Constraint Database Solutions to the Genome Map Assembly Problem

Viswanathan Ramanathan and Peter Revesz

Department of Computer Science and Engineering
University of Nebraska-Lincoln, Lincoln, NE 68588, USA
{ramanath, revesz}@cse.unl.edu

Abstract. Long DNA sequences have to be cut using restriction enzymes into small fragments whose lengths and/or nucleotide sequences can be analyzed by currently available technology. Cutting multiple copies of the same long DNA sequence using different restriction enzymes yields many fragments with overlaps that allow the fragments to be assembled into the order as they appear on the original DNA sequence. This basic idea allows several NP-complete abstractions of the genome map assembly problem. However, it is not obvious which variation is computationally the best in practice. By extensive computer experiments, we show that in the average case the running time of a constraint automata solution of the *big-bag matching* abstraction increases linearly, while the running time of a greedy search solution of the *shortest common superstring in an overlap multigraph* abstraction increases exponentially with the size of real genome input data. Hence the first abstraction is much more efficient computationally for very large real genomes.

1 Introduction

Constraint databases can be useful in the area of genomics. In particular, we can solve the genome map assembly problem, described in detail in Section 2, very efficiently using constraint databases and the closely related concept of constraint automata.

Not only is a solution possible in the constraint database framework, but it turns out to be a much more efficient solution than earlier proposals based on greedy search for a common shortest superstring in an overlap multigraph, which is also described in Section 2.

In this paper we propose a modification of the big-bag matching abstraction, which is then solved using a constraint automaton. The earlier constraint automaton solution was a pure fingerprints-based method. Our modification is to disambiguate the fingerprint data during a search by comparing the nucleotide sequences of the ambiguous data. Hence ours is a hybrid method between pure fingerprinting and pure string comparison methods.

By extensive computer experiments we show that the running time of the modified constraint automaton solution grows essentially linearly while the running time of the other solution grows approximately exponentially with the size

```
CATCGATCTCGGGAGGGATCCATTATCGATTCCCGGGCTCGGGGGATCCT
TCCATCGATGGGCCCGAGGCGGATCCCTACTATCGATCCCGGGGGGATCC
TTAATTCTCGAGAAGGCCTATCGATCAAGGATCCTATCGATCCCGAGTCC
CGGGAT
```

Fig. 1. A genome sequence part of Lambda bacterium.

of the input genome data. This is surprising, because both abstractions are known to be NP-complete, hence the two algorithms were hard to distinguish theoretically.

This paper is organized as follows. Section 2 describes some basic concepts and related work. Section 3 explains the proposed modified constraint automaton solution and its implementation. Section 4 presents the results and the statistical analysis of the experiments conducted. Section 5 discusses the use of constraint databases in this project. Finally, Section 6 gives some conclusions and directions for future work.

2 Basic Concepts and Related Work

Unfortunately, there have been only few papers exploring the connection between genomics and constraint databases, because of the need to understand a significant number of concepts in both areas. Hence we will start with a brief review of the key definitions and related work.

Genome. The genome of an organism is its set of *chromosomes*, containing all of its *genes* and the associated *deoxyribonucleic acid* (DNA) [4]. DNA is a double-stranded helix consisting of nucleotides. Each nucleotide has three parts: a sugar molecule (S), a phosphate group (P), and a structure called a nitrogenous base (A, C, G, and T). The DNA is built on the repeating sugar-phosphate units. The sugars are molecules of deoxyribose from which DNA receives its name. Joined to each deoxyribose is one of the four possible nitrogenous bases: *Adenine* (A), *Cytosine* (C), *Guanine* (G), and *Thymine* (T). These bases carry the genetic information, so the words *nucleotide* and *base* are often used interchangeably. Since the DNA is double-stranded, the bases on one strand are linked to the bases on the other strand to form a base pair. Adenine always pairs with Thymine and Guanine always pairs with Cytosine. The length of a DNA is measured in terms of number of base pairs.

For example, Fig. 1 shows a part of one strand of the Lambda bacterium DNA [5]. Its length is 156 base pairs.

Restriction Enzymes. *Restriction enzymes* are precise molecular scalpels that allow a scientist to manipulate DNA segments. They recognize specific base sequences in double-helical DNA and cut, at specific places, both strands of a duplex containing the recognized sequences [1]. They are indispensable tools for analyzing chromosome structure, sequencing very long DNA molecules, isolating genes, and creating new DNA molecules that can be cloned.

Many restriction enzymes recognize specific sequences of four to eight base pairs. Their names consist of a three-letter abbreviation (e.g., *Eco* for *Escherichia coli*) followed by a strain designation (if needed) and a roman numeral (if more than one restriction enzyme from the same strain has been identified) (eg. `EcoRI`).

For example, we can apply the restriction enzyme `ClaI`, which always cuts the genome at each site where `AT^CGAT` appears with the wedge symbol showing where the cut will take place. Then we obtain the subsequence shown in Fig. 2.

A1: CAT
A2: CGATCTCGGGAGGGATCCATTAT
A3: CGATTCCCGGGCTCGGGGGATCCTTCCAT
A4: CGATGGGCCCGAGGCGGATCCCTACTAT
A5: CGATCCCGGGGGGATCCTTAATTCTCGAGAAGGCCTAT
A6: CGATCAAGGATCCTAT
A7: CGATCCCGAGTCCCGGGAT

Fig. 2. Subsequences of the Lambda bacterium sequence.

If we use another restriction enzyme `BamHI`, which cuts at sites `G^GATCC`, then we obtain the subsequences shown in Fig. 3.

B1: CATCGATCTCGGGAGG
B2: GATCCATTATCGATTCCCGGGCTCGGGG
B3: GATCCTTCCATCGATGGGCCCGAGGCG
B4: GATCCCTACTATCGATCCCGGGGG
B5: GATCCTTAATTCTCGAGAAGGCCTATCGATCAAG
B6: GATCCTATCGATCCCGAGTCCCGGGAT

Fig. 3. Another set of subsequences of the same Lambda bacterium sequence.

Genome Sequencing and Mapping. *Genome sequencing* is the process of finding the order of DNA nucleotides, or bases, in a genome [4]. On the other hand, *genome mapping* is the process of finding the approximate position of landmarks (specific subsequences, often genes) in a genome without getting into the details of the actual sequence. A genome map is less detailed than a genome sequence [4]. A sequence spells out the order of every nucleotide in the genome, while a map simply identifies the order of the specified subsequences in the genome. Nevertheless, the two are closely related concepts, because by sequencing each landmark of a genome map, we can get a genome sequence.

For example, suppose we consider each of the seven subsequences in Fig. 2 as a landmark. Then a genome map would be the following:

$$A1\ A2\ A3\ A4\ A5\ A6\ A7$$

Sometimes we can have only a partial genome map. Suppose that only $A2$ and $A5$ are known landmarks. Then a partial genome map would be the following:

$$-\ A2\ -\ -\ A5\ -\ -$$

where the dashes are unknown regions on the genome map.

Genome Map Assembly Problem. DNA sequences are huge, having a length of around 200-300 million base pairs for many animals and plants. But current sequencing machines can not handle DNA sequences of length more than a couple of thousand base pairs. So the DNA sequences have to be cut into small subsequences using restriction enzymes. After the application of a restriction enzyme the subsequences obtained are floating a solution. Hence we no longer know their original order. So, once the subsequences are sequenced and analyzed, they have to be arranged and assembled to obtain the original sequence. This process is called *Genome Map Assembly* [8]. The problem of executing the Genome Map Assembly process is called the *Genome Map Assembly Problem* (GMAP) [20].

Overlap Multigraph The *overlap multigraph* method [25] is one way of doing the genome map assembly. This method is usually illustrated by a graph in which each node is one of the subsequences and each directed edge from node A to node B has a label $k \geq 0$ if and only if the last k nucleotides of A and the first k nucleotides of B are the same.

For example, let F be the union of the As and Bs in Figs. 2 and 3. Then part of the overlap multigraph of F is shown in Fig. 4. We only show the edges that have positive labels and are incident on both an A and a B node. In general, there are many edges possible between two nodes; hence it is called a multigraph. For example, we have two directed edges, one labeled by 2 the other by 16, from $B3$ to $A4$.

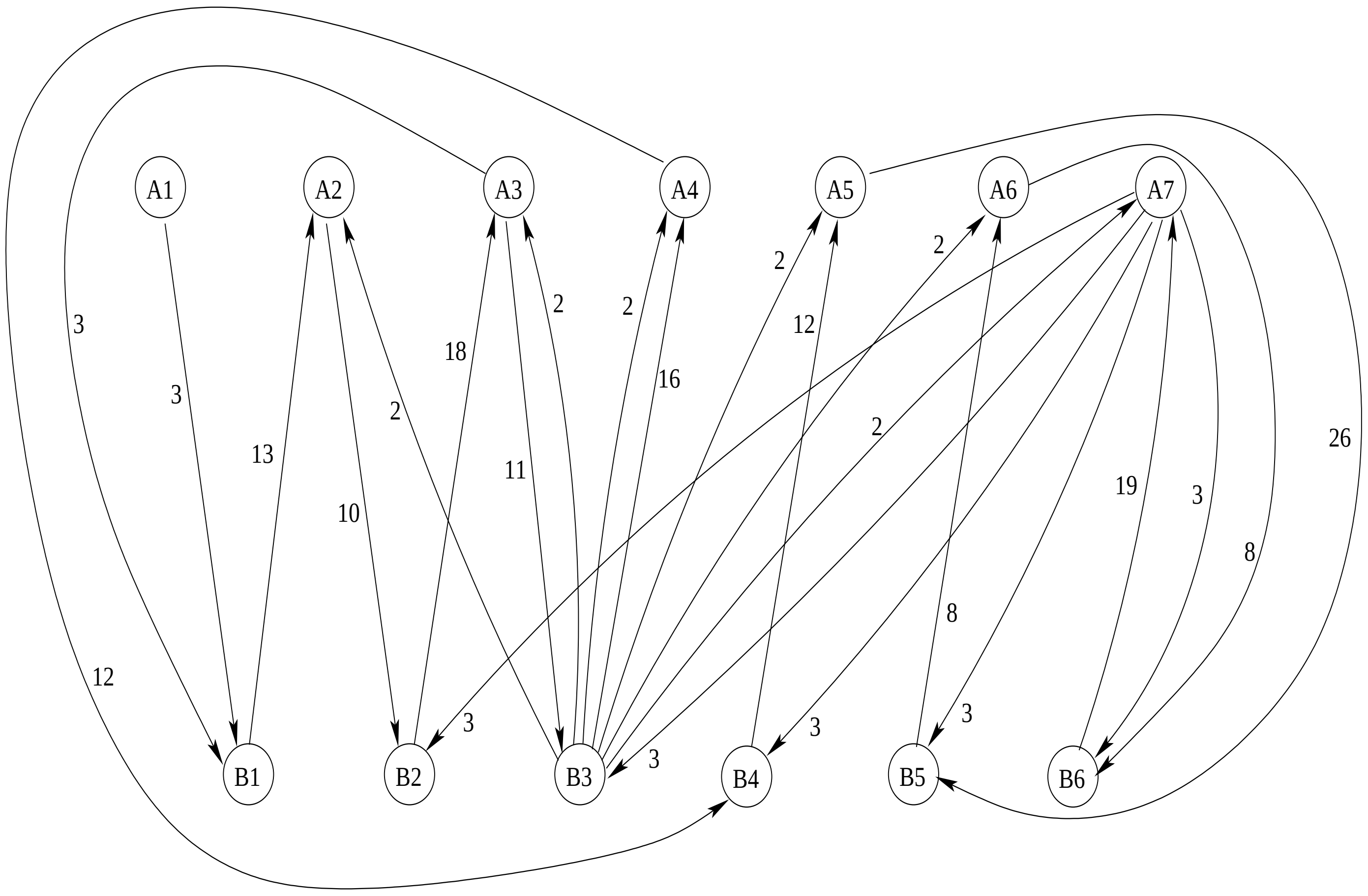

Fig. 4. The overlap multigraph for F.

It is well-known [25] that a Hamiltonian path in the overlap multigraph is a *shortest common superstring* of the sequences in F. Therefore, it is a solution to the genome sequencing problem.

For example, a Hamiltonian path is shown in Fig. 5. This Hamiltonian path matches the original genome sequence in Fig. 1, and incidentally also gives a genome map in terms of As on the top and in terms of Bs on the bottom.

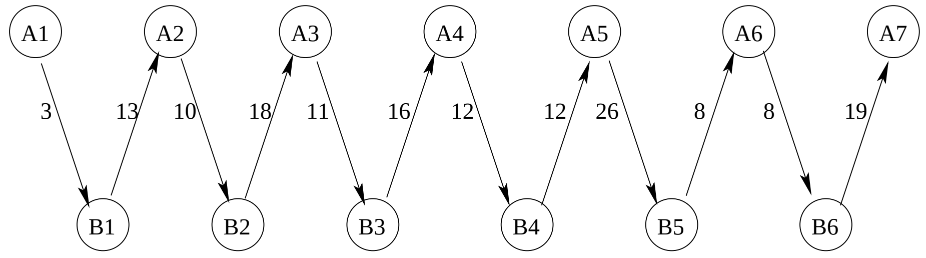

Fig. 5. A Hamiltonian path in the overlap multigraph.

Unfortunately, such a Hamiltonian path is hard to find. A typical greedy search of the graph would always choose to explore the edges with the highest label. The greedy search would start at node $A5$ because it has the highest label, namely 26, on one of its edges. However, $A5$ is a failure node. Then the greedy search would explore $B6$ because it has the edge with the next highest label, namely 19. Continuing this way, the last node that the greedy search will explore is $A1$ or $A7$. Out of these only $A1$ is a success node. Figure 6 shows that $A7$ is a failure node. It is clear that the greedy search on the overlap multigraph is doing a huge amount of work.

Fingerprints. It is computationally hard to compare and match long nucleotide sequences. *Fingerprinting* is a short-cut method that is often much faster. The ideas is to get a "fingerprint" of each subsequence (like those in Figs. 2 and 3) by applying one or more restriction enzymes and then measuring the lengths of the resulting fragments [29]. The multiset or *bag* of fragment lengths are usually unique to each subsequence.

For example, by applying the restriction enzymes `BamHI (G^GATCC)` and `AvaI (C^YCGRG)`, where `Y` means either `C` or T and `R` means either A or `G`, on the subsequences in Fig. 2, we get the fragments and the bag of their lengths shown in Fig. 7.

Revesz' fingerprint data. Revesz [20] proposed the following procedure using three restriction enzymes a, b, and c to obtain fingerprint data:

1. Take a copy of the DNA.

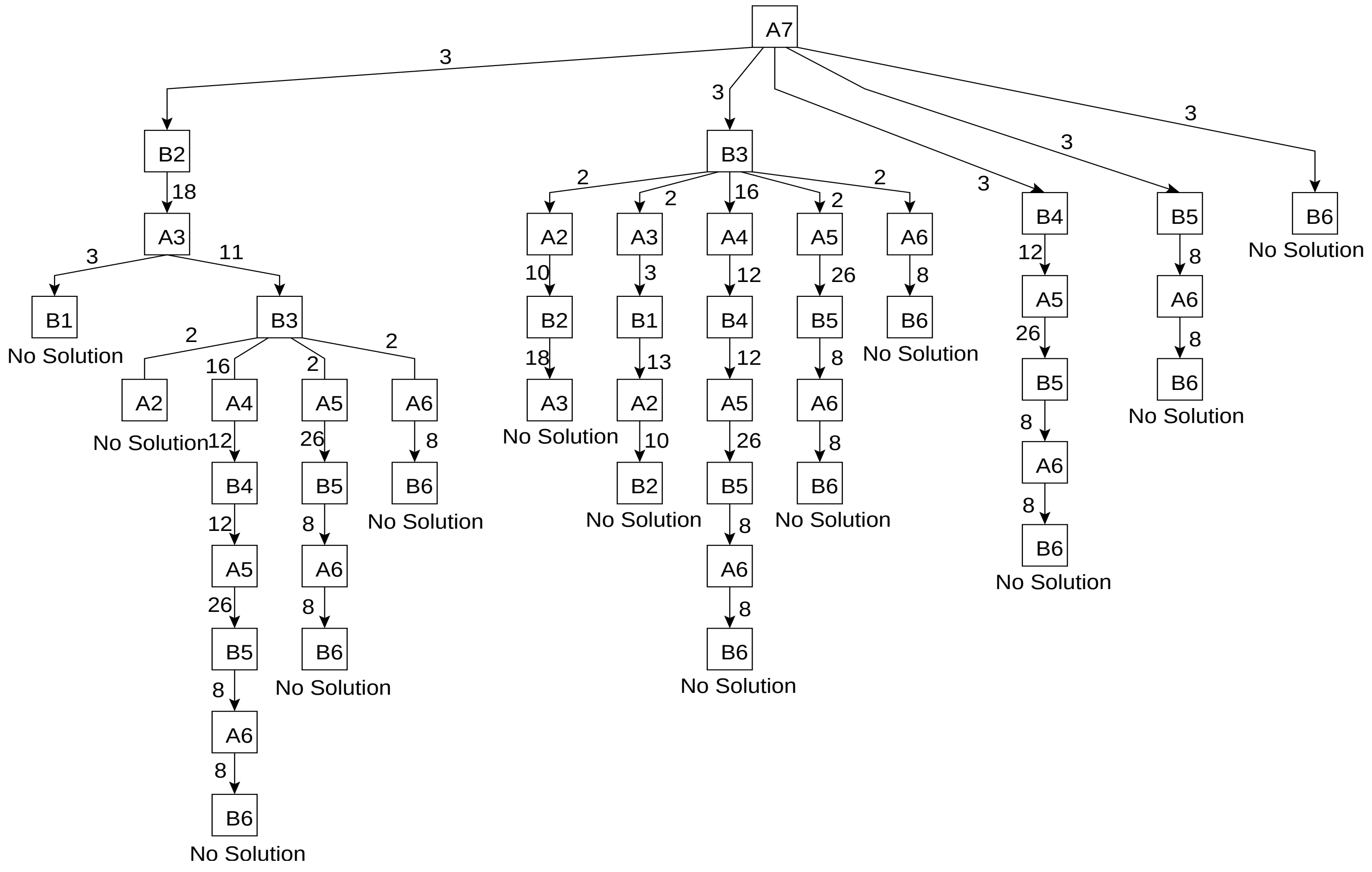

Fig. 6. Greedy search from node $A7$ of the overlap multigraph.

A1: CAT	{3}
A2: CGATC TCGGGAGG GATCCATTAT	{5,8,10}
A3: CGATTC CCGGGC TCGGGG GATCCTTCCAT	{6,6,6,11}
A4: CGATGGGC CCGAGGCG GATCCCTACTAT	{8,8,12}
A5: CGATC CCGGGGG GATCCTTAATTC TCGAGAAGGCCTAT	{5,7,12,14}
A6: CGATCAAG GATCCTAT	{8,8}
A7: CGATC CCGAGTC CCGGGAT	{5,7,7}

Fig. 7. Fingerprints of the *A* subsequences.

2. Apply restriction enzyme *a* to the copy.
3. Separate the subsequences.
4. For each subsequence apply restriction enzyme $b \cup c$, cutting the subsequence into fragments.
5. Find the lengths of the fragments.
6. Repeat Steps (1–5) using *b* instead of *a* and $a \cup c$ instead of $b \cup c$.

Each execution of Steps (1–5) is the generation of a *big-bag.* The subsequences obtained in Step (3) are the bags of the big-bag, and the lenghts of of the fragments obtained in Steps (4-5) are the elements of the bags. Once the DNA sequence is cut into subsequences using the above procedure, all the information about the original order is lost. After analyzing these subsequences, they have to be arranged or assembled into a single set of sequences called a *genome map.*

For example, let *a, b* and *c* be the restriction enzymes `ClaI`, `BamHI` and `AvaI`, respectively. Then after applying Step (1) we get the subsequences shown in Fig. 2. After Steps (4-5) we get the bags shown in Fig. 7.

In Step (6) when we repeat Step (2) using *b* instead of *a* we get the subsequences shown in Fig. 3. When we also repeat Steps (4-5) by applying the restriction enzymes `ClaI` ∪ `AvaI` on the latter set of subsequences, we get the fragments and the bag of their lengths shown in Fig. 8.

B1: CAT CGATC TCGGGAGG	{3,5,8}
B2: GATCCATTAT CGATTC CCGGGC TCGGGG	{10,6,6,6}
B3: GATCCTTCCAT CGATGGGC CCGAGGCG	{11,8,8}
B4: GATCCCTACTAT CGATC CCGGGGG	{12,5,7}
B5: GATCCTTAATTC TCGAGAAGGCCTAT CGATCAAG	{12,14,8}
B6: GATCCTAT CGATC CCGAGTC CCGGGAT	{8,5,7,7}

Fig. 8. Fingerprints of the *B* subsequences.

Note: In this method some care needs to be taken in the choice of the triplet of restriction enzymes so that they can be applied in any order, that is, they are *compatible*. We explain compatibility more precisely and give some examples in Section 4.1.

Why is this fingerprint data useful? Note that regardless of whether we start with the set of A or the set of B subsequences, after the triplet enzyme cuts are finished we get exactly the same set of fragments. These fragments are just grouped differently by the As and the Bs. Those As and Bs that that have a significant fingerprint overlap will tend to have also a nucleotide sequence overlap. Hence we can align or match the fragments in the As with the fragments in the Bs to find a solution.

For example, Fig. 9 shows an alignment that corresponds to the Hamiltonian path in Fig 5. In Fig. 9 each undirected edge is labeled with a set of values such that there are fragments with those set of lengths in the fingerprints of both incident nodes. The alignment requires that for each node the multiset union of the labels on the edges incident on it is the same as its fingerprint. For example, the multiset union of the labels $\{6, 6, 6\}$ and $\{11\}$ on the edges incident on $A3$ gives the fingerprint $\{6, 6, 6, 11\}$ of $A3$.

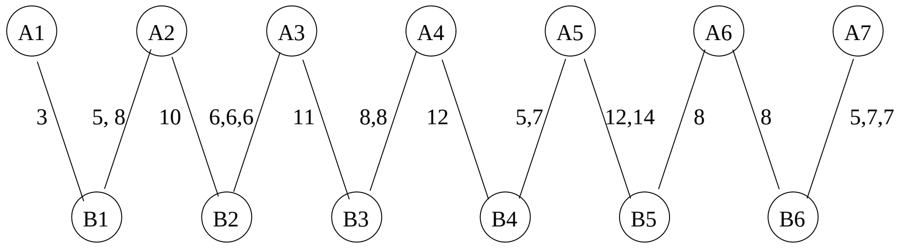

Fig. 9. A fingerprint overlap solution corresponding to the Hamiltonian path in Fig. 5.

We will see in Section 3 a way to find such fingerprint alignments using a constraint automaton.

Constraint Automata. *Constraint automata* are used to control the operation of systems based on conditions that are described using constraints on variables (see Chapter 6 of [21]). A constraint automaton consists of a set of states, a set of state variables, transitions between states, an initial state, and the domain and initial values of the state variables. Each transition consists of a set of constraints, called the guard constraints, followed by a set of assignment statements. The guard constraints are followed by question marks "?", and the assignment statements are shown using the symbol ":=".

A constraint automaton can move from one state to another if there is a transition whose guard constraints are satisfied by the current values of the state variables. The transitions of a constraint automaton may contain variables in addition to state variables. These variables are said to be existentially quantified variables, which means that some values for these variables must be found such that the guard constraints are satisfied and the transition can be applied.

GMAP using Constraint Automata. The Genome Map Assembly Problem can be solved using a constraint automaton [20]. The constraint automaton uses an abstraction of the GMAP called a *Big-Bag Matching Problem*, which is NP-complete [23].

This paper implements the Genome Map Assembly automaton and empirically investigates whether the time complexity for the average case is linear or exponential.

2.1 Related Work

The overlap multigraph method has several variations depending on how the subsequences are generated. Random substrings are picked from the DNA in Gillett et al. [6], Olsen et al. [18], Revesz [22], and Wong et al. [31] (which use restriction enzymes) or in Green and Green [7] and Harley et al. [9] (which use a technique called hybridization to test whether two substrings overlap). Lander and Waterman [16] gives some estimates about the number of random substrings of certain average size required to get a good coverage for a DNA of a given size. Unfortunately, the random-substrings approach has the limitation that there could be some gaps on the DNA, i.e., regions that are not covered by any randomly picked substring.

Derrida and Fink [3], Venter et al. [28], and Weber and Myers [30] developed the *whole-genome shotgun sequencing method*, which is another version of the overlap multigraph method applied to the entire genome.

Veeramachaneni et al. [27] abstract the problem of aligning fragmented sequences as an optimization problem and show the problem to be MAX-SNP hard. They also develop a polynomial time algorithm for their method. Kahveci and Singh [11] consider the problem of interactive substring searching using two different techniques: *local statistics-based interactivity* (LIS) and *global statistics-based interactivity* (GIS). These techniques have only 75 % accuracy [12].

One other abstraction of the genome map assembly problem, called the probed partial digestion problem, is considered in Tsur et al. [26]. Partial digestion occurs when a restriction enzyme fails to cut all the sites specific to it. Pop et al. [19] is a survey on the various genome sequence assembly algorithms.

Karp [14] shows several variations of the overlap multigraph problem to be NP-complete. Revesz [23] shows that the *Big-Bag Matching Problem* abstraction of the GMAP is also NP-complete.

3 Constraint Automata Solution

We now describe and illustrate the constraint automata solution of Revesz [20]. The original solution was a pure fingerprints-based method. The modification of *disambiguation* by checking the actual nucleotide strings in case of ambiguity in the fingerprint data during a search is a new idea we introduce in this paper.

Section 3.1 describes the constraint automata. The original automata was described as a non-deterministic automaton, but we give a presentation that is deterministic and uses the concept of "backtracking." Section 3.2 describes the implementation of the constraint automaton in the Perl programming language.

3.1 The Constraint Automata for GMAP

The working of the constraint automaton can be described easiest, if we first abstract the GMAP as a *Big-Bag Matching Problem.*

A *bag* is a multiset, a generalization of a set in which each element can occur multiple times [20]. A *big-bag* is a multiset whose elements are bags that can occur multiple times [20]. For example all the As can be put into a bag. Then when we replace each A_i by the bag describing its fingerprint, then we obtain a big-bag. Similarly, we can obtain a big-bag corresponding to the set of Bs.

Each permutation of the bags and permutation of the elements of each bag within a big-bag is called a *presentation* [20]. A big-bag can have several different presentations. The *big-bag matching decision problem* (BBMD) is the problem of deciding whether two big-bags match [20]. The *big-bag matching problem* (BBM) is the problem of finding matching presentations for two given big-bags if they match [20].

The following constraint automaton, shown in Fig. 10, uses "backtracking" and can give all the possible solutions.

The automaton uses the constraints "$\subseteq$" (subset) and "–" (set difference). It starts in the INIT state and ends in the HALT state. From the INIT state, the automaton moves from left to right by adding either bag A or B. Each bag represents a *contiguity constraint* for the elements it contains, i.e., any valid presentation must contain the elements within a bag next to each other. Therefore, adding a new bag really adds a new contiguity constraint to a set of other such constraints. Only if the set of constraints is solvable (and there could be several solutions) is the automaton allowed to continue with the next transition.

At any point either the list of A bags will be ahead of the list of B bags ("A_ahead" state) or vice versa ("B_ahead" state) or neither will be ahead, in which case it goes to the INIT state. The automaton has the following states: INIT, A_ahead, B_ahead, HALT and Backtrack. The state variables are: *UA* and *UB* indicating the set of unused A and B bags, respectively, S indicating the set of elements by which either the A or the B list is currently ahead, *Choices* indicating the set of options from which the next bag can be chosen, *SelBag* indicating the bag that was selected from *Choices* as the next bag, and *Cflag* indicating whether *Choices* is empty or not (if empty then it is set to "0", else it is set to "1"). The value of each state variable is saved after each transition.

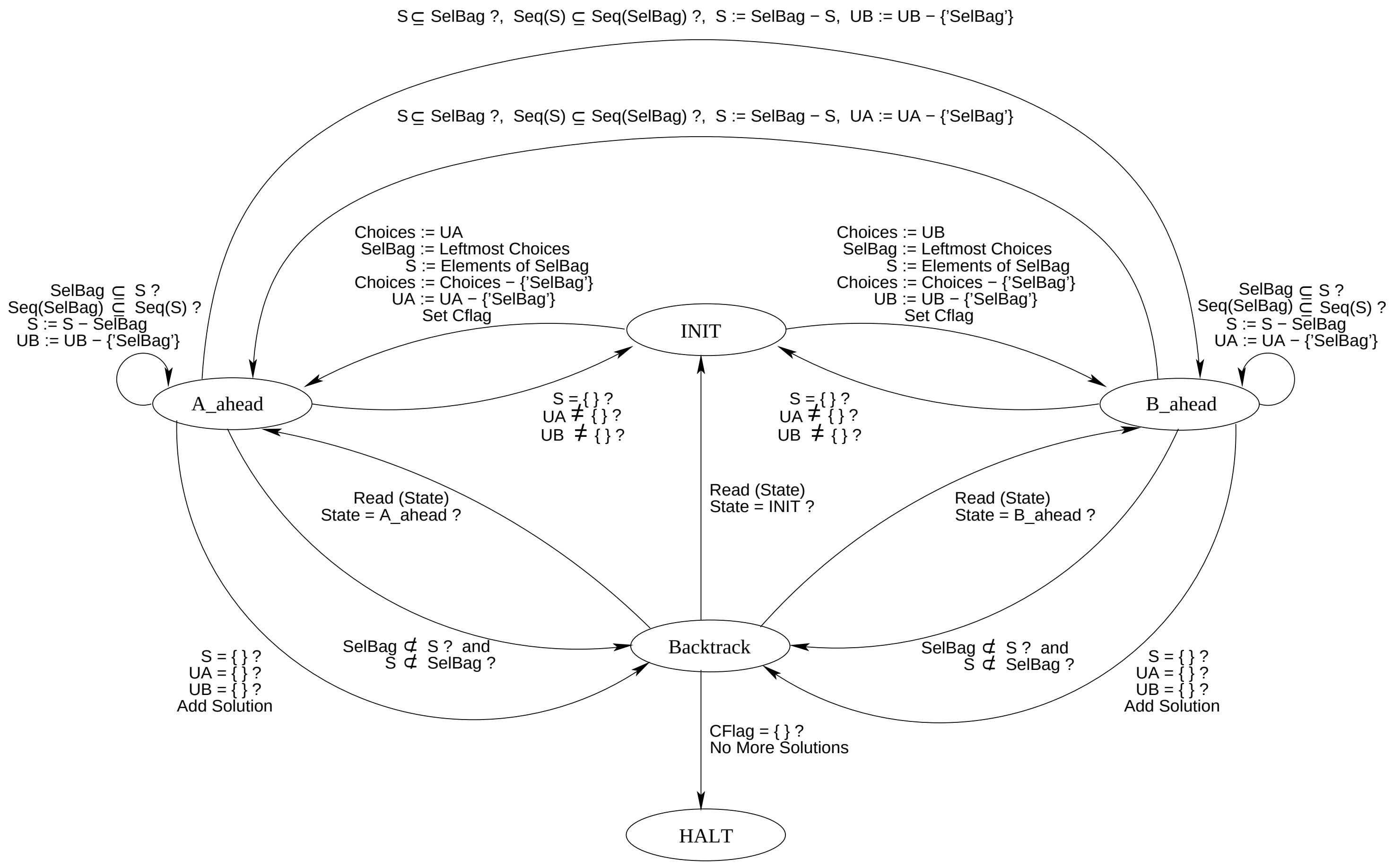

Fig. 10. Constraint Automata Solution

The automaton can be executed in ONE mode to find the first solution or ALL mode to find all the possible solutions. The working of the automaton is explained below.

1. The automaton is in the "INIT" state. The next bag is chosen from the list that has the lesser number of bags. If the next bag to be chosen is from the list of A bags then *Choices* will contain *UA*, else it will contain *UB*. The leftmost bag of *Choices* is removed from *Choices* and set to *SelBag*, the *Cflag* is set to "0" if *Choices* is empty or "1" if it is not empty, the elements of *SelBag* is set to *S*, and the automaton moves to either the "A_ahead" or the "B_ahead" state.

2. The set of options from which the next bag can be chosen is found by determining the bags which are either a subset or a superset of *S*. *Choices* contains this set of options.

3. If the automaton is in the "A_ahead" state, then the set of options is determined by *UB*. *SelBag* is the leftmost bag in *Choices*. We now need a *disambiguation* step to cut down on the choices. The nucleotide sequence corresponding to each element in *S* is compared with the nucleotide sequence of the corresponding element in *SelBag*. If they do not match, then the automaton moves to the "Backtrack" state. If the sequences of all the elements match, then

   (a) if $SelBag \subseteq S$, then the automaton remains in the "A_ahead" state and $S := S - SelBag$.

   (b) if $S \subseteq SelBag$, then the automaton moves to the "B_ahead" state and $S := SelBag - S$.

   In both cases, the elements of *S* and *SelBag* are matched as far as possible.

4. If the automaton is in the "B_ahead" state, then the set of options is determined from *UA* and the process followed is similar to that in Step 3.

5. If, in Steps 3 and 4, the difference of the values of *SelBag* and *S* is an empty set and *UA* and *UB* are not empty, then the automaton moves to the "INIT" state. If the difference is an empty set and *UA* and *UB* are also empty, then a solution has been found for the problem. If the execution of the automaton is in ONE mode then it moves to the "HALT" state and stops. If it is executed in ALL mode then the solution is saved and the automaton moves to the "Backtrack" state.

6. If, in Steps 3 and 4, *SelBag* is neither a subset nor a superset of S, then the automaton moves to the "Backtrack" state. In this state, the automaton will check for the last node whose *Cflag* has the value "1". The automaton then backtracks to this node and the information saved at this node is retrieved. Based on this information, the automaton will move to one of the states. If none of the nodes have their corresponding *Cflag* set to "1", then there is no more solution possible for the problem and the automaton moves to the "HALT" state and stops.

3.2 Implementation

The working of the automaton is similar to a pre-order traversal of a tree. Every node in the tree is a bag of either of the two big-bags. The two big-bags shown in Figs. 7 and 8 are used to explain the working of the automaton. The corresponding search tree is shown in Fig. 11. The dashed lines are part of the search tree only if we skip the disambiguation test. In the following we assume that we skip disambiguation. A partial pre-order traversal of the tree is shown step-wise in Table 1.

The execution starts at the origin (Node 0), which is the initial state (INIT). The set S is empty. The various options from which the automaton can choose a starting bag are all the bags in UA. Hence, the set *Options* contain all the bags of UA. A7 is taken as the starting bag (*SelBag*) and the remaining bags ($A1...A6$) are put into *Choices*. Since *Choices* is not empty, the flag *Cflag* for that node is set to "1". This means that there is another branch possible from this node.

The selected bag, $A7$, is then the node 1 of the tree and the current bag (*CurrBag*). S now contains the elements of bag $A7$. *UB* contains all the bags of big-bag B and *UA* contains all the bags of big-bag A except $A7$, which has been used. Since S contains the elements by which big-bag A is ahead of big-bag B, the automaton is in the "A_ahead" state. The next bag to be chosen is from *UB*, which contains the unused B bags. There is only one possible choice: $B6$. So we select bag $B6$. *Choices* is now empty, so we set *Cflag* to "0". Since the elements of S are contained in $B6$, we subtract the elements of S from the elements of $B6$. The resulting S will contain the elements by which big-bag B is ahead. So the automaton will move to the "B_ahead" state. Bag $B6$ is then removed from UB.

The execution continues in a similar fashion until it reaches node 6. At this node S is empty and there are no possible solutions. So the automaton moves to the "Backtrack" state. From the table we can see that the last node to have its *CFlag* set to "1" is node 2. So the automaton backtracks to that node. The bag in *Choices* is set to *SelBag*. Now *Choices* is empty, so *Cflag* is set to "0". The execution then continues as explained above.

At node 13, S, UA and UB are empty. Hence a solution is found. If the automaton is executed in ONE mode then it moves into the "HALT" state and the execution stops. If the execution is in ALL mode, then the solution is saved

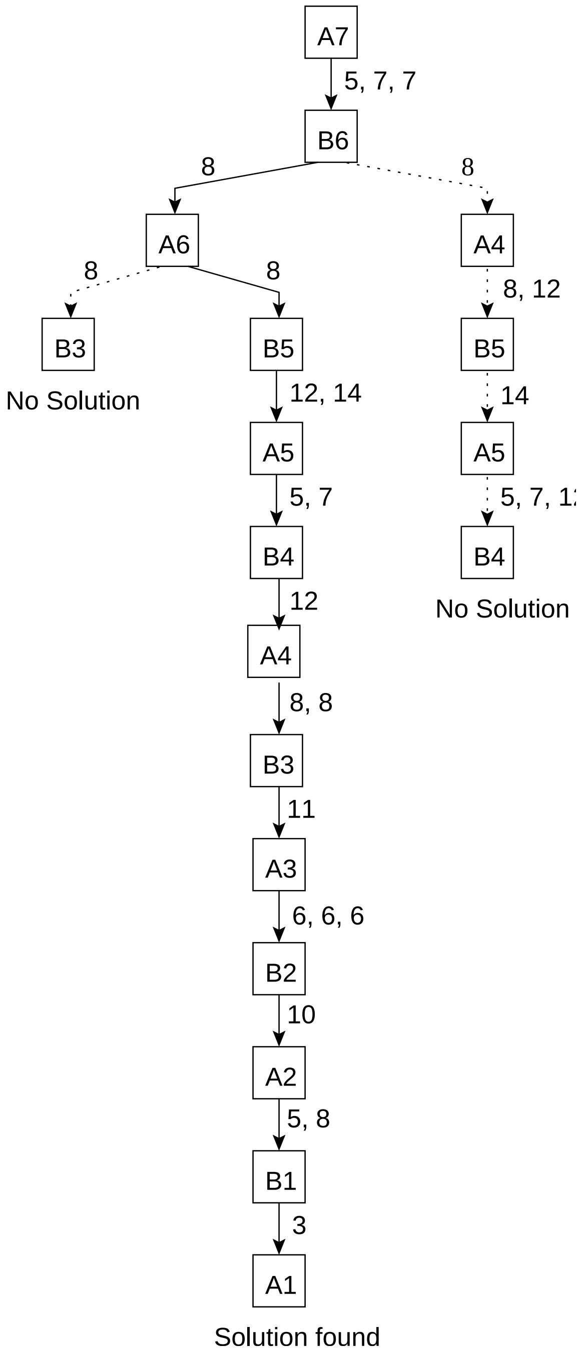

Fig. 11. The constraint automaton search tree from node $A7$.

Table 1. Step-wise partial pre-order traversal of the tree in Fig. 11

Node	CurrBag	S	UA	UB	State	Options	SelBag	Choices	Cflag
0	-	{}	A1,A2,A3,A4,A5,A6,A7	B1,B2,B3,B4,B5,B6	INIT	A1,...,A7	A7	A1,...,A6	1
1	A7	{5,7,7}	A1,A2,A3,A4,A5,A6	B1,B2,B3,B4,B5,B6	A_ahead	B6	B6	{}	0
2	B6	{8}	A1,A2,A3,A4,A5,A6	B1,B2,B3,B4,B5	B_ahead	A4,A6	A4	{A6}	1
3	A4	{8,12}	A1,A2,A3,A5,A6	B1,B2,B3,B4,B5	A_ahead	B5	B5	{}	0
4	B5	{14}	A1,A2,A3,A5,A6	B1,B2,B3,B4	B_ahead	A5	A5	{}	0
5	A5	{5,7,12}	A1,A2,A3,A6	B1,B2,B3,B4	A_ahead	B4	B4	{}	0
6	B4	{}	A1,A2,A3,A6	B1,B2,B3,B4	Backtrack	-	-	-	-
Cflag != 0; Backtrack to Node 2									
2	B6	{8}	A1,A2,A3,A4,A5,A6	B1,B2,B3,B4,B5	B_ahead	A6	A6	{}	0
3	A6	{8}	A1,A2,A3,A4,A5	B1,B2,B3,B4,B5	A_ahead	B3,B5	B3	{B5}	1
4	B3	{8,11}	A1,A2,A3,A4,A5	B1,B2,B4,B5	Backtrack	-	-	-	-
Cflag != 0; Backtrack to Node 3									
3	A6	{8}	A1,A2,A3,A4,A5	B1,B2,B3,B4,B5	A_ahead	B5	B5	{}	0
4	B5	{12,14}	A1,A2,A3,A4,A5	B1,B2,B3,B4	B_ahead	A5	A5	{}	0
5	A5	{5,7}	A1,A2,A3,A4	B1,B2,B3,B4	A_ahead	B4	B4	{}	0
6	B4	{12}	A1,A2,A3,A4	B1,B2,B3	B_ahead	A4	A4	{}	0
7	A4	{8,8}	A1,A2,A3	B1,B2,B3	A_ahead	B3	B3	{}	0
8	B3	{11}	A1,A2,A3	B1,B2	B_ahead	A3	A3	{}	0
9	A3	{6,6,6}	A1,A2	B1,B2	A_ahead	B2	B2	{}	0
10	B2	{10}	A1,A2	B1	B_ahead	A2	A2	{}	0
11	A2	{5,8}	A1	B1	A_ahead	B1	B1	{}	0
12	B1	{3}	A1	{}	B_ahead	A1	A1	{}	0
13	A1	{}	{}	{}	Backtrack	-	-	-	-
Cflag != 0; Backtrack to Node 1									

and the automaton moves into the "Backtrack" state. The last node to have its *Cflag* set to "1" is found and the automaton backtracks to that node.

Note that the search tree in Fig. 11 is much smaller than the search tree in Fig. 6, although both start from the same node $A7$. The search tree of Fig. 11 is even smaller, with the dashed lines deleted, if we use the disambiguation test. In general the constraint automaton will be smaller than the greedy search tree in the overlap multigraph for each starting node with a few exceptions.

There is also an interesting difference in the outcomes of the search when starting from node $A7$. The overlap multigraph method fails as shown in Fig 6. However, the constraint automaton search succeeds and returns the following sequence of A nodes:

$$A7\,A6\,A5\,A4\,A3\,A2\,A1$$

which is exactly the reverse order that was input. Like in nature, the information about left-to-right or right-to-left ordering is lost after restriction enzyme cuts. In general, the constraint automaton search can succeed starting from two nodes instead of only one in the overlap multigraph method. That also helps to increases the relative speed of the constraint automaton.

4 Results and Analysis

All experiments were carried out on an Athlon XP 1800+ desktop with 512 MB RAM and running the Windows XP Professional operating system. Perl v5.8.0 was used to implement the generation of input and the constraint automaton. The programs were executed in *Cygwin* v1.5.5-1, a Linux-like environment for Windows.

The input generation Perl program is executed on 70 sequences of the first four chromosomes of the mouse genome along with five sets of non-overlapping restriction enzyme triples [24], to generate the input big-bags. The lengths of the 70 sequences varied from 1.2 million to 36.8 million base pairs. The Perl implementation of the constraint automata takes these inputs and can be set by the user to find either only one or all possible solutions. The running times were measured for finding one solution.

4.1 Generation of Data and Results

The DNA sequences for generating the input for the Constraint Automata were taken from the Mouse Genome Resources website of the National Center for Biotechnology Information (NCBI) [17]. These sequences are various parts of the chromosomes of the mouse DNA. These sequences are subjected to restriction enzymes that are *compatible* with each other, that is, the recognition sequences of the enzymes do not overlap. For example, the three restriction enzymes `ClaI` (`AT^CGAT`), `BamHI` (`G^GATCC`), and `AvaI` (`C^YCGRG`) are compatible.

An example of a *non-compatible* restriction enzyme triple is `EcoRV` (`GAT^ATC`), `ClaI` (`AT^CGAT`) and `BamHI` (`G^GATCC`). That is non-compatible because the recognition sequence of `EcoRV` overlaps with the recognition sequence of `ClaI`. One overlap is that `ATC` is both the ending sequence of `EcoRV` and the starting sequence of `ClaI`. Another overlap is that `GAT` is both the ending of sequence of `ClaI` and the starting sequence of `EcoRV`.

If we use more than three restriction enzymes, there will be a large number of fragments. On the other hand, if we use less than three restriction enzymes, the number of fragments will be too small, resulting in huge fragment lengths. So we use three restriction enzymes to obtain a normal number of fragments, with a normal length.

The 70 sequences are subjected to the following five sets of non-overlapping restriction enzyme triples, which were carefully selected from a restriction enzyme database [24], to give us the input big-bags for the constraint automaton.

1. `HindIII, AccI, AceI`
2. `AauI, HindII, CacBI`
3. `BclI, AhyI, TaaI`
4. `PsiI, PciI, HhaII`
5. `PdiI, SurI, SspI`

The Perl implementation of the constraint automaton is then run using the input bags generated for each of the 70 sequences and the CPU time taken to find the first solution is calculated. This time is noted as the execution time. Further, the series of execution times is used as an indication of the time complexity of the constraint automata.

4.2 Data Analysis and Charts

Execution time data was subject to preliminary data cleaning procedures, which included checking for outliers and normality of distribution (i.e., skew) as per Hoaglin et al. [10] recommendations. Windsorizing procedures [10] were used for outlier analysis. Thus "too extreme" values were replaced with the "most extreme acceptable value". This has the advantage of not losing any data.

To investigate if the time complexity of the constraint automata solution is linear or exponential, we examined the relationship between the input for the constraint automata solution (number of bags) and the execution time. Using curve estimation regression modeling in SPSS, execution time was regressed onto the number of bags for each restriction enzyme triple. Thus five regression models were constructed, one for each restriction enzyme triple. Linear and exponential functions were plotted for each regression model and goodness of fit was determined by variance (R^2) estimates (Table 2) and by visual inspection of the charts (Figs. 12, 14, 16, 18, 20).

As seen in Table 2, a linear function provides a better fit (higher variance or R^2) than either a quadratic or an exponential function for all the restriction enzyme triples. A visual inspection of the charts in Figs. 12, 14, 16, 18, 20 also shows that the linear function is the best fit.

Table 2. R^2 values for the regression of execution time on number of bags

Rest. Enz.	Constraint Automata Solution			Overlap Multigraph		
Triple	Linear	Quadratic	Exponential	Linear	Quadratic	Exponential
1	0.897	0.872	0.814	0.869	0.931	0.959
2	0.908	0.869	0.804	0.875	0.860	0.936
3	0.976	0.846	0.825	0.904	0.868	0.942
4	0.963	0.902	0.865	0.874	0.925	0.956
5	0.909	0.848	0.819	0.887	0.877	0.937
Avg R^2	0.925	0.867	0.825	0.882	0.892	0.946

The corresponding model for the overlap multigraph is shown in the charts in Figs. 13, 15, 17, 19, 21. From these charts we see that an exponential function provides the best fit for all the restriction enzyme triples.

5 Relation to Constraint Databases

It is well-known that a constraint automaton can be translated into a Datalog query on a constraint database [13, 15, 21] that contains the input data. In our case the Datalog query needs only Boolean algebra equality, inequality and precedence constraints, where the particular Boolean algebra used is the one whose elements are finite subsets of the natural numbers and whose operators are interpreted as the set union, intersection and complement with respect to the set of natural numbers.

The latest version of the DISCO constraint database system [2] implements constraints over the Boolean algebra of the powerset of natural numbers. Hence we could have also implemented the constraint automaton in the DISCO system, although the solution would have been slower, and we could not have modified it as we liked to control backtracking. We chose Perl for the implementation of the constraint automaton, because it allowed us to control backtracking, including the occasional look-up of the nucleotide sequence for disambiguation, and because it is currently a standard language for genomics implementations.

Although our choice may disappoint some constraint database purists, they may take heart in the fact that constraint databases were useful in the discovery of the algorithm. It is by attempting to describe the GMAP in a Boolean algebra of sets, that the abstraction of the entire problem as a Big-Bag Matching Problem was discovered. After that the constraint automaton solution was a easy and natural step. Further, early prototyping in the DISCO system convinced us that the algorithm may be practical in practice and led to the present paper. Hence constraint databases played an essential part in the discovery of the algorithm.

Our experience illustrates a general principle that fewer tools are often better. A general purpose programming language provides too many tools for a

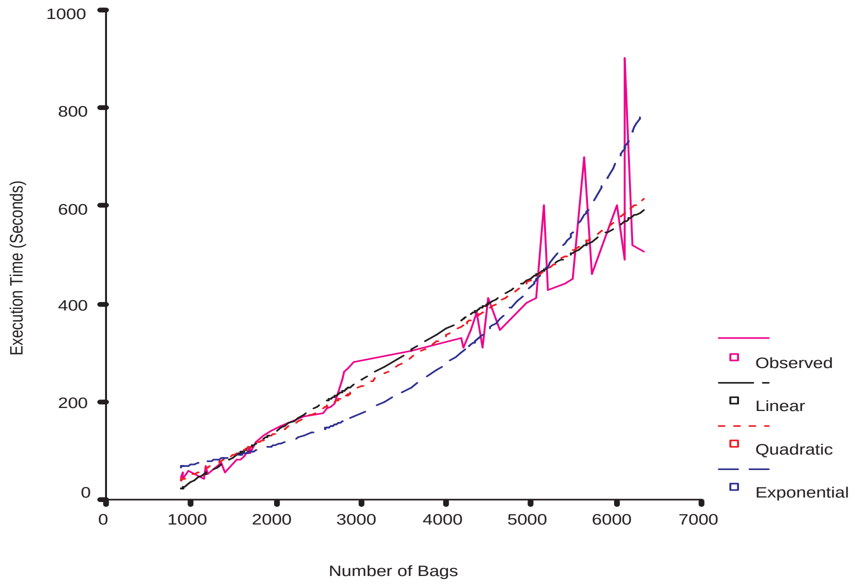

Fig. 12. Regression model for RE triple 1 for constraint automata.

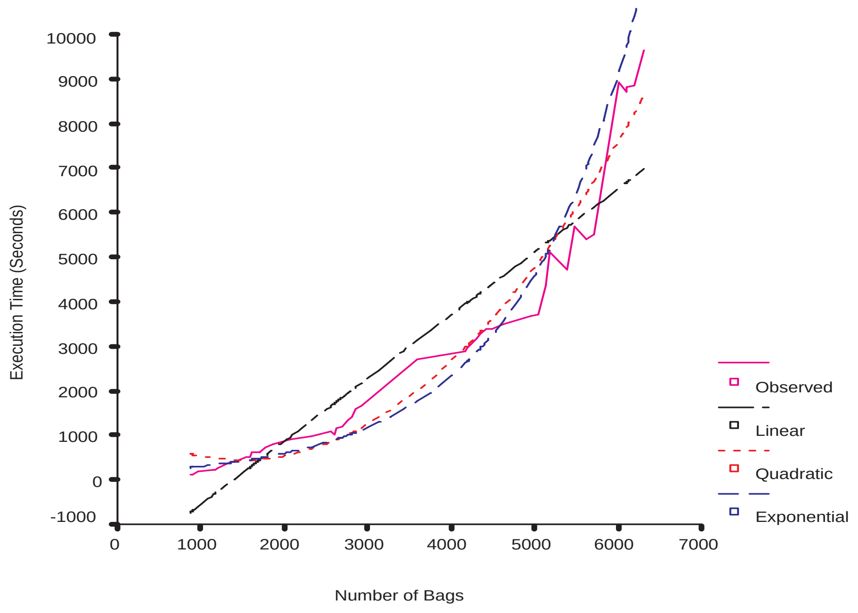

Fig. 13. Regression model for RE triple 1 for overlap multigraph.

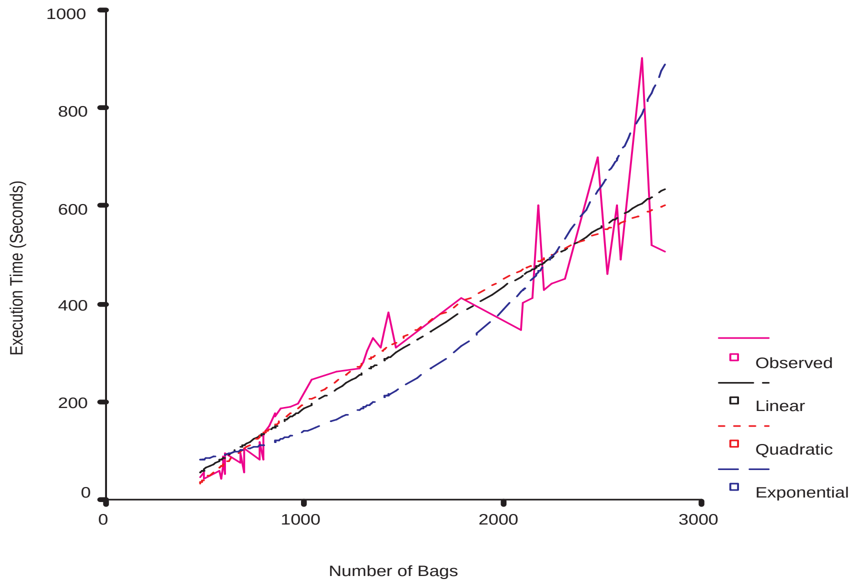

Fig. 14. Regression model for RE triple 2 for constraint automata.

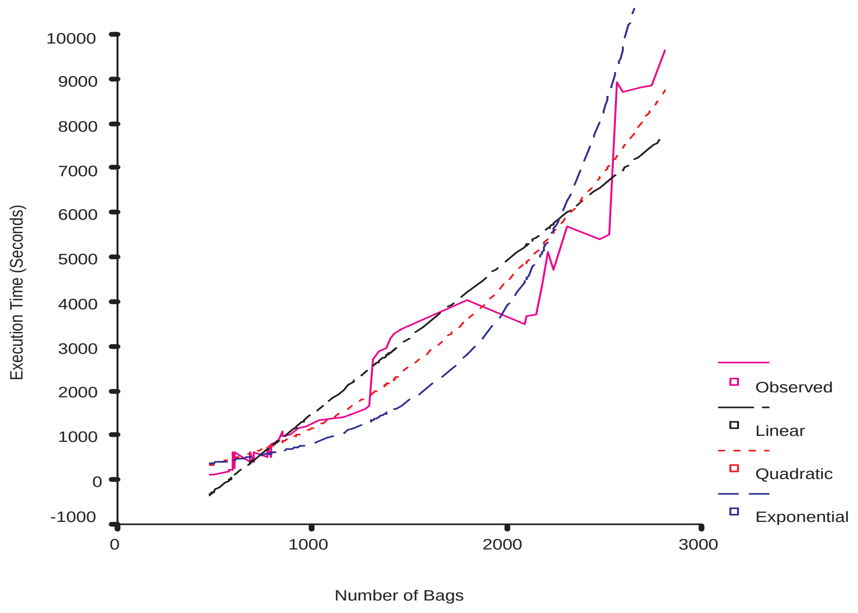

Fig. 15. Regression model for RE triple 2 for overlap multigraph.

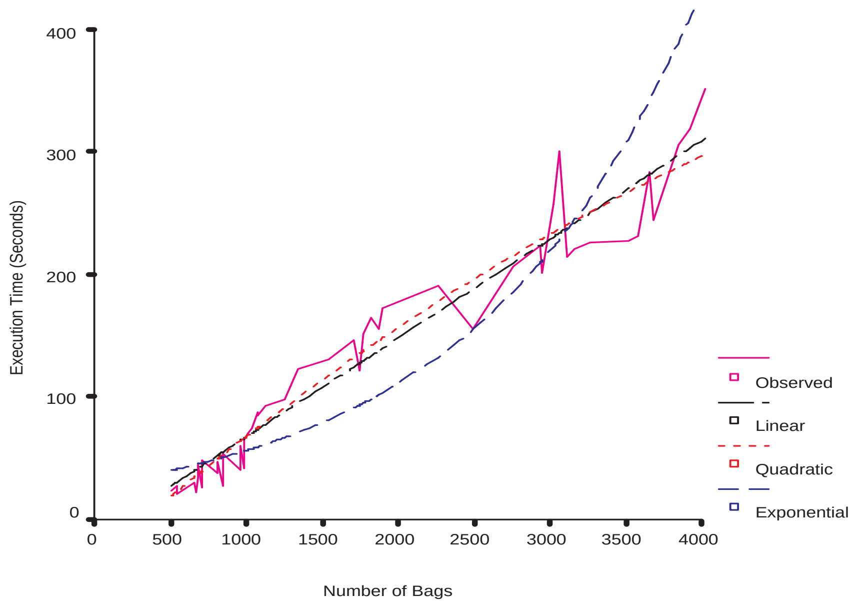

Fig. 16. Regression model for RE triple 3 for constraint automata.

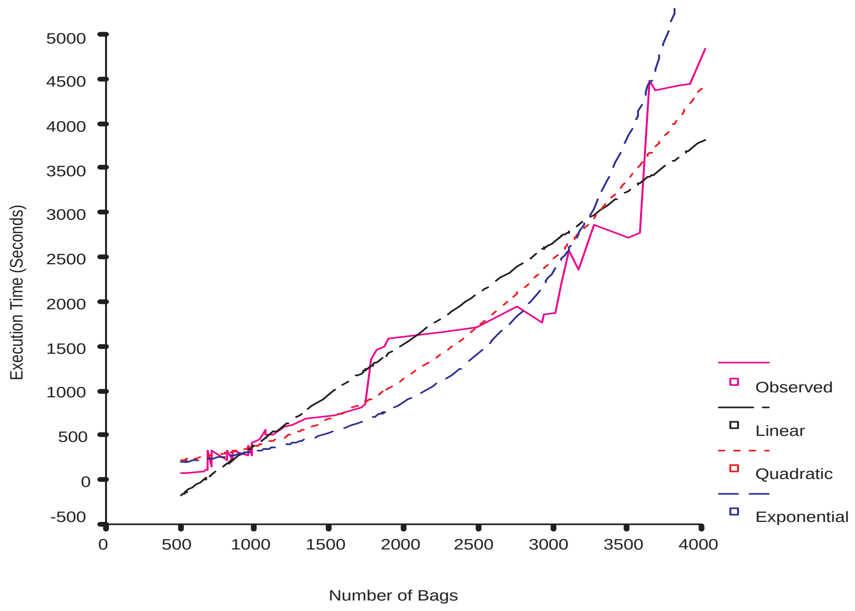

Fig. 17. Regression model for RE triple 3 for overlap multigraph.

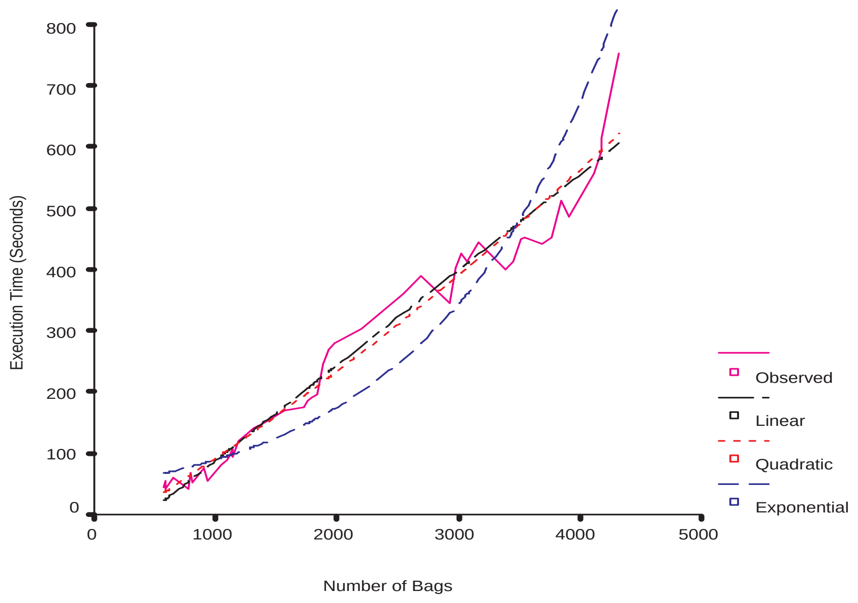

Fig. 18. Regression model for RE triple 4 for constraint automata.

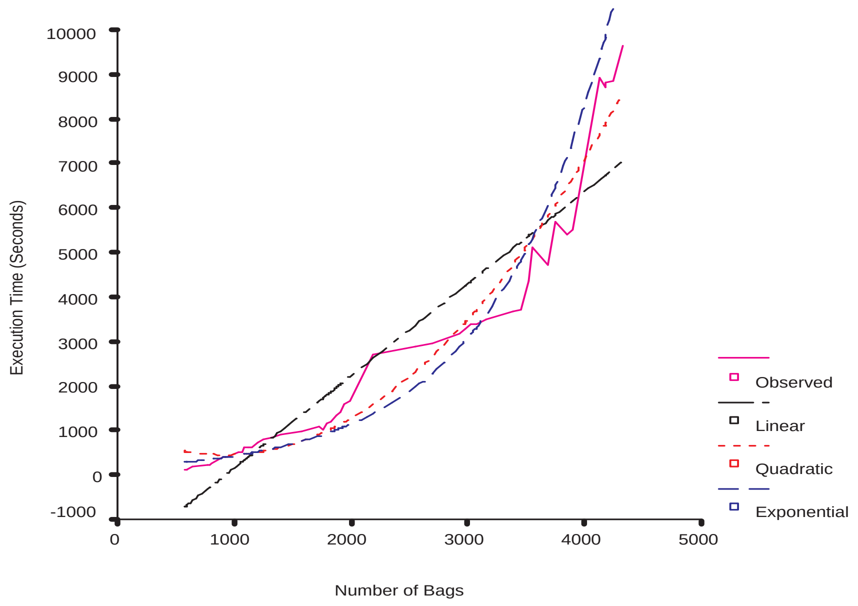

Fig. 19. Regression model for RE triple 4 for overlap multigraph.

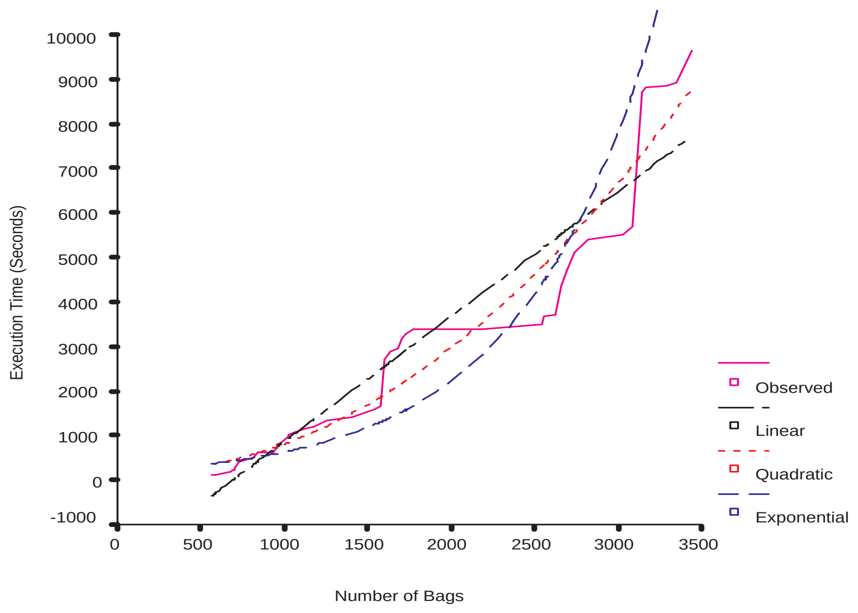

Fig. 20. Regression model for RE triple 5 for constraint automata.

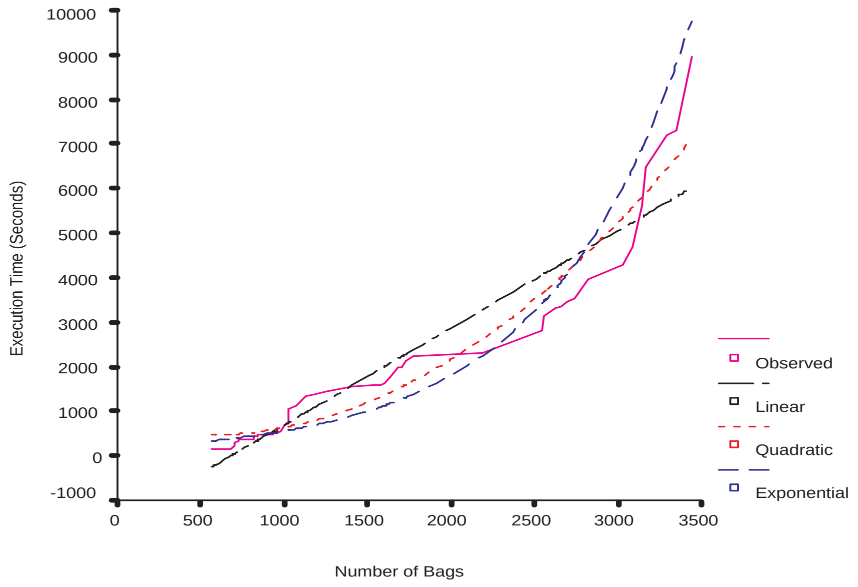

Fig. 21. Regression model for RE triple 5 for overlap multigraph.

programmer to work with. That is not always as helpful as it may seem at first glance, because one is lost in the enormous number of options, the endless number of possible data structures to implement and the also endless number of possible functions to implement on them. With a constraint language, one is not so lost. The programmer has the tool of the constraint language itself and is hence forced to express the problem using a high-level constraint abstraction.

This process either works or does not. As the old saying goes, if you have a hammer every problem starts to look like a nail. If the problem is indeed like a nail, then everything is fine. If it is not, then a new tool, in this case a new programming language, is needed. But that does not mean that starting with a hammer was a bad thing. For without the hammer we may not start to see even nail-like problems as such. The GMAP is a problem that can be expressed using Boolean algebra constraints and a simple automaton. This is a useful abstraction that one who only works with Perl may not discover so easily. Hence the process worked in the case of GMAP.

6 Conclusions and Future Work

We examined the relationship between the input for the constraint automata solution (number of bags) and the execution time using curve estimation regression modeling. This was done for five compatible restriction enzyme triples. Both variance estimates (R^2) and a visual inspection of the charts indicate that a linear function provides the best fit for the constraint automaton and an exponential provides the best fit for the overlap multigraph solution for the GMAP. The experimental results provide strong support for the earlier hypothesis that the constraint automaton solution is better than the overlap multigraph solution for large genomes.

In the future, we still need to evaluate the applicability of the constraint automaton solution to genome mapping in real biochemical laboratories. The constraint automaton can be further adapted to incorporate *error tolerance* as the actual DNA fragments and their length measures are not necessarily error-free. For example, if a fingerprint is $\{97, 120, 355\}$ and another fingerprint is $\{96, 121, 357\}$, then the two can be matched with an error tolerance of 2. The time complexity function for the error-prone data can then be compared with the time complexity for the error-free data. As the error tolerance value increases, the choices in the search increase slowing down the execution time of the constraint automaton. However, it is hypothesized that the constraint automaton with error tolerance will also show a linear time complexity on average.

We also discussed the advantages of trying to attack a problem using multiple programming languages. The more different are the programming languages, the greater is the potential benefit in trying each of them. The constraint automaton solution was discovered with ease after trying Datalog with Boolean algebra constraints. We hope that others will also experience with delight the ease of solving problems in constraint databases.

References

1. J.M. Berg, J.L. Tymoczko, and L. Stryer. *Biochemistry, 5 ed.* W.H. Freeman, New York, 2002.
2. J. Byon and P. Revesz. DISCO: A Constraint Database with Sets. *Proc. Workshop on Constraint Databases and Applications*, Springer LNCS 1034, pp. 68-83, 1995.
3. B. Derrida and T.M.A. Fink. Sequence determination from overlapping fragments: A simple model of whole-genome shotgun sequencing. In *Physical Review Letters*, 88(6):068106, 2003.
4. S.E. DeWeerdt. *What's a Genome?* The Center for Advanced Genomics, 2003.
5. Entrez Database. http://www.ncbi.nlm.nih.gov/entrez/
6. W. Gillett, L. Hanks, G.K-S. Wong, J. Yu, R. Lim, and M.V. Olsen. Assembly of high-resolution restriction maps based on multiple complete digests of a redundant set of overlapping clones. *Genomics*, 33:389-408, 1996.
7. E.D. Green and P. Green. Sequence-tagged site (STS) content mapping of human chromosomes: Theoretical considerations and early experiences. *PCR Methods and Applications*, 1:77-90, 1991.
8. E. Harley and A.J. Bonner. A flexible approach to genome map assembly. In *Proc. International Symposium on Intelligent Systems for Molecular Biology*, pages 161-69. AAAI Press, 1994.
9. E. Harley, A.J. Bonner, and N. Goodman. Good maps are straight. In *Proc. 4th International Conference on Intelligent Systems for Molecular Biology*, pages 88-97, 1994.
10. D.C. Hoaglin, F. Mosteller, and J.W. Tukey. *Understanding Robust and Exploratory Data Analysis.* John Wiley, New York, 1983.
11. T. Kahveci and A.K. Singh. Genome on demand: Interactive substring searching. In *Proceedings of the Computational Systems Bioinformatics.* IEEE Computer Society, 2003.
12. T. Kahveci and A.K. Singh. An interactive search technique for string databases. Technical Report 10, UCSB, 2003.
13. P. Kanellakis, G. Kuper, and P. Revesz. Constraint query languages. *Journal of Computer and System Sciences*, 51:26-52, 1995.
14. R.M. Karp. Mapping the genome: Some combinatorial problems arising in molecular biology. In *Proc. 25th ACM Symposium on Theory of Computing*, pages 278-85. ACM Press, 1993.
15. G. Kuper, L. Libkin and J. Paredaens: *Constraint Databases*, Springer, 2000.
16. E.S. Lander and M.S. Waterman. Genomic mapping by fingerprinting random clones: A mathematical analysis. *Genomics*, 2:231-9, 1988.
17. Mouse Genome Resources. http://www.ncbi.nlm.nih.gov/genome/guide/mouse/
18. M.V. Olson, J.E. Dutchik, M.Y. Graham, G.M. Brodeur, C. Helms, M. Frank, M. MacCollin, R. Scheinman, and T. Frank. Random-clone strategy for genomic restriction mapping in yeast. *Genomics*, 83:7826-30, 1986.
19. M. Pop, S.L. Salzberg, and M. Shumway. Genome Sequence Assembly: Algorithms and Issues. *Computer*, pages 47-54, July 2002.
20. P. Revesz. Bioinformatics. In *Introduction to Constraint Databases*, pages 351-60. Springer, New York, 2002.
21. P. Revesz. *Introduction to Constraint Databases*, Springer, New York, 2002.
22. P. Revesz. Refining restriction enzyme genome maps. *Constraints*, 2(3-4):361-75, 1997.

23. P. Revesz. The dominating cycle problem in 2-connected graphs and the matching problem for bag of bags are NP-complete. In *Proc. International Conference on Paul Erdos and His Mathematics*, pages 221-5, 1999.
24. R. J. Roberts. *REBASE: The Restriction Enzyme Database.* New England Biolabs, http://rebase.neb.com/rebase/rebase.html, 2003.
25. J. Setubal and J Meidanis. Fragment Assembly of DNA. In *Introduction to Computational Molecular Biology* pages 118-24, PWS Publishing, Boston, 1997.
26. S. Tsur, F. Olken, and D. Naor. Deductive databases for genome mapping. In *Proc. NACLP Workshop on Deductive Databases*, 1993.
27. V. Veeramachaneni, P. Berman, and W. Miller. Aligning Two Fragmented Sequences. *Discrete Applied Mathematics*, 127(1):119-43, 2003.
28. J.C. Venter, M.D. Adams, G.G. Sutton, A.R. Kerlavage, H.O. Smith, and M. Hunkapiller. Shotgun sequencing of the human genome. *Science*, 280:1540-2, 1998.
29. D. Voet and J. Voet. *Biochemistry, 3 ed., Vol. 1.* John Wiley, New York, 2003.
30. J.L. Weber and E.W. Myers. Human whole-genome shotgun sequencing. *Genome Research*, 7(5):401-9, 1997.
31. G.K-S. Wong, J. Yu, E.C. Thayer, and M.V. Olson. Multiple-complete-digest restriction fragment mapping: Generating sequence-ready maps for large-scale DNA sequencing. In *Proc. National Academy of Sciences, USA*, 94(10):5225-30, 1997.