Report Number: WUCS-91-23

1991-03-01

DNA Mapping Algorithms: Topological Mapping

Kenneth Moorman, Paul Poulosky, and Will Gillett

There are several basic approaches that can be used in attempting to produce high-resolution DNA restriction maps. A standard approach is the match/merge approach in which first the topology of the map units being mapped together is suppressed and lists of potential matches between fragments are generated, and second the topology is introduced to eliminate matchlists which are inconsistent with the topology. This technical report documents a different approach to DNA mapping, known as topological mapping. In topological mapping the precedence of the two criteria are reversed, i.e., the topology of the two map units is used as the... Read complete abstract on page 2.

DNA Mapping Algorithms: Topological Mapping

Kenneth Moorman, Paul Poulosky, and Will Gillett

Complete Abstract:

There are several basic approaches that can be used in attempting to produce high-resolution DNA restriction maps. A standard approach is the match/merge approach in which first the topology of the map units being mapped together is suppressed and lists of potential matches between fragments are generated, and second the topology is introduced to eliminate matchlists which are inconsistent with the topology. This technical report documents a different approach to DNA mapping, known as topological mapping. In topological mapping the precedence of the two criteria are reversed, i.e., the topology of the two map units is used as the primary search constraint and only those fragments within specific topologically constrained bounds are considered for a potential match. In this approach, the primary topological constraint reduces the number of fragment comparisons that must be considered, in comparison to the match/merge approach. The more topological structure is present in each map unit, the greater the reduction. The conceptual approach is discussed in general. Specific mechanisms for implementing the ideas are presented, and the structure of the software based in these mechanisms is described. Heuristic approaches which can be used for pre- and post-processing are suggested and analyzed. Execution results are given for the application of the software to a number of laboratory generated mapping problems, which were provided by the Olson laboratory.

DNA Mapping Algorithms: Topological Mapping

Kenneth Moorman
Paul Poulosky
Will Gillett

WUCS-91-23

March 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 the National Science Foundation under Grant CDA-9000823.

ABSTRACT

There are several basic approaches that can be used in attempting to produce high-resolution DNA restriction maps. A standard approach is the match/merge approach in which first the topology of the map units being mapped together is suppressed and lists of potential matches between fragments are generated, and second the topology is introduced to eliminate matchlists which are inconsistent with the topology. This technical report documents a different approach to DNA Mapping, known as topological mapping.

In topological mapping the precedence of the two criteria are reversed, i.e., the topology of the two map units is used as the primary search constraint and only those fragments within specific topologically constrained bounds are considered for a potential match. In this approach, the primary topological constraint reduces the number of fragment comparisons that must be considered, in comparison to the match/merge approach. The more topological structure is present in each map unit, the greater the reduction.

The conceptual approach is discussed in general. Specific mechanisms for implementing the ideas are presented, and the structure of the software based in these mechanisms is described. Heuristic approaches which can be used for pre- and post-processing are suggested and analyzed. Execution results are given for the application of the software to a number of laboritory generated mapping problems, which were provided by the Olson laboratory.

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF TABLES

1. Introduction

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 lengths of these fragments, which are known as **restriction fragments**. If the order of these restriction fragments within the original DNA can be determined, then the **restriction map** can be constructed.

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 (during electrophoresis), experimental error (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.

This technical report describes one basic DNA restriction mapping algorithm, known as **topological mapping**. This technique uses the **topology** (or structure) of the **map units** (preliminary partial restriction maps) being considered as the primary criterion for searching for a solution; this is in contrast to other techniques in which the topology is originally suppressed, and the possible pairings between restriction fragments in the map units being considered is used as the primary criterion for searching for a solution.

The technical report is organized in the following manner. Section 2 presents the basic approach and set of working assumptions used in all restriction mapping techniques. A number of examples are given, and the overall strategy is explained intuitively and by example. Section 3 extends these basic ideas by presenting a more concrete methodology of how the process of restriction mapping might be performed. Intrinsic problems with restriction mapping and specific aspects of the restriction mapping process are discussed. Section 4 gives an overview of the philosophy and structure of the basic topological mapping software that has been developed. High-level pseudocode is given, and different modes of interacting with

the software are discussed. Section 5 continues the discussion of the software by focusing on concrete data structures and pragmatic issues dealing with the quality of the solution found and the efficiency by which it was found. Section 6 presents a number of top-level strategies and heuristics that can be layered on top of the basic topological mapping software. Different techniques are compared and contrasted. Section 7 presents some results of applying the topological mapping software to a variety of different data sets. Finally, Section 8 attempts to put this work in perspective.

2. Handmapping of DNA

In order to understand a computer solution to DNA restriction mapping, it is necessary to understand a human approach to the problem. The process that humans use to create a DNA map will be referred to as **handmapping.**

2.1. Clones

The type of data considered in handmapping is the fingerprint clone data. The following is a more in-depth discussion of the process described in the introduction. Prior to any mapping, the original DNA sequence 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. This means that any region of DNA 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 con-

tain 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 handmapping 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.

The final stage of the data gathering process is to convert the position of the fragments along the gel into actual numeric data. This will result in a list of integers, where each integer represents the length of a restriction fragment (in number of base pairs) from the clone. The current method of converting the data from gel form to this list of fragment lengths is through the use of a digitizing tablet. A gel image is projected onto the tablet, and a human operator touches the tablet at the positions of the bands corresponding to fragment locations. The incoming data are displayed on the computer screen in real-time to allow for

error detection and editing.

Once the fragment position data are available, it is possible to calculate the lengths of the fragments by interpolation. These length data are what will be used in the process of DNA restriction mapping. It is impossible to be 100% certain of the data that are produced. There are (at least) two sources of error which create this uncertainty. First, at the current state of the technology for transferring the data from gel to numeric form, along with the chance of human error in the process, the exact fragment lengths cannot be determined. 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. It is thought this error window will shrink in the near future, due to better data gathering techniques and the elimination of some of the human involvement. This will be accomplished by utilizing a CCD digitizing camera to photograph the gel, eliminating the human/digitizing tablet combination. For now, however, the 3% error window must be kept in mind while mapping.

The other source of uncertainty comes from incorrectly extracting the fragment data from the gel. Two fragments of identical or nearly identical length may 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 contain a number of lengths which does 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 should be remembered while doing DNA mapping, either by hand or on the computer. 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.

2.2. Mapping two clones

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 a clone with fragments of length:

5000

4000

3000

2000

1000

and another with fragments of length:

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 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 they actually overlap, since two fragments of the same length are not necessarily the same fragment. One of the ways that this is taken into account while mapping is to require more than simply a one fragment 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 either three or four. This increases the probability that the two

clones about to be mapped are actually from the same region and should map together.

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.

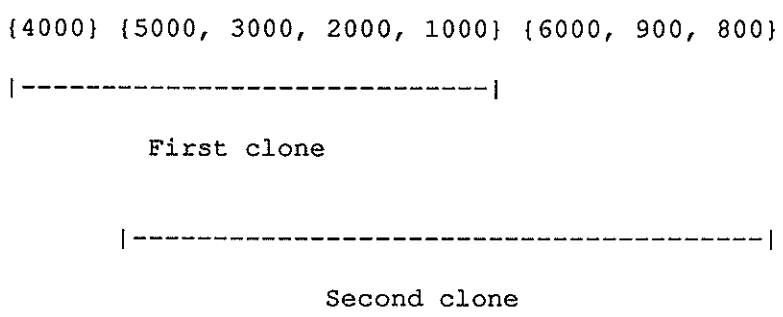
The four fragments which overlap must also be contiguous. This means that each clone can be divided into two sets, one set containing the fragments which overlap and the other set containing all the remaining fragments in the clone. In the first clone, these two sets are:

{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 side of the first clone. Similarly, there is a restriction site 7,700 (6000 + 900 + 800) bp in from the other side 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. It is also known how the three sets are positioned in relation to each other. It is not known, though, what the exact ordering of the fragments is in any one of the sets. To gain a higher level of refinement, more clones would need to be added to the map.

The previous example is a contrived 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.

2.3. Mapping a set of clones

The human DNA mapper would generally have a set of clones which are suspected of coming from a certain region of the genome being mapped. Figure 1 presents such a set of clones. The fragment lengths of each clone are sorted from longest to shortest, but this is for convenience only. At this stage, no knowledge about the ordering the fragments in any of the clones is known to the mapper. There are five clones, with the number of fragments ranging from five to eight.

The first consideration is to determine with which two clones the mapping should be started. 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 a mapper may follow; one of the easiest ones is to make initial choices 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.

There are several ways to approach a clone-clone mapping. The easiest way is just to start at the top of the two clones and begin to scan downward for matches. Please note at this point that "top" is used for the convenience of the human reader. It refers *only* to the way the clone data are being represented on

#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

paper. There is no direction associated with the data other than this representation. The top of a clone will always be the fragment with the longest length. Keeping this in mind, the first match discovered would be 6198—6109. Although not the same length, the two fragments are within the 3% error window. So, there is a chance that they are the same fragment.

After creating a match, neither fragment is available for subsequent matches. Having paired 6198 with 6109, the process of scanning for matches continues down 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. So, 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. Then, 637 is matched with 630. There is now just 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 an **equivalent match**. Note that the term *equivalent* as used here should not be confused with the term as used in the mathematical definition of *equivalence relation* or *equivalence class*. While the reflexive and symmetric properties do hold for this usage, the transitive property does not hold. *Equivalent match*, in this paper, simply means that since the only data being worked with are length data and a 3% error window exists, two such matches are (in most senses) equally valid. A choice must, however, be made. The procedure to follow is to choose the "better" match of the two. The problem lies in determining exactly what better means. Better is defined in this case to mean "a smaller number of base pairs separating the two lengths." Using this metric, the 513—515 match is better since

there is just a two base pair difference in the lengths. On the other hand, 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** which describes the matches which exist between the two clones. It is also known which fragments in each 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 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 units used to produce them. It is important to note that it is always possible to "pick out" the individual clones which exist in a map unit, as Figure 2 illustrates, because the fragments present in a clone must always be contiguous in a map unit.

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.

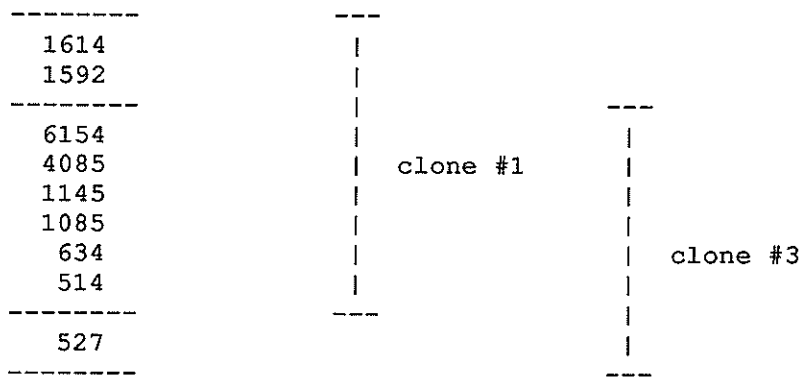


Figure 2: Map unit produced from mapping clone #1 and #3

Now that the first two clones are mapped together, 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 next, so the mapping now being attempted is the map unit just produced (Figure 2) with Clone #4. This mapping will not be presented in as much detail, but the same ideas as previously presented are being followed.

The 1614 and 1600 match, as do the 1592 and 1573. This is another case of equivalent matches, since 1614—1573 and 1600—1581 are also valid matches. Once again, the better set of matches is chosen; in this case that is 1614—1600 and 1592—1573 (a 14 bp and a 19 bp length difference as opposed to a 41 bp and a 19 bp difference). 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. There are still the equivalent matches involving the fragments with lengths in the 1500 to 1700 bp range to consider, but once again, the better match is chosen. Figure 4 shows the current state of the map unit being produced from the original set of data.

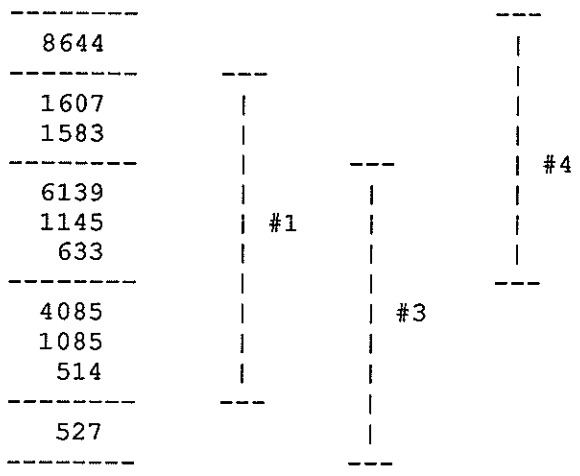


Figure 3: Clones 1, 3, and 4

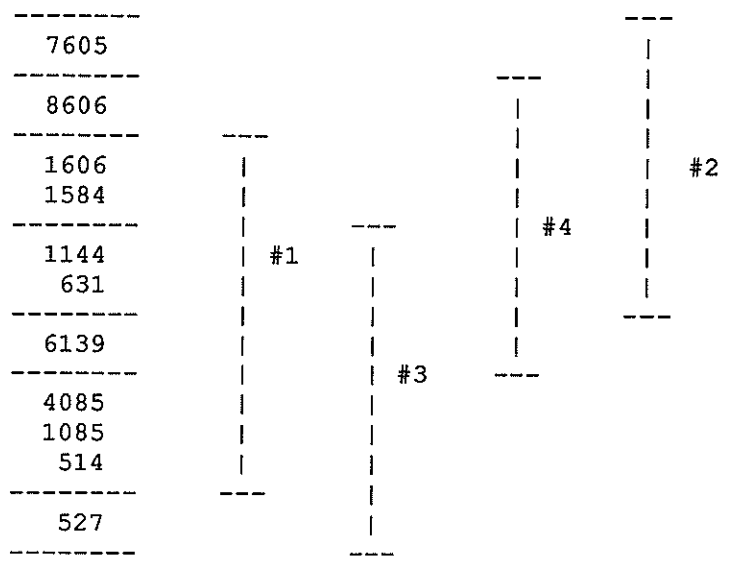


Figure 4: Clones 1, 2, 3, and 4

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.

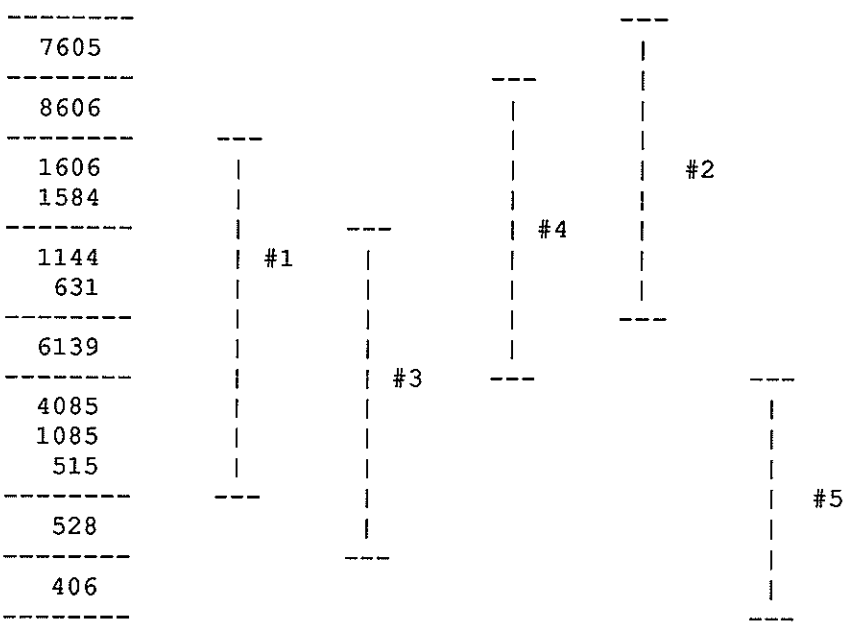


Figure 5: Completed mapping of clone set

This example was complex enough to demonstrate several key facts about the handmapping process. Most importantly, it showed that equivalent matches are a problem with which the algorithm must deal. Also, it showed when a mapping is finished, which is when all the fragments have been considered and are either matched with something or are found to be incapable of matching with anything. Plus, it illustrated the process of gradually building towards a final answer. All of these ideas must be considered when developing a computer algorithm.

2.4. Final thoughts on handmapping

At first glance, handmapping 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 make the procedure a difficult one to automate effectively. It is possible to do this, though. In fact, several algorithms exist which can be used to map DNA. With a background knowledge of how handmapping is done, it is now time to examine one of these algorithms in depth, specifically the topological mapping algorithm.

3. Topological mapping

This section presents the top-level ideas embodied in the topological mapping algorithm. The name "topological mapping" stresses the fact that this algorithm uses the topology of the items being mapped as an aid to the overall mapping process. As a contrast, another major algorithm is the **match/merge** algorithm. The match/merge algorithm initially disregards the structural information present in the two starting map units, obtaining a preliminary solution based solely on how fragments in the two map units might be paired with one another (given specific error ranges). Subsequently, it then reintroduces the structural information to determine if the topology is consistent with this preliminary solution. Because of this, it is possible for the match/merge algorithm to consider a preliminary solution which is inconsistent with the topology of the map units being mapped. Such a preliminary solution would then have to be discarded. Topological mapping, on the other hand, is primarily concerned with the map units' topological structures. It will only create mappings which are valid within this framework.

3.1. Topology as an aid in restricting searches

There are certain advantages to considering topology as a primary concern in a mapping attempt. One large benefit comes from using the structure to reduce the amount of work which the algorithm must perform. This **limiting property** is what makes topological mapping such a useful technique. The best way to understand how this works is to see an example, as presented in Figure 6. This shows a clone and a map unit which are to be mapped together. The map unit currently contains six divisions of fragment length data, while the clone contains only one.

The mapping begins as all the examples have, by starting at the top of the two units and scanning downward until a match is found. Preceding in this manner, the 944 of the map unit matches with the 944 of the clone. At this preliminary point, there is no difference between topological mapping and the type of

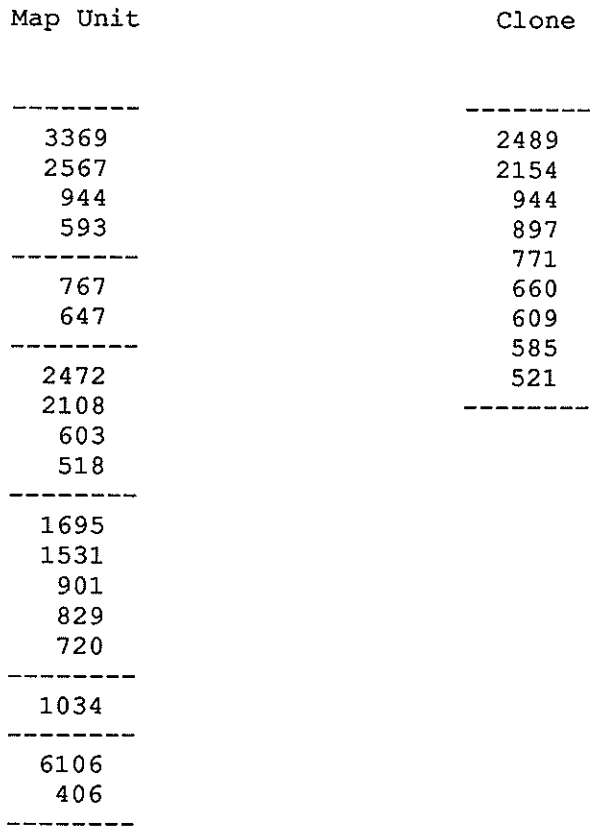


Figure 6: Example of topology's usefulness

mapping presented in the earlier cases. It is after this first match is made that topology can become useful.

If there are any more valid matches, then there must be a valid match in either the top set of the left map unit or the next set down. At first this may not be obvious, but its validity can be shown. The best way to illustrate this is by contradiction. Suppose there are no valid pairings involving the fragment lengths in the second set from the top in the first map unit. However, assume valid matches do exist between the clone data and data from the first, third, fourth, and fifth sets of the map unit. This mapping would be invalid. This is due to the fact that all the fragments from a given unit (whether this is a set, clone, or entire map unit) are known to be contiguous. The first set of the map unit is "next to" the second set which is "next to" the third set and so on. Similarly, all the fragments of the clone are contiguous, in some as yet undetermined order. If a clone is mapped into the map unit and there exists an unmatched fragment in the "middle" of the matching sets (of the map unit) that would imply that the fragments involved in the matchlist are a) contiguous in the clone, but b) not contiguous in the map unit. This is impossible. So, a clone must map onto a contiguous series of sets in a map unit to be considered a valid mapping. Similarly, in a mapping involving two map units, A and B, A must be contiguous along the sets of B, and B must be contiguous along the sets of A.

This fact can be used to limit the region which is being considered for possible matches. Once one set is known to be involved with the matchlist, only that set (or one "next" to it) needs to be considered for the next match. However, once a set **fills up** with matches (i.e., all the fragments represented by the lengths of that set are involved in pairings in the matchlist), then that set no longer needs to be considered. Instead, one of the sets on either side of it must now be taken into account as the mapping continues. The only sets which do not need to be full before they are removed from consideration are the sets on the "ends" of the region which is involved in the matchlist. Although the region of matches must be contiguous, all of the fragments in the end sets do not need to be included for this to occur.

When viewing a mapping in progress, there are three distinct regions in each of the areas of matches in the two units. One region is the **end set** of the area; another region is the **current set** being worked on; and the last region is all of the sets in between these two which have been filled with valid matches (**filled sets**).

Applying the ideas of the past few paragraphs to the example in Figure 6, the top set of the map unit is initially the current set. The next match discovered is 593 with 585. When another match is attempted, it is discovered that there are no more fragment lengths in this first set of the map unit which will map with anything from the clone. At this point, this set can stop being considered for valid matches, and the current set will become one of the adjacent sets. Since the mapping was started at one end of the map unit, there is only one adjacent set, so the second set down becomes the current one. The map unit now contains an end set and a current set.

The search for the next match only involves the fragment lengths of the current set, namely 767 and 647. Instead of having to search all eighteen lengths in the map unit for the next match, the search only involves two fragments. This restriction allows the topological algorithm to search efficiently for possible matches. Both of the fragment lengths in the current set match with lengths from the clone, 767 with 771 and 647 with 660.

With the current set now filled, the current set can move to the next set down (in this case, the third set from the top). An end set, a filled set, and a current set now all exist. The area involved with the matchlist will continue to grow in this fashion until all the matches have been made. The lengths in the third set of the map unit match with clone lengths as follows: 2472—2489, 2108—2154, 603—609, and 518—521. Finally, the 901 from the fourth set of the map unit matches with the 897 length from the clone. At this point, the mapping is finished because no more fragments in the clone are unmatched. As before, in the finished mapping (see Figure 7) it is possible to pick out the clone which exists within it. Also as before, the addition of the clone has added more information about the ordering of the fragment data. The new map unit contains eight sets, a refinement of the same information originally represented by the previous six sets.

3.2. The problem of ambiguous solutions

When a human is handmapping, he or she is concerned with finding a valid answer. Once this answer is found, it is easy to assume that it is the only mapping which can exist for a collection of data. Unfortunately, this is not always the case. It is possible for two map units to map together in two distinctly

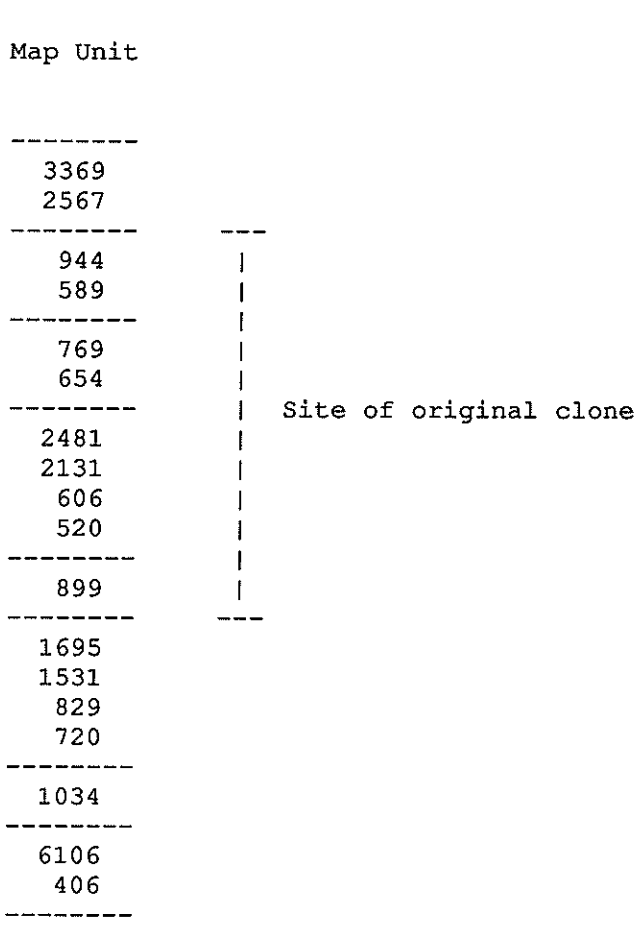


Figure 7: Result of mapping from Figure 6

different ways, each way being equally valid. These distinct mappings need not reduce to the case of equivalent matches which has been discussed earlier. When presented with this situation, it is impossible to know which mapping is the correct one, and neither answer can be used. Such a case is known as having **ambiguous** solutions. A simple example of this is shown in Figure 8. The two map units in Figure 8(a) can be used to form either of the solutions shown in Figure 8(b) and Figure 8(c). Since both are valid, this is an ambiguous mapping.

The only way to be sure that a mapping is unambiguous is to attempt to map the data in every possible manner. If more than one solution results, then ambiguity exists. The topological mapping algorithm attempts to discover every possible (valid) mapping which can be made between two map units. This is one area in which the automation of the mapping process can have tremendous benefit to the human

mapper. He or she gains the assurance that a mapping is unambiguous without the lengthy handmappings which would be required without the computer's help.

The strategy of not mapping two map units together if the mapping is ambiguous is a conservative approach intended to solve the following problem. If one of the possible maps is arbitrarily chosen, it may be the *wrong* map, and problems will occur in subsequent processing (i.e., since a piece of the puzzle was placed in the wrong position, other pieces placed subsequently will not fit correctly). Of course, one possibility is to take *each* solution in turn, using backtracking techniques (when subsequent processing reveals problems). However, backtracking has two disadvantages. First, it is computationally expensive, since it is exponential in nature. Second, since data are supplied for mapping incrementally, subsequent processing is effectively unbounded and it is unwieldy to keep track of backtracking information across distinct computational sessions. This conservative strategy of not mapping ambiguous data together relies on two possible resolutions of the ambiguity. First, the mapping of subsequent data may introduce topology which will eliminate all of the alternative maps, thus eliminating the ambiguity. Second, a human may be able to resolve the ambiguity either by applying data not available to the mapping algorithm itself or by heuristic insight.

3.3. Directionality of growth

Another problem of mapping is **directionality of growth.** In all of the preceding examples, the mapping has begun at the top end of the map units and continued towards the bottom. This may seem perfectly logical from a human perspective. However, "top" and "bottom" really have no meaning in these mappings. It is simply convenient to use these terms because it puts the data in a common environment so it may be discussed. In actuality, such directionality is meaningless.

Because of this, it is equally valid to begin the mapping at the bottom of the map units and move towards the top. This may seem like an unimportant detail. However, it is definitely not. It is possible that starting a mapping at two different locations will yield two different, ambiguous answers. Consider Figure 8 again. The solution in Figure 8(b) will result if the mapping is begun at the top of the two map units. If mapping is begun at the bottom, the solution in Figure 8(c) will result. Since the algorithm is concerned

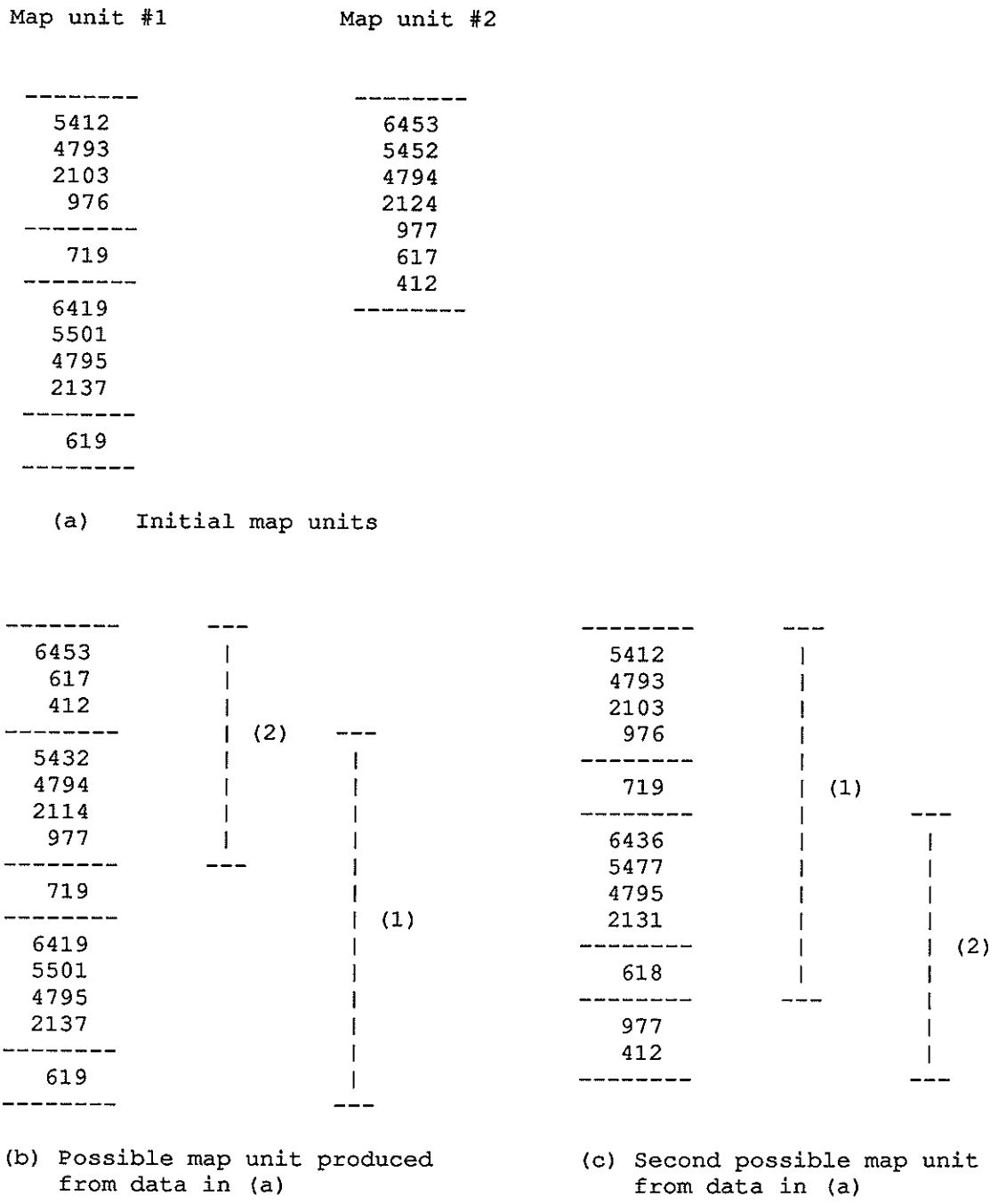


Figure 8: Ambiguous solutions

with generating all possible mappings, in order to identify ambiguities, it begins the mapping at several locations in the map units (exactly how it does this is discussed in Section 5.1). Therefore, it is imperative that the algorithm be able to map in either a downward or an upward direction.

Ideally, the topological mapping algorithm should handle both directions equally well. However, it is far easier to allow it to be biased in favor of one direction. Which direction is favored is unimportant. What is important is that both directions are eventually handled if needed. As currently implemented, the algorithm favors the downward direction; if at all possible, it will continue a mapping by advancing in this direction. Of course, the algorithm can map in an upward direction, it simply does not do so until advancement downward is no longer possible.

3.4. Orientation of the map units

The examples up to this point have left out an important detail which must be considered for the topological mapping algorithm to work. When a human handmaps, he or she is likely to ignore the orientation of the two units being mapped. In all of the examples so far, the units have mapped together by working on both of them from the top fragment down to the bottom one. It is possible, however, to have a situation in which one of the map units is oriented in the opposite direction. This is the case in Figure 9. Map unit #1 and map unit #2 will map together, but one of them is "upside down." A human working on this problem would realize this and simply keep it in mind while mapping. The algorithm, on the other hand, must handle this in a more explicit way.

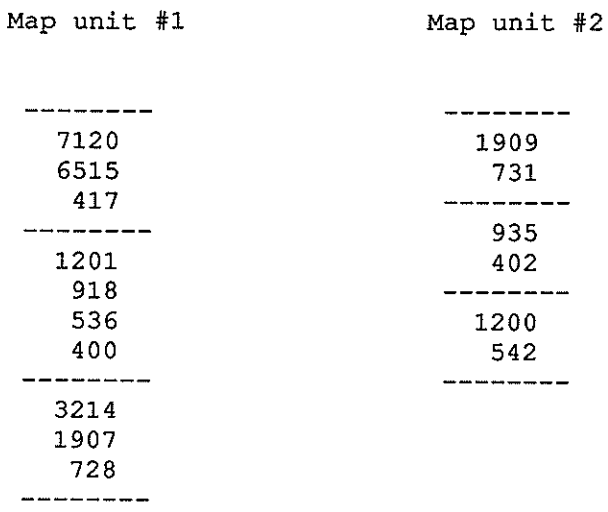


Figure 9: Map unit orientation

The solution to this orientation problem is a simple one; the algorithm simply flips the orientation of one of the map units during mapping. It is irrelevant which one is actually flipped; flipping either one in a case like this will result in both having a compatible orientation. Of course, it is not known if they are correctly oriented or not at the beginning of mapping. To handle this, the algorithm attempts to map the pair of map units exactly as they arrive. Then, it will flip the orientation of one and attempt to map the pair that way. This ensures that all possible solutions are discovered.

4. An overview of the code

The majority of the algorithm is embodied in just two routines: `topological_recurse` and `grow`. However, before examining the actual code, one should recognize and answer a crucial, underlying question; namely, how is the limiting property which is such an integral part of topological mapping going to be implemented in an algorithm? In order to understand this concept, it is useful first to consider exactly how the map unit data are stored by the software. Then, the larger question of the limiting property can be approached and understood.

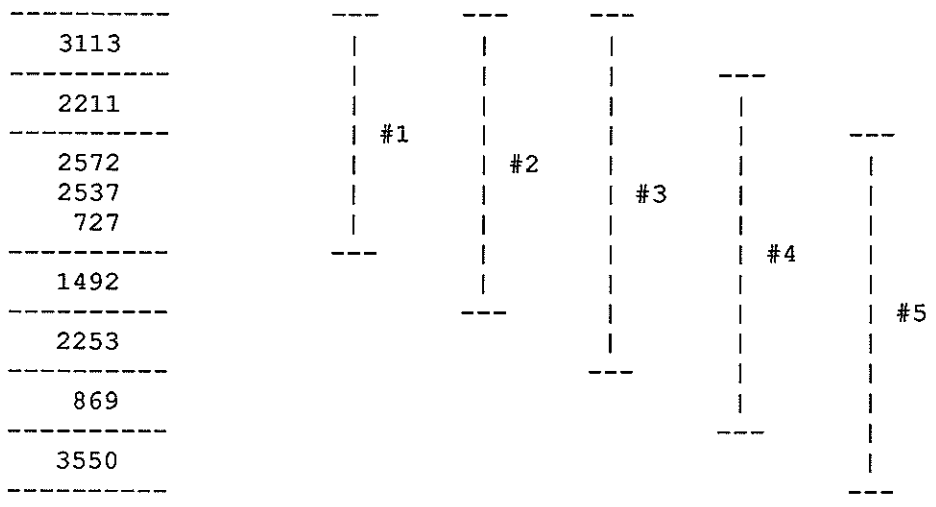
4.1. Internal representation of map units

The mechanism used to represent all map units is an abstract data type known as a **sequence-set tree**, or simply a **SST** for short. The name describes the underlying structure of this data type. The root node of the tree is a **sequence node** whose children are **set nodes**. These set nodes correspond to the distinct sets of fragment lengths in a map unit, for which the order (sequence) is not known.

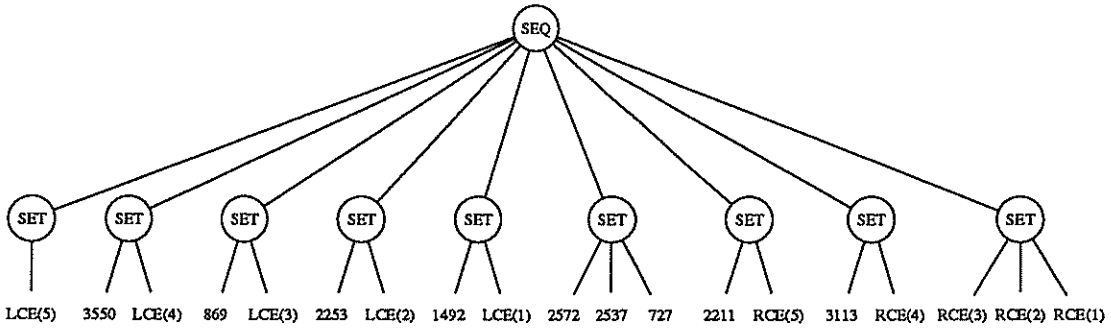
The children of the set nodes are of two types: fragment lengths and **clone ends**. A set node with five children, all of which are fragments, will correspond to a set in a map unit containing five fragments. Clone ends are the other data which may be associated with a set node. Clone ends are, literally, the ends of the clones being considered. Including clone ends in the sequence-set tree adds more information about the structure of the map unit. This can be particularly useful when sets of clones are being considered. They are also useful in a postprocessing test used to determine map unit equivalency (see Section 5.2). It is

difficult for a human to keep track of the clone ends in addition to the fragments. This is another benefit of using a computer to assist in mapping, since the computer can keep track of more data about the clones and map units.

Figure 10(a) shows a map unit along with the clones from which it was constructed. Figure 10(b) shows the corresponding SST which is used to represent the map unit. Here, the clone ends are assumed not to occur at restriction sites. Thus, the clone ends associated with a clone appear in the set node just outside the boundary of the clone as defined by the restriction fragments present within it. In fact, the clone end resides on the interior region of one of the fragments in the set node where the clone end appears.



(a) a map unit with its clone composition



(b) the corresponding SST

Figure 10: Representation of a map unit as a sequence-set tree

4.2. The focus

One of the major advantages of using the topological mapping algorithm is the decreased number of computations which must be performed in order to find the next match, due to the limiting property resulting from the structure of the map unit. To implement this idea in software, a data type known as a **focus** is used. A focus corresponds to the set of the map unit which can be searched for the next valid match. It operates in much the same way a human mapper would, by keeping track of the three different regions in a map unit mentioned previously: the end set, the filled sets, and the current set.

Each focus contains four pieces of information: the locations of the **border** and the **candidate**, and the current status of each. As new matches are made, the fragments in a set are internally shuffled to create two partitions. All the matched fragments are placed in one partition, and all the unmatched ones are placed in the other partition. The border indicates the boundary between the two partitions, while the candidate points to the next fragment length to consider for a possible match. The status of each simply reflects whether that particular component is uninitialized, active, no longer in the current set, or out of the map unit completely.

The focus is well suited for implementing the ideas of end set, current set, and filled sets. As mapping begins, a map unit always starts with only one focus active in it. As the mapping continues, and the area that contains matches grows, a second focus may be added. One of these two foci keeps track of the end set, and the other keeps track of the current set. All of the sets in between the ones to which these two foci point at are filled.

At the start of mapping, four foci are created: one in each map unit SST referred to as a bottom focus, and one in each known as a top focus. Initially, the border and candidate pointers in the bottom foci are set to the top fragment in one of the set nodes of the SST. The top foci's pointers are simply set to uninitialized, because these will not be needed until the scope of attention must be expanded.

As mapping proceeds, the border and candidate pointers will move to reflect the matches being made. When all of the fragment lengths in a particular set node have been considered, the candidate pointer will "run off the edge" of that set node the next time it tries to move to a new fragment. When this

happens, it is time to move to the next set node. Before doing so, however, the second focus is initialized. One of the two foci will remain in this set node as the other focus moves into a new set node. Which focus stays and which one advances depends on the current direction of growth. If growth is downward, then the top focus remains in this set node and the bottom focus **expands** (moves) into the next lower set node. If the direction of growth is upwards, then the reverse is true.

Eventually, a candidate pointer will not only run out of a set node, it will run off the end of the SST completely. When one top focus and one bottom focus have candidate pointers which are out of the SST, then the mapping attempt is finished. If these two foci belong to the same map unit, then the mapping was an **assimilation** (see Figure 11). This simply means that one map unit was totally absorbed into the other. On the other hand, if the two foci belong to different map units, then this is known as an **extension** (Figure 12), which means the obvious, i.e., the two map units contain an overlapping region, but each contains fragments not found in the other.

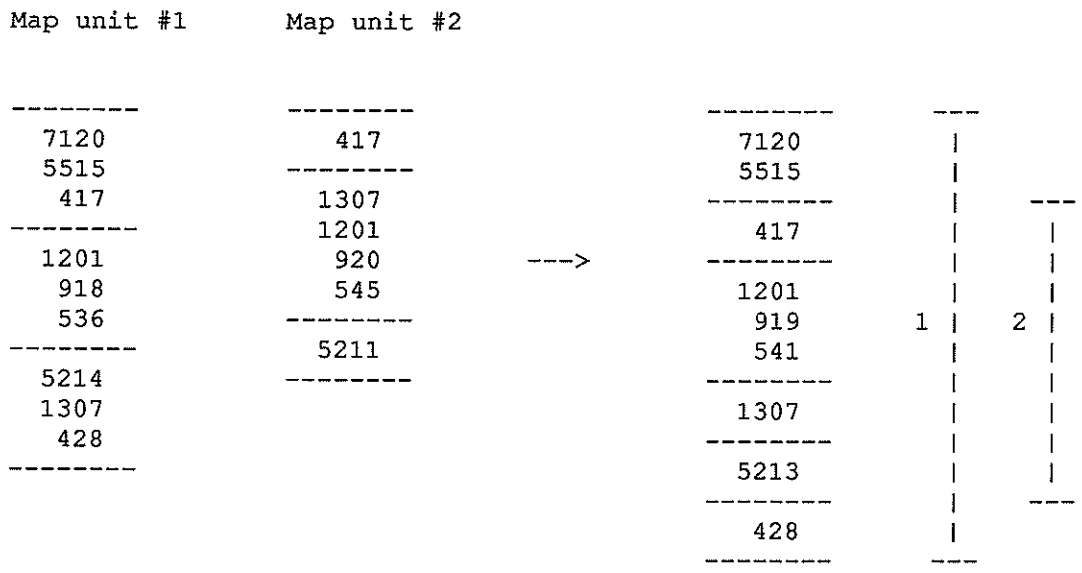


Figure 11: Example of an assimilation

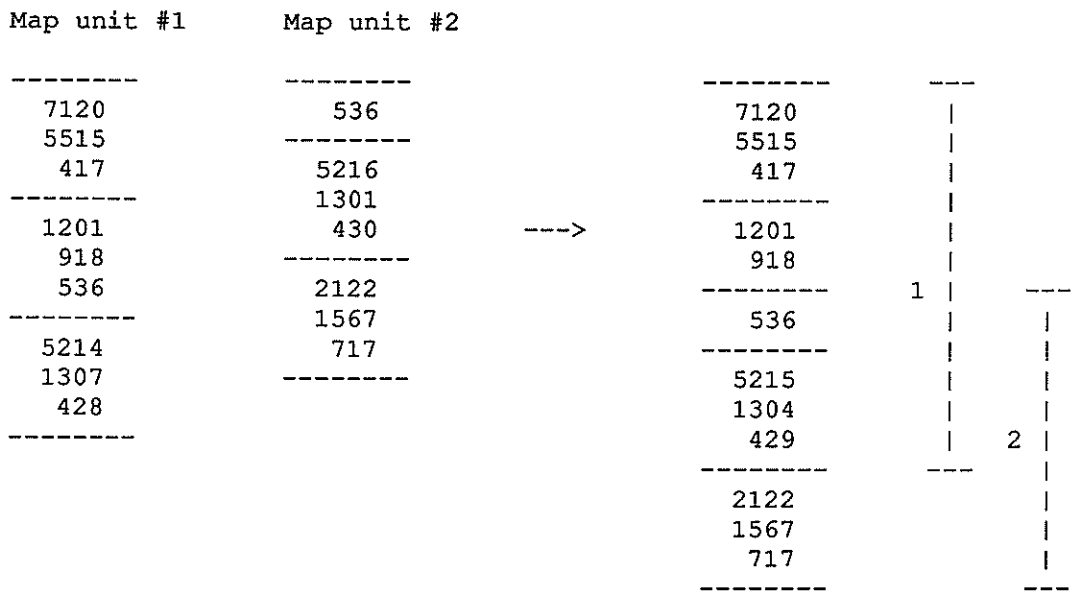


Figure 12: Example of an extension

4.3. Main routines of the algorithm

The routines which perform the majority of the work associated with topological mappings are `topological_recurse` and `grow`. As the name implies, `topological_recurse` is a recursive routine. `Topological_recurse` is also mutually recursive with `grow`, meaning that `grow` can call `topological_recurse` which in turn can call `grow` again.

Since the goal of a mapping attempt is to find every valid solution, all possible ways of combining the fragments in two map units must be considered. Many of these ways are destined to lead to dead-ends. When this occurs, the algorithm must have the ability to "back out" of its last move (the one which directly led to this failure) and attempt a different move. Also, since all solutions are desired, even a sequence of moves which leads to a correct answer must be backed out of, in order to find other valid solutions. This backing out of a move, or **backtracking** is implemented partially by the recursive nature of the routines. The majority of the backtracking, however, is accomplished through the use of **do-undo** procedures.

4.3.1. Do-Undo Backtracking

Do refers to those routines which cause a change in the state of the map units or in the state of foci associated with the map units. **Undo** refers to routines which restore the local computational environment to the exact state it was in prior to the calling of a do routine. The exact sequence of events which would be followed is :

(do action A) (call `topological_recurse`) (undo action A)

For example, a routine may be called which makes a match between the two fragment lengths being pointed to by the current candidate pointers. Once this match is made, the algorithm needs to consider all the possible solutions which can be reached with this match in effect, so it calls `topological_recurse`. Upon returning, all such solutions containing the match will have been considered. Since it is possible that solutions exist which do not include this match, this match is undone. At some point later, `topological_recurse` will be called again and will find all of the solutions which do not include this match.

4.3.2. Pseudocode for topological_recurse

With the understanding of the backtracking method used, it is now possible to understand the logic of `topological_recurse`. This is presented in pseudocode in Figure 13. The actual parameter list is suppressed here because the purpose of this pseudocode is only to convey the overall logic, and some of the physical parameters deal with concepts not presented here.

The first thing the routine checks for is to see if the current mapping is finished. As stated earlier, this is the case when one bottom focus and one top focus are out of the SST. If this criterion is met, then the routine simply calls `zip` which forms the resultant map unit for this mapping.

If the mapping is not finished, then the algorithm attempts to determine what it should do next by examining the status of the four foci. It is at this point that the bias towards downward growth can be seen. The first thing the routine always tries is to grow downward. It is only when a downward growth attempt is impossible or all the downward growth attempts have failed that the algorithm turns to the upward

direction.

4.3.3. Pseudocode of grow

The major routine that is called by `topological_recurse` is `grow`. Before this function can be called, there is some set-up which must be accomplished. `grow` accepts the foci as parameters, as well as the current direction of growth. Prior to each call to `grow`, then, in `topological_recurse`, there is a call to a routine named `advance_to_next_match`. This routine attempts to discover the next valid match, and moves the candidate pointers to this match. Then, immediately after the call to

```
topological_recurse()
{
     if (mapping is done)
          combine the two map units into an answer (zip)
     else
          if (4 foci active) and (bottom foci are in map units)
               attempt to find a match downward
               grow()
               undo this attempt
          else if (4 foci active) and (bottom foci are not in map units)
               attempt to find a match upward
               grow()
               undo this attempt
          else if (4 foci are NOT active)
               if (can find a match downward)
                    grow()
                    undo this attempt
          else
               flip() the direction of current set in first map unit
               topological_recurse()
               unflip() the current set

               force an attempt to grow first map unit upwards, mature()
               topological_recurse()
               undo this attempt, demature()

               flip() the direction of current set in second map unit
               topological_recurse()
               unflip() the current set

               force an attempt to grow second map unit upwards, mature()
               topological_recurse()
               undo this attempt, demature()
} end topological_recurse()
```

Figure 13: Pseudocode for topological_recurse()

`grow`, these pointers are moved backed to their original positions, an action performed by the routine `retreat_from_last_advance`.

As the pseudocode in Figure 14 indicates, the first action performed by `grow` is to force the match to be made between the fragment lengths pointed to by the candidate pointers. Then, the software checks for a need to expand. At this point, `topological_recurse` is called. Then, as in all major actions, this move is undone upon return from `topological_recurse`. This return may occur, however, after many more recursive calls to these two routines. Regardless of when this return takes place, it signals that all the possible mappings involving the matchlist currently being worked with have been considered. The success or failure of these considered mappings is irrelevant to the algorithm at this level, since all possible mappings must be examined at some point.

```
grow()
{
     make the match with current candidate pointer fragments
     check for possible need to expand
     topological_recurse()
     contract if expanded
     unmake the match

     if (growing downward)
          move the first unit's candidate pointer one fragment down
          topological_recurse()
          unmove the candidate pointer

          move the second unit's candidate pointer one fragment down
          topological_recurse()
          unmove the candidate pointer
     else   /* must be growing upward */
          move the first unit's candidate pointer one fragment up
          topological_recurse()
          unmove the candidate pointer

          move the second unit's candidate pointer one fragment up
          topological_recurse()
          unmove the candidate pointer
}  /* end grow() */
```

Figure 14: Pseudocode for grow()

4.3.4. Pseudocode for zip

The functions `topological_recurse` and `grow` are the heart of the topological mapping algorithm. There is, however, one additional routine which is important in its own right. This routine, `zip` (see Figure 15), is what creates the finished map unit produced by the mapping attempt. While doing this, `zip` is also responsible for checking for certain error conditions, as well as comparing answers for "bestness," as determined by some metric.

The `zip` function works with the matchlist which was built during the mapping process. This matchlist is used in three ways. First, it is used for an immediate test to see if this mapping can be valid. If the matchlist is empty, or contains fewer than a required minimum number of matches, then no new map unit is created. Second, the matchlist alone is the first thing which is tested by a metric for goodness, namely, matchlist subsetness, a topic which is discussed in greater detail in Section 5.3. Last, it is the matchlist and its relative position in the two map units which controls how the two map units are "zipped" together to form the finished result.

```
zip()
{
     if (matchlist is empty) or
        (matchlist does not contain minimum number of matches)
          return                /* no mapping exists */

     check matchlist against current list of answers for "best" one

     if (this is a bad mapping)
          return

     create the tree which will store this answer
     copy everything above matchlist in the first map unit into the tree
     add matchlist to the right end of the tree
     copy everything below matchlist in map unit #2 onto right end of tree

     check the "bestness" of this complete answer against previous ones
     if (answer is OK)
          insert answer into global list
}    /* end of zip */
```

Figure 15: Pseudocode for zip()

After creating the finished map unit, `zip` then checks this result against any which already exist. At this point, another test for "bestness" is performed, this time one involving tree equivalency (see Section 5.2). If the new map unit gets past this final test, then it is considered to be a valid mapping of the two starting map units, independent of any others which might have been formed to this point. If this is the case, the new map unit is added to the global list of current answers and control is returned to `topological_recurse`.

4.4. Communicating with the program: the user interface

The purpose of a **user interface**, in general, is to provide some sort of logical method for a human user to interact with a computer program. Ideally, the interface should be intuitive, easy to remember, and offer a more knowledgeable user the ability to accomplish complex tasks. The user interface for this implementation of the topological mapping algorithm is a menu driven one. This interface was designed with flexibility as a primary concern, as seen by the many options which it allows. But, it is also meant to shield the average user from inadvertent mistakes. For example, certain menu items are only shown and active at the times in which their selection would be valid and logical. This prevents the user from being bombarded by more data than he or she can handle, data which may have no meaning during most of the program use.

4.4.1. Ways of using the user interface

Since the user interface was designed with flexibility as a goal, it is possible to use the software package at several different levels.

4.4.1.1. Electronic bookkeeper

At the lowest computational level, it is possible to utilize the software for nothing more than a means of electronically organizing the handmapping process. When a human operator performs a mapping on even just two map units, the amount of data which must be kept track of is large. As the number of map units involved in the mapping attempt grows, so does the complexity of the bookkeeping. Then, since all

possible answers must be considered for an answer to be known to be unambiguous, the process must be repeated in an attempt to generate all of these possibilities. Finally, if a mistake is made with pencil and paper tracking of the process, time is lost while it is corrected and more paperwork is generated.

While it is possible to perform topological mapping totally by hand, it is a tedious process, prone to errors. This is where the lowest level of the software can be seen to be useful. Much in the same manner that a word processor gives a person an electronic notebook for thought composition, the topological software presents the user with an electronic mapping environment.

All of the functions needed while mapping are available as menu selections. A user can choose to load new map units from storage, choose which sets within a particular map unit to work on, match fragments of one map unit with fragments of another, flip a map unit, expand a focus into a new set if the current is filled, and so on. In addition, the software keeps track of certain potential problems. If, for example, the user selects to match a fragment of length 789 with a fragment of length 871, the program will warn that these lengths do not fall within the accepted error range of allowable matches. Still, the software never tries to limit the operator. Since it is possible that such a match might be valid (due to measurement errors, etc.), the program offers the user a chance to override this warning and perform the match anyway. Protecting without suffocating is the goal of the interface.

With the computer easing the bookkeeping side of topological mapping, it becomes possible for a human operator to map more efficiently. This is analogous to what happens when someone moves from a typewriter to a word processor--without the overhead of worrying about the details of the process, the process can go faster and with fewer errors. It also frees the mind of the human user for more thought on the intuitive side of the mapping instead of the more mundane, accounting side of the process.

4.4.1.2. Automatic utilization of the topological algorithm

Diametrically opposed to the bookkeeping use of the software is the software's ability to map data automatically. With the choices available on the menus, the user can load map units into the program and have it attempt to map them together. All valid answers are presented to the user, who then has the option

of saving one or more of them for future consideration. However, as DNA mapping involves a great deal of intuition, it is possible that a fully automatic mapping may not yield the best answer.

4.4.1.3. Sharing the burden

To answer the objection to the fully automatic mode, the software has the ability to interact at a high level with a human user. This process of working together can capitalize on the duality of the DNA mapping process; the human can provide the intuition while the computer handles the details.

To use the software in this manner, the human operator would first allow the program to attempt to find all the valid mappings for a set of data. Then, he or she would analyze these results to determine if there are any problems which the software did not address in a reasonable way. If this is the case, then certain parameters may be changed. After this, the software would again be applied to the data to see if any new and possibly better solutions are discovered. For example, one of the most common parameters to alter would be the 3% error window which fragment lengths must fall within to be considered the same. Often, two fragments are known to match, but they may differ by 3.1% or 3.2%. The human with his or her experience may realize that the two fragment lengths should match, at which time he or she can change the error window to a larger value.

This mode of operation is the most efficient. It uses the strengths of both participants in the mapping. The computer provides speed and thoroughness, while the human adds intuition and experience.

4.4.2. Undoing in the program

As part of the system's flexibility, it offers the choice of undoing any major action. This is not limited to a single action; instead, the user can undo any number of operations. In order to implement this ability, the program uses a **stack**. As the user selects operations from the menus, each of the actions is **pushed**, or placed onto the stack. Each single entry in the stack consists of the actual operation being performed, as well as any parameters needed for the operation.

When the user decides to undo an operation, the program simply **pops**, or removes the top element off of the stack. Since a stack operates in a last-in-first-out manner, the top element will be the last operation performed. After removing the operation from the stack, the program determines which operation it was. Then, it determines what operation needs to be performed in order to undo the original action.

This undoing process is aided by the fact that, as explained earlier, each major action needed for the topological algorithm has a corresponding function which will undo it. This greatly eases the whole process of undoing an action. The fact that the program allows a user to undo a sequence of actions frees the user from constantly worrying about the consequences of an action. This allows more experimentation to occur.

4.5. Control of parameters

One primary reason for using a computer is to increase the ease of doing something. One way that a software package can do this is to minimize the amount of adaptation that a user must make in order to use it effectively. With this idea in mind, the topological mapping program uses a set of routines to achieve parameter control. This allows a variety of options to be set at **run-time**; that is, it allows the user to alter the settings of the program at the time it is being used, without having to go through the lengthy process of recompiling it. This greatly speeds any sort of experimentation with the default values which the user may wish to undertake. The complete list of parameters available to the user are listed in Table 1. Some of these options have not yet been presented, but will be discussed in subsequent sections. These are accessible to the user through the menu option of **Param**.

5. Decisions which must be made during topological mapping

The lowest level of the software knows how to do one thing: given two map units and four foci pointing to various positions in them, the software will create a set of every possible way that the two map units can map together. By limiting the basic activity to this one task, it was possible to program it to do this one thing very well. However, there are several additional questions which must be addressed about a

Error Window Percentage
Minimum Number of Matches
Maximum Number of Backtracking Steps Allowed
Subsetness of Matchlists
Tree Equivalency
Top-level methods :
  Tree
  Linear
  Sorted
    Ascending
    Descending
  Subclone Handling
Mid-level methods :
  m × n (sets)
  m + n - 1 (sets)

Table 1
User options

mapping, in addition to generating all the possible answers which result from mapping the map units together. The first is perhaps the most obvious--how are the four foci to be initially chosen?

5.1. Choosing the foci

Choosing the positions of the foci at which mapping should be started is a **preprocessing** event, as it occurs before the base algorithm can start to work. The idea behind this needed action has been mentioned before. When mapping, a person wishes to be certain that all possible combinations of the two initial map units are considered, in order to ensure that ambiguity does not exist. Each map unit consists of a number of sets, each containing a number of fragments. Choosing different starting positions (that is, choosing two different sets as starting locations) for the foci may lead to different results. In order to find all possible solutions, the base algorithm must be started in a variety of ways, which in the aggregate allow any specific fragment in one map unit to be matched with any fragment in the other map unit.

5.1.1. Every fragment with every other fragment

When considering a mapping involving two map units, one way to ensure that all possible mappings are created is to begin with the candidates of the initial foci pointing to every possible fragment in the map units. The first mapping which this approach might try would begin by considering the first fragment in

each map unit as the initial candidate for that unit. After all the possible mappings had been generated with these two fragments as the initial candidates, the next step would be to move one of them to the next fragment. Then, every possible mapping with these two fragments as initial candidates would be generated. Eventually, all possible pairs of starting candidate fragments would be checked.

This method would certainly ensure that every possible starting situation had been considered. The drawback to this method is efficiency. This method is slow due to its thoroughness. To check every possible mapping using this method would involve

$$(\#\text{ fragments in map unit 1}) \times (\#\text{ fragments in map unit 2})$$

different starting positions. The software would generate all the possible mappings from each one of these. Since each starting situation might lead to several mappings, the amount of time required is staggering. For map units involving any reasonable number of fragments, this method is clearly unmanageable.

A careful analysis of the above approach will reveal a way of improving it. The approach ends up generating numerous maps for each beginning position of the candidate pointers. If all the possible mappings are generated with the candidates initially pointing to the top fragments in each map unit, and then all the possible mappings are generated with one candidate pointing to the top fragment of the first map unit and the second candidate pointing to the second fragment of the second map unit, it is highly likely that many of these mappings will simply be duplicates. The solution to this problem of efficiency is to eliminate some of this duplicated work.

5.1.2. Every set with every other set

Instead of considering a fragment-fragment match, there is a better method. By understanding that a great deal of redundancy is inherent to the fragment-fragment approach, it should be easy to see that it is possible simply to begin the mappings with every set of the first map unit matched with every set in the second map unit. The first fragment in each set would be the initial candidate of the foci.

The jump from every fragment to every set does accomplish the desired purpose of reducing the number of mappings which would be performed as well as lowering the number of redundant answers.

The important thing to consider is whether or not the set-set method is as thorough as the fragment-fragment method. It can be shown that it is.

Consider a single set of fragments in any map unit. In a set-set match, the candidate of the focus will be pointing to the first fragment of this set when the initial call to `topological_recurse` is made. This fragment may be matched with a fragment in the other map unit, and the mapping will continue from this point, with all mappings involving this first fragment being generated. Obviously, these are the same that would have been generated by the fragment-fragment method on its first attempt, which also involved this initial fragment.

At this point, due to the backtracking of `topological_recurse`, the candidate pointer in this set will move from the first fragment to the next one. The process will repeat itself, generating all mappings in which the second fragment matches with something in the other map unit. It is *crucial* to realize that this movement of the candidate pointer to the second fragment occurred because of backtracking. In the fragment-fragment method, backtracking would also result in this occurring. Then, after `topological_recurse` has been returned from at the top level, the fragment-fragment method would move the focus to the second fragment and call `topological_recurse` again. In summary, every call to `topological_recurse` with the candidate pointer of the focus pointing to a particular fragment will generate all of the mappings involving this fragment, as well as all of the mappings involving the remaining fragments in this set (these latter mappings coming about from the backtracking nature of the algorithm). There is no reason for the method of choosing initial foci positions to test each fragment-fragment pair in a set explicitly; it is sufficient to test only the initial fragment-fragment pair, and to allow backtracking to take care of the rest.

But the set-set approach still has efficiency problems. Through empirical testing, there is evidence that redundancy still exists, although at a greatly reduced level. Additionally, the number of starting configurations is still significantly large. The actual number of different starting positions will be

$$m \times n$$

where m is the number of sets in the first map unit, and n is the number of sets in the second map unit. One

primary reason for using the topological mapping algorithm in the first place was to capitalize on the information provided by knowledge of the map units' structure. But, the more that is known about the structure of the map units, the more sets they will contain. With the set-set strategy described above, more sets will yield longer running times. This fact tends to counteract the benefits of using topological mapping in the first place. A better method needed to be found.

5.1.3. A modified every set with every other set method

The original set-set ($m \times n$) approach is still relatively inefficient because one specific match between two sets in the different map units can produce essentially the same "shift" between the map units as another different match between the sets. The way to eliminate this inefficiency is to cause the two map units to "slide" continuously without reproducing any equivalent positionings. To follow the logic, consider the example shown in Figure 16. With the four sets of map unit 1 and the five sets of map unit 2, the set-set method would have to consider 20 (4×5) different starting configurations. Admittedly, this is a greatly reduced number than the fragment-fragment method, which results in 285 different starting configurations (15 fragments in map unit 1 and 19 fragments in map unit 2), but efficiency is still an issue. As before, the key to improving efficiency is to consider the problem of redundancy.

Consider the set-set method as it has been presented. The set-set method would first find all the mappings which result from initially matching set A with set Z. Then, A would be matched with Y. A would next be matched with X, W, and finally V. With A having now been matched with every set in map unit 2, B would now be matched with Z. After B is matched with all the sets in map unit 2, the process would be repeated for C and for D.

Now, however, carefully examine the case when A is matched with Z. At some point in the course of the mapping corresponding to this initial configuration, the focus of map unit 1 will move to B, and the focus of map unit 2 will move to Y. Alternatively, consider a subsequent starting configuration of B matched with Y. Since the mapping proceeds in both directions, at some point the foci will have moved into A and Z, respectively. This overlap of work is the source of the redundancy in the raw set-set method.

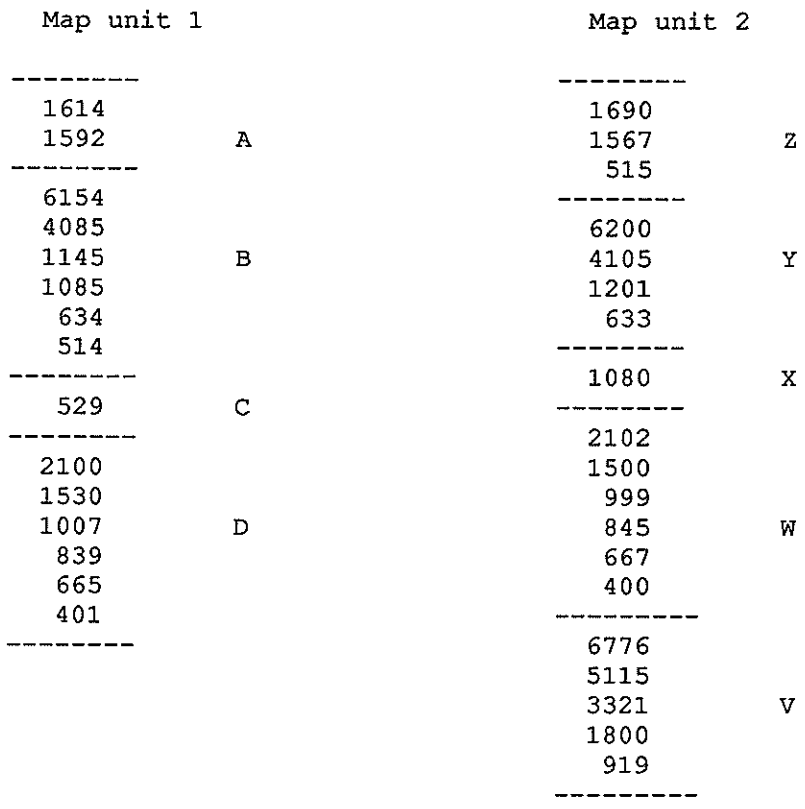


Figure 16: Example of set-set mapping attempt

In order to combat this, a modified set-set approach was designed. Instead of every set in one map unit being matched with every set in the other map unit, the two map units are treated as "sliders." Returning to the example of Figure 16, set A is matched with every set in map unit 2. Then, set Z is matched with sets B through D. Since the foci expand freely between the individual sets of a map unit, this reduced number of initial positions still allows all possible mappings to be discovered, while reducing the number of redundant answers produced. The actual number of starting configurations is reduced to

$$m + n - 1$$

with m and n again being the number of sets in each map unit. For this particular example, the number of starting configurations which need to be considered is only 8, as opposed to the 20 required for the original set-set method. With larger examples, the savings would be even greater.

5.2. Choosing the best mapping: map unit equivalency

As mentioned earlier, one problem associated with topological mapping is that of equivalent solutions. The term **tree equivalency** is used to describe the new situation, as define below by length equivalency. (This refers to the internal representation of the map units as sequence-set trees. The alternate term **map unit equivalency** represents the same concept.) When it occurs, it is necessary to determine which mapping is the "best" one and continue the mapping with that map unit. There are a number of methods by which "bestness" can be determined. The one which is currently implemented by the software is a rough analysis of **best calculated fit**.

Best calculated fit attempts to determine which of the mappings involves pairs of fragments with the least "spread" separating the fragment lengths. The mapping which contains the least difference calculated over all of the match pairs is considered "best".

The best calculated fit implements a multilevel test. First, the two map units in question are tested for **structure equivalency**. Two map units are defined as structurally equivalent if and only if they contain the same number of sets of fragments, the corresponding sets contain the same number of fragment lengths, and the positioning of clone ends in the corresponding sets is identical. If the two map units are determined to be structurally equivalent, they are next tested for **length equivalency**. This is the property of 1) being structurally equivalent and 2) having all the corresponding fragments in the two map units falling within the error threshold with regard to their lengths.

If the software determines that two map units are length equivalent, it then calculates a fit number for each. The map unit with the best (lowest) fit number is kept, and the other is discarded. This test is performed on every pair of map units produced by a mapping attempt. As with all of the pre- and post- processing routines, it is the option of the user to enable and disable this function. There may, for instance, be times when a user wished to be presented with all the possible answers, and not just the "best" ones.

5.3. Choosing the best answer: matchlist subsetness

When mapping, the "best" answer is often the answer which is the most compact. Consider the two map units in Figure 17. There is a total of four fragments which can be matched. Each of the mappings shown is a possible answer. In a case like this one, it is generally best to choose the first answer, because it contains fewer total fragments and is, therefore, more compact.

The key to identifying the most compact map unit lies in the matchlist being constructed by the algorithm. The most compact map unit has the matchlist containing the greatest number of matches. In fact, the matchlists of each of the nearly identical, but less compact map units, are simply sets of fragments which are subsets of the set of fragments contained within the longest matchlist. This property is defined as **matchlist subsetness**. Utilizing this property, the software has an option to invoke a postprocessing test

#1
6198
4082
1614
1392
1150
637
513

#2
8567
6105
4109
1586
1139

Answer A
8567
6157
4096
1600
1145
1392
637
513

Answer B
8567
6105
4096
1600
1145
6198
1392
637
513

Answer C
8567
6105
4109
1600
1145
6198
4082
1392
637
513

Answer D
8567
6105
4109
1586
1145
6198
4082
1614
1392
637
513

Figure 17: Four different answers

for matchlist subsetness. As each new potential answer is discovered and is attempted to be zipped into existence, `zip` first does a test in order to discover if the current matchlist is a subset of an existing one. If so, this potential answer is discarded.

There are two ways to determine matchlist subsetness. The first method was the one originally employed by `zip`. Matchlist A is a subset of matchlist B if and only if each fragment-pair of A exists in B. This involves a comparison of actual fragments; fragments which are equivalent are not used for this test.

In many cases, this method suffices. However, consider the case in which matchlist A is not a subset of matchlist B, but is a subset of matchlist C where matchlist C came from a map unit which was equivalent to the map unit which matchlist B came from, and was discarded earlier. The map unit which would be formed from matchlist A is not as compact as possible, and should be discarded, but the simple subsetness test would fail to discover it.

To combat this problem, a test for matchlist subsetness relying on length was developed. However, this method, in its pure form, would also be unsatisfactory. Consider a case of equivalent map units. These must have the same matchlist, at least as far as length is concerned. The subset test would eliminate one of the equivalent answers without incorporating a proper test to determine which was "better".

To overcome all of the deficiencies mentioned to this point with regard to subsetness, a combination of the two methods is used. This hybrid method uses the exact fragment subset test if the two matchlists have the same cardinality. This addresses all the possible equivalent answers, since two equivalent answers must have matchlists of the same length. If, on the other hand, matchlist A is a **proper subset** of matchlist B, that is, there is at least one fragment length in B which is not contained in A, then the subset test involving length is performed. By successfully combining the two ideas, a new method is created which still permits the later testing for map unit equivalency, yet discards any valid subsets.

5.4. Limiting the search for answers

The base topological mapping algorithm is designed to produce all the possible answers for a given set of map units. This can lead to a lengthy running time in some cases. The case where two map units have a large number of matches is an example of this. Consider two map units which have a ten fragment overlap. The algorithm will produce an answer incorporating a ten fragment matchlist. The algorithm will also generate an answer which contains a nine fragment matchlist; in fact, it will produce ten such answers (the number of nine fragment combinations from a field of ten fragments is given by the formula $\frac{10!}{(1!)(9!)}$ (or simply 10). The algorithm will also create possible answers containing eight fragment matchlists, of which there are $\frac{10!}{(2!)(8!)}$, or 45. Considering all of the possible sizes that a matchlist could be (from 10 down to whatever the minimum length of the matchlist is defined to be), the algorithm does a tremendous amount of work in order to produce all of these answers. Most of this work is unnecessary as well. With a ten fragment maximal matchlist, all nine fragment matchlists are subsets of the ten fragment one. All of these answers will be filtered out by the postprocessing matchlist subset routines. By this time, however, the program has already expended a lot of work.

The solution is to eliminate such answers as the algorithm is creating new potential maps. In order to accomplish this goal, a thorough understanding of the algorithm and its backtracking implementation was needed.

When a mapping is first attempted on two map units, the algorithm accepts the first matches that it finds at every level of recursion. Each level of recursion represents one entry in the matchlist. Because of this ordering, the first answer produced is one containing a matchlist with a maximal number of fragments. At this point, the last fragment match is undone, and a new match is searched for. If one is found, then two answers now exist with matchlists which differ by one fragment. If, on the other hand, no new match is discovered, then the next answer created is the one containing the maximal matchlist minus the last match. This matchlist is a subset which should be discarded.

In order to implement this idea of limiting searches to matchlists which are not subsets of others, it was necessary to limit the amount of backtracking allowed by the algorithm. By keeping correct count of

the calls between the two recursive routines, it was possible to keep a record of when backtracking was performed. Every call to `topological_recurse` checks this count to see if it exceeds a maximum allowed backtrack count. If it does then no further backtracking is allowed.

5.4.1. When is backtracking not backtracking

There is a special case of backtracking when it should not truly be counted as a backtracking attempt. Consider a case where one map unit contains a fragment of length 1000, and the other map unit contains fragments of lengths 1010 and 990. The initial match would be between 1000 and 1010. When backtracking is performed and this match is undone, 1000 and 990 would match next. But, this is not a new match, it is simply an equivalent match. No backtracking has actually taken place, from the point of view of the matchlist. It still contains the same lengths, at least when the error window is considered.

Added to the software to take care of this is a routine which checks the next match to determine if it is an equivalent match. If so, the backtrack count is decreased by one to offset its incorrect increase earlier. This checking routine is not foolproof, and can sometimes be fooled, especially if the equivalent match does not occur at the next match or is in another set of the map unit. Still, by allowing this level of checking, more refinement can be gained.

5.4.2. What is lost, what is gained?

The important question to consider is how this modification affects performance. Obviously, answers are discovered more rapidly, often in only a fraction of a second. There is also a high probability that the answers provided are the best of all the valid answers. There exists the possibility, however, that the backtracking check will inadvertently wipe out an equivalent answer which may be better than any it discovers. This should be kept in mind while using the software. The ideal usage of this tool would be to have the backtracking set to a low level at the start of a mapping attempt. This will rapidly generate answers which are highly likely to be the best ones. If the user determines that something is being lost, he or she always has the ability to increase the level of backtracking permitted, in order to discover these lost answers.

6. Top-level strategies

Up to this point, the discussion has centered on how to take two map units and produce all the valid mappings which can result from them. There is a higher level of operation than this one. This can be stated as :

Given a set of n map units, what is the best way to attempt
to map pairs of map units together, with the goal being to
create as few (hopefully one) final map units as possible?

In this case, set has an imprecise meaning, more akin to the common usage of it referring to any group of objects. For the purposes of the computer algorithm, the map units are actually stored in an abstract data structure called a **list**. This is to allow the algorithm to access the map units by position; for example, pick the fifth map unit in the list and perform some action on it.

Returning to the problem, a brute force method would work. Simply determine every sequence in which the list of map units can be mapped together. Then, try each of these mapping sequences. Finally, determine which of the sequences leads to the best final answer. With n map units in a list, this method would require n! sequences to be generated and attempted. Also, since each sequence consists of n map units, a minimum of n-1 mappings will need to be performed in order to map the data together for *each* sequence tried. Once again, it is not sufficient to stop after a successful mapping, as all the sequences must be generated and tested in order to assure that ambiguous answers do not exist. This means that (n-1)(n!) mappings need to be attempted. Finally, with each mapping attempt being time consuming, the amount of running time needed for this brute force method is too large to be effective.

The solution is to to create a heuristic which has a high probability of success. This heuristic would not have to consider all the possible mappings, just the most likely ones. Naturally, this leads to a decrease in time required to map a list of map units. On the other hand, no such heuristic is guaranteed to work in all cases. The problem becomes a time-certainty tradeoff.

6.1. Methods of ordering the mapping of a list of map units

At the most basic level, a method of determining the order in which to attempt the mapping of a list of map units must be developed. There are two basic ways of looking at this problem. Both have strengths and weaknesses, and neither has proven itself to be 100% useful.

6.1.1. The linear method

The linear method (see Figure 18) of mapping is the easiest to visualize. In fact, this is the method normally utilized by humans during handmapping, and was used during the example depicted in Figures 1 through 5. Consider a list containing five map units, labeled A through E. First, the algorithm would attempt to map A and B together. If successful, the number of total map units drops to four. Then, the new map unit A+B would be mapped with C, if possible. This process continues, with each new map unit simply being added to the map unit being built. If all of these map units correctly map together, then the linear method will make a top level call to `topological_recurse` a total of four times. In general, a best case performance of the linear algorithm on a list of n map units requires n-1 top level calls to `topological_recurse`.

If a mapping is unsuccessful, some type of action must be taken. Assume that A and B failed to map together and give a unique solution. In such a case, B would be delayed from further consideration until all the other map units had been examined. Consider a case where A, C, and D mapped together. After the first pass through the list, the map units now look like this:

A+C+D B E

with neither B or E being mapped successfully with A+C+D. Now the algorithm will stop considering A+C+D and attempt to map B and E together. If successful, the list is now A+C+D and B+E. The algorithm would attempt to map these two units together, and, if it failed, would return these two as the solution.

It is possible for a mapping attempt on a map unit list to be unsuccessful in mapping all of the map units together. If this is the case, then **islands** are said to have been formed. It might actually be that the

```
for i = 1 to (# of map units in list)  {
     success = TRUE
     while (success = TRUE) AND (more than one map unit) {
          success = FALSE
          get tree1 from position i of list
          for j = (i+1) to (# of map units)  {
               get tree2 from position j
               attempt to map tree1 and tree2
               if (successful mapping) {
                    success = TRUE
                    remove tree2 from the list
                    store answer
               }
          }
          replace position i with answer
     }
}
```

Figure 18: Pseudocode for linear method

initial data list contained map units from two noncontinuous regions of the original genome, or it may indicate that the software was simply unable to find a unique, unambiguous solution. It is the responsibility of the user to determine this, and to act on it. It is always possible to alter various parameters and then to resubmit the new list of map units to the program, in order to try to create a single map unit. It is this flexibility which makes the software useful.

6.1.2. The tree method

As an alternative to the linear method, the tree method (see Figure 19) was developed. Rather than the answer growing in a linear fashion from left to right, the answer is built up in pieces. Again, assume a working list containing five map units, labeled A through E. The tree method begins on the left and attempts to map A and B. Then, it attempts to map C and D. E is a leftover map unit and must be delayed until the next pass.

After the first pass, the list has become (assuming that all mapping attempts were successful):

A+B C+D E

Next, A+B is mapped with C+D, again leaving E as a loner. This results in A+B+C+D and E. Finally, the algorithm attempts to map A+B+C+D with E and finished. Notice that the number of top level calls to `topological_recurse` is again four, the same as with the linear method. In general, best case

performances of both methods yields a total of n-1 top level calls to `topological_recurse`, given a list of n map units.

As before, it is important to ask what happens if a mapping attempt is unsuccessful. If the algorithm attempts to map A with B and fails, it attempts A with C, then A with D, and, if needed, A with E. No map unit which fails to map is ever discarded completely since there is always the chance that it might map into the answer at some latter point.

6.1.3. Comparison of the two methods

Both the tree method and the linear method require the same number of top level calls to `topological_recurse`. Even so, the tree method is superior to the linear method for ordering map units. This is due to the fact that the tree method quickly adds topological information to the mapping attempts. The linear method does not give this added benefit. Since topological mapping depends on structure in the map units as a way to improve efficiency, it is advantageous to attempt to build this structure as quickly as possible.

While working on a map unit list with the linear method, there is one map unit which represents the answer being built. All of the other map units are the original ones. No new structure is added except

```
success = TRUE
while (success = TRUE) AND (more than one map unit) {
     success = FALSE
     for i = 1 to (# of map units in list)  {
          get tree1 from position i in list
          for j = (i+1) to (# of map units in list) {
               get tree2 from list at position j
               attempt to map tree1 and tree2
               if (successful mapping)  {
                    success = TRUE
                    remove tree1 and tree2 from the list
                    put (tree1 + tree2) in list at position i
               }
          }
     }
}
```

Figure 19: Pseudocode for tree method

when a map unit is finally mapped into the answer. This is in contrast to the tree method. After the first pass on a list of map units, several map units should have mapped together. This forms map units with more structure. Then, when the second pass is performed, the algorithm is working not with the original map units, but with the more structured results of the first pass mappings. Each subsequent pass is working with more structured map units.

Because of this fact, the tree method is best used along with the method of combining sets which results in m + n - 1 mappings. On the other hand, the m + n - 1 method gives no performance advantage over the m × n method on a map unit list being mapped with the linear method, if the list contains only clones. Clones contain only one set. The m × n method of combining sets will always yield m × 1 = m initial configurations with the linear method, while the m + n - 1 method will cause m + 1 - 1 = m initial configurations as well, providing no improvement. In contrast, the tree method will map these one set map units together, rapidly forming more complex map units, so that the savings of the m + n - 1 over the m × n method becomes more prevalent. This combination creates an extremely efficient method of attacking a mapping problem.

6.2. Preprocessing to aid the software

On top of the software as a whole, there are several techniques which have proven valuable in reducing the amount of time required to map a list of map units, as well as to improve the quality of the results. These are referred to as **preprocessing** techniques. Often, these preprocessing routines do not have to be as efficient as the routines contained in the base mapping algorithm itself, because preprocessing usually is performed only once.

6.2.1. Sorting the clones

The easiest type of preprocessing to perform on a map unit list is simply to sort them, based on the number of fragment lengths. Then, mapping can begin with the two map units with the most fragments. This is exactly what the original example in this report used when the two longest clones were selected. This helps in three areas. First, there is more information to work with in longer map units, providing a

large amount of data early in the mapping process. Second, if a minimum number of matches is required, it is more likely that two long map units will overlap with this minimum than two shorter map units. Last, shorter map units are delayed in the mapping attempt. Smaller map units tend to be more likely to lead to ambiguities than do larger ones.

6.2.2. Delay of subclones

Another method, similar to sorting, involves a delaying action performed on any subclones which exist in the data list. A **subclone** is defined as a clone whose fragment set is a proper subset of another clone's fragment set. This second clone is referred to as a **superclone**. A subclone may have several superclones with which it may be associated. The problem with subclones is that, by their very nature, the data which they contain adds no new knowledge about the fragments contained in a mapping being performed. A subclone can only add structural information. Unfortunately, if a mapping is attempted between a subclone and one of its superclones, this mapping will be ambiguous. For example, if the subclone

{5430, 3332, 1010, 876, 545}

is involved in a mapping attempt with its superclone

{5450, 3334, 2560, 2019, 1012, 880, 549}

then this problem of ambiguity arises. It is known that the set of fragments

{5440, 3333, 1011, 878, 547}

overlap. The problem is what to do with the two "leftover" fragments of the second clone. Do they both go on one "end" of the mapping, producing

{2560, 2019} {5440, 3333, 1011, 878, 547}

or do are they split, one on each "end" of the created map unit?

{2560} {5440, 3333, 1011, 878, 547} {2019}

Since there is so little structural information available in the clones, it is impossible to know the correct way for this assimilation to be resolved. So, this yields an ambiguous mapping. It is therefore advantageous to ensure that the algorithm does not attempt to map subclones onto superclones.

On the other hand, it is impossible to ignore subclones completely. Towards the end of a mapping activity, the data which they contain become invaluable for refining the final map unit produced. The subclones do not provide any information about new fragment lengths (since all the fragments of a subclone have already been handled by its superclone mapped earlier), but they can provide more information concerning the topology of the map unit in the region into which the subclone maps.

The software attempts to allow the topology-aiding benefits of subclones while protecting itself from the detrimental effects. In order to accomplish this duality, the software first sorts the list of map units under consideration. It then separates this sorted list into two lists, one of superclones and one of subclones. Next, the software attempts to create a mapping for the list of superclones. Finally, the list of subclones is rejoined to the list of answers produced from the initial mapping attempt. This new list is resubmitted to the mapping algorithm. This protects the software from including the subclones too early (increasing the chances that a subclone will not map directly to one of its superclones), yet still permits the data in the subclones to contribute to the final answer. Also, there are additional safeguards in the software which will catch a subclone if an attempt is made to map one onto a superclone. If so, that particular mapping attempt is discarded, just as a normal failure would be.

6.2.3. Godslist

An extremely promising form of preprocessing exists in a form known as **godslist**. This rather lofty title simply refers to a list which contains matches which are known to be correct--either through human intuition or by experimentation. The "god" in the name refers to the fact that the algorithm is ignorant of the source of the knowledge; it merely assumes that this knowledge is valid. From the algorithm's perspective the data comes from an all-knowing, infallible god.

As the algorithm proceeds in a mapping attempt, every move to a new candidate fragment causes a check to be made against godslist. If this fragment is found to be in godslist, then it is immediately matched with the correct fragment in the other map unit, and the mapping continues. This saves all of the time which would normally have been spent in order to find this match.

As with all other base routines, there is a routine which undoes a match from godslist. If the algorithm is backing out of a particular mapping attempt, and a godslist match is undone, then this routine handles the repositioning of the fragments and the foci pointers to their positions prior to the making of this match.

In addition to decreasing the time required to find certain matches, another helpful aspect of godslist lies in determining the validity of the final answer produced. If godslist contains five pairs of matched fragments, and a solution contains only four of these, then this solution is invalid, given the knowledge contained in godslist. In this way, godslist is an aid in cutting down postprocessing time. Godslist is also a valuable preprocessing tool. This is the area where its usefulness becomes most readily apparent.

6.2.3.1. Ways of using godslist as a preprocessing aid

The easiest way to use godslist in this fashion is simply to have the user select the fragments in the two map units which he/she knows belongs together. This knowledge may come from exact **DNA sequencing** of the data, or from the user's intuition that two particular pieces "have" to fit together, even if an initial application of the software failed to reveal this. As the algorithm proceeds in the mapping attempt, all of the matches in godslist will automatically be made, if possible. As mentioned earlier, this reduces the amount of time required for the final mapping.

Another potential use of godslist comes from an interesting fact concerning DNA fragments lengths. The probability of a certain fragment length occurring in a data list decreases as the length of this fragment increases. In other words, there is a higher probability of shorter fragments being present than longer ones. If two map units contain fragments of 6900 bp and 7100 bp in length, respectively, these two fragments have a high probability of being the same one, even though they are farther than 3% apart. By making use of this fact, the software could make a prescan of the map units, searching for such large fragments. When two long fragments had been discovered and there was a high likelihood that they pair, then this match could be added to godslist to be acted upon later.

6.2.3.2. Godslist and group interaction

Once a godslist is formed and being used, the question addressed earlier of how to match sets in one map unit with sets in the other becomes irrelevant. As long as one fragment pairing is known to exist between two map units, then the starting set in each map unit should be the set containing this known paired fragment. If more than one match is known, any of the corresponding sets may be used. This fact drops the number of top-level calls to `topological_recurse` from $m \times n$ (for set-set), or even from $m + n - 1$ (modified set-set) down to exactly one.

6.2.3.3. Using godslist to determine mapping probability

Another interesting preprocessing use for godslist is to determine whether or not a requested mapping is actually possible, given the current topology of the two map units, as well as the matches in godslist. It is possible to test two map units for a property known as **compatibility** with godslist. This idea is closely related to the simpler idea of the order of the fragment lengths.

Once a godslist is known for two map units, it would be possible to order this list based on the first fragment of each pair. One way of ordering would be to order the pairs of godslist in the same order that the first fragment of each appears in the first map unit. After this is done, the second map unit could be compared to the second fragment of each pair. If the order that these fragments appear in the now ordered godslist matches the order that they appear in in the second map unit, then a mapping is possible.

A second case would be that the order of the fragments in godslist is exactly the reverse of the order of the fragments in the second map unit. If this is the case, it is simply necessary to flip the second map unit and then a possible mapping could exist. The final case is that the two orders are completely different, in which case no mapping can exist between these two map units, since the validity of godslist is unquestionable by the software.

Unfortunately, this *simple* idea of order is not sufficient to determine compatibility. The specific order of the fragments within any particular set is irrelevant. A simple ordering of all the fragments is not able to take this into account, and would fail to identify some valid mapping attempts as such. Due to this

limitation, the idea of compatibility is used instead of order.

Two map units are said to be compatible with a given godslist if and only if the godslist could have been created from both map units. Using this idea, the rest of the checking routine is identical to the one presented for ordering of godslist. The two map units are checked for compatibility against godslist. If they are compatible, then there is a chance of producing a valid mapping. If they are not compatible, then the second map unit is flipped, and the test is performed again. If they are now compatible, then the new orientation may produce a valid mapping. If they are still incompatible, then no valid mapping can exist for these two map units given this godslist. By spending the small amount of time needed to check godslist compatibility, map units with no hope of mapping together are eliminated without any calls to `topological_recurse`. This can translate into very significant time savings.

7. Results of the methods presented

In order to examine the abstract ideas developed in this paper, the software was run on a variety of test cases. Although one can never *prove* a general idea by examining test cases, test runs allow valuable knowledge to be discovered about the concept.

7.1. Sources of data used

There were two sources of data used for the test cases. One was actual data collected from Maynard Olson's laboratory, for which maps had been created. These data were useful due to the fact they came from the "real world." However, it suffered from certain inherent limitations. Part of the reason for running test cases is to form some sort of idea about the accuracy of the results. The problem with this first set of data is that the true solution is not actually known. Since DNA mapping is a statistical puzzle, it is only possible to know that the answer produced from these past mappings has a high probability of being correct. If the topological algorithm produced different results, these results may be invalid or may be as valid as the ones produced earlier by the Olson laboratory.

In light of this, a DNA simulator was developed for use in the DNA mapping group. This was used to produce test cases as well. There is one major advantage of this type of data over the other; since the creation of the map unit data is simulated, it is possible to determine the true map. It becomes possible, then, to gauge the true strengths and weaknesses of the topological algorithm.

7.2. Results of real-world data

The tests performed on the real-world test cases were the most successful. There were a total of nine separate test cases, ranging from lists containing as little as four clones, up to lists containing thirteen clones. The results from the first test runs are shown in Table 2; all times are shown in seconds on a Sun 3/60.

These cases show several interesting facts about the algorithm. First, speed is not directly related to the number of clones involved. Instead, the time required for a mapping is directly related to the number of possible mappings which can exist within a set. This, in turn, is the result of the number of fragment lengths which match between any two map units. Because of this, the number of map units to be mapped

```
        Methods
====================================
A :  linear, m X n
B :  tree,   m + n - 1
C :  sort decreasing length, then A
D :  sort decreasing length, then B
```

# of clones	A	B	C	D	Successful
5	5.32	11.79	3.52	2.28	yes
7	6.02	5.30	3.94	4.86	yes
4	102.00	4.20	6.66	3.66	yes
5	585.58	45.54	99.99	99.68	yes
13	19.86	18.56	42.00	25.80	yes
10	23.58	37.08	270.90	50.68	yes
11	17.44	6.74	235.00	330.78	3 islands
11	5.62	8.26	81.82	85.64	2 islands
10	36.08	6.22	28.04	11.34	3 islands

Table 2
Results from initial test runs

is not an accurate method of estimating time requirements.

Second, the algorithm proved fairly effective. On this initial run, six of the nine sets of data were successfully resolved by the software. Human examination revealed that the remaining three data sets failed to map together due to fragments being slightly more than 3% from each other in length. By altering the error window parameter, it was eventually possible to map all nine of these test data sets successfully.

Last, as expected, the tree method of attempting a mapping proved to be faster, in general, than the linear method. In some cases, this difference was significant (for example, 585.85 seconds to 45.54 seconds). There were three cases where the time for the tree method actually increased in comparison to the linear method. Still, the tree method performed better overall. Also, sorting the sets proved effective in many cases, but there were some significant degradations of performance (most notably 6.74 seconds becoming 330.78 seconds after sorting).

To summarize these initial test cases, it is obvious that the software is effective. It is also obvious that no one method seems assured of always being the most efficient. From these results, though, method B seems to provide the best chances for success.

After these initial test runs, the software was modified by implementing the backtrack limiting idea discussed earlier in Section 5.4. By setting a backtracking level of 0, only mappings with the longest matchlists will be created. This greatly cuts down on the number of map units produced. It eliminates a good number of postprocessing tests for equivalency and subsetness. In general, the speedups from this method will be phenomenal.

Of course, nothing is gained without a price. The price in this case is the decreased number of successful mappings. Instead of six of the data sets immediately leading to valid mappings, only four did. However, these four were discovered in a fraction of the time it took using the original method. Again, the human always has the opportunity to alter the parameters and remap a data set. Because of this, all nine data sets can still be successfully mapped; it simply requires human intuition and intervention. Thus, data sets that can be mapped together easily can be mapped quickly. More difficult data sets will require more time and human intervention, but time has been saved, overall, by eliminating the simpler cases.

7.3. Results from simulator data

The results from the simulated data were less encouraging, but enabled several problem areas to be discovered. Test cases were run on simulator data containing 16, 32, and 64 clones. The 16 clone set proved no problem. However, neither the 32 nor the 64 clone set mapped successfully. Careful examination of the data revealed two sources of problems.

First, there existed more subclones in the data than expected. This cut down on the number of valid clones available for the initial mapping attempt. Second, the normal random error introduced into the fragment length data often resulted in a greater than 3% difference in fragment lengths. This, of course, led to fragment pairings being overlooked. When the error window was expanded to allow these matches to form, too many other fragments matched as well. This caused ambiguous solutions to be produced. A proper balance was never obtained.

The simulator test cases reemphasize the earlier point of combining the abilities of a human and the software. Although neither of the larger data sets was successfully mapped, the preliminary results which the software was able to obtain would reduce the amount of work a human would have to do in order to finish the mapping. Again, this is the utilization of the software which should prove the most effective and efficient in the majority of mapping cases.

8. Conclusion

DNA mapping is a complex and intuitive process. The fact that it relies on a great deal of data manipulation implies that it can be computerized, at least to some degree. The method of topological mapping presented in this paper is a promising one. The results it is capable of producing are encouraging. While the current implementation of it is not guaranteed to perform correctly in all situations, it has shown itself to be a valuable tool.

The software exists in a modularized form. When better top-level algorithms for ordering the map unit data are developed, they can easily be linked into the existing framework. Whether or not the degree

of accuracy reached by the topological mapping algorithm ever reaches that of a human remains to be seen. However, this was not the primary design consideration for this software. Its goal was to provide a product capable of aiding a human mapper, producing a combination which is more effective and accurate than the sum of its parts. In this respect the software is a success, even in its current state. Future modifications will improve its abilities even further.

Many of the ideas developed during the creation of this software have already been transferred to other forms of DNA mapping. For instance, the concept of subclone/superclone is now used elsewhere, and the strategies presented in Figures 18 and 19 have been incorporated as a general approach.