A Flexible Approach to Genome Map Assembly

Eric Harley
eharley@db.toronto.edu

Anthony J. Bonner
bonner@db.toronto.edu

University of Toronto
Department of Computer Science
Toronto, Ont, Canada M5S 1A4

Abstract

A major goal of the Human Genome Project is to construct detailed physical maps of the human genome. A physical map is an assignment of DNA fragments to their locations on the genome. Complete maps of large genomes require the integration of many kinds of experimental data, each with its own forms of noise, experimental error and anomalies. To facilitate the assembly of physical maps, we are developing a flexible approach to map assembly based on logic programming and data visualization. Logic programming provides a convenient and mathematically rigorous way of reasoning about data, while data visualization provides layout algorithms for assembling and displaying genome mapping data. The approach has several benefits: (*i*) it frees the user (or programmer) from data management problems, (*ii*) it allows him to see anomalies in the data visually, (*iii*) it helps him to rapidly develop and test rules for filtering out anomalous data and for combining data from different sources, and (*iv*) it displays a genome map as a graph with linear structure. To demonstrate the approach, this paper describes numerous rules for map assembly implemented in a data-visualization system called Hy+. Using these rules, we have successfully assembled contigs (partial maps) from real and simulated mapping data—data that is noisy, imprecise and contradictory.

Appears in *Proceedings of the International Symposium on Intelligent Systems for Molecular Biology (ISMB)*, pages 161–169. AAAI Press, Menlo Park, California. Held at Stanford University, August 14–17 1994.

This paper is available at the following URL:
ftp://db.toronto.edu/pub/bonner/papers/genome.mapping/ismb94.ps.gz

1 Introduction

A major goal of the Human Genome Project is to construct detailed physical maps of the human genome. A physical map is an assignment of DNA fragments to their locations on the genome. In assembling a physical map, a genetics expert typically reasons about the experimental data and how it fits together. The problem is to construct a coherent picture of the genome from data that is incomplete, imprecise, ambiguous and often contradictory. Usually, only an approximate map can be constructed, and sometimes only the relative order of the DNA fragments can be determined. For small-scale mapping projects, maps are often constructed manually, perhaps with the aid of a graphical editor. This is a tedious and time-consuming task. Large-scale mapping projects require better computational tools to efficiently handle the large volumes of data and the explosive combinatorics of the map-assembly problem.

Most existing work in computer-aided map assembly lies somewhere between two extremes: specialized editors and automatic map-assembly algorithms. Editors provide a convenient way to rearrange mapping data, but provide little or no reasoning capabilities. The SIGMA system developed at Los Alamos National Laboratories is an example of such an editor. Even with a good editor, however, constructing a genome map is a tedious and time-consuming task. Algorithms for map assembly alleviate this tedium by rearranging the data automatically. However, most current algorithms are limited to a narrow range of data. Some algorithms make strict assumptions about the data and the errors (sometimes assuming no error at all) in order to achieve mathematical proofs of correctness and optimality [3, 8, 11]. Map-assembly programs used at large genome centers make more realistic assumptions, but they can be inflexible, monolithic programs that are hard to modify or extend. The MAPMAKER program for genetic mapping, developed at the Whitehead/MIT Centre for Genome Research, is an example of such a program [10]. Programs for automatic assembly of *integrated* physical maps will be even more complex.

The reason for this complexity is that mapping data comes in a wide variety of forms, each with its own forms of imprecision and experimental error. Programs for integrating data into a single genomic map should therefore be flexible, so they can easily accommodate many forms of data, including new forms of data as they are developed. To address this need, we are developing a new approach to map assembly based on logic programming and data visualization. Logic programming provides a convenient and mathematically rigorous way of reasoning about data, while data visualization provides layout algorithms for assembling and displaying genome maps. The goal is to permit new ideas and techniques for map assembly to be rapidly implemented and tested with a minimum of programming effort. Logic programming is already known to facilitate the rapid prototyping and testing of software. Within our framework, contigs (partial maps) are assembled not by writing programs, but by specifying deductive rules for the logic of map assembly. The framework automatically translates these specifications into programs.

Logical rules can encode much of the biological knowledge used in assembling physical genome maps. These rules have a premise (*if* part), and a conclusion (*then* part). For example,[1]

if *two STS probes hit a common YAC,*
then *the two probes are close together on a chromosome.*

Logical rules can also integrate different kinds of mapping data, such as STS and fingerprint data. For example,

if *probe* s_1 *hits YAC* y_1*,*
and *probe* s_2 *hits YAC* y_2*,*
and *the fingerprints of* y_1 *and* y_2 *overlap,*
then *probes* s_1 *and* s_2 *are close together on a chromosome.*

This is the kind of reasoning that a biologist would employ in a small-scale mapping project. Logic programming provides a convenient and mathematically-rigorous way to automate this kind of reasoning for mapping projects of any size.

Even when the experimental data is flawed, it often contains useful information that can be extracted with logical rules. For example, a YAC insert may be *chimeric*, containing two DNA fragments from different parts of the genome. For data containing these *chimers*, two STS probes are considered close if they hit *two* common YACs (This is the double-linkage strategy of [4]). Other rules can specify how to resolve data ambiguities. For example, the Whitehead/MIT Genome Center has found that the pooling scheme used in their YAC screening leads to a high rate of false negatives. This often makes it impossible to say precisely which YAC is hit by an STS probe, though it is possible to say that the probe hits one of a small set of 8 to 12 YACs. This paper gives deductive rules for dealing with such ambiguities.

To test our approach, we have encoded rules for map assembly using the Hy+ data visualization system. Developed at the University of Toronto [6], Hy+ provides a graphical user interface to a number of logic-programming systems, including PROLOG, CORAL and LDL [12, 15]. It also has a number of algorithms for graph layout. Unlike many data visualization systems, Hy+ represents *both* logical rules and query answers as graphs. Using logical rules, Hy+ transforms mapping data into a graph, which is then displayed using layout algorithms. Each contig appears on the screen as a connected component with a linear structure.

Using Hy+, we have been able to rapidly implement and test numerous rules for assembling and exploring physical maps. We have tested these rules on real and simulated data provided by the Whitehead Institute/MIT Centre for Genome Research. The real data consists of their December 1993 release of 733 STS probes screened on the CEPH YAC library. The simulated data represents 876 STS probes screened on 2,490 YAC's, including 5,388

[1] As explained in more detail later, an STS is a Sequence Tagged Site, and a YAC is a Yeast Artificial Chromosome. Different laboratory techniques can be used to determine if an STS probe "hits" a YAC, such as hybridization or PCR.

hybridizations and 1,766 ambiguous hybridizations. To test and refine our approach, the Hy+ system has been installed at Whitehead/MIT, where it will be used in their large-scale mapping projects.

An important aspect of our work is that Hy+ is a *general* tool for querying and visualizing data, and was *not* designed with genomes or biology in mind. In fact, the initial applications of the Hy+ system were in software engineering and network management [6]. In our application, all biological knowledge is embodied in logical rules. Display of the resulting graphs is carried out by the graph layout algorithms in Hy+. These algorithms know a lot about graphs, but nothing about biology. In this way, we can assemble contigs with a minimum of programming effort by using a flexible and general-purpose database package. Indeed, the only programming involved is using a mouse to draw graphical patterns representing logical rules. By drawing different patterns, we can quickly and easily test the effect of different map-assembly rules. In addition, the query and data visualization facilities of Hy+ facilitate the exploration and debugging of genome maps, allowing a user to quickly locate interesting or problematic regions in a map. This is comparable to the use of Hy+ in understanding and debugging large software systems [6].

To illustrate our approach to map assembly and analysis, this paper focuses on a particular kind of physical map, called an STS content map. We show how experimental error in this data can be accommodated (and even exploited) by rules that account for the biological origins of the errors. We illustrate the effect of the rules on synthetic but realistic STS data provided by the Whitehead Institute/MIT Center for Genome Research [13]. This data contains simulated noise and experimental error, including chimers and false negatives. It was made available expressly for the purpose of testing new logic-based approaches to map assembly. In [13], a number of queries to physical maps are suggested, queries that an investigator might reasonably ask. We show how to answer these queries, assemble contigs, and more.

2 STS Content Mapping

The aim of STS content mapping is to determine the order of STS probes along a chromosome. Each STS probe is a very short piece of DNA that "sticks" to a particular site on the chromosome (a Sequence Tagged Site, or STS). We do not know the exact location of each STS, but we would like to know their relative order. In a detailed map, there are thousands of sites per chromosome.

The usual first step is to determine which pairs of probes are "close" together on the chromosome. To do this, the experimenter randomly cuts many copies of the chromosome into small fragments, called YACs (Yeast Artificial Chromosomes). If two STS probes "stick" to the same YAC, then one can infer that the two probes are close together, since they are no more than one YAC-length apart. This, at least, is the ideal situation. As we shall see shortly, because of noise and ambiguities in the data, additional reasoning is often needed to infer proximity. This kind of reasoning is easily automated as a logic program.

2.1 The Data

In a perfect experiment, the data tell us whether a probe "sticks" to or "hits" a YAC. This can be determined by different lab techniques, such as hybridization or polymerase chain reaction (PCR). The experimental results are not always clear cut however. For example, the MIT genome centre has found that the pooling scheme used in their YAC screening leads to a high rate of false negatives. This often makes it impossible to say precisely which YAC is hit by an STS probe, though it is possible to say that the probe hits one of a small set of 8 to 12 YACs. The data provided to us by Whitehead/MIT are comprised of two kinds of tuples:

- $hit(P, Y)$, meaning probe P definitely hits YAC Y,
- $amb\_hit(P, Yset)$, meaning probe P hits one of the YACs in the set $Yset$.

The relation *amb_hit* contains more information than is needed for the purposes of this paper. To simplify the presentation (and to avoid dealing with nested relations), we have transformed each tuple of the form $amb\_hit(P, Yset)$ into a set of tuples of the form $amb(P, Y)$, which means that probe P ambiguously hits YAC Y, (*i.e.*, probe P *might* hit YAC Y).

All of the logic programs described in this paper were tested on this data. The data is not ideal and includes numerous false negatives and chimers. The data includes 5,388 tuples of the form $hit(Y, P)$, involving 876 probes and 2,490 YACs. It also includes 16,520 tuples of the form $amb(P, Y)$, derived from 1,766 tuples of the form $amb\_hits(P, Yset)$. Figure 1 shows a graphical representation (using Hy+) of the *hit* and *amb* relations for two probes, $sts(874)$ and $sts(875)$. In the figure, the nodes on the left represent the two STS probes, and the nodes on the right represent the YACs which are hit or ambiguously hit by these probes. The rectangular insert in the figure shows a graphical pattern, which is the query that generated the graph. The next sections explain such queries.

3 Hy+

Hy+ is a data visualization system based on a generalization of labeled directed graphs, called hygraphs [5, 7]. Hygraphs can be effectively used to organize the visual presentation into a more informative one than would be possible in normal graphs. The user interface to Hy+ is a menu driven windowing system offering many graphical facilities and color options for the visual display of data and relations. The front end, written in Smalltalk, communicates with other programs including database backends written in PROLOG, CORAL and LDL [12, 15], which evaluate queries. Hy+ offers visual facilities for filtering the data to be displayed, and for defining new relations on the data. These query facilities are written in a graphical query language called *GraphLog* [5, 7].

A *graphical query* in GraphLog is composed of *define graphs* and *show graphs*. A define-graph-query defines new graphical relations, and a show-graph-query filters the data before presenting it to the user. Briefly, a define graph defines a logical rule of inference. The graph

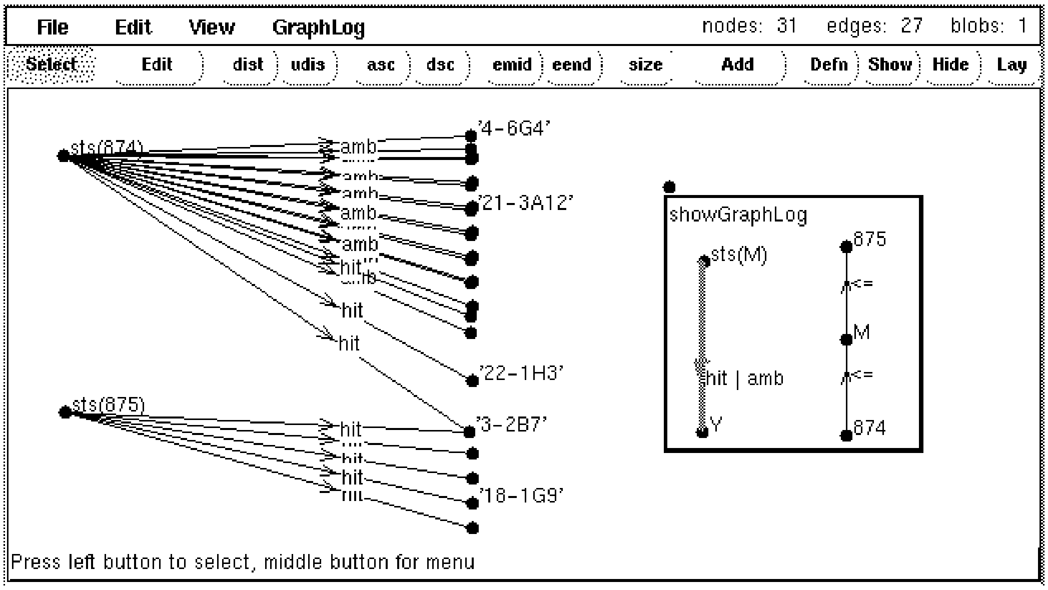

Figure 1: The hit and ambiguous-hit data for two probes.

has a single *distinguished* edge, which appears as a bold arc. If this arc is labeled $r(Z)$, and connects nodes X and Y, then the graph defines a Horn rule whose head is the atomic formula $r(X, Y, Z)$. Likewise, each undistinguished edge in the graph contributes an atomic formula to the premise of this rule. The translation from query graphs to Horn rules is described in detail in [5, p. 44]. Intuitively, a define graph says that if the undistinguished edges appear in a graph, then the distinguished edge should be added to the graph. In contrast to define graphs, a show graph may have more than one distinguished edge. A show graph has the effect of filtering the data: only distinguished nodes and edges in the show graph are displayed to the user. To reduce the number of define graphs, an edge may be labelled with a *path regular expression*. Rather than define these facilities precisely here, the next section illustrates some of them through examples of the assembly and analysis of mapping data.

4 Inference and Visualization

This section illustrates the use of the Hy+ system for assembling, visualizing and interrogating STS content maps. All examples in this section use the simulated data described above and were run on a SPARC station 10 workstation.

4.1 Proximity of Probes

The usual first step in an STS mapping project is to determine which probes are close together on a chromosome. This can often be inferred directly from the experimental data, but noise and ambiguity in the data can complicate the process. Each kind of data has its own kind of noise, and each must be given special treatment by a map-assembly program.

In the simplest case, if two probes, p_1 and p_2, both hit the same YAC, y, then we immediately infer that p_1 and p_2 are close together on the chromosome. In logic programming, this rule of inference can be written as follows:

$$\begin{aligned} close(P_1, P_2) \quad \leftarrow \quad & hit(P_1, Y), hit(P_2, Y), \\ & P_1 \neq P_2. \end{aligned} \tag{1}$$

Although simple, this inference is warranted only under ideal experimental conditions. Such conditions do not always hold. This is the case, for instance, if the YAC library is *chimeric*, *i.e.*, if a YAC may be a concatenation of two DNA fragments from different parts of the genome.

In such cases, we can still infer proximity of two probes, but we need different rules of inference. For example, if two probes hit *two* common YACs, then there is a very high probability that the two YACs are not chimeric and that the probes are indeed close together. In this case, we have two distinct probes, p_1 and p_2, two distinct YACs, y_1 and y_2, and four tuples in the hit relation:

$$\begin{array}{ll} hit(p_1, y_1) & hit(p_1, y_2) \\ hit(p_2, y_1) & hit(p_2, y_2) \end{array} \tag{2}$$

(This is the double linkage strategy of [4]). In logic programming, this inference can be written as the following rule:

$$\begin{aligned} close(P_1, P_2) \quad \leftarrow \quad & hit(P_1, Y_1), hit(P_1, Y_2), \\ & hit(P_2, Y_1), hit(P_2, Y_2), \\ & Y_1 \neq Y_2, P_1 \neq P_2. \end{aligned} \tag{3}$$

Hy+ represents such rules of inference as graphical patterns. These patterns act on one graph to produce another graph. The two boxes on the right side of Figure 2 show how the rule above is expressed visually in Hy+. These boxes show two graphical patterns. In the top box (the *defineGraphLog* box), nodes $P1$ and $P2$ denote probes, and nodes G and F denote YACs. The crossed out edges mean that the two probes are distinct, and the two YACs are distinct. The four edges labeled *hit* specify that both probes hit both YACs. If the data fits this pattern, then Hy+ will add a new edge to the graph, the bold edge labelled *hh* between $P1$ and $P2$. Intuitively, this edge means that probes $P1$ and $P2$ are close together. (The label *hh* reminds us that the inference comes from a double hit.) The pattern in the bottom box (the *showGraphLog* box) has a single bold edge labeled *hh*. This pattern specifies that only edges labelled *hh* should be displayed to the user. The two patterns are evaluated by Hy+ in about 3 minutes on the simulated data described above. The result is the graph of connected components shown on the left side of Figure 2. The Hy+ overview browser has pan and zoom buttons, so only a few of the connected components are visible in the figure. Each node in the graph represents an STS probe, and an edge means that two probes are close together on a chromosome.

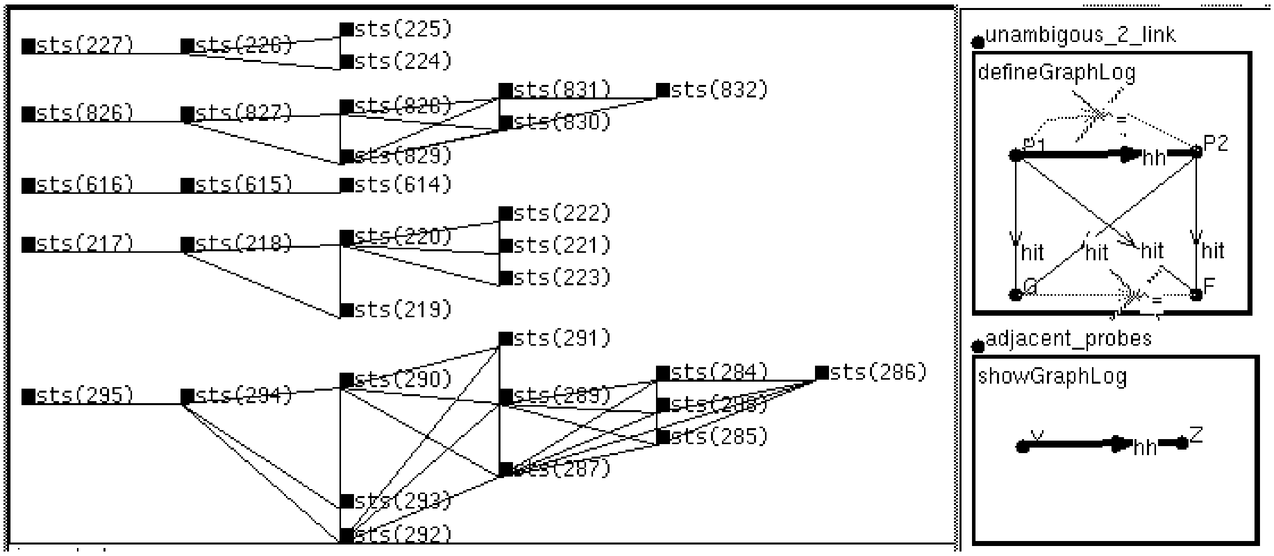

Figure 2: Inferring adjacent STS probes

4.2 Contig Assembly

The rules discussed above infer which probes are close together on a chromosome. Using this proximity information, the next step is to assemble a map of the chromosome, *i.e.*, to infer the relative order of the probes on the chromosome. Unfortunately, there is often insufficient data to infer a complete map. Typically, an experimenter will be able to determine the order of probes along certain contiguous regions of the chromosome ("contigs"), but there will be other regions ("gaps") about which nothing is known, because none of the probes stick to these regions.

This situation is apparent in the graph on the left side of Figure 2. In this graph, each connected component represents a distinct contig; and each simple (loop-free) path through a component is a potential map of the contig. Notice that each component has a somewhat linear structure. The structure is not completely linear because the data and/or analysis do not yet support a unique linear order of the probes on the contig. The data also do not tell us how one contig is related to another, *i.e.*, which is first, second, third, etc. We thus have a partial map with numerous contigs, but we do not yet know the location (or orientation) of each contig on the chromosome.

The graph in Figure 2 has 233 connected components, representing 233 contigs. Since the graph has 876 nodes (probes), the average contig contains 3.8 probes. A small number of large contigs would be better, and the ideal is a single huge contig, *i.e.*, a completely connected graph representing a complete map of the chromosome. Recall that the graph in Figure 2 was generated by rule (3). Rule (1) should lead to a smaller number of contigs, since it demands less evidence before concluding that two probes are adjacent. Indeed, the graph that it generates has only 69 connected components, with an average of 12.7

nodes each. Unfortunately, because of chimerism, we cannot be confident that each of these components represents contigs. In the next subsection, we explore a more reliable way of producing larger contigs.

4.3 Ambiguous Data

Rule (3) above provides a way of dealing with certain problems in experimental data (chimerism). However, this rule only uses *unambiguous* data, *i.e.*, data stored in the relation $hit(P, Y)$. There is not always enough data in this relation to infer that two STS probes are close together. In such cases, we can exploit *ambiguous* data, *i.e.*, data stored in the relation $amb(P, Y)$.

A simple way of doing this is suggested in [13]: if one of the four *hit* tuples in (2) above is actually an ambiguous hit, then we can still infer that probes p_1 and p_2 are close. We can represent this idea in logic programming by the following three rules:

$$\begin{array}{rcl} close1(P_1, P_2) & \leftarrow & amb(P_1, Y_1), hit(P_1, Y_2), \\ & & hit(P_2, Y_1), hit(P_2, Y_2), \\ & & \quad Y_1 \neq Y_2, P_1 \neq P_2. \\ close(P_1, P_2) & \leftarrow & close1(P_1, P_2) \\ close(P_2, P_1) & \leftarrow & close1(P_1, P_2) \end{array} \tag{4}$$

Unlike rule (3), the first rule above is not symmetric in P_1 and P_2. This is why we first define $close1$ (which is not symmetric), and then define *close* (which is symmetric). Intuitively, the second two rules say that if P_1 is close to P_2, then P_2 is close to P_1.

We now have two reliable ways of inferring that two probes are close together, depending on whether the probe hits are unambiguous or not. Rule (3) is based on unambiguous hits, and rules (4) are based on both ambiguous and unambiguous hits. Using Hy+, we combined all four of these rules into a single query. The resulting graph has 161 connected components, representing 161 contigs with an average size of 5.4 probes each. We thus get larger contigs than by using rule (3) alone, which gives an average contig size of only 3.8 probes. This improvement comes from exploiting information implicit in the ambiguous data.

4.4 Probable Proximity

As the examples above show, the experimental data may lead to a large number of small contigs. This is a long way from a complete physical map, *i.e.*, a single large contig. However, by being less rigorous with the evidence, we can create a smaller number of larger contigs. (Additional experiments could then confirm or refute these putative contigs.) One way to do this is to search the database for evidence suggesting that two probes are *probably* close on the chromosome, from which we can infer a set of probable contigs [13].

The first step is to find evidence that an ambiguous hit is actually a probable hit. To see how, suppose that probe p ambiguously hits YAC y. We infer that this is a probable hit if (i) a second probe, q, also hits y, and (ii) both probes hit a second YAC, w. This inference can be expressed as a logic-programming rule as follows:

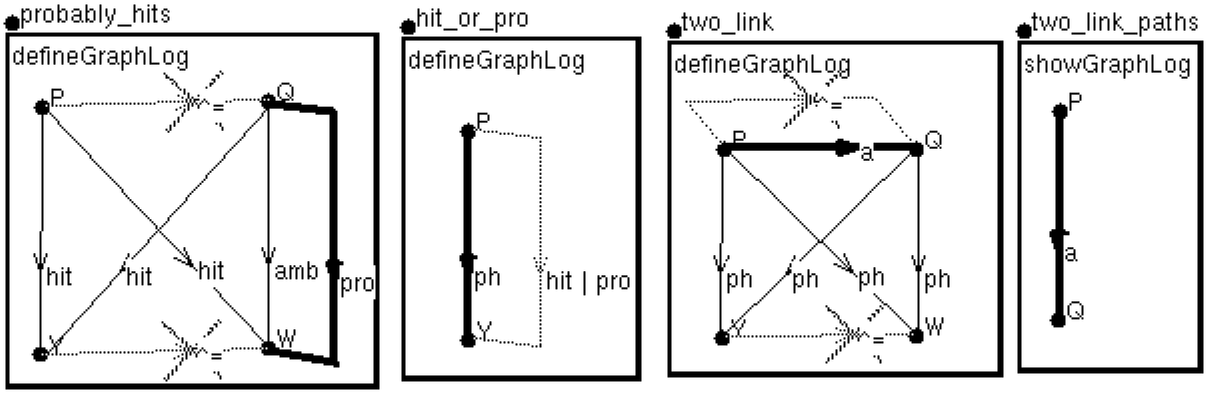

Figure 3: A query defining probably-adjacent probes

$$
\begin{aligned}
probHit(P, Y) \quad \leftarrow \quad & amb(P, Y), hit(Q, Y), \\
& hit(P, W), hit(Q, W), \\
& \ \ Y \neq W, P \neq Q.
\end{aligned}
$$

In Figure 3, the box labelled *probably_hits* shows a graphical pattern that expresses this rule. The pattern adds an edge labelled *pro* from probe P to YAC Y if P probably hits Y.

Using the notions of hits and probable hits, we can infer when two probes are probably close on a chromosome. Previously we defined two probes to be close if they both hit the same two YACs. Likewise, we define two probes to be probably close if they both hit or probably hit the same two YACs. This notion is defined by the following rules:

$$
\begin{aligned}
ph(P, Y) \quad &\leftarrow \quad hit(P, Y) \\
ph(P, Y) \quad &\leftarrow \quad probHit(P, Y) \\
probClose(P, Q) \quad &\leftarrow \quad ph(P, Y), ph(Q, Y), \\
& \qquad ph(P, W), ph(Q, W), \\
& \qquad \ \ Y \neq W, P \neq Q.
\end{aligned}
$$

The remaining boxes in Figure 3 show graphical patterns that express these rules. The *hit_or_pro* box adds an edge labelled *ph* from a probe to a YAC if the probe either hits or probably hits the YAC. The box labelled *two_link* then adds an edge labelled a (for "adjacent") between two probes if they hit or probably hit two YACs. Finally, the box labelled *paths* specifies that only edges labelled a should be displayed.

These rules use more of the ambiguous data than rules (3) and (4) do. They can therefore infer more adjacency edges between pairs of probes; so they can generate graphs with fewer, but larger, connected components. This can be seen in the table below, which represents a set of ambiguous and unambiguous hits. Each column represents a probe, and each row represents a YAC.

	p_1	p_2	p_3	p_4
y_1	*hit*	*amb*	*hit*	
y_2	*hit*	*hit*		
y_3		*hit*	*amb*	*hit*
y_4			*hit*	*hit*

Using rules (3) and (4), we can only infer two adjacencies, $a(p_1, p_2)$ and $a(p_3, p_4)$; but using the rules of this section, we can infer a third (probable) adjacency, $a(p_2, p_3)$. The former set of rules therefore generates an adjacency graph with two connected components, while the latter set generates one large component. This behavior has a simple biological interpretation: we are using high quality data to form contigs, and lower quality data to merge contigs. That is, rules (3) and (4) infer contigs based largely on unambiguous data. Using more of the ambiguous data, the rules of this section infer the probable order of one contig relative to another.

The query of Figure 3 when run on the simulated data described above generates 1,353 *pro* edges. The amount of time required to execute the query varies depending on the order Hy+ chooses for the predicates in the corresponding logical rules. When the query is rewritten to force an optimal order of evaluation of joins in the rule bodies, the time stabilizes to about 9 minutes. The result is a graph similar to that on the left side of Figure 2. Each simple path in this graph is called a 2-linked YAC path in [13], and is a potential map of the contig. There are 161 connected components in the graph, representing 161 probable contigs with an average size of 5.4 probes.

4.5 Inferring Probe Order

In the graph of Figure 2, any (simple) path between two probes is a possible ordering of the probes. Mere possibility, however, is a weak conclusion. We would prefer to know what ordering relationships are *necessary*, *i.e.*, are implied by the data. A number of strategies of different complexity can be used to extract these relationships. For simplicity, we illustrate an approach that works with good experimental data. If the data contain no false negatives, or if we have a sufficient amount of data, then a *minimal* path between two probes expresses only necessary relationships. A minimal path may not include all the probes in a contig, but for those that it does include, we shall know their relative order on the chromosome.

Figure 4 shows a query that defines the minimal paths between two probes, $sts(286)$ and $sts(295)$ in the graph of Figure 2. The query uses four define graphs to express a variation of Dijkstra's shortest path algorithm [1]. The first two boxes (*define_start* and *count_edges*) compute the shortest paths from the start node (sts(286)) to every other node in the same connected component. This is done by adding arcs labelled $b(M)$ to the graph. Intuitively, such an arc from node P to node Q means that beginning at the start node, the shortest paths to node Q have length M, and furthermore, P is the predessesor of Q on one of these paths. These edges are inferred recursively as follows. First, add an edge labelled $b(1)$ from the start node to each of its neighbours. Second, for any node pointed to by $b(M)$, add an

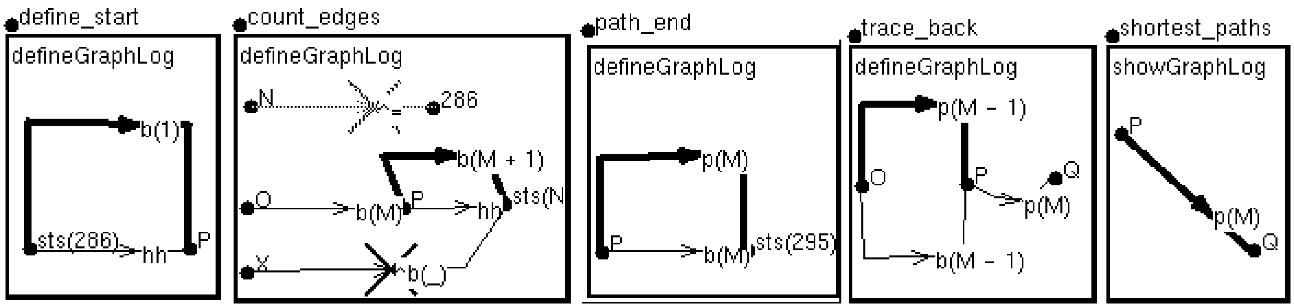

Figure 4: A shortest path query

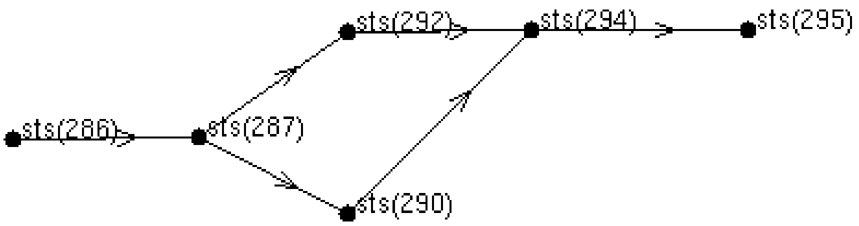

Figure 5: The shortest paths between probes sts(286) and sts(295)

edge labelled $b(M+1)$ from this node to each neighbouring node (other than the start node) that is not already pointed to by a b edge. The second two boxes in the query (*path_end* and *trace_back*) isolate the shortest paths from the start node to a *particular* end node (sts(295)) by tracing the path backwards: each time an edge labelled $b(M)$ is traversed, it is labelled it with $p(M)$. Finally, the show box specifies that only those edges on a minimal path (*i.e.*, those edges labelled $p(M)$) should be displayed. The result is shown in Figure 5. In this case, there are two minimal paths.

4.6 Exploratory Queries

Until enough data (and the right data) are generated, a physical map of a chromosome will remain incomplete, and we will not know the order of the STS probes. In Figure 2, for instance, we do not know the relative order of the contigs on the chromosome, and we do not know the relative order of the probes on each contig. Several possible orders are consistent with the data. In fact, any simple path through a connected component is a possible map of the corresponding contig. Not all probe orderings are equally likely however. In some orderings, the links between probes may be weak, or the ordering may imply too many false negatives in the data. Given a proposed probe ordering, we can use Hy+ to highlight its weaknesses.

For example, our data is not good enough to infer unambiguously the order of probes $sts(868)$ to $sts(874)$. However, let us suppose hypothetically that their order is $sts(868)$,

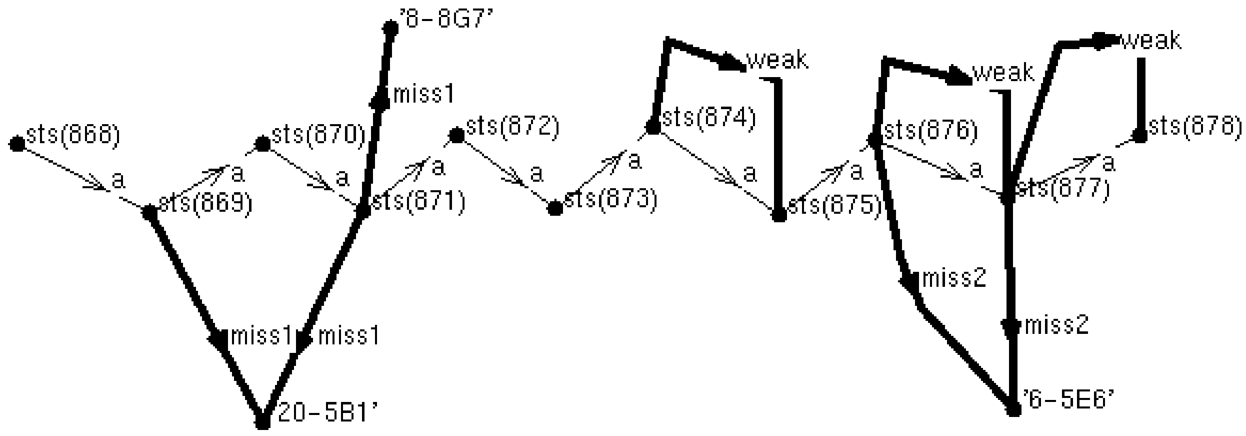

Figure 6: Evaluating a contig hypothesis

sts(869), *sts*(870), ... *sts*(878). Figure 6 shows this proposed contig and some of the problems with it. In this figure, the path whose edges are labelled *a* (for "adjacent") represents the proposed contig. Other edges represent potential problems. For instance, edges labelled *weak* indicate that two probes are not tightly linked; that is, there is little evidence to support the hypothesis that they are adjacent.[2] Three of the ten edges in Figure 6 are weak.

Edges labelled *miss*1 and *miss*2 indicate false negatives. Unlike other edges, these edges start at a probe and terminate at a *YAC*. An edge labelled *miss*1 indicates that the probe does *not* hit the YAC but *should* hit it according to our proposed ordering. A probe should hit a YAC if its left and right neighbours hit it. For example, in Figure 6, probe *sts*(869) does not hit YAC 20-5B1, but should, since its two neighboring probes, *sts*(868) and *sts*(870), both hit this YAC. If our proposed probe ordering is correct, then there must be a false negative in our experimental data. The more false negatives we find, the less likely it is that our probe ordering is correct. Figure 6 indicates five false negatives. Notice that probe *sts*(877) is particularly problematic. Not only is it connected by two weak edges, but it is associated with two false negatives. This would suggest correcting our proposed probe ordering by removing probe *sts*(877) from the contig.

The graph in Figure 6 was constructed by writing visual queries in Hy+. Other inconsistencies between the experimental data and a proposed map can easily be highlighted by constructing other Hy+ queries.

5 Summary and Discussion

We are developing a flexible approach to physical map assembly, an approach based on logic programming and data visualization. To demonstrate the advantages and the potential of

[2]In particular, they are not inferred to be close by rules (3) and (4).

the approach, we have implemented numerous rules for map assembly in the Hy+ data visualization system. We illustrated the use of Hy+ in assembling and analyzing contigs from simulated STS content data. We showed how to accommodate (and even exploit) noise in the data by using rules that encode biological knowledge. By using simulated data, we can compare our contigs to the correct probe order, and the results are encouraging.

The rules in this paper addressed forms of noise and experimental error that are relatively discrete. More generally, rules for probabilistic inference are needed. One could then reason about continuous noise and about known rates of false positives, false negatives, and chimerism. Probabilistic inference can be dealt with by logical expressions of the form $q(x) : p$, which intuitively means that statement $q(x)$ is true with probability p. These ideas have been extensively investigated in the Logic Programming and AI communities, *e.g.*, [9].

It should be noted that Hy+ was designed and built without genomes or biology in mind. In fact, its initial applications were in software engineering and network management [6]. In our application, all biological knowledge is encoded in graphical patterns, like those in Figure 4). Hy+ translates these patterns into logical rules, which are then passed to a logic-programming system for evaluation. In this paper, the resulting inferences provide information on what probes are close together on a chromosome. Contig assembly itself is carried out by graph layout algorithms in Hy+. These algorithms know a lot about graphs, but nothing about biology. The layout algorithm used in this paper isolates the connected components of a graph and displays them in a linear manner. In this way, contigs can be assembled with a minimum of programming effort by using a flexible and general-purpose database package. Indeed, the only programming involved is to use a mouse to draw graphical patterns (*i.e.*, rules). By drawing different patterns we can quickly and easily test the effects of different map-assembly rules.

Hy+ provides a number of logic-programming backends, including PROLOG, CORAL and LDL. Although PROLOG is the most well-known, we did not use it, for two reasons: recursive rules may not terminate in PROLOG, and PROLOG may reprove a fact many times, which can lead to gross inefficiency. Modern logic-programming systems have solved these problems. For instance, the CORAL system evaluates queries from the bottom up (forward chaining) and uses a process known as "magic sets" to provide the necessary goal-directed behavior [12]. Using CORAL, each of the queries in this paper was evaluated in a few minutes or less. Other logic-programming systems provide even more speed. For instance, the XSB system is an order of magnitude faster than CORAL [14]. This efficiency is achieved by memoing and by compiling logical rules into code for an extended Warren Abstract Machine (XWAM) [2]. We plan to install XSB as a back end for Hy+ and test its effectiveness in assembling and analyzing physical genome maps.

Acknowledgments: We gratefully acknowledge the expert advice of the following people: Alberto Mendelzon, Dimitra Vista, Mariano Consens, Masum Hasan, Gloria Kissin and Evan Steeg, at the University of Toronto; Nathan Goodman, Steve Rozen and Lincoln Stein at the Whitehead Institute/MIT Center for Genome Research; and Michael Kifer, David Warren and Terrance Swift at the State University of New York at Stony Brook.

References

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1983.

[2] H. Ait-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press, Cambridge, MA, 1991.

[3] F. Alizadeh, K.M. Karp, L.E. Newberg, and D.K. Weisser. Physical mapping of chromosome: A combinatorial problem in molecular biology. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 371–381. ACM Press, 1993.

[4] R. Arratia, E.S. Lander, S. Tavare, and M.S. Waterman. Genomic mapping by anchoring random clones: A mathematical analysis. *Genomics*, 11:806–827, 1991.

[5] M.P. Consens. Graphlog: "real life" recursive queries using graphs. Master's thesis, Department of Computer Science, University of Toronto, 10 King's College Rd, Toronto, Ont, Canada, 1989.

[6] M.P. Consens. *Creating and Filtering Structural Data Visualizations using Hygraph Patterns.* PhD thesis, Department of Computer Science, University of Toronto, 10 King's College Rd, Toronto, Ont, Canada, 1994.

[7] M.P. Consens and A.O. Mendelzon. Graphlog: A visual formalism for real life recursion. In *Proceedings of the ACM Symposium on the Principles of Database Systems (PODS)*, pages 404–416, 1990.

[8] R.M. Karp. Mapping the genome: Some combinatorial problems arising in molecular biology. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 278–285. ACM Press, 1993.

[9] M. Kifer and A. Li. On the semantics of rule-based expert systems with uncertainty. In *Proceedings of the International Conference on Database Theory (ICDT)*, number 326 in Lecture Notes in Computer Science, pages 102–117. Springer-Verlag, August/September 1988.

[10] E.S. Lander, P. Green, J. Abrahamson, A. Barlow, M. Daly, S. Lincoln, and L. Newburg. MAPMAKER: an interactive computer package for constructing primary genetic linkage maps of experimental and natural populations. *Genomics*, 1:174–181, 1987.

[11] A.J. Lee, E.A. Rundensteiner, S. Thomas, and S. Lafortune. An information model for genome map representation and assembly. Technical Report SDE-TR-163-93, University of Michigan, Dept of Electrical Engineering and Computer Science, Ann Arbor, MI 48109-2122, May 1993.

[12] R. Ramakrishnan, D. Srivastava, and P. Seshadri. Implementation of the CORAL deductive database system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 167–176, Washington, DC, May 1993.

[13] S. Rozen, M.J. Daly, M.-P. Reeve, and N. Goodman. Genome-map: Real-world test data and queries for logic databases. Whitehead/MIT Center for Genome Research, One Kendall Square, Cambridge, MA 02139. Unpublished draft, December 1993.

[14] K. Sagonas, T. Swift, and D.S. Warren. XSB as an efficient deductive database engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 442–453, Minneapolis, MN, May 1994.

[15] S. Tsur and C. Zaniolo. LDL: A Logic-Based Data-Language. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Kyoto, Japan, 1986.