BIOINFORMATICS

Vol. 21 Suppl. 1 2005, pages i413–i422
doi:10.1093/bioinformatics/bti1033

Efficient computation of close lower and upper bounds on the minimum number of recombinations in biological sequence evolution

Yun S. Song*, Yufeng Wu and Dan Gusfield

Department of Computer Science, University of California, Davis, CA 95616, USA

Received on January 15, 2005; accepted on March 27, 2005

ABSTRACT
Motivation: We are interested in studying the evolution of DNA single nucleotide polymorphism sequences which have undergone (meiotic) recombination. For a given set of sequences, computing the minimum number of recombinations needed to explain the sequences (with one mutation per site) is a standard question of interest, but it has been shown to be NP-hard, and previous algorithms that compute it exactly work either only on very small datasets or on problems with special structure.
Results: In this paper, we present efficient, practical methods for computing both upper and lower bounds on the minimum number of needed recombinations, and for constructing evolutionary histories that explain the input sequences. We study in detail the efficiency and accuracy of these algorithms on both simulated and real data sets. The algorithms produce very close upper and lower bounds, which match exactly in a surprisingly wide range of data. Thus, with the use of new, very effective lower bounding methods and an efficient algorithm for computing upper bounds, this approach allows the efficient, exact computation of the minimum number of needed recombinations, with high frequency in a large range of data. When upper and lower bounds match, evolutionary histories found by our algorithm correspond to the most parsimonious histories.
Availability: HapBound and SHRUB, programs implementing the new algorithms discussed in this paper, are available at http://wwwcsif.cs.ucdavis.edu/~gusfield/lu.html
Contact: yssong@cs.ucdavis.edu

1 INTRODUCTION

Recombination is a biological process common to most forms of life. Meiotic recombination takes two equal length sequences and produces a third sequence of the same length consisting of some prefix of one of the sequences followed by a suffix of the other sequence. Meiotic recombination is one of the principal evolutionary forces responsible for shaping genetic variation within species, and other forms of recombination allow the sharing of genetic material between species. Efforts to deduce patterns of historical recombination or to estimate the frequency or the location of recombination are central to modern-day genetics. An example is the recent study of the human genome (McVean *et al*., 2004).

Although the results and methods presented in this paper mostly use the terminology of meiotic recombination in populations, the results and methods themselves are applicable in a broader biological context. The results apply to any biological phenomena involving a movement of material (represented as a substring in a sequence) from one entity to another, as long as its location in the new sequence is the same as in the originating sequence. Thus, the results also apply to many problems in phylogenetic networks, lateral gene transfer and so on.

In studying recombination, a common underlying problem is to determine the minimum number of recombinations needed to generate a given set of molecular sequences from an ancestral sequence (which may or may not be known), using some specified model of the permitted site mutations. A mutation at a site is a change of state at that site not caused by recombination. A common assumption is the infinite sites model in population genetics, i.e. that any site (in the study) can mutate at most once in the entire history of the sequences. This implies that each site in any of the studied sequences can take on only two states, and hence the sequences we see today are binary sequences. The strongest current validation of this binary sequence assumption comes from data where each site is a single nucleotide polymorphism (SNP) site, i.e. a site where two of the four possible nucleotides appear in the population with a frequency above some set threshold.

Given a set M of binary sequences, we let $R_{\min}(M)$ denote the minimum number of recombinations needed to generate M from any ancestral sequence, allowing only one mutation per site over the entire history of the sequences. The problem of computing or estimating $R_{\min}(M)$ has been studied in a number of papers. A variation to the problem occurs when a specific ancestral sequence is known in advance. No polynomial-time algorithm for either problem is known, and the second problem is claimed to be NP-hard (Wang *et al*., 2001).

*To whom correspondence should be addressed.

© The Author 2005. Published by Oxford University Press. All rights reserved. For Permissions, please email: journals.permissions@oupjournals.org

Song and Hein (2003) developed an algorithm that computes $R_{\min}(M)$ exactly, but it takes super-exponential time, so the program based on it can handle only very small datasets. There is also a polynomial-time algorithm (Gusfield, 2004; Gusfield *et al*., 2004) that computes $R_{\min}(M)$ in special cases that arise frequently when the recombination rate is 'modest'. Hein (1990, 1993) studied a problem similar to computing $R_{\min}(M)$, but under a technically different model, and provided a heuristic and a super-exponential time algorithm for that problem. There are no known efficient algorithms (in theory or practice) that compute non-trivial upper bounds on $R_{\min}(M)$. In this paper we discuss practical algorithms that compute close upper and lower bounds on $R_{\min}(M)$.

1.1 Lower bounds on $R_{\min}(M)$

Since there are no efficient algorithms to compute $R_{\min}(M)$, several papers (Bafna and Bansal, 2004, 2005; Gusfield and Hickerson, 2004; Hudson and Kaplan, 1985; Myers and Griffiths, 2003; Song and Hein, 2004) have considered efficient computation of lower bounds on $R_{\min}(M)$, and other papers have shown that the use of good lower bounds can address some questions about recombination, such as finding potential recombination hotspots in genomes (Fearnhead *et al*., 2004).

The first lower bound method was developed by Hudson and Kaplan (1985), and it has been widely used both as a lower bound on $R_{\min}(M)$ and as a way to estimate where recombinations may have occurred [e.g. see Clark *et al*. (1998)]. We call this lower bound the HK bound.

Very little progress was made in improving the HK bound until the dissertation of Simon Myers (Myers, 2002; Myers and Griffiths, 2003). There, the haplotype lower bound was developed along with a 'composite method' that dramatically improves the quality of the bound. The composite version of the haplotype bound, which we will describe in detail below, was implemented in a program called RecMin. [RecMin also implements a different bound, called the history bound, but that bound is computable in practice for only small datasets, so when we refer to bounds computed by RecMin, we refer only to those based on haplotype bounds. Bafna and Bansal (2005) have recently made considerable progress on the history bound.] RecMin requires the setting of two parameters (s and w), which affect both the time and the quality of the bounds produced. Using the default parameter settings, RecMin runs very quickly and the bounds produced are typically much larger than the HK bounds (often between two and three times as large). Thus, RecMin is a tremendous improvement over HK, and it has also proved superior to all other known practical lower bound methods (until this paper). However, it is common that better bounds are produced by RecMin by increasing the parameters from their default settings, and there is no theory that specifies a good time/accuracy tradeoff for setting the parameters. Increasing the parameters never reduces the bound produced and often increases it, while typically increasing the running time. RecMin documentation advises the user to experiment with these parameters, gradually increasing them until the bound does not change further. That does not guarantee that the bound could not improve by an additional increase of the parameters.

Since even the default setting of RecMin produces very impressive bounds compared with other practical methods, but higher bounds are typically observed when the parameters are increased, and one does not know where to stop, we define the optimal RecMin problem: compute the lower bound that RecMin would produce (if allowed enough time) when both parameters are set to their maximum possible value. The bound produced is called the optimal RecMin bound.

The program RecMin might produce the optimal RecMin bound when the parameters are set below their maximum values, but the user would not know for sure that the optimal RecMin bound had been obtained. Assurance is possible only if the parameters are set to their maximum value, and in that case RecMin will typically not terminate in a reasonable time unless the problem is very small.

1.2 Main results: practical computation of close upper and lower bounds

In this paper we do several things. First, we introduce an algorithm that uses integer linear programming (ILP) to compute the optimal RecMin bound. Second, with additional ideas that dramatically speed up the ILP, we show through extensive experimentation using simulated and real datasets that this approach computes the optimal RecMin bound faster than RecMin (when RecMin can compute it) and that it can efficiently compute the optimal RecMin bound for problem sizes considered large in current applications (where RecMin fails). Third, we introduce additional ideas that allow the algorithm to find lower bounds even better than the optimal RecMin bound, and we show through extensive experiments that this approach remains practical on problem sizes considered large today. Thus, we provide a practical method that is superior to all other known practical lower bound methods. Fourth, on the upper bound side, we present an efficient algorithm that, when given an input set of sequences M, constructs a network that generates M using recombinations and one mutation per site. [For a formal definition of a network—or an ancestral recombination graph (ARG) in the population genetics literature—see (Gusfield, 2004; Gusfield *et al*., 2004).] The number of recombinations used in the network produced by the algorithm provides an upper bound on $R_{\min}(M)$, but the network itself is of independent interest. Fifth, and most importantly, through extensive experimentation with simulated and real data, we show that the computed upper and lower bounds are frequently very close, and are *equal* with high frequency for a surprisingly large range of data. Thus, with the use of a very effective lower bound and an efficient algorithm for computing upper bounds, this approach allows the efficient, *exact* computation of $R_{\min}(M)$ with high frequency in a large range of data, much larger than with the use of the algorithm in (Song and Hein, 2003). This is an important empirical result that is expected to have a very significant impact. Programs implementing the new algorithms discussed in this paper are available at http://wwwcsif.cs.ucdavis.edu/∼gusfield/lu.html. The lower bounds are computed by the program HapBound, and the upper bounds and networks are computed by the program SHRUB, which also produces code that can be input to an open source program to display the constructed network.

2 HAPLOTYPE AND COMPOSITE BOUNDS

Myers and Griffiths (2003) introduced a lower bound, $h(M)$, called the 'haplotype bound'. Consider the set of sequences M arrayed in a matrix. Then $h(M)$ is the number of distinct rows of M, minus the number of distinct columns of M, minus one. It is easy to establish that this is a lower bound on $R_{\min}(M)$. Simulations using sequences generated by the program MS (Hudson, 2002) show that $h(M)$ by itself is a very poor bound, often a negative number. However, using it with the composite method explained below leads to impressive lower bounds.

2.1 The composite method

Recall that the sites in M have a fixed linear order. Myers and Griffiths (2003) introduced the 'composite method' to combine lower bounds computed (by any method) over a family G of intervals of sites. For an interval $I \in G$, let $M(I)$ denote the matrix M restricted to the sites in I, and let $L(I)$ denote a lower bound computed (somehow) for $R_{\min}(M(I))$. Each $L(I)$ is called a 'local bound'. Then, a composite lower bound on $R_{\min}(M)$ is computed from these local bounds by picking the smallest number of points on the real line, so that for any interval $I \in G$, at least $L(I)$ of the selected points are contained in interval I. The selection of the points can be computed in linear time by a greedy left-to-right sweep of the intervals. In particular, whenever the right endpoint of an interval I is reached, if $z < L(I)$ points in I have already been selected, then select an additional $L(I) - z$ points as far right in I as possible. The correctness of this composite bound, which we leave to the reader, relies on the assumption that the linear ordering of the sites is fixed (as is true when the sites come from a chromosome).

The program RecMin implements the composite method and uses haplotype bounds for the local bounds, but with one added idea. For a subset of sites S (not necessarily contiguous), let $M(S)$ be M restricted to the sites in S, and $h(S)$ be the haplotype bound on $M(S)$. In RecMin, the user specifies two parameters s and w, and RecMin computes $h(S)$ for every subset S with s or fewer sites, provided that no pair of sites in S is more than w positions apart. Then, for every interval I, $L(I)$ is set to the largest $h(S)$, where S is contained in I. The advantage is that $h(S)$ may be larger than $h(M(I))$. These local bounds on intervals are then used in the composite method, as before, to obtain an overall lower bound on $R_{\min}(M)$. RecMin also uses heuristics to avoid the explicit examination of some of the specified subsets. The default settings for RecMin are $s = 8$ and $w = 12$, but we have found that it gives better bounds in reasonable time when we set $s = w = 20$. Overall, RecMin is a very impressive, efficient program for computing lower bounds on $R_{\min}(M)$, and it is far superior to any of the (previous) practical alternatives. However, as noted earlier, for large problem instances of the size of current interest, RecMin cannot compute the optimal RecMin bound and know that the bound has been computed.

3 NEW RESULT: COMPUTING THE OPTIMAL RecMin BOUND

We now begin to develop the new results in this paper, first showing an efficient method (in practice) to compute the optimal RecMin bound.

At the conceptual level, to compute the optimal RecMin bound, it suffices to find for each interval I in M, a subset of sites S^* in I (possibly strictly in I) that maximizes $h(S)$ over every subset of sites, S, in I. Then, for each I, we set $L(I) = h(S^*)$; the composite method uses these local bounds to obtain the optimal RecMin bound. It is easy to write an integer (0/1) linear programming formulation to find $h(S^*)$ for any interval I. That formulation uses one variable per site, one variable per row and one inequality per pair of rows. However, for any interval I, an optimal subset S^* can be found by finding a smallest subset of sites S in I so that every row in $M(S)$ is distinct after removing duplicate rows. [S.R. Myers, (personal communication) also observed this during his thesis work.] This problem is easily cast as a classic set cover problem, where each site j in M can cover a pair of rows $\{i, i'\}$ if $M[i, j] \neq M[i', j]$. Any such instance of set cover is easily formulated as an integer programming problem, with one variable per site and one inequality per pair of rows, but no explicit variables for rows. The result is that often many identical inequalities are produced, and duplicate inequalities can be removed, allowing faster solution of the ILP. After considerable experimentation, we found that this ILP formulation solves much faster than the formulation using row variables. For problem sizes considered large today, we can in general find S^* for a large interval I in somewhere between fractions of a second and several seconds. In our experiments, we have used the GNU ILP solver so that we can release a free version of our software, and we have also used the commercial ILP solver CPLEX. For large problems, CPLEX is significantly faster, but the GNU ILP solver is fast enough to illustrate the practicality of our programs (see Section 3.2 and Appendix). Recently, Bafna and Bansal (2005) also considered finding the haplotype bound using ILP. Unlike us, however, they did not solve their ILP exactly; they instead proposed a greedy algorithm for constructing good estimates.

In the above method to compute the optimal RecMin bound, we explicitly find $L(I')$ for every subinterval I' of $I = [1, m]$.

If M has m sites, then this approach executes $C(m, 2)$ integer programs. This is quadratic in m, rather than exponential in m (the worst case for RecMin), but each computation solves an (expensive) ILP problem. However, the number of intervals we need to examine, and the number of ILP computations, can be considerably reduced. Suppose we find an optimal subset S^* for interval $I = [1, m]$, and the leftmost and rightmost points of S^* are p and q, respectively. Then S^* will also be an optimal subset for any interval I' contained in $[1, m]$ but containing $[p, q]$. There is no need to solve an ILP problem for that I', and further $L(I')$ can be ignored in obtaining the overall composite bound. We can exclude those intervals from further consideration. However, that reasoning does not (yet) exclude the subintervals contained in $[1, q - 1]$, $[p + 1, m]$, $[p + 1, q]$ or $[p, q - 1]$. But we can recursively apply the same idea in each of these four intervals: for each of these four intervals, we find an optimal S^*, and then we recurse on four new subintervals defined from the interval and the span of S^*. Generally, over the entire computation, far fewer than $C(m, 2)$ ILP computations are needed. In our simulations, this simple idea greatly reduces the number of ILP problems that need to be solved. When m and n are about the same size, we typically need to solve ~25% of the $C(m, 2)$ problems, but when m is several times larger than n, the percentage typically falls to <5%. On problems of the size of current interest, HapBound runs in seconds to minutes (see Section 3.2 and Appendix).

3.1 Improving the optimal RecMin bound

The program HapBound has an option (-S) that typically produces a better lower bound than the optimal RecMin bound. The option increases the running time, but not beyond the range of practicality.

In the above method, if S^* is the optimal subset found for interval I, then $L(I)$ is $h(S^*) = n - |S^*| - 1$, where M is assumed to have n distinct rows. But an increase in the local bound for I may increase the composite bound. In the program HapBound, we have implemented a test to determine whether the sequences in $M(S^*)$ can actually be generated with only $h(S^*)$ recombinations. If not, then $L(I)$ can be set to $h(S^*)+1$. We can also test whether that bound is tight, or whether $L(I)$ should be increased again by one.

Given a set of sequences $M(S)$, we say that $M(S)$ is self-derivable (SD) if $M(S)$ can be generated from some ancestral sequence in $M(S)$ using recombinations and one mutation per site in S, and if during the generation of $M(S)$ only sequences in $M(S)$ are generated. Not every set of sequences is SD.

LEMMA 3.1. *Given a subset of sites S in M, the sequences $M(S)$ can be generated using only $h(S)$ recombinations, and one mutation per site of S, if and only if the sequences $M(S)$ are SD.*

We omit the proof, but it is an easy extension of the proof that $h(S)$ is a correct lower bound on $R_{\min}(M(S))$.

3.1.1 Algorithms for the self-derivability test We use an algorithm which runs in polynomial time in n, and exponential time in m (worst case), but is observed to be practical for large datasets of current interest. It works as follows. When only recombinations are permitted, the test for self-derivability of a set of sequences $M(S)$ from a fixed pair of ancestral sequences has an efficient solution (Kececioglu and Gusfield, 1998). To start, a 'reached set' of sequences consists of the ancestral sequences alone. Then, at each step, we try to expand the reached set by finding a pair of sequences (A, B) in the reached set that can recombine to create a sequence C that is in $M(S)$ but not in the reached set. Repeat until either the reached set contains all of $M(S)$ or until no further expansion is possible. This approach can be sped up by preprocessing (Kececioglu and Gusfield, 1998), but clearly it takes only polynomial time.

The above method must be modified when there is only one ancestral sequence and site mutations are allowed. However, if $M(S)$ is SD, then for every site $i \in S$, during any self-derivation of $M(S)$, a mutation at site i must occur in some sequence in $M(S)$ and generate another sequence in $M(S)$. Therefore those two sequences differ by exactly one site. [It also follows that if there is a site i such that no pair of sequences in $M(S)$ differs only at i, then $M(S)$ is not SD.] Let MUT_i be the set of sequence pairs that differ at exactly site i. We modify the self-derivability test for recombinations alone by now starting the reached set with a single ancestral sequence and allowing the reached set to expand either by a recombination, as before, or by the addition of a sequence B in $M(S)$ if a sequence A is already in the reached set and (A, B) is in MUT_i for some i, where no prior expansion of the reached set used a pair in MUT_i. That is, if $M(S)$ is SD, then there is a generation of $M(S)$ where for every site i in S, exactly one pair in MUT_i is used to expand the reached set. The self-derivability algorithm simply tries all sequences in $M(S)$ as the ancestral sequence, and all ways to choose exactly one pair from each MUT_i. The number of choices is $n \times \prod_{i \in S} |MUT_i|$, which is bounded by $[n/2]^m$ but is generally much less. Further, some combinations of choices can be immediately ruled out and additional effective heuristics reduce the number of choices (details omitted here).

The self-derivability test determines whether a local haplotype bound is tight, or whether the local bound should be increased by one. To test whether a local bound should be increased by two, we note that this is the case when $M(S)$ is not SD and including one new sequence not in $M(S)$ is also not sufficient to allow the expanded set to be SD. If one new sequence did allow the expanded set to be SD, then the new sequence must either be the recombination of two sequences in $M(S)$ or differ from a sequence in $M(S)$ at exactly one site in S. There are only a polynomial number of such candidate sequences, and so we can efficiently generate each new candidate sequence in turn, and test the resulting set for self-derivability. If the sequences are SD in none of these tests, then the local bound should be increased by two, otherwise just by one. We can continue in this way to determine whether the local bound should be increased by three, etc. However, the time for each test increases too rapidly for practical implementation.

The program HapBound, with option -S, tests each optimal subset S^* it finds to see whether $M(S^*)$ is SD, and if not, to see whether the local bound based on S^* should be increased by one or by two. For the datasets that we have examined, the extra computation time for the -S option does not reduce the practicality of the algorithm and frequently results in a lower bound on $R_{\min}(M)$ that is higher than the optimal RecMin bound (comprehensive test results are shown in the Appendix section).

3.2 Lower bounds for the LPL and ADH datasets

As an illustration, we examined the LPL data from Clark *et al.* (1998), containing 88 rows and (coincidently) 88 sites before removing sites with missing or non-SNP data. That dataset was also examined by Myers and Griffiths (2003). (The first paper uses 42 sites from the full data in the recombination analysis, and the second paper uses 48 sites. For clearer comparison, we also use the same 48 sites.)

Using CPLEX to solve the ILP problems, HapBound computed the optimal RecMin bound of 75 in 31 s, and HapBound -S computed a bound of 78 in 1643 s, on a 2 GHz machine. Using the GNU ILP solver on the same machine, the times were 871 and 3326 s, respectively. The program RecMin with the default settings produced a bound of 59 in 3 s. It found the optimal RecMin bound of 75 with parameters $s = w = 25$ in 7944 s. As mentioned earlier, a user would not know that this was the optimal RecMin bound and we set $s = w = 48$, but RecMin did not finish within 5 days of execution. Interestingly, the analysis by Myers and Griffiths (2003), based on RecMin, reports a lower bound on $R_{\min}(M)$ for these data of only 70, rather than 75. This is due to running RecMin with parameters that are too low (S.R. Myers, personal communication). This illustrates a central point of this section, that with RecMin one does not know which parameter settings are high enough, and illustrates the utility of program HapBound. For comparison, the HK bound is 22.

Th program HapBound has another option (-M) that is also based on the self-derivability test but is more time consuming, and it typically increases the lower bound by only a small amount. Still, for small datasets, the program HapBound -M has produced higher lower bounds than any other lower bound program. For example, the 'benchmark' dataset by Kreitman (1983) has 11 sequences and 43 sites. Song and Hein (2003) established that $R_{\min}(M)$ is exactly 7 for these data. The lower bound method by Song and Hein (2004) has never been implemented, but was manually applied to these data, producing a lower bound of 7. HapBound -M ran in ~3 s on these data and also computed the lower bound of 7. The improved history lower bound of Bafna and Bansal (2005) also produces 7. All other implemented lower bound methods that we know of (nine in total) produce lower bounds of 5 or 6.

4 UPPER BOUND

In this section, we describe a method of computing upper bounds on $R_{\min}(M)$.

4.1 Definition of the upper bound

Given a $k \times \ell$ matrix A of binary sequences, the recombination weight $w(r|A - r)$ associated with row r is defined as the minimum number of recombinations required to derive that row from some other rows in A. More formally, $w(r|A - r)$ is the minimum value of d such that row r can be written as

$$A_{i_1,1} \cdots A_{i_1,j_1} A_{i_2,j_1+1} \cdots A_{i_d,j_d} A_{i_{d+1},j_d+1} \cdots A_{i_{d+1},\ell}, \qquad (1)$$

where $i_1, \ldots, i_{d+1} \in \{1, \ldots, k\} - \{r\}$. We point out that some of the row indices $i_1, \ldots, i_{d+1}$ appearing in (1) may be the same, and that $w(r|A - r)$ can be computed very efficiently using a simple algorithm, which we describe in Section 4.2.

A column is called non-informative if it contains fewer than two 0s or fewer than two 1s. For a given dataset M, our upper bound on $R_{\min}(M)$ uses the following procedure[1]:

Step 1. Set $W = 0$.

Step 2. Collapse identical rows into one and remove non-informative columns. Repeat these operations until neither is possible.

Step 3. Let A denote the data passed on from Step 2. If A is empty, then stop. Otherwise, remove a row from A, say the r-th row, set $W \leftarrow W + w(r|A - r)$, and go back to Step 2.

Note that, because non-informative columns get removed in Step 2, it is always possible to write a row in A in terms of other rows, and therefore $w(r|A - r)$ is well-defined. Associated with the execution of the above procedure is a sequence (or a history) H of row removals considered in Step 3. Let $W_H(M)$ denote the final value of W for that sequence of row removals. Then, our full upper bound on $R_{\min}(M)$ is defined as

$$R_{\mathrm{u}}(M) := \min_{H \in \mathcal{H}} W_H(M), \qquad (2)$$

[1]We note that the procedure defined here is very similar to the procedure used in the aforementioned history lower bound (Myers and Griffiths, 2003). One difference between the two procedures is that we set $W \leftarrow W + w(r|A - r)$ in Step 3, whereas the history lower bound method sets $W \leftarrow W + 1$. However, even a single execution of our procedure will yield an upper bound, whereas a lower bound can be obtained only if one finds the minimum possible value over all possible sequences of row removals. This is a crucial difference, and unlike our upper bound computation, there is no 'fast' version of the history lower bound. In our upper bound approach, we try to explore many possible sequences of removals in order to improve the quality of the upper bound, but even a single execution of the procedure yields an upper bound. Another difference is that our upper bound needs to be computed for the entire data only, whereas the history lower bound method is a composite method, combining the local history bounds from all subintervals.

where $\mathcal{H}$ denotes the set of all possible sequences of row removals. If more than 30 or so sequences remain after the first execution of Step 2, finding the full upper bound can take a very long time, even if branch and bound techniques are used. In such cases, the following modified step can be used in lieu of Step 3 to obtain an efficient fast upper bound:

Step $3'$. When removing a row, choose one of the rows which minimize the sum $w(r|A-r) + L(A-r)$, where $L(A-r)$ denotes some efficient lower bound for $A-r$. Everything else remains the same as in Step 3.

Our current implementation—called SHRUB, an acronym for 'simulated history recombination upper bound'—can use the HK bound or an approximate haplotype bound for $L(A-r)$.

Our experimental study shows that fast and full upper bounds are either very close or equal quite frequently. In SHRUB, the user can choose whether the full or a fast upper bound should be computed. If the former option is chosen, a fast upper bound is first computed, which is then used in a full branch and bound search. If the latter option is chosen, the user can also specify the maximum number K of different row removals to be tried in Step $3'$. When K is less than the total number of rows that satisfy the condition in Step $3'$, SHRUB randomly chooses K rows from such rows. Because of the randomization idea just mentioned, different runs of SHRUB with a small value of K generally give different upper bounds. From our experience, however, it seems that setting K to a value between 3 and 5 yields bounds that are very close to the $K = \infty$ bounds. For $K > 1$, the algorithm uses the branch and bound ideas described in Section 4.3 to limit its search. Note that the algorithm runs in polynomial time for $K = 1$. When choosing $K = 1$, the user may also specify how many runs should be performed; if more than one is chosen, a run may terminate before completion if it is found that it cannot improve on the current best bound. This algorithm also runs in polynomial time and yields reasonably good upper bounds.

That $R_u(M)$ is an upper bound on $R_{\min}(M)$ follows from the fact that every execution of the above steps generates, backwards in time, an ARG consistent with the given data M. In fact, our software can generate an explicit representation of a possible evolutionary history with as many recombinations as that found by our method. In regard to constructing an ARG backwards in time, collapsing two identical rows corresponds to creating a new node where two edges meet (i.e. a coalescent event); removing a non-informative column corresponds to a mutation event; and removing the r-th row in Step 3 corresponds to creating $w(r|A-r)$ recombination and $w(r|A-r)+1$ coalescent nodes, which get joined between themselves and to other nodes in an appropriate way. Note that the specific recombination and coalescent events in Step 3 depend on how the removed row can be written in terms of some other rows in the computation of the recombination weight.

As a concrete example, consider the toy dataset shown in Figure 1. In Step 2 of the procedure, after the sixth column

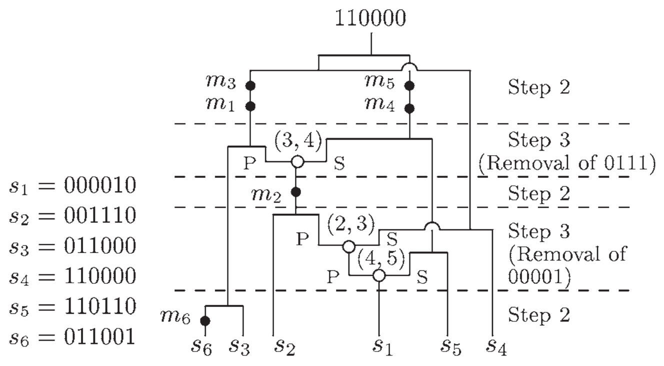

Fig. 1. An ARG for a toy dataset. This corresponds to a particular execution of the procedure described in Section 4.1. Closed circles represent mutation events and open circles recombination events. A mutation event at the i-th site is denoted by m_i. For each recombination event, we use (i, j) to indicate that the corresponding crossover point lies between columns i and j; the recombinant sequence gets its prefix (suffix) from the edge labeled 'P' ('S').

is removed, rows 3 and 6 become identical, thus allowing us to remove one of them. No further removal of columns or collapsing of rows is possible, so we proceed to Step 3. In Step 3, any of the remaining five rows may be removed. The ARG shown in Figure 1 corresponds to removing the row with 00001. The recombination weight for this row is 2. Note that there is more than one way to write this row in terms of other rows. The history shown in Figure 1 corresponds to the case where 00001 is derived from 00111, 11000 and 11011. When 00001 is removed, the second column of the matrix is now non-informative and therefore can be removed. No further reduction can be done, so we again carry out Step 3. The ARG under consideration corresponds to removing the row with 0111. Its associated recombination weight is 1, and it is derived from 0100 and 1011. The next execution of Step 2 completely eliminates the matrix, so the procedure terminates.

4.2 Computing the recombination weight

The recombination weight $w(r|D-r)$ can be computed efficiently by finding the number of consecutive right-maximal intervals, which we now define. For non-negative integers a and b satisfying $a \le b$, let $I(a, b)$ denote the integral interval $\{a, a+1, \dots, b-1, b\}$. Let $(r_1, r_2, \dots, r_\ell)$ be the r-th row in a $k \times \ell$ matrix $A = (A_{ij})$. We say that an interval $I(a, b)$ is right-maximal with respect to r in A if there exists a row i in $A - r$ satisfying $r_j = A_{ij}, \forall j \in I(a, b)$, but there exists no row in $A - r$ satisfying $r_j = A_{ij}, \forall j \in I(a, b+1)$.

LEMMA 4.2. *Suppose that there are $c+1$ consecutive intervals from 1 to ℓ that are all right-maximal with respect to r in A. Then $w(r|A-r) = c$.*

PROOF. Suppose that $I(1, m_1)$, $I(m_1+1, m_2), \dots,$ $I(m_{c-1}+1, m_c)$, $I(m_c+1, \ell)$ are such intervals. Since

$I(1, m_1)$ is right-maximal, $j_1 \leq m_1$ in every sequence of the form shown in (1). Similarly, since $I(m_1+1, m_2)$ is right-maximal, $j_2 \leq m_2$ in every sequence of the form shown in (1) and so on. Therefore, $d \leq c$ for every sequence of the form shown in (1).

4.3 Branch and bound

SHRUB uses two ideas to accelerate its search. The first one is perhaps somewhat obvious and the second one perhaps rather subtle. On some examples we analyzed, the second idea proved to be very powerful, sometimes accelerating the search by hundreds of times.

Looking forward. Let $L(A)$ denote an efficient lower bound for data A. Suppose that C is the current minimum value of $W_H(M)$ and that we have created A, with W as its thus far accumulated recombination weight. If $W + L(A) \geq C$, then we can backtrack the current search path.

Looking backward. There can often be more than one sequence of row removals that lead to the same matrix A. For any reduced matrix A, let W_A be the current value of accumulated recombination weights that is minimal over all hitherto considered search paths producing A. When considering a new sequence of row removals that creates A, we can backtrack if its accumulated recombination weight W_{new} is greater than or equal to W_A. If $W_{\text{new}} < W_A$, then we can set $W_A \leftarrow W_{\text{new}}$ and continue on that search path.

The ideas sketched above are the key to the applicability of our upper bound method. Combined with the aforementioned algorithm for fast upper bounds, branch and bound methods seem very effective for analyzing complex datasets. The provable worst-case running time for finding the full upper bound (2) is $O(2^n)$ rather than $\Theta(n!)$ as might be thought. For many problems of current interest, this difference allows practical computation. For example, we could obtain the optimal upper bound for a dataset containing 40 sequences and 100 sites in a few seconds; without branch and bound methods, an execution of the algorithm for computing the full upper bound did not finish even after 1 day.

5 APPLICATIONS OF CLOSE LOWER AND UPPER BOUNDS

5.1 Kreitman's ADH data

As discussed in Section 3.2, HapBound -M yields a lower bound of 7 for the ADH data of Kreitman (1983). Although the exact method described by Song and Hein (2003) always yields the minimum number, the currently available software can handle at most nine sequences after data reduction, and it generally requires large memory and long CPU time; for Kreitman's data, it required ~1.5 GB of memory and took ~30 min on 1.26 GHz Pentium III processor to produce the minimum number 7.

Table 1. Lower and upper bounds for the LPL data, partitioned into three site regions

	Our new methods			RecMin		
Population	Region 1	Region 2	Region 3	Region 1	Region 2	Region 3
Jackson	11 (13)	10 (10)	13 (16)	10	9	12
North Karelia	2 (2)	15 (17)	8 (10)	2	13	7
Rochester	1 (1)	14 (14)	8 (8)	1	12	7
All	13 (14)	21 (22)	25 (31)	12	21	22

Numbers in parentheses are upper bounds. Lower bounds on the left-hand side were computed using HapBound -S -M.

In contrast, the current implementation of our upper bound method took only a fraction of a second to analyze Kreitman's data, while also giving 7 for the number of recombination events. Independently of all other methods, the fact that our new lower and upper bounds agree implies that 7 is the minimum number of recombination events for Kreitman's data, and the ARG produced by SHRUB corresponds to a most parsimonious history.

5.2 The human LPL data

We applied our methods to the human LPL data (Nickerson *et al.*, 1998). These sequences were sampled from three populations—namely, Jackson, North Karelia and Rochester. In our analysis, we removed sites with missing or non-SNP data. [That is, we ignored insertions/deletions, unphased sites and sites with missing data. This is the treatment of the data that was used by Nickerson *et al.* (1998) but is a different treatment than in Section 3.2.] Following Myers and Griffiths, we partitioned the LPL data into three site regions [cf. Table 5 of Myers and Griffiths (2003)]. It has been suggested that region 2 corresponds to a recombination hotspot (Templeton *et al.*, 2000).

Shown on the left-hand side of Table 1 is a summary of our new lower and upper bounds for the three site regions. We considered the three populations separately as well as together. HapBound -S and HapBound -S -M produced similar lower bounds. The only difference was in region 2 of the Jackson population, where HapBound -S produced 9, whereas HapBound -S -M produced 10. For upper bounds, we used fast methods as well as the full method for most cases; only a fast upper bound method was used for regions 2 and 3 when all populations were considered. When populations were considered separately, most run times varied from a fraction of a second to a few minutes. Our lower and upper bounds are generally quite close. In particular, they match perfectly for the Rochester population.

Optimal RecMin bounds are shown on the right-hand side of Table 1 for comparison. This table differs from Table 5 of Myers and Griffiths (2003), because Myers and Griffiths did not remove insertion/deletion sites when they performed their analysis.

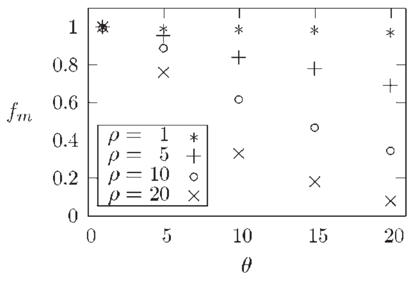

Fig. 2. The frequency f_m of the time that lower and upper bounds match for 25 sequences. We used 1000 simulated datasets for each pair of θ and ρ, the mutation and recombination rates, respectively. For $\theta, \rho \leq 5$, the two bounds agree more than 95% of the time.

5.3 Simulated data

To study more extensively how often lower and upper bounds agree, we used the program MS (Hudson, 2002) to generate simulated datasets. We used HapBound -S -M and the full upper bound method for this study. For varying values of the scaled mutation and recombination rates—denoted θ and ρ, respectively—the frequency f_m of having matching lower and upper bounds is shown in Figure 2 for 25 sequences. Each point shown in this figure comes from analyzing 1000 simulated datasets. Note that the lower and upper bounds agree quite often. In particular, for $\theta \leq 5$ and $\rho \leq 5$, the two bounds agree more than 95% of the time. This is a major accomplishment, as there currently exists no other method that can find $R_{\min}$ for more than nine sequences after data reduction. As the figure illustrates, the match frequency begins to fall off as θ or ρ increases. It would be interesting to find out whether this is largely due to the inaccuracy of the lower bound or the upper bound, or perhaps both.

Suppose that, out of k simulated datasets, d datasets had different lower and upper bounds, respectively denoted L_i and U_i, $i = 1, \ldots, d$. To examine by how much our lower and upper bounds differ when they do differ, we computed the following average of their ratio: $\lambda = \frac{1}{d}\sum_{i=1}^{d} L_i/U_i$. For the simulated datasets used in Figure 2, λ was as shown in Figure 3. It is interesting that the difference between our lower and upper bounds captured by this measure does not depend so much on θ and ρ. Although the numerical difference between lower and upper bounds is expected to grow as θ or ρ increases, their ratio seems more stable. We also remark that, for both lower and upper bounds, the average number of recombinations found when the two bounds do not match was much higher (by about a factor of 2 or 3 for most parameter values) than the average when they match.

We also studied the dependence of f_m on the number n of sequences. Our results are summarized in Figure 4. Again, each point in the figure is the result of analyzing 1000 simulated datasets. It is a significant finding that lower and upper bounds agree quite frequently even for $n = 100$. It seems that the match frequency does not depend on n as much as on θ or ρ.

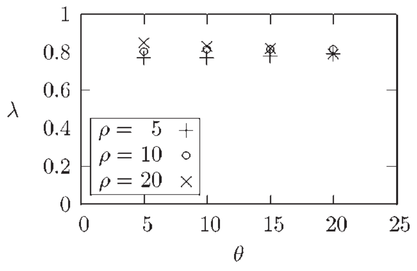

Fig. 3. Here, λ denotes the average of the ratio of the lower bound to the upper bound when they do not match. This plot is for the same simulated datasets as in Figure 2. Note that λ does not depend so much on θ and ρ, the mutation and recombination rates, respectively.

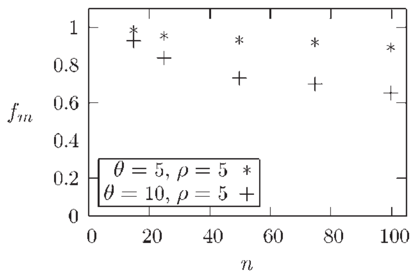

Fig. 4. The frequency f_m of the time that lower and upper bounds match for a varying number n of sequences. The parameters θ and ρ denote the scaled mutation and recombination rates, respectively. Compared with Figure 2, this plot shows that the match frequency does not depend on n as much as it does on θ or ρ.

ACKNOWLEDGEMENTS

We would like to thank Andrew Clark for providing us with the LPL data. We would also like to thank Charles Langley and Simon Myers for useful discussion. This research is supported by NSF EIA-0220154.

REFERENCES

Bafna,V. and Bansal,V. (2004) The number of recombination events in a sample history: conflict graph and lower bounds. *IEEE/ACM Trans. Comput. Biol. Bioinf.*, **1**, 78–90.

Bafna,V. and Bansal,V. (2005) Improved recombination lower bounds for haplotype data. In *Proceedings of RECOMB 2005*, in press.

Clark,A.G., Weiss,K.M., Nickerson,D.A., Taylor,S.L., Buchanan,A., Stengard,J., Salomaa,V., Vartiainen,E., Perola,M., Boerwinkle,E. and Sing,C.E. (1998) Haplotype structure and population genetic inferences from nucleotide-sequence variation in human lipoprotein lipase. *Am. J. Hum. Genet.*, **63**, 595–612.

Fearnhead,P., Harding,R.M., Schneider,J.A., Myers,S. and Donnelly,P. (2004) Application of coalescent methods to reveal fine scale rate variation and recombination hotspots. *Genetics*, **167**, 2067–2081.

Gusfield,D. (2004) Optimal, efficient reconstruction of Root-unknown phylogenetic networks with constrained recombination. *Technical Report*, Department of Computer Science, University of California, Davis, CA.

Gusfield,D. and Hickerson,D. (2004) A new lower bound on the number of needed recombination nodes in both unrooted and rooted phylogenetic networks. *Technical Report UCD-ECS-06*, University of California, Davis, CA.

Gusfield,D., Eddhu,S. and Langley,C. (2004) Optimal, efficient reconstruction of phylogenetic networks with constrained recombination. *J. Bioinf. Comput. Biol.*, **2**, 173–213.

Hein,J. (1990) Reconstructing evolution of sequences subject to recombination using parsimony. *Math. Biosci.*, **98**, 185–200.

Hein,J. (1993) A heuristic method to reconstruct the history of sequences subject to recombination. *J. Mol. Evol.*, **36**, 396–405.

Hudson,R. (2002) Generating samples under the Wright–Fisher neutral model of genetic variation. *Bioinformatics*, **18**, 337–338.

Hudson,R. and Kaplan,N. (1985) Statistical properties of the number of recombination events in the history of a sample of DNA sequences. *Genetics*, **111**, 147–164.

Kececioglu,J.D. and Gusfield,D. (1998) Reconstructing a history of recombinations from a set of sequences. *Discrete Appl. Math.*, **88**, 239–260.

Kreitman,M. (1983) Nucleotide polymorphism at the alcohol dehydrogenase locus of *Drosophila melanogaster. Nature*, **304**, 412–417.

McVean,G.A.T., Myers,S., Hunt,S., Deloukas,P., Bentley,D.R. and Donnelly,P. (2004) The fine-scale structure of recombination rate variation in the human genome. *Science*, **304**, 581–584.

Myers,S.R. (2002) The detection of recombination events using DNA sequence data. PhD Thesis, Department of Statistics, University of Oxford, Oxford.

Myers,S.R. and Griffiths,R.C. (2003) Bounds on the minimum number of recombination events in a sample history. *Genetics*, **163**, 375–394.

Nickerson,D.A., Taylor,S.L., Weiss,K.M., Clark,A.G., Hutchinson,R.G., Stengard,J., Salomaa,V., Vartiainen,E., Boerwinkle,E. and Sing,C.F. (1998) DNA sequence diversity in a 9.7-kb region of the human lipoprotein lipase gene. *Nat. Genet.*, **19**, 233–240.

Song,Y.S. and Hein,J. (2003) Parsimonious reconstruction of sequence evolution and haplotype blocks: finding the minimum number of recombination events. In Benson,G. and Page,R. (eds), In *Proceedings of the Third International Workshop on Alogarithms in Bioinfomatics (WABI 2003)*, Budapest, Hungary, September 15–20, *LNCS* 2812. Springer-Verlag, NY. pp. 287–302.

Song,Y.S. and Hein,J. (2004) On the minimum number of recombination events in the evolutionary history of DNA sequences. *J. Math. Biol.*, **48**, 160–186.

Templeton,A.R., Clark,A.G., Weiss,K.M., Nickerson,D.A., Boerwinkle,E. and Sing,C.F. (2000) Recombinational and mutational hotspots within the human lipoprotein lipase gene. *Am. J. Hum. Genet.*, **66**, 69–83.

Wang,L., Zhang,K. and Zhang,L. (2001) Perfect phylogenetic networks with recombination. *J. Comput. Biol.*, **8**, 69–78.

Table A1. Average ratio of the running time of RecMin to that of HapBound.

	$\theta = \rho$		
n	10	20	30
25	1.4	3.9	14.0
50	1.7	41.7	>111.3
75	11.2	26.5	>103.6

As usual, θ and ρ are scaled mutation and recombination rates, respectively, and n denotes the number of sequences.

Table A2. An example, with 25 sequences and 376 sites, for which RecMin fails to output the optimal RecMin bound after a long time

Program/options	Bound	Time
RecMin	36	1 s
RecMin -s 30 -w 30	42	3 min 25 s
RecMin -s 35 -w 35	43	24 min 2 s
RecMin -s 40 -w 40	43	2 h 9 min 4 s
RecMin -s 45 -w 45	43	10 h 20 min 59 s
HapBound	44	2 min 59 s
HapBound -S	48	39 min 30 s

A1 APPENDIX

A1.1 The running times of HapBound and RecMin

We compared the running times of RecMin and HapBound on simulated datasets, generated using MS (Hudson, 2002). A 2 GHz Pentium PC was used for this study and the GNU ILP solver was used for running HapBound. To get the running times of RecMin, we incremented the parameters s and w until RecMin computed the optimal RecMin bound, and then we increased those parameters by five each. The purpose of the last increase was to recreate what a user might do (increasing parameters until no further change in the bound is observed). We report only problem instances where RecMin was able to compute the optimal RecMin bound in a reasonable time. Average ratios of the running time of RecMin to that of HapBound are shown in Table A1.

For most datasets of interest, RecMin may take a very long time to compute the optimal RecMin bound when parameters are set to their maximum values, but it might compute the same bound in a reasonable time using smaller parameter settings. However, we note here that, as n or θ and ρ increase, we have frequently seen cases where RecMin cannot produce its optimal bound even after many hours of computation. One such a case, with 25 sequences and $\theta = \rho = 100$, is shown in Table A2. RecMin was not able to produce a bound equal to the optimal RecMin bound even after more than 10 h of computation. In contrast, HapBound found the optimal RecMin bound in ~3 min and HapBound -S produced a better bound in <40 min.

Table A3. The percentage of the time that HapBound -S gives a bound sharper than the optimal RecMin bound for 25 sequences

	ρ			
θ	1	5	10	20
1	0.0	0.4	0.5	1.5
5	0.7	4.0	10.4	27.0
10	1.4	9.2	17.8	40.4
20	1.4	10.5	27.8	45.4

A1.2 The bounds of HapBound -S and HapBound

We also studied how often HapBound -S computes a bound sharper than the optimal RecMin bound. Shown in Table A3 is a summary of comparison for 25 sequences. Note that as θ or ρ increases, HapBound -S becomes more and more likely to produce a bound that is sharper than the optimal RecMin bound, determined using HapBound without any option.