Journal of Bioinformatics and Computational Biology
Vol. 5, No. 2(a) (2007) 201–250
© Imperial College Press

DESIGN AND ANALYSIS OF AN EFFICIENT RECURSIVE LINKING ALGORITHM FOR CONSTRUCTING LIKELIHOOD BASED GENETIC MAPS FOR A LARGE NUMBER OF MARKERS

S. TEWARI
Department of Statistics, University of Georgia,
Athens, GA 30602-1952, USA
statsusant@yahoo.com

S. M. BHANDARKAR*
Department of Computer Science, University of Georgia,
Athens, GA 30602-7404,USA
suchi@cs.uga.edu

J. ARNOLD
Department of Genetics, University of Georgia,
Athens, GA 30602-7223,USA
arnold@uga.edu

Received 2 October 2006
Revised 2 December 2006
Accepted 24 January 2007

A multi-locus likelihood of a genetic map is computed based on a mathematical model of chromatid exchange in meiosis that accounts for any type of bivalent configuration in a genetic interval in any specified order of genetic markers. The computational problem is to calculate the likelihood (L) and maximize L by choosing an ordering of genetic markers on the map and the recombination distances between markers. This maximum likelihood estimate (MLE) could be found either with a straightforward algorithm or with the proposed recursive linking algorithm that implements the likelihood computation process involving an iterative procedure is called *Expectation Maximization* (EM). The time complexity of the straightforward algorithm is exponential without bound in the number of genetic markers, and implementation of the model with a straightforward algorithm for more than seven genetic markers is not feasible, thus motivating the critical importance of the proposed recursive linking algorithm. The recursive linking algorithm decomposes the pool of genetic markers into segments and renders the model implementable for hundreds of genetic markers. The recursive algorithm is shown to reduce the order of time complexity from exponential to linear in the number of markers. The improvement in time complexity is shown theoretically by a worst-case analysis of the algorithm and

*Corresponding author.

supported by run time results using data on linkage group-II of the fungal genome *Neurospora crassa*.

Keywords: Exchange; EM algorithm; recursive linking; time complexity; MLE.

1. Introduction

High density linkage maps are an essential tool for characterizing genes in many systems, fundamental genetic processes, such as genetic exchange between chromosomes, as well as the analysis of traits controlled by more than one gene (i.e. complex traits).[13] Since, genetic maps are most often the critical link between phenotype (what a gene or its product does) and the genetic material, genetic maps can be exploited to address how a particular trait[7] is controlled by one or more genes. Most model systems possess high density linkage maps that can assist in the analysis of complex traits. The bread mold, *Neurospora crassa* (*N. crassa*),[5] which gave us the biochemical function of genes, is no exception.[18] One approach to understanding the genetic basis of a complex trait is to follow, its segregation in offspring along with an array of genetic markers. One class of genetic markers frequently used are restriction fragment length polymorphisms (RFLPs), markers in the DNA itself. Another class are single nucleotide polymorphisms (SNPs). These markers, in essence, allow a triangulation on loci in the DNA affecting the complex trait of interest. Part of this triangulation process involves the construction of the relative positions of these several markers along a genetic map. This is a computationally challenging problem[12] and at the heart of understanding complex traits, such as human disease. In this paper, we address the problem of genetic map reconstruction from a large number of RFLP markers. We focus on map construction for a model system *N. crassa*,[2] where there is a wealth of published information about how markers segregate because the genetic makeup of gametes (as opposed to offspring) can be identified.[2] In this setting we can build a much more realistic model of the recombination process between chromosomes than in more complex eukaryotes.[23]

Given l markers or genetic loci, each with two or more alternate forms of a gene called *alleles*, the number of distinct types of offspring is 2^l. This implies that the computational complexity of a likelihood-based approach to estimating a genetic map would appear to scale at least as $O(25^{l-1})$.[25] At first sight the computational complexity of following the segregation of l markers to build a genetic map seems infeasible beyond about seven markers. It is remarkable that Lander and Green[12] were able to solve this problem for a special case with an algorithm whose computational complexity is linear in l. The likelihood, in their algorithm, is computed from a recombination fraction defined on each genetic interval. The limitation of their work is that they did not model the recombination process in detail; for example, chromatid exchanges are not explicitly modeled as discussed in the next section. Others have tried to circumvent the computational complexity of this problem by utilizing only pairwise information on genetic loci.[17] Solving

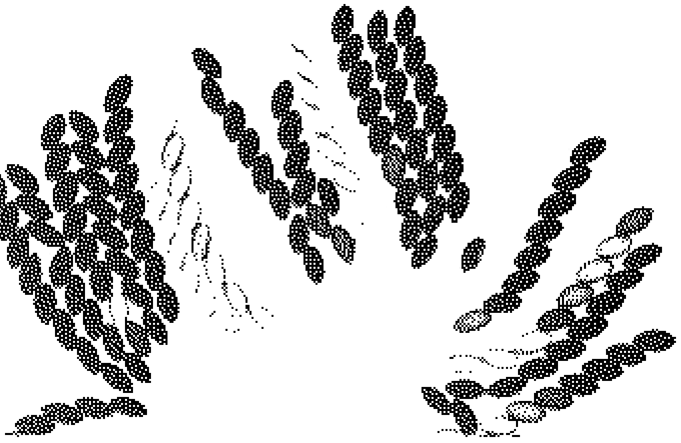

Fig. 1. Various forms of tetrads: (a) unordered; (b) linear; and (c) normally maturing asci of *N. crassa*. (From Raju[21] and Davis[5]).

this reconstruction problem is essential for geneticists to make use of the recently completed International HapMap[1] with thousands of markers scattered throughout the human genome to hunt down important disease causing genes. In addition, several model systems now possess the data for constructing dense genetic maps with thousands of markers.[28] Using such mapping data together with variation in a complex trait, such as heart disease, will allow researchers to triangulate on genes determining the trait of interest. Here, we focus on solving this problem in a setting considered ideal for geneticists. For some organisms, it is possible to observe the gametes (as opposed to the offspring) from a parent organized into a package called a tetrad[21] uncomplicated by what is going on in another parent. For organisms such as fungi and some more complex eukaryotes engineered to have this property,[4] gametes can be observed directly, as opposed to offspring with the contributions of the two parents. For this reason, we focus on reconstructing a genetic map where tetrads can be obtained. In fungi, such as *N. crassa*, the gametes can be typed directly (Fig. 1). A string of spores (or gametes) in Fig. 1 are the products of single cross. In this setting the recombination process can be modeled in detail. The challenge we address here is, in the best of all possible worlds can we reconstruct a genetic map with many markers? This is a very old and difficult problem without a good solution (particularly when the order of markers is unknown) in spite of the fact that hundreds of fungal laboratories around the world make use of this kind of tetrad data in genetic analysis. Zhao *et al.*[30,32] demonstrate a solution for a few number of markers (<10) under some assumptions on the exchange process.

2. Background Material on Meiosis, Recombination, and Exchange

The following biological background section is introduced to make the paper self-contained. The vast majority of genes in cells with nuclei (eukaryotes) occur on chromosomes. Depending on the organism and life stage, a eukaryotic species can have 1 to several copies of these chromosomes. For example, many eukaryotes, such as animals and flowering plants, possess two copies of their chromosomes and are referred to as *diploid*, while many fungi and algae may possess one copy of their chromosomes and are referred to as *haploid*. The ploidy for these organisms can vary with their life cycle. For example, diploid animals and plants produce haploid gametes; conversely, haploid fungi often produce a temporary diploid stage (a *meiocyte*) during sexual reproduction. The letter n is used to refer to the number of distinct chromosomes in a haploid condition. So, gametes have n chromosomes, and diploids have $2n$ chromosomes.

The process of recombination underlies the reconstruction of genetic maps and is thought to take place principally during a cell division process called *meiosis* underlying sexual reproduction. To understand how recombination takes place between chromosomes, it is necessary to have an understanding of how meiosis proceeds. In eukaryotes, meiosis is the production of specialized cells (such as sperm and eggs) called gametes. Prior to the onset of meiosis the genetic material is completely duplicated, and during meiosis two divisions take place to convert a diploid cell, for example, with $2n$ chromosomes into four haploid gametes called a tetrad, each with n chromosomes. This process is termed meiosis.

As a diploid cell, a meiocyte, enters meiosis, the meiocyte contains at least four copies of each chromosome, because DNA replication has taken place already. In the succeeding rounds of two cell divisions, the number of chromosomes in each daughter cell is reduced to one copy. The four copies of each chromosome in the diploid meiocyte are referred to as *chromatids*. During the earliest stage of meiosis, *prophase*, these four chromatids align into a structure called a *bivalent* (see Fig. 2). Two pairs of these chromatids are nearly identical, one being descended from the other by DNA replication immediately prior to meiosis. The related chromatids

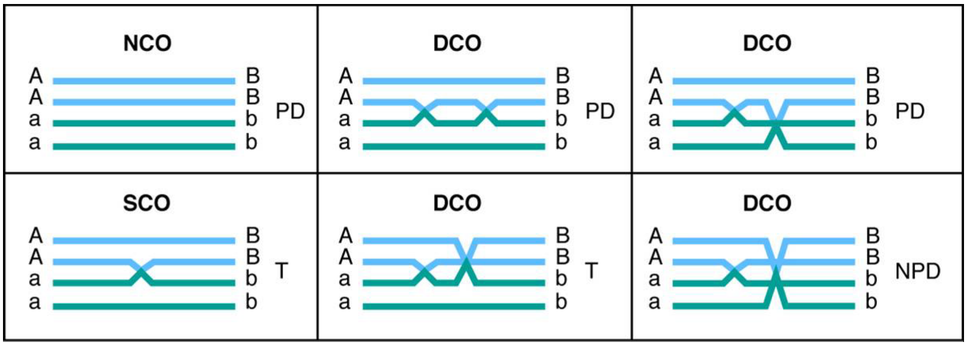

Fig. 2. The ascus classes produced by exchanges between linked loci. NCO, nonexchange meioses; SCO, single-exchange meioses; DCO, double-exchange meioses. Chromosomes of the same color are sister chromatids. (redrawn from http://www.ncbi.nlm.nih.gov/books/bv.fcgi?rid=iga.figgrp.1135)

are called *sister chromatids* (same color in Fig. 2), and if the two chromatids are not related by the immediately prior round of DNA replication, they are *nonsister chromatids*. A bivalent then consists of two pairs of sister chromatids.

The physical proximity of the chromatids in Prophase of meiosis leads to exchanges between chromatids. A exchange can arise as shown in Fig. 2. These exchanges are generated by random double stranded breaks, in which the ends of chromatids reanneal with the wrong chromatid as shown in Fig. 2. The resulting exchanges can take place anywhere along the chromatids, and the positions of the exchanges vary from meiocyte to meiocyte (as well as their products, i.e. from gamete to gamete). In that sister chromatids are so similar, exchanges are usually only observed chromatids between nonsister chromatids. Let us designate the sister chromatids from one parent as 1 and 2, and the sister chromatids from the other parent, 3 and 4. Exchanges are then usually only observed between chromatids 1 and 3, 1 and 4, 2 and 3, and 2 and 4 as depicted in Fig. 3. It is plausible that exchanges between each of these combinations of nonsister chromatids are equally likely because the exchanges are generated by random breakage events along the chromatids. This hypothesis or assumption is referred to as *No-Chromatid-Interference* (NCI). There is substantial empirical support for this hypothesis.[29]

The process of crossing-over underlying meiotic recombination shuffles the allele pairs in the diploid parent and deals them out randomly as the products of meiosis (such as egg and sperm). For clarity, *meiotic recombination* can be defined as the production of gametes from meiosis with genotypes that differ from the parental genotypes that combined to form the (diploid) parental meiocyte. Put more simply, if the descendant does not resemble the parent in its genotype, we say the descendant is a *recombinant*. The product of meiotic recombination is referred to as a recombinant, and the physical process generating a recombinant is hypothesized to be *crossing-over*. Thus, the hypothesis is that the breakage-and-rejoining process leading to exchanges is an explanation for recombination of genes on chromatids, making children different from their parents.

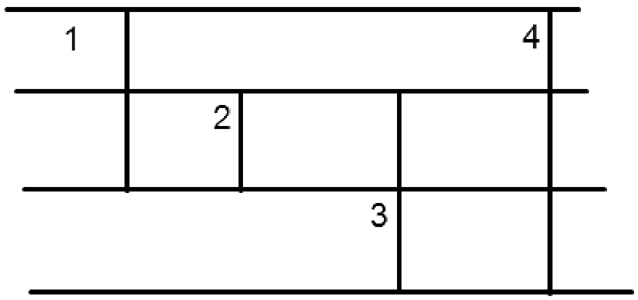

Fig. 3. The single exchange events are signified by a vertical line, and the exchanges take place between chromatids at the ends of the vertical lines. These four strands are found in Prophase-I.[5] All exchanges are equally likely under our model.

There are two different flavors of meiotic recombination: *independent assortment* of genes on different chromosomes and *crossing-over* between genes on the same chromosome. To see genetic recombination, it is necessary that the pair of genes involved in recombination in the diploid meiocyte each have two different alleles. That is, each gene has to be *heterozygous* in the diploid parental meiocyte. For constructing a genetic map, we only consider genes on the same chromosome, and thus do not consider independent assortment. In Fig. 2 we present how recombination works. In Fig. 2, one parental genotype (a.b) is labeled with all mutant alleles, and the other parent is labeled with what are called wild type alleles (A.B). A typical meiotic product from a single exchange (SCO) a.B would be termed a recombinant by the definition above, as the recombinant is genetically distinct from the haploid parents, a.b and A.B. In Fig. 2, other double exchange events, such as double exchanges (DCO), lead to different allelic combinations that are recombinant as well. The tetrads from one meiosis can then be classified as *parental ditype* (PD), in which no recombinants occur; as *tetrad type* (T), in which equal numbers of recombinants and non-recombinants co-occur; or as *non-parental ditype* (NPD), in which only recombinants occur.

3. Model Assumptions in Comparison to Existing Models

One critical feature of the exchanges is how they occur along a chromosome. One assumption that has been made by Lander and Green[12] is that crossovers along a chromosome are independent and uniformly distributed. This assumption of no crossover interference, referred to as the (NChI) hypothesis enabled the likelihood calculation to be tractable in their work.[12] For phase-known type data Lander and Green compute the likelihood of an order of markers by simply computing the fraction of recombinants for each interval. Although this give a likelihood against which to compare marker orders relative to each other, it does not go into the recombination process in detail. They do not consider the chromatid exchanges at all which is crucial to describing recombination. The problem with not having a model is that likelihood measures could differ arbitrarily for small changes in the marker order. The absence of a model also implies that there are no parameters one can monitor in order to examine the changes in the process underlying the model in response to the changes in the marker order.

There have been several approaches to modeling the crossover process presented in a series of papers (see Refs. 14, 30 and 32). For example, Zhao and Speed formulate a more general model of crossing-over based on a stationary renewal process[31] to derive the likelihood formulation based on the popular Chi-square model,[8] as a special case. They also perform statistical tests to evaluate the performance of the model for several important data sets.[30] The Chi-square model is observed to fit the data well while accounting for interference. However there are a few points that limit their model for broader application. In all of the models that have been proposed so far, the crossover process has been assumed to follow some mathematically tractable process, and several assumptions on the process are generally made to make it viable. For example, in the Chi-square model[8] the so-called "recombinational intermediates" (C) have been assumed to be uniform across the chromosome (i.e. with no interference) while their resolutions are assumed to follow a particular pattern. Though the model gives a measure of interference through its parameter m, the meaning of the model itself is quite abstract. For example, the model states that a non-exchange resolution (C_0) must result m times after each crossover resolution (C_x) followed by a crossover resolution (C_x).

Zhao and Speed have worked out very general expressions for the joint probability of multilocus recombinants in Theorem 2.2 of Ref. 31, but it is not clear how those can be actually computed except when the assumed process (for example, the Chi-square model) allows sums over all possible exchanges to be represented as explicit closed-form expressions (Appendix: Theorems 1 and 2 in Ref. 30). Even with assumptions that make it mathematically tractable as in Ref. 30, there is no analysis presented for more than 10 markers. It is not clear if their method of likelihood maximization (downhill simplex method) would scale for hundreds of markers without running into some kind of computational bottleneck. Though these models are quite helpful in fitting data, we feel that the assumptions underlying their model of the crossover process are hard to verify as exchanges cannot be observed directly enough to provide any direct validation. Note that a crossover process is used to connect the observed gametes to the bivalent configurations. Nonetheless, despite their aforementioned limitations, the methods presented by Zhao and Speed do provide insights into recombination.

In this paper, we do not make any assumptions regarding the underlying crossover process, if any, that gives rise to the observed gametes. We consider a probabilistic framework for all possible bivalent configurations along the chromosome using the NCI model, which assumes that chromatid exchange between non-sister chromatids is equally likely. The term *bivalent configuration* describes how chromatids are joined with their non-sister chromatids along the chromosome. Figure 2 depicts possible bivalent configurations in a tetrad for two intervals. The following sections detail the proposed model formulation and show how connections to key concepts such as, recombination fraction, and likelihood based marker order, can be made using the proposed model.

4. A Mathematical Formulation for Bivalent Configurations

The primary emphasis of this paper is to design and analyze the proposed recursive linking algorithm. In order to make the algorithm more readable, the theory underlying the algorithm, which is under submission as a separate work, is presented in a concise manner. For detailed proofs of the theorems, interested readers are referred elsewhere (unpublished).

Before describing the mathematical model in detail, we introduce the following glossary of mathematical terms and symbols

- $c_i =$ probability of a non-sister chromatid exchange for the ith genetic interval
- S_i = sample space for the ith genetic interval
- S^l = sample space for all the genetic intervals
- ϕ_k = kth unique chromatid exchange in S^l
- f_k = kth genotype
- n_j = cell frequency for the jth observed genotype
- R_k = tetrad obtained via eq. (5) for exchange ϕ_k
- p_j = cell probability for the jth observed genotype
- $N = d_g$ = total number of distinct genotypes (or cells).
- $n = \sum_{j=1}^{d_g} n_j$ = total number of genotypes observed.

We model the genotypes $1\,1, 0\,0, 1\,0$ and $0\,1$ (where 1 and 0 refer to paternal and maternal alleles, respectively) of a genetic interval (say, S_i), as a consequence of two simultaneous independent and identical chromatid exchange events (say, S). Note that this double chromatid exchange is modeled as a discrete event as opposed to an analogous continuous process. The four possible chromatid exchanges between non-sister chromatids are pictorially represented in Fig. 3. There are 5 possible ways of performing a chromatid exchange (including a non-exchange as one of these possibilities) between non-sister chromatids. Note that our model accounts for all possible bivalent configurations (a bivalent configuration is defined as the chromatid exchanges on four strands of a tetrad) that may result due to any number of exchanges that may occur along the chromosome. It is important to distinguish that our model does not represent crossovers per se, but rather a pair of abstract discrete events that is capable of mapping the bivalent configurations, which could arise from any number of physical exchanges (chiasma). Let c_i denote the probability of a chromatid exchange in S between any two non-sister chromatids in the ith genetic interval S_i at meiosis. Since under the NCI assumption, all chromatid exchanges are equally likely, we get

$$\begin{aligned} S &= \{0, 1, 2, 3, 4\}, \\ P(i) &= \frac{c}{4}; \quad i = 1, \ldots, 4; \ \ i \in S, \\ P(0) &= 1 - c; \quad 0 \in S. \end{aligned} \tag{1}$$

The element 0 in S indicates the absence of an exchange event. Elements 1, 2, 3, and 4 indicate that non-sister chromatids $(1, 3), (2, 3), (2, 4)$, and $(4, 1)$ took part in the exchange process respectively. The set S_i, defined by the Cartesian product $S \times S$, enumerates all possible bivalent configurations for the ith genetic interval.

5. Multi Locus Genetic Likelihood for a Specified Order of Genetic Markers

The probability distribution of S_i denoting bivalent configurations between locus A_i and $A_{(i+1)}(i = 1, \dots, l-1)$, where $l =$ total number of loci being studied, can be expressed in the following equation using Eq. (1).

$$P(\{i,j\}) = \frac{c_i^2}{16} I_{\{i\neq 0; j\neq 0\}} + \frac{c_i(1-c_i)}{4}\{I_{\{i=0;j\neq 0\}} + I_{\{i\neq 0;j=0\}}\} + (1-c_i)^2 I_{\{i=j=0\}}, \tag{2}$$

where $\{i,j\} \in S_i$.

Let ϕ_k denote a unique chromatid exchange on S^l as described below

$$\phi_k = i_1 \times i_2 \times \cdots \times i_{l-1} \tag{3}$$

where

$$k = i_1 \cdot i_2 \cdot i_3 \cdots i_{l-1}; \quad i_j \in S_i; \quad \phi_k \in S^l = \prod_{i=1}^{l-1} S_i$$

From this point on, for the sake of brevity, we may abbreviate term *chromatid exchanges* to simply *exchanges* when referring to ϕ_k.

Let f_k denote a multi-locus genotype with l loci

$$f_k = i_1 \times i_2 \times \cdots \times i_{l-1} \times i_l$$

where

$$k = i_1 \cdot i_2 \cdot i_3 \cdots i_l; \quad i_j = 0, 1; \quad \forall j = 1, \cdots, l$$

The indices $i_j = 1$ and $i_j = 0$ indicate the paternal and maternal alleles, respectively. The progeny are obtained by exchanges between homogeneous parents. The observed data set can be represented as:

$$\mathcal{D} = \left\{n_j;\ \forall j = 1, \dots, 2^l\right\}$$

where n_j is the observed frequency of f_j.

5.1. *Recombination fraction r_i*

There are a variety of ways that various researchers have connected the recombination process to the underlying exchange process. For example, Mather[15] connects the recombination fraction to the number of exchanges occurring on the chromosome. Others, such as Haldane,[9] relate the recombination fraction to a physical distance. More recently Zhao *et al.*[30] relate recombination to an underlying exchange process. In the proposed model, the probability of recombination (or recombination fraction) can be calculated simply from the probability distribution of the bivalent configurations in set S_i.

The recombination fraction r_i is defined as follows

$$\begin{aligned} r_i &= P \text{ (at least one meiotic product is recombinant in } S_i) \\ &= c_i\left(1 - \frac{c_i}{2}\right) \\ &= \phi(c_i). \end{aligned} \tag{4}$$

Also

$$\max_{c_i \in [0,1]} r = \max_{c_i \in [0,1]} \phi(c_i) = \frac{1}{2}, \quad \forall\, i = 1, \ldots, l-1,$$

which is the theoretical maximum of the recombination fraction from the theory of Mendelian genetics.

5.2. *Probability distribution on S^l*

Let us define the following functions

$$\begin{aligned} f^0(a) &= (a_1, a_2, a_3, a_4)' \\ f^1(a) &= (a_3, a_2, a_1, a_4)' \\ f^2(a) &= (a_1, a_3, a_2, a_4)' \\ f^3(a) &= (a_1, a_4, a_3, a_2)' \\ f^4(a) &= (a_4, a_2, a_3, a_1)', \end{aligned} \tag{5}$$

where

$$\begin{aligned} a &= (a_1, a_2, a_3, a_4)' \\ a_i &= 0, 1 \quad \forall i \end{aligned}$$

Note that Eq. (5) encodes the elements of set S [see Eq. (1)] as mathematical functions. For example, the first function $f^0(a)$ encodes element 0, showing no chromatid exchange whereas function $f^4(a)$ encodes element 4, indicating that the first strand and the fourth strand have had an exchange.

The function $f_{ij}(a) = f_j(f_i(a))$ corresponds to the events in S_i accounting for all possible bivalent configurations. For a particular chromatid exchange ϕ_k we can generate a model tetrad at meiosis using the function f_{ij}. The matrix R_k of size $4 \times l$ defines the simulated tetrad as follows[24]:

$$R_k = \left(R_0 R_1 \cdots R_{(l-1)}\right) \tag{6}$$

where

$$R_0 = (1100)'; \quad R_i = f_{jk}(R_{i-1}), \quad \forall\, i = 1, \ldots, l-1$$

and the i^{th} genetic interval S_i contains the observed chromatid exchange $\{j, k\}$. In other words, R_k has 4 rows which correspond to the 4 gametes in a tetrad during meiosis if the chromatid exchange ϕ_k had occurred according to our model.

The conditional distribution of f_i for a given ϕ_k is

$$P(f_i|\phi_k) = \frac{1}{4}\sum_{j=1}^{4} I_{f_i \in R_{k(j,.)}} \tag{7}$$

where $R_k(j,.)$ is the j^{th} row of R_k.

The marginal density of a single spore f_i is given by

$$\begin{aligned} P(f_i) &= \sum_k P(f_i|\phi_k) \times P_k \\ &= C \times P \end{aligned} \tag{8}$$

where C is the conditional probability matrix given by

$$\left.\begin{aligned} C &= ((c_{ki})) \\ c_{ki} &= P(f_i|\phi_k) \quad \text{[from equation (7)]} \end{aligned}\right\} \tag{9}$$

and P is given by

$$\begin{aligned} P &= (P_k; \quad \forall k)' \\ P_k &= P(\phi_k) \\ &= \prod_{j=1}^{l-1} P(I_j = i_{k,j}) \end{aligned} \tag{10}$$

where $i_{k,j} \in S_j$ and the probability distribution $P(I_j = i_{k,j})$ is as defined in Eq. (2).

Let $\Theta = (c_1, c_2, \ldots, c_{l-1})'$ denote the unknown parameter vector in the model. The log-likelihood of $\mathbf{f} = (f_i, i = 1, \ldots, 2^{l-1})'$, viewed as a function of Θ, is given by

$$l(\Theta|D) = \sum_{i=1}^{N} n_i \log \left[\sum_k \left[\frac{1}{4} \sum_{j=1}^{4} I_{\{f_i \in R_k(j,.)\}} \prod_{j=1}^{l-1} P_j(I_{k,j} = i_{k,j}) \right] \right] \tag{11}$$

Note that the log-likelihood in Eq. (11) is distinct from the one in Zhao *et al.*[30] Zhao and Speed have formulated a log-likelihood function similar to the one in Eq. (11) for ordered tetrads in the Appendix of Ref. 32. However, the log-likelihood in Eq. (11) does not hypothesize an exchange process. Both, Lander and Green[12] and Zhao and Speed[32] have used the EM algorithm in the context of genetic map reconstruction.

The following two theorems maximize the log-likelihood in Eq. (11) using a set of recurrence relations obtained via the Expectation–Maximization (EM) algorithm.[6] The proofs are not given in the interest of brevity, but the development of Eq. (12) is described elsewhere.

Theorem 1. *The EM-iterative equations are given below.*

$$\Theta^{(h+1)} = \left(c_m^{(h+1)}, \forall m = 1, \ldots, l-1\right)'$$

where

$$c_m^{(h+1)} = \left(\frac{2N_{2,m} + N_{1,m}}{2N_m}\right)^{(h)} \tag{12}$$

where

$$\begin{aligned}
N_m &= \sum_k n_k^{(h)} = N_{0,m} + N_{1,m} + N_{2,m}\\
N_{0,m} &= \sum_{k|i_{k,m}=(0,0)} n_k^{(h)}\\
N_{1,m} &= \sum_{\substack{k|i_{k,m}=(i_1,i_2)\\ i_1=0 \text{ (Strict) OR } i_2=0}} n_k^{(h)}\\
N_{2,m} &= \sum_{\substack{k|i_{k,m}=(i_1,i_2)\\ i_1\neq 0 \text{ AND } i_2\neq 0}} n_k^{(h)}\\
n_k^{(h)} &= \sum_j n_j \pi_{k|j}(\Theta^{(h)})\\
\pi_{k|j}^{(h)} &= P(x_{kj} = 1|f_j) = \frac{\pi_{j|k}^{(h)} \times \pi_k^{(h)}}{p_j^{(h)}}\\
p_j^{(h)} &= \sum_k \pi_{j|k}^{(h)} \times \pi_k^{(h)}\\
\pi_{j|k}^{(h)} &= c_{ki} \textit{ in Eq. } (9) \textit{ for the } h^{th} \textit{ iteration}\\
\pi_k^{(h)} &= P_k \textit{ in Eq. } (10) \textit{ for the } h^{th} \textit{ iteration}
\end{aligned}$$

Note that, $i_{k,m}$ *denotes an event in* S_m *for the exchange* ϕ_k.

The following theorem provides an initialization of c for the recurrence relations in Eq. (12).

Theorem 2. *Let* $\mathbf{f} = (f_1, f_2, f_3, f_4)'$ *be the observed frequency vector corresponding to all possible meiotic products for parental genes M and O for two markers. The genotype vector for* $\mathbf{f}$ *is (MM MO OM OO)'. The maximum likelihood estimator*[22] *of the exchange probability c under the model represented by Eq.* (1) *is unique and* *is given as follows*:

1. *If* $f_1 + f_4 < f_2 + f_3$ *then* $c_{\text{mle}} = 1$
2. *If* $f_1 + f_4 \geq f_2 + f_3$ *then* c_{mle} *is given by the unique solution* (*in the interval* $[0, 1]$) *of the following equation*:

$$f(c) = c^2 - 2c + D = 0 \tag{13}$$

where

$$D = \frac{2(f_2 + f_3)}{N}; \quad N = \sum_{i=1}^{4} f_i$$

This theorem is used to obtain the starting values of c_m *for the EM-iterative equations in Theorem* 1.

6. The Straightforward Algorithm

6.1. *The RFLP data*

Some of the RFLP data for chromosome-I of *N. crassa*[18] are shown in Table 1. In the data, at each locus, the symbols "M" and "O" denote genes of the parents which have been encoded in the pseudocode description of the algorithms to follow as 1 and 0, respectively. A dash (–) indicates that the scoring was not done or was equivocal and thus denotes a missing observation. Note that in the pseudocode description of the algorithms to follow, any value other than a 0 or 1 indicates a missing value.

6.2. *Treating the missing scores*

For a particular order of genetic markers (for e.g. the rows of the Table 1) the spores (for e.g. the columns of the Table 1) are searched to see if they contain any missing score. The spores with missing scores are substituted in the following manner. If a missing genotype at any particular locus is surrounded by the same type (for e.g. genotypes of the pattern M–M or O–O), the missing genotype is substituted by the surrounding genotype. On the other hand, if the missing genotype is surrounded by different genotypes (for e.g. genotypes of the pattern M–O or O–M), a genotype between O and M is chosen with equal probability and substituted for the missing value. We do not attempt to address the issue of missing observations

Table 1. A partial view of the RFLP data.

	A	A	B	B	C	C	D	D
	1	4	6	7	1	4	5	7
AP10.1, AP10.4	M	O	M	O	M	O	M	O
AP8d.4	M	—	M	O	M	O	M	O
AP38a.1, R237	O	M	M	O	M	O	O	O

in this paper beyond this simple adjustment and spores with any other pattern of missing values (such as successive missing genotypes, though very rare) are simply removed from the study. The pseudocode below describes the algorithm used to achieve this. We have used Java syntax[10] throughout this paper in illustrating the various algorithms.

Algorithm: Treating Missing Values

```
temp1=new int[sporeNumber];
for int index=0;index<sporeNumber;index++ do
  count1+=checkMissing(getCol(data,index+1),order,missingValue);
  if checkMissing(getCol(data,index+1),order,missingValue) == 1 then
    temp1[index]=0; // missing Value is present
  else
    temp1[index]=1;
  end if
end for
{Treating for Possible Missing values}
for int index2=0;index2<sporeNumber;index2++ do
  for int index1=0;index1<geneNumber-1;index1++ do
    if !(data[index1][index2]==0 OR data[index1][index2]==1) AND index1>0
    then
      if data[index1-1][index2]==0 AND data[index1+1][index2]==0 then
        data[index1][index2]=0;
      end if
      if data[index1-1][index2]==1 AND data[index1+1][index2]==1 then
        data[index1][index2]=1;
      end if
      if (data[index1-1][index2]==1 AND data[index1+1][index2]==0)
          OR (data[index1-1][index2]==0 AND data[index1+1][index2]==1)
      then
        if Math.random()<0.5 then
          data[index1][index2]=0;
        else
          data[index1][index2]=1;
        end if
      end if
    end if
  end for
end for
{Spores with Persistent Missing values are Removed:}
count2=0;
int[][] trimdData1=new int[loci][sporeNumber-count1];
```

```
for int index1=0;index1<sporeNumber;index1++ do
  if temp1[index1]==1 then
    for int index2=0;index2<loci;index2++ do
      trimd_data1[index2][count2]=data[order[index2]-1][index1]
    end for
    count2++;
  end if
end for
{trimdData1 contains usable genotypes for the order specified}
```

6.3. *Identifying the distinct genotypes and computing their count*

For l markers there are 2^l possible genotypes and as many categories for a multinomial distribution[19] if each genotype is considered a category. Obviously, the categories are mutually exclusive and exhaustive, and the trials are independent if the spores are assumed to have come from different diploid parents. We identify the distinct genotypes and count their number using the following algorithm.

Algorithm: Counting Distinct Genotypes

```
int[][] trimdData2;
int[] freq;
searchList=new int[trimdData1[0].length];
exhaustList=new int[trimdData1[0].length];
for int index=0;index<trimdData1[0].length;index++ do
  searchList[index]=1
  exhaustList[index]=1
end for
compareTo=1;
count3=0;
count4=0;
while counter5==1 do
  for int index1=0;index1<loci;index1++ do
    for int index2=0;index2<searchList.length;index2++ do
      if searchList[index2]==1 then
        if trimdData1[index1][compareTo-1]!= trimdData1[index1][index2]
        then
          searchList[index2]=0;
        end if
      end if
    end for
  end for
  for int index3=0;index3<searchList.length;index3++ do
    if searchList[index3]==1 then
```

```
            exhaustList[index3]=0;
            count3++;
        end if
        searchList[index3]=exhaustList[index3];
    end for
    count4++; //This is counting the distinct loci patterns
    if count4==1 then
        trimdData2=addCol(getCol(trimdData1,compareTo));
        {addCol(.) returns a column in the type of $trimdData2$}
        freq=addElement(count3);
        {addElement(.) returns $count3$ as a vector (in the type of $freq$)}
    else
        trimdData2=addCol(trimdData2,getCol(trimdData1,compareTo));
        {$trimdData2$ is updated with an additional column}
        freq=addElement(freq,count3);
        {$freq$ is updated with an additional element $count3$}
    end if
    count3=0;
    check=0;
    counter5=0;
    for check=0;check<searchList.length;check++ do
        if searchList[check]==1 then
            counter5=1;
            break;
        end if
    end for
    compareTo=check+1; // $compareTo$ denotes position rather than array index
end while

{$trimdData2$ contains the distinct genotypes in the order of the markers}
{$freq$ contains the frequency count of the distinct genotypes in $trimdData2$}
```

6.4. *Initial probability estimates*

Theorem 2 is used to compute the initial estimates of the exchange probabilities for each interval in the given order of markers. These probability estimates are interval based and do not take into account the interdependence of the markers in the order, but rather serve as good initial estimates to implement the EM algorithm detailed in Sec. 1. The algorithm for computing the initial estimates of c is given below.

```
Algorithm: Computing Initial Probability Estimates of $c$
for int $i$=0;$i$<loci-1;$i$++ do
  int count=0;
  double d,c;
  for int index=0;index<sporeNumber;index++ do
    if (trimdData1[i][index]==1 AND trimdData1[i+1][index]==0) OR
    (trimdData1[i][index]==0 AND trimdData1[i+1][index]==1) then
      count++;
    end if
  end for
  d=(double)2*count/obs;
  if d > 1.0 then
    cProbOld[index]=1.0;
  else
    cProbOld[index]=1-sqrt(1-d);
  end if
end for
{cProbOld holds the initial probability estimates}
```

6.5. *Computing the likelihood and implementing the EM algorithm*

To compute the likelihood described by Eq. (11), it is evident that we need to process 25^{l-1} exchanges. The algorithm has to be extremely efficient to handle a large number of genetic markers to allow for several computations of the log-likelihood for different orderings of the markers (see Sec. 11).

Given the huge computational complexity of this task, it would not be possible to store either the matrices C or R on account of their prohibitive size. Note that we must compute the likelihood incrementally (and hence on the fly), and the nested FOR loops in algorithm *Loop(0)* must be dynamically created (as l is a variable). Hence a recursive function needs to be designed to accomplish this task. Furthermore, to implement the EM algorithm to iterate the value of the probability vector c described in the series of Eq. (12) one must again process 25^{l-1} exchanges. Consider the following two equations from the series of Eq. (12).

$$\pi_{k|j}^{(h)} = P(x_{kj} = 1|f_j) = \frac{\pi_{j|k}^{(h)} \times \pi_k^{(h)}}{p_j^{(h)}}$$

$$p_j^{(h)} = \sum_k \pi_{j|k}^{(h)} \times \pi_k^{(h)}$$

The computation of $\pi_{k|j}^{(h)}$ requires the knowledge of $p_j^{(h)}$ — whose value is based on the computation of 25^{l-1} exchanges. This makes the simultaneous implementation of the computation of the likelihood and running the EM algorithm (to update the exchange probability vector c) a non-trivial problem. This problem has been bypassed by implementing both, the likelihood computation and the EM algorithm in a recursive loop, which avoids computing the cell probabilities $p_j^{(h)}$ until all the exchanges have been processed and readjusts them in the end as they are computed during the likelihood computation. The following algorithm illustrates this point.

Algorithm: Computing the Likelihood and Implementing the EM Algorithm

```
for iteration=0;iteration<iterationLimit;iteration++ do
  probFOld=new double[freq.length];
  cProbNew=new double[loci-1];
  counter=new int[loci-2];
  pCount=new double[3][loci-1][freq.length];
  Loop(0);
  Loop(0) creates both $pCount$(ie $cProbNew$) and $probFOld$ at the same time
  /* Reporting the likelihood */
  logLikelihood=0.0;
  for int index=0;index<freq.length;index++ do
    logLikelihood+=freq[index]*Math.log(probFOld[index]);
  end for
  System.out.println(”The likelihood is ”+ logLikelihood +” at iteration ” + iteration);
  /* Getting the posterior counts */
  for int index1=0;index1<3;index1++ do
    for int index2=0;index2<loci-1;index2++ do
      for int index3=0;index3<freq.length;index3++ do
        pCount[index1][index2][index3]=pCount[index1][index2][index3]/
          probFOld[index3]
      end for
      postCount[index1][index2]=Sum(pCount[index1][index2])
    end for
  end for
  /* Getting the c probs */
  for int index1=0;index1<loci-1;index1++ do
    cProbNew[index1]=$\frac{(\text{postCount[1]}[index1]+2*\text{postCount[2][index1]})}{(2.0*\text{Sum(getCol(postCount,index1+1)))}}$
  end for
  if Convergence(cProbOld,cProbNew,0.0000001) ==1 then
    break;
    {$Convergence(.)$ checks if the absolute difference in all dimensions is $< 0.0000001$}
  end if
  cProbOld=setEqual(cProbNew);
  {$setEqual(.)$ creates a copy of the object}
end for
```

Algorithm: Loop(0)

```
void Loop2(int index )
if index < loci-2 then
  for counter[index]=0;counter[index]<24;counter[index]++ do
    Loop2(index+1);
  end for
end if
if index == loci-2 then
  for int check1=0;check1<24;check1++ do
    for int check2=0;check2<loci-2;check2++ do
      k[check2]=counter[check2];
    end for
    k[loci-2]=check1;
    r=getRMatrix(k);
    if kIsWorthy(r,trimdData2)==1 then
      prob=kProb(cProbOld,k);
      double[] sum=new double[freq.length];
      for int index1=0;index1<freq.length;index1++ do
        sum[index1]=countMatch(r,getCol(trimdData2,index1+1))
        *prob*freq[index1]/(4.0*totalObs)
        {$countMatch(.)$ counts the number of tetrads (out of 4) in $r$ that match
        with the current distinct genotype}
        probFOld[index1]+=countMatch(r,getCol(trimdData2,index1+1))/
        4.0*prob
      end for
      for int index2=0;index2<loci-1;index2++ do
        pCount[CellSpecial(k[index2])][index2]+=sum;
        { The above addition is a vector addition}
      end for
    end if
  end for
end if
```

In the pseudocode above, k denotes a particular exchange ϕ_k as defined in Eq. (3). The function *getRMatrix* implements Eq. (6) to create the R_k matrix corresponding to the exchange ϕ_k. The function *kProb* computes the marginal probability P_k due to exchange ϕ_k as defined in Eq. (10). The matrix R_k and the probability P_k in turn create matrices C and P progressively during the course of the recursive loop to compute the marginal probability of each observed distinct genotype as defined in Eq. (8). These marginal probabilities along with the counts for the distinct genotypes are used to compute the log-likelihood in Eq. (11). A particular exchange ϕ_k does not enter into the computation of the likelihood so long as it does not have positive probability for at least one distinct genotype. The elimination of such exchanges is achieved with the function $kIsWorthy$, which implies that at least some amount of computation cannot be avoided for each exchange. In the recursive algorithm that we propose in this paper, this feature is handled more efficiently where a large number of exchanges are eliminated by performing checks on a few. In the pseudocode the vector *sum* computes the conditional probabilities across all distinct genotypes. The conditional probability is computed with the help of the function *countMatch* that implements Eq. (7), by counting the number of strands (out of 4) in R_k that match with the observed genotype. The vector $freq$ has the observed count corresponding to the distinct genotypes (d_g), *totalObs* is the total sample size, and *probFOld* stores the marginal probability of each distinct genotype using Eq. (8). The array $pCount$ implements the EM algorithm via Eq. (12) by re-categorizing the vector *sum* based on the exchange values along the chromosome. Note that the denominator of $\pi_{k|j}$, i.e. p_j, the marginal probability due to the j^{th} distinct genotype, is omitted from its $\pi_{k|j}$ computation as that requires going through all the exchanges, and is currently being progressively computed by *probFOld*. To compute $n_k^{(h)}$ in Eq. (12) we need to sum up the inverse probabilities $\pi_{k|j}$ across the distinct genotypes but, as their marginal probabilities are not computed yet, it is not possible to do so. We work around this problem by adding another dimension along the number of distinct genotypes to the structure $pCount$. The first dimension of $pCount$ is of magnitude 3 to account for $N_{0,m}, N_{1,m}$, and $N_{2,m}$ in Eq. (12), whereas the second dimension runs along m, accounting for the $(l-1)$ genetic intervals. Once all the exchanges are processed and the marginal probabilities computed, the elements in the third dimension are divided by their corresponding marginal probabilities and then added up across the dimension. This gives us the two dimensional structure *postCount* containing values of $N_{0,m}, N_{1,m}$, and $N_{2,m}$ for all the genetic intervals. The new value of c_i for each genetic interval is then computed using Eq. (12), and the process iterates until convergence is reached. Despite being a recursive algorithm (exchanges are generated recursively), it suffers from the computational bottleneck of having to process a large number of exchanges i.e. 25^{l-1} for l loci. This problem is overcome using the proposed recursive linking algorithm.

7. The Proposed Recursive Linking Algorithm

Let the entire order of genetic markers be decomposed into segments of equal width (h), such that all the intervals are covered. Thus, for l genetic markers the number of segments s is given by the following equation

$$s = \frac{(l-1)}{(h-1)} \tag{14}$$

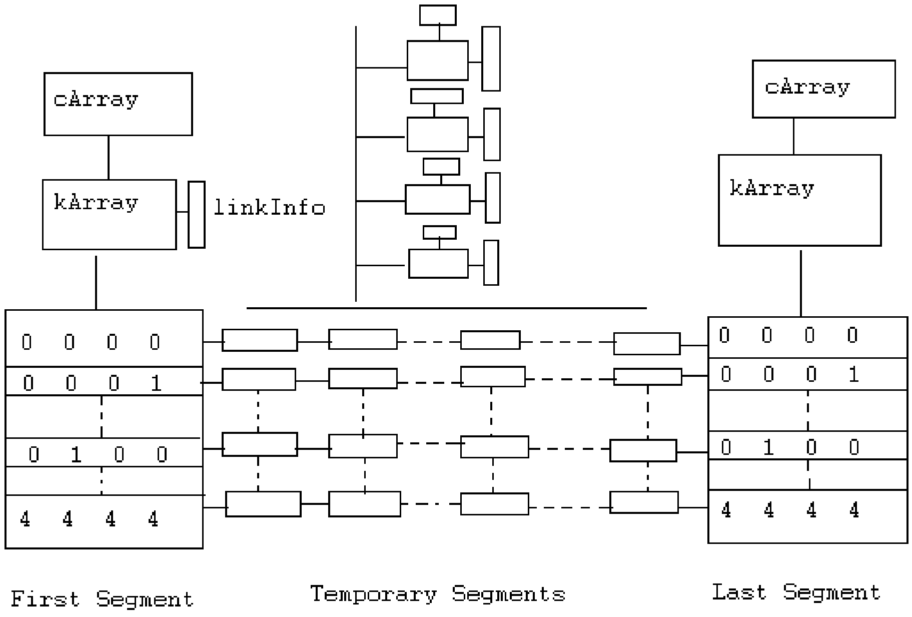

Fig. 4. A visual depiction of the data structures.

7.1. *Overview of the recursive linking algorithm*

The first segment has an associated array called *kArrayFirst* that contains all the exchanges for this segment. The array *linkInfoFirst* stores the last row generated by the matrix R for each exchange of the segment. The array *cArrayFirst* checks for each exchange and for each observed genotype of the first segment, which strands of the simulated tetrad [based on the model described in Eq. (1)], obtained via matrix R, match with the genotype. The matching status forms the last dimension of the array with length 4 and consists of symbols 1 and 0 indicating a match(1) or mismatch(0) respectively. For example, a matching status 1 0 0 1 for the first distinct genotype corresponding to the exchange pattern 0 0 2 3 4 in the first segment level indicates that among the 4 tetrads in meiosis generated by the exchange pattern 0 0 2 3 4 in the first segment, the observed genotype in question was found only on the first and the fourth tetrad. When we use this information over a combined segment formed using two segments, only a match at the same tetrad position will ensure a match for the combined segment. Note that R depends on exchange values on all intervals of the segment and its columns are sequentially dependent on each other with a lag of one. Similarly, we have structures *kArrayLast* and *cArrayLast* for the last segment. Each exchange in the first segment branches out to $25^{(h-1)}$ exchanges in the following segment and creates $25^{2(h-1)}$ combined exchanges. This continues till the last segment is accounted for. In order to move along the segments following the model described in equation (1), we need to know the last row (a tetrad pattern for the last locus of the segment) generated by the matrix R for the linked exchange of the previous segment corresponding to each combined exchange of the two segments. The following lemma states that only certain patterns are possible at the terminal locus of the adjoining segments. Hence, we create arrays similar to *kArray*, *linkInfo*, and *cArray* for all the following segments between the first segment and the last segment and call them *kArrayTemp*[], *linkInfoTemp*[], and *cArrayTemp*[], respectively, where the array index denotes the segment numbers.

Lemma 1. *Under the model described by Eq.* (1) *at any specified locus only one of the tetrad patterns* $1100, 0110, 1010, 1001, 0101$ *and* 0011 *could occur.*

Proof. Recall that a tetrad pattern is an arrangement of the alleles of a particular gene in the simulated tetrad generated by a given exchange under the model described by Eq. (1), where 1 and 0 indicate the parental alleles for the particular locus. Let $P_1, P_2, \ldots, P_6$ denote the tetrad patterns $1100, 0110, 1010, 1001, 0101$, and 0011, respectively. The tetrad pattern at locus i for an exchange value s_i in the i^{th} genetic interval S_i is given by

$$T_{s_i} = f_k(f_j(T_{s_{i-1}}))$$

where $s_i = \{j, k\} \in S_i$ in Eq. (2) and $f_k(.)$ and $f_j(.)$ are obtained from Eq. (5). Note that $T_{s_0} = P_1$. In order to generate the patterns beginning with pattern T_{s_0} along with their sample points, (Table 2) we can see that all the sample points with the source pattern $T_{s_0} = P_1$ correspond to the patterns within P_i $(i = 1, \cdots, 6)$. Next, we enumerate tetrad patterns beginning with source patterns P_i $(i = 1, 2, \ldots, 6)$. From Table 2 it is clearly seen that tetrad patterns cannot lie outside the set P_i $(i = 1, \ldots, 6)$. Hence the lemma is proved. $\square$

7.2. *Counting active exchanges*

Analogous to the function *kIsWorthy* in the straightforward algorithm we implement the concept of counting active exchanges for each segment. We call an exchange of a particular segment active if it has positive probability for at least one distinct genotype. Note that active exchanges will be different across segments as an active exchange depends on both, the observed genotypes (that vary across segments) and the tetrad pattern used in the generation of the matrix R. It is important to emphasize the substantial computational savings achieved by the elimination of the exchanges on a segment-wise basis in the proposed algorithm compared to the straightforward algorithm which eliminates exchanges one at a time. Elimination of a single exchange in the first segment has the effect of elimination of 25^{l-2} exchanges in the straightforward algorithm. In general, elimination of a single exchange in the ith segment has the effect of eliminating 25^{l-i-1} exchanges in the straightforward algorithm. The following algorithm shows how active exchanges are computed.

Table 2. Possible tetrad patterns

Source Pattern	Generated Patterns	Sample Points
P_1	P_1	(0,0), (1,1), (2,2), (3,3), (4,4)
	P_2	(0,1), (1,0), (1,2), (1,4)
	P_3	(0,2), (2,0), (2,1), (2,3)
	P_4	(0,3), (3,0), (3,2), (3,4)
	P_5	(0,4), (4,0), (4,1), (4,3)
	P_6	(1,3), (3,1), (2,4), (4,2)
P_2	P_1	(0,1), (1,0), (2,1), (4,1)
	P_2	(0,0), (0,2), (2,0), (0,4), (4,0), (1,1), (2,2), (3,3), (4,4), (4,2), (2,4)
	P_3	(1,2), (3,4)
	P_4	(1,3), (3,1)
	P_5	(1,4), (3,2)
	P_6	(0,3), (3,0), (2,3), (4,3)
P_3	P_1	(0,2), (2,0), (1,2), (3,2)
	P_2	(2,1), (2,3), (4,3)
	P_3	(0,0), (0,1), (1,0), (0,3), (3,0), (1,1), (2,2), (3,3), (4,4), (1,3), (3,1)
	P_4	(4,1)
	P_5	(4,2), (2,4)
	P_6	(0,4), (4,0), (1,4), (3,4)
P_4	P_1	(0,3), (3,0), (2,3), (4,3)
	P_2	(1,3), (3,1)
	P_3	(1,4), (3,2)
	P_4	(0,0), (0,2), (2,0), (0,4), (4,0), (1,1), (2,2), (3,3), (4,4), (4,2), (2,4)
	P_5	(1,2), (3,4)
	P_6	(0,1), (1,0), (2,1), (4,1)
P_5	P_1	(0,4), (4,0), (1,4), (3,4)
	P_2	(4,1)
	P_3	(4,2), (2,4)
	P_4	(2,1), (2,3), (4,3)
	P_5	(0,0), (0,1), (1,0), (0,3), (3,0), (1,1), (2,2), (3,3), (4,4), (1,3), (3,1)
	P_6	(0,2), (2,0), (1,2), (3,2)
P_6	P_1	(1,3), (3,1), (4,2), (2,4)
	P_2	(0,3), (3,0), (3,2), (3,4)
	P_3	(0,4), (4,0), (4,1), (4,3)
	P_4	(0,1), (1,0), (1,2), (1,4)
	P_5	(0,2), (2,0), (2,3), (2,1)
	P_6	(0,0), (1,1), (2,2), (3,3), (4,4)

Algorithm: Counting Active Exchanges

```
int startRowIndex,endRowIndex;
activeKFirst=0;
activeKLast = new int[6];
k=new int[height-1];
counter=new int[height-2];
startRowIndex=0;
endRowIndex=startRowIndex+height-1;
```

```
data1=GetData(startRowIndex,endRowIndex,0,(distinctGenotypes-1));
{Extracting genotype data for the first segment using data $trimdData2$}
startRowIndex=loci-height;
endRowIndex=loci-1;
data2=GetData(startRowIndex,endRowIndex,0,(distinctGenotypes-1));
{Extracting genotype data for the last segment using data $trimdData2$}
ActiveKLoopFirstAndLast(0,height);
```

Algorithm: ActiveKLoopFirstAndLast(0,height)

```
void ActiveKLoopFirstAndLast(int index,int height)
if index < height-2 then
  for counter[index]=0;counter[index]<24;counter[index]++ do
    ActiveKLoopLast(index+1,height);
  end for
end if
if index == height-2 then
  for int check1=0;check1<24;check1++ do
    for int check2=0;check2<height-2;check2++ do
      k[check2]=counter[check2];
    end for
    k[height-2]=check1;
    int[][]r;
    r=getRMatrix(k,firstCol);
    {$firstCol$ is 1100}
    {FOR THE FIRST SEGMENT}
    if kIsWorthy(r,data1)==1 then
      activeKFirst++;
    end if
    {FOR THE LAST SEGMENT}
    for int i=0;i<6;i++ do
      r=getRMatrix(k,tempVec[i]);
      {$getRMatrix(.)$ computes equation (6) for the $(i+1)^{th}$ tetrad pattern in
      Lemma 1}
      if kIsWorthy(r,data2)==1 then
        activeKLast[i]++;
      end if
    end for
  end for
end if
```

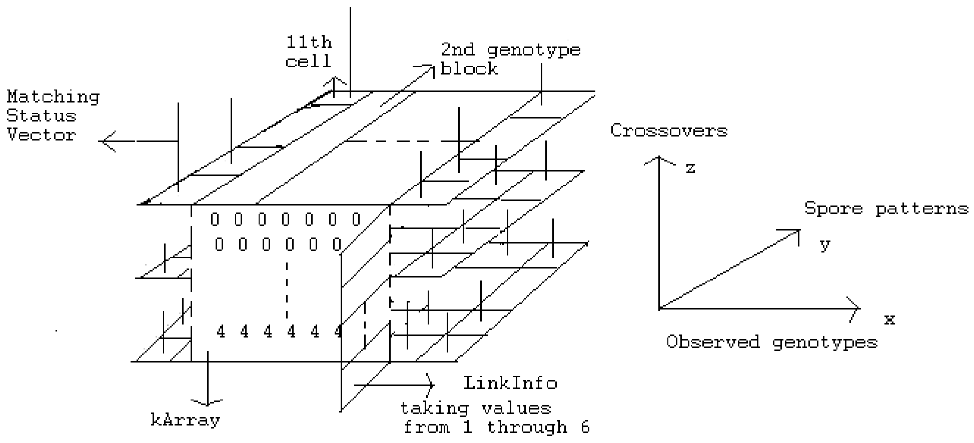

Fig. 5. A visual depiction of the data structures *kArray*, *LinkInfo*, and *cArray*.

7.3. *Computing kArray, LinkInfo, and cArray for the first and the last segment*

Figure 5 shows the data structures *kArray*, *linkInfo*, and *cArray* as they are linked to each other on a particular segment. This structure is true for both the first and the last segments. As described in Sec. 7.1, the data structure *linkInfo* keeps track of the tetrad pattern which is one of the 6 possibilities described in Lemma 1. Structure *cArray* helps in computing the matching status over the combined segments, which in turn helps to compute the conditional probability of observed genotypes given the progressive consideration of exchanges as more and more segments are linked. This process is best understood by the following algorithm.

Algorithm: Computing $kArray$**,** $LinkInfo$**, and** $cArray$

```
startRowIndex=0;
endRowIndex=height-1;
kCount1=0;
kCount = new int[6];
k=new int[height-1];
counter=new int[height-2];
kArrayFirst= new int[activeKFirst][height-1];
inkInfoFirst=new int[activeKFirst];
cArrayFirst=new int[activeKFirst][distinctGenotypes][4];
endRowIndex=height-1;
data1=GetData(startRowIndex,endRowIndex,0,(distinctGenotypes-1));
startRowIndex=loci-height;
endRowIndex=loci-1;
kArrayLast= new int[6][][];
```

```
cArrayLast= new int[6][][][];
for int i=0;i<6;i++ do
   kArrayLast[i]=new int[activeKLast[i]][height-1];
   cArrayLast[i]=new int[activeKLast[i]][distinctGenotypes][4];
end for
data2=GetData(startRowIndex,endRowIndex,0,(distinctGenotypes-1));
ComputeFirstAndLast(0,height);
```

Algorithm: ComputeFirstAndLast(0,height)

```
void LoopLast(int index,int height)
if index < height-2 then
   for counter[index]=0;counter[index]<24;counter[index]++ do
      LoopLast(index+1,height);
   end for
end if
if index == height-2 then
   for int check1=0;check1<24;check1++ do
      for int check2=0;check2<height-2;check2++ do
         k[check2]=counter[check2];
      end for
      k[height-2]=check1;
      int[][]r;
      {THE FOLLOWING CODE COMPUTES OBJECTS FOR THE FIRST
      SEGMENT}
      r=getRMatrix(k,firstCol);
      if kIsWorthy(r,data1)==1 then
         kArrayFirst[kCount1]=setEqual(k);
         linkInfoFirst[kCount1]=Map6ToInt(r[r.length-1]);
         {$Map6ToInt$(.) converts the tetrad patterns in Lemma 1 to an integer
         between 1 and 6}
         for int index1=0;index1<distinctGenotypes;index1++ do
            cArrayFirst[kCount1][index1]=MatchArray(r,getCol(data1,index1+1));
            {$MatchArray$(.) creates a matching status vector for the genotype and
            segment in question}
         end for
         kCount1++;
      end if
      {THE FOLLOWING CODE COMPUTES OBJECTS FOR THE LAST
      SEGMENT}
      for int i=0;i<6;i++ do
         r=getRMatrix(k,tempVec[i]); // r is segment-1 by 4
         if kIsWorthy(r,data2)==1 then
```

```
                kArrayLast[Map6ToInt(tempVec[i])][kCount[i]]=setEqual(k);
                for int index1=0;index1<distinctGenotypes;index1++ do
                    cArrayLast[Map6ToInt(tempVec[i])][kCount[i]][index1]
                    =MatchArray(r,getCol(data2,index1+1));
                end for
                kCount[i]++;
            end if
        end for
    end for
end if
```

7.4. *Computing tempSumIndex*

Although not absolutely necessary in the implementation of the recursive linking algorithm, we compute a variable called *tempSumIndex* for the last segment which helps to speed up the EM iterations. Consider a combined exchange for all the segments. To compute Eq. (7), i.e. to count the matches for the entire exchange we have to examine the matching status of all the segments and update them. To be considered a match for the entire segment at a particular position (out of 4 possible positions) one must have a match for all the segments at that position. Hence, when the matching status values of two segments are updated, the resulting matching status is 1 if and only if both the segments have a value 1 at that position and 0 otherwise. This updated matching status is termed a *spore* in this paper. The distinction between a spore and matching status is that while a matching status is the original status of the segment, the spore is the matching status obtained after updating the matching status of all the previous segments. The following lemma restricts the number of possible spore patterns.

Lemma 2. *Under model described by Eq.* (1), *for any exchange and any observed genotype the spore patterns* $0111, 1011, 1101, 1110$, *and* 1111 *are not possible.*

Proof. Recall that a spore pattern denotes the matching status of the tetrad of a diploid cell corresponding to an order of genetic markers. A 1 indicates a match, and 0 a mismatch of the observed genotype to the simulated tetrad for the exchange at hand, according to the model. Note that since each parent has a characteristic allele, it is not possible to have matches at 3 locations. Hence, spore patterns with 3 or more matches are not possible. Hence the lemma is proved. □

As a visual image of the interplay along different dimensions, we define a plane corresponding to each exchange. The blocks along the x dimension refer to distinct genotypes whereas its cells denote the sum of the updated matching status values for the 11 possible matching status values from the previous segment as specified by

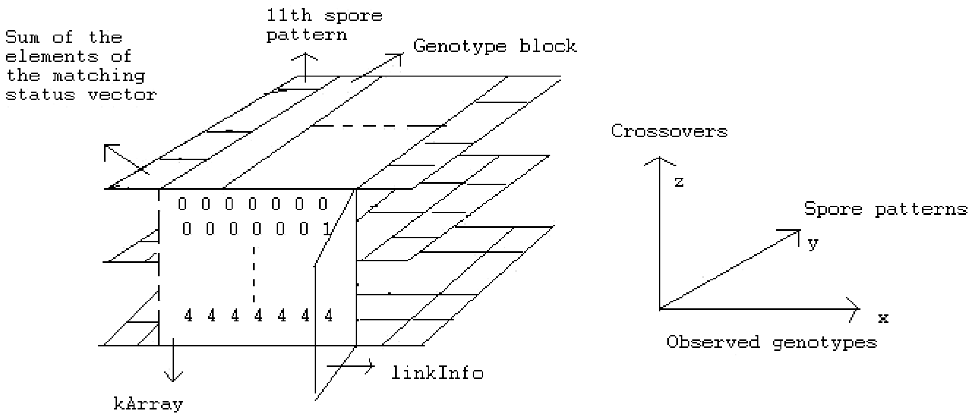

Fig. 6. A visual depiction of structures *kArray*, *LinkInfo*, and *cArray* of the last chromosome segment after computation of the sum of elements of the updated matching status vector.

Lemma 2. The structure *tempSumIndex* essentially computes Eq. (9) for the last segment except that the matching status value is computed for all possible tetrad patterns described in Lemma 1. Note that Lemmas 1 and 2 restrict the array size and ensure efficient memory usage. The following algorithm illustrates the computation of *tempSumIndex*

Algorithm: Computing *tempSumIndex*

```
int[][] spores=getSpores();
{getSpores(.) creates all the 11 possible spores (out of 16) using Lemma 2}
tempSumIndex = new int[6][][][];
for int i=0;i<6;i++ do
  tempSumIndex[i]=new int[distinctGenotypes][11][activeKLast[i]];
end for
for int index0=0;index0<6;index0++ do
  for int index1=0;index1<distinctGenotypes;index1++ do
    for int index2=0;index2<11;index2++ do
      for int index3=0;index3<activeKLast[index0];index3++ do
        tempSumIndex[index0][index1][index2][index3]
        =Sum(UpdateSpore(cArrayLast[index0][index3][index1],spores[index2]));
        {UpdateSpore(.) updates the matching status as described earlier}
        {Sum(.) adds up the elements of its argument}
      end for
    end for
  end for
end for
```

7.5. *Computing the recursive structures tempSum and tempPCount*

The structure *tempSumIndex* created for the last segment is used in conjunction with certain operations performed on each plane to compress the information for the last segment. Each exchange has a positive probability (since inactive exchanges have been omitted) defined in Eq. (10) which is same for all the cells in the attached plane. Cells in a plane refer to the conditional probability up to a constant (divisor 4) for all possible situations. If we multiply cells by the exchange probability of the plane and sum across all the planes we essentially implement Eq. (8) except that we do it for various matching status configurations in the previous segment. This allows us to implement Eq. (8) for the combined segment as if the combined segment were never decomposed into segments. Note that all these operations are performed for a specific value of θ. We need to implement Eq. (12) to update θ via the EM algorithm. This is done using the *pipes* shown in Fig. 7. A cell on a plane has as many pipes as there are genetic intervals and each pipe has 3 associated categories for computing $N_{0,m}$, $N_{1,m}$, and $N_{2,m}$ in Eq. (12). Note that as planes are added up (compressed) each exchange is categorized within its pipes according to its exchange values in the genetic intervals. This logic is similar to that underlying the implementation of *pCount* in the straightforward algorithm. The structures *tempSum* and *tempPCount* together constitute the recursive structure of the last segment. In the following pseudocode, structures *tempSum* and *tempPCount* implement Eqs. (8) and (12) respectively.

Algorithm: Computing the Recursive Objects *tempSum* **and** *tempPCount*

```
double[][][] FirstTempSum=new double[6][distinctGenotypes][11];
double[][][][][] FirstTempPCount=new double[6][distinctGenotypes][11][3][loci-1];
double[] lastCProb=getCProb(cProbOld,0,height-1)
{getCProb(.) extracts the current segment from the exchange probability cProbOld }
for int index0=0;index0<6;index0++ do
  for int index1=0;index1<distinctGenotypes;index1++ do
    for int index3=0;index3<11;index3++ do
      for int index2=0;index2<activeKLast[index0];index2++ do
        double tempDouble=kProb(lastCProb,kArrayLast[index0][index2]);
        {kProb(.) computes the exchange probability for the
        exchange value in kArrayLast[index0][index2]}
        double condProb=tempSumIndex[index0][index1][index3][index2]/4.0;
        FirstTempSum[index0][index1][index3]+=condProb*tempDouble;
        for int index=loci-height;index<loci-1;index++ do
          FirstTempPCount[index0][index1][index3]
          [CellSpecial(kArrayLast[index0][index2]
```

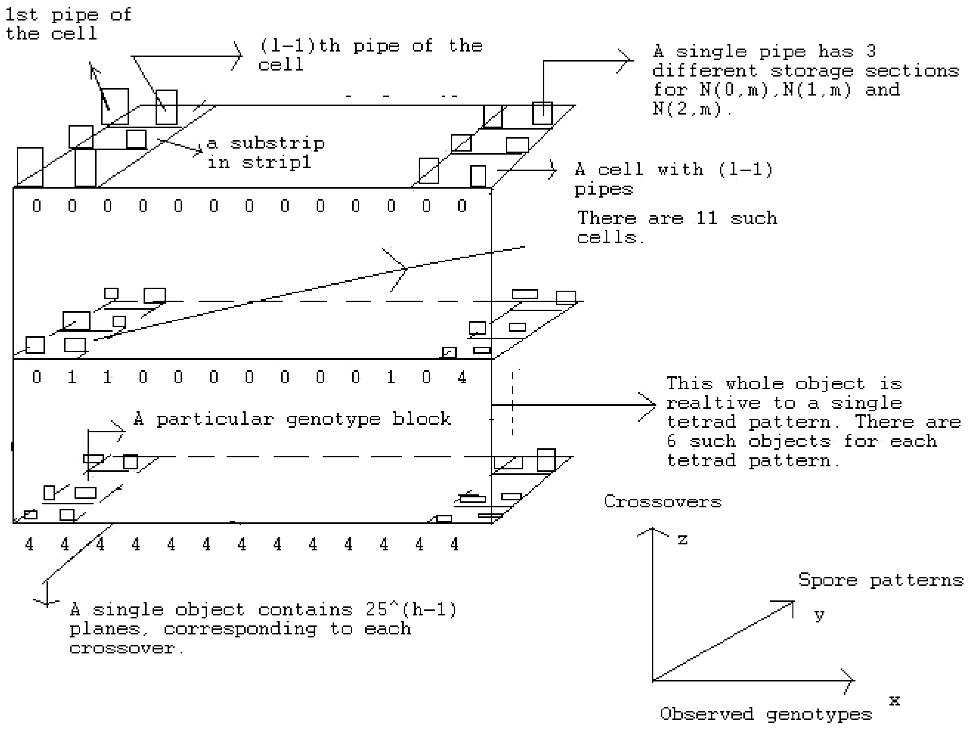
1st pipe of the cell
(l-1)th pipe of the cell
A single pipe has 3 different storage sections for N(0,m),N(1,m) and N(2,m).
a substrip in strip1
A cell with (l-1) pipes
There are 11 such cells.
0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 0 0 0 0 0 0 0 1 0 4
This whole object is realtive to a single tetrad pattern. There are 6 such objects for each tetrad pattern.
A particular genotype block
Crossovers
z
4 4 4 4 4 4 4 4 4 4 4 4 4
Spore patterns
y
A single object contains 25^(h-1) planes, corresponding to each crossover.
x
Observed genotypes

Fig. 7. A visual depiction of structures *tempSum* and *tempPCount* of the last chromosome segment.

```
                [index-loci+height])][index]+=condProb*tempDouble;
            end for
        end for
    end for
  end for
end for
```

7.6. *Traversing the segments by updating the recursive structure*

As we traverse the segments, for each segment we create the associated structures *kArray* (which is the same for all segments if equal width segments are assumed), *linkInfo* and *cArray*, by first computing the active exchanges and then computing the structures in a similar manner as described in Secs. 7.2 and 7.3. Given a recursive structure for the last segment (on the right in Fig. 8) the goal is to generate a recursive structure for the second to last segment using the first one. This process is outlined in Fig. 8. This structure has the property that $25^{(h-1)}$ exchanges have already been processed in a form such that equation (12) can be implemented and any spore generated from the previous segment can be handled.

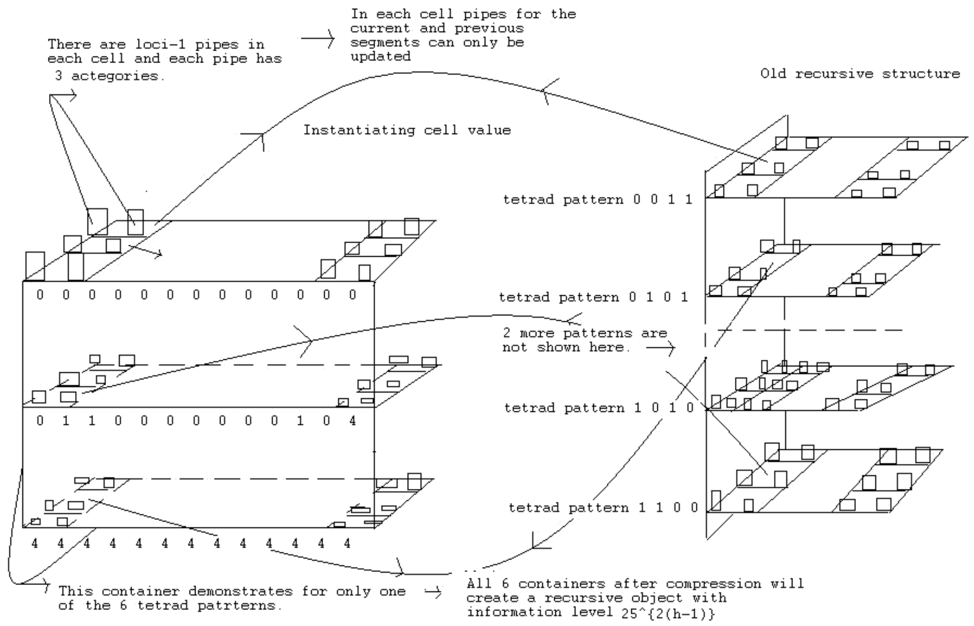
In each cell pipes for the current and previous segments can only be updated
There are loci-1 pipes in each cell and each pipe has 3 actegories.
Old recursive structure
Instantiating cell value
tetrad pattern 0 0 1 1
tetrad pattern 0 1 0 1
2 more patterns are not shown here.
tetrad pattern 1 0 1 0
tetrad pattern 1 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 0 0 0 0 0 0 0 1 0 4
4 4 4 4 4 4 4 4 4 4 4 4 4
This container demonstrates for only one of the 6 tetrad patrterns.
All 6 containers after compression will create a recursive object with information level 25^{2(h-1)}

Fig. 8. A visual depiction of the updating of the recursive structure.

Note that when an exchange from the previous segment (the second to last segment) is processed(allowing for all possible spore patterns of the previous segment), the corresponding spore is computed, its equivalent cell on the last segment is identified and $25^{(h-1)}$ combined exchanges of these two segments are processed. After all the exchanges from the previous segment are processed, the proposed algorithm generates a recursive structure that is exactly the same as that of the last segment without adding to the storage requirement. The recursive structure for the last but one segment now has $25^{2(h-1)}$ exchanges processed within it with provision for all possible spores from its preceding segment. This very feature shows how we can geometrically increase the information base (in terms of exchanges) of the "table look up" procedure and avoid traversing all the exchanges one at a time. One of the reasons this procedure works is because if we examine the combined exchanges of two segments, we see that exchange values for the left segment change only once for every $25^{(h-1)}$ combined exchanges. This allows us to delay the probability value updates when linking the segments. In the following few paragraphs we discuss in detail the updating procedure.

The structure on the left in Fig. 8, called a *container*, is essentially a recursive structure except that has not been yet instantiated. We choose a cell on the plane corresponding to an exchange in the *container*. Note that the cell that is visited depends on the matching status of the *container*'s preceding segment and the *genotype* of the cell. The matching status of the cell in the *container* is updated with the matching status of the genotype corresponding to the segment of the *container* and a cell for the updated matching status is identified in the recursive structure (on the right in Fig. 8), for that particular genotype.

The structure *tempSum* for the *container* is instantiated by multiplying the structure *tempSum* of the recursive structure with the exchange probability of the plane in *container*. Here, the term multiplication is used in a loose sense of multiplication. Note that a single multiplication operation processes 25^{h-1} exchanges at a time. We have $l-1$ pipes in each cell, and for the cell under consideration, the pipes are updated in the following manner. The pipes for the segments other than the segment of the *container* and following segments are left unfilled. The pipes for the *container*'s segment are filled for the categories (one of the 3) corresponding to the exchange values of each plane by multiplying the structure *tempSum* of the recursive structure with the plane probability (i.e., the exchange probability for the plane in question). For the pipes corresponding to the following segments (segments that have been processed already), all the pipes in the identified cell are copied and multiplied by the plane probability. This process is carried out for all the cells on all the planes of a *container*. The other 5 containers corresponding to the tetrad patterns in Lemma 1 are processed similarly. All of the planes for all the 6 containers are compressed as described earlier to generate a recursive structure for the segment corresponding to the *container*. Note that $\pi_{k|j}$ in Eq. (12) is computed only up to a constant value. The denominator p_j cannot be computed at the same time since that also involves traversing all exchange paths and is handled by the structure *tempSum* in the pseudocode. It is important to note that this creates no problem as at the end of the processing, the value of p_j is adjusted. The same issue arises in the implementation of the straightforward algorithm discussed earlier. The process is best understood by examining the pseudocode below and noticing how the arrays *tempSum* and *tempPCount* are updated in the recursive linking process.

Algorithm : Updating the Recursive Structure

```
double[][][] temp1TempSum = FirstTempSum;
double[][][][][] temp1TempPCount = FirstTempPCount;
double[][][] temp2TempSum=new double[6][distinctGenotypes][11];
double[][][][][] temp2TempPCount=new double[6][distinctGenotypes][11][3]
[loci-1];
INTERMEDIATE STEPS
int startRowIndex=0;
int endRowIndex=loci-height;//note that there is a common gene in each interval
int startPos =0;
int endPos=loci-height;
for int indexS=0;indexS<segments-2;indexS++ do
  startPos=endPos-height+2;
  double[] firstCProb=ChopAtoB(cProbOld,startPos,endPos);
```

```
{$ChopAtoB(.)$ extracts elements in $cProbOld$ from $startPos$ to $endPos$}
int[][] spores = getSpores();
{Setting up the current temporary segment}
kCount = new int[6];
k=new int[height-1];
counter=new int[height-2];
activeKTemp = new int[6];
**Get the relevant data**
startRowIndex=endRowIndex-height+1; // endRowIndex is changed in the bottom.
data=GetData(startRowIndex,endRowIndex,0,(distinctGenotypes-1));
Calculate active exchanges. Stored in activeKTemp.
ActiveKLoopTemp(0,height,indexS); // note this uses the object $data$.
{$ActiveKLoopTemp(.)$ is similar in function to the previously
explained function $ActiveKLoopFirstAndLast(.)$ except being for an intermediate segment}
**Allocate cells**
cArrayTemp = new int[6][][][];
linkInfoTemp = new int[6][];
kArrayTemp = new int[6][][];
for int i=0;i<6;i++ do
  cArrayTemp[i] = new int[activeKTemp[i]][distinctGenotypes][4];
  linkInfoTemp[i] = new int[activeKTemp[i]];
  kArrayTemp[i] = new int[activeKTemp[i]][height-1];
end for
{Run the loop to set $kArrayTemp$,$cArrayTemp$ AND $linkInfoTemp$}
LoopTemp(0,height,indexS); // note this uses the object $data$
{$LoopTemp(.)$ is similar in function to the previously explained function
$ComputeFirstAndLast(.)$ except being for an intermediate segment}
{**Linking at work**}
for int index0=0;index0<6;index0++ do
  for int index1=0;index1<distinctGenotypes;index1++ do
    for int index2=0;index2<11;index2++ do
      for int indexK=0;indexK<activeKTemp[index0];indexK++ do
        int[] k=setEqual(kArrayTemp[index0][indexK]);
        double prob=kProb(firstCProb,k);
        int index01=linkInfoTemp[index0][indexK];
        int spike=getSporeMatch(UpdateSpore
        (cArrayTemp[index0][indexK][index1],spores[index2]));
        {$getSporeMatch(.)$ converts spores in Lemma 2 to one of the integers
        from 1 to 11}
```

```
            temp2TempSum[index0][index1][index2]+=temp1TempSum[index01]
            [index1][spike]*prob;
            for int index=startPos-1;index<endPos;index++ do
              temp2TempPCount[index0][index1][index2]
              [CellSpecial(k[index-startPos+1])][index]+=
              temp1TempSum[index01][index1][spike]*prob;
            end for
            for int i=0;i<3;i++ do
              for int index=endPos;index<loci-1;index++ do
                temp2TempPCount[index0][index1][index2][i][index]+=
                temp1TempPCount[index01][index1][spike][i][index]*prob;
              end for
            end for
          end for
        end for
      end for
    end for
    CHANGE TEMP1 TO TEMP2
    temp1TempSum = setEqual(temp2TempSum);
    temp1TempPCount = setEqual(temp2TempPCount);
    SET TEMP2 TO ZEROS FOR SECURITY
    temp2TempSum=new double[6][distinctGenotypes][11];
    temp2TempPCount=new double[6][distinctGenotypes][11][3][loci-1];
    endPos=startPos-1;
    endRowIndex=startRowIndex;
  end for
```

7.7. *Linking with the first segment*

Linking with the first segment is the last step of the likelihood computation process. In this phase we do not have any previous matching status vectors and instead of *tempPCount*, we have the structure *pCount* as in the straightforward algorithm. Note that the length of the first segment must be adjusted to account for both, even and odd number of genetic markers. That entails a trivial modification of the algorithm, and hence we do not mention the details. In the interest of clarity of description of the algorithm, we have assumed all the segments to be of equal length, and hence both an even and odd number of markers cannot be implemented without first changing the length of at least one segment; preferably that of the first one.

Algorithm: Linking with the First Segment

```
double[] firstCProb=getCProb(cProbOld,1,height-1);
for int index1=0;index1<activeKFirst;index1++ do
```

```
    int[] k=setEqual(kArrayFirst[index1]);
    double prob=kProb(firstCProb,k);
    int[] spike=new int[distinctGenotypes];
    {Spike is updated spore before the last segment}
    for int i=0;i<distinctGenotypes;i++ do
      spike[i]=getSporeMatch(cArrayFirst[index1][i]);
    end for
    {probFOld and pCount are global variables}
    int index0=linkInfoFirst[index1];
    for int index3=0;index3<distinctGenotypes;index3++ do
      probFOld[index3]+=temp1TempSum[index0][index3][spike[index3]]*prob;
      for int index=0;index<height-1;index++ do
        pCount[CellSpecial(k[index])][index][index3]+=
        temp1TempSum[index0][index3][spike[index3]]
        *prob*genotypeFreq[index3]/totalObs;
      end for
      for int i=0;i<3;i++ do
        for int index=height-1;index<loci-1;index++ do
          pCount[i][index][index3]+=
          temp1TempPCount[index0][index3][spike[index3]]
          [index]*prob*genotypeFreq[index3]/totalObs;
        end for
      end for
    end for
  end for
```

7.8. *Implementing the EM algorithm by recursive linking*

The following algorithm wraps around the entire EM algorithm as implemented in the proposed recursive linking algorithm. This is similar in many aspects to the straightforward algorithm except that the function *SpiralUp*() joins the broken segments and updates the exchange probability (c) estimate and treats the entire process as if it were never decomposed. The description of the algorithm is as follows

Algorithm: Implementing the EM Algorithm by Recursive Linking

```
for int iteration=0;iteration<iterationLimit;iteration++ do
  double[] cProbNew=new double[order.length-1];
  probFOld=new double[distinctGenotypes];
  pCount=new double[3][order.length-1][distinctGenotypes];
  double[][] postCount=new double[3][order.length-1];
  RECURSIVELY JOINING THE BROKEN GENETIC SEG-
  MENTS AND IMPLEMENTING THE EM ALGORITHM
  SpiralUp();
  {SpiralUp() Implements the following algorithms previously outlined in the
  order they appear:
  Algorithm1 : Computing the Recursive Objects tempSum and tempPCount
  Algorithm2 : Updating the Recursive Structure
  Algorithm3 : Joining with the First Segment}
  Reporting the likelihood
  logLikelihood=0.0;
  for int index=0;index<distinctGenotypes;index++ do
    logLikelihood+=genotypeFreq[index]*Math.log(probFOld[index]);
  end for
  System.out.println("The log-likelihood is "+logLikelihood+" at iteration
  "+iteration);
  GETTING THE POSTERIOR COUNTS: POSTCOUNT
  OF SIZE 3
  for int index1=0;index1<3;index1++ do
    for int index2=0;index2<order.length-1;index2++ do
      for int index3=0;index3<distinctGenotypes;index3++ do
        pCount[index1][index2][index3]=pCount[index1][index2][index3]/ prob-
        FOld[index3];
      end for
      postCount[index1][index2]=Sum(pCount[index1][index2]);
    end for
  end for
  Getting the c probability estimates
  for int index1=0;index1<loci-1;index1++ do
    cProbNew[index1]=
    (postCount[1][index1]+2*postCount[2][index1])/(2.0*Sum(getCol
    (postCount,index1+1)));
  end for
  if Convergence(cProbOld,cProbNew,0.01) ==1 then
    System.out.println("Convergence Achieved !");
    break;
  end if
  cProbOld=setEqual(cProbNew);
end for
```

8. Time Complexity Comparison of the Algorithms

In the analysis of time complexity of the genetic map reconstruction algorithms,[26] the running variable is l, the number of genetic markers. The core function of the algorithm is to process a large number of exchanges in real time. The algorithm would be required even if one wanted to compute just the likelihood for the initial exchange probabilities and not use the subsequent EM iterations. Hence in both the straightforward and the proposed algorithm we provide run time complexity analysis for the main computationally intensive phase, namely processing all the exchanges for a single iteration.

The loop in the straightforward algorithm runs for $25^{(l-1)}$ iterations. In each iteration the computation time associated with matrix R is $O(l)$, since matrix R has l columns and the computation time for each column is fixed. Since the observed genotypes are not known in advance we do a worst-case analysis for the computation of the vector *sum*. The worst case occurs when there is a match between an observed genotype and any of the 4 columns of the matrix R resulting in run time complexity of order $O(l)$. The vector *sum* has d_g elements and its size does not vary with l. Hence, the total execution time for computing vector *sum* is $O(l)$. The vector addition involved in computing $pCount$ entails the processing of d_g elements and thus accounts for run time complexity of $O(l)$. Hence the total run time complexity of the straightforward algorithm is given by

$$R_{st} = O(l25^{(l-1)}) \tag{15}$$

In the recursive linking algorithm the run time for each of the s segments is $O((h-1)25^{(h-1)})$. So the total run time for the recursive algorithm is given by

$$R_{rl} = O(s(h-1)25^{(h-1)}) \tag{16}$$

which is minimized for $h = 3$ for any odd number of loci l. Hence for $h = 3$ the run time complexity for the recursive linking algorithm is of order $O(l)$ using Eqs. (14) and (16).

9. Run Time Results

We ran several jobs on a DELL PC (Model DM051 Pentinum(R) 4 CPU 3.40GHz and 4GB of RAM) for different number of genetic markers in their natural order of precedence on a data set from the linkage group-I of *N-crassa*.[18] for both algorithms. It was verified that both the algorithms provide the same likelihood for the same order of markers as they essentially solve the same problem but in different ways. The run time corresponds to the average time taken for a single EM iteration before convergence upon multiple starts. The resulting speedup is clearly evident from results in Table 3.

10. Additional Application of the Recursive Linking Algorithm

Besides computing the likelihood, the recursive linking algorithm can be used successfully to compute the standard error of c_i for each genetic interval S_i. Without this algorithm, computation of a standard error suffers from the same computational bottleneck as the computation of the likelihood. In the following sections, we briefly describe the theory used to compute a standard error and show in detail how the recursive linking algorithm is used to implement the theory. The theory in Sec. 10.1 has appeared elsewhere but is mentioned here briefly in order to make this paper self-contained and the algorithm more readable.

Table 3. Run time comparison of straight forward and recursive algorithm

	Runtime (secs)	
Loci(l)	Straight Forward Algorithm	Recursive Linking Algorithm
5	2.49	0.073
7	1336.45	0.298
9	> 43200.0	0.66
21	∞	5.61
41	∞	8.93
61	∞	13.03

10.1. *Computing the standard error of MLE*

We use the *SEM* algorithm[16] to compute the standard errors of $\Theta = (c_m, m = 1, \ldots, l-1)$. The large sample variance-covariance matrix is given by [Eq. (2.3.5) in Ref. 16)

$$V = I_{oc}^{-1} + \Delta V \tag{17}$$

where

$$\Delta V = I_{\text{oc}}^{-1} D(I-D)^{-1}$$

and D is the matrix corresponding to the rate of convergence of EM and I_{oc} is defined as below:

$$I_{\text{oc}} = E[I_o(\theta|Y)|Y_{\text{obs}}, \theta]\Big|_{\theta=\theta^*} \tag{18}$$

where $I_o(\theta|Y)$ is the complete-data observed information matrix.

The EM algorithm described in Sec. 1 implicitly defines a mapping $\theta \rightarrow M(\theta)$ via Eq. (12) from the parameter space of θ,$(0,1]^{l-1}$, to itself such that

$$\theta^{(t+1)} = M(\theta^{(t)}), \quad \text{for } t = 0, 1, \ldots.$$

Since $M(\theta)$ is continuous and $\theta^{(t)}$ converges to the MLE θ^* (using EM algorithm), then θ^* must satisfy

$$\theta^* = M(\theta^*).$$

Therefore in the neighborhood of θ^*, using a Taylor series expansion, we get

$$\theta^{(t+1)} - \theta^* \approx (\theta^{(t)} - \theta^*)D$$

where

$$D = \left(\frac{\partial M_j(\theta)}{\partial \theta_i}\right)\Bigg|_{\theta=\theta^*}$$

is the $(l-1) \times (l-1)$ Jacobian matrix for $M(\theta) = (M_1(\theta), \ldots, M_{l-1}(\theta))$ evaluated at $\theta = \theta^*$.

10.2. *Computation of D*

Let d_{ij} be the (i,j)th element of D and define $\theta^{(t)}(i)$ to be

$$\theta^{(t)}(i) = \left(\theta_1^*, \ldots, \theta_i^{(t)}, \theta_{i+1}^*, \ldots, \theta_{l-1}^*\right) \tag{19}$$

i.e. only the ith component in $\theta^{(t)}(i)$ is active in the sense that the other components are fixed at their MLE values. By the definition of d_{ij}, we have

$$\begin{aligned} d_{ij} &= \frac{\partial M_j(\theta^*)}{\partial \theta_i} \\ &= \lim_{\theta_i \to \theta_i^*} \frac{M_j\left(\theta_1^*, \ldots, \theta_{i-1}^*, \theta_i, \theta_{i+1}^*, \ldots, \theta_{l-1}^*\right) - M_j(\theta^*)}{\theta_i - \theta_i^*} \\ &= \lim_{\theta_i \to \theta_i^*} \frac{M_j(\theta^{(t)}(i)) - \theta_j^*}{\theta_i - \theta_i^*} \\ &= \lim_{t \to \infty} d_{ij}^{(t)} \end{aligned}$$

The following steps are performed to compute the d_{ij}'s.

INPUT: θ^* and $\theta^{(t)}$.

Step 1. Run the usual E and M steps to obtain $\theta^{(t+1)}(i)$.

Repeat steps 2–3 for $i = 1, \ldots, l-1$.

Step 2. Calculate $\theta^{(t)}(i)$ from Eq. (19), and treating it as the current estimate of θ, run an additional iteration of EM to obtain $\theta^{\ (t+1)}(i)$.

Step 3. Obtain the ratio

$$d_{ij} = \frac{\theta_j^{\ (t+1)}(i) - \theta_j^*}{\theta_i - \theta_i^*}, \quad \text{for } j = 1, \ldots, l-1$$

OUTPUT: $\theta^{(t+1)}$ and $\{d_{ij}^{(t)}, i, j = 1, \ldots, l-1\}$.

We obtain $d_{ij}$when the sequence $d_{ij}^{(t^*)}, d_{ij}^{(t^*+1)}, \ldots$ is stable for some t^*. This process may result in using different values of t^* for different d_{ij} elements.

10.3. *Evaluation of I_{oc}^{-1}*

The complete-data information for the (i,j)th element $(i=1,\ldots,l-1$ and $j=1,\ldots,l-1)$ is given by:

$$\begin{aligned} I^o(i,j) &= -\frac{\partial^2 f^c(x,\theta)}{\partial \theta_i \partial \theta_j} \\ &= -\sum_k x_k \frac{\pi_k \frac{\partial^2}{\partial \theta_i \partial \theta_j}(\pi_k) - \frac{\partial}{\partial \theta_i}(\pi_k)\frac{\partial}{\partial \theta_j}(\pi_k)}{\pi_k^2} \end{aligned}$$

The complete-data information for the $(i,j)^{\text{th}}$ element ($i = 1, \ldots, l-1$ and $j = 1, \ldots, l-1$) is given by

$$\begin{aligned} I^o(i,j) &= -\frac{\partial^2 f^c(x,\theta)}{\partial\theta_i \partial\theta_j} \\ &= -\sum_k x_k \frac{\pi_k \frac{\partial^2}{\partial\theta_i\partial\theta_j}(\pi_k) - \frac{\partial}{\partial\theta_i}(\pi_k)\frac{\partial}{\partial\theta_j}(\pi_k)}{\pi_k^2} \end{aligned}$$

Using Eq. (18),

$$\begin{aligned} I_{\text{oc}} &= E[I^o(\theta|Y)|Y_{\text{obs}},\theta]\big|_{\theta=\theta^*} \\ &= \sum_k -x_k \frac{\pi_k \frac{\partial^2}{\partial\theta_i\partial\theta_j}(\pi_k) - \frac{\partial}{\partial\theta_i}(\pi_k)\frac{\partial}{\partial\theta_j}(\pi_k)}{\pi_k^2}[E\left(x_k|n,\theta^*\right)]] \\ &= \sum_k \left[-x_k \frac{\pi_k \frac{\partial^2}{\partial\theta_i\partial\theta_j}(\pi_k) - \frac{\partial}{\partial\theta_i}(\pi_k)\frac{\partial}{\partial\theta_j}(\pi_k)}{\pi_k^2}\left[\sum_{j'=1}^{N} n'_j \pi_{k|j'}(\theta^*)\right]\right] \\ &= \sum_{j'=1}^{N} \frac{n_{j'}}{p_{j'}}\left[\sum_k \left[\frac{\pi_{j'|k}}{\pi_k}\frac{\partial}{\partial\theta_i}(\pi_k)\frac{\partial}{\partial\theta_j}(\pi_k) - \pi_{j'|k}\frac{\partial^2}{\partial\theta_i\partial\theta_j}(\pi_k)\right]\right] \quad \text{[using Eq. (12)]} \end{aligned} \tag{20}$$

Note that this is the only part in the computation of standard error that requires processing an exponential number of exchanges and the proposed recursive linking algorithm can be successfully used to address the computational bottleneck.

10.4. *Algorithm to compute the standard error of MLE*

In the following algorithm active exchanges are computed and the cells allocated in a similar manner as in the likelihood computation. Here also we start with the last segment with all suitable indices as before and attempt to compute Eq. (20). The structure *probDeri* computes the double partial derivatives while *probSingle* and *probSingle2* compute the univariate derivatives in Eq. (20). The structure *FirstTempInfo* accounts for the conditional probability term along with the double derivative whereas the structure *FirstTempInfo2* accounts for the conditional probability term along with the product of the univariate derivatives. Notice that the structures *FirstTempInfo* and *FirstTempInfo2* will be recursively created for all the segments as they progressively help to move across all the exchanges as in the likelihood computation process.

Compute Active Exchanges and do Cell Allocations for all the Segments

{ For each segment the structures *kArray*,*linkInfo* and *cArray* are realized using similar algorithms as in Secs. 7.2 and 7.3. The structures *kArray*, *linkInfo*, and *cArray* for the intermediate segments are arrays *kArrayTemp*[], *linkInfoTemp*[]

and *cArrayTemp*[], where the array index indicates a segment as earlier mentioned in the algorithm in Sec. 7.6.

```
}
Last Part First
double[] lastCProb=ChopAtoB(cProbFinal,loci-height+1,loci-1);
double probDeri=0.0,probSingle=0.0,probSingle2=0.0;
double[][][][][] FirstTempInfo=new
double[6][11][height-1][height-1][distinctGenotypes];
double[][][][][] FirstTempInfo2=new
double[6][11][height-1][height-1][distinctGenotypes];
double[][][][] FirstTempSingleD=new double[6][11][height-1][distinctGenotypes];
double[][][] FirstTempSum=new double[6][distinctGenotypes][11];
for int index1=0;index1<6;index1++ do
  for int index6=0;index6<activeKLast[index1];index6++ do
    double prob=kProb(lastCProb,kArrayLast[index1][index6]);
    for int index2=0;index2<11;index2++ do
      for int index5=0;index5<distinctGenotypes;index5++ do
        condProb=Sum(UpdateSpore(cArrayLast[index1][index6][index5],
        spores[index2]))/4.0;
        for int index3=loci-height;index3<loci-1;index3++ do
          probSingle=kProbSingle(lastCProb,kArrayLast[index1][index6],
          index3-loci+height+1);
          for int index4=loci-height;index4<loci-1;index4++ do
            if  index3! = index4 then
              probDeri=kProbDeriOff(lastCProb,kArrayLast[index1][index6],
                  index3-loci+height+1,index4-loci+height+1);
            else
              probDeri=kProbDeriDiag(lastCProb,kArrayLast[index1][index6],
                  index3-loci+height+1);
            end if
            FirstTempInfo[index1][index2][index3-loci+height]
                [index4-loci+height][index5]+=condProb*probDeri;
            probSingle2=kProbSingle(lastCProb,kArrayLast[index1]
                [index6],index4-loci+height+1);
            FirstTempInfo2[index1][index2][index3-loci+height]
                [index4-loci+height][index5]+=condProb/prob*probSingle*
                probSingle2;
          end for
          FirstTempSingleD[index1][index2][index3-loci+height]
              [index5]+=condProb*probSingle;
        end for
        FirstTempSum[index1][index5][index2]+=condProb*prob;
```

```
         end for
      end for
   end for
end for
```

Let us consider how the information matrix is configured when the second to last segment is processed. Note that the dimension of the information matrix (which is inherently a square matrix) has now increased from $height - 1$ to $2(height - 1)$ to account for intervals of 2 segments. We have 4 different cases for rows and columns arising from these 2 segments. For each of these cases, the double derivatives and single derivatives are processed individually. For the first case, the rows and columns are both obtained from the second to last segment; for the second case, the rows are obtained from the second to last segment and the columns from the last segment and so on, until for the fourth case, both the rows and columns are obtained from the last segment. Notice in the pseudocode below how double derivative computation is necessary only for the first case and for the remaining cases everything can be computed from the previous segment except for computation of the single derivatives. The probability adjustments are made in a similar manner as during the likelihood computation process.

Recursive Loop across all the Segments

```
startRowIndex=0;
endRowIndex=loci-height;//note that there is a common gene in each interval
int startPos =0;
int endPos=loci-height;
double[][][][][]temp1Info=FirstTempInfo;
double[][][][][]temp1Info2=FirstTempInfo2;
double[][][][]temp1SingleD=FirstTempSingleD;
double[][][]temp1Sum=FirstTempSum;
for int indexS=0;indexS<segments-2;indexS++ do
   for int index1=0;index1<6;index1++ do
      for int indexK=0;indexK<activeKTemp[index1];indexK++ do
         double prob=kProb(tempCProb,kArrayTemp[index1][indexK]);
         for int index2=0;index2<11;index2++ do
            for int index5=0;index5<distinctGenotypes;index5++ do
               int spike=getSporeMatch(UpdateSpore(cArrayTemp[index1][indexK]
                     [index5],spores[index2]));
               int index01=linkInfoTemp[index1][indexK];
               temp2Sum[index1][index5][index2]+=temp1Sum[index01][index5]
               [spike]*prob;
               for int index3=startPos-1;index3<loci-1;index3++ do
                  for int index4=startPos-1;index4<loci-1;index4++ do
                     if  (index3>=startPos-1 AND index3<=endPos-1)
      AND (index4>=startPos-1 AND index4<=endPos-1) then
  if index3 != index4 then
    probDeri=kProbDeriOff(tempCProb,kArrayTemp[index1]
         [indexK],index3-startPos+1+1,index4-startPos+1+1);
  else
    probDeri=kProbDeriDiag(tempCProb,kArrayTemp[index1]
         [indexK],index3-startPos+1+1);
  end if
  temp2Info[index1][index2][index3-startPos+1]
       [index4-startPos+1][index5]+=temp1Sum[index01]
           [index5][spike]*probDeri;
  probSingle=kProbSingle(tempCProb,kArrayTemp[index1]
       [indexK],index3-startPos+1+1);
  probSingle2=kProbSingle(tempCProb,kArrayTemp[index1]
       [indexK],index4-startPos+1+1);
  temp2Info2[index1][index2][index3-startPos+1]
       [index4-startPos+1][index5]+=temp1Sum[index01]
           [index5][spike]*probSingle*probSingle2/prob;
else
  if (index3>=startPos-1 AND index3<=endPos-1)
       AND (index4>=endPos AND index4<=loci-2)  then
    probSingle=kProbSingle(tempCProb,kArrayTemp[index1]
         [indexK],index3-startPos+1+1);
    temp2Info[index1][index2][index3-startPos+1]
         [index4-startPos+1][index5]+=temp1SingleD[index01]
           [spike][index4-endPos][index5]*probSingle;
    temp2Info2[index1][index2][index3-startPos+1]
         [index4-startPos+1][index5]+=temp1SingleD[index01]
           [spike][index4-endPos][index5]*probSingle;
  else
    if (index3>=endPos AND index3<=loci-2)
    AND (index4>=startPos-1 AND index3<=endPos-1)  then
      probSingle=kProbSingle(tempCProb,kArrayTemp
        [index1][indexK],index4-startPos+1+1);
      temp2Info[index1][index2][index3-startPos+1]
        [index4-startPos+1][index5]+=temp1SingleD
        [index01][spike][index3-endPos][index5]*probSingle;
      temp2Info2[index1][index2][index3-startPos+1]
        [index4-startPos+1][index5]+=temp1SingleD
        [index01][spike][index3-endPos][index5]*probSingle;
    else
      if (index3>=endPos AND index3<=loci-2)
```

```
                    AND (index4>=endPos AND index4<=loci-2) then
                     temp2Info[index1][index2][index3-startPos+1]
                     [index4-startPos+1][index5]+=temp1Info[index01]
                        [spike][index3-endPos][index4-endPos]
                         [index5]*prob;
                     temp2Info2[index1][index2][index3-startPos+1]
                     [index4-startPos+1][index5]+=temp1Info2
                     [index01][spike][index3-endPos][index4-endPos]
                        [index5]*prob;
                   end if
                 end if
               end if
             end if
             if (index3>=startPos-1 AND index3<=endPos-1) then
               probSingle=kProbSingle(tempCProb,kArrayTemp[index1]
                 [indexK],index3-startPos+1+1);
               temp2SingleD[index1][index2][index3-startPos+1][index5]+=
                 temp1Sum[index01][index5][spike]*probSingle;
             else
               temp2SingleD[index1][index2][index3-startPos+1][index5]+=
                 temp1SingleD[index01][spike][index3-endPos][index5]*prob;
             end if
           end for
         end for
       end for
     end for
   end for
 end for
 Change temp1 to temp2
 temp1Info = setEqual(temp2Info);
 temp1Info2 = setEqual(temp2Info2);
 temp1SingleD = setEqual(temp2SingleD);
 temp1Sum=setEqual(temp2Sum);
 endPos=startPos-1;
 endRowIndex=startRowIndex;
end for
```

Computation of double and single derivatives for all the segments completes with joining with the first segment as earlier.

Linking with the First Segment

```
double[] firstCProb=ChopAtoB(cProbFinal,1,height-1);
double[][][] tempInfo=new double[loci-1][loci-1][distinctGenotypes];
double[][][] tempInfo2=new double[loci-1][loci-1][distinctGenotypes];
double[][] tempSingleD=new double[loci-1][distinctGenotypes];
for int index1=0;index1<activeKFirst;index1++ do
  int[] spike=new int[distinctGenotypes];
  for int i=0;i<distinctGenotypes;i++ do
    spike[i]=getSporeMatch(cArrayFirst[index1][i]);
  end for
  prob=kProb(firstCProb,kArrayFirst[index1]);
  index01=linkInfoFirst[index1];
  for int index2=0;index2<distinctGenotypes;index2++ do
    probFOld1[index2]+=temp1Sum[index01][index2][spike[index2]]*prob;
    for int index3=0;index3<loci-1;index3++ do
      for int index4=0;index4<loci-1;index4++ do
        if  (index3>=0 AND index3<=height-2)
             AND (index4>=0 AND index4<=height-2) then
          if index3 != index4 then
            probDeri=kProbDeriOff(firstCProb,kArrayFirst[index1],
            index3+1,index4+1);
          else
            probDeri=kProbDeriDiag(firstCProb,kArrayFirst[index1],
            index3+1);
          end if
          tempInfo[index3][index4][index2]+=temp1Sum[index01][index2]
            [spike[index2]]*probDeri;
          probSingle=kProbSingle(firstCProb,kArrayFirst[index1],index3+1);
          probSingle2=kProbSingle(firstCProb,kArrayFirst[index1],index4+1);
          tempInfo2[index3][index4][index2]+=temp1Sum[index01][index2]
            [spike[index2]]*probSingle*probSingle2/prob;
        else
          if (index3>=0 AND index3<=height-2)
            AND (index4>=height-1 AND index4<=loci-2)  then
            probSingle=kProbSingle(firstCProb,kArrayFirst[index1],index3+1);
            tempInfo[index3][index4][index2]+=temp1SingleD[index01]
            [spike[index2]]
              [index4-endPos][index2]*probSingle;
            tempInfo2[index3][index4][index2]+=temp1SingleD[index01]
              [spike[index2]][index4-endPos][index2]*probSingle;
          else
            if (index3>=height-1 AND index3<=loci-2)
              AND(index4>=0 AND index4<=height-2)  then
```

```
                probSingle=kProbSingle(firstCProb,kArrayFirst[index1],index4+1);
                tempInfo[index3][index4][index2]+=temp1SingleD[index01]
                   [spike[index2]][index3-endPos][index2]*probSingle;
                tempInfo2[index3][index4][index2]+=temp1SingleD[index01]
                   [spike[index2]][index3-endPos][index2]*probSingle;
             else
                if (index3>=height-1 AND index3<=loci-2)
                   AND (index4>=height-1 AND index4<=loci-2) then
                   tempInfo[index3][index4][index2]+=temp1Info[index01]
                      [spike[index2]][index3-endPos][index4-endPos][index2]*prob;
                   tempInfo2[index3][index4][index2]+=temp1Info2[index01]
                      [spike[index2]][index3-endPos][index4-endPos][index2]*prob;
                end if
             end if
          end if
       end if
    end for
    if index3>=0 AND index3<=height-2 then
       probSingle=kProbSingle(firstCProb,kArrayFirst[index1],index3+1);
       tempSingleD[index3][index2]+=temp1Sum[index01][index2]
          [spike[index2]]*probSingle;
    else
       tempSingleD[index3][index2]+=temp1SingleD[index01]
          [spike[index2]][index3-endPos][index2]*prob;
    end if
  end for
 end for
end for
```

We compute the information matrix using Eq. (20). Computing the variance-covariance matrix using Eq. (17) is straightforward. We do not show the computation of matrix D in equation (17) represented by *dm2mat* in the pseudocode since the implementation is straightforward.

Getting the Expected Information Matrix

```
double[][] info=new double[loci-1][loci-1];
double[][] info1=new double[loci-1][loci-1];
double sumCov=0.0;
for int index1=0;index1<loci-1;index1++ do
   for int index2=0;index2<loci-1;index2++ do
      sumCov=0.0;
      for int index3=0;index3<distinctGenotypes;index3++ do
```

```
            sumCov+=(tempInfo2[index1][index2][index3]-
            tempInfo[index1][index2][index3])
                  *genotypeFreq[index3]/probFOld1[index3];
        end for
        info[index1][index2]=sumCov;
    end for
end for
Matrix inverse=new Matrix(info).inverse();
Matrix dm2Mat =new Matrix(dm2);
Matrix delV=inverse.times(dm2Mat).times((Matrix.identity(loci-1,loci-1).minus
(dm2Mat)).inverse());
Matrix varCov=inverse.plus(delV);
for int index=0;index<varCov.getRowDimension();index++ do
    SE[index]=Math.sqrt(varCov.get(index,index));
end for
```

11. Genetic Mapping from the RFLP Data of *N. crassa*

We use our algorithm to create a genetic map for the entire genome of *N. crassa*. The chromosomes involve a large number of markers (around 60). The computation of the likelihood along with the estimates of c_i and their standard errors would have been impractical with any straightforward algorithm, however it was made possible with the proposed algorithm. We used the widely known stochastic optimization technique *simulated annealing* (SA) to determine the best genetic map by performing combinatorial optimization over the space of possible marker orders.

11.1. *The use of SA for searching the best order*

Several combinatorial optimization problems can be tackled using SA as a stochastic optimization technique.[11] We use SA to determine the best order of genetic markers by sampling stochastically from the space of all possible gene orders. Simulated annealing has been used previously,[3] to reconstruct chromosomes based on binary scoring of DNA fragments and a Hamming distance-based objective function. In our case, the objective function is the likelihood of a particular order of genes on a genetic map obtained upon convergence of the EM algorithm.[17,27] Thus, in our case a single computation of the objective function for a single order of markers is quite expensive. The SA provides a stochastic sampling technique to search for a good order as follows

(1) Choose a random order of probes, Π, and calculate $f(\Pi)$.
(2) Choose a random segment within the ordering Π.
(3) Perform a segment reversal and call the new ordering Π'.
(4) Compute $f(\Pi')$.

(5) If $f(\Pi')$ is less than $f(\Pi)$, then retain the new order. However, if $f(\Pi')$ is larger than $f(\Pi)$, then generate a random number between 0 and 1. If this random number is less than $E(-(f(\Pi') - f(\Pi))/T$, then retain the new order. Here T is the "temperature" of the annealing schedule.
(6) Proceed to anneal the temperature in multiplicative steps, i.e., decrease T by a factor F at each SA iteration. Hold each value of T constant until D re-orderings have been attempted or S successful re-orderings have resulted, whichever comes first. If the number of successes equals zero for a given step, the process is complete; otherwise go to step 2.

SA has been used to create a genetic map of chromosome-II in *N. crassa*.[18,20] For each order in SA, the likelihood is computed, and the order with the maximum value of the likelihood is selected as the desired map. The values of the parameters used in the search of the best order are $T = 10.0$,$F = 0.25$,$D = 500$ and $S = 50$. For the best order, c_is, their standard errors and their corresponding recombination fractions (see Sec. 5.1) for each genetic interval are computed. These statistics are reported for the best order in Table 4. The best order for linkage group-II turned out to be different from the published order[18] and has log-likelihood value of −63.9341. There are at least two reasons for the discrepancy. The simplest is that

Table 4. Best genetic map with estimates of parameters (c), its standard error (σ_c) and recombination fraction (r) on linkage Group-II of *N. Crassa*.

Genes	c[a]	σ_c	r	Genes[b]	c	σ_c	r
Tel IIL	0	0.0022	0	Ncr-5	0.1588	0.2164	0.1462
hsps-1	0.2127	0.2019	0.1901	preg, cit-1	0.1524	0.2056	0.1408
00008	0.0002	0.0078	0.0002	12:11C, pma-1	0.3229	0.308	0.2708
pSK2-1A	0.1263	0.1709	0.1183	nuo78	0.0012	0.0248	0.0012
3:9A	0.1409	0.1894	0.131	21:D3	0.0012	0.0248	0.0012
Fsr-52	0.0007	0.0182	0.0007	X18:9A	0.0012	0.0248	0.0012
AP32b.1,R31.1, vph-1	0.1555	0.2268	0.1434	11:F3	0.1614	0.2196	0.1484
Fsr-32	0.0011	0.0238	0.0011	ccg-7	0.0012	0.0241	0.0012
Cen II, arg-5	0.0011	0.0238	0.0011	vma-2	0.0012	0.0241	0.0012
25:1D	0.0011	0.0238	0.0011	DB0001	0.0012	0.0241	0.0012
G8:11H	0.0011	0.0238	0.0011	Fsr-3	0.1588	0.2164	0.1462
atp-2	0.0011	0.0238	0.0011	X24:A11	0.1588	0.2164	0.1462
cya-4, Fsr-55	0.0011	0.0238	0.0011	eas=ccg-2=bli-7	0.1588	0.2164	0.1462
AP5i.1	0.0011	0.0238	0.0011	bli-4	0.1555	0.2119	0.1434
AP5.1	0.0011	0.0238	0.0011	Fsr-34	0.3153	0.3057	0.2656
AP3.2	0.154	0.196	0.1421	Fsr-17	0.001	0.0223	0.001
AP8u.4	0.154	0.196	0.1421	AP13.4 , 8:4GL	0.001	0.0223	0.001
H3H4	0.0011	0.0237	0.0011	AP32c.2, R32.2	0.001	0.0223	0.001
Ncr-2	0.0011	0.0237	0.0011	leu-6	0.1557	0.2129	0.1436
				Tel IIR			

Note: [a] c, σ_c, and r correspond to the genetic interval formed by the genes at the current row and the following row.
[b]Continuation of the first 4 columns of Table 4.

the maps were constructed by hand, so there is only a limited number of possible solutions considered. Metzenberg (personal communication) has also indicated that he used additional data that helped him decide on the map, data which the current computation did not have access to. The maps and their statistics for the remaining six chromosomes of *N. crassa* can be found at http://gene.genetics.uga.edu.

12. Conclusions

A multi-locus genetic likelihood was computed based on a mathematical model of the chromatid exchange in meiosis that accounts for any type of bivalent configuration in a genetic interval in any specified order of genetic markers. We proposed an algorithm that can reduce the time complexity of computations, for example, computing the likelihood, finding estimates of parameters (c_i) and their standard errors from exponential to linear time in the number of genetic markers. To illustrate the advantages of the proposed algorithm, we compared it alongside a straightforward algorithm presenting both, theoretical worst-case analysis and runtime results. Finally as an application, we reconstructed a genetic map of the entire genome of the model fungal system *Neurospora crassa* which otherwise would have been impractical to achieve, considering the number of markers involved.

Acknowledgments

The research was supported in part by a grant from the U.S. Department of Agriculture under the NRI Competitive Grants Program (Award No: GEO-2002-03590) to Drs. Bhandarkar and Arnold. We thank the College of Agricultural and Environmental Sciences, University of Georgia for their support.

References

1. The International Hapmap Consortium, A haplotype map of the human genome, *Nature* **437**:1299–1320, 2005.
2. Barratt RW, Newmeyer D, Perkins DD, Garnjobst L, Map construction in *Neurospora crassa*, *Advances in Genetics* **6**:1–93, 1954.
3. Cuticchia AJ, Arnold J, Timberlake WE, The use of simulated annealing in chromosome reconstruction experiments based on binary scoring, *Genetics* **132**:591–601, 1992.
4. Daphne P, Seung YR, Davis RW, Tetrad analysis possible in arabidopsis with mutation of the QUARTET(QRT) genes. *Science* **264**:1458–1460, 1994.
5. Davis RH, *NEUROSPORA Contributions of a Model Organism.* Addison-Wesley, 2000.
6. Dempster A, Laird N, Rubin D, Maximum likelihood from incomplete data via the EM algorithm, *J Royal Stat Soc Series B* **39**(1):1–38, 1977.
7. Doerge RW, Zeng ZB, Weir BS, Statistical issues in the search for genes affecting quantitative traits in experimental populations, *Stat Sci* **12**:195–219, 1997.
8. Foss E, Lande R, Stahl FW, Steinberg CM, Chiasma interference as a function of genetic distance, *Genetics* **133**:681–691, 1993.

9. Haldane JBS, The combination of linkage values and the calculation of distance between the loci of linkage factors, *J Genetics* **8**:229–309, 1919.
10. John L, William L, *Java Software Solutions*, Oxford University Press, 2004.
11. Kirkpatrick S, Gelatt CD Jr, Vecchi MP, Optimization by simulated annealing, *Science* **220**:671–680, 1983.
12. Lander ES, Green P, Construction of multi-locus genetic linkage maps in humans, *Natl Proc Sci Acad USA* **84**:2363–2367, 1987.
13. Lander ES and Schork NJ, Genetic dissection of complex traits, *Science* **265**:2037–2048, 1994.
14. Lange K, Zhao H, Speed TP, The Poisson-skip model of crossing-over, *The Annals of Applied Probability* **7**:299–313, 1997.
15. Mather K, *The Measurement of Linkage in Heredity*, John Wiley & Sons NY, New York, 1951.
16. Meng XL, Rubin DB, Using EM to obtain asymptotc variance-covariance matrices: The SEM algorihm, *J Amer Stat Assoc* **86**:899–909, 1991.
17. Mester D, Romin Y, Minkov D, Nevo E, Korol A, Constructing large-scale genetic maps using an evolutionary strategy algorithm, *Genetics* **165**:2269–2282, 2003.
18. Nelson MA, Crawford ME, Natvig DO, Restriction polymorphism maps of *Neurospora crassa*: 1998 update, *http://www.fgsc.net/fgn45/45rflp.html*, 1998.
19. Norman LJ, Samuel K, Adrienne WK, *Univariate Discrete Distributions*. Wiley-Interscience, 2 ed., 1993.
20. Perkins D, Radford A, Sachs M, *The Neurospora Compendium*, Academic Press, 2001.
21. Raju NB, Meiosis and ascospore genesis in Neurospora, *J Eur Biol Cell* **23**:208–223, 1980.
22. Rao CR, *Linear Statistical Inference and Its Application*, 2 ed., Wiley-Interscience, 2002.
23. Schwager SJ, Mutschler MA, Federer WT, Scully BT, The effect of linkage on sample size determination for multiple trait selection, *Theoretical and Applied Genetics/Theoretische und angewandte Genetik* **86**:964–974, 1993.
24. Searle SR, *Matrix Algebra Useful for Statistics*, John Wiley & Sons, 1982.
25. Tewari S, Bhandarkar SM, Arnold J, Efficient recursive linking algorithm for computing the likelihood of an order of a large number of genetic markers, In Markstein P, Xu Y, editors, *Computational Systems and Bioinformatics Conference*, pp. 191–198, 2006.
26. Thomas HC, Charles EL, Ronald LR, Clifford S, *Introduction to Algorithms*, 2 ed., The MIT Press, 2001.
27. Weeks ED, Lange K, Preliminary ranking procedures for multilocus ordering, *Genomics* **1**:236–242, 1987.
28. Winzeler EA, Richards DR, Conway AR, Goldstein AL, Kalman S, McCullough MJ, McCusker JH, Stevens DA, Wodicka L, Lockhart DJ, Davis RW, Direct allelic variation scanning of the yeast genome, *Science* **281**:1194–1197, 1998.
29. Zhao H, McPeek MS, Speed TP, Statistical analysis of chromatid interference, *Genetics* **139**:1057–1065, 1995.
30. Zhao H, McPeek MS, Speed TP, Statistical analysis of crossover interference using the Chi-square model, *Genetics* **139**:1045–1056, 1995.
31. Zhao H, Speed TP, Stochastic modeling of the crossover process during meiosis, *Communications in Statistics, Theory and Methods* **27**:1557–1580, 1998.
32. Zhao H, Speed TP, Statistical analysis of ordered tetrads, *Genetics* **150**:459–472, 1998.