∅. The fact that lfs_2 is the empty set means that the local matchlist does not introduce any topological violations. scan_right is called to reset the CFSs so that scanning may continue.

scan_right takes as input a local matchlist ml, a PARTIAL_SCAN ps, a stack s, LFSs lfs_1 and lfs_2 and an integer starts (which in this case are [(1711,1715)], ps_1, an empty stack, {1523}, ∅ and 1, respectively). It creates a new PARTIAL_SCAN from the given one by (1) appending the local matchlist to the ATVML of the given PARTIAL_SCAN and if necessary, (2) extracting the leftmost set of virtual fragments in the VFSS of the starting window and assigning it to be the new CFS of that window. Then this new PARTIAL_SCAN is pushed onto s, and s is returned.

In the running example, the PARTIAL_SCAN ps_2 is created (see Figure 32). Note that the local matchlist ([(1711,1725)]) has been appended to the ATVML of ps_1 ([]) to produce the ATVML of ps_2. Also note that the leftmost set in $ps_2.ss_2$ has been extracted and assigned to $ps_2.cfs_2$. These actions set up for further ATVML construction by bringing out the next group's virtual fragments into the CFS, where it is possible for them to become part of the next local matchlist. ps_2 is pushed onto s and scan_right returns.

atvml = [] cfs_1 = {1523,1711} cfs_2 = {1715}
ss_1 = [{772},{2174},...] ss_2 = [{1633,2163},{848},...]
$discards_1$ = ∅ $discards_2$ = ∅ first_set = TRUE

Figure 31: The PARTIAL_SCAN ps_1

atvml = [(1711,1725)] cfs_1 = {1523} cfs_2 = {1633,2163}
ss_1 = [{772},{2174},...] ss_2 = [{848},{446,1861,2992},...]
$discards_1 = \varnothing$ $discards_2 = \varnothing$ first_set = TRUE

Figure 32: The PARTIAL_SCAN ps_2

Control returns to topological_scan_fixed_start and now s contains only ps_2. The while loop continues and ps_2 is popped off of s to become the current PARTIAL_SCAN. Now the maximum size local matchlists between the CFSs ({1523} and {1633,2163}) are computed. The only one in this case is the empty list. The LFSs are computed, resulting in lfs_1 = {1523} and lfs_2 = {1633,2163}. Since neither LFS is empty, some topological violation has occurred. Since ps_2.first_set is TRUE, fix_by_ignore is called.

In this call to fix_by_ignore, the ATVML of the given PARTIAL_SCAN is non-empty. Thus, condition xx_2 does not hold and the virtual fragments in the LFS of the starting window can be ignored. So a new PARTIAL_SCAN ps_3 is created (see Figure 33) and pushed onto s. No virtual fragments in the starting group of the starting window are still in the CFS of that window because they have just been ignored and the field ps_3.first_set is set to FALSE to reflect this fact. This will prevent any other virtual fragments from being ignored when building upon the ATVML of ps_3.

Control returns to topological_scan_fixed_start and now s contains only ps_3. Again, the while loop continues and ps_3 is popped off of s to become the current PARTIAL_SCAN. Now the maximum size local matchlists between the CFSs ({772} and {1633,2163}) are computed. Once again, the only matchlist is the empty list. Recall that ps_3.first_set is FALSE. Thus, fix_by_ignore cannot be called at this point. Instead, the function fix_by_discard is called.

fix_by_discard takes as input a local matchlist ml, a PARTIAL_SCAN ps, a stack of PARTIAL_SCANs s, and two LFSs lfs_1 and lfs_2. The objective of fix_by_discard is to find reasonable (but not necessarily all) ways of discarding virtual fragments in the current LFSs so that the construction of ATVMLs can continue.

Recall that the reason for the construction of ATVMLs is to determine possible overlap between two windows without resorting to an algorithm where (1) topological constraints are completely ignored or (2) topological constraints are strictly enforced. The objective is to have an approach which is somewhere between these two extremes, while leaning toward the second extreme. In other words, although topological constraints should not be strictly enforced, one does not want to allow too many topological constraints to be violated. In order to fulfill this objective, the following rules are enforced when making decisions about discarding virtual fragments.

atvml = [(1711,1725)] cfs_1 = {772} cfs_2 = {1633,2163}
ss_1 = [{2174},{1997},...] ss_2 = [{848},{446,1861,2992},...]
$discards_1 = \varnothing$ $discards_2 = \varnothing$ first_set = FALSE

Figure 33: The PARTIAL_SCAN ps_3

(dc_1) No more than one virtual fragment in a particular group may be discarded.

(dc_2) No more than a predetermined number (MAX_DISCARDS) of virtual fragments in a particular window can be discarded.

In this report, it is assumed that the value of MAX_DISCARDS is two. This value seems to work well for the kinds of examples to which FIX has been applied. It is certainly possible that a different value may work better for examples with different characteristics. However, the particular value one chooses has an effect upon the analysis to be presented in §3 and §4. A slightly different analysis would be required for values of MAX_DISCARDS other than two. This will become clear in §3 and §4.

Also keep in mind that dc_1 and dc_2 are only heuristics. More sophisticated rules similar to dc_1 and dc_2 might improve the performance of FIX. Again, changes to these rules might require a different analysis in §3 and §4. The appeal of dc_1 and dc_2 is that they are relatively simple and seem to work well for the kinds of examples to which FIX has been applied.

It may be reasonable to avoid topological violations by (1) discarding a virtual fragment from W_1, (2) discarding a virtual fragment from W_2 or (3) discarding a virtual fragment from both W_1 and W_2. The primary question is: How should fix_by_discard decide which of these possibilities are reasonable in a particular situation? The key to answering this question is in the function best_match_ahead (see Figure 34). best_match_ahead answers the question: Does a topologically valid matchlist exists between the virtual fragments in the LFS associated with a window and virtual fragments in the other window, if the virtual fragments in the LFS associated with the second window are discarded? To see why this question is important, consider the situation illustrated in Figure 35.

The solid lines between the windows Z_1 and Z_2 denote the current ATVML. Suppose that at this point, the LFS associated with the Z_1 contains virtual fragments from the group g_1 and the LFS associated with Z_2 contains only a virtual fragment from the group g_2. Suppose that one finds that a topologically valid matchlist exists between the virtual fragments in the LFS of Z_1 and the virtual fragments in the groups immediately to the right of g_2. (This matchlist is represented by the dashed lines.) This situation indicates that if the virtual fragment in the LFS of Z_2 is discarded, the chances of constructing an ATVML of significant size are greatly increased. That is, because a matchlist is found by "looking ahead", discarding a particular virtual fragment is determined to be a reasonable course of action. The function best_match_ahead performs this "lookahead" and returns whether such matchlists are found.

best_match_ahead takes as input a set of virtual fragments s and a VFSS ss. It returns one of the following three codes: MATCHED, NOT_MATCHED or RAN_OUT. best_match_ahead returns MATCHED if a topologically valid matchlist exists between the virtual fragments in s and ss such that (1) it starts at the left end of ss and (2) it uses all members of s. best_match_ahead returns RAN_OUT if a topologically valid matchlist exists between the virtual fragments in s and ss such that at least one member of s is used, but no matchlist exists which uses all members of s. Otherwise, best_match_ahead returns NOT_MATCHED. (In the example illustrated in Figure 35, best_match_ahead returns MATCHED.)

Now the discussion returns to the operation of fix_by_discard. fix_by_discard calls best_match_ahead twice. The first time it uses lfs_1 and ps. ss_2. (I.e., is there a matchlist between the LFS of W_1 and the virtual fragments beyond the current group of W_2?) The code returned is then assigned to mt_1. The second time it uses lfs_2 and ps. ss_1. The code returned is then assigned to mt_2. In the running example, the first call to best_match_ahead (where s = {772} and ss = [{848},{446,1861,2992},{10362},{2296}]) returns NOT_MATCHED. The second call (where s = {1633,2163} and ss = [{2174},{1997},{521},{2163},{421,1000,1730},{445,852,1649},{1857}] also returns NOT_MATCHED.

```
MATCH_TYPE
best_match_ahead(s,ss)
      SET               s;
      LIST              ss;
{
      MATCH_TYPE        ans;
      SET               s',mls;
      LIST              ss',ml;

      if (s = ∅)
            then ans ← MATCHED;
            else
                  if (ss = [])
                        then ans ← RAN_OUT;
                        else
                              ans ← NOT_MATCHED;
                              mls ← find_best_matchlists(s,ss[1]);

                              for ml ∈ mls do
                                    s' ← s — left_elements_of(ml);
                                    if (s' = ∅)
                                          then ans ← MATCHED;
                                          else
                                                if (|ml| = |ss[1]|)
                                                      then
                                                            ss' ← ss[2,...];
                                                            if (best_match_ahead(s',ss') = MATCHED)
                                                                  then ans ← MATCHED;
                                                            fi
                                                fi
                                    fi
                              rof
                  fi
      fi

      return(ans);
}
```

Figure 34: Pseudocode for best_match_ahead

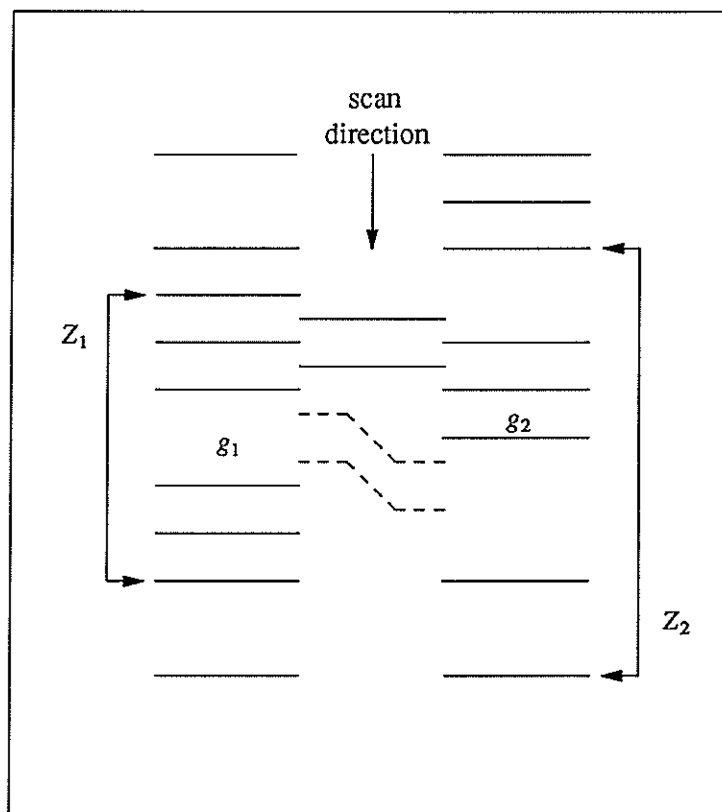


Figure 35: Looking ahead for a matchlist

Now, fix_by_discard must decide which (if any) virtual fragments can be discarded based on rules dc_1 and dc_2 and the results of the calls to best_match_ahead. The results from the running example indicate that (1) discarding just the virtual fragment in the LFS of W_1 does not greatly increase the chances of building a large ATVML and (2) discarding just the virtual fragment in the LFS of W_2 does not greatly increase the chances of building a large ATVML. (Note that since the LFS of W_2 contains more than one virtual fragment, they cannot be discarded without violating dc_1 anyway.) Thus, the only way to increase the chances of building a large ATVML is to discard the virtual fragments in both LFSs. However, since the LFS of W_2 contains more than one member, they cannot be discarded without violating rule dc_1. Thus, fix_by_discard has no reasonable way to discard virtual fragments at this point, so it creates no new PARTIAL_SCANs and simply returns the stack s.

Control returns to topological_scan_fixed_start and s becomes the empty stack. Thus, the while loop is exited and topological_scan_fixed_start returns $\emptyset$, indicating that no ATVMLs could be constructed using $\{1523,1711\}$ as the starting group of W_1.

The next five calls to topological_scan_fixed_start (which use $\{772\}$, $\{2174\}$, $\{1997\}$, $\{521\}$ and $\{2163\}$ as the starting group of W_1, respectively) all return $\emptyset$. In fact, each of these calls is unable to build an ATVML of size one, much less one of a significant size.

The next call to topological_scan_fixed_start is different; it uses $\{421,1000,1730\}$ as the starting group of W_1. It first calls find_initial_partial_scan, which creates the PARTIAL_SCAN ps_4 (see Figure 36). ps_4 is pushed onto s, the while loop is entered and ps_4 is popped off of s to become the current PARTIAL_SCAN. The maximum size local matchlist between the CFSs ($\{421,1000,1730\}$ and $\{1715\}$)

atvml = [] cfs_1 = {421,1000,1730} cfs_2 = {1715}
ss_1 = [{445,852,1649},{1857}] ss_2 = [{1633,2163},{848},...]
$discards_1 = \varnothing$ $discards_2 = \varnothing$ first_set = TRUE

Figure 36: The PARTIAL_SCAN ps_4

are computed. One matchlist is found, [(1730,1715)]. The LFSs are computed, resulting in lfs_1 = {421,1000} and $lfs_2 = \varnothing$. Since lfs_2 is empty, no topological violations exist and scan_right is called. scan_right creates a new PARTIAL_SCAN ps_5 (see Figure 37) and returns a stack containing only ps_5.

Control returns to topological_scan_fixed_start and ps_5 is popped off of s to become the current PARTIAL_SCAN. The maximum size local matchlists between the CFSs ({421,1000} and {1633,2163}) are computed. The only such matchlist is the empty list. The LFSs are computed again, resulting in lfs_1 = {421,1000} and lfs_2 = {1633,2163}. Since neither LFS is empty, some topological violation exists. Since ps_5.first_set is TRUE, fix_by_ignore is called.

fix_by_ignore successfully ignores the virtual fragments in the LFS of W_1, creating the PARTIAL_SCAN ps_6 (see Figure 38) and returning a stack containing only ps_6.

Control returns to topological_scan_fixed_start and ps_6 is popped off of s to become the current PARTIAL_SCAN. The maximum size local matchlists between the CFSs ({445,852,1649} and {1633,2163}) are computed. One matchlist is found, [(1649,1633)]. The LFSs are computed again, resulting in lfs_1 = {445,852} and lfs_2 = {2163}. Since neither LFS is empty, some topological violation exists. Since ps_6.first_set is FALSE, fix_by_discard is called.

As in the previous call to fix_by_discard, the function best_match_ahead is called twice. In the first call, best_match_ahead successfully finds a topologically valid matchlist between lfs_1 and $ps_6.ss_2$ (namely [(445,446),(852,848)]) and thus it returns MATCHED. In the second call, best_match_ahead fails to find any matchlists between lfs_2 and $ps_6.ss_1$ and thus it returns NOT_MATCHED. Again, fix_by_discard must decide what is a reasonable course of action given these results while still following rules dc_1 and dc_2. In this case, the chances of constructing a significant ATVML are increased if the member of lfs_2 (i.e., 2163) is discarded (because the first call to best_match_ahead returned MATCHED). Since $|lfs_2| = 1$ and there have been no previous discards, rules dc_1 and dc_2 are not violated.

atvml = [(1730,1715)] cfs_1 = {421,1000} cfs_2 = {1633,2163}
ss_1 = [{445,852,1649},{1857}] ss_2 = [{848},{446,1861,2992},...]
$discards_1 = \varnothing$ $discards_2 = \varnothing$ first_set = TRUE

Figure 37: The PARTIAL_SCAN ps_5

atvml = [(1730,1715)] cfs_1 = {445,852,1649} cfs_2 = {1633,2163}
ss_1 = [{1857}] ss_2 = [{848},{446,1861,2992},...]
$discards_1 = \varnothing$ $discards_2 = \varnothing$ first_set = FALSE

Figure 38: The PARTIAL_SCAN ps_6

The chances of constructing a significant ATVML are not increased if the members of the lfs_1 are discarded (because the second call to best_match_ahead returned NOT_MATCHED). Thus, fix_by_discard creates one new PARTIAL_SCAN ps_7 (see Figure 39) and returns a stack containing only ps_7. (Note that the set ps_7.discards$_2$ contains 2163, the discarded virtual fragment.)

Control returns to topological_scan_fixed_start and ps_7 is popped off of s to become the current PARTIAL_SCAN. The maximum size local matchlists between the CFSs ({445,852} and {848}) are computed. One matchlist is found, [(852,848)]. The LFSs are computed again, resulting in lfs_1 = {445} and $lfs_2 = \varnothing$. Since lfs_2 is empty, no topological violations have been introduced and scan_right is called. scan_right creates a new PARTIAL_SCAN ps_8 (see Figure 40) and returns a stack containing only ps_8.

The next two passes through the while loop in topological_scan_fixed_start are similar, resulting in the creation of PARTIAL_SCANs ps_9 and ps_{10} (see Figures 41 and 42, respectively). So at this point ps_{10} is the current PARTIAL_SCAN. However, unlike all previous cases, one of the CFSs is NULL (i.e., there are no more virtual fragments left in one of the VFSSs). Thus, the construction of ATVMLs cannot continue. The question at this point is: Does ps_{10} represent an ATVML of interest? It does if the following conditions hold.

(ia_1) The size of the ATVML is greater than or equal to the minimum size fragment overlap required for window incorporation (MIN_OVERLAP).

atvml = [(1730,1715),(1649,1633)] cfs_1 = {445,852} cfs_2 = {848}
ss_1 = [{1857}] ss_2 = [{446,1861,2992},{10362},{2296}]
$discards_1 = \varnothing$ $discards_2$ = {2163} first_set = FALSE

Figure 39: The PARTIAL_SCAN ps_7

atvml = [(1730,1715),(1649,1633),(852,848)] cfs_1 = {445} cfs_2 = {446,1861,2992}
ss_1 = [{1857}] ss_2 = [{10362},{2296}]
$discards_1 = \varnothing$ $discards_2$ = {2163} first_set = FALSE

Figure 40: The PARTIAL_SCAN ps_8

atvml = [(1730,1715),(1649,1633),(852,848),(445,446)] cfs_1 = {1857} cfs_2 = {1861,2992}
ss_1 = [] ss_2 = [{10362},{2296}]
$discards_1 = \varnothing$ $discards_2$ = {2163} first_set = FALSE

Figure 41: The PARTIAL_SCAN ps_9

atvml = [(1730,1715),(1649,1633),(852,848),(445,446),(1857,1861)] cfs_1 = NULL cfs_2 = {2992}
ss_1 = [] ss_2 = [{10362},{2296}]
$discards_1 = \varnothing$ $discards_2$ = {2163} first_set = FALSE

Figure 42: The PARTIAL_SCAN ps_{10}

(ia_2) The VFSS associated with the window that is supposed to end later (indicated by ends) is empty.

Condition ia_1 ensures that the ATVML is of significant size. Condition ia_2 ensures that the configuration of the windows determined by the ATVML agrees with the configuration that was expected.

Assuming the value of MIN_OVERLAP is four (a common value), ps_{10} satisfies conditions ia_1 and ia_2. Thus, ps_{10} is placed in ans, resulting in ans = $\{ps_{10}\}$. The stack s is now empty, and thus the while loop is exited and topological_scan_fixed_start returns $\{ps_{10}\}$.

The final two calls to topological_scan_fixed_start for the configuration cf_1 use {445,852,1649} and {1857} as the starting group of W_1, respectively. Both calls return ∅, primarily because no virtual fragments are available in W_2 to match with 1715. Thus, the call to topological_scan_fixed_direction with $w_1 = W_1$, $w_2 = W_2$, c = cf_1 and dir = LEFT returns $\{ps_{10}\}$.

So the left scan of W_1 and W_2 for the configuration cf_1 is complete. Next, the right scan of W_1 and W_2 for the configuration cf_1 occurs. None of the details of this scan will be presented. It is relatively easy to see that the same ATVML found in the left scan is found in the right scan, except that it is constructed in the opposite order. The call to topological_scan_fixed_direction with $w_1 = W_1$, $w_2 = W_2$, c = cf_1 and dir = RIGHT returns $\{ps_{11}\}$ (see Figure 43). Thus, the function topological_scan_fixed_configuration returns $\{ps_{10},ps_{11}\}$.

Control finally returns to find_all_scans and pss is set to $\{ps_{10},ps_{11}\}$. Each PARTIAL_SCAN in pss is used to create a SCAN type object. Some of the information in the PARTIAL_SCAN is no longer necessary and the information that is necessary is placed in the SCAN that is created. The SCAN s_1 (see Figure 44) is created from ps_{10}. The SCAN s_2 (see Figure 45) is created from ps_{11}. Both s_1 and s_2 are placed in the set ans.

The next three calls to topological_scan_fixed_configuration (which use cf_2, cf_3 and cf_4) all return ∅. Thus, find_all_scans returns $\{s_1,s_2\}$.

atvml = [(1857,1861),(445,446),(852,848),(1649,1633),(1730,1715)]
cfs_1 = {421,1000} cfs_2 = NULL
ss_1 = [{2163},{521},...] ss_2 = []
$discards_1$ = ∅ $discards_2$ = {2163} first_set = FALSE

Figure 43: The PARTIAL_SCAN ps_{11}

atvml = [(1730,1715),(1649,1633),(852,848),(445,446),(1857,1861)]
$discards_1$ = ∅ $discards_2$ = {2163} c = cf_1

Figure 44: The SCAN s_1

atvml = [(1857,1861),(445,446),(852,848),(1649,1633),(1730,1715)]
$discards_1$ = ∅ $discards_2$ = {2163} c = cf_1

Figure 45: The SCAN s_2

Thus, the result of the topological scanning of W_1 and W_2 results in two ATVMLs, one being the reverse of the other. In both cases, one virtual fragment had to be discarded and the topological scan has pinpointed a definite problem preventing the incorporation of W_1 and W_2. The next sections show how this information can be used to help repair a fragment matching mistake.

3. Detecting Fragments to Split

In this section, another component of FIX is examined in detail. This component uses (a) information that can be obtained from the maps (primarily from topological scanning) and (b) information about how a fragment matching mistake affects a map in order to make educated guesses about which virtual fragments may be part of a fragment matching mistake. In particular, it is concerned with fragment matching mistakes that require a split to repair. This component is the result of a detailed case analysis which links possible observable results from topological scanning with possible underlying realities containing a fragment matching mistake.

In §3.1, an overview of this component is presented. In §3.2, the method for enumerating possible underlying realities is discussed. One specific underlying reality is analyzed in-depth in §3.3. The results of the analysis of each underlying reality is presented in §3.4. Finally in §3.5, the method for constructing a table which identifies virtual fragments that are likely to contain a fragment matching mistake is presented.

3.1. Overview

3.1.1. Heuristic Basis for Detecting Fragments to Split

The ultimate goal of this component is to be able to determine which virtual fragments are likely to contain a fragment matching mistake. This component is based upon the following proposition.

Proposition P: If a correct map and a map containing a particular fragment matching mistake exhibit a set of properties X, then two different maps which exhibit the properties in X contain that particular fragment matching mistake.

Obviously, P is going to be incorrect some of the time. To make matters worse, it is possible that two sets of maps involving similar, but not identical, fragment matching mistakes may exhibit the same set of properties. Thus, given two maps that exhibit a particular set of properties, there may be more than one fragment matching mistake which is consistent with the observable results. It cannot be known which mistake is the one that actually occurred.

However, there are some things that can be done to increase the probability that P is correct. The probability that P is correct increases as the set of properties of interest becomes larger and if the properties used are particularly sensitive to fragment matching mistakes.

For this report, there are two properties which are used in conjunction with P. The first property is the results obtained from topological scanning. The number of class 3 fragments is especially important. The second property is of interest only if one of the maps assimilates into the other. This property is the size of a "regular" (i.e., non-topological) maximum size matchlist in the minimum size window in the map which is assimilating. (This will become clearer later in the report.)

3.1.2. Limited Enumeration of Possible Underlying Realities

If it were possible to completely enumerate and describe each underlying reality (i.e., the sites in a genome and the set of clones used to construct two maps) that leads to a fragment matching mistake, one could look at each reality and determine what the resulting two computed maps would look like. Then given two maps, m_1 and m_2, where one of the maps contains a fragment matching mistake, one could find the underlying reality (or realities) which result in m_1 and m_2 and the correct versions of m_1 and m_2 could be determined.

Unfortunately, it is intractable to perform such a complete enumeration. So instead of trying to completely enumerate and describe the possible underlying realities, a limited enumeration is performed. This enumeration is limited in two ways.

The first is to discard some underlying realities that can be deemed to be of no interest in solving the current problem. The second is not to be concerned with all properties of the underlying reality, but only with a few selected properties deemed important to the current problem. With this approach, a large set of underlying realities can be collapsed into a single case when they exhibit the same values for the selected set of properties.

Since this report is concerned with detecting a single fragment matching mistake, any underlying reality which results in no fragment matching mistake or more than one fragment matching mistake will be classified as of no interest. This eliminates many underlying realities from having to be considered. The limited enumeration is based upon the assumption that the underlying reality is such that a single fragment matching mistake has occurred. This assumption must be made more specific to be of use in this analysis. Figure 46 illustrates the situation which is assumed to exist. The following conditions are assumed to hold:

(1) There are two genomic fragments gf_1 and gf_2 of similar length about one clone's length apart.

(2) There is a clone c_1 containing a real fragment rf_1 that corresponds to gf_1.

(3) There is a clone c_2 containing a real fragment rf_2 that corresponds to gf_2.

(4) There is a map m_1, containing c_1 and c_2, which is correct except that rf_1 and rf_2 are incorrectly matched to form a virtual fragment vf_0.

(5) There is a map m_2 which is correct and may contain virtual fragments vf_1 and vf_2, which correspond to rf_1 and rf_2, respectively.

Given these assumptions, m_1 and m_2 are likely to have significant overlap. However, they will not incorporate because vf_1 and vf_2 cannot both match with vf_0. If topological scanning were applied to m_1 and m_2, it is likely an ATVML of 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.

The details important to the enumeration are broken into several levels. The first level is concerned with the position of m_2, w_1 and w_2 with respect to vf_0, vf_1, vf_2, gf_1 and gf_2. More specifically, the following conditions are important:

(jj_1) m_2 covers the stretch of genome containing gf_1.

(jj_2) m_2 covers the stretch of genome containing gf_2.

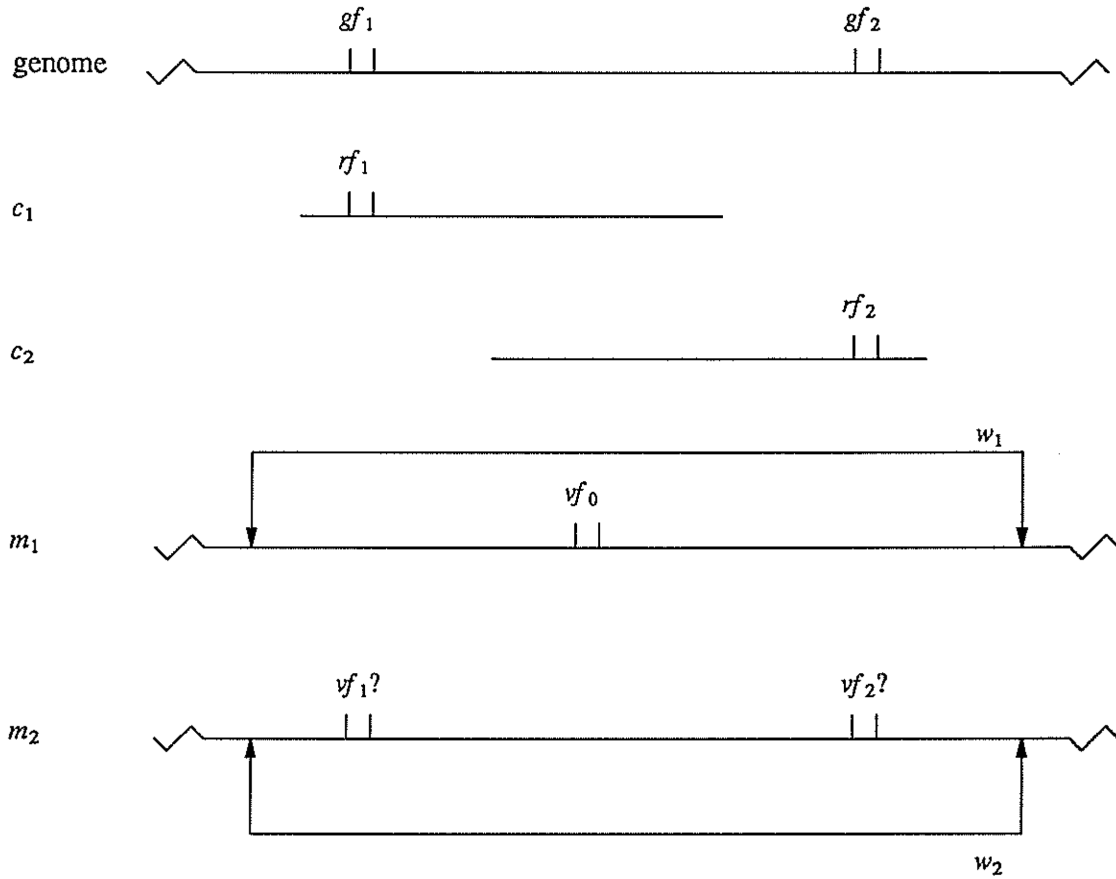


Figure 46: Illustration of the assumptions for the limited enumeration

(jj_3) w_2 contains vf_1.

(jj_4) w_2 contains vf_2.

(jj_5) w_1 contains vf_0.

Each condition can be true or false. Thus there are 32 possible combinations of the values of jj_1 through jj_5. Luckily, most of the combinations are either impossible, very unlikely, of no interest or are symmetric to other cases. The impossible, unlikely and uninteresting combinations are completely ignored and only one of each symmetric pair is considered. (The determination of which cases are of interest is examined in §3.2.)

In the second level of enumeration, each first-level case that is still of interest is broken up into subcases. There is only one detail of importance in the second level of enumeration: the true overlap relationship of m_1 and m_2. Sometimes, not all overlap relationships are possible for a specific subcase of the first level. (Again, the determination of which overlap relationships are possible for a given first-level subcase is presented in §3.2.)

In the third level of enumeration, each combination of a first and second-level subcase that is still of interest is broken down even further. The important detail for the third level of enumeration is the number of **pushed** virtual fragments of length similar to vf_0 in w_1. What exactly pushed virtual fragments are and

why they are important is examined in §3.2.

These three levels of case breakdown create a number of final cases. A specific example of one such case is illustrated in Figure 47. In this illustration, jj_1, jj_3 and jj_5 are true, jj_2 and jj_4 are false, m_1 extends to the right of m_2 and there is one pushed virtual fragment vf' in w_1 of length similar to vf_0.

3.1.3. Determining the Important Properties of a Possible Underlying Reality

Each final case (like the one presented at the end of the previous section) is specific enough that one can ask detailed questions about the possible observable results of a topological scan. The approach taken here is to initially focus on the class of certain virtual fragments of interest in m_1. Given that these fragments are of a particular class, often the class of the virtual fragments of interest in m_2 is uniquely determined, or at the very least restricted somewhat.

Consider the case illustrated in Figure 47. The virtual fragments of interest are vf_0 and vf' in m_1 and vf_1 in m_2. Since vf_0 and vf' are in w_1, neither can be class 1. (Recall that class 1 fragments are, by definition, not in the window.) However, vf_0 and vf' might be class 2, 3 or 4. So it is assumed that vf_0 is class j and vf' is class k (where $j, k \in \{2,3,4\}$); for each of these possibilities, the analysis determines the class of vf_1. Table 2 illustrates the result of that analysis.

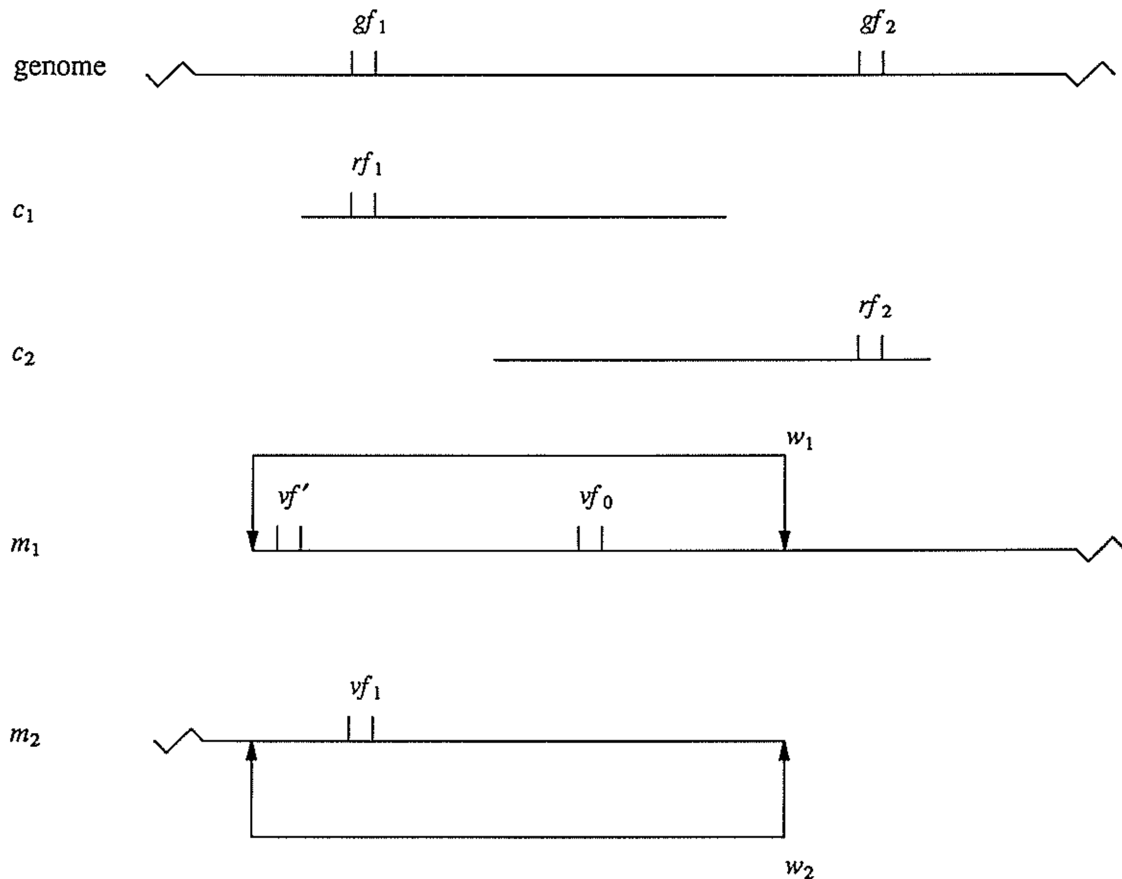


Figure 47: A specific final case

$class(vf_0) \rightarrow$ $class(vf') \downarrow$	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 2
Possible results of a topological scan

In Table 2, a row represents a fixed class for vf' and a column represents a fixed class for vf_0. A cell in Table 2 shows the class of each virtual fragment of interest. A cell containing "IMP" indicates that the particular combination of classes for vf_0 and vf' represented by that cell is impossible, and hence is of no interest in this analysis. A cell containing "INC" indicates that the particular combination of classes for vf_0 and vf' represented by that cell results in incorporation of m_1 and m_2. However, this entire analysis is based upon the assumption that m_1 and m_2 do not incorporate, and hence such a case is of no interest in this analysis.

Exactly how this table is derived in explained in more detail in §3.3. What is important to note at this point is that the table links an underlying reality (i.e., the particular final case) and observable properties of the two maps (i.e., the results of a topological scan); these observable properties correspond to identifying which fragments fall into the different classes. This link is the key is being able to determine the location of virtual fragments containing fragment matching mistakes.

For instance, suppose that two non-incorporating maps x_1 and x_2 existed where

(1) it appears that x_1 extends to the right of x_2 and

(2) a topological scan of x_1 and x_2 results in one class 3 fragment in x_1 and class 2 fragments in x_1 and x_2 that are of similar length to the class 3 fragment.

These results match those of entry (1,2) in Table 2. (Entry (r,c) of a table is the cell in row r and column c. Note this refers to the row and column position and not the class the row or column represents.) It may be the case that the underlying reality that resulted in x_1 and x_2 is that illustrated in Figure 47. If that is truly the case, one could deduce that the class 3 virtual fragment in x_1 is a result of a fragment matching mistake. Thus, it should be possible to incorporate x_1 and x_2 if that fragment is split.

3.1.4. Constructing the Split Table

If each final case is analyzed in the manner described in the previous section, the data from each table created can be organized into a larger table which indicates which virtual fragments to split given the results of topological scanning. This table is called the **split table**. The format and construction of the split table is presented in §3.5. The split table is the end result of the analysis of this section and is used by FIX.

3.2. Limited Enumeration of Possible Underlying Realities

In this section, the limited enumeration of the possible underlying realities is discussed in more detail than in the previous section.

The enumeration begins by assuming the existence of a fragment matching mistake in a map m_1 due to the incorrect matching of two nearby genomic fragments gf_1 and gf_2 of similar length. Then the existence of a mistake free map m_2 is assumed. (This situation is illustrated in Figure 46.) Note that no generality is lost upon assuming that m_1 contains the fragment matching mistake and not m_2, since the maps could easily be renamed.

3.2.1. The First Level of the Enumeration

The enumeration of the possible underlying realities is done in several levels. In the first level, the conditions jj_1 through jj_5 are used to create 32 possible subcases (since each condition may be true or false). However, only four of these cases really need to be considered for further analysis. The other 28 cases are either impossible, highly unlikely, of no interest to the current situation or are symmetric to other cases. No further analysis is performed on those cases. Table 3 contains each of the 32 cases and a remark for each indicating whether it is of interest or not. In order to clarify why certain cases are of no interest, several are examined at this point.

Consider case S32 from Table 3. Since jj_1 and jj_2 are both false, m_2 does not overlap m_1 anywhere near the location of the fragment matching mistake. Thus, it is highly unlikely that m_2 is going to provide any information about the location of the fragment matching mistake. Thus, case S32 is simply of no interest in this analysis and is ignored.

The following theorems are used to eliminate a number of cases.

Theorem 5: $jj_3 \rightarrow jj_1$

Proof: Assume jj_3 is true. Then w_2 contains vf_1. Since vf_1 is the virtual fragment corresponding to gf_1, m_2 must cover the stretch of genome containing gf_1. Therefore, jj_1 is true. □

Theorem 6: $jj_4 \rightarrow jj_2$

Proof: Similar to the proof of Theorem 5. □

Theorems 5 implies that any case where jj_3 is true and jj_1 is false is impossible. This eliminates cases S17-S20 and S25-S28. Theorem 6 implies that any case where jj_4 is true and jj_2 is false is impossible. This eliminates cases S9-S10, S13-S14, S25-S26 and S29-S30.

Some of the cases are deemed unlikely enough that they are not analyzed further. For instance, assume that jj_1 is true, i.e., m_2 covers the stretch of genome containing gf_1. Thus, m_2 contains vf_1. However, m_1 also covers the stretch of genome containing gf_1. Thus, it is highly likely that vf_1 will be in w_2, because m_1 and m_2 overlap near gf_1 and windows always cover at least the portion of the maps which overlap. Thus, it is highly likely that jj_3 is true. So any case where jj_1 is true and jj_3 is false is deemed unlikely enough to be ignored. This eliminates cases S5-S8 and S13-S16.

A similar argument exists for jj_2 and jj_4. This eliminates cases S3-S4, S7-S8, S19-S20 and S23-S24. Another similar argument exists to show that if jj_1 and jj_2 are true, then it is highly likely that jj_5 is true. This eliminates cases S2, S4, S6 and S8.

Case	jj_1	jj_2	jj_3	jj_4	jj_5	Remark
S1	T	T	T	T	T	OK
S2	T	T	T	T	F	Unlikely
S3	T	T	T	F	T	Unlikely
S4	T	T	T	F	F	Unlikely
S5	T	T	F	T	T	Unlikely
S6	T	T	F	T	F	Unlikely
S7	T	T	F	F	T	Unlikely
S8	T	T	F	F	F	Unlikely
S9	T	F	T	T	T	Impossible
S10	T	F	T	T	F	Impossible
S11	T	F	T	F	T	OK
S12	T	F	T	F	F	OK
S13	T	F	F	T	T	Impossible
S14	T	F	F	T	F	Impossible
S15	T	F	F	F	T	Unlikely
S16	T	F	F	F	F	Unlikely
S17	F	T	T	T	T	Impossible
S18	F	T	T	T	F	Impossible
S19	F	T	T	F	T	Impossible
S20	F	T	T	F	F	Impossible
S21	F	T	F	T	T	Symmetric to S11
S22	F	T	F	T	F	Symmetric to S12
S23	F	T	F	F	T	Unlikely
S24	F	T	F	F	F	Unlikely
S25	F	F	T	T	T	Impossible
S26	F	F	T	T	F	Impossible
S27	F	F	T	F	T	Impossible
S28	F	F	T	F	F	Impossible
S29	F	F	F	T	T	Impossible
S30	F	F	F	T	F	Impossible
S31	F	F	F	F	T	OK
S32	F	F	F	F	F	Of No Interest

Table 3
The first-level cases for Split

Some of the cases are symmetric to other cases, at least for the purposes of this analysis. Consider cases S11 and S21. The analysis of case S21 will be the same as the analysis of case S11, except that in case S21, vf_2 appears wherever vf_1 appears in case S11. This is true because vf_1 and vf_2 are of similar length and are in similar positions with respect to the other fragments of interest in the analysis (most importantly vf_0). Thus, case S21 is not analyzed further. A similar situation exists for cases S12 and S22. Thus, case S22 is not analyzed further.

So at the end of the first level of the enumeration, there are four major cases (S1, S11, S12 and S31) which are to be analyzed by a second level of enumeration. Details about this second level are presented in

the next section.

3.2.2. The Second Level of the Enumeration

The second level of the enumeration involves breaking down the remaining first-level cases based upon the true overlap relationship of the maps m_1 and m_2. Since there are four possible overlap relationships between any two maps, each of the first-level cases is broken into four subcases. However, just like many first-level cases, some of these second-level cases are either impossible or are symmetric to other cases. Table 4 summarizes the status of the second-level cases.

A cell in Table 4 containing "OK" indicates that the particular combination of first-level case and overlap relationship of m_1 and m_2 represented by that cell is valid and will undergo further analysis. A cell in Table 4 containing "IMP" represents an impossible combination of first-level case and overlap relationship. These subcases will not be analyzed further. A cell in Table 4 containing "SYM" represents a combination that is symmetric to some other combination. These subcases will not be analyzed further.

It turns out that only one subcase is ignored due to symmetry. The case represented by entry (1,2) is symmetric to the case represented by entry (1,1). Note that is does not matter which map extends to the right because one could "flip" both maps of entry (1,1) and obtain the maps of entry (1,2) and vice-versa. Thus, it is arbitrarily decided to use entry (1,1) and ignore entry (1,2).

There are several subcases which are ignored because they are impossible. For instance, consider the subcases of the first-level case S11. In case S11, jj_1 is false and thus m_2 does not cover the stretch of genome containing gf_2. However, m_1 does cover the stretch of genome containing gf_2. Thus, it is impossible for m_1 to assimilate into m_2. For the same reason, m_2 cannot extend to the right of m_1. This eliminates the subcases represented by entries (2,2) and (2,3) of Table 4. Similar analysis are used to construct the remainder of Table 4.

So at the end of the second level of the enumeration, there are eight subcases which are to be analyzed by a third level of enumeration. Details about this third level are presented in the next section.

3.2.3. The Third Level of the Enumeration

The third level of the enumeration involves breaking down the remaining second-level cases based upon the number of pushed virtual fragments in w_1 that are of length similar to vf_0.

Let x_0 be a virtual fragment containing a fragment matching mistake. Often, there are virtual fragments near x_0 that are of similar length. Often these virtual fragments are composed of real fragments

Overlap Relationship → Case ↓	or_1	or_2	or_3	or_4
S1	OK	SYM	OK	OK
S11	OK	IMP	IMP	OK
S12	OK	IMP	IMP	OK
S31	IMP	IMP	IMP	OK

Table 4
The second-level cases for Split

that correspond to one of the genomic fragments that is involved in the matching mistake. Sometimes, a virtual fragment of this variety exists on both sides of x_0. These virtual fragments tend to get "pushed" away from their true position. This is illustrated in Figure 48. The fragments gf_1, gf_2, rf_1 and rf_2 have the same properties as in previous illustrations. What is important to note here is that there could be other clones besides c_1 and c_2 which contain gf_1 and produce real fragments which correctly match to form the virtual fragment x_1. However, because x_0 contains one of the real fragments that truly belongs to x_1, x_1 is pushed away from x_0, causing x_1 to appear a little to the left of where it should be. A similar phenomena can occur with x_0 and x_2. Virtual fragments like x_1 and x_2 are called pushed fragments.

Because pushed fragments are slightly out of position, they may not match with virtual fragments in other maps, as one would expect. Thus, pushed fragments may stand out in topological scans and help pinpoint the location of the fragment matching mistake.

The third level of the enumeration is based upon the number of pushed fragments within w_1. Each second-level case (except one) is broken into a subcase where there are no pushed fragments and another subcase where there is exactly one pushed fragment. The enumeration would be more complete if subcases for more than one pushed fragment were considered, but using just two subcases accounts for most realistic situations and keeps the analysis from becoming more complex. Notice that in case S31, there is no final case for the existence of one pushed fragment. This final case was deemed unlikely because gf_1 and gf_2 lie outside of w_1.

The tree illustrated in Figure 49 summarizes the limited enumeration. Each final case is given a label for future reference.

This section has presented considerable detail on the manner in which possible underlying realities are enumerated. The next section presents a detailed look at how one particular final case of the enumeration is analyzed, so that it may be used to construct the split table.

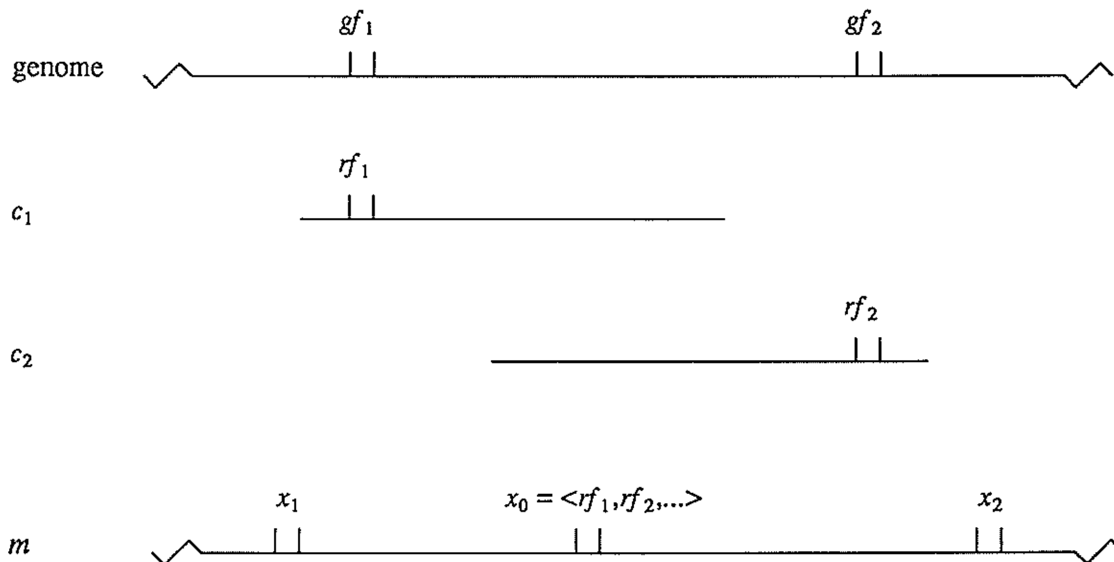


Figure 48: Pushed virtual fragments

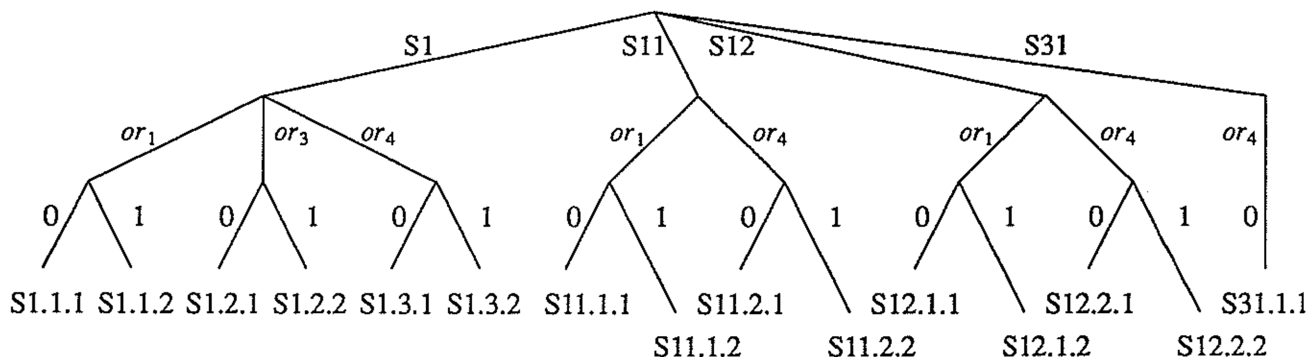


Figure 49: Summary of the limited enumeration

3.3. Detailed Analysis of a Particular Case

Each of the final cases in the limited enumeration represents one possible underlying reality. The next step in the analysis is to examine each final case and determine the important observable properties of this underlying reality. In this section, the determination of the important visible properties of final case S11.2.2 is presented. (This is the subcase corresponding to Table 2.) Case S11.2.2 is illustrated in Figure 50. Since jj_1 is true, vf_0 is in w_1. Since jj_3 is true, vf_1 is in w_2. In addition, m_2 almost assimilates into m_1 and exactly one pushed fragment vf' exists in w_1.

There are two properties of interest. The first is the result of a topological scan of m_1 and m_2. The second is the size of a non-topological matchlist with respect to the number of virtual fragments in the minimum size window (of an ATVML between m_1 and m_2) of the assimilating map. (The second property is ignored if the overlap relationship of the two maps does not involve an assimilation.) Final case S11.2.2 is now examined with respect to the these two properties.

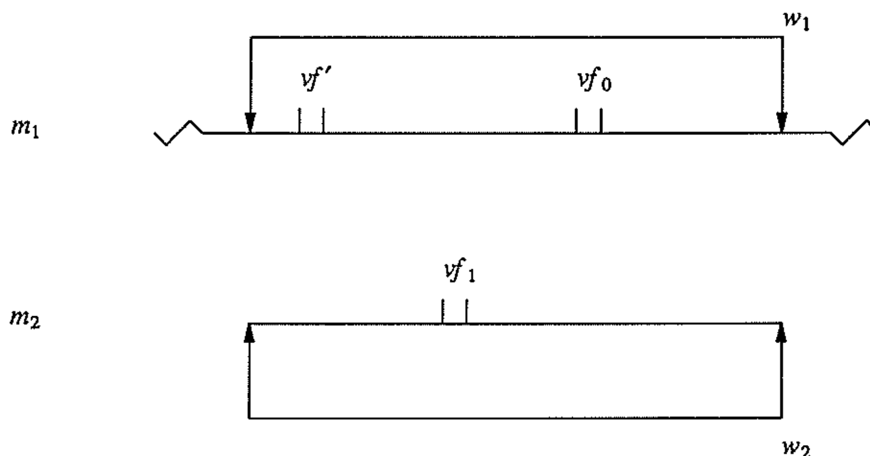


Figure 50: Final case S11.2.2

Recall that m_2 is assumed to be a correct map and that m_1 is correct except for one fragment matching mistake. This means that all of the virtual fragments in Figure 50 are correct, except for those in w_1 that are explicitly denoted in Figure 50 (i.e., vf_0 and vf'). Thus, those virtual fragments in w_1 and w_2 that are not explicitly denoted match in a way that corresponds to the underlying reality. Given this, what would happen if one tried to construct a non-topological matchlist between the virtual fragments in w_1 and w_2? All the virtual fragments in w_2 that are not explicitly denoted match with the virtual fragments in w_1 that are not explicitly denoted. The only explicitly denoted fragment in w_2 is vf_1. It has a length similar to vf_0 and vf'. Thus, vf_1 matches with either vf_0 or vf'. (Recall that topological constraints are not considered.) So all virtual fragments in w_2 are in the non-topological matchlist between w_1 and w_2. In other words, the size of non-topological matchlist *is equal to* the number of virtual fragments in the window of the assimilating map. This is the first observable property of the two maps.

The other observable property of interest is the result of a topological scan. The approach taken here is to assume that certain virtual fragments in m_1 are of a particular class. (Recall that the class of a virtual fragment is determined by a topological scan.) Then the class of certain virtual fragments in m_2 are determined, or at least restricted.

The virtual fragments of interest in final case S11.2.2 are those which are explicitly denoted in Figure 50 (i.e., vf_0, vf_1 and vf'). These fragments are of interest because they are the fragments that have some relationship to the fragment matching mistake. It is these fragments that are likely to stand out (i.e., be class 3) and give clues about the location of the mistake. The other fragments have little relationship to the mistake and thus are not likely to provide useful information about its location.

Fragments vf_0 and vf' are assumed to be of particular classes and from this the class of vf_1 is determined. The analysis is simplified by observing that since vf_0 and vf' are within w_1, they cannot be class 1 fragments. However, they could be class 2, 3 or 4. Each combination is examined and the results are summarized in Table 2.

First, consider entry (1,1) of Table 2. This entry represents the assumption that both vf_0 and vf' are class 2 fragments. This means that both vf_0 and vf' are matched in the ATVML of the topological scan. The virtual fragments in w_1 that are not explicitly denoted match with the virtual fragments in w_2 that are not explicitly denoted in a way that corresponds to the underlying reality. This means that all virtual fragments in w_2 that are not explicitly denoted match with some virtual fragment in w_1 that is not explicitly denoted. Thus, neither vf_0 nor vf' match a virtual fragment in w_2 that is not explicitly denoted. Therefore, the only virtual fragment that vf_0 or vf' could match is vf_1. However, it is impossible for both fragments to match vf_1 at the same time in a particular ATVML. Thus, it is impossible for both vf_0 and vf' to be class 2 fragments. This is reflected in entry (1,1) of Table 2.

Next, consider entry (3,1) of Table 2. This entry represents the assumption that vf_0 is class 2 and vf' is class 4. This means that vf_0 is in the ATVML. It also means that vf' does not match anything, but is near the end of w_1, where it would not prevent the incorporation of m_1 and m_2. The earlier statement about how the fragments not explicitly denoted match applies here as well. Thus, the only virtual fragment in w_2 that vf_0 can match is vf_1. Thus, all virtual fragments in w_1 match or are near the end of w_1 and all virtual fragments in w_2 match. Therefore, m_1 and m_2 incorporate. However, this entire analysis is based upon the assumption that m_1 and m_2 do not incorporate. Thus, this case is of no interest and is not analyzed further. This is reflected in entry (3,1) of Table 2. The analysis of entry (1,3) is essentially the same as that for entry (3,1) and thus is of no interest.

Now consider entry (3,3) of Table 2. This entry represents the assumption that both vf_0 and vf' are class 4 fragments. This means that neither vf_0 nor vf' match, but both are near the end of w_1, where they would not prevent the incorporation of m_1 and m_2. This means no virtual fragment in w_1 is available to match vf_1. Thus, vf_1 cannot be class 2. If vf_1 were class 4, then m_1 and m_2 would incorporate. Again, this entire analysis is based upon the assumption that m_1 and m_2 do not incorporate. Thus, vf_1 cannot be

class 4. Therefore, vf_1 must be class 3. This is reflected in entry (3,3) of Table 2.

Next, consider entry (2,1) of Table 2. This entry represents the assumption that vf_0 is class 2 and vf' is class 3. Thus, vf_0 matches some virtual fragment in w_2. The only virtual fragment in w_2 that it could match is vf_1. Therefore, vf_1 is class 2. This is reflected in entry (2,1) of Table 2. The analysis of entry (1,2) is symmetric to that of entry (2,1). This is reflected in entry (1,2) of Table 2.

Now consider entry (2,2) of Table 2. This entry represents the assumption that both vf_0 and vf' are class 3 fragments. This means that neither vf_0 nor vf' match any fragments in w_2, but both are in a position that would prevent the incorporation of m_1 and m_2. That is, neither vf_0 nor vf' are near an end of w_1. However, it is highly likely that vf_1 is somewhere between vf_0 and vf'. Thus, it is highly unlikely that vf_1 is near an end of w_2. Therefore, it is unlikely that vf_1 is class 4. In addition, since neither vf_0 nor vf' match any fragments in w_2, there are no fragments that could match with vf_1. So vf_1 is not class 2 either. Thus, vf_1 must be class 3. This is reflected in entry (2,2) of Table 2.

Next, consider entry (3,2) of Table 2. This entry represents the assumption that vf_0 is class 3 and vf' is class 4. For reasons similar to those of entry (2,2), it is highly unlikely that vf_1 is near an end of w_2 and there are no fragments that could match vf_1. Thus, vf_1 is class 3. This is reflected in entry (3,2) of Table 2. The analysis of entry (2,3) is symmetric to that of entry (3,2). This is reflected in entry (2,3) of Table 2.

Now each entry in Table 2 has been examined in detail. In the next section, the results of similar analyses for the other final cases in the limited enumeration are presented, without the level of detail given in this section.

3.4. 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. For each case, a sketch of the underlying reality and a table (in the same format as Table 2) summarizing the results of the analysis are given. In cases that involve an assimilation, the data concerning the maximum size non-topological matchlist (denoted *ml*) is given directly above the table.

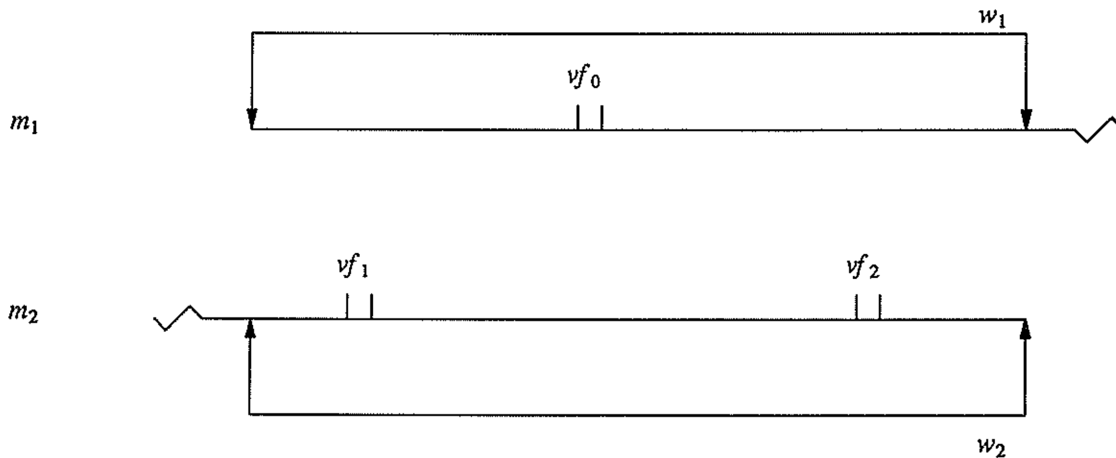


Figure 51: A sketch of final case S1.1.1

class(vf_0)	
2	vf_0=2, vf_1=2, vf_2=3 OR vf_0=2, vf_1=3, vf_2=2
3	vf_0=3, vf_1=3, vf_2=3
4	vf_0=4, vf_1=3, vf_2=3

Table 5
Possible results of a topological scan for final case S1.1.1

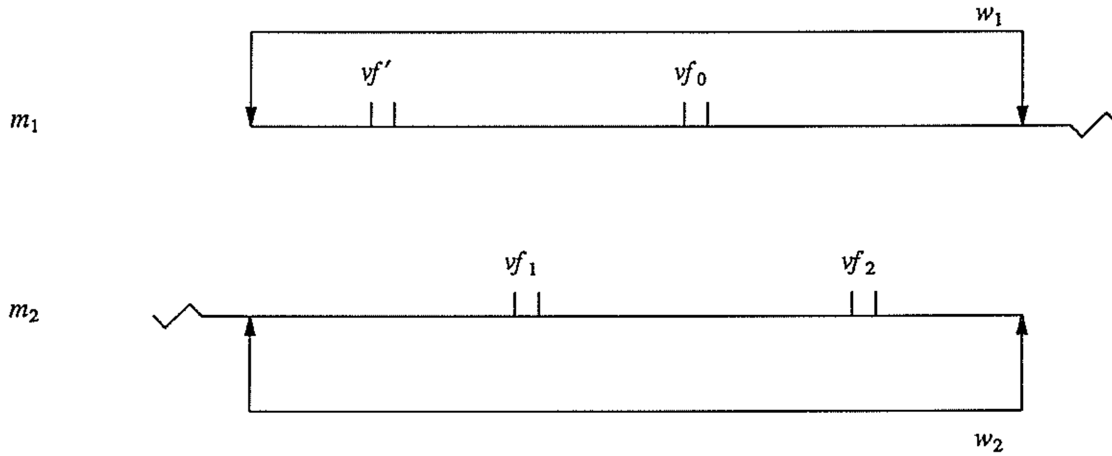


Figure 52: A sketch of final case S1.1.2

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

Table 6
Possible results of a topological scan for final case S1.1.2

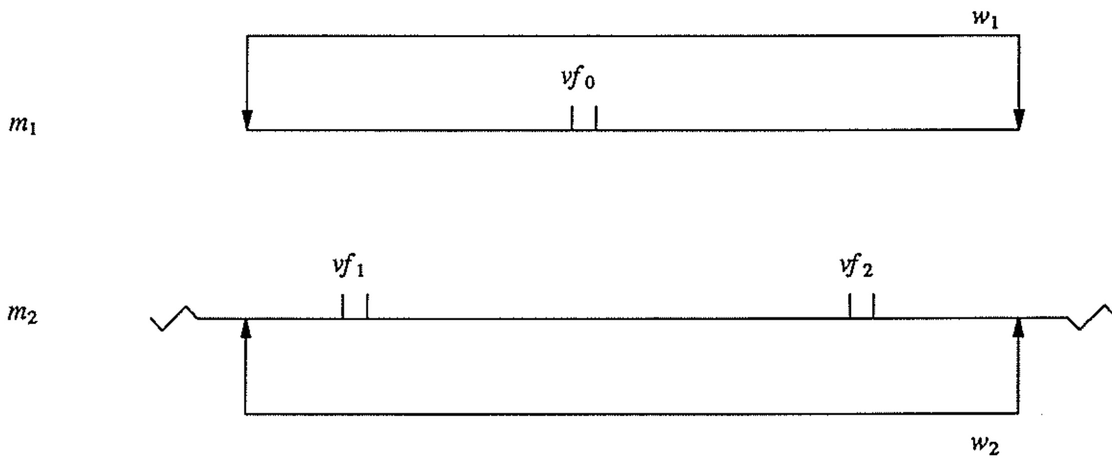


Figure 53: A sketch of final case S1.2.1

$|ml| = |w_1|$

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

Table 7
Possible results of a topological scan for final case S1.2.1

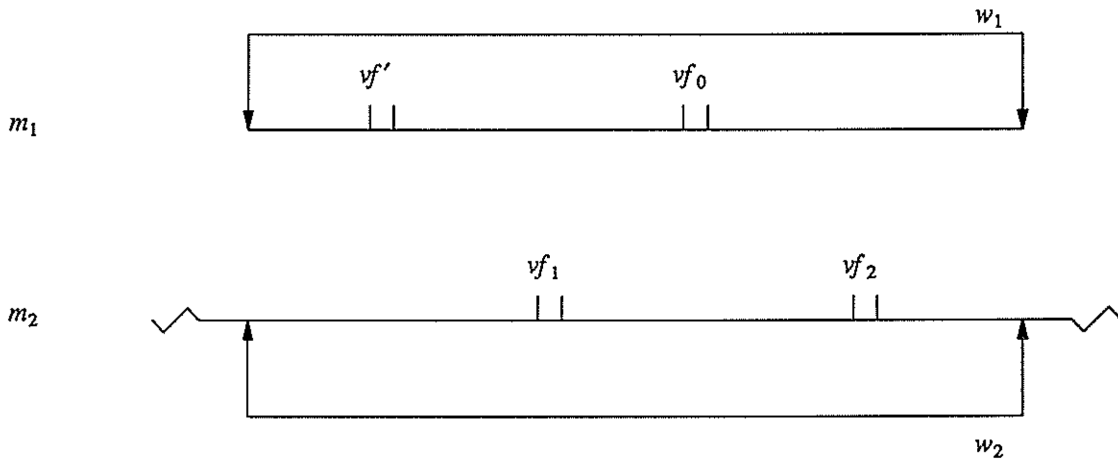


Figure 54: A sketch of final case S1.2.2

$|ml| = |w_1|$

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

Table 8
Possible results of a topological scan for final case S1.2.2

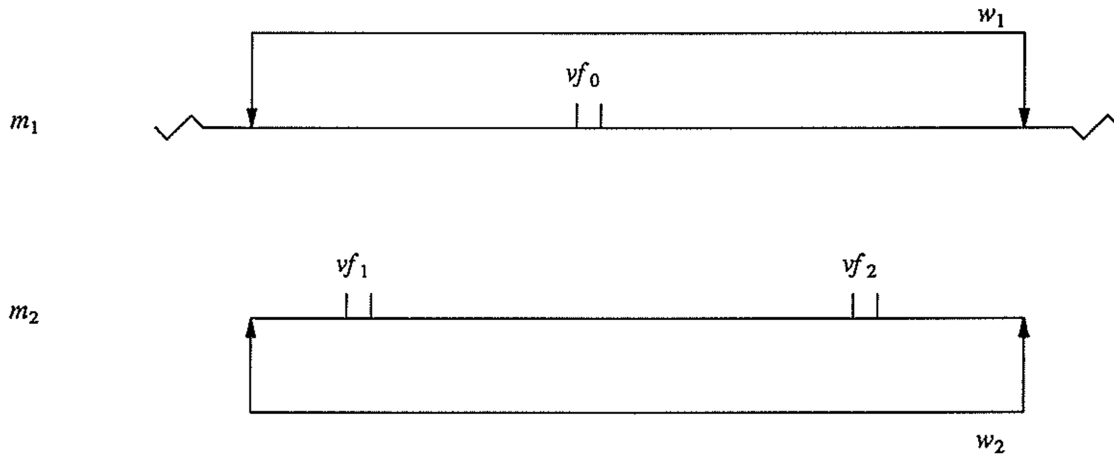


Figure 55: A sketch of final case S1.3.1

$|ml| = |w_2| - 1$

class(vf_0)	
2	vf_0=2, vf_1=3, vf_2=2 OR vf_0=2, vf_1=2, vf_2=3
3	vf_0=3, vf_1=3, vf_2=3
4	vf_0=4, vf_1=3, vf_2=3

Table 9
Possible results of a topological scan for final case S1.3.1

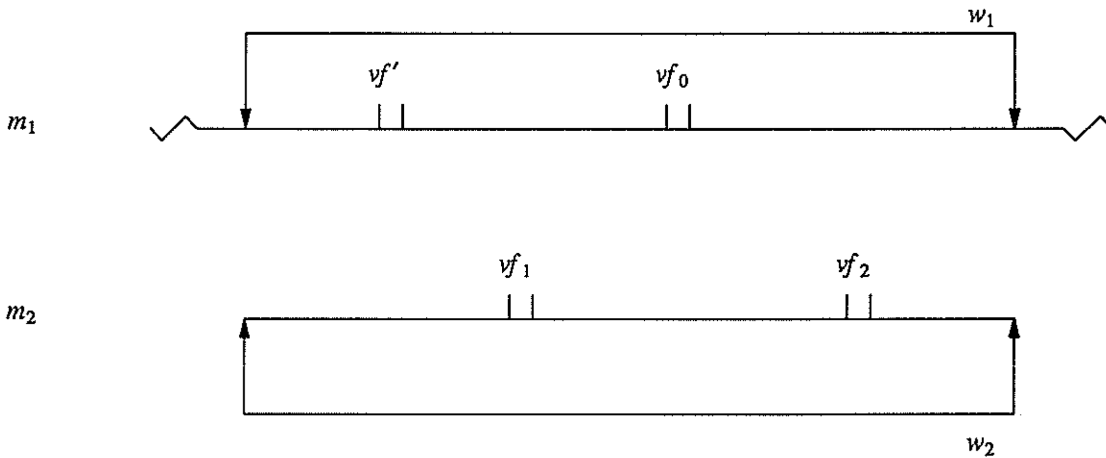


Figure 56: A sketch of final case S1.3.2

$|ml| = |w_2|$

class(vf_0) → class(vf') ↓	2	3	4
2	INC	vf_0=3, vf_1=3, vf_2=2, vf'=2 OR vf_0=3, vf_1=2, vf_2=3, vf'=2	vf_0=4, vf_1=3, vf_2=2, vf'=2 OR vf_0=4, vf_1=2, vf_2=3, vf'=2
3	vf_0=2, vf_1=3, vf_2=2, vf'=3 OR vf_0=2, vf_1=2, vf_2=3, vf'=3	vf_0=3, vf_1=3, vf_2=3, vf'=3	vf_0=4, vf_1=3, vf_2=3, vf'=3
4	vf_0=2, vf_1=3, vf_2=2, vf'=4 OR vf_0=2, vf_1=2, vf_2=3, vf'=4	vf_0=3, vf_1=3, vf_2=3, vf'=4	vf_0=4, vf_1=3, vf_2=3, vf'=4

Table 10
Possible results of a topological scan for final case S1.3.2

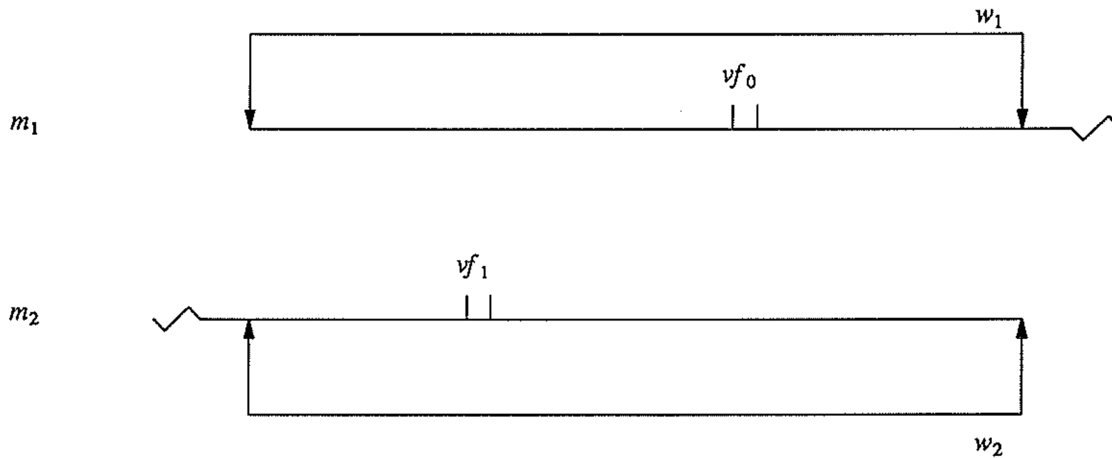


Figure 57: A sketch of final case S11.1.1