Washington University in St. Louis
Washington University Open Scholarship

All Computer Science and Engineering Research | Computer Science and Engineering

Report Number: WUCS-91-50

1991-10-01

DNA Mapping Algorithms: Fragment Splitting and Combining

James Daues and Will Gillet

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. This technical report discusses techniques for fragment splitting and fragment combining, which attempts to correct maps containing a fragment matching mistake, given that the location of the mistake is known. These techniques are based on operations that decompose and merge virtual fragments (a collection of matched real fragments), add virtual fragments to existing groups in the map and insert virtual fragments between existing groups in the map. Examples of... **Read complete abstract on page 2.**

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

Recommended Citation
Daues, James and Gillet, Will, "DNA Mapping Algorithms: Fragment Splitting and Combining" Report Number: WUCS-91-50 (1991). *All Computer Science and Engineering Research.*
https://openscholarship.wustl.edu/cse_research/668

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/668

DNA Mapping Algorithms: Fragment Splitting and Combining

James Daues and Will Gillet

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. This technical report discusses techniques for fragment splitting and fragment combining, which attempts to correct maps containing a fragment matching mistake, given that the location of the mistake is known. These techniques are based on operations that decompose and merge virtual fragments (a collection of matched real fragments), add virtual fragments to existing groups in the map and insert virtual fragments between existing groups in the map. Examples of applying these techniques to both contrived and actual DNA maps are presented.

DNA Mapping Algorithms: Fragment Splitting and Combining

James Daues and Will Gillett

WUCS-91-50

October 1991

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

This work was supported by the James S. McDonnell Foundation under Grant 87-24 and NIH under Grant 1R01 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. This technical report discusses techniques for fragment splitting and fragment combining, which attempt to correct maps containing a fragment matching mistake, given that the location of the mistake is known. These techniques are based on operations that decompose and merge *virtual fragments* (a collection of matched real fragments), add virtual fragments to existing groups in the map and insert virtual fragments between existing groups in the map. Examples of applying these techniques to both contrived and actual DNA maps are presented.

TABLE OF CONTENTS

1. Introduction to the Fragment Splitting Problem 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 3
1.2.3. Mapping a Set of Clones 4
1.3. Important Properties of Map Units 9
1.4. The Fragment Matching Mistake 9

2. Basic Operations in the General Fragment Splitting Algorithm 14
2.1. The add Operation 15
2.2. The insert Operation 15
2.3. The split Operation 17
2.4. The combine Operation 18

3. The General Fragment Splitting Algorithm 19
3.1. A High Level Description 19
3.2. Examples of the General Fragment Splitting Algorithm 20
3.2.1. Example One 20
3.2.2. Example Two 21
3.2.3. Example Three 23
3.3. Problems with the General Fragment Splitting Algorithm 25

4. The Restricted Fragment Splitting Algorithm 27
4.1. Desirable Map Unit Characteristics 27
4.1.1. Compactness 27
4.1.2. Single Mistake Fixing 27
4.2. New Characteristics of the Basic Operations 27
4.2.1. Directionality 28
4.2.2. Operation Bias 28
4.2.3. Bipartite Splits 29
4.3. Basic Operations in the Restricted Fragment Splitting Algorithm 30
4.3.1. The add Operation 30
4.3.2. The insert Operation 31
4.3.3. The Split Operations 32
4.3.3.1. The undirected_split Operation 32
4.3.3.2. The directed_split Operation 34
4.3.3.3. The internal_split Operation 38
4.3.3.4. The top_level_split Operation 40
4.3.4. The Combine Operations 40
4.3.4.1. The directed_combine Operation 42
4.3.4.2. The undirected_combine Operation 44
4.3.4.3. The top_level_combine Operation 46
4.3.5. Calling Relationship between the Operations 47
4.4. High Level Description 47

4.5. Examples of Using top_level_split 47
4.5.1. Example One 47
4.5.2. Example Two 49
4.5.3. Example Three 50
4.5.4. Example Four 51
4.5.5. Example Five 53
4.6. Example of Using top_level_combine 54
4.7. Examples on Real Map Units 55
4.7.1. Map Unit CONTG309 55
4.7.2. Map Unit MERG0515 57
4.7.3. Map Units MERG0615 and MERG0576 59
4.8. Implementation Considerations 62
4.8.1. Bipartitions of Clones 62
4.8.2. Actions 63
4.8.3. User Options 64

5. Conclusion 65

Appendix A: Description of Functions and Procedures in the Psuedocode 67

TABLE OF FIGURES

Figure 1: Set of clones to map 4
Figure 2: Map unit produced from mapping Clone #1 and #3 5
Figure 3: Clones 1, 3, and 4 6
Figure 4: Clones 1, 2, 3, and 4 6
Figure 5: Completed mapping of clone set 7
Figure 6: An ambiguous assimilation 8
Figure 7: Subset ambiguity 10
Figure 8: Two overlapping clones 11
Figure 9: The fingerprints of the clones in Figure 8 11
Figure 10: Map unit built from the fingerprints in Figure 9 12
Figure 11: Map unit resulting from a split and two adds 12
Figure 12: Three overlapping clones 13
Figure 13: The fingerprints of the clones in Figure 12 13
Figure 14: Map unit built from the fingerprints in Figure 13 13
Figure 15: Map unit resulting from a split, combine and two adds 14
Figure 16: Psuedocode for the GSA version of add 15
Figure 17: Psuedocode for the GSA version of insert 16
Figure 18: Clone configuration for Example one 20
Figure 19: Map unit built from clones in Figure 18 20
Figure 20: Map unit resulting from the split 21
Figure 21: Clone configuration for Example two 22
Figure 22: Map unit built from clones in Figure 21 22
Figure 23: The first map unit resulting from the split 22
Figure 24: The second map unit resulting from the split 23

Figure 25: An alternate clone configuration for Example two ... 23
Figure 26: Clone configuration for Example three ... 24
Figure 27: Map unit built from clones in Figure 26 ... 24
Figure 28: Map units resulting from the split ... 26
Figure 29: A possible clone configuration ... 28
Figure 30: A possible clone configuration ... 29
Figure 31: Map unit built from clones in Figure 30 ... 30
Figure 32: Psuedocode for the RSA version of add ... 31
Figure 33: Psuedocode for the RSA version of insert ... 32
Figure 34: Subfragment placement and directionality constraints ... 33
Figure 35: Construction of a solution map unit in undirected_split ... 35
Figure 36: Psuedocode for undirected_split ... 36
Figure 37: Psuedocode for directed_split ... 37
Figure 38: Psuedocode for internal_split ... 39
Figure 39: A typical internal_split ... 40
Figure 40: Psuedocode for top_level_split ... 41
Figure 41: Psuedocode for directed_combine ... 43
Figure 42: Psuedocode for undirected_combine ... 45
Figure 43: Psuedocode for top_level_combine ... 46
Figure 44: Calling relationship between the RSA operations ... 48
Figure 45: Clone configuration for Example two ... 49
Figure 46: Map unit built from clones in Figure 45 ... 49
Figure 47: Map unit resulting from the top_level_split ... 51
Figure 48: Clone configuration for Example four ... 52
Figure 49: Map unit built from clones in Figure 48 ... 52
Figure 50: Map unit resulting from the top_level_split ... 53
Figure 51: Map unit built from clones in Figure 25 ... 53
Figure 52: Map unit resulting from the top_level_split ... 53
Figure 53: Map unit resulting from the top_level_split with direction ... 54
Figure 54: Clone configuration for the top_level_combine example ... 55
Figure 55: Map unit built from clones in Figure 54 ... 55
Figure 56: Map units CONTG309, MERG0542 and a matchlist ... 56
Figure 57: Map units CONTG309*, MERG0542 and a matchlist ... 57
Figure 58: Map units MERG0515, MERG0411 and a matchlist ... 58
Figure 59: Map units MERG0515*, MERG0411 and a matchlist ... 59
Figure 60: Map units MERG0615, MERG0576 and a matchlist ... 60
Figure 61: Map units MERG0615, MERG0576* and a matchlist ... 61
Figure 62: Map units MERG0615*, MERG0576* and a matchlist ... 62
Figure 63: Regions of a map unit not affected ... 64
Figure 64: Header in the implementation of the top_level_split operation ... 65
Figure 65: Header in the implementation of the top_level_combine operation ... 65

1. Introduction to the Fragment Splitting Problem

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 a "base". There are 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 "pair" 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 DNA molecule is made of two complementary DNA nucleotide strands bound together by this base pairing, the base sequence on one strand determining the complementary sequence on the other strand.

DNA restriction mapping deals with determining the positions of specific sites of interest along a given DNA strand, or **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 exactly these restriction sites. Thus, a restriction enzyme reacting with a strand of DNA will produce fragments of DNA whose lengths are exactly the distance between two successive restriction sites along the original DNA. The process of **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 fracturing 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 restriction fragments within it. 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 due to 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) order of large groups of fragments may be inferrable. As more clones are found to overlap a specific region of the original genome, the random 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 most compact is 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 the fingerprint clone data. Prior to any mapping, the original 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 restriction enzyme; this produces random **clone inserts**. The partial digestion process causes different copies of the DNA to be cleaved at some (randomly selected) restriction site, but not all restriction sites. This tends to produce clone inserts which have random overlap with one another. These clone inserts are then inserted into a biological organism known as a **lambda phage**. The size of these inserts is limited to roughly between 10,000 and 25,000 base pairs (bp); these length restrictions are caused by the packaging mechanism used by the lambda phage. The combination of the lambda phage and the inserted DNA is known as a **clone**.

During the creation of this initial biological data, enough lambda clones are created so that a **redundancy** of approximately five is produced, i.e., any region of DNA from the original genome is likely to appear in about five clones. 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 overlap may range from **partial overlap**, where each insert contains DNA besides the region of overlap, to **total overlap**, where one insert is simply 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 rejoined in the order in which they existed in the original genome.

After the clones are formed, further processing is done on them. First, the clones are separated by a multi-level dilution process, and colonies resulting from a single clone are grown to produce enough DNA for subsequent processing. For each clone, the clone DNA extracted from this augmentation 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 (in base pairs) are then measured using **electrophoresis gel** technology. Upon placing an electric current 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 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 on the gel. After staining the gel, these bands can be detected and their positions on the gel determined. **Reference lanes**, containing DNA fragments of known length, also are present on the gel. Using the positions of the bands present in these reference lanes and the process of interpolation, it is possible to estimate the lengths of restriction fragments in the data lanes. Unfortunately, electrophoresis technology is incapable of accurately detecting fragments whose lengths are outside the range of 400 bp to 7.5 kilobase pairs (kb). However, since the majority of the restriction fragment length data falls in this range, this is not a serious problem.

There are (at least) two sources of error which create uncertainty about the data produced by electrophoresis. The first is the classical problem of measurement error. From experimental evidence, there is approximately a 3% error window around the true length of the fragment, 1.5% on either side of the actual length. Because of this, a fragment which is 1000 bp in length may be measured anywhere from 985 bp to 1015 bp. The second deals with determining the multiplicity of fragments of similar length. Two fragments of identical or nearly identical length will migrate to the "same" location on the gel. Thus, it is possible for two (or three, or four, etc.) fragments to be in the same band when the gel is stained. If this is not taken into account, the list of fragment lengths will not accurately reflect the number of fragments which exist in the clone. It is possible but difficult to identify this situation. The intensity of the stained DNA bands should decrease along the expanse the gel, due to the fact that there is less DNA material 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.

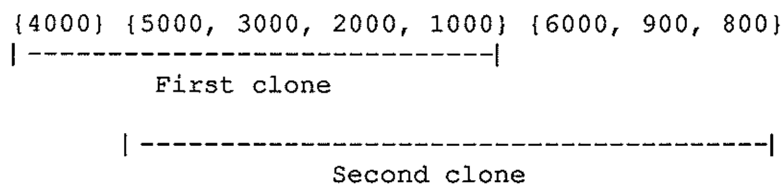
Both of these sources of error complicate the process of DNA mapping. If something will not map together, but there is a high likelihood that it should, it is possible that data extracted from the gel were incorrectly handled in the data gathering process. By going back and examining the original gel, the inconsistency may be explained and/or resolved.

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 because they contain four fragments of the same lengths is no guarantee that they *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 multiple **apparent overlap** 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 an overlapping region 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 any one of the 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 1 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, no knowledge about the ordering the fragments in any of the clones is known.

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 first. 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 **matchlist** (i.e., 6198—6109, 4082—4087, 1150—1139, 1092—1078, 637—630, 513—

#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 1: Set of clones to map

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 2. 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 can be formed from 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 2 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 $rf_1, \ldots, rf_n$. 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 fewer number of fragments will be considered more **compact** than the map unit containing the larger number of fragments.

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 2). 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 3.

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 4 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 shown in Figure 5.

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

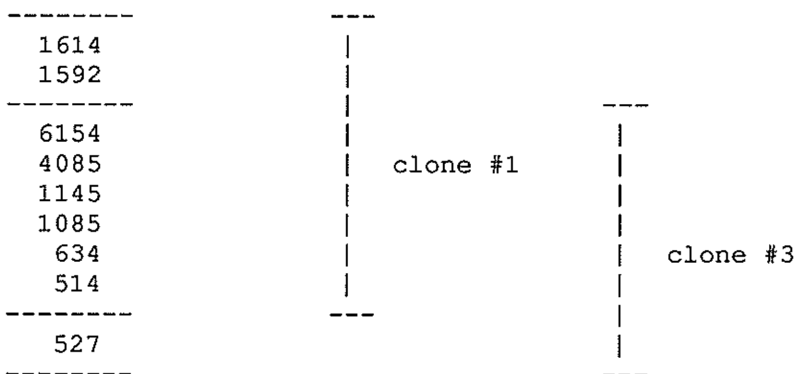


Figure 2: Map unit produced from mapping Clone #1 and #3

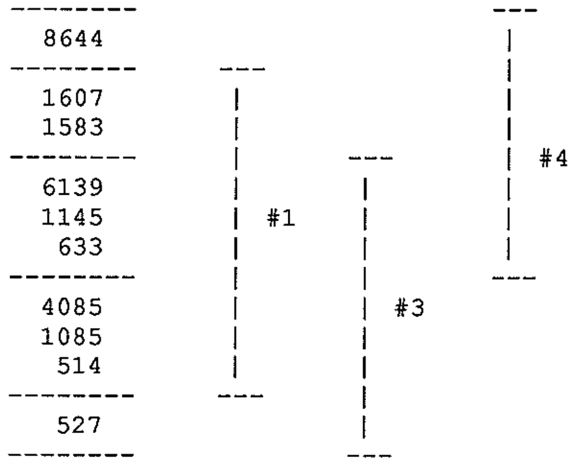


Figure 3: Clones 1, 3, and 4

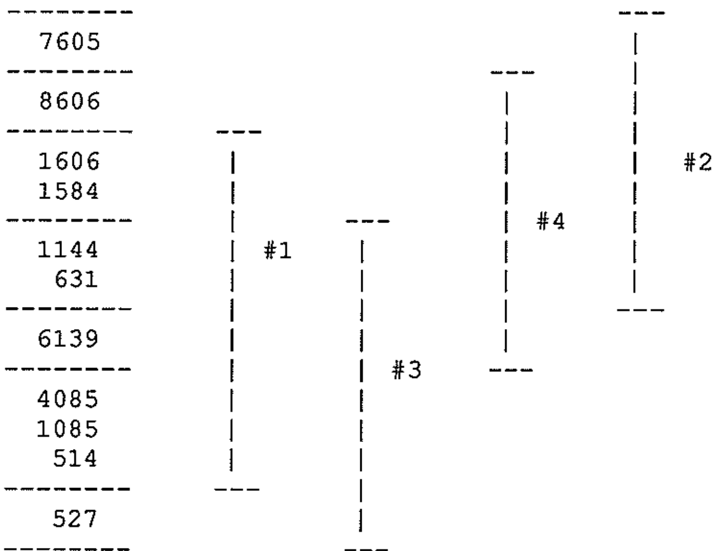


Figure 4: Clones 1, 2, 3, and 4

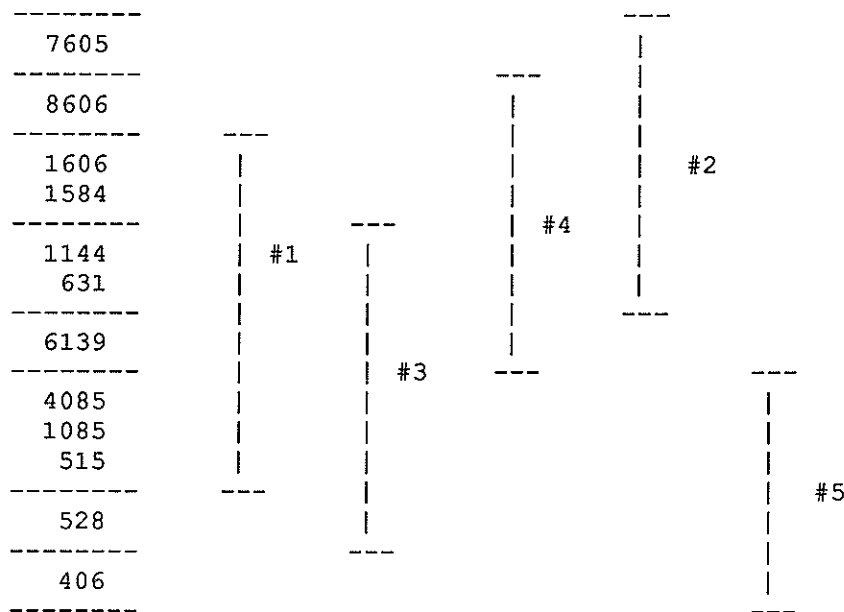


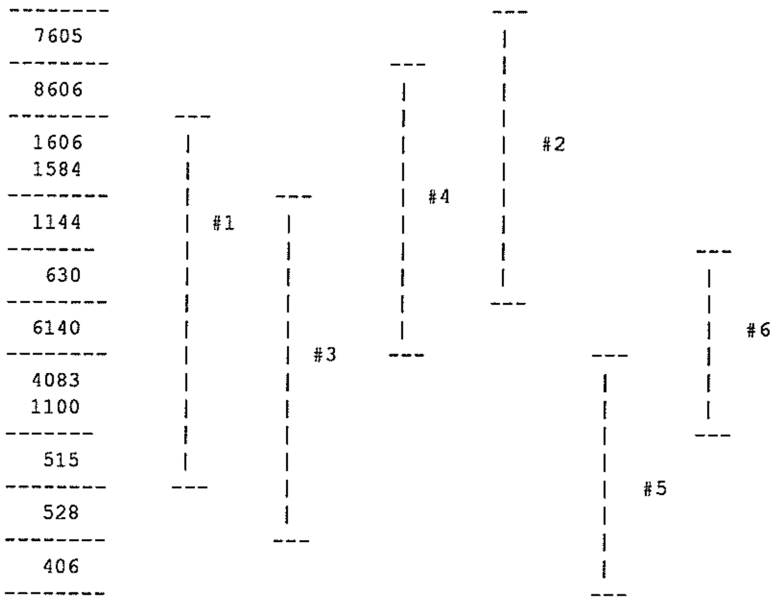
Figure 5: Completed mapping of clone set

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.

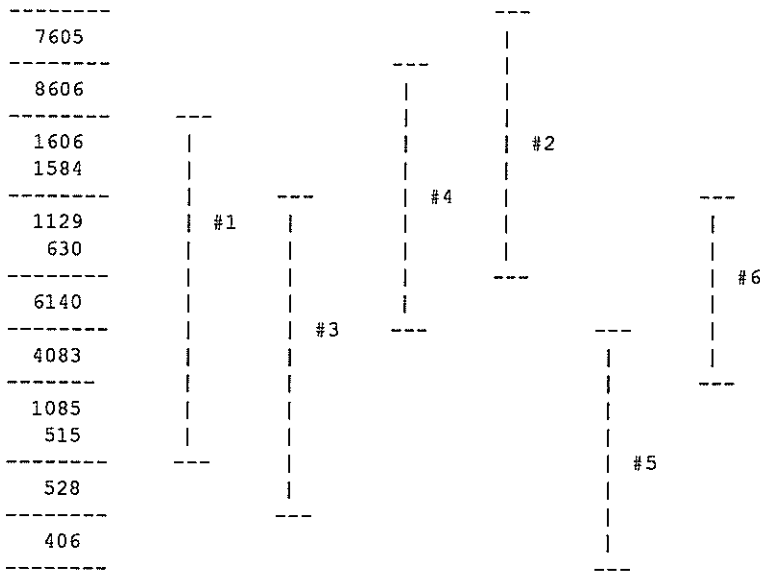
It might be possible to incorporate a clone into a map unit in more than one way. Such a situation is referred to as **ambiguous**. As an example, assume that there is a sixth clone, Clone #6 with fragment set {6142, 4081, 1115, 629}, which is suspected to be from the same region of the genome as Clones #1 through #5. This new clone can be assimilated into the map unit shown in Figure 5. In fact, it can be assimilated in two different ways. The two possible matchlists 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 6.

The structure of these two map units is significantly different. For instance, the map unit in Figure 6(a) contains one more group than the map unit in Figure 6(b). Also the map unit in Figure 6(a) restricts fragment 1144 to be adjacent to the group containing fragments 1584 and 1606, whereas the map unit in Figure 6(a) does not. Making a decision *now* about where a fragment *must* reside (when the decision is clearly in question) can have significant ramifications to 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 matchlists allow map units of different structure to be constructed. This is illustrated by the example associated with Figure 6. 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



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



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

Figure 6: An ambiguous assimilation

fragments of the group. As an example of this type of ambiguity, assume the state of clone mapping is as depicted in Figure 2, 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 matchlist (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 7. Each of these map units represents a significantly different set of underlying realities, and none of the map units are compatible with one another.

This example was complex enough to demonstrate the general nature 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.

1.3. Important Properties of Map Units

In this section, some properties of map units that are particularly important for fragment splitting are discussed.

Recall that a virtual fragment is a set of real fragments, each real fragment coming from some specific clone. The **active clone set** (or **ACS**) of vf is defined to be the set of clones from which the real fragments composing vf come. All virtual fragments within a particular group have the same ACS. This is a result of the way that clone overlap is used to build map units. Thus, the ACS of a group is defined to be the ACS of any of the virtual fragments contained in the group. [ACS(x) denotes the ACS of an object x.]

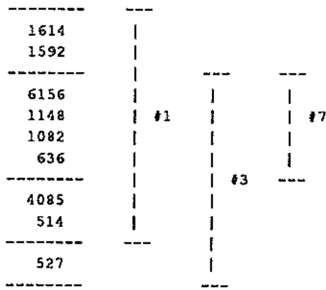
Using the style of map unit building described in §1.1 and §1.2, it is impossible for two different groups in a map unit to have the same ACS. This is a result of the way clone overlap is used to build map units and the fact that ambiguous mappings are not used. (The fact that mappings with subset ambiguity are not used is particularly important.)

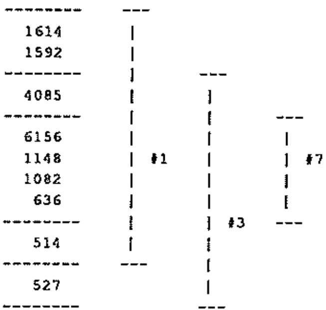
It is assumed that all clones used in mapping contain two or more fragments, because, for the most part, clones containing less than two fragments are not useful. The primary reason is that regardless of the minimum required overlap, a clone with less than two fragments cannot refine the fragment ordering of a map unit upon fusing. Another reason that clones with less than two fragments are not useful is related to the fragment overlap requirement used during mapping. In order to fuse a clone with a map unit, the number of fragments that match must be above some predetermined threshold x. If this is the case, a clone with less than x fragments can never be fused with a map unit. Thus, all clones with less than x fragments can be discarded before any mapping occurs. So given a map unit m created using the threshold x, one knows that all clones in m contain at least x fragments. Thus if the threshold is more than one, no clones with less than two fragments are ever fused into the map unit. Using a threshold of one is possible, but it is very impractical. Consequently, in this paper, it is assumed that the threshold is high enough so that no clones with less than two fragments are used in the mapping process.

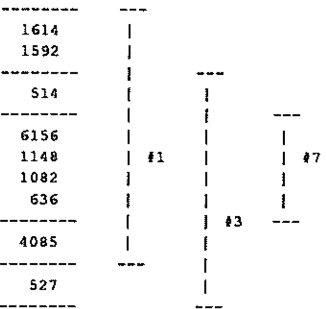
1.4. The Fragment Matching Mistake

This section attempts to expose, by example, the fundamental fragment confusion error that often occurs during mapping, for which a fragment splitting algorithm is needed to resolve. Consider the simple case in which two clones are being considered for possible overlap. 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 greater than or equal to some predetermined threshold (four is a typical value), then actual overlap is declared and the real fragment matches are used to produce virtual fragments within the resulting map unit.

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 (within 3%) does not imply they do actually come from the







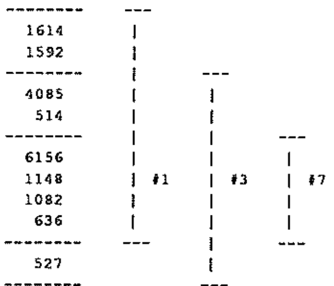


Figure 7: Subset ambiguity

exact same section of the original genome. A **fragment matching mistake** can occur when two real

fragments of similar length occur in the genome roughly one clone length apart. Consider the example illustrated in Figure 8.

The top horizontal line in Figure 8 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 8 represents reality, Figure 9 gives two fingerprints that might be obtained from gel electrophoresis with 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 labelled f_1 and f_2.

It is the data in Figure 9 that is used to determine if Clones c_1 and c_2 overlap. 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 maximum size matchlist.) There is no information available to indicate that real fragments f_1 and f_2 do not come from the same position of the genome, and thus should not be matched. Thus this maximum matching would be used to form the Map unit m in Figure 10, 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

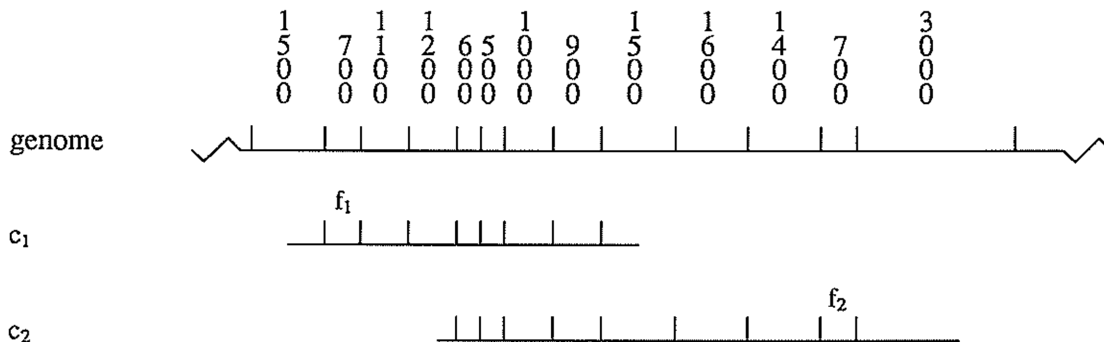


Figure 8: Two overlapping clones

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

Figure 9: The fingerprints of the clones in Figure 8

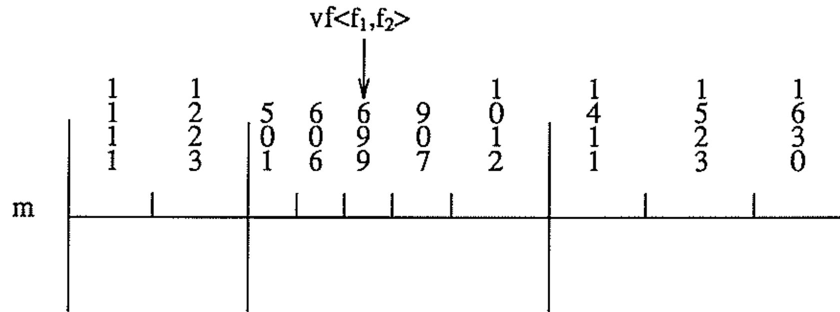


Figure 10: Map unit built from the fingerprints in Figure 9

or she would want to decompose, or **split**, it into two virtual fragments, in order to undo the effects of the incorrect match. The only way to split $vf<f_1,f_2>$ is to create virtual fragments $vf<f_1>$ and $vf<f_2>$. In the process of splitting this virtual fragment, $vf<f_1,f_2>$ is taken out of the map unit, and the resulting fragments, $vf<f_1>$ and $vf<f_2>$, must be incorporated in some manner. The appropriate place to put a virtual fragment vf is in a group whose virtual fragments are composed of real fragments from the same set of clones as vf. Thus, $vf<f_1>$ is placed in, or **added** to, the leftmost group, since both have an ACS of $\{c_1\}$. Likewise, $vf<f_2>$ is added to the rightmost group, since both have an ACS of $\{c_2\}$. These modifications produce the map unit given in Figure 11. Note that this map unit corresponds to the underlying reality.

The previous example is very straightforward. A slightly more complicated example is now presented. Suppose Figure 12 represents the underlying reality. Figure 13 gives possible fingerprints for the three clones. (Again, random error has been introduced.) Fusing Clones c_1 and c_2 first, followed by Clone c_3, produces the Map unit m given in Figure 14. When c_1 and c_2 are fused, f_1 incorrectly matches with f_2. Because of this, when Clone c_3 is fused, any matchlist that matches f_3 with $vf<f_1,f_2>$ will not be topologically valid. Thus, any successful fusion of c_3 does not match f_3 with $vf<f_1,f_2>$.

Suppose that the mapper suspects that $vf<f_1,f_2>$ in m needs to be split. As in the previous example, there is only one way to split $vf<f_1,f_2>$. After $vf<f_1,f_2>$ is removed from m, virtual fragments $vf<f_1>$ and $vf<f_2>$ must be incorporated into the map unit. The proper place for $vf<f_1>$ is in the leftmost group of m. As in the previous example, $vf<f_2>$ could be added to the group with an ACS of $\{c_2\}$. However, since $vf<f_2>$ was incorrectly matched, one should look for fragments that it *could* have matched had the mistake not occured. There is a virtual fragment nearby of similar length, namely $vf<f_3>$. Thus $vf<f_3>$ is removed from m, and is matched with $vf<f_2>$ to make a new virtual fragment, $vf<f_2,f_3>$. That is, $vf<f_2>$ and $vf<f_3>$ are **combined** to form $vf<f_2,f_3>$. Then the proper place for

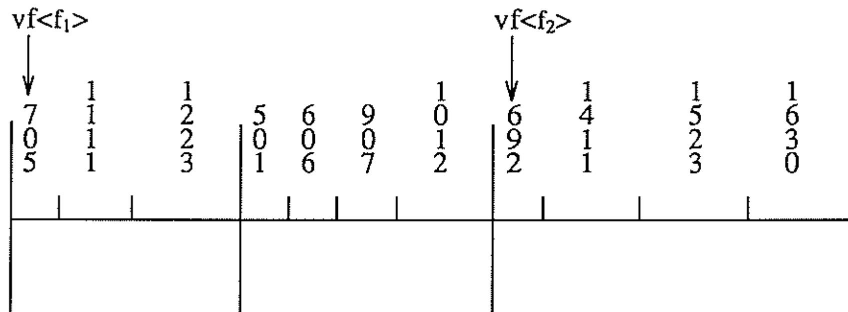


Figure 11: Map unit resulting from a split and two adds

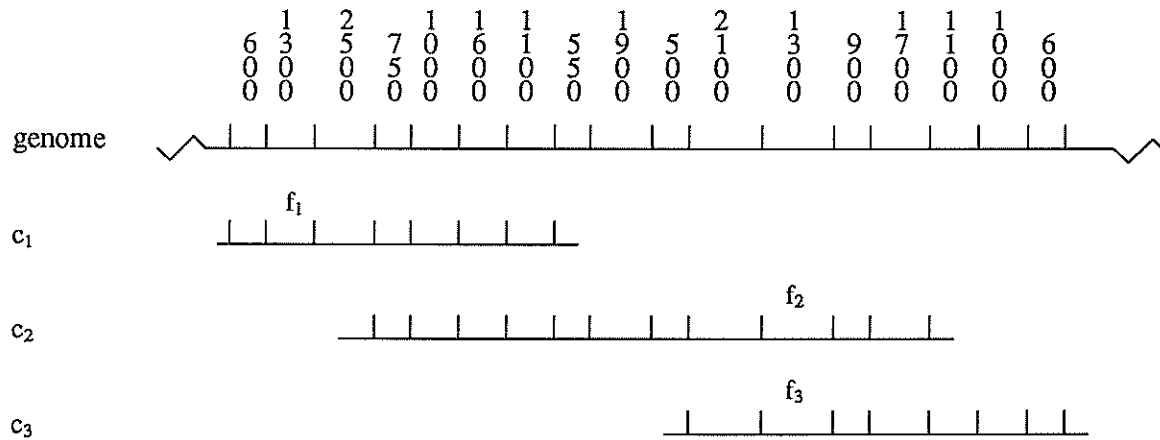


Figure 12: Three overlapping clones

c_1	c_2	c_3
598	506	608
741	557	892
989	752	993
1092	904	1091
1292(f_1)	1009	1295(f_3)
1585	1110	1711
2481	1303(f_2)	2103
	1605	
	1707	
	1913	
	2113	

Figure 13: The fingerprints of the clones in Figure 12

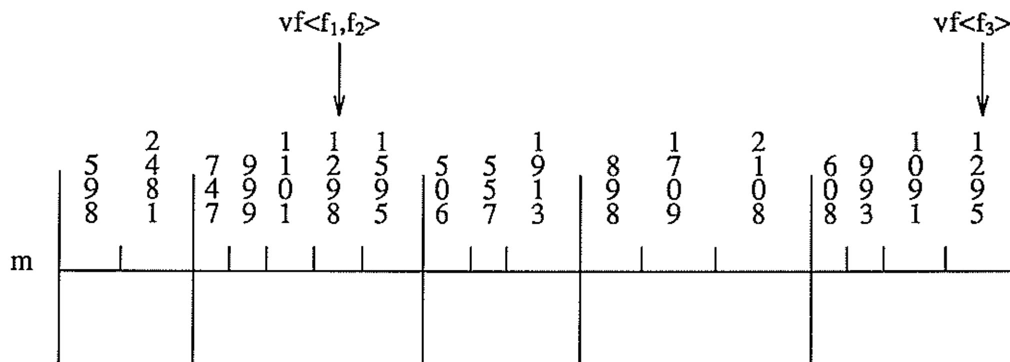


Figure 14: Map unit built from the fingerprints in Figure 13

$vf<f_2,f_3>$ is in the group with an ACS of $\{c_2,c_3\}$. The map unit that results is given in Figure 15. Note that this map unit corresponds to the underlying reality.

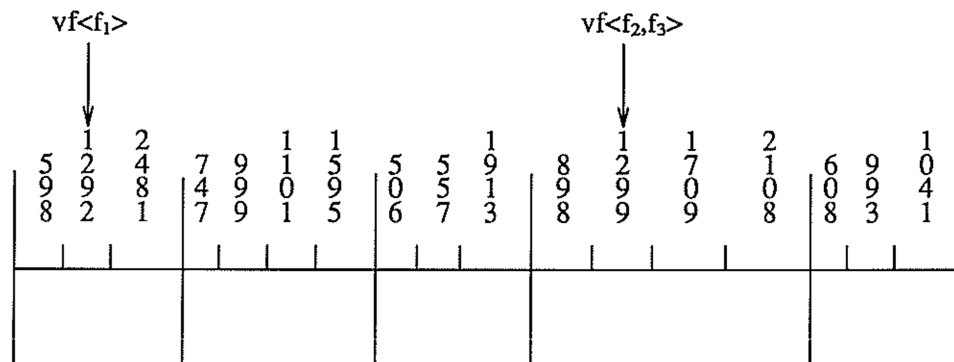


Figure 15: Map unit resulting from a split, combine and two adds

The two examples presented here are relatively simple. However, even in more complex situations, the same type of mistake can occur and the same type of actions may help resolve the situation. In the more complicated situations, a virtual fragment formed by a combination (such as $vf<f_2,f_3>$) may have no group to which it could be added. Instead of declaring the attempt to fix the map unit a failure, one should split this virtual fragment into several virtual fragments and then try to incorporate these fragments into the map unit. Thus, the original split may lead to a second split, which in turn may lead to a third split, and so on. This "rippling" effect is an important feature of the algorithm that is to be presented.

Thus there are two major issues that must be confronted:
(1) How does one detect when and where a fragment matching mistake has occured?
(2) How does one "fix" a map unit once a fragment matching mistake has been detected?

Issue (2) will be addressed in this paper. In §2, four basic operations on virtual fragments: add, insert, split and combine, are presented. These operations form the basis for the **General Fragment Splitting Algorithm** (alternatively called the **General Splitting Algorithm** or **GSA**), which finds *all* map units that result from splitting a virtual fragment. The GSA is described, primarily through examples, in §3. However, the GSA is not practical (computationally) and is difficult to implement. To resolve these problems, the operations in §2 are modified, by introducing the concepts of directionality and operation bias, to create a new set of operations which form the basis for the **Restricted Fragment Splitting Algorithm** (alternatively called the **Restricted Splitting Algorithm** or **RSA**). The intent of the RSA is to find only the more desirable map units that result from splitting a virtual fragment, instead of finding all of them. (Only the RSA has actually been implemented.) The RSA is described, using psuedocode and examples, in §4. Issue (1) will be the subject of a future paper.

2. Basic Operations in the General Fragment Splitting Algorithm

The goal is to develop an algorithm which takes a map unit containing a fragment matching mistake and generates the map unit (or map units) that would have been built if the mistake had not occured. In addition, the algorithm should accomplish this without having to start the mapping process from the beginning. The second property is desirable because a large map unit containing many clones can take a long time to build, and the part of the map unit affected by the fragment matching mistake may be very small in comparison to the size of the map unit.

Such an algorithm can be built on top of a set of four operations that manipulate virtual fragments within a map unit: add, insert, split and combine. The following sections describe these operations.

2.1. The add Operation

Recall that a group is a set of virtual fragments in a map unit whose relative order is unknown. That is, the order of the virtual fragments within the group is unknown. In the examples in §1.4, it can be seen that when attempting to fix a fragment matching mistake, one may want to place a virtual fragment within a particular group. This is the purpose of the add operation. The psuedocode for the add operation is given in Figure 16. (All of the functions and procedures used in the psuedocode are described in Appendix A.) Add takes a virtual fragment vf and a group g in a map unit m and attempts to place vf in g. However, one cannot place a specific virtual fragment in an arbitrary group. The placing of vf must not violate certain constraints placed on m by the previously inferred clone overlaps.

It was stated in §1.3 that all virtual fragments within a group must have the same ACS. Consequently, the add operation succeeds if and only if ACS(vf) is equal to ACS(g).

If ACS(vf) is equal to ACS(g), then vf is added to the set of fragments contained in g_{new} (the group in the newly created map corresponding to g), and the new map unit created is returned. If the ACSs are not equal, then a special token FAIL is returned indicating that the add did not succeed.

Note that add does not use any of the other three operations: insert, split or combine. Thus add is called a **primitive** operation. Operations that use the other operations are called **nonprimitive** operations. Also note that add does not change the sequence of groups in the map unit in any way, it merely changes the contents of one group.

2.2. The insert Operation

The insert operation is very similar to the add operation. With add, a group with the proper ACS had to exist for a particular virtual fragment. Insert is like an add that *creates* a (empty) group where the fragment will be placed. Although neither example in §1.4 required this type of modification to the map unit, it will become clear later why such an operation is necessary. The psuedocode for insert is given in Figure 17.

```
add(vf,g,m)
    VIRTUAL_FRAGMENT        vf;
    GROUP                   g;
    MAP_UNIT                m;
{
    GROUP                   g_new;
    MAP_UNIT                m_new;

    if (acs(vf) = acs(g))
        then
            m_new ← m;
            g_new ← group_with_acs(acs(g),m_new);
            vfs_of(g_new) ← vfs_of(g_new) ∪ {vf};
            return(m_new);
        else
            return(FAIL);
    fi
}
```

Figure 16: Psuedocode for the GSA version of add

```
insert(vf,g_left,g_right,m)
    VIRTUAL_FRAGMENT        vf;
    GROUP                   g_left,g_right;
    MAP_UNIT                m;
{
    GROUP                   g_leftnew,g_ins;
    MAP_UNIT                m_new;

    if (acs(vf) ⊆ acs(g_left) ∪ acs(g_right) and acs(g_left) ∩ acs(g_right) ⊆ acs(vf))
        then
            m_new ← m;
            g_leftnew ← group_with_acs(acs(g_left),m_new);
            g_ins ← create_new_group_right_of(g_leftnew);
            vfs_of(g_ins) ← {vf};
            return(m_new);
        else
            return(FAIL);
    fi
}
```

Figure 17: Psuedocode for the GSA version of insert

Insert takes a virtual fragment vf and two consecutive groups g_{left} and g_{right} in a map unit m (it is assumed g_{left} is to the left of g_{right}) and attempts to create a new group between g_{left} and g_{right} with an ACS such that vf can be placed in the new group. (I.e., the ACS of the new group is ACS(vf).)

Just as one cannot add a specific virtual fragment to an arbitrary group, one cannot create a new group with a specific ACS at an arbitrary position in a map unit. The existing clone structure cannot be violated. In order to examine the situation in more detail, assume one wants to create a group g between g_{left} and g_{right}.

First, suppose there is a clone c that is an element of both ACS(g_{left}) and ACS(g_{right}). This means any virtual fragment in either group contains a real fragment from c. Then any virtual fragment in g must contain a real fragment from c. If the virtual fragment in g did not, then c would have two physically separate pieces. However, any clone is contiguous in the genome. Therefore, c must be an element of ACS(g). This restriction can be concisely expressed as the constraint (i_1) $\text{ACS}(g_{\text{left}}) \cap \text{ACS}(g_{\text{right}}) \subseteq \text{ACS}(g)$.

Now, suppose c is an element of ACS(g_{left}) but not of ACS(g_{right}). This means that the right end of c is somewhere to the right of the rightmost virtual fragment in g_{left}. (Note that it may not be known which virtual fragment in g_{left} is rightmost, and it does not matter.) Thus, when g is created, the question is whether or not the virtual fragments in g contain real fragments from c. Either situation is possible, because c might extend into g or it might not. Thus, it is valid for c to be a member of ACS(g), and it is valid for c not to be a member of ACS(g). A similar argument holds in the case where c is in ACS(g_{right}) but not in ACS(g_{left}). Thus, the ACS of g can contain any combination of the clones that are in the ACS of *only* one of the neighboring groups.

Finally, suppose c is not of a member of ACS(g_{left}) or ACS(g_{right}). Can c be a member of ACS(g)? First two facts should be mentioned. The first fact is that the GSA does not create or destroy real fragments; it merely rearranges the real fragments into a different configuration of virtual fragments. The second fact is that clones with less than two fragments are not used in mapping. (See §1.3.) If c is an element of ACS(g), then there will be exactly one real fragment from c in g. (This is because only one virtual fragment is to be placed in g.) Thus, there must be another real fragment from c somewhere else in the

map unit. However, this other real fragment is not in a virtual fragment in either g_{left} or g_{right}, since c is not in ACS(g_{left}), nor is it in ACS(g_{right}). Thus, it must be in some other group. This implies there are two real fragments from c which are non-contiguous. However, c must be contiguous in the genome. Therefore, c cannot be in ACS(g). These restrictions can be concisely expressed as the constraint (i_2) ACS(g) $\subseteq$ ACS(g_{left}) $\cup$ ACS(g_{right}).

If constraints i_1 and i_2 are satisfied, then a new copy m_{new} of the map unit m is created. The group in m_{new} corresponding to g_{left} is found and assigned the name g_{leftnew}. A new group g_{ins} is created immediately to the right of g_{leftnew}. Finally, vf is assigned to be the only fragment in g_{ins}, and m_{new} is returned. If i_1 and i_2 are not satisfied, then FAIL is returned to indicate that the insert operation did not succeed.

Note that, like add, insert is primitive. However, unlike add, insert does change the structure of the map unit it is given.

Insert can easily be extended to create a new group at either end of the map unit. If one perceives there to be a "phantom" group at both ends of the map unit that contains no virtual fragments and has an empty ACS, then the operation requires no modification.

For a particular virtual fragment, there may be more than one pair of consecutive groups where an insert could succeed. This is not contradictory with the previous claim concerning the uniqueness of a group where an add could succeed. It merely means that each place where an insert succeeds must be considered as a separate option.

2.3. The split Operation

A fragment matching mistake produces a virtual fragment that contains two or more real fragments that are incorrectly matched. An operation is needed that repairs the effects of such a mistake. Such an operation would decompose the virtual fragment into two or more virtual fragments where the real fragments that have been incorrectly matched are in separate virtual fragments. (Note that this occured in both examples in §1.4.) The split is this operation. Split takes a virtual fragment vf, a partition p (with more than one block) of the set of real fragments that compose vf, and a map unit m.

First, vf is taken out of m. The partition p determines how vf will be decomposed; each block of real fragments in p forms a new virtual fragment. These new virtual fragments will be called **virtual subfragments** (or just **subfragments**). Note that since there is a one-to-one correspondence between the real fragments of vf and the clones from which those real fragments come, using a partition of clones is equivalent to using a partition of real fragments. Split then must incorporate the subfragments of vf into m. Since no subfragment has the same ACS as vf, none of them belong in the group that contained vf. This is where the other three operations come into play.

Split must try to add, insert and combine each subfragment in all valid ways. Specifically, split should attempt to add each subfragment to any group in m. If all add operations for a particular subfragment fail, then split should attempt to insert the subfragment between any two groups in m. Along with attempting add or insert, split should attempt to combine each subfragment with all other virtual fragments in m. (The details of combine will be discussed in §2.4.) Obviously, split is a nonprimitive operation.

The split operation is successful if and only if a combination of the other three operations that successfully places each subfragment in the map unit can be found. For example, suppose split is given a partition that decomposes vf into virtual subfragments vf_1 and vf_2. If there is a group in m such that add is successful with vf_1 and a pair of groups such that insert is successful with vf_2, then the split is successful. Split will return a list of solution map units. There will be one solution for each such combination that is successful.

When the virtual fragment being split is contained in the map unit, the split is said to **occur** at the group containing that virtual fragment. A situation exists where the virtual fragment being split is not actually contained in the map unit, and will be discussed in §4.3.3.3. In that case, the place where the split occurs must be defined a little differently.

The psuedocode for the GSA version of split is not given. In the RSA, there will be a set of split operations. At that point, psuedocode will be given for each type of split operation.

2.4. The combine Operation

The split is intended to separate incorrectly matched real fragments into different virtual fragments, but the opposite operation may be appropriate also. (This is seen in the second example in §1.4.) There may be real fragments in different virtual fragments which really should be placed in the same virtual fragment. This is the function of the combine operation.

The combine operation takes two virtual fragments vf_1 and vf_2, and a map unit m. It attempts to make a new virtual fragment vf_3 which is composed of all the real fragments from vf_1 and all the real fragments from vf_2. This new virtual fragment is called a **combined virtual fragment** (or just **combined fragment**). However, one cannot always construct a valid combined virtual fragment from two arbitrary virtual fragments. There are three properties which vf_1 and vf_2 must satisfy to merge together and form a valid combined virtual fragment.

The first property concerns the clone compositions of vf_1 and vf_2. Remember that a virtual fragment is a collection of real fragments that are believed to come from the same section of the genome. Therefore, it is meaningless to have two real fragments from the same clone within the same virtual fragment. Thus, ACS(vf_1) and ACS(vf_2) must be disjoint.

The second property concerns the lengths of the real fragments composing vf_1 and vf_2. When one fuses clones with a map unit, virtual fragments are constructed one real fragment at a time. Any time a real fragment is added to an existing virtual fragment, the length of the real fragment must be the same (within some experimental error bounds) as the length of the virtual fragment. Thus, for vf_3 to be a valid virtual fragment, it must be ensured that it could have been built one real fragment at a time. This means there must be a way to order the real fragments of vf_3 such that selecting one at a time, in that order, allows one to reconstruct vf_3 using the same (error bound) matching criteria used in the building of m. It is easy to see that if such an ordering exists, then using the real fragments in ascending or descending order by length will successfully reconstruct vf_3 as well.

The third property is concerned with whether or not the combine is accomplishing its intended purpose. The reason for using the combine operation on vf_1 and vf_2 is that one believes there is some real fragment in vf_1 from some clone c_1 that should be matched with some real fragment in vf_2 from some clone c_2. If it can be deduced that there is no place in the map unit that a virtual fragment containing real fragments from c_1 and c_2 could be placed, then there is no reason to construct vf_3.

If there is a group with c_1 and c_2 in its ACS, then this group is a potential spot for a virtual fragment containing the matched real fragments. If there are two consecutive groups where one group contains c_1 in its ACS and the other group contains c_2 in its ACS, then a new group that is a potential spot could be created. Otherwise, there is no place in the map unit that any virtual fragment containing real fragments from c_1 and c_2 could be placed, and thus it is useless to construct vf_3.

However, when a combine is performed, it may not be known which real fragments are the ones that are intended to match. Thus, the above condition cannot be checked only with some specific c_1 and c_2, but must be checked with all c_1 in ACS(vf_1) and all c_2 in ACS(vf_2). If any such pair of clones can be found to satisfy the condition, then the combine should proceed.

For the sake of clarity, the three properties discussed above are summarized below.

(p_1) $\text{ACS}(vf_1) \cap \text{ACS}(vf_2) = \emptyset$.

(p_2) There is an order of the real fragments of vf_1 and vf_2 such that, together, they can be matched with the same criteria used during the construction of the original map unit.

(p_3) There must exist $c_1 \in \text{ACS}(vf_1)$ and $c_2 \in \text{ACS}(vf_2)$ such that either (1) there is a group g where $c_1, c_2 \in \text{ACS}(g)$ or (2) there are consecutive groups g_1 and g_2 where $c_1 \in \text{ACS}(g_1)$ and $c_2 \in \text{ACS}(g_2)$.

If either p_1, p_2 or p_3 is not satisfied, then combine returns FAIL, indicating that the combine did not succeed. If p_1, p_2 or p_3 are satisfied, then vf_1 and vf_2 are removed from m and vf_3 is created. Now, vf_3 must be incorporated into m.

Combine attempts to add vf_3 to each group in m. If an add succeeds, then the map returned by add is one solution. If all add operations fail, then combine attempts to insert vf_3 between each pair of two consecutive groups in m. Any map units returned by insert are solutions.

In addition to the add or insert, combine attempts to split vf_3 with all possible partitions of the real fragments of vf_3, except the partition that results in subfragments vf_1 and vf_2. Any maps returned by split are solutions. Combine collects all the solutions and returns them. If no solutions are found, then FAIL is returned.

The psuedocode for the GSA version of combine is not given. In the RSA, there will be a set of combine operations. At that point, psuedocode will be given for each type of combine operation.

3. The General Fragment Splitting Algorithm

3.1. A High Level Description

The four operations and a small amount of external control over the sequence of these operations form an algorithm that computes the map units that would have been built if a fragment matching mistake had not occured. If one suspects that a virtual fragment contains two or more incorrectly matched real fragments, all that needs to be done is to select that virtual fragment and the partition that decomposes the fragment as desired, and call the split operation. Split will use the other operations (which may in turn call split) to compute and return all possible solutions. If none are found, FAIL is returned.

However, it must be ensured that the call to split does not cause the operations to call each other in such a way that an infinite cycle of operations occurs. A sequence of operations could lead to a configuration of the current map unit and operation which appeared previously during that sequence. Thus, the sequence would then repeat, starting an infinite cycle of operations. Simply keeping a history of all the configurations encountered during the sequence of operations is one mechanism that can prevent infinite cycles. More efficient mechanisms may exist, but for now assume that an unspecified mechanism that prevents infinite cycles exists. Also, an external mechanism that prevents the fragment originally split from being placed back into the map unit is assumed to exist.

The four operations and the two external mechanisms described above form the General Fragment Splitting Algorithm. The GSA is not implemented in the DNA Mapping software. However, examining its functionality, advantages and disadvantages, serves as a basis for the discussion of the Restricted Fragment Splitting Algorithm.

3.2. Examples of the General Fragment Splitting Algorithm

The GSA has been described in a high level fashion, and thus it may not be clear exactly how it computes solutions. In this section, some examples of the application of the GSA are given with the hope that it will clarify the way it works.

3.2.1. Example One

The first example is the simplest example possible. Suppose Figure 18 represents the underlying reality. The horizontal and vertical lines have the same meanings as in Figure 8. The only difference is that in Figure 18, not all restriction sites are explicitly denoted. Only the sites which form the real fragments that are of interest are denoted. It is assumed that there is enough real fragment overlap between clones c_1 and c_2 to declare actual overlap. It is also assumed that the measured lengths of real fragments f_1 and f_2 are close enough to allow them to match and that no other real fragments in the two clones have lengths in that range.

Given these assumptions, the Map unit m constructed using the maximum matching is given in Figure 19. First, some of the conventions concerning the display of map units in this report should be mentioned. The ACS of a group may be given below the group. Only the virtual fragments of interest are explicitly denoted. Although not all virtual fragments are displayed, it is assumed there is at least one virtual fragment in each group.

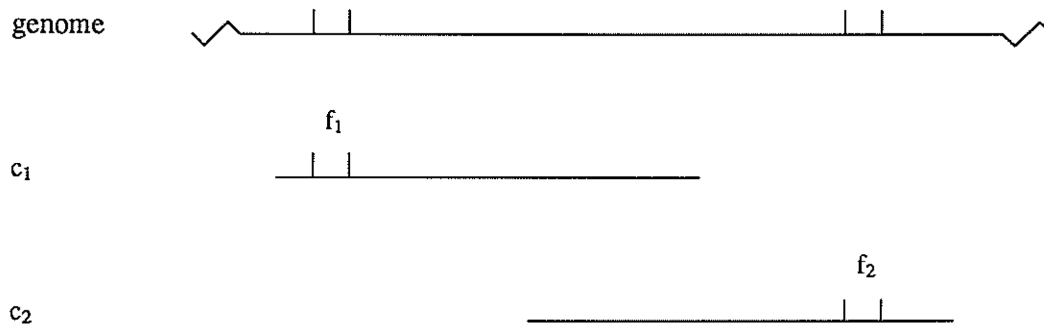


Figure 18: Clone configuration for Example one

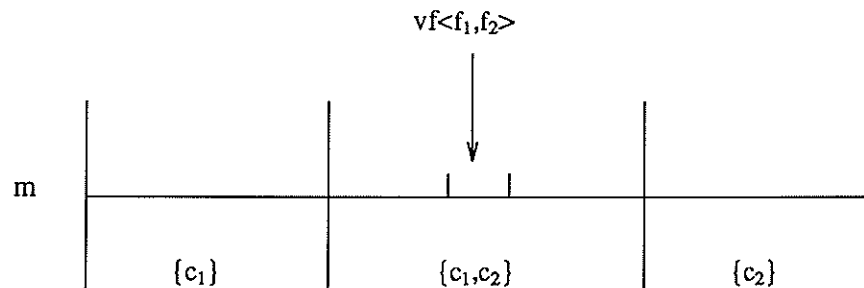


Figure 19: Map unit built from clones in Figure 18

Map unit m contains a fragment matching mistake because of the similarity in the lengths of f_1 and f_2. Suppose it is determined that $vf<f_1,f_2>$ should be split. There is only one partition of the set $\{f_1,f_2\}$ with more than one block, namely $\{\{f_1\},\{f_2\}\}$. Thus, our only option is to call split with parameters $vf<f_1,f_2>$, $\{\{f_1\},\{f_2\}\}$ and m.

The fragment $vf<f_1,f_2>$ is removed from m, and the subfragments $vf<f_1>$ and $vf<f_2>$ are created. Now both subfragments must be placed. First, an add with $vf<f_1>$ is attempted with each group. The add will succeed only with the group whose ACS is $\{c_1\}$. Second, an add with $vf<f_2>$ is attempted with each group. This add will succeed only with the group whose ACS is $\{c_2\}$. Thus one solution is computed, and is given in Figure 20.

Since the add operations were successful, the insert operation is not attempted. The combine operation still must be attempted. However, there are no other virtual fragments whose length is near enough to the length of $vf<f_1>$, and thus it does not successfully combine with any other virtual fragments. The same is true for $vf<f_2>$, and thus it does not successfully combine with any other virtual fragments. Therefore, there are no solutions other than the one shown in Figure 20. An examination of the map unit in Figure 20 reveals that it corresponds to the underlying reality in Figure 18.

3.2.2. Example Two

The second example is slightly more complicated and shows how the combine helps produce solutions. Suppose Figure 21 represents the underlying reality. This is basically the same as the previous example, except one more clone is involved. Assume that real fragments f_1, f_2 and f_3 are of similar length and that no other fragments in the three clones have lengths in that range. If one fuses Clones c_1 and c_2 first, and then Clone c_3, the Map unit m in Figure 22 is produced.

Suppose it is determined that $vf<f_1,f_2>$ should be split. Then $vf<f_1,f_2>$ is removed from m and subfragments $vf<f_1>$ and $vf<f_2>$ are created. The add operation is successful with both $vf<f_1>$ and $vf<f_2>$ just as in the previous example, and thus one solution is the map unit given in Figure 23.

However, in this example, not all combine operations fail. Consider attempting to combine $vf<f_1>$ and $vf<f_3>$. The conditions p_1, p_2 and p_3 must be checked. First, the ACS of $vf<f_1>$ is $\{c_1\}$, and the ACS of $vf<f_3>$ is $\{c_3\}$. These sets are disjoint, so p_1 is satisfied. Second, f_1 and f_3 match, and thus p_2 is satisfied. However, p_3 is not satisfied, because there is no group in m that contains c_1 and c_3 in its ACS, nor are there two consecutive groups where one contains c_1 in its ACS and the other contains c_3 in its ACS. Consequently, combine with $vf<f_1>$ and $vf<f_3>$ fails.

Now consider attempting to combine $vf<f_2>$ and $vf<f_3>$. It is easy to see that p_1 and p_2 are satisfied. This time, p_3 is satisfied because there is a group in m with an ACS of $\{c_2,c_3\}$. Thus,

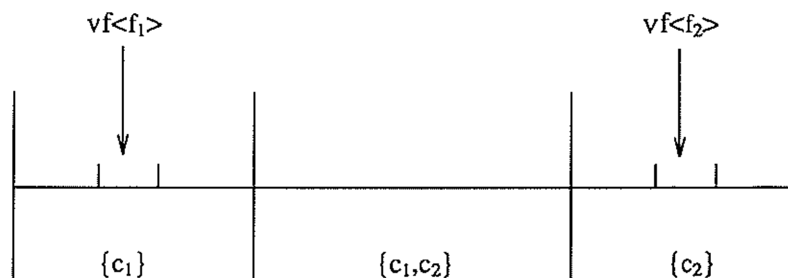


Figure 20: Map unit resulting from the split

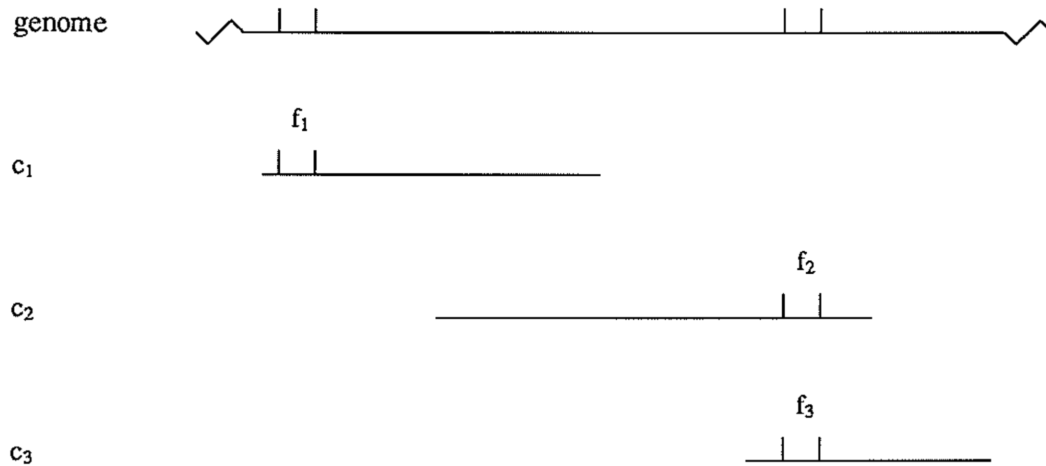


Figure 21: Clone configuration for Example two

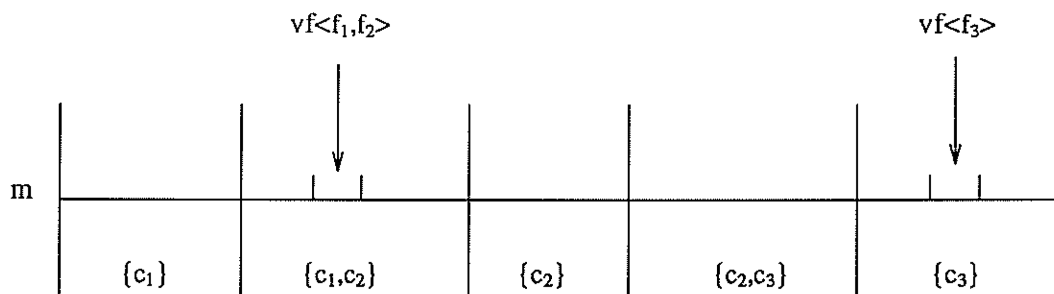


Figure 22: Map unit built from clones in Figure 21

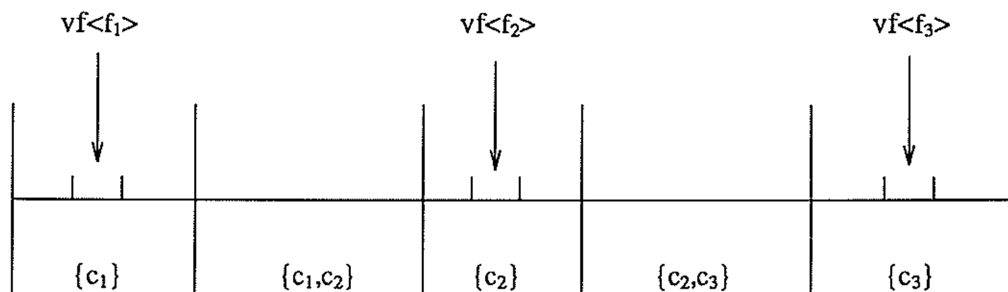


Figure 23: The first map unit resulting from the split

$vf<f_2,f_3>$ is formed and must be incorporated into m. Note that $vf<f_3>$ is taken out of the map unit, but that $vf<f_2>$ is not because it was never in the map unit to begin with.

The add operation succeeds with $vf<f_2,f_3>$ at the group with an ACS of $\{c_2,c_3\}$. So a second solution is found, and is given in Figure 24. Since the add operation was successful, insert is not attempted. Now combine calls split on $vf<f_2,f_3>$ using all possible partitions of $\{f_2,f_3\}$ containing more than one block. However, there is only one such partition, and it results in subfragments $vf<f_2>$ and $vf<f_3>$, the fragments that formed $vf<f_2,f_3>$. Thus, this partition is not used. So there are no partitions

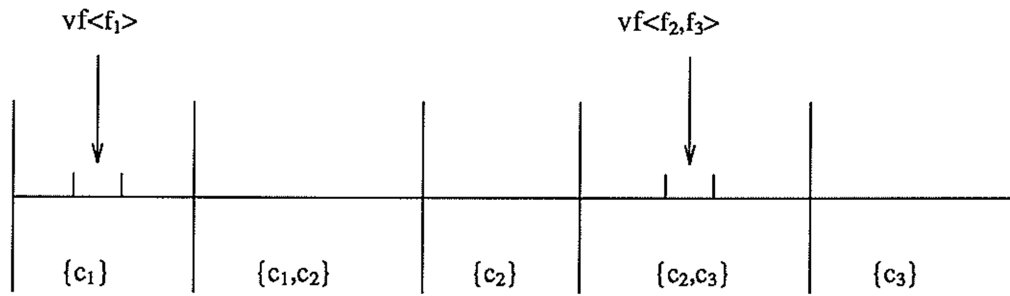


Figure 24: The second map unit resulting from the split

to try, and thus there are no more solutions to the original split.

Only one of the solutions (in this case, Figure 24) corresponds to the underlying reality in Figure 21. So why does the solution in Figure 23 exist? It exists because it corresponds to another configuration of restriction sites that is consistent with the original clone data. If Figure 25 represents the underlying reality, fusing the clones in the same order as before will produce the same map unit, i.e., Figure 22. Since Figure 23 corresponds to the underlying reality in Figure 25, Figure 23 is considered a valid solution. Note that there is less actual real fragment overlap in Figure 25 than in Figure 21. This will become important in the RSA. (See §4.1.1.)

3.2.3. Example Three

The third and final example is an extension of the second example. In this example, the need for the combine to call split becomes clear. Suppose Figure 26 represents the underlying reality. Assume real fragments f_1, f_2, f_3 and f_4 are of similar length and that no other real fragments in the four clones have lengths in that range. If Clones c_1 and c_2 are fused first, followed by c_3 and then finally c_4, the Map unit m in Figure 27 is produced. It is important to note that because f_2 was incorrectly matched with f_1, the real fragment f_3 was left in a position such that it was incorrectly matched with f_4. The first matching mistake

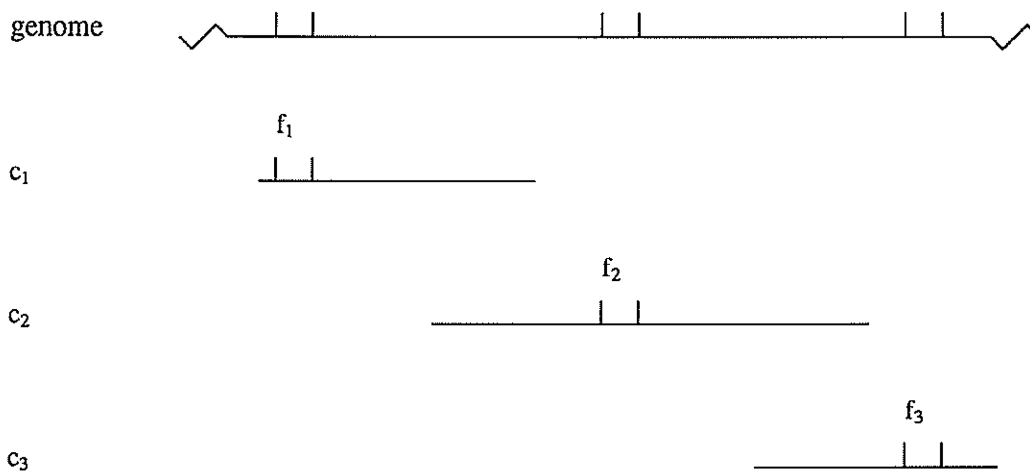


Figure 25: An alternate clone configuration for Example two

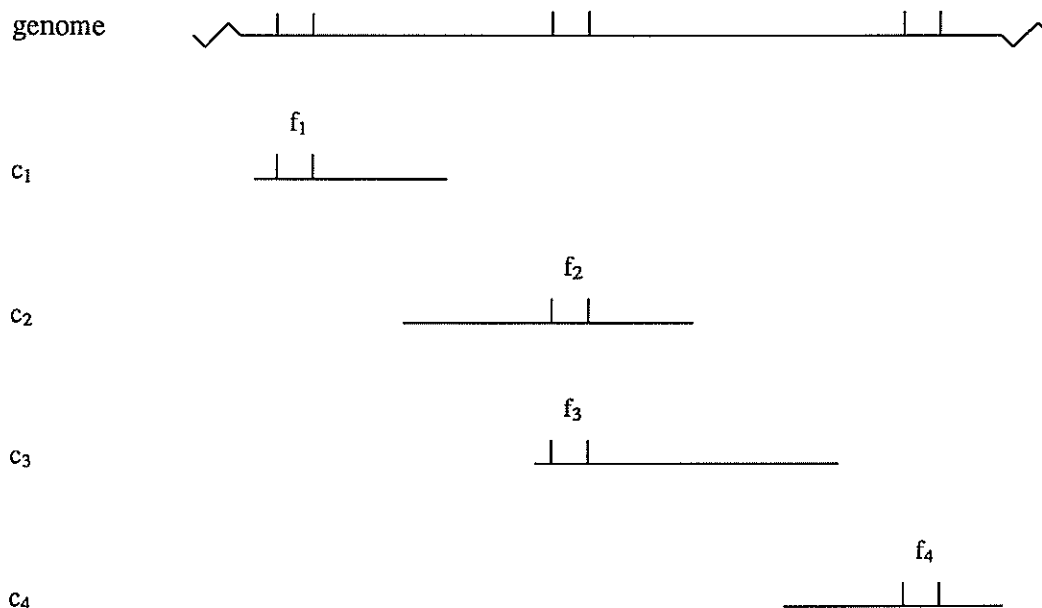


Figure 26: Clone configuration for Example three

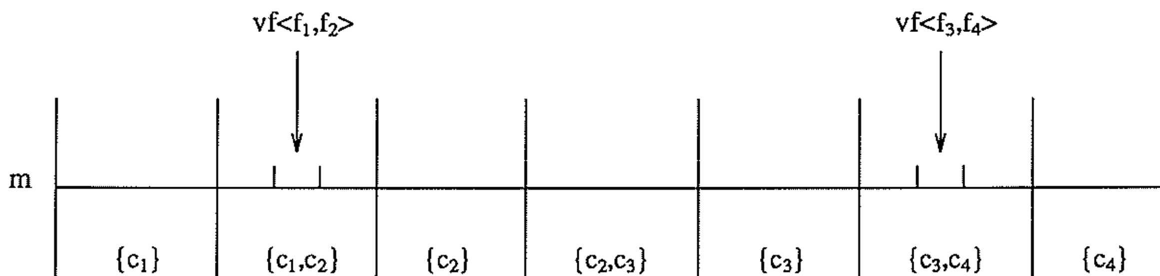


Figure 27: Map unit built from clones in Figure 26

caused another to occur during later fusion (i.e., this is not a case of two independent matching mistakes).

Suppose it is determined that $vf<f_1,f_2>$ should be split. There is only one partition of interest, namely $\{\{f_1\},\{f_2\}\}$. So subfragments $vf<f_1>$ and $vf<f_2>$ are created and must be placed. Like Example one, add succeeds with $vf<f_1>$ and $vf<f_2>$. The map unit that results is m_1 in Figure 28(a). However, there are more solutions to be found.

Split attempts to combine $vf<f_1>$ with some other virtual fragment. The only virtual fragment whose length is similar is $vf<f_3,f_4>$, but p_3 is not satisfied. Thus, all combine operations with $vf<f_1>$ fail.

Then split attempts to combine $vf<f_2>$ with some other virtual fragment. As with $vf<f_1>$, the only virtual fragment whose length is similar enough is $vf<f_3,f_4>$. However, there is a group whose ACS is $\{c_2,c_3\}$, and thus the combined virtual fragment $vf<f_2,f_3,f_4>$ is formed. Now something must be done with $vf<f_2,f_3,f_4>$.

There is no group with an ACS of $\{c_2,c_3,c_4\}$, and thus any add with $vf<f_2,f_3,f_4>$ fails. There are not two consecutive groups between which a group with an ACS of $\{c_2,c_3,c_4\}$ can be created, so any insert with $vf<f_2,f_3,f_4>$ fails. Then $vf<f_2,f_3,f_4>$ is split using all partitions of $\{f_2,f_3,f_4\}$ containing more than one block. There are three such partitions: $\{\{f_2\},\{f_3,f_4\}\}$, $\{\{f_2,f_3\},\{f_4\}\}$ and $\{\{f_2\},\{f_3\},\{f_4\}\}$.

The partition $\{\{f_2\},\{f_3,f_4\}\}$ recreates the virtual fragments that originally created $vf<f_2,f_3,f_4>$, and thus is not used. The partition $\{\{f_2,f_3\},\{f_4\}\}$ creates subfragments $vf<f_2,f_3>$ and $vf<f_4>$. The add operation succeeds with both of these fragments to produce the Map unit m_2 in Figure 28(b). (Now, one can see how the combine operation's call to split allows more solutions to be found.) Now that the add operations are complete, combine operations are attempted. The only virtual fragment that could combine with $vf<f_2,f_3>$ is $vf<f_1>$. Conditions p_1, p_2 and p_3 are satisfied, and thus $vf<f_1,f_2,f_3>$ is formed. However, $vf<f_1,f_2,f_3>$ does not successfully add or insert, and the only way to split it causes $vf<f_1,f_2>$ to be placed back into the map unit. Consequently, this combine fails. The only virtual fragment that might combine with $vf<f_4>$ is $vf<f_1>$, but p_3 is not satisfied. Thus, no combine operations with $vf<f_2,f_3>$ or $vf<f_4>$ lead to solutions. This means that all solutions produced as a result of the split with the partition $\{\{f_2,f_3\},\{f_4\}\}$ have been found.

The last partition to use with split is $\{\{f_2\},\{f_3\},\{f_4\}\}$, which creates subfragments $vf<f_2>$, $vf<f_3>$ and $vf<f_4>$. The add operation succeeds with each subfragment to produce the Map unit m_3 given in Figure 28(c). The only virtual fragment that $vf<f_2>$, $vf<f_3>$ or $vf<f_4>$ can combine with is $vf<f_1>$. When using fragments $vf<f_3>$ and $vf<f_4>$, p_3 is not satisfied, although when using $vf<f_2>$, it is. However, a combine with $vf<f_1>$ and $vf<f_2>$ causes $vf<f_1,f_2>$ (the fragment that was originally split) to be placed back into the map unit. Consequently, none of these combine operations result in solutions.

There are no more partitions of $\{f_2,f_3,f_4\}$ to use, and so all solutions have been found as shown in Figure 28. All the solutions in Figure 28 are valid because each corresponds to an underlying reality that could have produced the initial map unit.

3.3. Problems with the General Fragment Splitting Algorithm

As a practical tool for helping a mapper fix common mistakes easily and quickly, the GSA leaves much to be desired. There are difficulties concerning (1) run time, (2) the number of solutions obtained, (3) preventing the undoing of work, (4) preventing infinite sequences of operations, and (5) finding solutions multiple times.

First, it should be apparent that the GSA is tremendously time-consuming. If a combine must call a split with a virtual fragment vf that is composed of a large number of real fragments, the number of calls to split can be very large because the number of partitions of the real fragments of vf is very large. In addition, combine may be called many times in a long map unit that contains many fragments of roughly equal length. To make matters worse, it is the real fragments of similar length that occur repeatedly in the same area of the genome that are most likely to be involved in incorrect matches.

Second, the number of solutions generated can be very large. When the map unit is long, there are potentially many virtual fragments that can participate in a combine, and add and insert operations are more likely to succeed. Many of the solutions that are returned correspond to underlying realities that are not likely. (Although all indeed are possible.) It would be burdensome to the mapper to have to sort through all the solutions to pick out one that appears appropriate.

Third, the undoing of the effects of operations occurs. For example, suppose a combine is called with virtual fragments vf_1 and vf_2 to form vf_3. The reason vf_1 and vf_2 are combined is that it is believed that a real fragment from vf_1 should be matched with a real fragment from vf_2. If subsequent calls to split break up the real fragments in vf_3 so that all real fragments originally in vf_1 are separated from all the real fragments originally in vf_2, then the effect of the combine is completely gone. Essentially, it means the

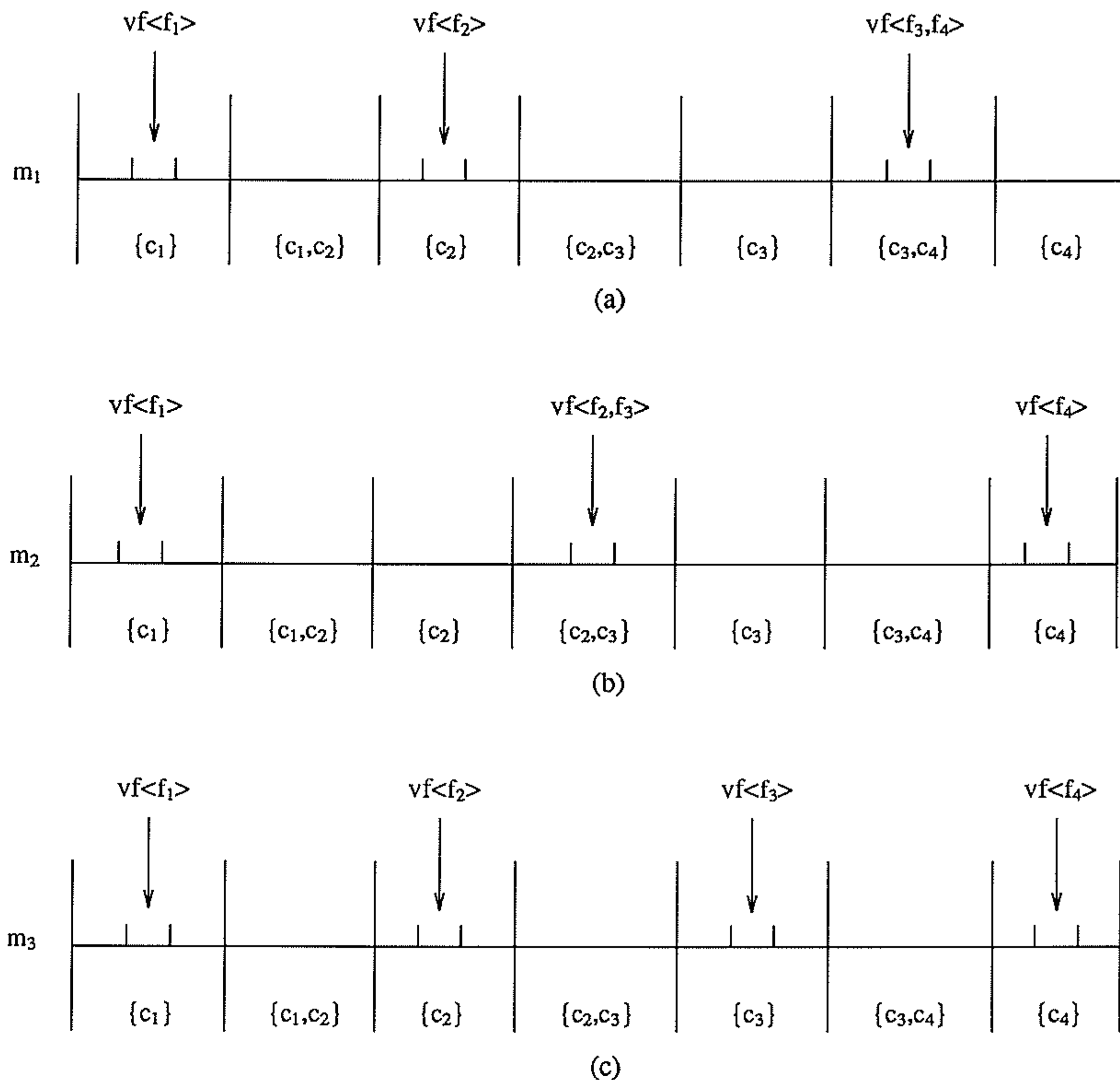


Figure 28: Map units resulting from the split

combine was useless. If this occurs often, the GSA will waste an enormous amount of time.

Fourth, as mentioned earlier, an infinite sequence of operations can occur if steps to prevent it are not taken. It was assumed that a mechanism existed to detect and terminate such sequences. Indeed a mechanism exists, but it is very time and space consuming. It is likely that any such scheme will either be very time and space consuming, or it will require very complex coding.

Fifth, without outside intervention, it is possible that a particular solution could be found more than once. In a way, this is related to the infinite sequence problem, because there needs to be some mechanism which can recognize a state that has been encountered before. It may be the case that both problems can be solved using the same mechanism. In any case, this multiplicity of solutions is a problem to be dealt with which will result in either space/time consumption or complex coding.

4. The Restricted Fragment Splitting Algorithm

In order to overcome some of the drawbacks of the GSA, a new algorithm that searches for solutions in a more restricted way is needed. This new algorithm is the Restricted Fragment Splitting Algorithm, or RSA. The RSA will not be as time and space consuming as the GSA. In addition, the RSA will be easier to implement. Naturally, the RSA will not return all possible solutions. The intent is to use the concepts of directionality (where one side of the map unit is treated differently than the other) and operation bias (where certain operations are given precedence over others) to find only those solutions which are "desirable" in some respect.

§4.1 discusses the characteristics that make one solution more desirable than another. §4.2, presents some general characteristics of RSA operations. The RSA operations are discussed in great detail in §4.3. The RSA, as a whole, is discussed briefly in §4.4. In §4.5, §4.6, and §4.7, examples of the usage of the RSA are presented. Finally, §4.8 considers some implementation issues.

4.1. Desirable Map Unit Characteristics

4.1.1. Compactness

During the discussion about the construction of map units (see §1.2), it was stated that more compact map units are preferred. The philosophy underlying this preference is the following: If there are two real fragments that can be matched, then it is believed that they really are from the same section of the genome unless there is evidence to the contrary. That is, one uses the maximum amount of overlap possible. The more overlap that is used, the more compact the resulting map unit.

In fragment splitting, the preference for compact map units still exists. This may seem inappropriate, since it is this preference that creates the fragment matching mistake in the first place. However, the initial call to split repairs the fragment matching mistake, and thus anything that occurs subsequently should use the compactness philosophy.

4.1.2. Single Mistake Fixing

In order to reduce the scope of the problem that the algorithm has to solve, a philosophy of **single mistake fixing** is used. This means that the RSA is designed to repair the effects of only one fragment matching mistake. For example, suppose Figure 29 represents the underlying reality. Assume fragments f_1, f_2, f_3 and f_4 are of similar length, and no other fragments in the four clones have lengths in that range. The map unit that results from fusing Clones c_1 and c_2 first, followed by Clone c_3 and then Clone c_4, is given in Figure 27. The important thing to notice is that two completely independent fragment matching mistakes occur. Fragment f_1 incorrectly matches with f_2, and fragment f_3 incorrectly matches with f_4. However, neither mistake has any effect on the other. It is not the case that the first mistake leads to a situation that produces the second. The RSA is not intended to repair the effects of both mistakes in one application. Thus, the RSA is not expected to find the correct map in this situation. (Whereas the GSA would obtain the correct map unit.)

4.2. New Characteristics of the Basic Operations

The primary difference between the GSA and the RSA is in the definition of the operations. The differences can be divided into three categories: The introduction of directionality, the introduction of operation bias and the restriction to bipartite splits.

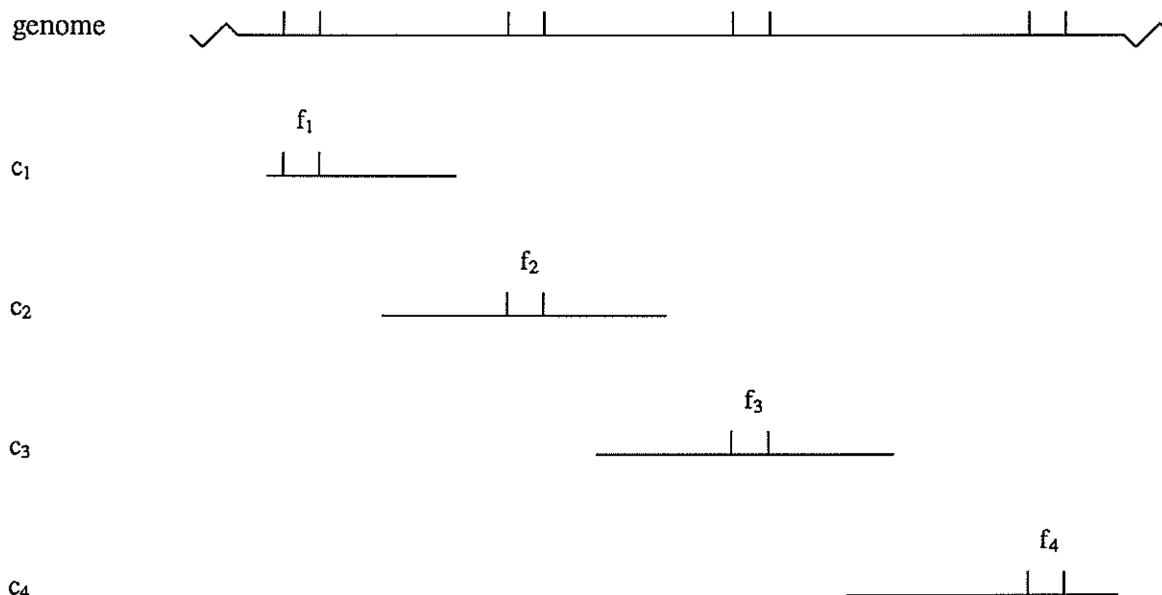


Figure 29: A possible clone configuration

4.2.1. Directionality

The operations in the GSA can be described as lacking in direction. For example, a virtual fragment can combine with another virtual fragment just about anywhere in the map unit. This combined fragment can then be used by an add or insert any place in the map unit where the appropriate ACS structure exists. Furthermore, the virtual subfragments resulting from a split can combine, add or insert just about anywhere in the map unit.

It is this lack of restrictiveness (among other things) that allows all solutions to be obtained. However, it also makes the GSA difficult to control and consumes a lot of time. In the RSA, some of the operations will have a *direction* associated with them. This will restrict the way in which these operations call other operations, and it will restrict the region of the map unit to which an operation is applicable. For example, a split that has direction will call certain operations that apply to the portion of the map unit to the left of where the split occurs, but it will call slightly different operations to apply to the portion of the map unit to the right of where the split occurs.

It will become clear that the incorporation of directionality into the operations makes controlling the sequence of RSA operations easier and aides in solving the previously discussed problems with infinite sequences of operations, preventing the undoing of previous work and the multiplicity of solutions.

4.2.2. Operation Bias

In the GSA, all of the operations are roughly of equal importance. That is, there is no preference to a solution based on the sequence of operations that was used to compute that solution. (Since *all* solutions computed are used.)

In the RSA, a bias toward particular operations is introduced. For example, suppose a subfragment resulting from some split can add successfully to a group and can combine (in some form) successfully with another virtual fragment. The solution generated by the combine will be preferred over the solution generated by the add. The combine will be attempted first, and if it results in a solution, the add operation

will not even be attempted. Thus, there is a bias toward combine (over add and insert) when trying to place subfragments of a split. The bias is toward the combine because more compact solutions will be produced.

4.2.3. Bipartite Splits

In the GSA, a virtual fragment *vf* can be split into two *or more* virtual subfragments. The GSA consumes a tremendous amount of time when it must split *vf* using *all* possible partitions of more than one block, because the number of partitions to use increases exponentially as the number of real fragments in *vf* increases. To make matters worse, *vf* is usually a combined virtual fragment, which typically contains more real fragments than most virtual fragments actually in the map unit.

In order to reduce the run time, split operations will be restricted to breaking virtual fragments into *two* subfragments. That is, *vf* will be decomposed with a *bipartition* of the set of real fragments composing it. Even though this will greatly reduce the number of split operations required, the number of bipartitions also increases exponentially as the number of real fragments in *vf* increases.

The intuitive reason for using only bipartitions is as follows. There is some sequence of operations which occurs to the left of where the split occurs to produce a solution. These operations require certain real fragments from *vf*. It should not matter whether those real fragments are grouped together in a single virtual fragment or as several virtual fragments. The subsequent operations should manipulate (using split, combine, etc.) those real fragments as needed to produce a solution. A similar statement can be made about the sequence of operations occuring to the right of where the split occurs. Thus, it is adequate to split *vf* into two subfragments. One of the subfragments will start the sequence of operations to the left of where the split occurs, and the other subfragment will start the sequence of operations to the right of where the split occurs.

Unfortunately, this intuition is not exactly correct. There are solutions that will not be found if only bipartitions are used, but it is believed that these solutions represent more complex mistakes than the kind that the RSA is intended to resolve. Suppose Figure 30 represents the underlying reality. It is assumed that real fragments f_1, f_2 and f_3 are of similar length and no other fragments in the three clones have lengths in that range. The map unit constructed by fusing Clones c_1 and c_2 first, followed by c_3 is given in Figure 31.

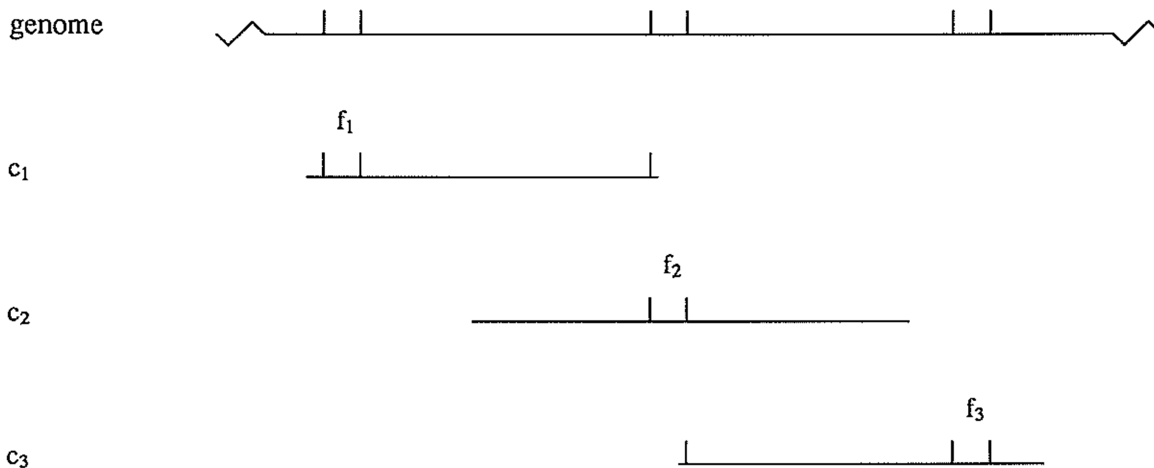


Figure 30: A possible clone configuration

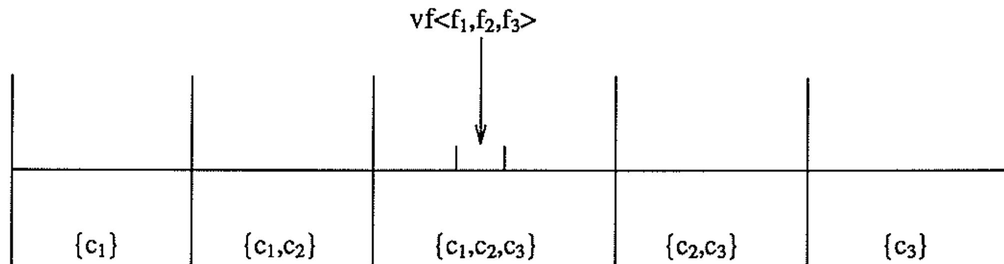


Figure 31: Map unit built from clones in Figure 30

First, note that $vf<f_1,f_2,f_3>$ is the only virtual fragment in the group with an ACS of $\{c_1,c_2,c_3\}$. If $vf<f_1,f_2,f_3>$ is split with the partition $\{\{f_1\},\{f_2\},\{f_3\}\}$, the add operation succeeds with the subfragments $vf<f_1>$ and $vf<f_3>$, and the insert operation succeeds with $vf<f_2>$. The resulting map unit then corresponds to the underlying reality.

The problem is that there is no *bipartition* such that one split of $vf<f_1,f_2,f_3>$ results in the correct map unit. However, it took more than just one simple fragment matching mistake to arrive at this state. There are *three* separate real fragments, roughly one clone length apart, which result in *two* independent matching mistakes. First f_1 and f_2 are incorrectly matched, and then f_3 is incorrectly matched with $vf<f_1,f_2>$. These mistakes are independent of each other, because even if f_1 and f_2 had not been incorrectly matched, f_3 would have incorrectly matched with f_2. In §4.2.2, it was stated that the RSA is not intended to correctly repair map units containing more than one fragment matching mistake. Thus, it is of no concern that the correct map unit is not obtained when using only bipartitions, in this particular situation.

In this case, it is possible to obtain the correct map unit using only bipartitions, but it requires that a second split operation be called on the result of the first. (In addition, the second must be called with slightly different parameters than the first.)

Another reason to use only bipartitions is that it conforms with the philosophy of favoring more compact map units. The number of subfragments created is minimized, and consequently the solutions produced will tend to be more compact.

4.3. Basic Operations in the Restricted Fragment Splitting Algorithm

Now the operations that exist in the RSA will be discussed in more detail. Hopefully, it will become apparent how the ideas in §4.2 are incorporated into the definitions of the operations. The operations will be described in a conceptual manner. The psuedocode is written to describe the effects the operations have on the map unit while minimizing the presentation of implementation details. There are some significant differences between the psuedocode to be presented and the actual implementation of these operations in the DNA Mapping software. (Some of these differences are discussed in §4.7.) However, the basic idea behind each operation is specified by the psuedocode.

4.3.1. The add Operation

The add operation in the RSA is not much different than the add operation in the GSA. However, it will prove useful to think differently about the way in which it is called. The psuedocode for add is given in Figure 32. As input, the RSA add takes a virtual fragment vf, two groups g_{left} and g_{right}, and a map unit m. (g_{left} is assumed to be to the left of g_{right}.) If there is a group between g_{left} and g_{right} (not including either) with the proper ACS, then vf is placed in that group and the new map unit is returned. If no such

group is found, then FAIL is returned. The parameter g_{left} could be a special token representing the left end of the map, and g_{right} could be a special token representing the right end of the map.

4.3.2. The insert Operation

The RSA version of the insert operation is similar to the GSA version. The same rules for creating a new group as in the GSA apply in the RSA. As with add, it is useful to view insert as taking a range of groups to search for a location to create the new group. The psuedocode for insert is given in Figure 33. In this, as in later psuedocode, [] denotes the empty list, [o] denotes the list containing the object o, and || is the list concatenation operator. As input, the RSA insert takes a virtual fragment vf, two groups g_{left} and g_{right}, and a map unit m. (g_{left} is assumed to be to the left of g_{right}.) The insert operation searches for two consecutive groups (between g_{left} and g_{right}) between which a group with an ACS of ACS(vf) can be created. Note that the new group can be created directly to the right of g_{left} or directly to the left of g_{right}.

Recall that in §2.2, it was noted that there could be more than one pair of consecutive groups in a map unit between which a particular virtual fragment could be inserted. So insert returns either a *list* of map units or the FAIL token.

```
add(vf,g_left,g_right,m)
    VIRTUAL_FRAGMENT        vf;
    GROUP                   g_left,g_right;
    MAP_UNIT                m;
{
    GROUP                   g,g_new;
    MAP_UNIT                m_new;

    g ← group_right_of(g_left);

    while (g ≠ g_right) do
        if (acs(g) = acs(vf))
            then
                m_new ← m;
                g_new ← group_with_acs(acs(g),m_new);
                vfs_of(g_new) ← vfs_of(g_new) ∪ {vf};
                return(m_new);
            else
                g ← group_right_of(g);
        fi
    od

    return(FAIL);
}
```

Figure 32: Psuedocode for the RSA version of add

```
insert(vf,g_left,g_right,m)
    VIRTUAL_FRAGMENT        vf;
    GROUP                   g_left,g_right;
    MAP_UNIT                m;
{
    GROUP                   g1,g2,g_new1,g_ins;
    MAP_UNIT                m_new;
    LIST                    solutions;

    solutions ← [];
    g1 ← g_left;

    while (g1 ≠ g_right) do
        g2 ← group_right_of(g1);
        if (acs(vf) ⊆ acs(g1) ∪ acs(g2) and acs(g1) ∩ acs(g2) ⊆ acs(vf))
            then
                m_new ← m;
                g_new1 ← group_with_acs(acs(g1),m_new);
                g_ins ← create_new_group_right_of(g_new1);
                vfs_of(g_ins) ← {vf};
                solutions ← solutions || [m_new];
        fi
        g1 ← group_right_of(g1);
    od

    if (solutions = [])
        then return(FAIL);
        else return(solutions);
    fi
}
```

Figure 33: Psuedocode for the RSA version of insert

4.3.3. The Split Operations

The concepts of directionality and bias complicate the split operation in the RSA. The result is that, in the RSA, there are actually four different split operations: undirected_split, directed_split, internal_split and top_level_split. All RSA split operations have a bias toward combining over add and insert, and directionality exists to a varying degree.

4.3.3.1. The undirected_split Operation

Of the four types of splits, the undirected_split is most similar to the split operation in the GSA. However, the concepts of bias and directionality cause undirected_split to be slightly different from split.

The psuedocode for undirected_split is given in Figure 36. In the psuedocode, (x, y) denotes the pair with left element x and right element y. As input, undirected_split takes a virtual fragment vf, the group g in which vf is located, an ordered pair bp and a map unit m. The ordered pair bp represents a bipartition of the set of real fragments that compose vf. The left element of bp is a set of real fragments that form one subfragment, and the right element is a set of real fragments that form the other subfragment.

Using *bp*, *vf* is decomposed into two subfragments $subvf_{left}$ and $subvf_{right}$. The real fragments in the left element of *bp* form $subvf_{left}$, and the real fragments in the right element of *bp* form $subvf_{right}$. The fragment $subvf_{left}$ is called the **left virtual subfragment** (or just **left subfragment**), and $subvf_{right}$ is called the **right virtual subfragment** (or just **right subfragment**). (Because undirected_split does not attempt swapping the elements of *bp*, the top_level_split operation has this responsibility.)

A major problem with the GSA is that a sequence of operations may jump all over the map unit, and thus it is difficult to predict or control where subsequent operations are going to occur. In the RSA, limitations are placed upon where operations can occur. For instance, any operation using $subvf_{left}$ must occur to the left of *g*. Furthermore, the sequence of operations triggered by $subvf_{left}$ must move progressively toward the left end of *m*. Thus, these operations will have *left* directionality. Similarly, any operation using $subvf_{right}$ must occur to the right of *g*, and operations triggered by $subvf_{right}$ must move toward the right end of *m*, and thus these operations will have *right* directionality. These concepts are illustrated in Figure 34.

In particular, what happens to the left subfragment? In the GSA, split attempts to combine and add (or insert) the subfragment. All solutions that result are returned. However, with a bias toward combine, if a solution can be found using a combine, then the add (or insert) operation is not attempted. This tends to produce more compact solutions.

First, *m* is searched for virtual fragments that satisfy the three conditions for combining in the RSA (p_1', p_2' and p_3') with $subvf_{left}$ and that are to the left of *g*. (In the RSA, the conditions for combining are slightly different from those in the GSA. This is discussed in more detail in §4.3.4.) This means the left subfragment is not allowed to combine with *any* virtual fragments in or to the right of *g*. When the list of fragments that satisfy the conditions is obtained, a directed_combine with left directionality (a **left** directed_combine) is attempted with each member of this list. A left directed_combine is used because all operations triggered by $subvf_{left}$ must have left directionality. If any of the left directed_combine operations return map units, then these map units are placed in the list $maps_{left}$, and attention is focused on the right subfragment.

If all left directed_combine operations return FAIL, then an add with $subvf_{left}$ is attempted over the range of groups to the left of *g*. If the add fails, then an insert is attempted over the range of groups to the left of *g*. Any map units that are returned by these operations are placed in $maps_{left}$.

If directed_combine, add or insert could not successfully place the left subfragment, then undirected_split returns FAIL. Otherwise, undirected_split attempts to place the right subfragment.

When the left subfragment is placed successfully, a similar sequence of operations is attempted with the right subfragment. The left and right subfragments are treated similarly because the undirected_split has no direction. First, *m* is searched for virtual fragments that, with $subvf_{right}$, satisfy p_1', p_2' and p_3'.

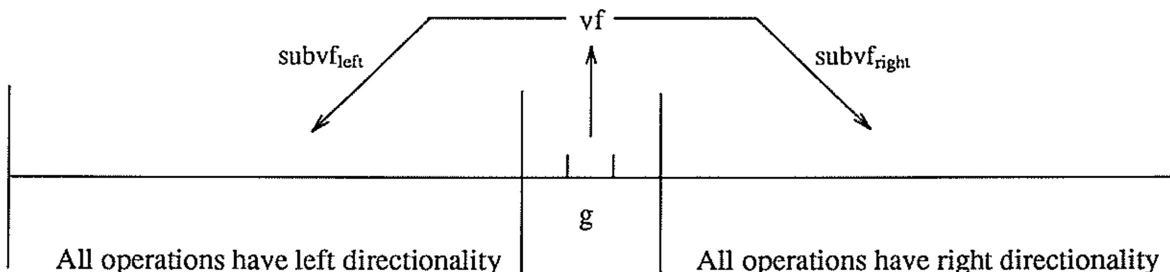


Figure 34: Subfragment placement and directionality constraints

Then a directed_combine operation with right directionality (a **right** directed_combine) is attempted with each. If none are successful, then an add with $subvf_{\text{right}}$ over the region of m to the right of g is attempted. If the add is unsuccessful, then an insert to the region of m to the right of g is attempted. Any map units returned by these operations are placed in $maps_{\text{right}}$. If none are returned, then undirected_split returns FAIL.

Assume that both the left and right subfragments are successfully placed. The elements of the list $maps_{\text{left}}$ are the map units that result from placing the left subfragment, and the elements of the list $maps_{\text{right}}$ are the map units that result from placing the right subfragment. However, a true solution to the undirected_split has both the left and right subfragment placements. Thus, the elements of $maps_{\text{left}}$ and $maps_{\text{right}}$ must be merged in some way.

Suppose m_{left} is an element of $maps_{\text{left}}$ and m_{right} is an element of $maps_{\text{right}}$. Due to the nature of the RSA operations, the placement of the left subfragment affects only groups to the left of g. Likewise, the placement of the right subfragment affects only groups to the right of g. Thus, any placement of the left subfragment is independent of any placement of the right subfragment. So one can take the portion of m_{left} to the left of and including g , and remove vf. Then, take the result of this and the portion of m_{right} to the right of g , and "concatenate" them to form a valid solution, m_{new}, to the undirected_split. This process is illustrated in Figure 35. The fragment vf is not removed before the call to directed_combine because its location serves as a convenient marker in the directed_combine.

If this process is performed using all possible pairings of an element in $maps_{\text{left}}$ with an element in $maps_{\text{right}}$, then all possible solutions to the undirected_split are generated. Thus, if there are l_1 elements of $maps_{\text{left}}$ and l_2 elements of $maps_{\text{right}}$, then there are $l_1 l_2$ solutions to the undirected_split.

4.3.3.2. The directed_split Operation

The second type of split operation in the RSA is the directed_split operation. A directed_split has either left of right directionality. (It is said the operation is a **right** directed_split or a **left** directed_split.) The effect that direction has upon the behavior of an operation should now become clear.

Like the undirected_split, directed_split creates a left and right subfragment. The undirected_split operation treats the left and right subfragments similarly. This is not the case with the directed_split. The subfragment with the opposite directionality of the operation, called the **opposite direction subfragment,** is not allowed to participate in any type of combine operation. (For example, the left subfragment in a right directed_split is the opposite direction subfragment.) So the opposite direction subfragment must add or insert in order to be placed. The subfragment with the same directionality as the operation, called the **same direction subfragment**, can be placed with a directed_combine, add or insert, just like the subfragments in undirected_split.

Because of this restriction, there is more control over the sequence of operations than existed in the GSA. It was stated earlier that an operation with left directionality causes subsequent operations to move progressively toward the left end of the map unit. This is accomplished by the left directed_split because it is known that only one operation (a single add or insert) occurs to the right of the group where the left directed_split occurs. Thus, the only direction a sequence of more than one operation could lead is toward the left end of the map unit. Similarly, it is known that only one operation occurs to the left of the group where a right directed_split occurs. Thus, the only direction a sequence of more than one operation could lead is toward the right end of the map unit.

The psuedocode for the directed_split is similar to the psuedocode for undirected_split, and it is given in Figure 37.

Note that the convention is that the real fragments in the left element of bp form the same direction subfragment, and the real fragments in the right element form the opposite direction subfragment.

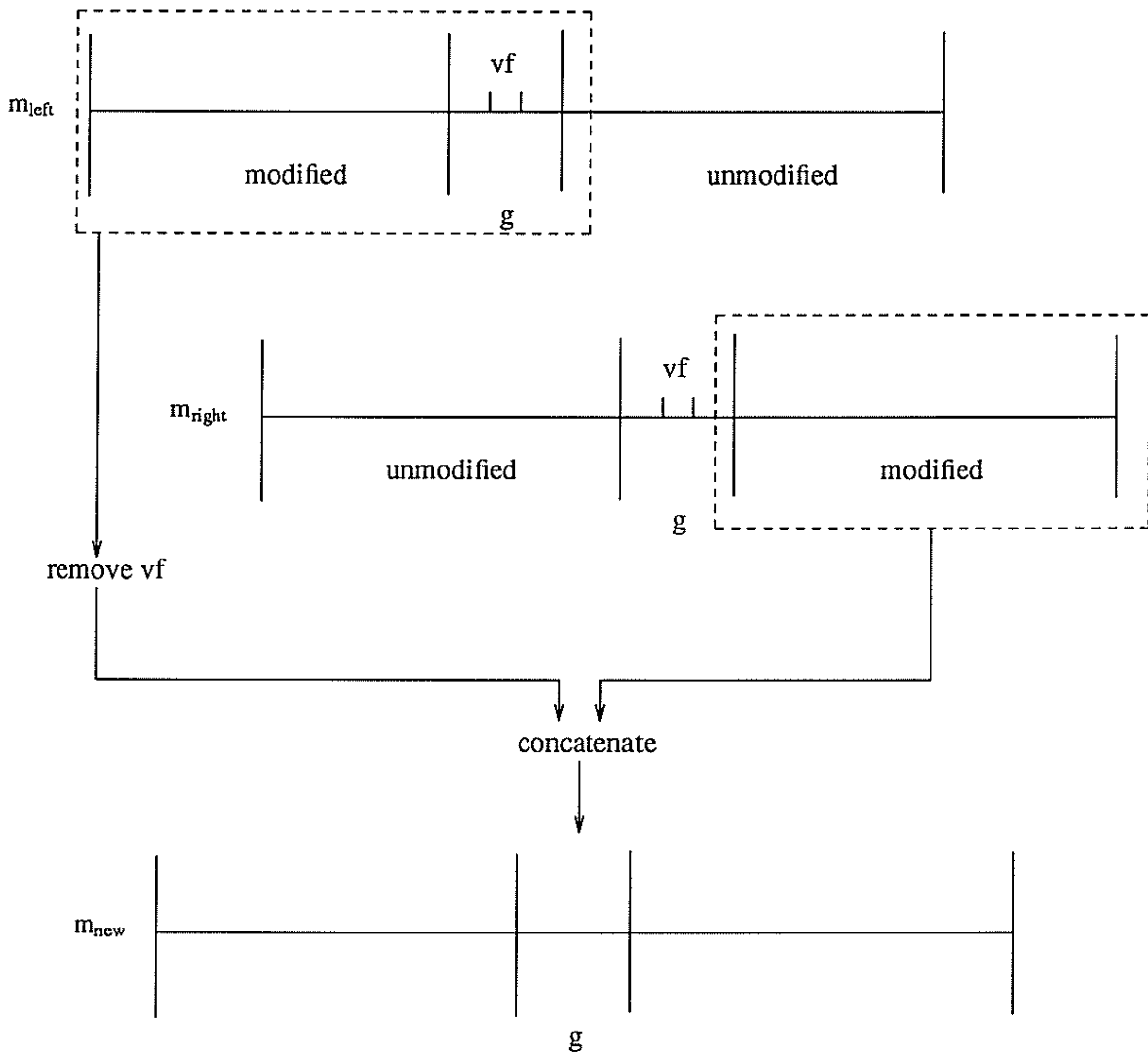


Figure 35: Construction of a solution map unit in undirected_split

```
undirected_split(vf,g,bp,m)
    VIRTUAL_FRAGMENT                vf;
    GROUP                           g;
    PAIR                            bp;
    MAP_UNIT                        m;
{
    VIRTUAL_FRAGMENT                subvf_left,subvf_right,vf_comb;
    MAP_UNIT                        m_add;
    LIST                            maps,maps_left,maps_right,maps_comb,vfs_comb;

    (subvf_left,subvf_right) ← vf_split(vf,bp);
    maps_left ← [];
    vfs_comb ← combining_vfs(subvf_left,LEFT_END,g);

    for vf_comb ∈ vfs_comb do
        maps_comb ← directed_combine(subvf_left,vf_comb,g,group_of(vf_comb),LEFT,m);
        if (maps_comb ≠ FAIL) then maps_left ← maps_left || maps_comb; fi
    rof

    if (maps_left = [])
        then
            m_add ← add(subvf_left,LEFT_END,g,m);
            if (m_add = FAIL)
                then maps_left ← insert(subvf_left,LEFT_END,g,m);
                else maps_left ← [m_add];
            fi
    fi

    if (maps_left = FAIL)
        then return(FAIL);
        else
            maps_right ← [];
            vfs_comb ← combining_vfs(subvf_right,g,RIGHT_END);

            for vf_comb ∈ vfs_comb do
                maps_comb ← directed_combine(subvf_right,vf_comb,g,group_of(vf_comb),RIGHT,m);
                if (maps_comb ≠ FAIL) then maps_right ← maps_right || maps_comb; fi
            rof

            if (maps_right = [])
                then
                    m_add ← add(subvf_right,g,RIGHT_END,m);
                    if (m_add = FAIL)
                        then maps_right ← insert(subvf_right,g,RIGHT_END,m);
                        else maps_right ← [m_add];
                    fi

                    if (maps_right = FAIL) then return(FAIL); fi
            fi

            maps ← concatenate_left_and_right_parts_of_maps(maps_left,maps_right,vf,g);

            return(maps);
    fi
}
```

Figure 36: Psuedocode for undirected_split

```
directed_split(vf,g,bp,dir,m)
    VIRTUAL_FRAGMENT                vf;
    GROUP                           g;
    PAIR                            bp;
    DIRECTION                       dir;
    MAP_UNIT                        m;
{
    VIRTUAL_FRAGMENT                subvf_opp,subvf_same,vf_comb;
    MAP_UNIT                        m_add;
    LIST                            maps,maps_opp,maps_same,maps_comb,vfs_comb;

    (subvf_same,subvf_opp) ← vf_split(vf,bp);

    if (dir = LEFT)
        then m_add ← add(subvf_opp,g,RIGHT_END,m);
        else m_add ← add(subvf_opp,LEFT_END,g,m);
    fi

    if (m_add = FAIL)
        then
            if (dir = LEFT)
                then maps_opp ← insert(subvf_opp,g,RIGHT_END,m);
                else maps_opp ← insert(subvf_opp,LEFT_END,g,m);
            fi
        else maps_opp ← [m_add];
    fi

    if (maps_opp = FAIL)
        then return(FAIL);
        else
            maps_same ← [];
            if (dir = LEFT)
                then vfs_comb ← combining_vfs(subvf_same,LEFT_END,g,m);
                else vfs_comb ← combining_vfs(subvf_same,g,RIGHT_END,m);
            fi

            for vf_comb ∈ vfs_comb do
                maps_comb ← directed_combine(subvf_same,vf_comb,g,group_of(vf_comb),dir,m);
                if (maps_comb ≠ FAIL) then maps_same ← maps_same || maps_comb; fi
            rof

            if (maps_same = [])
                then
                    if (dir = LEFT)
                        then m_add ← add(subvf_same,LEFT_END,g,m);
                        else m_add ← add(subvf_same,g,RIGHT_END,m);
                    fi

                    if (m_add = FAIL)
                        then
                            if (dir = LEFT)
                                then maps_same ← insert(subvf_same,LEFT_END,g,m);
                                else maps_same ← insert(subvf_same,g,RIGHT_END,m);
                            fi
                        else maps_same ← [m_add];
                    fi

                    if (maps_same = FAIL) then return(FAIL); fi
            fi

            if (dir = LEFT)
                then maps ← concatenate_left_and_right_parts_of_maps(maps_same,maps_opp,vf,g);
                else maps ← concatenate_left_and_right_parts_of_maps(maps_opp,maps_same,vf,g);
            fi

            return(maps);
    fi
}
```

Figure 37: Psuedocode for directed_split

4.3.3.3. The internal_split Operation

The directed_split and undirected_split operations are intended to perform the initial split of the RSA. Typically, these operations will be splitting a virtual fragment that was chosen by the user or some DNA Mapping application program. Recall that in the GSA, it is sometimes necessary to perform several split operations within some sequence of operations. This occurs when the combine operation calls the split. A type of RSA split operation which performs this "non-initial" split is needed. This is the purpose the internal_split operation.

Two key ideas behind the internal_split operation are (1) it is (almost always) called only after some other split operation has been called, and (2) it is called only by a directed_combine. The internal_split operation uses the location of the most recent split in the sequence of operations which lead to it to restrict the range of the map unit where its subfragments can be placed. In addition, internal_split always has directionality. It inherits the direction from the directed_combine that calls it, because this sequence of operations should be progressing toward one end of the map unit.

The psuedocode for the internal_split operation is almost identical to that of the directed_split. Therefore, only the portion containing the differences is given in Figure 38. Each line containing a difference is followed by three asterisks. As input, internal_split takes a virtual fragment vf, a group g where the split is to occur, a pair bp representing the bipartition of the real fragments composing vf, a map unit m, a direction dir and a group g_{prev} that is the location of the previous split operation (the one that called the directed_combine that called the internal_split).

To examine the behavior of internal_split in more detail, consider Figure 39. Suppose vf_1 and vf_2 are virtual fragments which have been merged to form vf_3 by a right directed_combine. The fragment vf_1 is a subfragment of some split operation (undirected_split, directed_split or internal_split) that occured at the group g_1. The fragment vf_2 is from the group g_2. Furthermore, suppose that the right directed_combine then calls internal_split in its effort to place vf_3. In this situation, the internal_split is said to occur at g_2.

It can be shown that not all biparitions of the real fragments composing vf_3 need to be attempted. In particular, any bipartition which results in a real fragment originally in vf_1 ending up in the same direction subfragment in the internal_split need not be attempted. (Generating the bipartitions to attempt is the responsibility of directed_combine, and thus the reasoning for this restriction is deferred until §4.3.4.1.) For now, assume directed_combine is calling internal_split with some bipartition that is worth attempting. As in directed_split, vf_3 is broken into two subfragments: the opposite direction subfragment vf_{opp} and the same direction subfragment vf_{same}. This means that in this case, vf_{opp} must be placed somewhere to the left of g_2, and vf_{same} must be placed somewhere to the right of g_2. Internal_split then behaves exactly like directed_split, with one exception. Consider the placement of vf_{opp}, the opposite direction subfragment, which is only allowed to add or insert. It can be shown that an add or insert with vf_{opp} can only succeed between groups g_1 and g_2.

Let R_1 be the region of the map unit to the left of and including g_1. Let R_2 be the region of the map unit between, but not including, groups g_1 and g_2. Let R_3 be the region of the map unit to the right of and including g_2. Recall that vf_3 is a combined virtual fragment formed from vf_1 and vf_2. Thus, ACS(g_1) and ACS(g_2) are disjoint. (Note that this is slightly different than in the GSA.) Therefore, the ACS of any group in R_1 is disjoint from ACS(g_2), and the ACS of any group in R_3 is disjoint from ACS(g_1).

Because no real fragments originally in vf_1 are in vf_{same}, all of the real fragments originally in vf_1 must be in vf_{opp}. Thus, there is at least one real fragment in vf_{opp} that was originally in vf_2. (Otherwise vf_3 is broken into the subfragments that formed it.) This means that any add or insert with vf_{opp} will fail in R_1, since no elements of ACS(g_2) exist in R_1. Likewise, any add or insert with vf_{opp} will fail in R_3, since no elements of ACS(g_1) exist in R_3. Thus, one need only attempt add or insert operations with vf_{opp} in R_2. This is why the group where the previous split occured is important. It helps define the region of the map unit where vf_{opp} can be placed.

```
internal_split(vf,g,g_prev,bp,dir,m) ***
    VIRTUAL_FRAGMENT          vf;
    GROUP                     g,g_prev; ***
    PAIR                      bp;
    DIRECTION                 dir;
    MAP_UNIT                  m;
{
    VIRTUAL_FRAGMENT          subvf_opp,subvf_same,vf_comb;
    MAP_UNIT                  m_add;
    LIST                      maps,maps_opp,maps_same,maps_comb,vfs_comb;

    (subvf_same,subvf_opp) ← vf_split(vf,bp);

    if (dir = LEFT)
        then m_add ← add(subvf_opp,g,g_prev,m); ***
        else m_add ← add(subvf_opp,g_prev,g,m); ***
    fi

    if (m_add = FAIL)
        then
            if (dir = LEFT)
                then maps_opp ← insert(subvf_opp,g,g_prev,m); ***
                else maps_opp ← insert(subvf_opp,g_prev,g,m); ***
            fi
        else maps_opp ← [m_add];
    fi

    .
    .
    .
    same as directed_split
    .
    .
    .

}
```

Figure 38: Psuedocode for internal_split

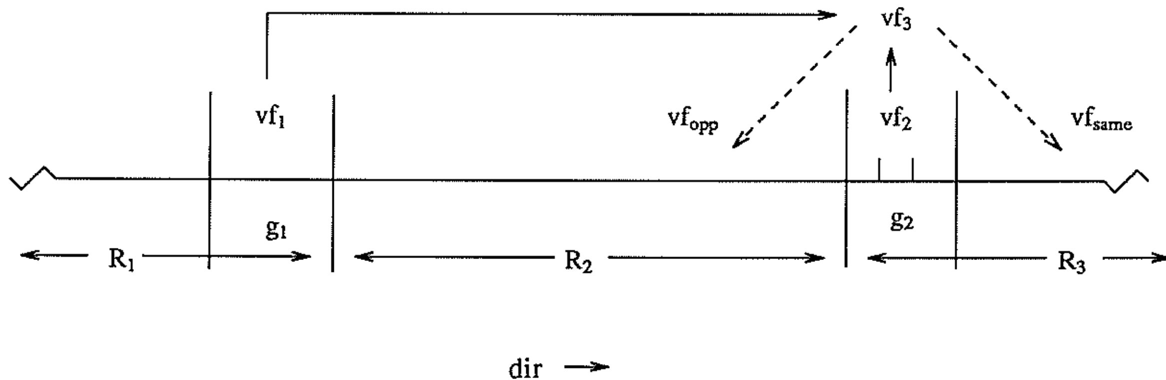


Figure 39: A typical internal_split

4.3.3.4. The top_level_split Operation

The top_level_split operation acts primarily as an interface to the splitting functionality implemented by the directed_split, undirected_split and internal_split operations. This is the operation that is called by the user or DNA Mapping application program. One specifies the virtual fragment to be split, the bipartition determining how the virtual fragment should be decomposed, the direction the initial split should have (if any) and the map unit containing the virtual fragment. The psuedocode for top_level_split is given in Figure 40. In the actual implementation, it is more complicated, but these details are discussed in §4.7.2.

The top_level_split operation has a special responsibility. Recall that the directed_split and undirected_split operations take an ordered pair as the representation of the bipartition that determines how the virtual fragment is to be decomposed. These operations use the elements of the pair in a specific manner. For example, when undirected_split takes a pair *bp*, it always uses the left element of *bp* to construct the left subfragment and the right element of *bp* to construct the right subfragment. However, there may be a solution using the same subfragments, but sending them in the opposite directions. (I.e., a solution might be obtained where the left element of *bp* is the right subfragment and the right element of *bp* is the left subfragment.) Thus, undirected_split should be called twice, once with *bp* and once with the left and right element of *bp* swapped. Then all the solutions for a given *bipartition* are found. A similar argument can be made with directed_split. Therefore, it is the responsibility of top_level_split to call the appropriate split operation twice and to concatenate the lists of solutions it receives from the two calls.

4.3.4. The Combine Operations

Like the split, the concepts of directionality and operation bias complicate the combine operation in the RSA. The result is that now there are actually three different combine operations: directed_combine, undirected_combine and top_level_combine.

The conditions for combining two virtual fragments are slightly different in the RSA than those the GSA (i.e., p_1, p_2 and p_3). Suppose one wants to know if virtual fragments vf_1 and vf_2 can form a valid combined virtual fragment vf_3. Let g_1 be the group that vf_1 is in or the group where the split that creates vf_1 occurs. Let g_2 be defined similarly for vf_2. Like the GSA condition p_2, the real fragments composing vf_3 must have some ordering that allows them to be matched one at a time using the same matching criteria used to build the map unit. (This is p_2'.) Like the GSA condition p_3, for some c_1 in ACS(vf_1) and some c_2 in ACS(vf_2), there must be a group g where $\{c_1,c_2\}$ is a subset of ACS(g) or there must be two consecutive groups where one contains an element of ACS(vf_1) and the other contains an element of ACS(vf_2). (This is p_3'.)

The condition that is different is the one concerning ACS disjointness. In the GSA condition p_1, it is required that ACS(vf_1) and ACS(vf_2) be disjoint. In the RSA condition p_1', it is required that ACS(g_1)

```
top_level_split(vf,bp,dir,m)
    VIRTUAL_FRAGMENT          vf;
    PAIR                      bp;
    DIRECTION                 dir;
    MAP_UNIT                  m;
{
    GROUP                     g;
    LIST                      solutions,sol1,sol2;

    solutions ← [];
    g ← group_of(vf);

    if (dir = NONE)
        then
            sol1 ← undirected_split(vf,g,bp,m);
            sol2 ← undirected_split(vf,g,swap(bp),m);
        else
            sol1 ← directed_split(vf,g,bp,dir,m);
            sol2 ← directed_split(vf,g,swap(bp),dir,m);
    fi

    if (sol1 = FAIL and sol2 = FAIL)
        then solutions ← FAIL;
        else
            if (sol1 ≠ FAIL and sol2 ≠ FAIL)
                then solutions ← sol1 || sol2;
                else
                    if (sol1 = FAIL)
                        then solutions ← sol2;
                        else solutions ← sol1;
                    fi
            fi
    fi

    return(solutions);
}
```

Figure 40: Psuedocode for top_level_split

and ACS(g_2) be disjoint. The distinction is subtle, but important. If both vf_1 and vf_2 are actually in the groups g_1 and g_2, respectively, there is no difference between the conditions because the ACS of the fragment is the same as the ACS of the corresponding group. However, if the fragment is actually a subfragment from a split occuring at the group, then the ACS of the fragment is a proper subset of the ACS of the group. Thus, this new requirement is more difficult to satisfy than the old one. Fragments vf_1 and vf_2 could be composed of real fragments from completely different sets of clones and still not allowed to combine, because the ACSs of the groups are not disjoint.

The condition p_1' creates a number of nice properties which help alleviate problems concerning the undoing of previous work. Naturally, this also reduces the number of solutions that will be found. For the sake of clarity, the three conditions in the RSA are summarized below.

(p_1') $\quad$ $\text{ACS}(g_1) \cap \text{ACS}(g_2) = \varnothing$.

(p_2') There is an order of the real fragments of vf_1 and vf_2 such that, together, they can be matched with the same criteria used during the construction of the original map unit.

(p_3') There must exist $c_1 \in$ ACS(vf_1) and $c_2 \in$ ACS(vf_2) such that either (1) there is a group g where $c_1, c_2 \in$ ACS(g) or (2) there are consecutive groups g_1 and g_2 where $c_1 \in$ ACS(g_1) and $c_2 \in$ ACS(g_2).

4.3.4.1. The directed_combine Operation

The directed_combine is similar to the combine operation in the GSA. It is primarily intended to be called by split operations. (This should be evident from the psuedocode for directed_split, undirected_split and internal_split Directed_combine merges a subfragment from a split with a virtual fragment that exists in the map unit. If a directed_split or internal_split calls directed_combine, then directed_combine inherits its direction so that the sequence of operations continues toward one end of the map unit. If an undirected_split calls directed_combine, then the direction is based upon which subfragment of the undirected_split is being combined. A left subfragment causes a left directed_combine, and a right subfragment causes a right directed_combine.

The psuedocode for directed_combine is given in Figure 41. As input, directed_combine takes two virtual fragments vf_1 and vf_2, two groups g_1 and g_2, a direction *dir* and a map unit *m*. The fragment vf_1 is assumed to be a subfragment resulting from a split occuring at g_1. The fragment vf_2 is assumed to exist in *m* in the group g_2. If vf_1 and vf_2 satisfy p_1', p_2' and p_3', the combined virtual fragment vf_3 is formed.

In the split operations, there is an operation bias toward combining. In the combine operations, there is an operation bias toward add and insert. This is because of the preference for compact map units. A solution that requires vf_3 to be split tends to be less compact than a solution obtained when vf_3 can add or insert, because two virtual fragments must be placed in the map unit instead of only one.

Directed_combine first attempts to add vf_3. If the add is successful, the map unit that results is placed in the list *maps*. If the add fails, then directed_combine attempts to insert vf_3. If the insert is successful, the map units that result are placed in the list *maps*. It can be shown that the only region where the add or insert operation can succeed is between g_1 and g_2. This is true for basically the same reasons that the opposite direction subfragment in the internal_split can only add or insert within a certain region of the map unit. Assume, without loss of generality, that g_1 is to the left of g_2. Let R_1 be the region of the map unit to the left of and including g_1. Let R_2 be the region of the map unit between but not including groups g_1 and g_2. Let R_3 be the region of the map unit to the right of and including g_2. It is known that ACS(g_1) and ACS(g_2) are disjoint. Thus, the ACS of any group in R_1 and ACS(g_2) are disjoint. Also, the ACS of any group in R_3 and ACS(g_1) are disjoint. Since ACS(vf_3) contains at least one element of ACS(g_1) and of ACS(g_2), directed_combine cannot possibly add or insert vf_3 in R_1 or R_3. Therefore, an add or insert with vf_3 can only succeed in R_2.

If both the add and insert fail, then vf_3 must be split. Since this is not the first split to occur, an internal_split is used. In the GSA, the combine operation calls the split operation using all partitions of the set of real fragments composing vf_3 containing more than one block. In the RSA, only bipartitions are used. In addition, there are some bipartitions which need not be tried.

Let vf_{same} be the same direction subfragment of the internal_split to be performed at g_2. Then vf_{same} must be placed somewhere in region R_3. Since the ACS of any group in R_3 is disjoint from ACS(vf_1), any bipartition that puts a real fragment originally in vf_1 into vf_{same} can be skipped. Otherwise, a real fragment from some clone c is being sent into a region of the map unit where it is known that c does not exist.

Thus, directed_combine calls internal_split using all bipartitions where all of the real fragments originally in vf_1 are put in the opposite direction subfragment. All the solutions obtained from the

```
directed_combine(vf1,vf2,g1,g2,dir,m)
    VIRTUAL_FRAGMENT                    vf1,vf2;
    GROUP                               g1,g2;
    DIRECTION                           dir;
    MAP_UNIT                            m;
{
    VIRTUAL_FRAGMENT                    vf_comb;
    GROUP                               g_left,g_right;
    PAIR                                bp;
    SET                                 rfs_same,rfs_opp;
    LIST                                maps,maps_split;
    MAP_UNIT                            map_new;

    g_left ← leftmost_group_of(g1,g2);
    g_right ← rightmost_group_of(g1,g2);
    maps ← [];

    if (vfs_might_combine(vf1,vf2,g1,g2,m))
        then
            vf_comb ← vf_combine(vf1,vf2);
            map_add ← add(vf3,m,g_left,g_right);

            if (map_add = FAIL)
                then
                    maps ← insert(vf_comb,m,g_left,g_right);

                    if (maps = FAIL)
                        then
                            maps ← [];

                            for rfs_same ∈ {x | x ⊂ rfs_of(vf2)} do
                                rfs_opp ← rfs_of(vf1) ∪ (rfs_of(vf2) — rfs_same);

                                if (rfs_have_matching_order(rfs_opp) and
                                   rfs_have_matching_order(rfs_same))
                                     then
                                         bp ← (rfs_opp,rfs_same);
                                         maps_split ← internal_split(vf_comb,g2,g1,bp,dir,m);

                                         if (maps_split = FAIL)
                                             then maps ← maps || maps_split;
                                         fi
                                fi
                            rof
                    fi
                else maps ← [map_add];
            fi
    fi

    if (maps = [])
        then return(FAIL);
        else
            for map_new ∈ maps do remove_vf(vf2,map_new); rof
            return(maps);
    fi
}
```

Figure 41: Psuedocode for directed_combine

internal_split operations are collected in the list *maps*. Finally, if any solutions are in *maps*, then vf_2 is removed from each map unit in *maps*, and *maps* is returned. (Note that since vf_1 is a subfragment, it is not actually in m and thus does not have to be removed.) Otherwise, FAIL is returned.

4.3.4.2. The undirected_combine Operation

The directed_combine fits nicely into the scheme of splitting a virtual fragment because the initial split creates two sequences of operations. In one sequence, the operations have left directionality, and in the other sequence, the operations have right directionality. However, it may be that the user wishes to take two virtual fragments in a map unit, merge them into a single virtual fragment, and see if this new fragment can be incorporated in the map unit somehow.

The directed_combine performs this kind of action, except that there is a difference in the way the left and right regions of the map unit are treated. In the context of a fragment split, this difference has a purpose. However, if a combine is used as the initial operation, there may be no reason to treat one region of the map unit differently than the other. Thus, a combine operation that lacks directionality is necessary. This is the purpose of the undirected_combine operation. The psuedocode for undirected_combine is given in Figure 42.

How does the undirected_combine work? Suppose virtual fragments vf_1 and vf_2 are located in the map unit m in the groups g_1 and g_2, respectively. If vf_1 and vf_2 satisfy conditions p_1', p_2' and p_3', then the combined virtual fragment vf_3 is formed.

Like in the directed_combine, first an add with vf_3 is attempted. If the add fails, then an insert with vf_3 is attempted. For the same reasons as in the directed_combine, the add or insert can only succeed between g_1 and g_2. If add and insert both fail, then vf_3 must be split.

However, this split is not as straightforward as a split in a directed_combine. To begin with, at what group does this split occur? Since there is no directionality an undirected_combine, there is no reason to favor one group over the other. Also, are there any limits on where the subfragments can be placed? Fortunately, there are some observations that simplify the situation. As with any split, the fragment vf_3 is broken up into two subfragments vf_4 and vf_5 using some bipartition, except the bipartition that creates subfragments vf_1 and vf_2. Assuming that g_1 is to the left of g_2, define the regions R_1, R_2 and R_3 as in §4.3.4.1.

There may be virtual fragments in R_2 that satisfy p_1', p_2' and p_3' with either vf_4 or vf_5. However, combines will not be allowed in that region because if the combined virtual fragment that is formed cannot add or insert, once again a virtual fragment must be split where there is no directionality and no limitations on where the subfragments may be placed. (I.e., nothing has been gained.) Therefore, subfragments may only add or insert in R_2.

Given this, it is impossible for both vf_4 and vf_5 to add or insert to R_2. To show that this is true, assume that vf_4 and vf_5 successfully add or insert in R_2. Let g_4 be the group containing vf_4 and let g_5 be the group containing vf_5. (Note that $ACS(g_4) \cap ACS(g_5) = \emptyset$.) Assume that g_4 is to the left of g_5. Then either (1) $ACS(g_4) \subset ACS(g_1)$ (i.e., vf_4 is composed of real fragments originally in vf_1), (2) $ACS(g_5) \subset ACS(g_2)$ (i.e., vf_5 is composed of real fragments originally in vf_2) or (3) $ACS(g_4) \cap ACS(g_1) \neq \emptyset$, $ACS(g_4) \cap ACS(g_2) \neq \emptyset$, $ACS(g_5) \cap ACS(g_1) \neq \emptyset$ and $ACS(g_5) \cap ACS(g_2) \neq \emptyset$ (i.e., vf_4 and vf_5 contain real fragments originally in vf_1 and vf_2). In case (1), there exists a clone c such that $c \in ACS(g_1)$ and $c \in ACS(g_5)$. Since g_4 is between g_1 and g_5, then $c \in ACS(g_4)$. Thus, $ACS(g_4) \cap ACS(g_5) \neq \emptyset$, which is impossible. In case (2), essentially the same argument holds as in case (1) (with g_4 and g_5 switched). In case (3), the same argument holds as in case (1). Therefore, it is impossible for both vf_4 and vf_5 to add or insert to R_2.

It is also impossible for one subfragment to be placed in R_1 while the other subfragment is placed in R_3. To show that this is true, suppose vf_4 is placed in R_1 and vf_5 is placed in R_3. The ACS of any group in R_1 is disjoint from $ACS(g_2)$. Thus, vf_4 contains no real fragments originally in vf_2. The ACS of any group in R_3 is disjoint from $ACS(g_1)$. Thus, vf_5 contains no real fragments originally in vf_1. However, the only subfragments that satisfy these two constraints are vf_1 and vf_2, and the bipartition that creates these subfragments is not used. Therefore, no bipartition (of interest) creates subfragments such that one is placed in R_1 and the other is placed in R_3.

```
undirected_combine(vf₁,vf₂,g₁,g₂,m)
    VIRTUAL_FRAGMENT            vf₁,vf₂;
    GROUP                       g₁,g₂;
    MAP_UNIT                    m;
{
    MAP_UNIT                    m_new;
    LIST                        maps,maps_left,maps_right;

    if (not(vfs_might_combine(vf₁,vf₂,g₁,g₂,m)))
        then return(FAIL);
    fi

    maps ← [];
    maps_right ← directed_combine(vf₁,vf₂,g₁,g₂,RIGHT,m);

    if (maps_right ≠ FAIL)
        then
            for map_new ∈ maps_right do remove_vf(vf₁,map_new); rof
            maps ← maps || maps_right;
    fi

    maps_left ← directed_combine(vf₂,vf₁,g₂,g₁,LEFT,m);

    if (maps_left ≠ FAIL)
        then
            for map_new ∈ maps_left do remove_vf(vf₂,map_new); rof
            maps ← maps || maps_left;
    fi

    if (maps = [])
        then return(FAIL);
        else
            remove_equivalent_maps(maps);
            return(maps);
    fi
}
```

Figure 42: Psuedocode for undirected_combine

Therefore, exactly one subfragment of the split must always be placed in R_2. So there are really two possibilities. The first possibility is where one subfragment is placed in R_1 and the other in R_2. This placement can be obtained by performing a left directed_split at g_1. This is what a left directed_combine at g_1 will do if vf_3 does not successfully add or insert. The second possibility is where one subfragment is placed in R_3 and the other in R_2. This placement can be obtained by performing a right directed_split at g_2. This is what a right directed_combine at g_2 will do if vf_3 does not successfully add or insert.

Thus, undirected_combine consists of a left directed_combine at g_1 and a right directed_combine at g_2. Most solutions, but not all, obtained by each directed_combine are different. A solution where add or insert succeeds with vf_3 can be obtained by both directed_combine operations. Thus, undirected_combine must eliminate the duplicate solutions before returning.

In addition, undirected_combine must remove vf_1 and vf_2 from the map units it returns. The undirected_combine assumes that both vf_1 and vf_2 are actually in the map unit. However, the directed_combine operation assumes that one of the virtual fragments being combined is not actually in the map unit, and thus undirected_combine is responsible for removing the virtual fragment that directed_combine does not remove.

4.3.4.3. The top_level_combine Operation

Like top_level_split, top_level_combine is primarily an interface to the combining functionality. The virtual fragments to combine, the direction of the combine and the map unit must be specified. Although it may not seem that one would ever want an initial combine to have direction, it should be allowed if the user really wishes. (The user may have outside information that compels him or her to do so.)

The top_level_combine operation determines which fragment and group is to the left and to the right, calls the appropriate combine operation and makes minor modifications to the solutions when necessary. The psuedocode is given in Figure 43.

```
top_level_combine(vf1,vf2,dir,m)
    VIRTUAL_FRAGMENT          vf1,vf2;
    DIRECTION                 dir;
    MAP_UNIT                  m;
{
    VIRTUAL_FRAGMENT          vf_left,vf_right;
    GROUP                     g_left,g_right;
    LIST                      maps;
    MAP_UNIT                  map_new;

    vf_left ← leftmost_frag_of(vf1,vf2);
    vf_right ← rightmost_frag_of(vf1,vf2);
    g_left ← group_of(vf_left);
    g_right ← group_of(vf_right);

    if (dir = NONE)
        then
            maps ← undirected_combine(vf_left,vf_right,g_left,g_right,m);
        else
            maps ← directed_combine(vf_left,vf_right,g_left,g_right,dir,m);
            if (maps ≠ FAIL)
                then for map_new ∈ maps do remove_vf(vf_left,map_new) rof
            fi
    fi

    return(maps);
}
```

Figure 43: Psuedocode for top_level_combine

4.3.5. Calling Relationship between the Operations

Figure 44 illustrates the relationship between the operations described in §4.3. Each circle represents an operation and an arrow from circle x to circle y means that operation represented by x calls the operation represented by y. One can now see how the internal_split and directed_combine form the mechanism that allows a sequence of operations to continue and move toward the end of the map unit. In addition, one can see how all the operations eventually lead to add and insert, the two primitive operations.

4.4. High Level Description

The GSA operations form the core of the GSA, but there is some external control that is necessary to prevent infinite sequences of operations. Because of their design, the RSA operations do not require such external control. Thus, it can be said that the RSA operations really are the RSA. The user or application program simply calls the top_level_split or top_level_combine operation to start the RSA.

4.5. Examples of Using top_level_split

In order to better illustrate how the operations in the RSA work together to produce solutions, some examples of splitting virtual fragments using the top_level_split operation are now presented.

4.5.1. Example One

The first example is the same situation as the example in §3.2.1. This is the simplest possible situation involving a fragment matching mistake. Suppose Figure 18 represents the underlying reality. Then the Map unit m produced is given in Figure 19.

The fragment $vf<f_1,f_2>$ should be split. There is only one bipartition of the set of real fragments that compose $vf<f_1,f_2>$, namely $\{\{f_1\},\{f_2\}\}$. Thus, the initial call is: top_level_split ($vf<f_1,f_2>$,($\{f_1\},\{f_2\}$),NONE,m). Since the parameter *dir* has the value NONE, the undirected_split is called twice.

In the first call to undirected_split, the pair ($\{f_1\},\{f_2\}$) determines how to decompose $vf<f_1,f_2>$. So $vf<f_1,f_2>$ is decomposed into the left subfragment $vf<f_1>$ and the right subfragment $vf<f_2>$. Let g be the group containing $vf<f_1,f_2>$. First, undirected_split attempts to place the left subfragment in the region of the m between the left end and the group g. Undirected_split attempts a left directed_combine with the left subfragment. It searches for virtual fragments in the region of interest that satisfy p_1', p_2' and p_3' with $vf<f_1>$. In this case, no such virtual fragments are found. Thus, undirected_split attempts to perform an add with $vf<f_1>$ in the region of interest. The leftmost group of m, which has an ACS of $\{c_1\}$, is in this region. Thus, the add with $vf<f_1>$ succeeds here.

Now that the left subfragment is placed, undirected_split attempts to place the right subfragment. The region of interest now is between the right end of m and the group g. Undirected_split attempts a right directed_combine. It searches for virtual fragments that satisfy p_1', p_2' and p_3' with $vf<f_2>$ in the region of interest. Again, no such virtual fragments are found. Then, undirected_split attempts to perform an add in the region of interest. The rightmost group of m, which has an ACS of $\{c_2\}$, is in this region. Thus, the add with $vf<f_2>$ succeeds here.

Since the left and right subfragment are successfully placed, a solution has been found and is given in Figure 20.

In the second call to undirected_split, the pair ($\{f_2\},\{f_1\}$) determines how to decompose $vf<f_1,f_2>$. The same subfragments are created, but this time $vf<f_1>$ is the right subfragment and $vf<f_2>$ is the left subfragment. First, undirected_split attempts to find virtual fragments that can

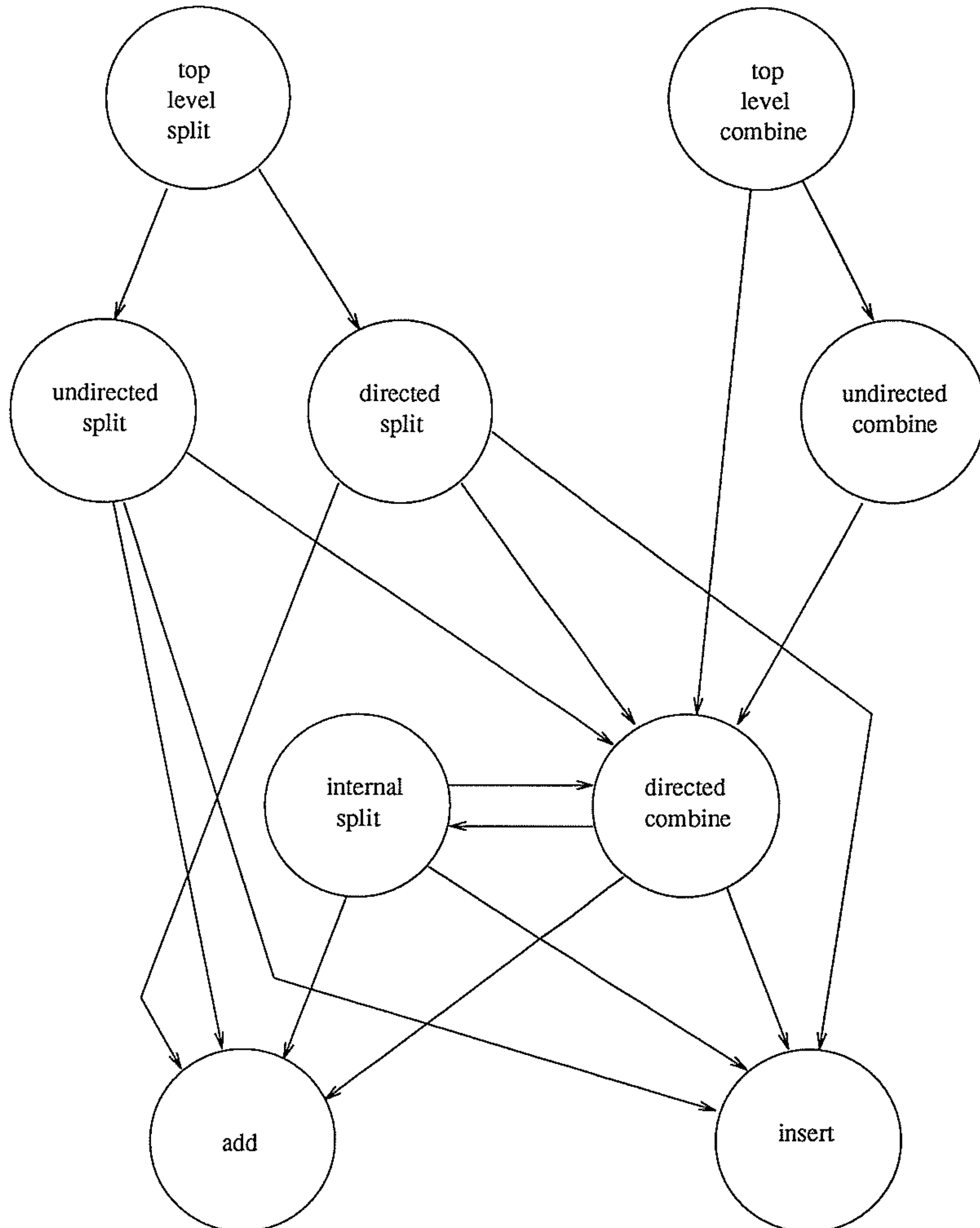


Figure 44: Calling relationship between the RSA operations

combine with $vf<f_2>$ in the region of m between the left end and g. None are found, and thus undirected_split attempts to add $vf<f_2>$ in the same region. Since no group *in that region* has an ACS

of $\{c_2\}$, the add fails. Next, undirected_split attempts to insert $vf<f_2>$ in the region. However, there is no place to create a group with an ACS of $\{c_2\}$ in that region, and thus insert fails. The left subfragment could not be placed. Consequently, the second call to undirected_split returns FAIL.

So top_level_split obtains one solution (the one in Figure 20) from the two calls to undirected_split. It then returns this solution to the user. Note that this is the same solution obtained by the GSA.

4.5.2. Example Two

The second example is more complicated than the first and shows how the RSA tends to return more compact map units as solutions. Suppose Figure 45 represents the underlying reality. Fusing in the order c_2, c_3, c_4 and c_1 produces the Map unit m in Figure 46.

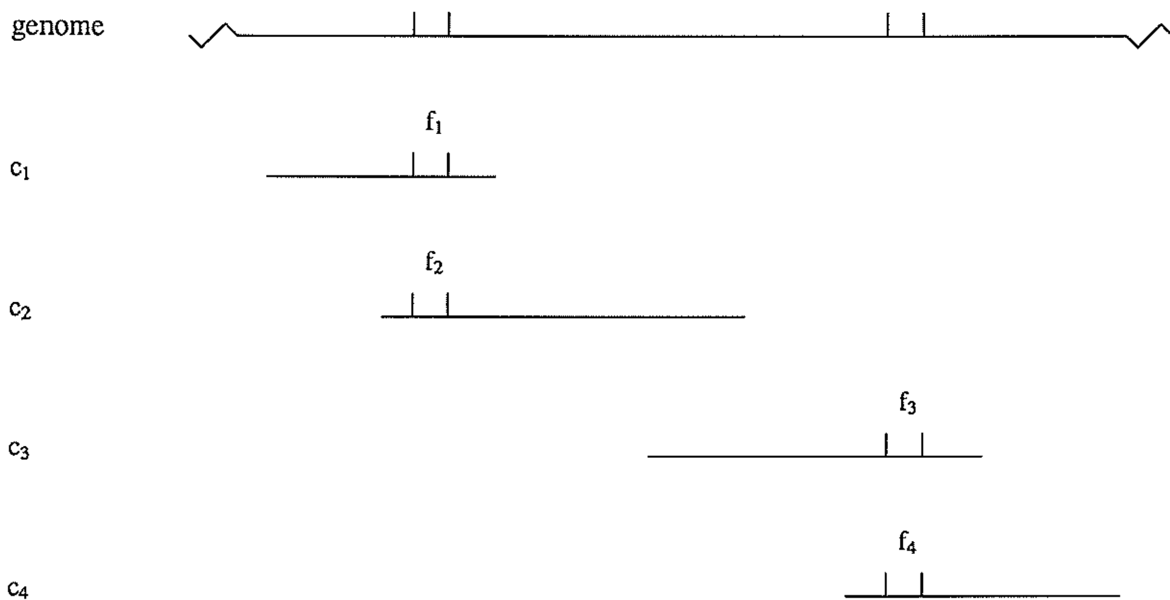


Figure 45: Clone configuration for Example two

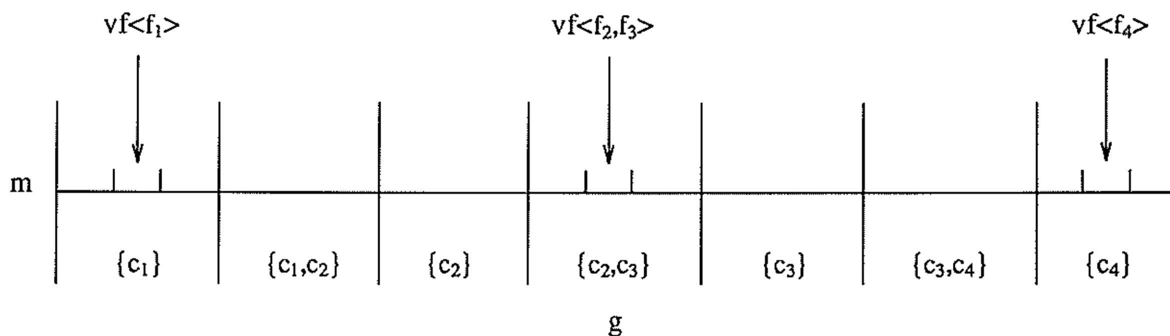


Figure 46: Map unit built from clones in Figure 45

The virtual fragment to be split is $vf<f_2,f_3>$. There is only one bipartition, and thus the initial call is: top_level_split ($vf<f_2,f_3>$,($\{f_2\},\{f_3\}$),NONE,m). As in the first example, this results in two calls to undirected_split.

The first call has ($\{f_2\},\{f_3\}$) as the value of the parameter *bp*. Thus, $vf<f_2>$ is the left subfragment and $vf<f_3>$ is the right subfragment. Let g be the group containing $vf<f_2,f_3>$. First, undirected_split searches for virtual fragments in the region of m between the left end and g to combine with $vf<f_2>$. The virtual fragment $vf<f_1>$ is the only one that satisfies p_1', p_2' and p_3' with vf_2. Thus, a left directed_combine with $vf<f_1>$ and $vf<f_2>$ is attempted, and the combined virtual fragment $vf<f_1,f_2>$ is formed. Then, directed_combine can successfully add $vf<f_1,f_2>$ to the group with an ACS of $\{c_1,c_2\}$. Since the add is successful, directed_combine returns the new map unit to undirected_split.

Since undirected_split successfully placed the left subfragment, it next attempts to place the right subfragment. The placement of the right subfragment is similar to that of the left subfragment. Undirected_split searches for virtual fragments in the region of m between the right end and g to combine with $vf<f_3>$. The virtual fragment $vf<f_4>$ is the only one that satisfies p_1', p_2' and p_3' with vf_3. A right directed_combine is attempted and the combined virtual fragment $vf<f_3,f_4>$ is formed. Then, add is successful with $vf<f_3,f_4>$. The map unit that results is returned to undirected_split. Undirected_split uses the map units resulting from the placement of the left and right subfragments to form a single solution (using the process illustrated in Figure 35), given in Figure 27.

The second call to undirected_split has ($\{f_3\},\{f_2\}$) as the value of the parameter *bp*. Thus, $vf<f_2>$ is the right subfragment and $vf<f_3>$ is the left subfragment. There are no virtual fragments in the region of m between the left end and g that satisfies p_1', p_2' and p_3' with $vf<f_3>$. (Note that $vf<f_1>$ and $vf<f_3>$ do not satisfy p_3'.) So undirected_split attempts to add $vf<f_3>$ in that region, but no group with an ACS of $\{c_3\}$ exists there. The insert fails for similar reasons, and thus the left subfragment cannot be placed. Consequently, the second call to undirected_split returns FAIL.

So as in Example one, top_level_split obtains one solution (the one in Figure 27) from the two calls to undirected_split. This is the solution returned to the user. The GSA obtains more than one solution in this situation. The additional solutions that it obtains are given in Figure 28. However, note that each of these solutions is less compact than the one found by the RSA. Thus, out of all the map units that could have resulted from fixing the fragment matching mistake, the RSA returns the preferred map unit. The primary reason that the less compact solutions are not found is because the operation bias toward directed_combine in the directed_split operation prevented add operations with $vf<f_2>$ and $vf<f_3>$ from occuring.

4.5.3. Example Three

There are some structural similarities between the second and third example. Here the need for the internal_split should become clear. Suppose Figure 26 represents the underlying reality. Then the Map unit m produced by using the clone order c_1, c_2, c_3 and c_4 is given in Figure 27. Let g_1 be the group containing $vf<f_1,f_2>$, and let g_2 be the group containing $vf<f_3,f_4>$.

Here, it is not clear which virtual fragment should be split. Both $vf<f_1,f_2>$ and $vf<f_3,f_4>$ are candidates. It turns out that in this case, the outcome is the same regardless of which is chosen. For now, assume that $vf<f_1,f_2>$ is chosen as the fragment to split. Again, there is only one bipartition, so the initial call is: top_level_split ($vf<f_1,f_2>$,($\{f_1\},\{f_2\}$),NONE,m).

The first call to undirected_split uses the pair ($\{f_1\},\{f_2\}$) to decompose the fragment. The left subfragment is $vf<f_1>$, and the right subfragment is $vf<f_2>$. First, undirected_split searches for virtual fragments in m between the left end and g_1 to combine with $vf<f_1>$. However, no fragments with the correct length exist, and thus undirected_split attempts to add $vf<f_1>$ in the region of interest. The add is successful at the group with an ACS of $\{c_1\}$. Next, undirected_split attempts to place the right subfragment. The virtual fragment $vf<f_3,f_4>$ is the only fragment satisfying p_1', p_2' and p_3' with $vf<f_2>$. Thus,

a right directed_combine is attempted with $vf<f_2>$ and $vf<f_3,f_4>$, and the combined virtual fragment $vf<f_2,f_3,f_4>$ is formed. Then directed_combine attempts to add $vf<f_2,f_3,f_4>$, but this fails. Next, it attempts to insert $vf<f_2,f_3,f_4>$, but this fails as well. Thus, directed_combine calls a right internal_split on $vf<f_2,f_3,f_4>$, using all pairs (i.e., bipartitions) of interest. In this case, the pairs that are tried are $(\{f_2,f_3\},\{f_4\})$ and $(\{f_2,f_4\},\{f_3\})$.

In calling internal_split using $(\{f_2,f_3\},\{f_4\})$, opposite direction subfragment $vf<f_2,f_3>$ and same direction subfragment $vf<f_4>$ are created. Internal_split first attempts to add $vf<f_2,f_3>$ between g_1 and g_2. This is successful because a group with an ACS of $\{c_2,c_3\}$ exists. Next, internal_split searches for fragments to the right of g_2 to combine with vf_4. None exist, and thus internal_split attempts to add $vf<f_4>$ to the right of g_2. This is successful, because a group with an ACS of $\{c_4\}$ exists. So one solution is found, and it is given in Figure 47.

Calling internal_split using the pair $(\{f_2,f_4\},\{f_3\})$ creates opposite direction subfragment $vf<f_2,f_4>$ and same direction subfragment $vf<f_3>$. The opposite direction subfragment $vf<f_2,f_4>$ fails to add or insert between g_1 and g_2. Because the opposite direction subfragment is only allowed to add or insert, internal_split now fails. So the two calls to internal_split result in a single solution.

The second call to undirected_split uses the pair $(\{f_2\},\{f_1\})$ to decompose $vf<f_1,f_2>$. Now, the left subfragment is $vf<f_2>$, and the right subfragment is $vf<f_1>$. Obviously, there are no fragments to the left of g_1 with which $vf<f_2>$ can combine. Neither add nor insert are successful with $vf<f_2>$ to the left of g_1, and so the second call to undirected_split fails.

Thus, the only solution returned by top_level_split is the one in Figure 47. Like Example two, the GSA obtains more than one solution. However, the solution found by the RSA is more compact than any obtained solely by the GSA.

4.5.4. Example Four

The fourth example demonstrates the need for the insert operation. Suppose Figure 48 represents the underlying reality. Assume f_1 and f_2 are of similar length and that no other fragments in the three clones have lengths in that range. Note that the right end of c_1 and the left end of c_3 are in the same section of the genome as f_2. Fusing with the clone order c_1, c_2 and c_3 produces the Map unit m given in Figure 49. Let g_1 be the group with an ACS of $\{c_1,c_2\}$, and let g_2 be the group with an ACS of $\{c_2,c_3\}$.

The virtual fragment $vf<f_1,f_2>$ should be split. The initial call is: top_level_split $(vf<f_1,f_2>,(\{f_1\},\{f_2\}),\text{NONE},m)$.

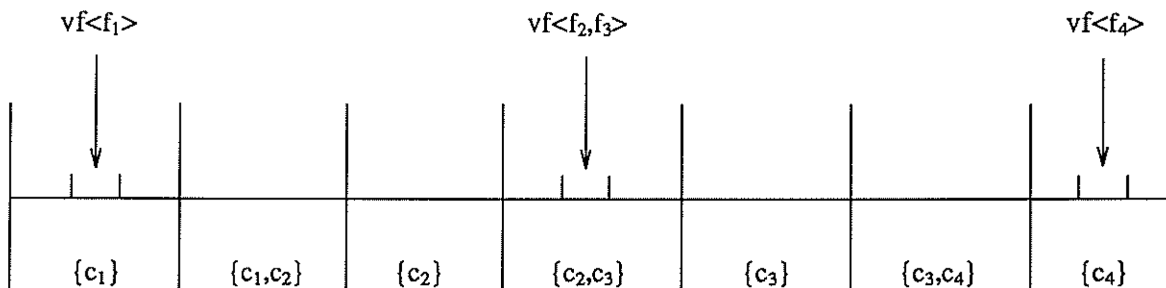


Figure 47: Map unit resulting from the top_level_split

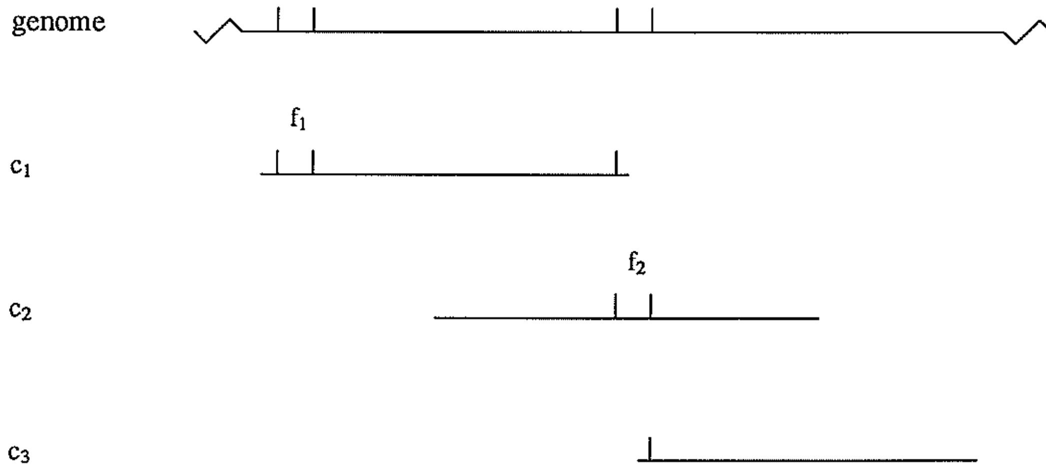


Figure 48: Clone configuration for Example four

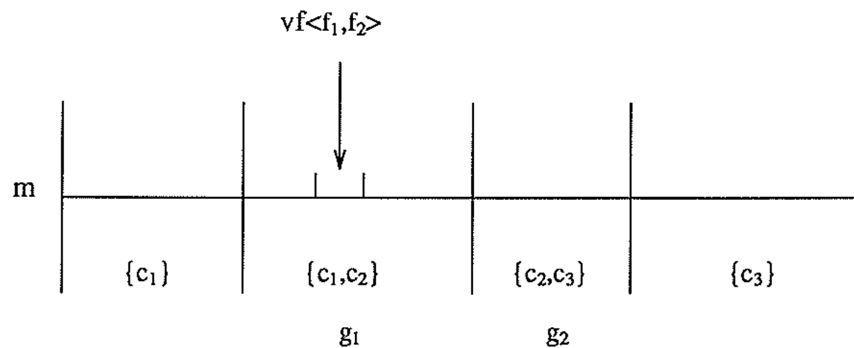


Figure 49: Map unit built from clones in Figure 48

The first call to undirected_split uses the pair $(\{f_1\},\{f_2\})$, which means $vf<f_1>$ is the left subfragment and $vf<f_2>$ is the right subfragment. Since there are no virtual fragments to the left of g_1 that satisfy p_1', p_2' and p_3' with $vf<f_1>$, undirected_split attempts to add $vf<f_1>$ to the left of g_1. The add operation successfully places $vf<f_1>$ in the group with an ACS of $\{c_1\}$. There are no virtual fragments to the right of g_1 to combine with $vf<f_2>$, so an add is attempted. The add operation fails, since there is no group with an ACS of $\{c_2\}$. However, the insert operation succeeds, because a group with an ACS of $\{c_2\}$ can be created between groups g_1 and g_2. Thus, the first call to undirected_split results in one solution, given in Figure 50.

The second call to undirected_split uses the pair $(\{f_2\},\{f_1\})$, and so $vf<f_2>$ is the left subfragment and $vf<f_1>$ is the right subfragment. It is easy to see that there are no fragments to the left of g_1 that satisfy p_1', p_2' and p_3' with $vf<f_2>$. Also, add and insert operations with $vf<f_2>$ to the left of g_1 fail. Consequently, the second call to undirected_split fails. Therefore, the only solution found by top_level_split is the one in Figure 50. In this case, the GSA returns only this solution as well.

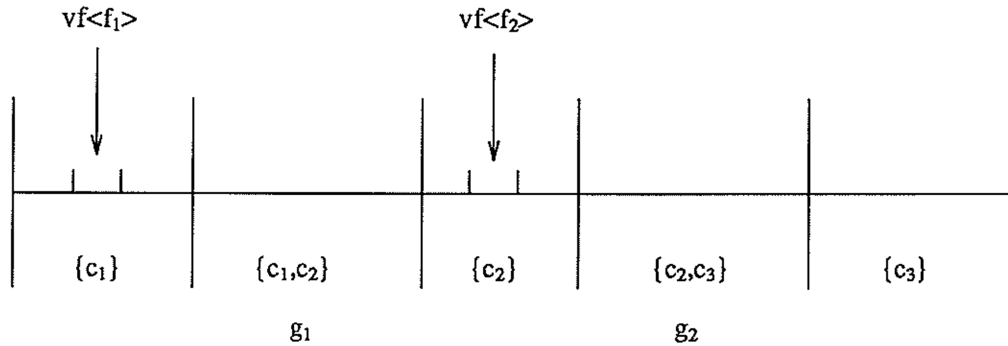


Figure 50: Map unit resulting from the top_level_split

4.5.5. Example Five

In the first four examples, all of the initial splits were undirected. (I.e., the *dir* parameter was set to NONE.) The fifth and final example is intended to show why one would use a directed split as the initial split. Suppose Figure 25 represents the underlying reality. Fusing with the clone order c_1, c_2 and c_3 produces the Map unit m_1 given in Figure 51.

Using the call top_level_split ($vf<f_1,f_2>,(\{f_1\},\{f_2\})$,NONE,$m_1$) results in one solution, m_2, given in Figure 52. However, this solution does not correspond to the underlying reality. In this case, the most compact map unit is not the correct map unit. Suppose at some point in time, it is determined that

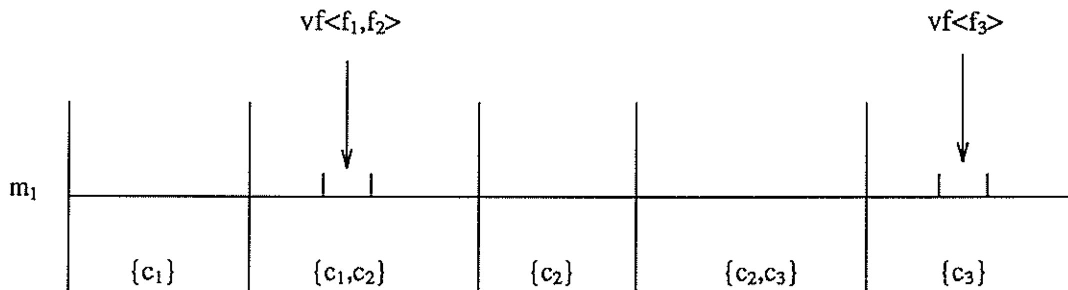


Figure 51: Map unit built from clones in Figure 25

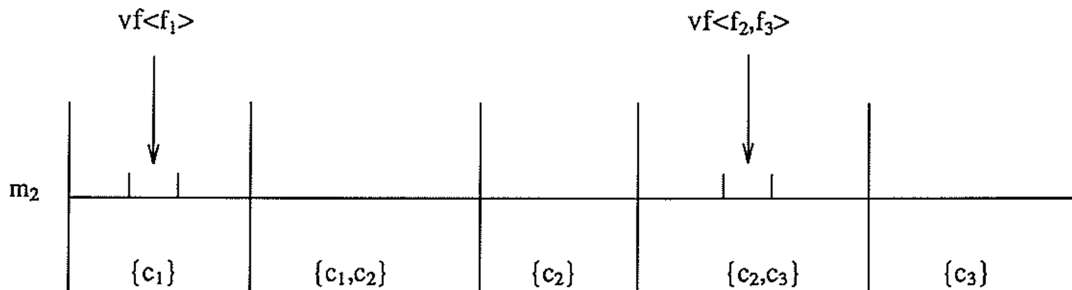


Figure 52: Map unit resulting from the top_level_split

something is wrong with m_2 and that $vf<f_2,f_3>$ should be split.

If $vf<f_2,f_3>$ (in m_2) is split with no direction, one solution is obtained. Unfortunately, that solution is m_1. Thus, a top_level_split with no direction is inadequate to fix m_2. The problem is that when $vf<f_2,f_3>$ is split, the subfragment $vf<f_2>$ combines with $vf<f_1>$ to form $vf<f_1,f_2>$, a fragment that is known to be the result of an incorrect match. Somehow, the splitting of $vf<f_2,f_3>$ must be done in such a way that $vf<f_1,f_2>$ is not formed. The use of direction with the top_level_split accomplishes this.

Remember that in a directed_split, the opposite direction subfragment is not allowed to participate in a combine operation. In this example, the *left* subfragment $vf<f_2>$ must not be allowed to combine. Thus, the split should have *right* directionality. To get the correct map unit, the call is: top_level_split ($vf<f_2,f_3>$,($\{f_2\}$,$\{f_3\}$),RIGHT,m_2). Then add is successful with $vf<f_2>$ and $vf<f_3>$ to produce a single solution, m_3, given in Figure 53.

What is important to note is that the use of direction in top_level_split gives the user finer control over the sequence of operations that occurs. This is helpful when the user has some outside knowledge or intuition about what the final map unit should look like.

4.6. Example of Using top_level_combine

One example of the use of top_level_combine operation is now presented. Suppose Figure 54 represents the underlying reality. In actuality, f_1, f_2 and f_3 are the same fragment, but suppose that due to measurement error, f_1 and f_2 are just beyond the error threshold for matching. Then, fusing with the clone order c_1, c_2 and c_3 produces the Map unit m given in Figure 55. Let g_1 be the group containing $vf<f_1>$, and let g_2 be the group containing $vf<f_2,f_3>$.

Later on, one may suspect that a matching should have been made involving fragments with lengths in the range of f_1, f_2 and f_3. Thus, one might attempt to combine $vf<f_1>$ and $vf<f_2,f_3>$. The initial call is: top_level_combine ($vf<f_1>$,$vf<f_2,f_3>$,NONE,m). Top_level_combine calls undirected_combine, which in turn makes two calls to directed_combine.

The first call to directed_combine has right directionality and occurs at the group g_2. The combined virtual fragment $vf<f_1,f_2,f_3>$ is formed because the three real fragments do have a matching order: f_1, f_3 and f_2. (This is true because f_1 and f_2 are very close to having equivalent lengths.) First, directed_combine attempts to add $vf<f_1,f_2,f_3>$ between g_1 and g_2. The add succeeds because a group with an ACS of $\{c_1,c_2,c_3\}$ exists in that region. This results in the solution given in Figure 31.

The second call to directed_combine has left directionality and occurs at the group g_1. Like the first call, the combined virtual fragment $vf<f_1,f_2,f_3>$ is formed, and the add operation succeeds, resulting

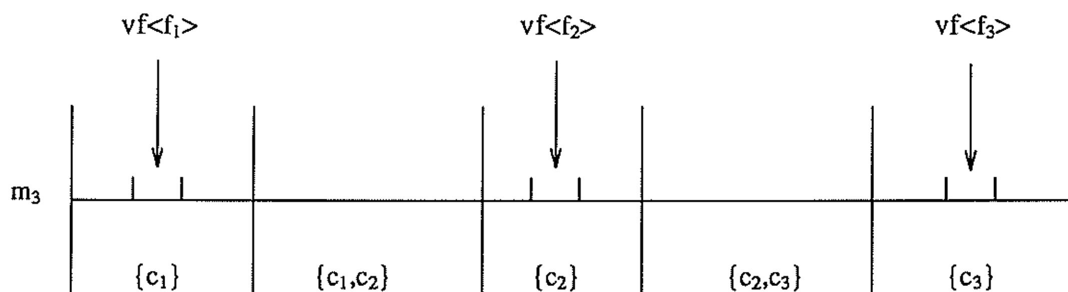


Figure 53: Map unit resulting from the top_level_split with direction

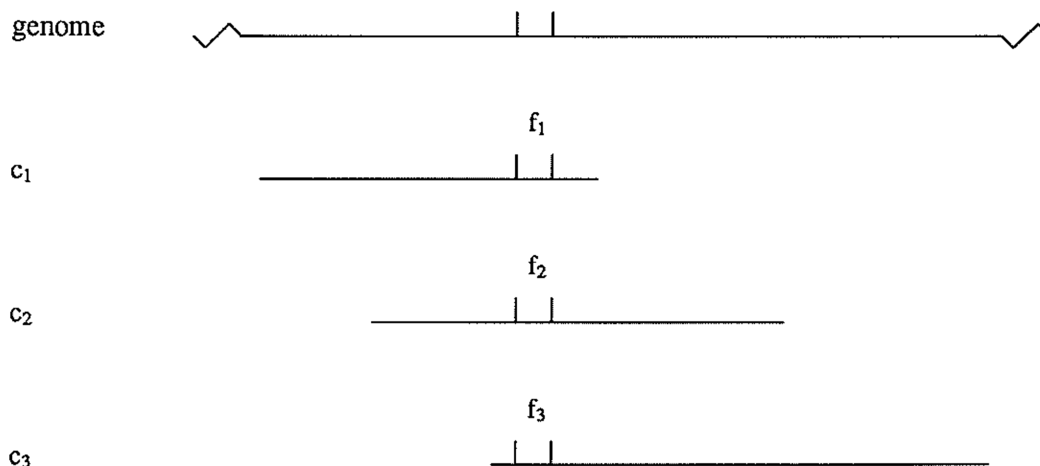


Figure 54: Clone configuration for the top_level_combine example

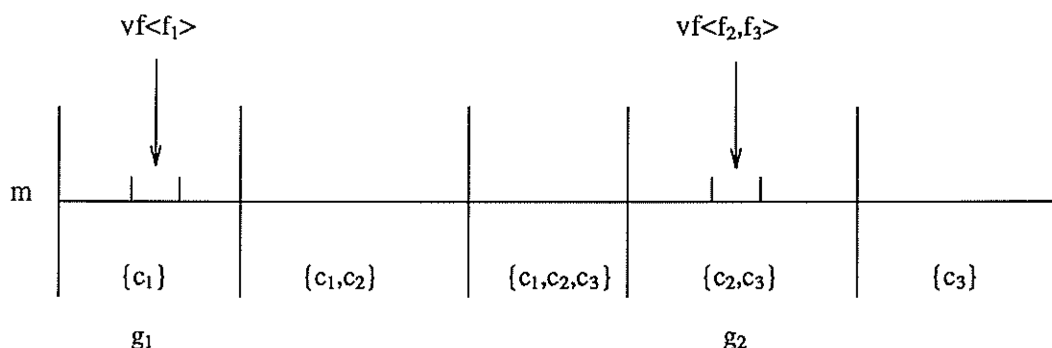


Figure 55: Map unit built from clones in Figure 54

in the solution in Figure 31. So undirected_combine obtains the same solution from each call to directed_combine. It then returns the solution to top_level_combine, which returns it to the user. Note that the solution in Figure 31 corresponds to the underlying reality.

4.7. Examples on Real Map Units

The examples presented in §4.4 and §4.5 are very small examples intended to illustrate certain features of the RSA. The real test for the RSA is to use it on actual DNA restriction site maps. Thus, three examples of its performance on real DNA maps are presented. The maps used in this section were obtained from the yeast mapping project of Dr. Maynard Olson at the Washington University School of Medicine, Department of Genetics.

4.7.1. Map Unit CONTG309

In Figure 56, the Map units CONTG309 and MERG0542 are given. Also displayed is a matchlist that is almost topologically valid. The numbers in Figure 56 are virtual fragment lengths (in base pairs), and the horizontal lines represent group boundaries. In this representation of the map unit, the group at the top is considered to be the "leftmost" group. A solid line going from a virtual fragment in one map unit to

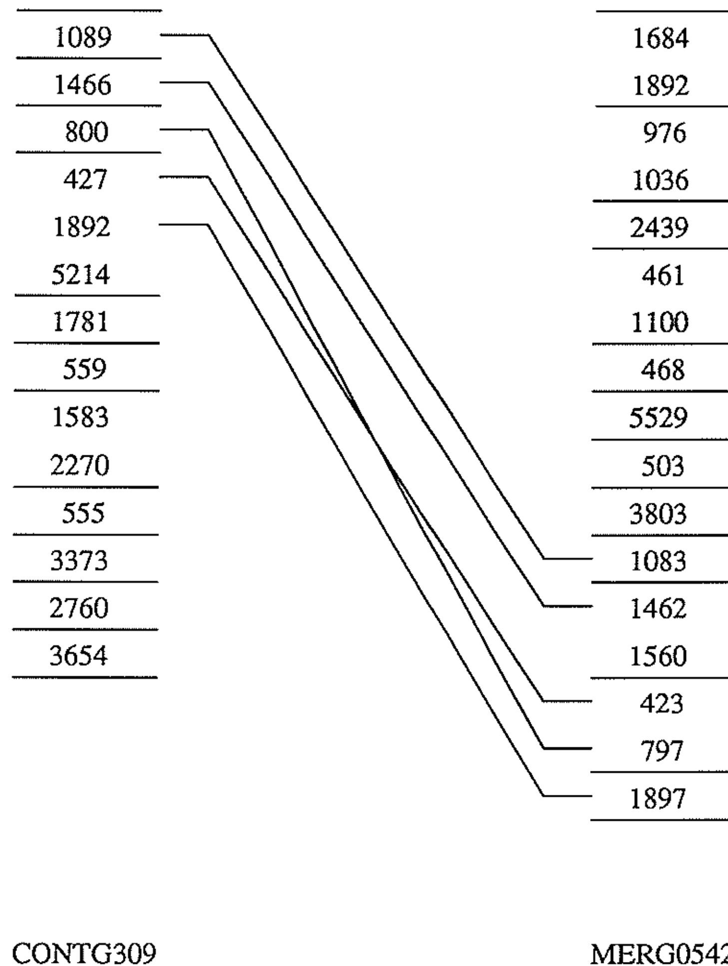


Figure 56: Map units CONTG309, MERG0542 and a matchlist

a virtual fragment in the other map unit represents a match. There is a large amount of apparent overlap between these two map units, but they do not quite fuse.

The ends of the map units fit together well, except for the virtual fragment of length 1560 in MERG0542. There is no virtual fragment with a length in that range at an appropriate spot in CONTG309. However, in the middle of CONTG309, there is a virtual fragment of length 1583 (call this *vf*). The fragment *vf* is about the length that is needed. Thus, splitting *vf* may result in a map unit with a virtual fragment in the 1560 bp range in a proper location.

Instead of guessing which bipartition of the real fragments of *vf* would give the desired result, a top_level_split is made with each possible bipartition. Using this technique, seventeen solutions are returned, three of which fuse with MERG0542. One of those three, CONTG309*, is given in Figure 57 (along with the matchlist that allows the two map units to fuse) and corresponds exactly to the modification that Dr. Olson's Laboratory made to CONTG309 to allow it to fuse with MERG0542.

When the RSA is run with the insert operation disabled (see §4.7.2.), the only solution returned is CONTG309*.

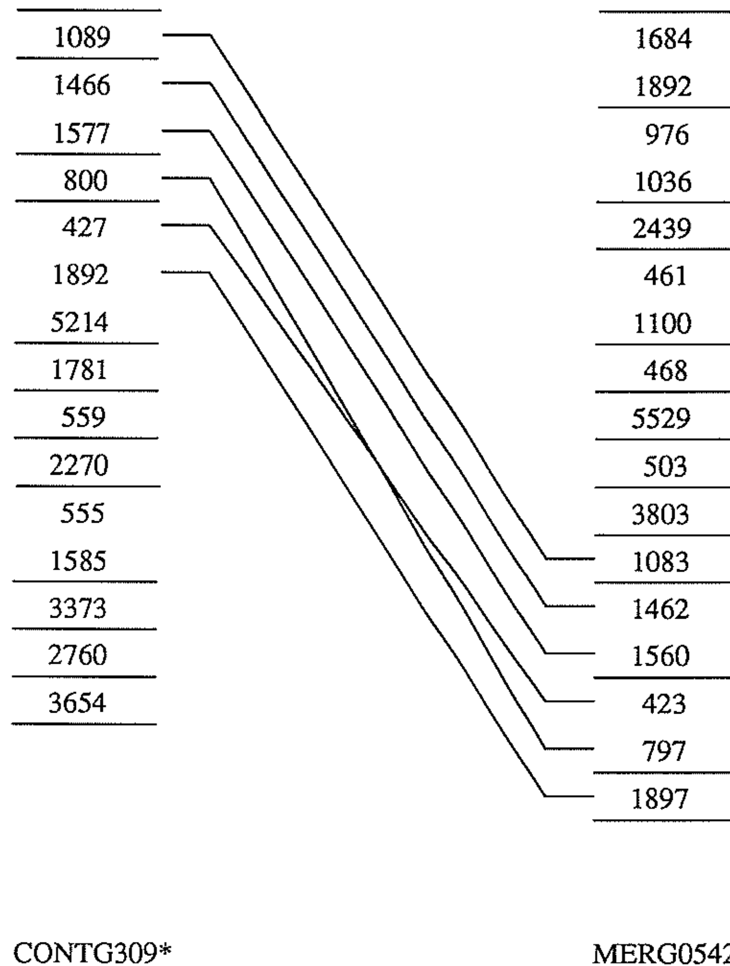


Figure 57: Map units CONTG309*, MERG0542 and a matchlist

4.7.2. Map Unit MERG0515

In Figure 58, the Map units MERG0515 and MERG0411 are given. Again, there is a large amount of apparent overlap, but they do not quite fuse. The problem here is that MERG0515 has no virtual fragment to match with the fragment of length 2163 in MERG0411. (MERG0515 has a fragment of length 2163 but it is not in an appropriate location.) However, near the middle of MERG0515, there is a virtual fragment of length 2174. This fragment is a candidate for splitting.

Calling top_level_split using all bipartitions, the RSA generates nine solutions, two of which fuse with MERG0411. One of those two, MERG0515*, is given in Figure 59 and corresponds exactly to the modification made by the Laboratory. If the insert operation is disabled, then the RSA returns three solutions, of which only one (MERG0515*) fuses with MERG0411.

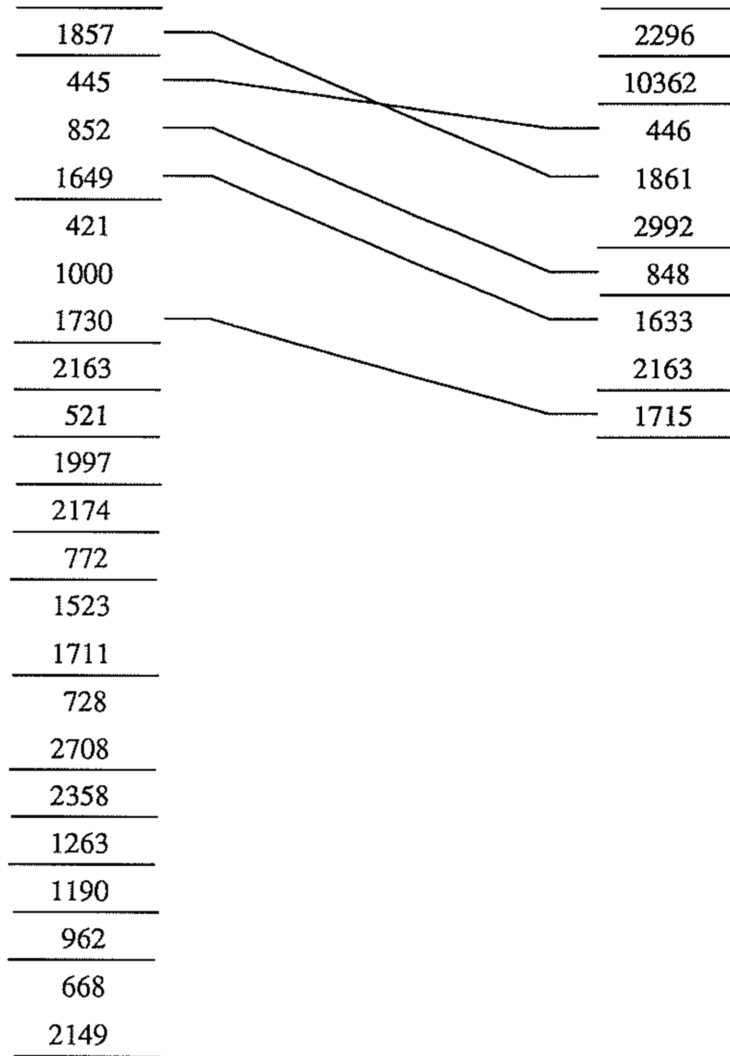


Figure 58: Map units MERG0515, MERG0411 and a matchlist

1857
445
852
1649
2183
421
1000
1730
2163
521
1997
772
1523
1711
728
2170
2708
2358
1263
1190
962
668
2149

MERG0515*

2296
10362
446
1861
2992
848
1633
2163
1715

MERG0411

Figure 59: Map units MERG0515*, MERG0411 and a matchlist

4.7.3. Map Units MERG0615 and MERG0576

In Figure 60, the Map unit MERG0615 and a portion of the Map unit MERG0576 are given. Again, a large amount of apparent overlap exists, but the two map units do not fuse. Here, the problem is a little more complicated than the previous examples. In fact, there are really two problems. The virtual fragment of length 2525 at the top of MERG0576 (call this vf_1) and the virtual fragment of length 2651 in the fifth group from the top of MERG0576 (call this vf_2) have nothing to match with in MERG0615. Fragment vf_1 is composed of one real fragment, and thus a top_level_split is not applicable. However, there is another fragment of nearly the same length further down MERG0576; specifically, it is the virtual fragment of length 2557 (call this vf_3).

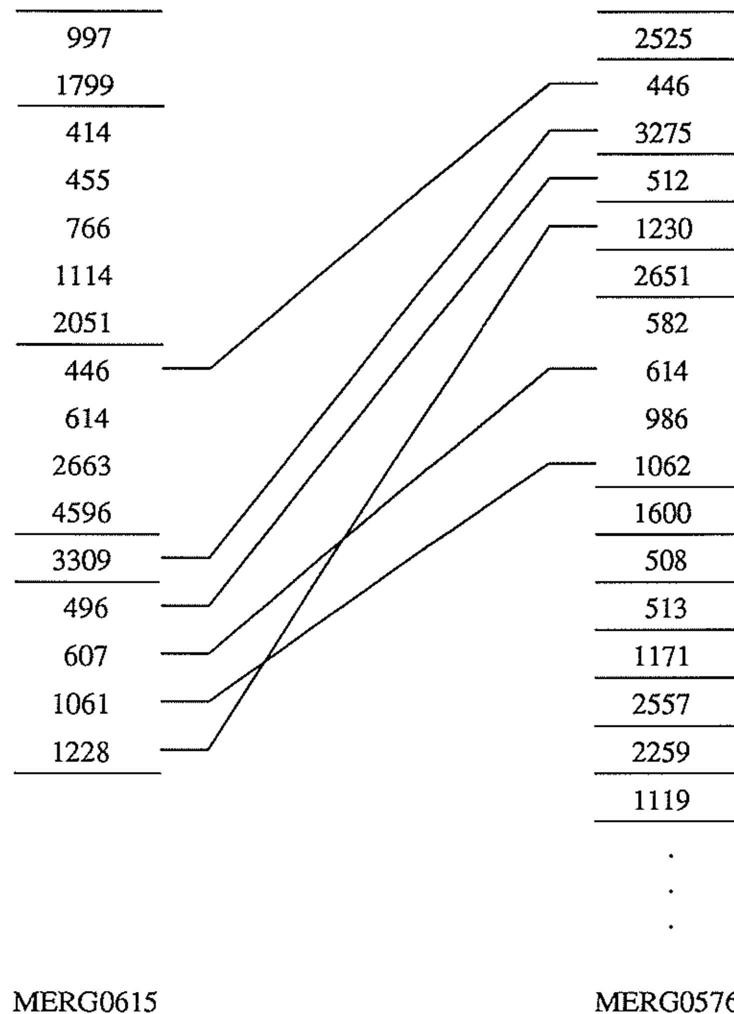


Figure 60: Map units MERG0615, MERG0576 and a matchlist

If vf_1 and vf_3 successfully combine, then vf_1 will be taken out of the map unit and hopefully the combined virtual fragment will be placed in a location that allows more clones to fuse. A top_level_combine of vf_1 and vf_3, with no direction, succeeds and the Map unit MERG0576*, given in Figure 61, is the only solution returned. Also given in Figure 61 is a new matchlist, and one can see that the problem with vf_2 still exists.

A top_level_split of vf_2 might fix the problem, but it will create a left subfragment that must be placed somewhere to the left of (i.e., above) the group containing vf_2. This subfragment will probably interfere with fusion in the same way that vf_2 does. So maybe the problem lies within MERG0615. In the middle of MERG0615, there is a virtual fragment of length 2663 (call this vf_4). If vf_4 is split, its right subfragment may be placed in a position where it can match with vf_2. Calling top_level_split on vf_4 using all bipartitions and with the insert operation disabled results in a single solution, MERG0615*, which is given in Figure 62. A matchlist between MERG0615* and MERG0576* that is topologically valid is also given in Figure 62. Using this matchlist, MERG0615* and MERG0576* successfully fuse. The new map unit obtained corresponds to the one produced by the Laboratory.

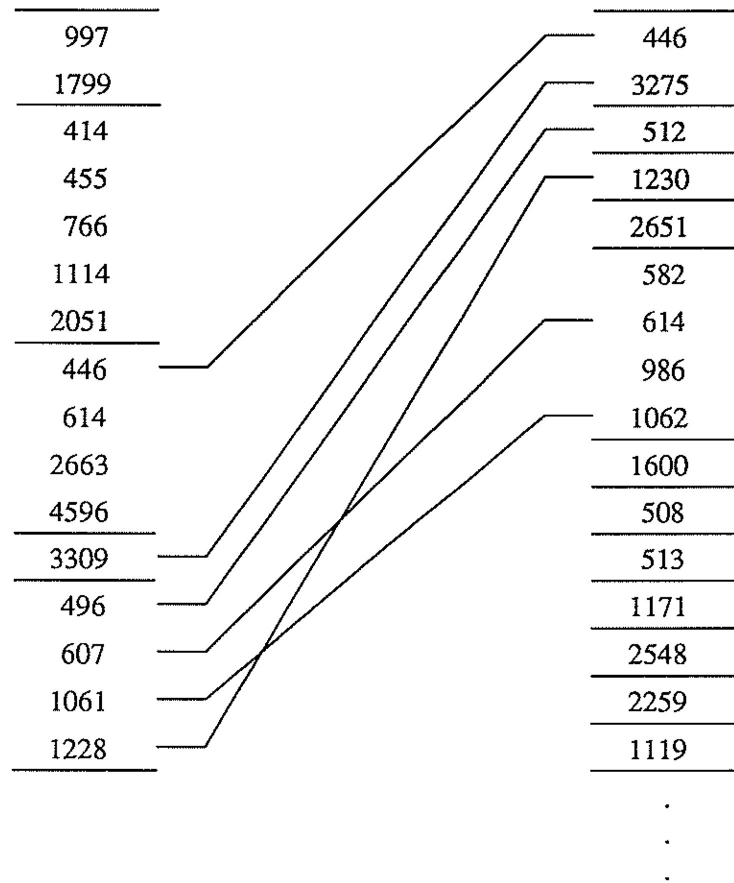


Figure 61: Map units MERG0615, MERG0576* and a matchlist

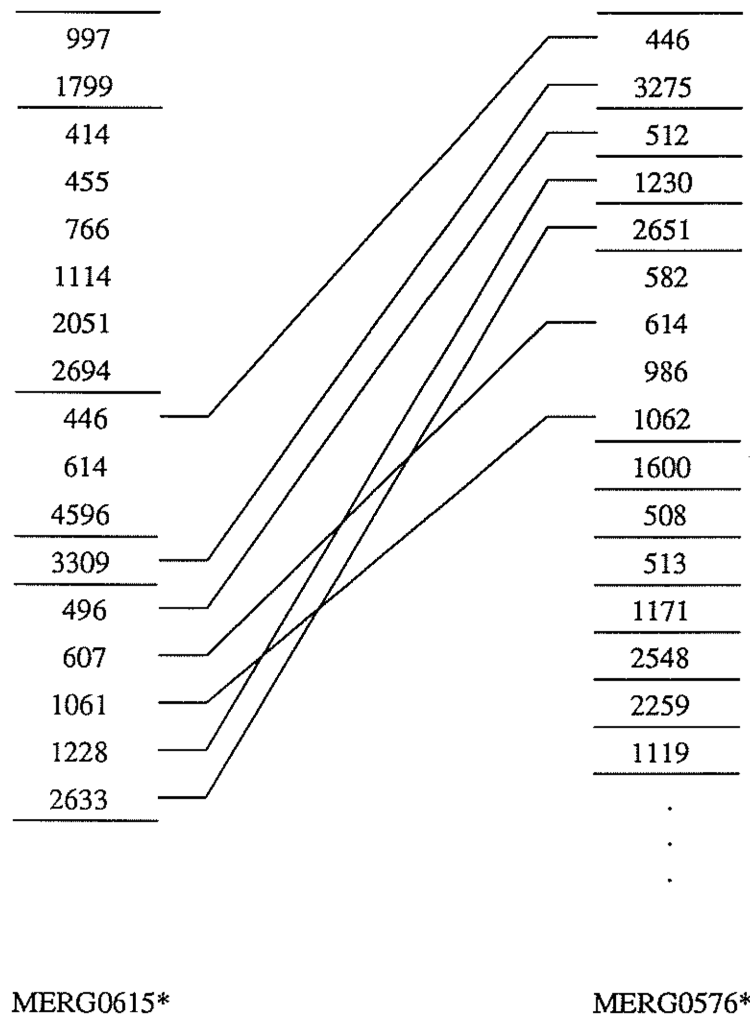


Figure 62: Map units MERG0615*, MERG0576* and a matchlist

4.8. Implementation Considerations

The psuedocode of the operations is intended to describe how the operations behave in a conceptual manner, so that the reader knows what kind of solutions one should get when using them. In the actual implementation of these operations (which is in the C language), the code is different than is indicated by the psuedocode. There are statements in the C code and are not in the psuedocode that have less to do with fragment splitting and more to do with software engineering conventions that exist in the DNA Mapping software as a whole. Some differences that are more relevant to fragment splitting will now be discussed.

4.8.1. Bipartitions of Clones

In the psuedocode, a bipartition is represented as a pair of sets of real fragments. It was stated earlier that using bipartition of clones is equivalent to using bipartition of real fragments. In fact, the implementation does not use real fragments. Instead, a character string, which is the internal name of the clone that a real fragment comes from, is used. This is done to increase efficiency.

4.8.2. Actions

In the psuedocode, changes to the map unit (such as adding a virtual fragment to a group) are made during the operation. Then the modified map unit(s) is returned to the calling operation. This means that many copies of the original map unit (each with small modifications) might exist. This approach consumes quite a bit of memory space if the original map unit is large.

In the GSA, it is difficult to do anything else. However, in the RSA, there is a way of greatly reducing the space requirements. The basic idea is to construct a list of **actions** that indicate what changes need to be made to the original map unit, instead of making a new copy of the original map unit and actually making the changes. Any solution map can be constructed from the original map unit through the use of three types of actions: ADD, INSERT and REMOVE. Keep in mind that these actions (as used here) are data structures, not functions.

ADD(vf) indicates that the virtual fragment vf should be placed in a group in the map unit. Since there is at most one group in the map unit that vf can legally be placed in, it is not necessary to store the group in the action. However, depending on the nature of the implementation, it may be more efficient to do so. REMOVE(vf) indicates that the virtual fragment vf should be removed from the map unit. If a virtual fragment can be uniquely identified without specifiying the group it is contained in, then it is not necessary to store the group in the action. Again, it may be more efficient to do so. INSERT(vf, g) indicates that a new group should be created immediately to the right of the existing group g in the map unit. Then, the virtual fragment vf should be placed in the new group. Depending on the nature of the implementation, one may need a special code to indicate when the insertion should occur at the end of the map unit. However, if clone ends are explicitly stored in the map unit, then this should not be necessary.

All the operations (add, insert, and the various splits and combines) simply pass the original map unit and refer to it. Whenever an operation must make a modification to the original map unit, it creates the appropriate action (or list of actions). Thus, the operations really do not return maps. Instead, they return lists of actions, where each list contains the modifications to the original map unit that produce the desired map unit. The responsibility for actually creating the new map units from the actions is given to top_level_split and top_level_combine. After collecting the lists of actions, each is applied to the original map unit to generate a solution. With this mechanism, the number of copies of the original map unit is kept to a minimum.

One may question whether the RSA works correctly if all operations refer to the original map unit. The reason it still works is that an operation never refers to a portion of the map unit modified by the operations before it. A closer examination reveals that this is true. A group is **affected** if an operation in the current sequence has created an action that pertains to that group. (I.e., that group has been "modified".)

The directed_split, undirected_split and undirected_combine are called before any actions are created. So no groups in the map unit are affected, and these operations can legally refer to the original map unit.

Some actions are created by the time a directed_combine is called, when it is called by an internal_split. Suppose an internal_split occuring at a group g_1 creates a subfragment vf_1 and then attempts a directed_combine with another virtual fragment vf_2, which is in the group g_2. This situation is illustrated in Figure 63. The combined virtual fragment vf_3 is formed. First, directed_combine attempts to add or insert vf_3 between groups g_1 and g_2. Although g_1 and g_2 are obviously affected because of the internal_split, the groups in between are not affected. Thus, directed_combine can refer to the original map unit when attempting to add or insert. If the add and insert fail, then directed_combine calls internal_split on vf_3. Thus, the question now is whether this second internal_split can legally refer to the original map unit.

Let vf_{opp} be the opposite direction subfragment, and let vf_{same} be the same direction subfragment created by the internal_split. In order for the internal_split to succeed, add or insert must succeed with vf_{opp} between g_1 and g_2. However, it has already been demonstrated that the groups in that region of the

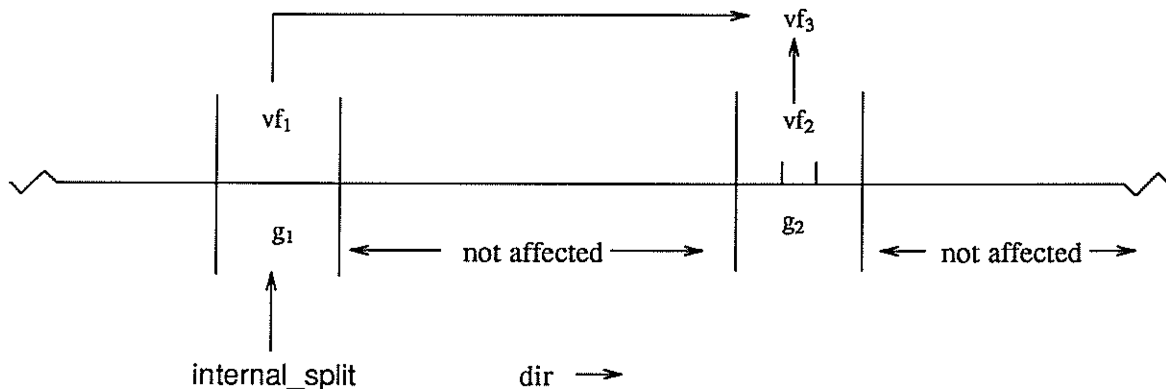


Figure 63: Regions of a map unit not affected

map unit are not affected. In addition, directed_combine, add or insert must succeed with vf_{same} beyond g_2. However, note that g_2 is the furthest affected group, and thus groups beyond it are not affected. Thus, internal_split can legally refer to the original map unit.

The add and insert operations can refer to the original map unit as long as the range of groups they are called upon are not affected. It has been shown that directed_combine and internal_split call them appropriately, and it is easy to see that directed_split, undirected_split and undirected_combine call them appropriately as well.

4.8.3. User Options

There are some options available in the implementation of the RSA that are not expressed in the psuedocode, in order to keep it as simple as possible. These options will be discussed in this section. The options are (1) attempting all bipartitions, (2) disabling the insert operation, and (3) requiring a unique solution.

The first option is concerned with the bipartitions used to decompose virtual fragments into subfragments. In the psuedocode for top_level_split, the user is required to give the bipartition by which the virtual fragment should be split. However, in a realistic mapping situation, it can be difficult for the user to see which bipartitions will lead to the desired results. The user may not even have a clear idea of what the desired results are. The user may merely know that a particular fragment seems out of place and suspects a top_level_split may be in order. Consequently, in the implementation, one can specify a particular bipartition to top_level_split, or one can ask that each possible bipartition of the real fragments composing the virtual fragment be attempted.

The second option concerns the insert operation. In map units where the number of real fragments in a typical virtual fragment is high, it is relatively easy for insert operations to succeed. This leads to a large number of solutions. Although there are times when the insert operation is necessary to generate the correct solution, one may favor a solution that does not require insert operations. Obtaining a solution that does not require the insert operation indicates that the group structure of the original map unit is correct, and only a rearrangement of fragments is necessary. Obtaining a solution that does require the insert operation indicates that the group structure of the original map unit is slightly incorrect. Thus, one may favor solutions not requiring the insert operation, because it implies less of a departure from what was considered to be the true situation. Therefore, in the implementation, the top_level_split and top_level_combine operations have a Boolean parameter that allows the user to specify whether or not insert operations are to be used. The implementation of this option is fairly easy. Statements are added that check the Boolean parameter any time an insert operation is about to be attempted.

The third option concerns the number of solutions returned. If a top_level_split or top_level_combine operation returns more than one solution, the user must select which one to use, or discard all of them. However, the user may know beforehand that all solutions will be discarded if more than one is returned. Given this, it would be convenient if the algorithm stopped once more than one solution has been found. This is particularly time-saving when one is using all bipartitions of the virtual fragment in the initial split (i.e., the first option). Consequently, in the implementation, top_level_split and top_level_combine have a Boolean parameter that specifies whether or not a unique answer is desired. If this is set to TRUE, then the algorithm stops and returns with failure once more than one solution has been found.

In Figure 64, the header of the top_level_split operation (called `fs_split` in the C code) in the implementation is given. In Figure 65, the header of the top_level_combine (called `fs_combine`) is given.

5. Conclusion

In this report, four basic operations on virtual fragments (add, insert, split and combine) are defined and form the basis for the GSA, an algorithm that computes all the possible map units that result from the correction of a fragment matching mistake. However, the GSA is difficult to implement and does not promise to be of much use in constructing DNA maps. Thus, the operations of the GSA are modified to give new operations that form the basis for the RSA, an algorithm that computes only the most compact map units that result from the correction of a fragment matching mistake. The RSA is implementable and has the potential to be a useful aide in constructing DNA maps.

However, the RSA must be told the virtual fragment to split, or which two fragments to combine. Therefore, it depends on human intuition or on other software. A future report will discuss the automatic detection of fragments to split and fragments to combine.

Some related topics that may be the focus of future work are (1) using the RSA to identify mistakes in the interpretation of electropheresis gels, (2) developing strategies so that map units containing more than one independent matching mistake can be corrected, (3) adjusting the operations in the RSA so that it

```
fs_split(vf_node,bipart,mu,dir,unique,inserting)
   TREE_TYPE        vf_node;
   PAIR_TYPE        bipart;
   TREE_TYPE        mu;
   DIRECTION_TYPE   dir;
   BOOLEAN          unique,
                    inserting;
```

Figure 64: Header in the implementation of the top_level_split operation

```
fs_combine(vf_node1,vf_node2,mu,dir,unique,inserting)
   TREE_TYPE        vf_node1,
                    vf_node2,
                    mu;
   DIRECTION_TYPE   dir;
   BOOLEAN          unique,
                    inserting;
```

Figure 65: Header in the implementation of the top_level_combine operation

finds the correct solution more frequently and (4) extending the RSA to work on a broader range of map units (e.g., map units where two or more distinct groups with the same ACS exist).

APPENDIX A

Description of Functions and Procedures in the Psuedocode

```
SET
acs(x)
        VIRTUAL_FRAGMENT x;
or
        GROUP           x;
```

Returns the active clone set of the object x.

```
LIST
concatenate_left_and_right_parts_of_maps(l1,l2,vf,g)
        LIST    l1,l2;
        VIRTUAL_FRAGMENT vf;
        GROUP           g;
```

Returns the list of map units generated using the process illustrated in Figure 35. The portion of each map in l_1 to the left and of including g is paired with the portion of each map in l_2 to the right of g. The virtual fragment vf is removed from the maps constructed before returning.

```
VOID
create_new_group_right_of(g)
        GROUP           g;
```

Creates a new group immediately to the right of g in the map unit that g is contained in. Nothing is returned.

```
LIST
combining_vfs(vf,g1,g2)
        VIRTUAL_FRAGMENT vf;
        GROUP           g1,g2;
```

Returns a list of the virtual fragments, in the map unit containing g_1 and g_2, that satisfy the conditions p_1', p_2' and p_3' with vf. In addition, these virtual fragments must be in groups that are between g_1 and g_2.

```
GROUP
group_of(vf)
        VIRTUAL_FRAGMENT vf;
```

Returns the group that contains vf.

```
GROUP
group_right_of(g)
        GROUP           g;
```

Returns the group, in the map unit containing g, that is immediately to the right of g.

```
GROUP
group_with_acs(s,m)
        SET     s;
        MAP_UNIT        m;
```

Returns the group in m that has the active clone set s.

```
VIRTUAL_FRAGMENT
leftmost_frag_of(vf1,vf2)
        VIRTUAL_FRAGMENT vf1,vf2;
```

Assuming vf_1 and vf_2 are contained in the same map unit but not in the same group, vf_1 is returned if it is to the left of vf_2, and vf_2 is returned otherwise.

```
GROUP
leftmost_group_of(g1,g2)
        GROUP         g1,g2;
```

Assuming g_1 and g_2 are distinct and are contained in the same map unit, g_1 is returned if it is to the left of g_2, and g_2 is returned otherwise.

```
VOID
remove_equivalent_maps(l)
        LIST    l;
```

Duplicate map units in the given list of map units l are eliminated. Nothing is returned.

```
VOID
remove_vf(vf,m)
        VIRTUAL_FRAGMENT vf;
        MAP_UNIT       m;
```

Removes vf from the map unit m. Nothing is returned.

```
BOOLEAN
rfs_have_matching_order(l)
        LIST    l;
```

Given a list l of real fragments, TRUE is returned if the real fragments can match, otherwise FALSE is returned.

```
SET
rfs_of(vf)
        VIRTUAL_FRAGMENT vf;
```

Returns the set of real fragments that compose vf.

```
VIRTUAL_FRAGMENT
rightmost_frag_of(vf1,vf2)
        VIRTUAL_FRAGMENT vf1,vf2;
```

Assuming vf_1 and vf_2 are contained in the same map unit but not in the same group, vf_1 is returned if it is to the right of vf_2, and vf_2 is returned otherwise.

```
GROUP
rightmost_group_of(g1,g2)
        GROUP         g1,g2;
```

Assuming g_1 and g_2 are distinct and are contained in the same map unit, g_1 is returned if it is to the right of g_2, and g_2 is returned otherwise.

```
PAIR
swap(p)
        PAIR    p;
```

Returns a new pair whose left element is the right element of p and whose right element is the left element of p.

```
VIRTUAL_FRAGMENT
vf_combine(vf1,vf2)
        VIRTUAL_FRAGMENT vf1,vf2;
```

Returns a virtual fragment created by merging the real fragments composing vf_1 with the real fragments composing vf_2.

```
PAIR
vf_split(vf,p)
        VIRTUAL_FRAGMENT vf;
        PAIR    p;
```

Given a virtual fragment vf and a pair of sets of real fragments p, two virtual fragments are created by breaking up the real fragments in vf as determined by p. The two virtual fragments are placed in a pair and returned.

```
BOOLEAN
vfs_might_combine(vf1,vf2,g1,g2,m)
        VIRTUAL_FRAGMENT vf1,vf2;
        GROUP           g1,g2;
        MAP_UNIT        m;
```

TRUE is returned if the conditions p_1', p_2' and p_3' are satisfied, otherwise FALSE is returned. vf_1 and vf_2 are the virtual fragments to combine. The group g_1 is the group that vf_1 is from or where the split creating vf_1 occured. The group g_2 is defined similarly for vf_2. All groups and fragments are contained in the map unit m.

```
SET
vfs_of(g)
        GROUP           g;
```

Returns the set of virtual fragments contained in g.