

Genomic Database Applications in DISCO[★]

Peter Z. Revesz

Department of Computer Science and Engineering
University of Nebraska, Lincoln, NE 68588 USA
`revesz@cse.unl.edu`

Abstract. A genome map is an ordering of a set of clones according to their believed position on a DNA string. Simple heuristics for genome map assembly based on single restriction enzyme with complete digestion data can lead to inaccuracies and ambiguities. This paper presents a method that adds additional constraint checking to the assembly process. An automaton is presented that for any genome map produces a refined genome map where both the clones and the restriction fragments in each clone are ordered satisfying natural constraints called step constraints. Any genome map that cannot be refined is highly likely to be inaccurate and can be eliminated as a possibility.

1 Introduction

Deoxyribonucleic acid or DNA, the genetic material that encodes the blueprint of any living organism, is composed of a string of nucleotides that are adenine (A), thymine (T), cytosine (C), and guanine (G). Clones are copies of random substrings of a given DNA string. Clones may overlap in a clone database.

Restriction enzymes are enzymes that cut a nucleotide string always at specific sites.² Each different restriction enzyme cuts the nucleotide string at different sites. After cutting all information about the original order of the small fragments is lost, because the small fragments just float randomly in the restriction enzyme.³ In this paper we assume that each clone is cut at all restriction sites that are specific for the enzyme applied. This is called complete digestion. Clones can be copied easily and completely digested by various enzymes and the lengths of the fragments after the digestion can be measured.

[★] This work was supported in part by NSF grants IRI-9625055 and IRI-9632871. This work appears in *CP96 Workshop on Constraints and Databases*, Springer-Verlag LNCS 1191, pp. 234-245, Cambridge, Massachusetts, August 1996.

² For example, one restriction enzyme may cut the DNA at each occurrence of the substring GTTAAC (from left to right or right to left, it does not matter for the enzyme) into GTT and AAC.

³ The only exception is that we may identify the two ending fragments of each clone and take out these from each clone [13]. Hence only fragments that are cut at both ends by the restriction enzyme are considered.

A genome map is a sequence of clones c_1, \dots, c_n of some DNA string such that the left end of c_i precedes the left end of c_{i+1} on the DNA string. A refined genome map according to a specific enzyme is a genome map where the restriction fragments of each clone are also ordered from left to right according to their position on the DNA string. We can summarize the problem of genome map assembly with single enzyme complete digestion as follows:

INPUT: For an enzyme and for each of a set of N clones of the same DNA string the approximate fragment lengths after complete digestion of the clone by the enzyme.

OUTPUT: All possible map assemblies compatible with the input.

For example, the following may be an input to the genome map assembly problem:

Clone	Fragment Lengths
C1	5,10,15,25,30,35
C2	5,10,15,20,35,40
C3	5,20,25,30,45,50
C4	5,10,20,25,30,45
C5	5,10,15,20,25,35
C6	8,15,20,25,40
C7	15,20,25,30,45,50
C8	8,20,25,35,40
C9	15,40,45,50

The unit of measurements in the above table is hundred nucleotides. For any fragment its measurement can be off by a few nucleotides from its actual length. Small errors can be eliminated from consideration by rounding the measurements to the nearest tenths or hundreds of nucleotides.

There are some good heuristics for assembling genome maps, although the problem is known to be NP-complete in the number of restriction fragments in the clones [6, 10]. However, the simple heuristics often lead to incorrect maps or ambiguities. The refinement method introduced in this paper aims to correct inaccuracies and eliminate ambiguities in genome maps.

In this paper we devise a nondeterministic automaton that refines genome maps. Given any unrefined genome map (derived using some simple heuristic), the automaton decides whether it can be refined, that is, whether the restriction fragments can be ordered in each clone in a way that the orderings in the individual clones are compatible. If a refinement is not possible then the unrefined genome map assembly is highly likely to be invalid.

The automaton can be implemented in programming languages that provide (1) representation of sets, (2) representation and solving of set constraints, and (3) recursion. One such language is provided in the DISCO database system (short for *Datalog with Integer and Set order CO*nstraints), which is currently implemented at the University of Nebraska [1]. We expressed the automaton in DISCO and tested it on small sample problems.

The paper is organized as follows. Section 2 describes heuristics for the genome map assembly problem. Section 3 presents an automaton for refining genome

maps. Section 4 provides implementation in DISCO of the automaton and some sample results. Section 5 discusses related work. Section 6 presents some conclusions and open problems.

2 Basic Genome Map Assembly Heuristics

The basic strategy of the solution is the following. We try to find the order of the clones according to the position of their left ends on the DNA string. Then we walk through from left to right on the clones. That is we pick our best guess for the leftmost clone. Then we try to identify the next leftmost clone and so on until we reached the end of the DNA string. Next we consider several heuristics for choosing the next clone.

Threshold Heuristic: If $A = \{a_1, \dots, a_m\}$ is the current clone, choose as the next clone out of the yet unused ones the clone B_i such that the cardinality of $A \cap B_i$ is above a certain constant threshold.

For our example, the cardinality of the intersections considering the enzyme discussed are the following:

	C2	C3	C4	C5	C6	C7	C8	C9
C1	4	3	4	5	2	3	2	1
C2		2	3	5	3	2	3	2
C3			5	3	2	5	2	2
C4				4	2	4	2	1
C5					3	3	3	1
C6						3	4	2
C7							2	3
C8								1

Maximal Overlap Heuristic: This is a heuristic that we introduce and use in this paper. This extends the previous heuristic with the following. In case several clones could be chosen according to the enzyme threshold heuristic, then choose the one with the highest overlap.

Suppose we know that clone 8 is the leftmost clone on the DNA string. Then using the maximal overlap heuristic, we obtain for our example the search tree shown above. The search tree shows that there is some ambiguity in the genome map assembly. We have seven possible genome maps to consider.

3 A Refined Genome Map Assembly Automaton

In the refined genome map assembly problem we are not only interested in the order of the clones, but also the order and alignment of the restriction fragments in each clone. Sometimes, it is hard to know how to align two clones even when we know that they overlap. Suppose that the cardinality of the intersection of clones A and B is 4. Then we can be fairly sure that these two overlap if our

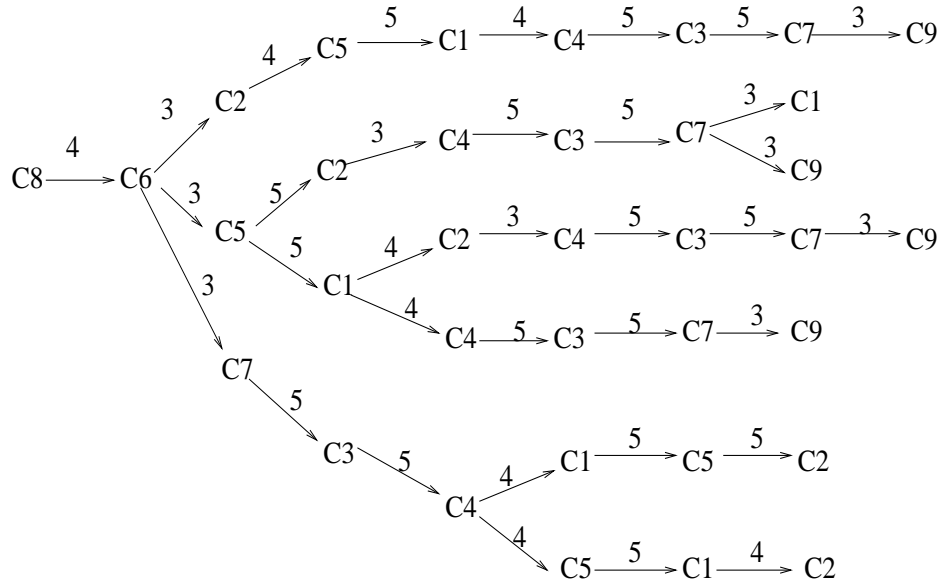


Fig. 1. The search tree using the maximal overlap heuristic

threshold value is 3.⁴ The restriction fragments measured as 495 and 502 may or may not be actually the same may be

But do they overlap on each of the four segments that they have in common, or only on three, two or one of them? The maximal alignment seems the most reasonable assumption under the following conditions, which we assume to hold in the rest of this paper:

- (1) No clone contains another clone.
- (2) No clone contains two different restriction fragments with the same length. However, different clones may contain different restriction fragments that have the same length.
- (3) Each restriction fragment is contained in at most k clones, where k is some fixed number.
- (4) The right end of each clone overlaps the left end of at least one other clone when they are mapped to their correct positions on the DNA string.

Note that all the unrefined genome maps that we want to check and refine further if we can will automatically satisfy (4). To solve the example in the previous sections, we test each of the seven genome maps in Figure 2 separately. Out of the seven only one, namely the genome map corresponding to the topmost

⁴ The threshold value is also influenced by the accuracy of the measurements. For example, some of the 5s in the table in the introduction may reflect measurements of 495, or 502. In general, the more accurate the measurements, the smaller threshold values we need [11].

branch of the tree, can be refined with maximal alignment as shown in Figure 3. Hence this genome map is the most likely to reflect the real order of clones on the DNA string.

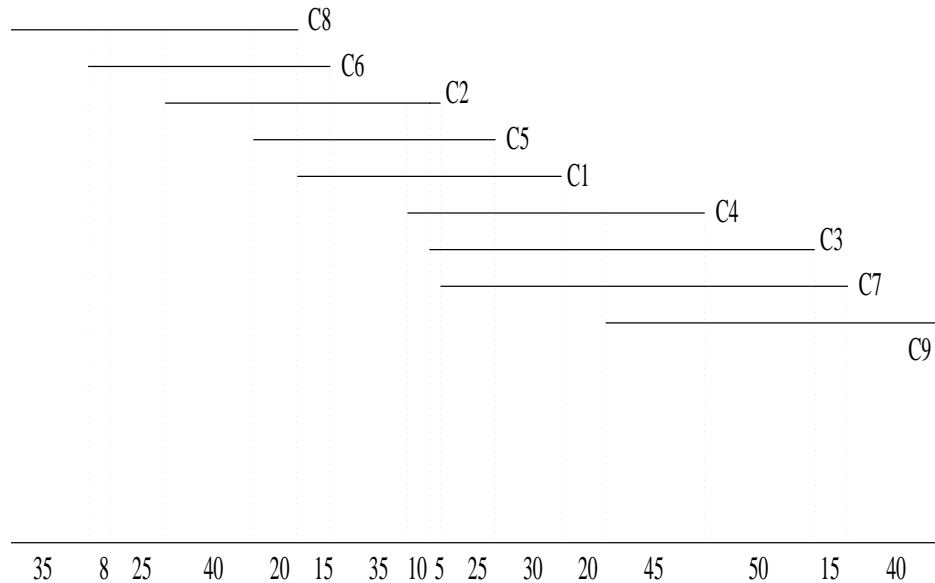


Fig. 2. A refined genome map

We present a refinement automaton (see Figure 3) that tests whether a candidate unrefined genome map is valid or not under the assumptions listed above. The basic idea is to build a staircase like the one shown in Figure 3.⁵

We call any subset of the staircase with $k + 1$ clones a *window*. As we refine the genome map this window will move to the right and down the staircase. Let at all times the elements of the window be called A_1, \dots, A_{k+1} in order.

We say that a clone is active if it must contain the element that we pick next. At any time one to five clones are active. The main idea in the automaton is to have a separate state i for the case when i clones are active. We also keep two counters, d and u to signal that we must move to a lower or higher state. When $d = 0$ we will move to a lower state. Moving to a lower state is needed when we

⁵ Any set of clones satisfying conditions (1) and (4) can be arranged into a staircase. Just sort the clones according to the position of their left endpoints on the DNA string. The sorted order forms a staircase because condition (1) implies that if the left end of clone A precedes the left end of clone B, then the right end of clone A also precedes the right end of clone B and condition (4) ensures that the adjacent clones overlap.

read the last item from the leftmost clone A_1 . Moving to a lower state occurs again after picking cardinality of $A_2 \setminus A_1$ number of items, because then we will read the last item of A_2 . Hence when we move to a lower state we set the counter d to $card(A_2 \setminus A_1)$. When moving to a lower state we also advance the window one step downward. This means that we assign to A_i the previous value of A_{i+1} for each $1 \leq i \leq k$ and assign to a_{k+1} the next clone of the genome map that we wanted to refine. If no next clone exists, that is, when we are reading the last few clones in the genome map, then the value of A_{k+1} will be the empty set.

Similarly, when $u = 0$ we move to a higher state. Moving from state i to state $i + 1$ is needed when we already picked all items in $A_i \setminus A_{i+1}$. Moving to a higher state occurs again after picking all items in $A_{i+1} \setminus A_{i+2}$, because then we read all items in A_{i+1} which don't occur in A_{i+2} . Hence when we move to a higher state we set the counter u to $card(A_{i+1} \setminus A_{i+2})$.

Finally, there are transitions from each state i to itself. On these transitions we pick a value from the leftmost i clones in the window. On each self-transition, we also decrease by one the counters d and u .

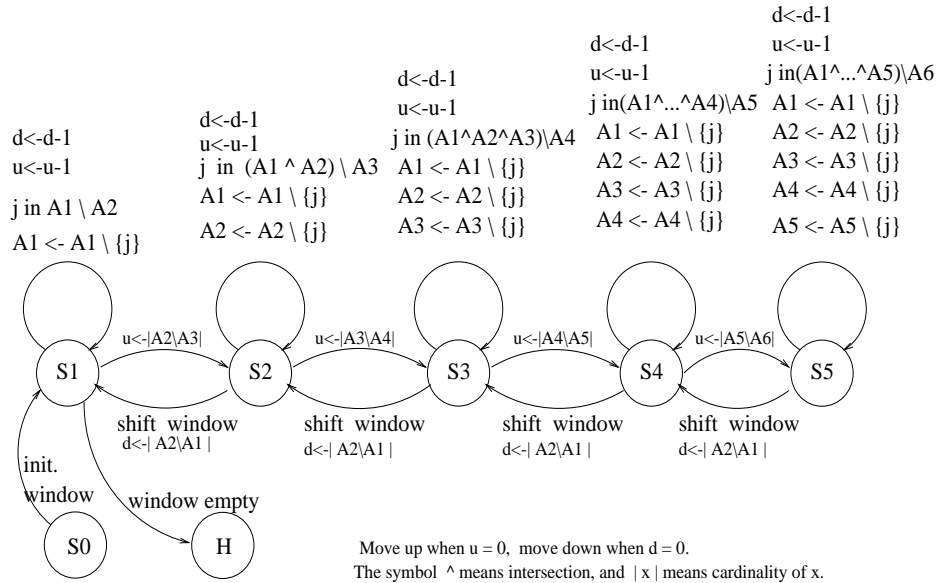


Fig. 3. The automaton for $k = 5$.

Let d_1 be the number of fragments in c_1 , and let d_i be the cardinality of $c_i \setminus c_{i-1}$ for $1 < i \leq n$. Let u_i be the cardinality of $c_i \setminus c_{i+1}$ for each $1 \leq i < n$, and let u_n be the cardinality of c_n . Let $\alpha(c_i)$ be the number of elements picked from clone c_i .

Lemma 3.1 Suppose that the automaton reaches the halt state given a genome map c_1, \dots, c_n . Then the following must be true for each $1 < i < n$:

$$\alpha(c_{i+1}) = \alpha(c_i) + d_{i+1} - u_i$$

□

Next we prove that to reach the halt state the automaton must pick all the items from each clone. At first, note the following identity (*) for each $1 < i < n$:

$$\text{card}(c_{i+1}) = \text{card}(c_i) + \text{card}(c_{i+1} \setminus c_i) - \text{card}(c_i \setminus c_{i+1}) = \text{card}(c_i) + d_{i+1} - u_i$$

Lemma 3.2 Suppose that the automaton reaches the halt state given a genome map c_1, \dots, c_n . Then the following must be true for each $1 \leq i \leq n$:

$$\alpha(c_i) = \text{card}(c_i)$$

□

For each adjacent pair of clones c_i, c_{i+1} we call the requirement that the items in $c_i \setminus c_{i+1}$ precede the items in $c_i \cap c_{i+1}$ which are aligned and precede the items in $c_{i+1} \setminus c_i$ the *ith step constraint*.

Theorem 3.1 The automaton reaches the halt state if and only if the input genome map can be refined such that all step constraints are satisfied. □

4 Solution in DISCO

DISCO is a constraint database system. Constraint databases generalize the relational data model by describing extensional database predicates using constraint tuples, with each constraint tuple being a quantifier-free conjunctive formula of atomic constraints in some constraint theory. Constraint query languages [9] map input constraint databases to output constraint databases.

The advantages of constraint database systems include the following (1) flexibility of implicit representation of data, (2) compact representation of large and possibly infinite relations, (3) more efficient pruning of the search space than in non-constraint based systems, and (4) higher expressive power. Because of (4) it is possible to express and solve many problems in constraint database systems that traditional database systems cannot handle.

4.1 The DISCO Query Language

The syntax of the query language of DISCO, denoted $\text{Datalog}^{< \mathbf{z}, \subseteq \mathcal{P}(\mathbf{z})}$, is that of traditional Datalog (Horn clauses without function symbols) where the bodies of rules can also contain a conjunction of integer or set order constraints. That is, each program is a finite set of rules of the form: $R_0 \text{ :- } R_1, R_2, \dots, R_l$. The expression R_0 (the rule *head*) must be an atomic formula of the form $p(v_1, \dots, v_n)$, and the expressions R_1, \dots, R_l (the rule *body*) must be atomic formulas of one of the following forms:

1. $p(v_1, \dots, v_n)$ where p is some predicate symbol.
2. $v\theta u$ where v and u are integer variables or constants and θ is a relational operator $=, \neq, <, \leq, >, \geq, <_g$ where g is any natural number. For each g the atomic constraint $v <_g u$ is used as shorthand for the expression $v + g < u$.
3. $V \subseteq U$ or $V = U$ where V and U are set variables or constants.
4. $c \in U$ or $c \notin U$ where c is an integer constant and U is a set variable or constant.

Atomic formulas of the form (2) above are called gap-order constraints and of the form (3-4) are called set order constraints. In this paper we will always use small case letters for integer variables and capital letters for set variables. Set variables always stand for a finite or infinite set of integers.

Let \mathcal{M} be the set of all possible ground tuples over the integers and sets of integers. Let P be a $Datalog^{<z, \subseteq_{P(z)}}$ program and d be a constraint database. Let \mathcal{D} be the set of ground tuples implied by d . The function T_P from and into \mathcal{M} is defined as follows.

$T_P(\mathcal{D}) = \{t \in \mathcal{M} : \text{there is a rule } R_0 :- R_1, \dots, R_k \text{ in } P \text{ and an instantiation } \theta \text{ such that } R_0\theta = t, \text{ and } R_i\theta \text{ holds if } R_i \text{ is a constraint and } R_i\theta \in \mathcal{D} \text{ otherwise for each } 1 \leq i \leq k. \}$

It is shown in [1] that $Datalog^{<z, \subseteq_{P(z)}}$ queries have a fixpoint model which coincides with the least model. It was also shown in [16] that the data complexity of DISCO queries is DEXPTIME-complete. Therefore DISCO can express a wide variety of combinatorial and logical problems. For example, DISCO can express the basic NP-complete problem of testing the satisfiability of a propositional formula in conjunctive normal form by a fixed query and a variable size database [1].

4.2 Expressing the Refinement Automaton in DISCO

We start by expressing the clone data by the input database relation:

clone(1, {5, 10, 15, 25, 30, 35}).
clone(2, {5, 10, 15, 20, 35, 40}).
clone(3, {5, 20, 25, 30, 45, 50}).
clone(4, {5, 10, 20, 25, 30, 45}).
clone(5, {5, 10, 15, 20, 25, 35}).
clone(6, {8, 15, 20, 25, 40}).
clone(7, {15, 20, 25, 30, 45, 50}).
clone(8, {8, 20, 25, 35, 40}).
clone(9, {15, 40, 45, 50}).

We define the *card_diff*(a, b, k) relation in the input database to be true if the cardinality of the difference of the elements of clones a and b is k . We define *card*(a, k) to be true if the cardinality of clone a is k . We also define the relation *next* that allows us to count from 0 to the largest k in the previous two relations.

$next(0, 1).$

\vdots

$next(6, 7).$

The candidate genome map assembly that we would like to refine is represented by the input relation $next_clone(a, b)$ which is true if after clone a the next clone is clone b , and the relation $first(a)$ which is true if a is the first clone. We also add a dummy clone $C10$ to the end of the list with empty set of fragments and declare $next_clone(10, 10)$ to be a true fact in the database.

Let L be the set of length values that occur in any of the clones. In the example of Figure 3 the set L is $\{5, 8, 10, 15, 20, 25.30, 35, 40, 45, 50\}$. Only the values in L need to be tested for membership in sets. Hence for each constant $c \in L$ we add the following constraint tuples to the input database:

$in(c, X) \quad :- \quad c \in X.$

$not_in(c, X) \quad :- \quad c \notin X.$

We define the relation $cut(i, X, Y)$ which means that $Y \subseteq X \setminus \{i\}$ as follows.

$cut(i, X, Y) \quad :- \quad Y \subseteq X, not_in(i, Y).$

To express the automaton, we keep the configuration of the automaton in the relation $si(j, a_1, A_1, \dots, a_k, A_k, d, u)$. This configuration indicates that we are in state i , the current window is the remaining parts A_1, \dots, A_k of the clones a_1, \dots, a_k , and if $j \neq -1$, then j was the last item picked, otherwise we just made a transition from another state to state i . The transitions from state i to itself for each $1 \leq i \leq 5$ can be expressed as follows.

$$s_i(j, a_1, B_1, \dots, a_k, A_k, d_1, u_1) \quad :- \quad s_i(j_1, a_1, A_1, \dots, a_k, A_k, d, u), \\ in(j, A_1), \dots, in(j, A_i), not_in(j, A_{i+1}), \\ cut(j, A_1, B_1), \dots, cut(j, A_i, B_i), \\ next(d_1, d), next(u_1, u).$$

The transitions from lower states to higher states can be expressed as:

$$s_{i+1}(-1, a_1, A_1, \dots, a_k, A_k, d, u) \quad :- \quad s_i(j, a_1, A_1, \dots, a_k, A_k, d, 0), \\ card_diff(a_{i+1}, a_{i+2}, u).$$

The transitions from higher states to lower states can be expressed as:

$$s_{i-1}(-1, a_2, A_2, \dots, a_{k+1}, A_{k+1}, d, u) \quad :- \quad s_i(j, a_1, A_1, \dots, a_k, A_k, 0, u), \\ card_diff(a_2, a_1, d), \\ next_clone(a_k, a_{k+1}), \\ clone(a_{k+1}, A_{k+1}).$$

Finally, the initial transition from state 0 to state 1 and the final transition from state 1 to state h can be expressed as:

$$h(a_1, \dots, a_k) \quad :- \quad s1(j, a_1, \{\}, \dots, a_k, \{\}, 0, u).$$

$$s0(-1, a_1, A_1, \dots, a_k, A_k, d, u) \quad :- \quad first(a_1), \\ next_clone(a_1, a_2), \dots, next_clone(a_{k-1}, a_k), \\ clone(a_1, A_1), \dots, clone(a_k, A_k), \\ card(a_1, d), card_diff(a_1, a_2, u).$$

We tested the algorithm on the sample data shown in Figure 3 and described in Section 2. In the actual implementation, we added an extra sixteenth temporal argument t which was initialized to 0 and was increased by 1 for each transition

from state i to itself. This helped us to keep track of the order of the tuples and to check the correctness of the program. (In the future, we will add a trace routine by which DISCO will be able to print out the proof trees.)

In the output computed we can ignore the tuples where instead of a restriction fragment length a -1 occurs which indicate only the initialization and transitions between states. The other tuples can be ordered according to the value of t . Ignoring all the values except for t and the fragment length j we had:

```
s1(1, 35, ...)  
s2(2, 8, ...)  
s2(2, 25, ...)  
s2(3, 8, ...)  
s2(3, 25, ...)  
s3(4, 40, ...)  
s4(5, 20, ...)  
:  
s1(16, 40, ...)
```

From this it is clear that the sequence: 35, 8, 25, 40, 20, 15, 35, 10, 5, 25, 30, 20, 45, 50, 15, 40. or the same sequence with the second and third elements interchanged are the solutions. On this problem DISCO was run on a SUN sparc 2 workstation in 579 CPU seconds.

5 Related Work

There are many variations of the genome map assembly problem beside the one we considered in the paper. One other version called the probed partial digestion problem is considered in [19]. Here the task is to align a number of DNA fragments each of which overlaps the same small piece of the DNA string. This is different from our problem, where some pairs of clones (for example in Figure 3 clones C8 and C9) do not overlap. The probed partial digestion problem is solved using the deductive database system LDL [18].

In a second version of the genome mapping problem, the clones do not all overlap and the information of overlap derives from probe data and not from restriction fragment length similarities as in our problem. This problem was considered for example in [7]. Here a rough map is constructed first and refined by considering certain graph properties that must hold to make a map linear.

A third version of the mapping problem considered in [2, 20] relies on multiple restriction enzyme data. In this version, we are given for three copies A, B, and C of the same DNA string the following information: For A the set of fragment lengths that are obtained after enzyme one is applied, for B the lengths after enzyme two is applied, and for C the lengths when the two enzymes are applied together. The problem is to order the restriction fragments in the three copies in parallel such that every time there is a cut in copy A or in copy B, there is also a cut in copy C. This problem is solved in [20] using the constraint logic programming system CLP(R) [8].

The DISCO constraint database system is related to several other systems that allow some type of set constraints [3, 4, 5, 12, 14, 17]. The automaton described in Section 3 can be implemented in these systems as well. However, the running times and scaling for larger size problems could be very different in these systems because they use different methods for solving set constraints.

6 Conclusions and Open Problems

As is apparent, there are several variations on the genome map assembly problem. These variations suggest that modifications may be necessary in tailoring any algorithm to particular applications. Hence it seems that declarative, high-level and easily maintainable solutions will have a special advantage. We also plan to modify our solution by adding probe data information and use them together with restriction fragment lengths information.

References

1. J. Byon, P. Z. Revesz. DISCO: A Constraint Database System with Sets. *Proc. Workshop on Constraint Databases and Applications*, Springer-Verlag LNCS 1034, pp. 68–83, September 1995.
2. T.I. Dix, C.N. Yee. A restriction mapping engine using constraint logic programming, *Proceedings of 2nd International Conference on Intelligent Systems for Molecular Biology*, AAAI Press, pp. 112-120, 1994.
3. A. Dovier, G. Rossi. Embedding extensional finite sets in CLP. *International Logic Programming Symposium*, 1993.
4. ECLIPSE. *Eclipse user manual*. Technical report. ECRC, 1994.
5. C. Gervet. Conjunto: Constraint Logic Programming with Finite Set Domains. *Proc. International Logic Programming Symposium*, 339–358, 1994.
6. W. Gillett. DNA Mapping Algorithms: Strategies for Single Restriction Enzyme and Multiple Restriction Enzyme Mapping. Washington Univ. Tech. Report WUCS-92-29.
7. E. Harley, A. Bonner, N. Goodman. Good maps are straight. *Proc. 4th Int. Conf. on Intelligent Systems for Molecular Biology*, p. 88-97, June 1996.
8. J. Jaffar, S. Michaylov, P.J. Stuckey, R.H. Yap. The CLP(*R*) Language and System. *ACM Transactions on Programming Languages and Systems*, 14:3, 339-395, 1992.
9. P. C. Kanellakis, G. M. Kuper, P. Z. Revesz. Constraint Query Languages. *Journal of Computer and System Sciences*, vol. 51, 26–52, 1995.
10. R. M. Karp. Mapping the genome: some combinatorial problems arising in molecular biology. *Proc. 25th ACM Symp. on Theory of Computing*, 278-285, 1993.
11. E. S. Lander, M. S. Waterman. Genomic mapping by fingerprinting random clones: A mathematical analysis. *Genomics*, Vol. 2, 231-239, 1988.
12. B. Legnard, E. Legros. Short overview of the CLPS System. *Proc. PLILP*, 1991.
13. M. V. Olson et al. Random-clone strategy for genomic restriction mapping in yeast. *Genomics*, vol. 83, 7826–7830, 1986.

14. R. Ramakrishnan, D. Srivastava, S. Sudarshan. CORAL: Control, Relations and Logic. *Proc. VLDB*, 1992.
15. P. Z. Revesz. A Closed Form Evaluation for Datalog Queries with Integer (Gap)-Order Constraints, *Theoretical Computer Science*, vol. 116, no. 1, 117-149, 1993.
16. P. Z. Revesz. Datalog Queries of Set Constraint Databases, *Fifth International Conference on Database Theory*, Springer-Verlag LNCS 893, pp. 425-438, Prague, Czech Republic, January, 1995.
17. D. Srivastava, R. Ramakrishnan, P.Z. Revesz. Constraint Objects. *Proc. 2nd Workshop on Principles and Practice of Constraint Programming*, 274-284, 1994.
18. S. Tsur and C. Zaniolo. LDL: A Logic-Based Data-Language. *Proc. VLDB*, pp 33-41, 1986.
19. S. Tsur, F. Olken, D. Naor. Deductive databases for genome mapping. Technical Report LBL-29577, Lawrence Berkeley Laboratory, Berkeley, CA 94720.
20. R.H.C. Yap. A Constraint Logic Programming Framework for Constructing DNA Restriction Maps, *Artificial Intelligence in Medicine*, Vol. 5, 447-464, 1993,