Recombination Systems

Mikko Koivisto, Pasi Rastas, and Esko Ukkonen[⋆]

Department of Computer Science, P.O. Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki, Finland
{mikko.koivisto, pasi.rastas, esko.ukkonen}@cs.helsinki.fi

Abstract. We study biological recombination from language-theoretic and machine learning point of view. Two generative systems to model recombinations are introduced and polynomial-time algorithms for their language membership, parsing and equivalence problems are described. Another polynomial-time algorithm is given for finding a small model for a given set of recombinants.

1 Introduction

Recombination is one of the main mechanisms producing genetic variation. Simply stated, recombination refers to the process in which the DNA molecules of a father chromosome and a mother chromosome get entangled and then split off to produce the DNA of the child chromosome, composed of segments taken alternately from the father DNA and the mother DNA (Fig 1) [1].

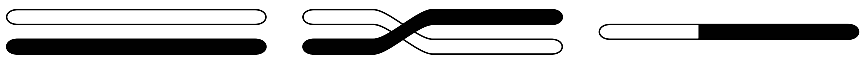

Fig. 1. Recombination

Combinatorial structures created by iterated recombinations have attracted lots of interest recently. The discovery of so-called haplotype blocks [3, 5] has also inspired the development of new efficient algorithms for the analysis of structural regularities of the DNA, from various perspectives; e.g. [9, 4]. Some methods for genetic mapping such as the recent approach of [7] also model recombinations.

In this paper we study recombination from language-theoretic and machine learning point of view. Two simple systems are introduced to generate recombinants starting from certain founding strings. Membership, parsing and equivalence problems for these systems turn out in general easy. More interesting and also much harder is the problem of inverting recombinations: given a sample set of recombinants we want to construct a smallest possible system generating a language that contains the sample.

[⋆] Supported by the Academy of Finland under grant 201560.

The paper is organized as follows. Section 2 introduces simple recombination systems. Such a system is specified just by giving a set of strings, the "founders" of a population. Section 3 introduces another system, called the fragmentation model, in which the strings that can be used as segments of recombinants are listed explicitly. Language membership, parsing and equivalence problems for these two systems are polynomial-time solvable, by well-known techniques from finite automata [6] and string matching [2]. In Section 4 we consider a machine learning type of problem of constructing a good fragmentation model for a sample set of recombinants. We give a polynomial-time algorithm that finds a smallest model in a special case. Also in the general case the algorithm seems useful although the result is not necessarily minimal.

2 Simple Recombination Systems

A *recombination* is an operation that takes two strings u and v of equal length n and produces a new string w, also of length n, called a *recombinant* of u and w, such that

$$w = xy$$

where x is a prefix of u and y is a suffix of v or x is a prefix of v and y is a suffix of u. The recombinant w is said to have a *cross-over* at location $|x|$. For simplicity we assume that a recombinant may have only one cross-over. As x or y may be the empty string, u and v themselves are recombinants of u and v.

Let A be a set of strings of length n. The set of strings generated from A in one recombination step is denoted

$$\mathcal{R}(A) = \{w \mid w \text{ is a recombinant of some } u, v \in A\}.$$

Let Σ be a finite alphabet. A *simple* $m \times n$ *recombination system* in Σ is defined by a set $F \subseteq \Sigma^n$ consisting of m strings of length n in Σ. The strings in F are called the *founders* of the system. System F generates new sequences by iterating the recombination operation. The generative process has a natural division into generations giving the corresponding languages $G_0(F), G_1(F), \ldots$ as follows:

$$\begin{aligned} G_0(F) &= F \\ G_1(F) &= \mathcal{R}(F) \\ &\vdots \\ G_i(F) &= \mathcal{R}\Big(G_{i-1}(F)\Big). \end{aligned}$$

As $G_0(F) \subseteq G_1(F) \subseteq \cdots \subseteq G_i(F) \subseteq \cdots \subseteq \Sigma^n$ there must be j such that after the jth generation nothing new can be produced, that is, $G_{j'}(F) = G_j(F)$ for all $j' \geq j$. We call $L(F) = G_j(F)$ the *full recombinant language* of system F.

Example 1. Let $\Sigma = \{0, 1\}$, $n = 4$, and consider 2×4 system $F = \{0000,\ 0111\}$. Then $G_1(F) = \{0000,\ 0111,\ 0100,\ 0110,\ 0011,\ 0001\}$ and $G_2(F) =$ $\{0000,\ 0111,\ 0100,\ 0110,\ 0011,\ 0001,\ 0101,\ 0010\}$. Language $G_2(F)$ consists of all strings in Σ^4 that start with 0. This is also the full language $L(F)$.

It should be obvious that w is in $L(F)$ if and only if w can be written as

$$w = \alpha_1 \alpha_2 \cdots \alpha_p \tag{1}$$

for some non-empty strings $\alpha_i \in \Sigma^+$ such that each α_i occurs in some founder string $f_j \in F$ at the same location as in w. That is, we have $f_j = \gamma \alpha_i \delta$ for some γ such that $|\gamma| = |\alpha_1 \cdots \alpha_{i-1}|$. Each decomposition (1) of w into *fragments* α_i is called a *parse* of w with respect to F.

String w may have several different parses. Two of them are of special interest. First, if w has some parse (1) then it also has a parse such that $|\alpha_i| = 1$ for all $i = 1, 2, \ldots, p$ and $p = n$. We then note that a string $w_1 w_2 \cdots w_n$, where $w_i \in \Sigma$, belongs to $L(F)$ if and only if for each w_i there is some $f_j \in F$ whose ith symbol is w_i. Let us denote by Σ_i the symbols in Σ that occur at the ith location of some string in F. We call Σ_i the *local alphabet* of F at i. Summarized we get the following simple result.

Theorem 1. $L(F) = \Sigma_1 \Sigma_2 \cdots \Sigma_n$ □

This immediately gives a language equivalence test for recombination systems. Let E and F be two recombination systems of length n, and let $\Pi_1, \Pi_2, \ldots,$ Π_n be the local alphabets of E and $\Sigma_1, \Sigma_2, \ldots, \Sigma_n$ the local alphabets of F. Then $L(E) = L(F)$ if and only if $\Pi_i = \Sigma_i$ for all $i = 1, 2, \ldots, n$. So, for example systems $\{0000,\ 1111\}$ and $\{0101,\ 1010\}$ are equivalent as all local alphabets are equal to $\{0, 1\}$.

The simplicity of the equivalence test also indicates that the sequential structure of the founders has totally disappeared in $L(F)$. Therefore it is more interesting to look at strings that have a parse consisting of a small number of fragments α_i. This leads us to define the canonical parses.

Let $w \in L(F)$. Then a parse $w = \alpha_1 \alpha_2 \cdots \alpha_p$ of w with respect to F is *canonical*, if

1. p is smallest possible; and
2. among parses of w with p fragments, each $|\alpha_1 \alpha_2 \cdots \alpha_i|$, $1 \leq i < p$, is largest possible.

A canonical parse of w is easily seen unique. It can be found by the following *greedy parsing algorithm*. First find the longest prefix of w that is also a prefix of some string in F. This prefix is fragment α_1 of the canonical parse. Then remove $|\alpha_1|$ symbols long prefix from w and from all members of F. Repeat the same steps to find longest prefix that becomes α_2, and so on, until the entire w has been processed or it turns out that parsing can not be continued to the end of w, in which case $w \notin L(F)$.

We will use the number $p - 1$ of the cross-overs in the canonical parse as a distance measure for strings: the *recombination distance* $\rho(w, F)$ between w

and F is $p-1$, the smallest possible number of cross-overs in a parse of w with respect to F. Note that $\rho(w,F) \leq n-1$ for all $w \in L(F)$. If $w \notin L(F)$, we let $\rho(w,F) = \infty$.

The greedy parsing algorithm finds $\rho(w,F)$. The algorithm works without any preprocessing of F and can be organized to run in time $O(mn)$, i.e. linear in the total length of the strings in F. We now describe a preprocessing of F which constructs a collection of trie structures such that canonical parsing of any string with respect of F can be done in optimal time $O(n)$.

In the canonical parsing by the greedy method one has to find the longest prefix of the current suffix of w that is common with the corresponding suffix of some founder $f \in F$. Let $w^i = w_i w_{i+1} \cdots w_n$ and $f_j^i = f_{ji} f_{ji+1} \cdots f_{jn}$ denote the ith suffixes of w and the founders, and let T^i denote the trie representing strings $f_1^i, f_2^i, \cdots, f_m^i$. The longest common prefix, that will become the first fragment of the parse, can be found by traversing the path of T^1 for w^1 until a symbol of w^1 is encountered, say w_h, that is not present in T^1 (or w^1 ends). The scanned prefix is the fragment α_1 of the canonical parse. We needed $O(|\alpha_1|)$ time to find it this way. The parsing continues by next traversing the path of T^h for w^h, giving α_2 in time $O(|\alpha_2|)$, and so on.

To make this work we need the tries $T^1, T^2, \ldots, T^n$. A straightforward construction of a trie for $m = |F|$ strings of length n takes time $O(mn)$ assuming that $|\Sigma|$ is constant. Hence the total time for all tries would be $O(mn^2)$. We next describe a suffix-tree based technique for constructing these tries in time $O(mn)$.

The suffix-tree of a string x is a (compacted) trie representing all the suffixes of x. The size of the tree is $O(|x|)$, and it can be constructed in time $O(|x|)$ by several alternative algorithms [2]. To get the tries T^h that form our parser for F we first augment the founder strings with explicit location indices, such that founder string $f_i = f_{i1} f_{i2} \cdots f_{in}$ becomes $\hat{f_i} = (f_{i1}, 1)(f_{i2}, 2) \cdots (f_{in}, n)$. Now construct the suffix-tree T for string $\hat{f} = \hat{f_1} \hat{f_2} \cdots \hat{f_m}$. Then trie T^h consists of the subtrees of T representing suffixes that start with symbols (a, h), where $a \in \Sigma$. Hence tries T^h can be extracted from T in one scan through the edges that are adjacent to the root.

This construction can be performed in $O(mn)$ time, i.e., linear time in the length of $\hat{f}$ although we have formally used alphabet of non-constant size $|\Sigma|n$. This is because the non-root nodes of T may only have $|\Sigma|$ branches and hence the branching degree at such nodes does not depend on n. While the root node can have $O(|\Sigma|n)$ branches, the dependency on n can be made constant by direct indexing (or bucketing) on the second component of a symbol.

Finally note that the tries T^h extracted from T are of compacted form, i.e., the non-branching nodes of the trie are represented only implicitly. The edges of a compacted trie correspond to strings (instead of single symbols), represented by pairs of pointers to the original strings in F. In our greedy parsing algorithm such tries can be used as well, without significant overhead.

Theorem 2. *Given an $m \times n$ recombination system F, a greedy parser for F can be constructed in time $O(mn)$. For any string w, the parser computes in time $O(n)$ the canonical parse of w with respect to F and the recombination distance $\rho(w, F)$.* □

Canonical parsing is not the only possible use of the parser of Theorem 2. All possible parses of w can be generated if, instead of greedily traversing the current trie as far as possible, the parsing jumps to the next trie at any point on the way. It is also possible to check whether or not w has a parse with given cross-over points: Then the parsing should jump to the next trie exactly on these points. The parses can also be utilized to find a string w with largest possible distance $\rho(w, F)$.

3 Generalized Recombination Systems and Fragmentation Models

Parsing a string as introduced in the previous section means decomposing the string into fragments taken from the founders. The available fragments were implicitly defined by the founders: any substring of a founder can be used in a parse.

We now go one step further and introduce models in which the available fragments are listed explicitly.

A *fragmentation model* of length n in alphabet Σ is a state-transition system $M = (S, Q, \Sigma, n)$ consisting of a finite set S of the *states* and a set Q of *transitions*. Each $s \in S$ is a pair (i, v), where i is an integer $1 \leq i \leq n$, and string $v \in \Sigma^*$ is the *fragment* of the state such that $|v| \leq n - i + 1$. We call $b(s) = i$ the *begin location* and $b(s) = i + |v|$ the *end location* of s. A state s is a *start state* if $b(s) = 1$ and an *end state* if $e(s) = n + 1$. The transition set Q is any subset of $S \times S$ such that if $(r, s) \in Q$ then $e(r) = b(s)$, that is, the location intervals covered by r and s should be next to each other.

The language $L(M)$ of M consists of all strings generated along the transition paths from a start state to an end state. More formally, $e \in L(M)$ if and only if there are states $s_1, s_2, \ldots s_p$ such that $(s_i, s_{i+1}) \in Q$ for $1 \leq i < p$, s is a start state and s_p is an end state, and $w = v_1 v_2 \cdots v_p$ where v_i is the fragment of state s_i. Note that all $w \in L(M)$ are of length n. Also note that fragmentation models are a subclass of finite-state automata. Hence for example their language equivalence is solvable by standard methods [6].

Example 2. A simple $m \times n$ recombination system F of the previous section consisting of m founders $f_j = f_{j1} f_{j2} \cdots f_{jn}$ can be represented as a fragmentation model $M = (S, Q, \Sigma, n)$ as follows: set S consists of all states (i, v) where $1 \leq i \leq n$ and $v = f_{ji} f_{ji+1} \cdots f_{jh}$ for some $1 \leq j \leq m$ and $i \leq h \leq n$. The transition (r, s) is included into Q for all r, s such that $e(r) = b(s)$. Note that M is much larger than F. It has $O(mn^2)$ states and $O(m^2 n^3)$ transitions, and the fragments of the states have total length $O(mn^3)$. □

Each transition path gives for the generated string a parse into fragments. Different parses for the same string can be efficiently enumerated and analyzed for example by using dynamic programming combined with breadth-first traversal of the transition graph of M. We describe next an algorithm for finding a parse with smallest number of fragments, i.e., a shortest path through M that generates the string to be parsed.

Let w be the string to be parsed. We say that state (i, v) of M *matches* w if $w = xvy$ where $|x| = i - 1$. We associate with each state s a counter $c(s)$ whose value will be the length of a shortest path to s that will generate the prefix of length $b(s) - 1$ of w. Variable P will be used to store the length of a shortest parse. The parsing algorithm is as follows:

```
1. Let $s_1, s_2 \dots s_t$ be the states of $M$ ordered according to increasing value of $e(s_j)$
2. Initialize the counters
       $P \leftarrow \infty$
       $c(s_j) \leftarrow \begin{cases} 0 & \text{, if } s_j \text{ is a start state} \\ \infty & \text{, otherwise} \end{cases}$
3. for $j \leftarrow 1, 2, \dots, t$ do
       if $c(s_j) < \infty$ and $s_j$ matches $w$ then
           if $s_j$ is an end state then
               $P \leftarrow \min(P, c(s_j) + 1)$
           else
               for all $s_k$ such that $(s_j, s_k) \in Q$
                   $c(s_k) \leftarrow \min(c(s_k), c(s_j) + 1)$
```

The algorithm can be implemented such that the running time is linear in the size of M. We also observe that testing whether or not some states of M and w match can be done very fast by first constructing Aho-Corasick multi-pattern matching automaton [2] for the fragments of the states and then scanning w with this automaton.

4 Model Reconstruction Problems

The language membership and equivalence well as parsing problems for recombination systems turned out solvable by fast algorithms, not unexpectedly as we are dealing with a limited subclass of the regular languages. We now discuss much harder problems concerning inversion of recombinations.

Given a set D of strings of length n we want to find a model that could have generated D. This question was addressed in [8] in the case of simple recombination systems. For example, an algorithm was given in [8] that constructs an $m \times n$ recombination system F such that $D \subseteq L(F)$ and the average recombination distance of the elements of D from F is minimized. Here we will consider the problem of finding fragmentation models for D. The fragments of such a model can be thought to represent the "conserved" substrings of D.

The goodness of a fragmentation model M for set D can be evaluated using various criteria. A possibility is to consider probabilistic generalizations of fragmentation models and apply model selection criteria such as the Minimum Description Length principle. We will resort to combinatorial approach and consider the following parsimony criterion: find a fragmentation model $M = (S, Q, \Sigma, n)$ such that $D \subseteq L(M)$ and the number of states in Q is smallest possible. We call this the *minimal fragmentation model reconstruction problem*.

Example 3. Let D consist of strings

$$\begin{array}{c} 0\,0\,0\,0\,1\,1 \\ 0\,0\,0\,1\,1\,1 \\ 0\,0\,1\,0\,1\,1 \\ 0\,0\,0\,0\,0\,0 \\ 0\,0\,0\,1\,0\,0 \\ 1\,1\,1\,0\,1\,1 \\ 1\,1\,0\,0\,0\,0 \\ 1\,1\,0\,1\,0\,0 \\ 1\,1\,1\,0\,0\,0 \end{array} \tag{2}$$

By taking the strings in D as such (and nothing else) as the fragments we get a fragmentation model which generates exactly D and has 9 states. However, the model depicted in Fig 2 has only 7 states. It generates a language that properly contains D. □

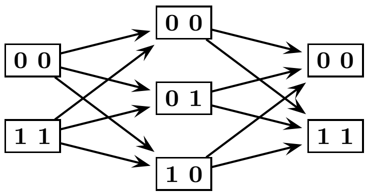

Fig. 2. A fragmentation model (begin locations of states not explicitly shown)

We rephrase now the minimal fragmentation model reconstruction in terms of certain tilings of D. Let us refer to the m strings in D by $1, 2, \ldots, m$; the ith string is $d_{i1}d_{i2}\cdots d_{in}$. Then any triple $\tau = (A, h, k)$, where $A \subseteq \{1, 2, \ldots, m\}$ and h and k are integers such that $1 \leq h \leq k \leq m$, is a *tile* of D. Set A is the *row-set* of τ. The tile τ *covers* all substrings $d_{ih}d_{ih+1}\cdots d_{ik}$ where $i \in A$. The tile is *uniform* if all substrings it covers are equal, i.e., $d_{ih}\cdots d_{ik} = d_{jh}\cdots d_{jk}$ for all $i, j \in A$. A set T of tiles of D is a *uniform tiling* of D if the tiles in T are uniform and disjoint and cover D, i.e., for each d_{ij} there is exactly one tile in T that covers d_{ij}.

A fragmentation model M such that $D \subseteq L(M)$ induces a uniform tiling $T(M)$ of D as follows. Fix for each string $d \in D$ a path of M that spells out d. For any state $s = (i, v)$ of M, let A be the set of strings in D whose path goes through s. Then add the tile $(A, i, i + |v| - 1)$ to $T(M)$. It should be clear that $T(M)$ is a uniform tiling of D. Note that there are several different tilings $T(M)$ if some $d \in D$ is ambiguous with respect to M, i.e., if M has more than one path for d.

On the other hand, given a uniform tiling T of D, one may construct a fragmentation model $M(T)$ as follows. For each tile $(A, h, k) \in T$, add to $M(T)$ a state $s = (h, v)$ where $v = d_{ih} \cdots d_{ik}$ for some $i \in A$. Also add a transition (s, s') to $M(T)$ if the tiles (A, h, k) and (A', h', k') in T that correspond to s and s' are such that row-set intersection $A \cap A'$ is nonempty and $k + 1 = h'$.

As the number of states of $M(T)$ equals the number of tiles in T, and the number of tiles in $T(M)$ is at most the number of states of M, we get the following result.

Proposition 1. *The number of states of the smallest fragmentation model M such that $D \subseteq L(M)$equals the number of tiles in the smallest uniform tiling of D.*

To solve the minimal fragmentation model reconstruction we will construct small uniform tilings for D. We will proceed in two main steps. First a rather simple dynamic programming algorithm is given to find optimal tilings in a subclass called the column-structured tilings. In the second step we apply certain local transformations to further improve the solution.

A uniform tiling of D is *column-structured* if the tiles cover D in columns: for each two tiles (A, h, k) and (A', h', k'), if $h = h'$ then $k = k'$. The corresponding class of fragmentation models (models whose fragments with the same begin location are of equal length) is also called column-structured models. If a column-structured tiling is smallest possible, then the number of tiles in each column should obviously be minimal. Such minimal tiling for a column is easy to find as follows. Consider set $D(h, k)$ consisting of strings $d_{ih} \cdots d_{ik}$ for $1 \leq i \leq m$. Let $A_1, A_2, \ldots A_p$ be the partition of $\{1, 2, \ldots, m\}$ such that i and j belong to the same class A_r if and only if $d_{ih} \cdots d_{ik} = d_{jh} \cdots d_{jk}$. Then the tiling $\Big((A_1, h, k), \ldots, (A_p, h, k)\Big)$ of $D(h, k)$ is uniform and has the smallest possible number of tiles among tilings whose tiles are from h to k. We denote this tiling by $t(h, k)$ and its size p by $\sigma(h, k)$.

Let $S(j)$ be the size of smallest column-structured tiling of $D(1, j)$. Then $S(j)$ can be evaluated for $j = 0, 1, \ldots, n$ from

$$\begin{cases} S(0) = 0 \\ S(j) = \min_{i<j} \Big(S(i) + \sigma(i + 1, j)\Big) \end{cases} \tag{3}$$

and $S(n)$ gives the size of smallest column-structured uniform tiling of entire D. The usual trace-back of dynamic programming can be used to find the end locations $j_1, j_2, \ldots, j_q = n$ of the corresponding columns. Then the smallest tiling itself is $t(1, j_1) \cup t(j_1 + 1, j_2) \cup \cdots \cup t(j_{q-1} + 1, n)$.

Evaluation of (3) takes time $O(n^2)$ plus the time for evaluating tables σ and t which can be accomplished in time $O(n^2m)$ using straightforward trie-based techniques. We have obtained the following theorem.

Theorem 3. *Minimal column-structured fragmentation model for D can be constructed in time $O(n^2m)$ where n is the length and m the number of strings in D.*

Example 4. The fragmentation model in Fig 2 for the set (2) of Example 3 is column-structured and minimal. If string 011010 is added to (3), then algorithm (3) will give the column-structured model in Fig 3(a). However, the model in Fig 3(b) is smaller. □

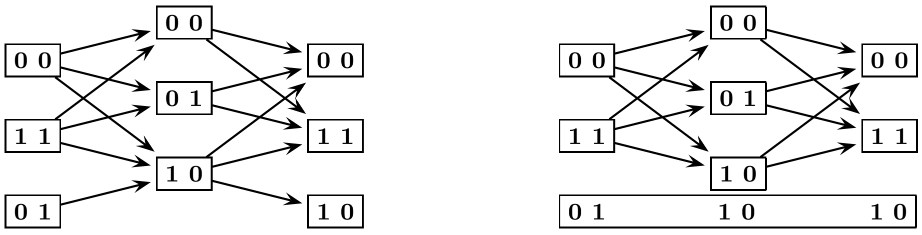

Fig. 3. (a) A column-structured fragmentation model (b) A smaller model

The tilings given by the column-wise approach can further be improved by applying local transformations. The transformations use the following basic step. Assume that our current tiling has adjacent tiles $(A, h, k-1)$ and (B, k, r). We may replace these tiles by the tiles

$$\begin{aligned}&(A \cap B, h, r),\\&(A \setminus B, h, k-1),\\&(B \setminus A, k, r),\end{aligned}$$

and the tiling stays uniform and covers still the entire D. The replacement operation has no effect if row-set $A \cap B$ is empty. Otherwise it changes the structure of the tiling. If $A = B$, the number of tiles is reduced by one; if $A \subseteq B$ or $B \subseteq A$, the number stays the same; and if $A \cap B$, $A \setminus B$ and $B \setminus A$ are all non-empty, the number increases by one.

Given any tiling T we can improve it by the following iterative reduction rule: apply the above local transformation on any pair of tiles $(A, h, k-1)$ and (B, k, r) such that $A \subseteq B$ or $B \subseteq A$ (i.e., transformation does not increase the number of tiles). Repeat this until the local transformation is not any more applicable. It is easy to see that the process stops in $O(mn)$ iterations. Note that the seemingly useless transformation steps that do not make the number of tiles smaller are indirectly helpful: they make the tiles narrower (and longer) and hence may create possibility for true size reduction in the next step.

There are two possible ways to include the reduction step into algorithm (3). On can apply it only on the final result of (3). This would, for example, improve the tiling in Fig 3(a) into that in Fig 3(b). Other possibility is to apply the reduction rule also on the intermediate tiling obtained for each $D(1, j)$ during algorithm (3) and to use the reduced tiling in subsequent computation. Sometimes this strategy will give better results than the previous one.

There is also another local transformation that makes the tiles longer and narrower without reducing the number of tiles. This transformation eliminates certain loop-like structures from the tiling, defined as follows. The *inclusion graph* of a tiling T at j is a bipartite graph which has as its nodes all tiles (A, h, k) and (A', h', k') such that $k = j - 1$ and $h' = j$ and as its (undirected) arcs all $\big((A, h, j-1), (A', j, k')\big)$ such that row-set intersection $A \cap A'$ is not empty. A connected component of this graph is a *simple loop* if it contains as many nodes as arcs. In a simple loop every node has degree 2 (i.e., two arcs are adjacent to a node). The number of tiles in a simple loop equals the number of their non-empty pairwise row-set intersections. But this means that applying our local transformation on all such pairs will keep the number of tiles unchanged. Hence the loop-removal transformation can safely be added to the local transformations one should apply to make the tiling smaller.

Summarized, we get an optimization algorithm that combines dynamic programming and local transformations. It finds a local optimum with respect to the local transformations. Running-time is polynomial in the size of D.

5 Conclusion

We introduced two simple language–generating systems inspired by the recombination mechanism of the nature. For the model reconstruction problem we delineated some initial results while many questions remained open, most notably the complexity status and approximability of the minimal model reconstruction. Probabilistic generalizations of our models are another interesting direction for further study.

References

1. Creighton H. and McClintock B.: A correlation of cytological and genetical crossing-over in Zea mays. *PNAS* **17** (1931), 492-497
2. Crochemore, M. and Rytter, W.: *Jewels of Stringology.* World Scientific 2002
3. Daly, M., Rioux, J., Schaffner, *et al.*: High-resolution haplotype structure in the human genome. *Nature Genetics* **29** (2001), 229–232
4. Koivisto, M., Perola, M., Varilo, *et al.*: An MDL method for finding haplotype blocks and for estimating the strength of haplotype block boundaries. In: *Pacific Symposium on Biocomputing (PSB2003),* pp. 502–513. World Scientific 2003
5. Patil, N., Berno, A. and Hinds, D.A. *et al.*: Blocks of limited haplotype diversity revealed by high-resolution scanning of human chromosome 21. *Science* **294** (2001), 1719–1723
6. Salomaa, A.: *Jewels of Formal Language Theory.* Computer Science Press 1981
7. Sevon, P., Ollikainen, V. and Toivonen, H.T.T.: Tree pattern mining for gene mapping. *Information Sciences* (to appear)

8. Ukkonen, E.: Finding founder sequences from a set of recombinants. In: *Algorithms in Bioinformatics (WABI 2002),* LNCS 2452, pp. 277–286. Springer 2002
9. Zhang, K., Deng, M., Chen, T., *et al.*: A dynamic programming algorithm for haplotype block partitioning. *PNAS* **99** (2002), 7335–7339