Contents

Chapter 1

First Lower Bounds on *Rmin*

In the last chapter we introduced $Rmin(M)$, the minimum number of recombination nodes used in any ARG to derive the set of sequences M, and we noted that the problem of computing *Rmin* is known to be NP-hard. Hence no provably-correct, worst-case polynomial-time algorithm is known for computing *Rmin*, and we do not expect there will be one. However, several worst-case polynomial-time algorithms have been developed that compute empirically-good *lower bounds* on *Rmin*, and there are several other lower-bound methods whose worst case time is not polynomially-bounded, but are fast in practice. Some of the lower bounds apply only to ARGs, and some of them apply more generally to other kinds of reticulate networks. The literature on computing lower bounds on *Rmin* is contained in many papers, including [2, 3, 1, 9, 13, 19, 25, 10, 27, 33, 4].

Using methods that construct ARGs (which we will discuss in later chapters), we can compare the number of recombination nodes used in those ARGs to the lower bounds given by the methods. We will see that some of the lower bound methods return values equal to *Rmin* or very close to *Rmin* on many datasets. Moreover, there are biological questions concerning recombination (for example, finding recombination *hotspots* [7]) that have been successfully addressed using lower bounds on *Rmin* rather than using *Rmin* itself [30, 1, 3, 31, 32]. Therefore, the development and study of lower bounds on *Rmin* has been a valuable contribution to the algorithmic understanding of recombination and reticulate networks.

In this chapter we discuss several specific lower bounds on *Rmin*, and a very effective general method for amplifying lower bounds. These bounds are either computable in worst-case polynomial time, or have been shown to be computable in practical time on datasets of current interest in biology. In later chapters we will discuss three additional lower bounds. These are deferred because additional background must be developed first, and because two of the lower bounds are less efficiently computed than the ones discussed in this chapter, although they give somewhat higher bounds.

Before discussing any specific bounds, we clarify several different ways that the term *lower bound* is used.

Definition Given a specific binary matrix M a *lower bound on* $Rmin(M)$ is a *number* that is less than or equal to $Rmin(M)$. More generally, a *lower bound on* $Rmin$ is a *function* of M which is guaranteed to be a lower bound on $Rmin(M)$ for any M. A *lower bound algorithm* for $Rmin$ is an algorithm that computes a lower bound on $Rmin$.

In practice, people often use the term 'lower bound' for each of these three different meanings, letting the context indicate the intended use.

The first *trivial* lower bound, of one, is given by the Four-Gametes Theorem: For any binary matrix M, $Rmin(M) \geq 1$ if and only if there is some incompatible pair of sites in M.

1.1 The first combinatorial observation

This book develops and exploits combinatorial structure that an ARG must posses when it derives a set of sequences M. Here we develop the first combinatorial observation, leading to a simple, but important fact.

Definition In an ARG $\mathcal{N}$, let v be a node that has two directed paths out of it that meet at some recombination node x. Those two paths together define a *recombination cycle* Q. Node v is called the *coalescent node* of cycle Q. For example, the nodes labeled 00000 and 01100 in Figure **??** are the coalescent nodes of the two recombination cycles.

Lemma 1.1.1 *Let $\mathcal{N}$ be any ARG that derives a set of sequences M, and let c and d be two incompatible sites in M. Then the edges, e_c and e_d, in $\mathcal{N}$ that are labeled with sites c and d must be contained together in a common recombination cycle in $\mathcal{N}$.*

Proof Without loss of generality, assume that the states of both c and d are 0 at the root of $\mathcal{N}$. Suppose first that e_d is not in the subnetwork of $\mathcal{N}$ below e_c, and e_c is not in the subnetwork below e_d. Note that those two subnetworks cannot have a node v in common, for then e_c and e_d would be contained together in a recombination cycle with the recombination node v. Therefore, the c, d state-pair of 1,1 cannot appear at any node in $\mathcal{N}$, and hence no sequence in M has the c, d state-pair of 1,1.

Now suppose that e_d is in the subnetwork of e_c. The c, d state-pair at the head of the edge labeled c is 1,0. If, through recombination, the state of c is 0 at the tail of the edge labeled d, then the c, d state-pair of 0,1 will be created, but the state-pair of 1,1 will not be possible. Similarly, if the state of c is 1 at the tail of the edge labeled d, then the c, d state-pair of 1,1 will be created, but the state-pair of 0,1 will not be possible. The case where e_c is in the subnetwork

of e_d is symmetric, and omitted. In all cases, only three c, d state-pairs can be created in $\mathcal{N}$, and so M cannot have all four binary combinations in sites c, d, so sites c and d must be compatible, a contradiction. ■

1.2 HK: The first non-trivial lower bound

The first published, and most basic, lower bound on *Rmin* is due to R. Hudson and N. Kaplan [13], and it is refered to as the *HK bound*. The bound is obtained as follows:

> Given an n by m binary matrix M, find all of the *incompatible* pairs of sites in M.
>
> Consider the m sites of M to be *integer* points $1...m$ on the real line, and find a *minimum-sized* set of *non-integer* points, called $R^*(M)$, so that for every pair of incompatible sites (p, q) in M, there is at least one point in $R^*(M)$ that is (strictly) between p and q. The quantity $|R^*(M)|$ is called the *HK-bound* for M, and is denoted *HK(M)*.

For example, in the data shown in Figure **??**, site pairs (1,3), (1,4) and (2,5) are incompatible. A single point strictly between sites 2 and 3 intersects the three intervals defined by those incompatible pairs, giving an HK-bound of one. Of course, a lower bound of one is not of great interest, because by the Four-Gametes Theorem, the existence of any incompatible pair means that there must be at least one recombination node in any ARG that derives the sequences. However, in general, the HK-bound is more informative than in this specific example.

Theorem 1.2.1 *The HK-bound is a lower bound on Rmin. That is, HK(M)* $\leq$ *Rmin*(M)*, for any* M*.*

Before proving Theorem 1.2.1, we establish an important lemma.

Lemma 1.2.1 *For every pair of incompatible sites* (p, q) *in* M*, where* $p < q$*, every ARG that derives* M *must have a recombination node whose crossover-index is greater than* p *and less than or equal to* q*, that is, in the interval* $(p, q]$ *where the left end is open and the right end is closed. So, in the underlying chromosome, the recombination breakpoint must occur somewhere in the interval between sites* p *and* q*.*

Proof For contradiction, suppose there is an ARG $\mathcal{N}$ that derives M, where every recombination node x in $\mathcal{N}$ has a crossover-index b_x such that $b_x \leq p$ or $b_x > q$.

Let $M(p,q)$ denote the sequences in M restricted to the sites p and q. We will modify $\mathcal{N}$ to obtain a perfect phylogeny T for the set of sequences $M(p,q)$. To do this, first remove all labels (mutations) on edges other than p and q. Next, at each recombination node x in $\mathcal{N}$, remove *exactly one* of the incoming edges as follows. If the crossover-index b_x at x is less than or equal to p, remove the P-labeled edge into x and retain the S-labeled edge. Conversely, if $b_x > q$, remove the S-labeled edge, and retain the P-labeled edge into x. See Figures 1.1 and 1.2. The resulting graph is now a directed tree T that derives the sequences in $M(p,q)$. There is one edge in T labeled with p and one edge in T labeled with q, so T is a perfect phylogeny for M with the same root node and ancestral sequence as $\mathcal{N}$ has. But, by Theorem **??**, $M(p,q)$ can have a perfect phylogeny only if p and q are compatible, contradicting the assumption that p and q are incompatible. Hence, for any pair of incompatible sites in M, any ARG that derives M must have a recombination node with crossover-index b_x where $p < b_x \leq q$. ■

Using Lemma 1.2.1, we can now prove Theorem 1.2.1.

Proof of Theorem 1.2.1 Let $\mathcal{N}$ be an ARG that derives M, and let B be the set of crossover-indices associated with recombination nodes in $\mathcal{N}$. For each recombination node x, let point $p_x = b_x - \epsilon$, where $0 < \epsilon < 1$. By Lemma 1.2.1, the result is a set of non-integer points B' such that for every pair of incompatible sites (p,q) in M, there is at least one point in B' (strictly) between p and q. Therefore $|B| = |B'| \geq |R^*(M)|$, and in particular, when $\mathcal{N}$ is a MinARG using $Rmin(M)$ recombination nodes, we see that $Rmin(M) \geq |R^*(M)|$, so the HK-bound is a lower bound on $Rmin$. ■

Before going on, we note the relationship of Theorem 1.2.1 to Lemma 1.1.1. Lemma 1.1.1 showed that if sites p and q are incompatible in M, then they must be contained together in some recombination cycle in any ARG $\mathcal{N}$ that derives M. Theorem 1.2.1 shows that if p and q are incompatible, then any ARG that derives M must have a recombination node x whose crossover-index is in the interval $(p,q]$. In fact, both of these facts must hold simultaneously.

Lemma 1.2.2 *If c and d are incompatible in M, then they must be contained together in some recombination cycle in $\mathcal{N}$ whose recombination node has crossover-index in the interval $(p,q]$.*

We leave the proof to the reader. For CS 224 - Homework problem

The root-known version of the HK-bound The discussion of the HK-bound so far did not assume that a fixed ancestral sequence was known. If a fixed ancestral sequence S_r is given, a lower bound on $Rmin_{S_r}(M)$ can be computed simply by adding S_r to M and computing the resulting bound on $Rmin$.

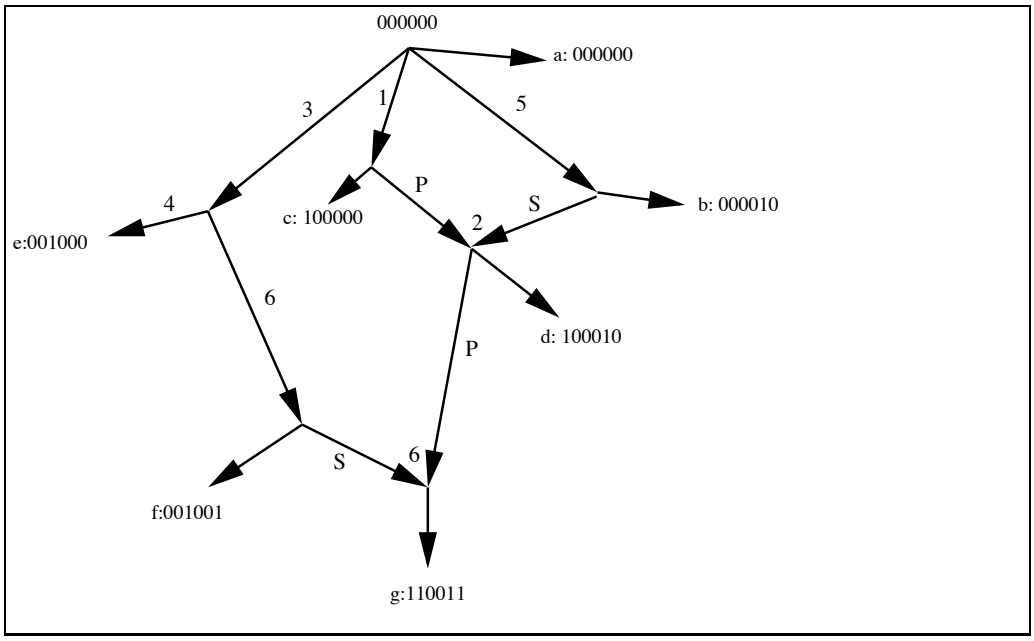


Figure 1.1: Network to illustrate the proof Theorem 1.2.1. For contradiction, sites 3 and 5 are assumed to be incompatible, but as shown, there is no recombination node x with crossover-index b_x in the interval (3,5].

1.2.1 Computing the HK-bound

Let each incompatible pair (p, q) in M define the closed interval $[p, ..., q]$ on the real line, and let $\mathcal{L}$ denote the set of those intervals. Note that the endpoints of any interval in $\mathcal{L}$ are on integer points. The problem of finding $R^*(M)$ can be restated as:

> **The Interval Coverage Problem**: Find a minimum-sized set of *non-integer* points $R^*(M)$ so that each interval in $\mathcal{L}$ contains at least one point in $R^*(M)$.

The *Interval Coverage Problem* can be efficiently solved by a greedy, left-to-right scanning algorithm shown in Figure 1.3.

Theorem 1.2.2 *Algorithm Interval-Scan correctly computes the HK-bound, HK(M). In particular, at the end of the scan, R is a minimum-sized set such that every interval I in $\mathcal{L}$ contains at least one point in R.*

Proof By the way the scan works, the requirement that every interval I in $\mathcal{L}$ contains a non-integer point in R is clearly satisfied. To see that R is minimum-sized, suppose that I and I' are two intervals in $\mathcal{I}$, and that I was placed in $\mathcal{I}$ before I', so the right end of I is to the left of the right end of I', and

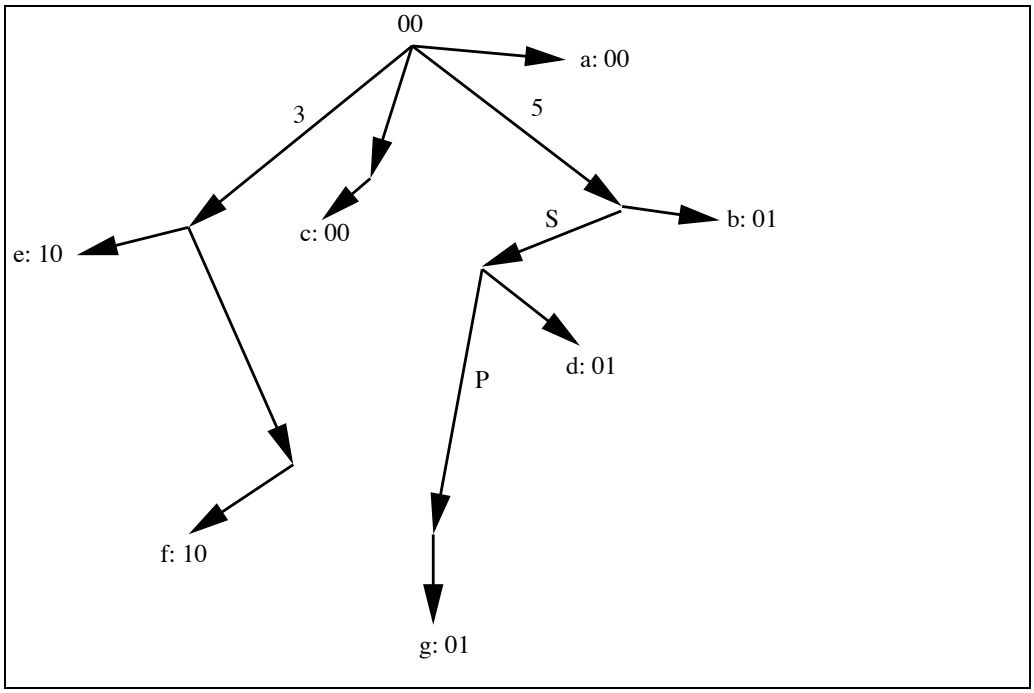


Figure 1.2: Continued illustration for the proof of Theorem 1.2.1. The tree resulting from the network in Figure 1.1 after removal of one edge into each recombination node, and removal of sites other than 3, 5. The tree correctly derives the sequences in M restricted to sites 3 and 5. Therefore, sites 3 and 5 can't be incompatible.

ALGORITHM INTERVAL-SCAN (M)

Sort the intervals in $\mathcal{L}$ by their *right* endpoints, smallest (leftmost) first,
where ties are broken arbitrarilly.
Let L_r be the resulting sorted set of intervals.

Set $R = \emptyset$ and $\mathcal{I} = \emptyset$, and let r be the *left* endpoint
of the interval at the head of L_r.

while (L_r is not empty) **do**

Remove the interval I at the head of L_r.
if (point r is not strictly in I) **then**
place the point $q - \epsilon$ into R, where $0 < \epsilon < 1$,
and q is the right endpoint of interval I; Set r to q, and place I into $\mathcal{I}$
{$\mathcal{I}$ is a set of intervals which will be used in the proof of correctness}.
endif

endwhile

Figure 1.3: Algorithm *Interval-Scan* solves the *Interval Coverage Problem* and computes the HK-bound, given the set of intervals $\mathcal{L}$ obtained from M.

the two intervals cannot share more than a single point (the right end of I and the left end of I'). See Figure 1.2.1. If the two intervals did share more than a single point, then the point placed into R when I was placed into $\mathcal{I}$ would also be contained in I', which contradicts the fact that the algorithm placed both I and I' into $\mathcal{I}$. So, no pair of intervals in $\mathcal{I}$ can have a non-trivial intersection. It follows that no point on the real-line can be strictly between the endpoints of two different intervals in $\mathcal{I}$, and hence $|\mathcal{I}|$ is a lower bound on $|R^*(M)|$. But $|R| = |\mathcal{I}|$, so $|R| = |R^*(M)| = HK(M)$, proving that this scanning procedure correctly computes the HK-bound. ■

It should be clear that the time to compute the HK-bound, given the sorted set L_r, is just $O(|L_r|)$, and so the HK-bound is efficiently computable.

Another characterization of the HK-bound As an aside, we note another result that can be deduced from the reasoning in the proof of Theorem 1.2.2. Let $\mathcal{I}^*(\mathcal{M})$ denote the *largest* set of intervals in $\mathcal{L}$ such that no pair of intervals in $\mathcal{I}^*(\mathcal{M})$ has a non-trivial intersection. Clearly, $|R^*(M)| \geq |\mathcal{I}^*(\mathcal{M})|$, since a distinct point must be chosen for each interval in $\mathcal{I}^*(\mathcal{M})$. But the scanning algorithm chooses a set of points R and a set of intervals $\mathcal{I}$ (such that no pair has a non-trivial intersection) where $|R| = |\mathcal{I}|$. The next Corollary then follows:

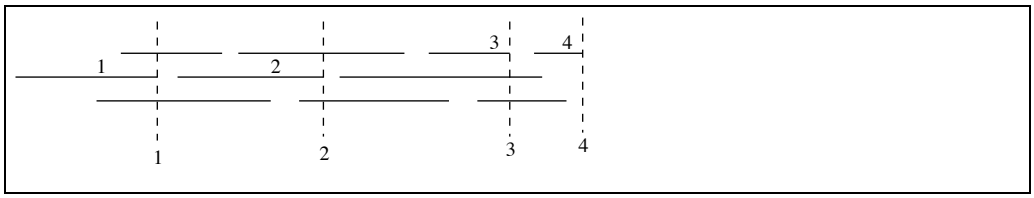


Figure 1.4: A cartoon illustrating the workings of the scan used in Algorithm *Interval-Scan* to compute the HK-bound. The intervals in $\mathcal{L}$ are represented by horizontal lines, and the points chosen for R are represented by dashed vertical lines. The numbers labeling intervals identify the intervals in $\mathcal{I}$, and correspond to the points in R, in the order that points are added to R.

Corollary 1.2.1 *The set of intervals $\mathcal{I}$ found by the scanning algorithm has the maximum possible size. That is $|\mathcal{I}| = |\mathcal{I}^*(\mathcal{M})|$, and the output of Algorithm Interval-Scan (and hence the HK-bound) can be described as the size of the largest set of intervals in $\mathcal{L}$ such that no pair of intervals have a non-trivial intersection.*

The characterization of the HK-bound given in Corollary 1.2.1 is closer to the original characterization given in [13] than is the definition of the HK-bound in terms of $R^*(M)$.

The HK-bound has been widely used in the biological literature, and the choice of the points in $R^*(M)$ has sometimes even been used as an estimate for where the recombination crossovers may have occurred in the true (but unknown) ARG that derived M. However, we will see that the HK-bound is a relatively weak (low) lower bound[1] in comparison to other bounds on $Rmin(M)$ that were developed after it.

1.3 The Haplotype Bound

The HK-bound was introduced in 1985, and very little progress was made in improving lower bounds on *Rmin* until the dissertation of Simon Myers in 2003 [18, 19]. There, the *Haplotype Lower Bound*, was introduced along with a "composite method" that dramatically improves the quality of that bound. We will explain the composite method after discussing the original haplotype bound. Consider the set of sequences M arrayed in a matrix.

Definition Let $D_r(M)$ and $D_c(M)$ be the number of distinct *rows* and *columns*

[1]Ironically, the study of the HK bound in [13] was partly intended to show that the bound is often considerably lower than the true number of recombinations in simulated ARGs, and to therefore discourage its use. However, the HK-bound continued to be widely used, since for some time it was the only non-trivial lower bound method known. There was also confusion in the literature, which sometimes implied (in different terminology) that the HK-bound is equal to $Rmin(M)$, although less than the true number of recombinations.

of M, respectively. The *Haplotype Lower Bound* on M, denoted $H(M)$, is defined to be $D_r(M) - D_c(M) - 1$.

Actually, we assumed earlier that all the n rows of M are distinct, so we could have simplified the haplotype bound to $n - D_c(M) - 1$, but the statement of the bound as $D_r(M) - D_c(M) - 1$ emphasizes that the bound applies even when some rows are not distinct.

For example, consider the data M in Figure **??**. In that example, there are 7 distinct rows and 5 distinct columns, so $H(M) = 1$, which is the same as the *HK* bound. However, if we modify M by adding the additional sequence $h = 00000$, creating input M', then the number of distinct rows becomes 8 and the number of distinct columns remains 5, so $H(M') = 2$. Note that the addition of sequence h does not create any more incompatible pairs in the data, and so the *HK* bound would remain equal to 1.

Theorem 1.3.1 *For any M, $H(M) \leq Rmin$.*

Proof First, if there are some non-distinct columns of M, arbitrarily remove duplicate columns from M so that all columns in the resulting matrix M' are distinct, and M' has exactly $D_c(M)$ columns. Note that $D_r(M') = D_r(M)$.

Any ARG $\mathcal{N}$ that derives M also derives M'; simply remove any site labeling an edge if that site is not in M'. It follows that $Rmin(M') \leq Rmin(M)$. That is, the number of recombination nodes needed in an ARG to derive M' cannot be larger than the number needed to derive M. So any lower bound on $Rmin(M')$ is a lower bound on $Rmin(M)$. Let $\mathcal{N}'$ be a MinARG that derives M', i.e., using $Rmin(M')$ recombination nodes. ARG $\mathcal{N}'$ must contain nodes labeled by $D_r(M')$ or more distinct sequences, since all the distinct sequences in M' must be generated in $\mathcal{N}'$. Note that the root of $\mathcal{N}'$ might be labeled with one of the sequences in M'. Since each character can mutate at most once, there can be at most $D_c(M')$ edges labeled by the sites in M', and so there are at most $D_c(M')$ distinct sequences that label tree-nodes or leaves whose entering edge is labeled with a site in M' (recall that edges into recombination nodes have no edge labels). So at least $D_r(M') - (D_c(M') + 1)$ of the distinct sequences must label recombination nodes in $\mathcal{N}'$. Each recombination node has only one label, so there must be at least $D_r(M') - D_c(M') - 1 = D_r(M) - D_c(M) - 1 = H(M)$ recombination nodes in $\mathcal{N}'$. So, the actual number of recombination nodes in M' must be at least $H(M)$, and since this holds for any N', it follows that $Rmin(M') \geq H(M)$. But we saw earlier that $Rmin(M) \geq Rmin(M')$, so $Rmin(M) \geq H(M)$. ■

Continuing the example started above, note that the ARG in Figure **??** has two recombination nodes, and if we add an edge from the root to a new leaf labeled h, the new ARG derives the set of sequences M'. Since $H(M')$ is two, we can conclude that $Rmin(M') = 2$, and the modified ARG is a MinARG for

M'. However, since $H(M)$ is only one, we cannot (yet) conclude that the ARG in Figure **??** is MinARG for M.

We can also establish the following, using a proof similar to the proof for Theorem 1.3.1.

Theorem 1.3.2 *If M is generated on an ARG $\mathcal{N}$ whose root sequence is not in M, then the number of recombination nodes in $\mathcal{N}$ must be at least the $D_r(M) - D_c(M) = H(M) + 1$.*

Close examination of the proof of Theorem 1.3.1 yields the following:

Corollary 1.3.1 *If $\mathcal{N}$ is an ARG for M that has only $H(M)$ recombination nodes (and hence is a MinARG), and every site in M is distinct, then every node in $\mathcal{N}$ is labeled by a sequence in M.*

Conversely,

Corollary 1.3.2 *If $\mathcal{N}$ is an ARG for M where each node is labeled by a distinct sequence in M, and no edge is labeled by more than a single site, then $\mathcal{N}$ has exactly $H(M)$ recombination nodes (and hence is a MinARG).*

Two additional observations There are two observations about the haplotype bound that will be useful later. First, the haplotype bound is unaffected by the given *order* of the characters in M. We could permute the order of the columns of M and the numbers of distinct rows and columns would not change. Second, the haplotype bound is valid for biological processes other than single-crossover recombination. For example, it is valid for multiple-crossover recombination, or as a lower bound on the number of hybridization events needed to generate a set of binary sequences. What is required for these and other applications of the haplotype bound, is that each mutation occurs at most once, and that each reticulation event (recombination, hybridization etc.) only generates a single new sequence.

1.3.1 Removing compatible characters usually boosts the haplotype bound

The haplotype bound $H(M)$ can be efficiently computed, but the bounds computed for data generated by the program *ms* [12] show that $H(M)$ by itself is a very poor bound, and often is a negative number! However, when used with the *composite-bound method* explained below, it leads to much improved, i.e., higher, lower bounds for *Rmin*, compared to the *HK* bound. Before developing the composite-bound method, we explain another significant way to boost $H(M)$, that also comes from [18, 19]. Recall that a character c in M is called

compatible with all other characters, if c is not incompatible with any character in M.

Theorem 1.3.3 *Suppose character c is compatible with all other characters in M. Then the haplotype bound either increases or remains the same when character (column) c is removed from M. That is, letting M' denote the matrix M after removal of character c, $Rmin(M) \geq H(M') \geq H(M)$.*

For CS 224, this is a homework problem.

Myers and Griffiths [18, 19] suggest that the removal of all characters that are compatible with all other characters in M can speed up computations by reducing the size of the matrix. That is one benefit. But the more important benefit that we have observed is the very substantial increase in the resulting lower bound on $Rmin$ when all compatible characters are removed, particularly when used locally in the composite-bound method to be explained next.

1.3.2 The Composite-Bound Method

Here we introduce a general method developed by Myers and Griffiths [18, 19], called the *composite-bound method*, that can (and usually does) substantially boost several lower bounds on $Rmin(M)$. We will explain the method in general, and also discuss in particular how it boosts the haplotype lower bound.

As noted earlier, the haplotype bound, when applied to a *whole* dataset M, is often very low and can even be a negative number. Intuitively, the haplotype bound will tend to be low when the number of columns of M is large relative to the number of rows of M. This is not guaranteed, but tends to be the case. Conversely, the haplotype bound tends to be higher when the number of columns is small relative to the number of rows. We would like to take advantage of this intuition to obtain a better lower bound on $Rmin(M)$. This is what the composite method does.

Recall that the sites in M have a *fixed linear order*, so we can unambiguously specify an interval of sites. The fact that the sites have a fixed linear order is critical in the composite method. The composite method *combines* several (local) lower bounds computed (by any method) over a family $\mathcal{L}$ of intervals of sites. For simplicity of exposition, we assume that the m sites of M are arrayed on the real line, on the *integer* points 1, ..., m. Hence, when we speak of an interval of sites I, the left and right ends of I are at integer points.

Definition Given an interval of sites I, let $M(I)$ denote the matrix M restricted to the sites in I, and let $b(I)$ denote a lower bound computed (somehow) for $Rmin(M(I))$. The bound $b(I)$ is called a "local bound".

Definition Given a family $\mathcal{L}$ of intervals of sites, let $\mathcal{B} = \{b(I) : I \in \mathcal{L}\}$, i.e., the set of local bounds, one for each interval in $\mathcal{L}$.

Definition Given $\mathcal{L}$ and $\mathcal{B}$, $R^*(\mathcal{L})$ is defined as the *minimum-sized* set of points such that for for each interval $I \in \mathcal{L}$, there are at least $b(I)$ points in $R^*(\mathcal{L})$ that fall strictly in the interior of interval I. The number $|R^*(\mathcal{L})|$ is called a *composite-bound*.

The reader should see that the composite-bound is a generalization of the HK-bound. In particular, if each pair of incompatible pairs of sites (p, q) in M defines an interval $I = [p, q]$, and for each such interval I, $b(I)$ is set to 1, then the composite bound, $|R^*(\mathcal{L})|$, is exactly the HK-bound for M.

Theorem 1.3.4 *For any M, if $b(I)$ is a lower bound on $Rmin(M(I))$, for each interval $I \in \mathcal{L}$, then $|R^*(\mathcal{L})| \leq Rmin(M)$. That is, the composite-bound is a valid lower bound on $Rmin(M)$.*

Proof The proof is a simple extension of the proof used to show that the HK-bound is a valid lower bound on $Rmin(M)$. We first show that for every interval $I = [p, ..., q]$ in $\mathcal{L}$, every ARG that derives M must have at least $b(I)$ recombination nodes whose crossover-indices are in the interval $(p, .., q]$, which is open on the left end and closed on the right end. For contradiction, suppose there is an ARG $\mathcal{N}$ which derives M, where this is not the case. Then modify $\mathcal{N}$ to obtain an ARG $\mathcal{N}'$ just for the set of sequences $M(I)$. We do this by removing all labels on edges that fall outside of I, and at each recombination node x in $\mathcal{N}$, where the crossover-index b_x is *not* in the interval $(p, ..., q]$, we remove exactly one of the incoming edges as follows. If the crossover-index b_x at x is less than or equal to p, remove the P-labeled edge into x and retain the S-labeled edge; conversely, if b_x is greater than q, remove the S-labeled edge, and retain the P-labeled edge into x. Note that at any recombination node x where $b_x \in (p, ..., q]$, nothing is changed. The resulting network is an ARG $\mathcal{N}'$ that derives the sequences $M(I)$ and where every recombination node x has a crossover-index $b_x \in (p, ..., q]$. But $\mathcal{N}'$ has fewer than $b(I)$ recombination nodes, contradicting the assumption that $b(I)$ is a lower bound on $Rmin(M(I))$.

Now, for any ARG $\mathcal{N}$ that derives M, let $\mathcal{B}$ be the set of crossover-indices at recombination nodes of $\mathcal{N}$, and for each recombination node x, create the point $p_x = b_x - \epsilon_x$, where $0 < \epsilon_x < 1$ and ϵ_x is different from ϵ_y for any other recombination node y. The result is a set of $|\mathcal{B}|$ distinct points such that for every interval $I \in \mathcal{L}$, there are at least $b(I)$ points in the set that are strictly in the interior of I. Therefore $|\mathcal{B}| \geq |\mathcal{R}^*(\mathcal{L})|$, and in particular, when $\mathcal{N}$ is the ARG with $Rmin(M)$ recombination nodes, we see that $|R^*(\mathcal{L})| \leq Rmin(M)$, so the composite-bound is a lower bound on $Rmin(M)$. ■

1.3.2.1 Computing the composite-bound

As probably anticipated by the reader, given the sets $\mathcal{L}$ and $\mathcal{B}$, the composite-bound can be efficiently computed by an extension of Algorithm *Interval-Scan*

used to compute the HK-bound. As in Algorithm *Interval-Scan*, let L_r be the sorted set of right endpoints of the intervals in $\mathcal{L}$, smallest (leftmost) first. Then $R^*(\mathcal{L})$ can be found efficiently by a single scan of L_r, corresponding to left-to-right sweep of the points in L_r. In particular, when the right endpoint q of an interval $I = [p, ..., q]$ is examined, if $z < b(I)$ points in I have already been selected, then place an additional $b(I) - z$ copies of point $q - \epsilon$, for $0 < \epsilon < 1$, into $R^*(\mathcal{L})$.

The proof of correctness of this method is a simple extension of the proof of correctness of Algorithm *Interval-Scan*. We leave the formal proof to the reader. Also, as in the computation of the HK-bound, after the sorted list L_r is known, the time required for the method is linear in the size of L_r.

1.3.2.2 The simplest use of the composite-bound method

The simplest way to apply the composite-bound method is to first compute a local bound for each of the $\binom{m}{2}$ intervals in a dataset M with m sites, and then compute the composite-bound using these local bounds. This can be done with any local bound, but this approach has been shown to be particularly effective when the local bound for each interval is the haplotype bound.

1.3.3 The composite-bound greatly boosts the haplotype bound

As mentioned earlier, the haplotype bound is generally not very large when computed on data with a large number of sites compared to the number of taxa. But by computing local haplotype bounds for all the intervals, or for the smaller intervals only, and then computing the composite-bound using those local haplotype bounds, the resulting lower bound on $Rmin(M)$ can be greatly increased. This has been consistently shown empirically. Additional increases are obtained through the use of Theorem 1.3.3 in each interval where the haplotype bound is computed. That is, for each interval I where a local haplotype bound is to be computed, first remove every site that is not incompatible with some other site *in interval* I. Note that a site might be removed in an interval I, even though it is incompatible with some site outside of I. We call the resulting lower bound the *Interval RecMin bound*.

An example of the effectiveness of the composite-bound method using local haplotype bounds is shown in Figure 1.5. The haplotype bound computed from the entire data is -3 (a useless bound), but the Interval RecMin bound is 6. The data used in that example is the widely-studied SNP data obtained by M. Kreitman [15] in 1983, which was one of the first significant SNP datasets published. The HK bound is five, and $Rmin(M)$ for this data is 7, as we will discuss in Sections 2.2.5 and **??**.

```
00000000110000000011011101111000000000000
00100000000000000011011101111000000000000
00000000000000000000000000000000010000101
00000000000000001100000000000000010011000
00011000101100111100000000000000001000000
00100000000000010000000000000010101110000 10
00100000000000010000000000000011111101000000
11111000101110010000000000000011111101100000
11111000101110010000000000000011111101100000
11111000101110010000000000000011111101100000
11111111100001010000100010000111111101000000
```

Figure 1.5: In this data (Kreitman's data), the HK-bound is 5; the haplotype bound computed on the entire data is -3; the composite-bound computed using the haplotype bound as the local bound in each of the intervals is 5; the composite-bound computed using the haplotype bound as the local bound, when compatible columns are removed in each interval (the Interval RecMin bound) is 6. $Rmin(M)$ is actually 7.

Why, intuitively, does the composite-bound method helps raise the haplotype bound It is worth considering why the composite-bound is effective when used together with the haplotype bound. In addition to the fact that the haplotype bound tends to be low when intervals are large, and the composite method uses intervals of all sizes, the key point is that the haplotype bound is unaffected by the linear order of the sites. That is, we could permute the order of the sites in M and the resulting haplotype bound would be unchanged. So, the haplotype bound does not incorporate any constraints imposed by the physical reality of the fixed linear order of the sites, and the fact that a crossover is an event that takes place at a particular location among the sites. The effect of a recombination event is constrained and influenced by a given linear order of the sites, but that constraint is not incorporated into the haplotype bound. Intuitively, the composite-bound approach is effective when combined with the haplotype-bound because it imposes *ordered* constraints (that at least $b(I)$ recombinations must occur *inside* each local interval I) and hence reflects the fixed linear order of the sites, and the spacial aspect of a recombination crossover.

The same intuition holds for other local lower bounds that are not affected by the linear order of the sites. For example, we will discuss the *connected-component* lower bound in Section **??** and see that it is also unaffected by the order of the sites. Again, the composite-bound method improves the resulting lower bound on $Rmin(M)$ when each local lower bound is a connected-component lower bound. In contrast, the composite method might not be as effective when the local bounds are already affected by the linear order of the

sites. For example, when each local bound is an HK-bound (which is already affected by the linear order of the sites) the composite-bound is just the HK-bound applied to the entire data, and so the composite-bound method offers no improvement at all.

1.3.4 Applying the Haplotype Bound to *Subsets* of Sites: a major improvement

Myers and Griffiths [18, 19] introduced another way to significantly boost the basic haplotype bound, with an increase in computation time. The basic idea is to compute the haplotype bound on *subsets* of columns in an interval, rather than on all of the columns in the interval. We formalize that idea here.

Definition For a *subset* of sites S (*not* necessarily contiguous) in M, let $M(S)$ be the sequences in M restricted to the sites in S, and let $H(M(S))$ be the haplotype bound computed on $M(S)$.

Lemma 1.3.1 *Let S be a subset of sites in M, whose leftmost point is p and whose rightmost point is q. $H(M(S))$ is a valid local lower bound for interval $I = [p, ..., q]$. That is, any ARG that derives M must have at least $H(M(S))$ recombination nodes whose crossover-indices are in the interval $(p, q]$.*

Proof Similar to the first part of the proof of Theorem 1.3.4, consider a MinARG $\mathcal{N}$ for the sequences $M(I)$, and then remove all of the sites from $\mathcal{N}$ that are not in S. The result is an ARG for $M(S)$ with $Rmin(M(I)$ recombination nodes (and now some of the recombinant sequences might be equal to a parental sequence, in which case some recombination nodes might be removed). So, $Rmin(M(S)) \leq Rmin(M(I))$. Since $H(M(S)) \leq Rmin(M(S))$, the lemma follows. ■

Given Lemma 1.3.1 we could compute a valid local lower bound $b(I)$ for an interval I by computing $H(M(S))$ for *each* subset S of sites in I (with length denoted $|I|$), and then taking $b(I)$ to be the largest of those $2^{|I|}$ haplotype bounds.

It may not be intuitive, but the approach of looking at subsets of sites in an interval often yields a local lower bound that is larger than $H(I)$, e.g., the basic haplotype bound computed for the entire set of sites in I. However, we have already seen one indication of this, in Theorem 1.3.3, when compatible sites are removed from M. For a more general illustration, consider the example in Figure 1.6 where there are 8 distinct rows and 6 distinct columns, giving a haplotype bound of 1. Removal of sites 1, 3 and 4 decreases the number of distinct columns by three, but does not reduce the number of distinct rows, and so the haplotype bound computed on the remaining columns is increased to 4. This type of situation will be more formally treated in Theorem 1.4.2.

	c_1	c_2	c_3	c_4	c_5	c_6
r_1	0	0	1	0	0	0
r_2	0	0	1	0	1	0
r_3	0	1	1	0	0	0
r_4	0	1	1	1	1	0
r_5	0	0	1	1	0	1
r_6	1	0	1	1	1	1
r_7	1	1	0	0	0	1
r_8	1	1	0	0	1	1

Figure 1.6: Dataset M with 8 distinct rows and 6 distinct sites, so $H(M) = 1$. However, if sites 1, 3 and 4 are removed, the resulting dataset still has 8 distinct rows but now only has three distinct sites, so the resulting haplotype bound is 4.

Clearly, increasing local lower bounds cannot reduce the resulting composite lower bound, and may lead to a larger composite-bound for *Rmin*.

The Optimal RecMin Bound **Definition** Let $S^*(I)$ be a subset of sites in interval I that maximizes $H(M(S))$ over all subsets S of sites in I. Then subset $S^*(I)$ is called the *optimal subset* for I, and $H(M(S^*(I)))$ is called the *Optimal Haplotype Bound* for I.

Definition When, for every interval I in M, the local lower bound $b(I)$ is set to the optimal haplotype bound for I, the resulting composite lower bound on $Rmin(M)$ is called the *Optimal RecMin Bound* on M.

Clearly, the *Optimal RecMin Bound* is at least as high as the Interval RecMin Bound. It has been empirically demonstrated that the *Optimal RecMin Bound* is typically much larger than the Interval RecMin bound, reflecting the utility of considering subsets of sites inside of each interval. We will see an example of this later.

Since a subset of sites S in an interval I is also a subset in any interval that contains I, it is more efficient to compute the *Optimal RecMin Bound* by first enumerating every subset of sites S, computing $H(M(S))$ for each subset S, and then using these bounds to determine the local bound $b(I)$ for each interval I. Still, because all $\theta(2^m)$ subsets of sites must be explicitly enumerated in this approach, the time required will generally be prohibitive. Consistent with the fact that full enumeration is computationally infeasible, Bafna and Bansal [1, 3] proved that the problem of computing the *Optimal RecMin Bound* is NP-hard. Given these realities, we will discuss two alternative approaches, one (program *RecMin*) that enumerates only *some* subsets of sites, and a second approach

that computes the *Optimal RecMin Bound* without explicitly enumerating any subsets of sites, although it again involves a computation that required worst-case exponential time.

1.3.5 Program RecMin

Myers and Griffiths [18, 19] encapsulated the use of local haplotype-bounds computed over *subsets* of sites, and the composite-bound method, in a program they call *RecMin*. However, because enumerating all subsets of sites is generally impractical (taking days on even moderate sized data), RecMin generally computes local bounds for a *restricted* set of subsets of sites.

In *RecMin*, the user specifies two parameters s and w, and *RecMin* computes the haplotype-bound $H(M(S))$ for every subset S of M, with s or fewer sites, provided that no pair of sites in S is more than w positions apart. The parameter s is called the *subset size* parameter, and the parameter w is called the *subset width* parameter. Let $\mathcal{S}$ be the family of subsets that obey the restrictions imposed by s and w. RecMin computes $H(M(S))$ for every subset S in $\mathcal{S}$, and then for every interval I, the local lower bound $b(I)$ is set to the largest of value $H(M(S))$ where S is contained in I. *RecMin* then computes the composite-bound using those local bounds. *RecMin* also uses heuristics to avoid the explicit examination of some of the specified subsets.

The default settings for *RecMin* were initially $s = 8$ and $w = 12$, but we have found that *RecMin* 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 $Rmin(M)$, and far superior to any of the practical alternatives that came before it. However, as noted earlier, for large problem instances of the size of current interest, *RecMin* cannot compute the *Optimal RecMin Bound*, and be sure it has been computed. The only way that *RecMin* can guarantee to compute the *Optimal RecMin Bound* is to set $s = w = m$, which is usually impractical.

We next discuss an alternative to RecMin which is able to compute the *Optimal RecMin Bound* efficiently on data of current interest, and much larger than the data for which RecMin can compute the *Optimal RecMin Bound*.

1.4 Program HapBound: Practical computation of the Optimal RecMin Bound, and beyond

We first discuss the Integer Linear Programming approach developed in [27] to efficiently compute, in practice, the *Optimal RecMin Bound* on biological datasets of current interest.

1.4.1 The key idea

The following theorem is immediate from the definition of the *Optimal RecMin Bound*.

Theorem 1.4.1 *If, for each interval I in M, we set the local lower bound $b(I)$ equal to $H(S^*(I))$ and use those local bounds to compute a composite-bound, the resulting lower bound is the Optimal RecMin Bound on $Rmin(M)$.*

Given Theorem 1.4.1, to compute the *Optimal RecMin Bound*, we want to find the optimal subset $S^*(I)$ for each interval I in M. This is the key idea, but how can we efficiently (at least in practice) find the subset $S^*(I)$ for each I? The next theorem begins to explain the answer.

Recall that for any subset S of columns, $M(S)$ denotes the matrix M restricted to the columns in S, and for any interval I, $M(I)$ denotes the matrix M restricted to the columns in I.

Definition For any interval I, let $\tilde{M}(I)$ denote the matrix $M(I)$ after removal of any duplicate rows in $M(I)$. For a subset of sites S in I, $\tilde{M}(I,S)$ denotes $\tilde{M}(I)$ restricted to the sites in S.

Theorem 1.4.2 *([21]) An optimal subset $S^*(I)$, can be found by finding a smallest subset of sites S in I such that that every row in $\tilde{M}(I,S)$ is distinct.*

Given Theorem 1.4.2, we have

Definition A subset of sites is called a *minimum* $S^*(I)$ if it is a smallest subset of sites S in I such that every row in $\tilde{M}(I,S)$ is distinct.

As an example, consider the dataset M in Figure 1.6, and let I in this case be the entire interval 1..6. The eight rows are distinct, so $\tilde{M}(I) = M$. The set of sites $\{2,5,6\}$ is a smallest set S such that the eight rows are distinct in $M(S)$. So those three sites yield the highest haplotype bound possible in the interval 1..6, and hence form a minimum $S^*(I)$ for M.

Proof (of Theorem 1.4.2). First, observe that for any subset of sites S in interval I, two columns in $M(S)$ are distinct if and only if they are distinct in $\tilde{M}(I,S)$, so $H(M(S)) = H(\tilde{M}(I,S))$. It follows that the removal of duplicate rows in $M(I)$ (creating $\tilde{M}(I)$) does not affect which subset of columns in I maximizes the haplotype bound, and so $S^*(I)$ is an optimal subset for I in both $M(I)$ and $\tilde{M}(I)$. Let $n'(I)$ denote the number of rows in $\tilde{M}(I)$.

We next show that all rows in $\tilde{M}(I, S^*(I))$ are distinct, or an equally good subset of sites can be found where this is true. Suppose for contradiction that two rows r_1, r_2 in $\tilde{M}(I, S^*(I))$ are identical. Since r_1, r_2 are not identical in $\tilde{M}(I)$, there must be a site c in $I - S^*(I)$ such that the state of r_1 at c differs

from the state of r_2 at c. So, if we add c to $S^*(I)$ we increase the number of distinct rows by *at least* one, while increasing the number of distinct sites by *at most* one. Therefore $H(\tilde{M}(I, S^*(I) + c)) \geq H(\tilde{M}(I, S^*(I)))$. But, by the choice of $S^*(I)$, it can only be that $H(\tilde{M}(I, S^*(I) + c)) = H(\tilde{M}(I, S^*(I)))$. If the set of rows $\tilde{M}(I, S^* + c)$ still contains two identical rows, we can again find a new column to add to $S^*(I)$, and repeat this step until we finally have an optimal subset of sites $S^*(I)$ where all the rows of $\tilde{M}(S^*(I))$ are distinct.

Similarly, we can assume that all the sites in $S^*(I)$ are distinct, for if not, the removal of any duplicate copies cannot change the number of distinct sites nor the number of distinct rows in the resulting matrix. Therefore, we assume that all the $n'(I)$ rows and all the $|S^*(I))|$ sites of $\tilde{M}(I, S^*(I))$ are distinct, so $H(\tilde{M}(I, S^*(I)) = n'(I) - |S^*(I)| - 1$.

Now consider a smallest set of sites S in I such that all rows in $\tilde{M}(I, S)$ are distinct. Clearly, if any pair of sites in S are identical, then one of the copies can be removed to create a smaller set of sites $\overline{S}$ where all rows in $M(\overline{S})$ are distinct. So, all sites in S are distinct. Also, the number of (distinct) rows in $\tilde{M}(I, S)$ is $n'(I)$, so $H(\tilde{M}(I, S)) = n'(I) - |S| - 1$. We want to show that $H(\tilde{M}(I, S)) = H(\tilde{M}(I, S^*(I)))$. If not, then $n'(I) - |S| - 1 = H(\tilde{M}(I, S)) < H(\tilde{M}(I, S^*(I))) = n'(I) - |S^*(I)| - 1$, so $|S^*(I)| < |S|$, contradicting the assumption that S is a smallest subset of sites in $\tilde{M}(I)$ such that all the rows of $\tilde{M}(I, S)$ are distinct. ■

1.4.2 Finding $S^*(I)$ by Integer Linear Programming

Theorem 1.4.2 tells us precisely what to look for in order to find an optimal subset for I, $S^*(I)$, but it does not tell us how to do it. That problem is answered in this section.

Definition We say that a site c in $\tilde{M}(I)$ *distinguishes* two rows r_1 and r_2 if $\tilde{M}(I)[r_1, c] \neq \tilde{M}(I)[r_2, c]$. We say that a subset of sites S distinguishes the rows in $\tilde{M}(I)$ if for each pair of rows r_1, r_2 in $\tilde{M}(I)$, there is a site c in S that distinguishes r_1 and r_2.

The following Lemma is almost self-evident and a formal proof is left to the reader.

Lemma 1.4.1 *Let S be a subset of sites in interval I. Then, the rows in $\tilde{M}(I, S)$ are distinct if and only if S distinguishes the rows of $\tilde{M}(I)$.*

Given Theorem 1.4.2 and Lemma 1.4.1, in order to find a haplotype optimal subset for an interval I, we want to find a smallest set of sites that distinguishes the rows of $\tilde{M}(I)$. That task can be formulated and solved as an *Integer Linear Program* as follows.

For any pair of rows r_1, r_2, let $D(r_1, r_2)$ denote the set of sites in I that distinguish rows r_1 and r_2 of $\tilde{M}$. Since the rows of $\tilde{M}$ are distinct, $D(r_1, r_2)$

cannot be empty. For each site c in I, let $X(c)$ be a binary integer linear programming variable, i.e., one that is only allowed to take on values 0 or 1. In a solution, variable $X(c)$ will be set to 1 to indicate that site c should be taken into S, and set to 0 to indicate it should not be taken into S. The linear program will have the inequality

$$\sum_{c \in D(r_1, r_2)} X(c) \geq 1$$

for each pair of rows r_1, r_2 in $\tilde{M}$. The effect of these inequalities is to force the selection of a subset of sites that distinguish the rows of $\tilde{M}(I)$. Finally, the objective function is

$$Minimize \sum_{c \in I} X(c).$$

A solution to the integer linear program sets each variable $X(c)$ to either 0 or 1, and hence specifies a smallest set S that distinguishes every pair of rows in $\tilde{M}(I)$. Thus, by Theorem 1.4.2 and Lemma 1.4.1, a solution to the integer program identifies $S^*(I)$, a optimal subset for I.

As an aside, for those familiar with the *set cover* problem or the *minimum test set* problem [8], one can view this integer program as solving an instance of the set cover or test set problems, where each pair of rows (r_1, r_2) defines the set of sites $D(r_1, r_2)$, and the problem is to choose the smallest set of sites to cover all of those sets.

Basic HapBound The above approach, using integer programming to find $S^*(I)$ and $b(I) = H(S^*(I))$ for each interval I, and then using these local lower bounds in the composite-bound method to obtain the *Optimal RecMin Bound*, has been implemented in a program called *HapBound* [27]. In HapBound each of the integer programs is solved by using the GNU ILP solver, GLPK. GNU allows GLPK to be incorporated into other programs, and using GLPK allows a complete version of HapBound to be released to users. With GLPK, HapBound is able to find $S^*(I)$ efficiently (in fractions of a second, to several seconds) for many problem sizes of current interest. This is robust enough to illustrate the practicality of the integer programming approach. However, for large problem instances, the commercial ILP solver, CPLEX, is significantly faster than GLPK. HapBound is not built around CPLEX because users would then need to have a CPLEX license to run HapBound. We will discuss HapBound in more detail in the next section, and some empirical results in Section 1.4.5.

The Bafna-Bansal method A different alternative approach to improving on RecMin, developed by V. Bafna and V. Bansal in [1, 3], formulates the problem of finding $S^*(I)$ (in different notation) in each interval I, and also uses

the composite method to find a global lower bound on $Rmin(M)$. However, instead of using a method that is guaranteed to find $S^*(I)$, they use a fast heuristic algorithm to find a set S that distinguishes every pair of rows.

In more detail, their method first cleans up the input M to reduce the size of the problem in a way that does not change the number of needed recombination events (see Algorithm *Clean* in Section 2.2.1), and that results in a matrix $\tilde{M}$ with no duplicate rows. Then the algorithm computes a lower bound on $Rmin(\tilde{M})$ as follows: Until a set of columns has been selected that distinguishes all pairs of rows in $\tilde{M}$, successively select the column that distinguishes the most pairs of rows that are not already distinguished by any previously selected column. Let S denote the set of selected columns. The lower bound used is the number of distinct rows in $\tilde{M}(S)$ minus $|S|$ minus one.

Although the lower bound computed in this way is not guaranteed to be as large as the *Optimal RecMin Bound*, and cannot ever be larger than it, empirical tests reported in [1, 3] show that it produces consistently higher lower bounds than the *HK* bound and the bound returned by *RecMin* with its default settings (see Section 1.4.5).

Missing Data The Bafna-Bansal method in [3] extends naturally to the realistic problem that biological data often has missing, unknown entries. In molecular sequence data the rate of missing entries can be 1 to 5% percent, while in phylogenetic data the percentage can often be as high as 35%. It is a common practice to remove columns or rows of M so that the remaining submatrix, M', has no cell with a missing value. A lower bound on $Rmin(M')$ can then be computed. However, this approach can substantially reduce the resulting lower bound. Instead, we can define the following problem:

Problem MDOR Given a matrix M with missing entries, fill in the missing entries with binary values in the way that *minimizes* the *Optimal RecMin Bound* on the resulting matrix.

It may seem incorrect to want to *minimize* the resulting lower bound rather than *maximizing* it. However, it is only through minimizing the lower bound that we obtain a value that is guarnateed to be a true lower bound on $Rmin(M_t)$, where M_t is the correct original matrix, with no missing values, from which M was derived.

Of course, we don't know how to solve Problem MDOR efficiently, but if we could solve it, the *Optimal RecMin Bound* on the resulting matrix would be larger or equal to the *Optimal RecMin Bound* computed on M', the matrix resulting from removal of all columns containing missing values.

The effect of including missing entries in the computation of lower bounds can be considerable, even if the lower bound method used is not the best lower bound method available for use on *complete* data. Table 1.1 shows several lower bound values for nine datasets, computed by Program *HapBound* (discussed in

detail in the next section) which is which guaranteed to produce a value that is as high or higher than the *Optimal RecMin Bound*. Each of the nine dataset corresponds to a human subpopulation and a region in the LPL gene *after* sites were deleted to remove any cells with missing entries. In contrast, the weaker lower bound method from [3] (discussed above) was applied to the whole data, including cells with missing entries. In many cases, the lower bounds obtained were higher than the ones shown in Table 1.1, and even higher than the *upper bounds* computed for the *cleaned-up* data. For example, the Table 1.1 shows a lower bound of 13 and an upper bound of 16 for cleaned-up data in the Jackson population in LPL region 3, but the Bafna-Bansal method computed a lower bound of 17 for the data when sites with missing data were not removed. In the N. Karlia population in region 3, the Table shows a lower bound of 8 and an upper bound of 10, while the lower bound from [3] with missing data included is 13. More impressive, the combined data over the three populations in region 3 gives a lower bound of 36 when sites with missing data are included in the analysis, while the lower bound after removing those sites is only 25. However, there are also cases where *HapBound*, which only runs on complete data, gives a higher bound on cleaned-up data than did the Bafna-Bansal method on the data with missing values included.

1.4.3 Program HapBound: speedups and extensions

The basic ideas behind program HapBound were introduced in the prior section. However, HapBound incorporates additional ideas that make it run faster, and often allows it to compute lower bounds that are higher than the *Optimal RecMin Bound*. We first discuss the major way to speedup HapBound.

Speeding up the computation In the discussion so far, the computation of the *Optimal RecMin Bound* requires that we explicitly find $b(I) = H(S^*(I))$ for *every* interval I of M. If M has m sites, this approach executes $\binom{m}{2}$ integer programs. That number grows quadratically in m, rather than exponentially in m, which characterizes the number of subsets that RecMin needs to examine in order to find, and be sure it has found, the *Optimal RecMin Bound*. However, in worst case, the time to solve integer programming problems grows exponentially with increasing problem size. Thus, compared to RecMin, HapBound does a quadratic number of (worst case) exponential-time computations, instead of a guaranteed exponential number of simple, polynomial-time computations. It was an empirical question whether this substitution would work to compute the *Optimal RecMin Bound* efficiently in practice. The empirical results are that it does work, but additional speedups are also desired.

The main speedup that is possible is to reduce the number of intervals that HapBound needs to explicitly examine, and thus reduce the number of required

ILP computations.

Suppose we find an optimal subset of sites $S^*(I)$ for interval $I = [1, m]$, and the leftmost and rightmost points of $S^*(I)$ are p and q respectively. Then $S^*(I)$ 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, the local lower bound $b(I')$ can be ignored in obtaining the overall composite-bound. We can exclude those intervals from further consideration, speeding up the total time needed to compute the composite bound.

Excluding the intervals that contain $[p, q]$ is helpful, but that does not (yet) exclude the need to examine all of the subintervals contained in $[1, q-1]$, $[p+1, m]$, $[p+1, q]$, or $[p, q-1]$. But the number of those subintervals that have to be examined can be reduced by recursively applying the same exclusion idea: For each of the four intervals $I = [1, q-1]$, $[p+1, m]$, $[p+1, q]$, and $[p, q-1]$, find an optimal subset $S^*(I)$ for interval I, and then recurse on four new subintervals defined from interval I and the span of $S^*(I)$. In this way, over the entire computation, fewer than $\binom{m}{2}$ intervals are explicitly examined, and fewer than $\binom{m}{2}$ integer programming 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 about 25% of the $\binom{m}{2}$ problems, and when m is several times larger than n, the percentage typically falls to under 5%. As a consequence, on simulated problems of size of current biological interest, HapBound runs in seconds to minutes.

1.4.4 HapBound can often compute larger bounds than the Optimal RecMin Bound

As described, Program HapBound is guaranteed to compute the *Optimal RecMin Bound*. However, HapBound has an option (-S) that typically produces an even higher lower bound. The option increases the running time, but not beyond the range of practicality.

In the method described so far, if $S^*(I)$ is an optimal subset found for interval I, then $b(I)$ is set to $H(S^*) = n - |S^*(I)| - 1$, where M is assumed to have n distinct rows. But an increase in the local bound for I may increase the resulting composite bound. HapBound implements a test to determine whether the sequences in $M(S^*(I))$ can actually be generated on an ARG with only $H(S^*)$ recombinations. If not, then $b(I)$ can be set to $H(S^*(I)) + 1$. HapBound can also test if that bound is tight, or if $b(I)$ should again be increased to $H(S^*(I)) + 2$. These are small increases, but small increases in several local bounds can result in a large increase in the overall composite bound.

Definition Given a set of sequences $M(S)$, we say that $M(S)$ is *self-derivable* (SD) if $M(S)$ can be generated on an ARG $\mathcal{N}$ with an ancestral sequence that

is in $M(S)$, and where *all* of the node labels in $\mathcal{N}$ are in $M(S)$.

That is, $M(S)$ is self-derivable if the sequences in $M(S)$ can be generated without generating any sequences not in $M(S)$. Not every set of sequences is self-derivable.

Lemma 1.4.2 *Given a subset of sites S in M, if the sequences $M(S)$ can be generated on an ARG using exactly $H(M(S))$ recombination nodes, then $M(S)$ must be self-derivable.*

We will not prove Lemma 1.4.2 for an arbitrary subset S, but only for a subset that is minimum $S^*(I)$ for some interval I. The proof is easier with this restriction, and we will only apply Lemma 1.4.2 in the situation when S is a minimum $S^*(I)$ for an interval I.

CS224 HW problem: Prove Lemma 1.4.2.

The converse of Lemma 1.4.2 does not hold in general, but does hold if we modify the definition of an ARG to require that at most one site is allowed to label any edge in the ARG.

Lemma 1.4.2 is useful because it says that if we determine that in an interval I, the set of sequences $M(S^*(I))$ is not self-derivable, then $b(I)$ should be set to $H(M(S^*(I))) + 1$, rather than to $H(M(S^*(I)))$. We next discuss how we can determine whether or not a set of sequences is self-derivable.

A first algorithm to test Self-Derivability: We can test whether $M(S)$ is self-derivable with an an algorithm that runs in time that is polynomial in n, but exponential in m (in worst case). However, it is observed to be practical for large data sets of current interest. To describe the algorithm, we first consider a simplified situation.

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 [14]: To start, a "reached set" of sequences consists of the two ancestral sequences alone. Then at each step, we try to expand the reached set by finding a pair of sequences (f, g) in the reached set that can recombine to create a sequence h that is in $M(S)$ but not yet in the reached set. This is repeated until either the reached set contains all the sequences in $M(S)$, or until no further expansion of the reached set is possible. In the first case, $M(S)$ is generated by recombinations, starting from an ancestral pair, and only using sequences in $M(S)$. In the second case, it is not hard to prove that $M(S)$ cannot be generated from the chosen ancestral pair, only using recombinations and only generating sequences in $M(S)$. This algorithm can be sped up by preprocessing [14], but clearly only takes polynomial time.

The above algorithm, for the simplified situation where there are two ancestral sequences and no mutations are allowed, is not a test for whether $M(S)$ self-derivable. To become a test, it must be modified so that there is only *one*

ancestral sequence rather than a pair, and so that site mutations are *allowed*. The key insight for this modification is that if $M(S)$ is self-derivable, then for every site $i \in S$, during any self-derivation of $M(S)$, a mutation at site i must occur in some sequence that is definitely in $M(S)$ and generate another sequence that is definitely in $M(S)$. Therefore those two sequences differ by exactly one site, namely site i. (It also follows that if there is a site i such that no pair of sequences in $M(S)$ differ only at i, then $M(S)$ is not self-derivable.)

To exploit this key insight, let MUT_i be the set of sequence pairs that differ at exactly site i. We modify the algorithm for the simplified situation by now starting the reached set with just a single ancestral sequence. We also allow the reached set to expand in two different ways. As before, the reached set can expand by a recombination of two sequences in the reached set if the recombination creates a sequence in $M(S)$ that is not yet in the reached set. But the reached set can also be expanded to include a sequence h in $M(S)$, if some sequence f is already in the reached set, and (f, h) is in MUT_i for some i, where no prior expansion of the reached set used a pair in MUT_i. Clearly, if $M(S)$ is self-derivable starting from the chosen ancestral sequence, then there is a generation of $M(S)$ by this algorithm, where for every site i in S, exactly one pair in MUT_i is used to expand the reached set.

The algorithm for the self-derivability test tries all sequences in $M(S)$ as the ancestral sequence, and all ways to choose exactly one pair from each set MUT_i. The number of choices is $n \times \prod_{i \in S} |MUT_i|$ which is bounded by $n \times [\frac{n}{2}]^m$, but is generally much smaller. Further, some combinations of choices can be immediately ruled out and additional effective heuristics reduce the number of choices (details left to the reader).

A second algorithm to test Self-Derivability: It is also possible to test for self-derivability with a dynamic programming algorithm that has worst-case running time bounded by a function that is exponential in n and polynomial in m.

Let K denote a subset of the n rows in $M(S)$, and let $SD(K)$ be a boolean variable that will be set to **TRUE** if the sequences in K are self-derivable, and will be set to **FALSE** otherwise. To start, we set $SD(K)$ to **TRUE** for every singleton set K, i.e., if $|K| = 1$. Then in order of size, we consider each subset K, and set $SD(K)$ to TRUE if there is a sequence $s \in K$ such that $SD(K - \{s\})$ has been set **TRUE** and either of the following conditions holds:

1) s can be created by a recombination of two sequences in $K - \{s\}$, or

2) s differs from some sequence in $K - \{s\}$ by a single mutation at a site c and all of the sequences in $K - \{s\}$ have the same state at site c. The last condition ensures that a mutation at site c has not yet been used in the generation of $K - \{s\}$.

A crude analysis, establishes that the worst-case time for this method is $O(n^2 m^2 2^n)$ although the polynomial terms can be reduced with the use of the ap-

propriate preprocesing and data structures [14]. The correctness of this method is based on the observation that if K is self-derivable in an ARG $\mathcal{N}$, then there is at least one sequence $s \in K$ that is not further mutated or used in a recombination. Therefore, the removal of the leaf labeled s, and its parent if it is also labeled s, from $\mathcal{N}$, leaves an ARG that self-derives $K - \{s\}$. A full proof of correctness and time analysis is left to the reader. In summary we have

Theorem 1.4.3 *Let $M(S)$ be a set of sequences with n taxa and m sites. The self-derivability can be tested in time that is polynomial in n and exponential in m, or that is polynomial in m and exponential in n. Further, if $M(S)$ is self-derivable, then the two algorithms produce an ARG that self-derives $M(S)$.*

Further Increases The self-derivability test is used to determine if the set of sequences $M(S^*(I))$ can be generated on an ARG that has only $H(M(S^*(I))$ recombination nodes. If not, then there cannot be any ARG $\mathcal{N}$ that generates $M(I)$ using only $H(M(S^*(I))$ recombination nodes, because $\mathcal{N}$ can be modified to generate $M(S^*(I))$ using no additional recombinations. Therefore, if $M(S^*(I))$ is not self-derivable, then the local bound $b(I)$ should be set to $H(M(S^*(I)) + 1$ rather than to $H(M(S^*(I))$.

To test if a local bound should be increased by *two*, we note that this is the case when $M(S)$ is not self-derivable, and the inclusion of one new sequence not in $M(S)$ is also not sufficient to allow the expanded set to be self-derivable. If one new sequence did allow the expanded set to be self-derivable, then the new sequence must either be generated by the recombination of two sequences in $M(S)$, or must differ from a sequence in $M(S)$ at exactly one site in S. There are only a polynomial number of such candidate sequences, and we can efficiently generate each new candidate sequence in turn and test the resulting set for self-derivability. If the sequences are not self-derivable in any of these tests, then the local bound should be increased by two. We can continue in this way to determine if the local bound should be increased by three, etc. but the time for each test increases too fast for practical implementation.

Program HapBound, with option -S, tests each minimum $S^*(I)$ subset it finds, to see if $M(S^*(I))$ is self-derivable, and if not, to see if the local bound should be increased by one or by two. For the data sets 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 $Rmin(M)$ that is higher than the *Optimal RecMin Bound* (some comprehensive test results are shown in the next section). For example, for the sequences shown in Figure **??** (on page **??**) the *Optimal RecMin Bound* is two, but HapBound -S returns the value of three.

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 data sets, Program HapBound

with option -M has produced higher lower bounds than any other lower bound program. An example will be shown in the next section. See [27] for idea behind the -M option.

1.4.5 Lower bounds for the LPL and ADH data sets

As an illustration of the efficiency and efficacy of program HapBound, we discuss here the SNP data from [5]. This seminal data is from the LPL locus in humans and contains 88 rows and (coincidently) 88 sites before removing sites with missing or non-SNP data. That data set was also examined in [19]. (The first paper uses 42 sites from the full data in their recombination analysis, while the second paper uses 48 sites. For clearer comparison, we discuss results that included the same 48 sites.)

Using CPLEX to solve the ILP problems, HapBound computed the *Optimal RecMin Bound* of 75 in 31 seconds, and HapBound -S computed a higher bound of 78 in 1,643 seconds, on a 2 GHz machine. Using the GNU ILP solver on the same machine, the times were 871 and 3,326 seconds respectively. Program *RecMin* with the default settings of $s = 8$ and $w = 12$ produced the lower bound of 59 in 3 seconds. It found the *Optimal RecMin Bound* of 75 with parameters $s = w = 25$, in 7,944 seconds. As mentioned earlier, a user would not know that this was the *Optimal RecMin Bound*. To simulate what the user would need to do in order to be sure of getting the *Optimal RecMin Bound*, we set $s = w = 48$ but *RecMin* did not finish within five days of execution. The analysis in [19], based on *RecMin*, reports a lower bound on $Rmin(M)$ for this data of only 70, rather than 75. This is due to running *RecMin* with parameters that are too low [21]. 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 only 22, illustrating the major advance that RecMin made, and the importance of using it or HapBound in place of the *HK* bound[2].

As mentioned above, HapBound has another option, the -M option, that runs efficiently on moderate size data sets and produces the highest lower bounds on those data. For example, the benchmark data set [15] for the ADH locus in humans (shown in Figure 1.5) has 11 sequences and 43 sites. Song and Hein [25] established that $Rmin(M)$ is exactly 7 for this data. The lower bound method

[2]Additional comparisons of the *HK* bound to the lower bound produced by RecMin (with the default parameter settings) [3] further demonstrate the weakness of the *HK* bound, particularly as the recombination rate increases. For example, averaged over 100,000 datasets with a high level of recombination, the mean RecMin lower bound is 36.80, while the mean *HK* bound is 12.07. The Bafna-Bansal lower bound [3] (discussed on page 22) which can never be larger than the *Optimal RecMin Bound* gave mean value of 49.69 on the same 100,000 datasets. We don't know what the average *Optimal RecMin Bound* would be on that data, but it would be at least as high as 49.69.

in [25] has never been implemented, but was manually applied to this data, producing a lower bound of 7. HapBound -M ran in about three seconds on this data and also computed the lower bound of 7. The improved History lower bound developed in [1, 3], which we will discuss in Section **??**, also produces a lower bound of 7. All other implemented lower bound methods that we know of (nine in total) produce lower bounds of only 5 or 6.

1.4.5.1 The human LPL data in more detail

Here we report on the observed deviation between the lower bounds on $Rmin(M)$, computed by HapBound, and the number of recombination nodes used in ARGs that generate data sets that come from human LPL data, reported in [20]. We will discuss the methods used to construct those ARGs in Chapter 2. Unless the lower bound exactly matches the number of recombination nodes in the corresponding ARG, we do not know $Rmin(M)$ exactly, and so we refer to the number of recombination nodes used in an ARG as an "upper bound" on $Rmin(M)$.

The sequences examined were sampled from three populations—namely, Jackson, North Karelia, and Rochester populations. In the analysis, sites with missing or non-SNP data was removed, ignoring insertions/deletions, unphased sites, and sites with missing data. This is the treatment of the data that was used in [20]. Following Myers and Griffiths, sites of the LPL data were partitioned into three regions (c.f. Table 5 of [19]). It has been suggested that region 2 corresponds to a recombination hotspot[**?**].

The left hand side of Table 1.1 gives a summary of the lower bounds produced by HapBound -S -M, and of the upper bounds on $Rmin$, i.e., the number of recombination nodes actually used in ARGs constructed for the data. The three populations were considered separately as well as together. HapBound -S and HapBound -S -M produced similar lower bounds. The only difference was in region 2 of Jackson population; HapBound -S produced 9, whereas HapBound -S -M produced 10. The lower and upper bounds are generally quite close. In particular, they exactly match in each of the three regions for the Rochester population, and hence we know that $Rmin(M)$ exactly for those regions. Moreover, the lower bounds computed by HapBound were generally higher than the *Optimal RecMin Bound*. Optimal *RecMin* bounds are shown on the right hand side of 1.1 for comparison[3].

[3]This table differs from Table 5 of [19], because Myers and Griffiths did not remove insertion/deletion sites when they did their analysis.

	HapBound -S -M			The Optimal RecMin Bound		
Population	reg 1	reg 2	reg 3	reg 1	reg 2	reg 3
Jackson	11 (13)	10 (10)	13 (16)	10	9	12
N. Karelia	2 (2)	15 (17)	8 (10)	2	13	7
Rochester	1 (1)	14 (14)	8 (8)	1	12	7
All	13 (14)	21 (23)	25 (31)	12	21	22

Table 1.1: Lower and upper bounds for the LPL data, where the sites are partitioned into three regions. The numbers in parentheses are upper bounds, i.e., the actual number of recombination nodes used in ARGs constructed for the data. Lower bounds on the left hand side were computed using HapBound -S -M. The exact values of $Rmin$ for parts of this data will be reported in Section **??**.

Chapter 2

General ARG Construction Methods

In the previous chapter we considered the problem of constructing a MinARG, but only for the special case that there is a galled-tree for the input M. In this chapter we consider the problem of constructing good ARGs and MinARGs for arbitrary input M.

The problem of constructing a MinARG for a set of sequences M, or even of computing $Rmin(M)$, is NP-hard. Thus, we do not have, nor do we expect to have, a worst-case *polynomial-time* algorithm for those problems. Instead, we have heuristic algorithms that empirically run fast (and sometimes can be made to run in worst-case polynomial time) that produce ARGs with a number of recombination nodes "close" to $Rmin(M)$ on meaningful data. When comparing the number of recombination nodes in those ARGs to the highest available lower bound on $Rmin(M)$, we see that those methods often, but not always, produce MinARGs. We also have *exponential-time* or *superexponential-time* algorithms that compute $Rmin(M)$ exactly and that build MinARGs; these methods are practical for different, but generally small-sized, ranges of biological data. In this chapter, we will discuss two examples of the first kind of algorithm, implemented into programs *SHRUB* [27] and mARGarita [17]; and three examples of the second kind of algorithm, two of which have been implemented into programs *Beagle* [16] and *RecMinPath* [24, 26]. We will also mention two related methods and will discuss an extension of *SHRUB* to handle the case of gene-conversion [22, 23].

The ideas in *SHRUB* are related to a lower bound on $Rmin(M)$ called the *History Bound*. So after discussing *SHRUB*, we will discuss the History Bound, and another related lower bound, called the *Forest Bound*.

2.1 Building a MinARG in provably exponential time

Here we first establish the theoretical result that a MinARG for any input M can be constructed in time that is bounded by an *exponential* function of the size of M. While an exponential function grows very rapidly, even this time bound was not obvious when the construction of ARGs was first studied. That is reflected by the *super-exponential* time required by the first method [24, 26] that is guaranteed to find a MinARG (a method that we will detail in Section **??**), and the question posed in [2] of whether exponential-time MinARG methods are possible. As stated in [2]: "This problem is computationally challenging and has resisted efforts for even an exponential time algorithm."

Theorem 2.1.1 *A MinARG for any input M of n taxa and m sites, can be constructed by a method whose running time is bounded by an exponential function of nm, the size of M.*

Proof let $\mathcal{N}$ be a MinARG for M. We can assume that every node in $\mathcal{N}$ has a distinct label. $\mathcal{N}$ has at most m tree nodes since each is the head of an edge labeled by a distinct site of M. Also, $R(\mathcal{N})$, the number of recombination nodes in $\mathcal{N}$, is at most $nm/2$ since by Theorem **??** (page **??**) M can be generated on an ARG with only $nm/2$ recombinations. Therefore, the number of interior nodes in $\mathcal{N}$ and the number of distinct labels on those nodes is bounded by $nm + m$. So the enumerative algorithm in Figure 2.1 certainly finds a MinARG for M.

Now we will analyze the running time of *Algorithm MinARG*. The node labels in $\mathcal{N}$ are selected from the set of all 2^m binary sequences of length m. It follows that the set of distinct node labels in $\mathcal{N}$ is one of $\binom{2^m}{R(M)}$ sets. Now $\binom{2^m}{R(M)} \leq \binom{2^m}{nm)} \leq 2^{nm^2}$. Also, $\sum_{k=0}^{k=R(M)} \binom{2^m}{k} \leq \sum_{k=0}^{k=R(M)} 2^{mk} \leq R(M)2^{mR(M)} \leq nm2^{nm^2}$. So *Algorithm MinARG* examines at most $nm2^{nm^2}$ subsets of labels. When a subset K is examined, *Algorithm MinARG* tests whether K is self-derivable. By Theorem 1.4.3 (page 28), each self-derivability test can be done in $O(n^2m^2 2^n)$ time (although the polynomial terms can be reduced), so a MinARG for M can be found in $O(n^3m^3 2^{n^2m^2})$ time. ■

The time analysis shown in the proof of Theorem 2.1.1 is very crude and the algorithm can be sped up in several ways. The point is simply to show that it is possible to find a find a MinARG for M whose worst case running time is bounded by an exponential function of the size of M. In the remainder of this chapter, we present several methods to build ARGs or MinARGs that are more practical than *Algorithm MinARG*, or that have some additional conceptual value.

```
ALGORITHM MINARG (M)
    Let P(m) be the set of all binary sequences of length m
  for (k = n ... nm + m) do
    for each subset K of P(m) that has size k and includes M do
      Test if K is self-derivable.
      If it is, then stop and output the ARG constructed in that test.
    endfor
  endfor
```

Figure 2.1: *Algorithm MinARG* is guaranteed to find a MinARG for M.

2.2 ARG construction methods that destroy M

Theorem 2.1.1 shows that a MinARG for any M can be constructed in exponential time, but that is often too slow for practical construction. In this section we develop a different approach to ARG construction. The approach will be used in several practical methods for finding good ARGs (but ones that are not guaranteed to be MinARGs), and in one program that is guaranteed to construct MinARGs, and is practical on small to moderate sized data.

Many ARG construction methods build ARGs *backwards in time*, from the leaves of the ARG up to the root[1]. The primitive operations in those methods act on the input sequences, M, by removing (or "destroying") rows or columns of the matrix M, until it consists of a single row with no sites. An ARG $\mathcal{N}$ for M can be built in parallel with the destruction of M by associating each destructive step on M with a *constructive* step that adds to the growing ARG $\mathcal{N}$. This is a very common approach to ARG construction (and in reasoning about ARGs), so it is important to clearly understand how destructive operations on M translate into constructive operations for an ARG. In general there are *three* types of destructive operations, and we will describe each of them in this chapter. However, it is instructive to start with the simplest case, when M has a perfect-phylogeny with all-zero ancestral sequence. In that case, only two of the destructive operations are needed. So we begin with that case.

2.2.1 The perfect-phylogeny case with all-zero ancestral sequence

Matrix M has n rows and m columns, and each row f in M is associated with the singleton set $\{f\}$. Since the all-zero sequence is the required ancestral sequence, we assume that it is in M, adding it to M if it is not originally in M. As the matrix M is destroyed, we let $\tilde{M}$ denote the remaining submatrix of M. Initially, $\tilde{M} = M$. At any point in the destructive process, each row in the current $\tilde{M}$ will be associated with a subset of taxa, and those sets will partition the original n taxa of M. The two destructive rules remove columns and rows as follows:

[1]This follows the way that trees and ARGs are thought of, analyzed, and constructed in *coalescent theory* [11, 29].

ALGORITHM CLEAN (M)
Set $\tilde{M}$ to M.
Execute Rules **Dc** and **Dr** on $\tilde{M}$ in any order until neither rule applies.

Figure 2.2: Algorithm *Clean*

> **Rule Dc:** If a column c of $\tilde{M}$ contains at most one entry with value 1, then remove column c from $\tilde{M}$.
>
> **Rule Dr:** If two rows in $\tilde{M}$ are identical, let $\mathcal{K}$ be the union of the two sets of taxa associated with those two rows. Then *merge* the two rows (i.e. remove one) and associate the merged row with the set $\mathcal{K}$.

Assuming there is a perfect-phylogeny with all-zero ancestral sequence for M, the Algorithm *Clean*, shown in Figure 2.2, reduces M to a matrix containing a single row with no sites.

Note that the execution of Rule **Dc** may create the conditions where Rule **Dr** applies, and the converse is also true. See Figures 2.4 through 2.12. Since Rules **Dc** and **Dr** can be applied in any order, and to different columns and rows, it is conceivable that different executions of Algorithm *Clean* could produce different results. However, that is not true.

Lemma 2.2.1 *The resulting submatrix $\tilde{M}$ of M created by running Algorithm Clean on M is invariant over all executions of Algorithm Clean.*

Proof Consider two executions of Algorithm *Clean*. Let c be a column removed in some execution of Algorithm *Clean*, and let P be the series of operations before column c is removed. Clearly, any permutation of the operations in P that could be executed by Algorithm *Clean* would also lead to the conditions where Rule **Dc** applies to c. Moreover, the insertion of any additional operations into P (or a permitted permutation of P) that remove additional columns or merge additional pairs of rows still lead to the conditions where Rule **Dc** applies to c. So, in any execution of Algorithm *Clean* where all of the operations in P are applied, column c will eventually be removed. A similar argument can be made for any pair of rows that are merged.

So we will prove, by induction on the length of P, that every execution of Algorithm *Clean* will apply all the operations in P. Clearly, the first operation in P must be an application of Rule **Dc** or **Dr** that applies to an original column c' or original pair of rows $\{f, g\}$ in M. Two rows that are identical remain identical no matter what columns are removed, and a column with at most one entry of value 1 continues to have that property no matter what rows are merged. So the first rule applied in P will also be applied somewhere in any other execution of Algorithm *Clean*. So the basis of the inductive claim

is established. Next, assume that the inductive claim is true for P of length $k > 1$, and consider P of length $k+1$. By the inductive hypothesis, in any other execution of Algorithm *Clean*, the first k operations of P will be applied. So, as argued above, the conditions required for the application of the operation $k+1$ of P will eventually be established in any execution of Algorithm *Clean*, and so that operation will eventually be applied. ■

The parallel construction of a perfect-phylogeny To construct a perfect-phylogeny T for M with all-zero ancestral sequence (assuming one exists), we start with a forest $\tilde{\mathcal{F}}$ containing one node v and one unlabeled edge directed into v for each taxon f in M; we associate the singleton set $\{f\}$ with node v. Then the forest grows and coalesces to a single tree as *Algorithm Clean(M)* is executed. At any point in construction of T, we say that a node v in the growing forest $\tilde{\mathcal{F}}$ is *associated* with the set of taxa that label the leaves in the subtree rooted at v. Two nodes might be associated with the same subset of taxa, but if this happens then one of the nodes will be an ancestor of the other. Therefore, for any subset of taxa that is associated with some node(s) in $\tilde{\mathcal{F}}$, there is a well-defined *most ancestral* node that is associated with that subset. The constructive rules will easily imply (inductively) that if any row in the current $\tilde{M}$ is associated with a set of taxa $\mathcal{K}$, then some node in the current $\tilde{\mathcal{F}}$ will also be associated with $\mathcal{K}$.

Each execution of destructive Rule **Dc** or **Dr** triggers a parallel execution of constructive Rule **Cc** or **Cr** respectively. Those constructive rules are:

> Rule **Cc:** If column c has exactly one entry with value 1, then let $\tilde{r}$ be the row in $\tilde{M}$ that contains the only entry of value 1 in column c. Let $\mathcal{K}$ be the subset of taxa associated with row $\tilde{r}$ in $\tilde{M}$.
>
> If $|\mathcal{K}| = 1$ (so that $\tilde{r}$ is an original row in M), let e be the (existing) unlabeled edge in $\tilde{\mathcal{F}}$ directed into the leaf for $\tilde{r}$. Then add the label c to edge e.
>
> If $|\mathcal{K}| > 1$, then create a new node v and an edge directed from v into the most ancestral node u in $\tilde{\mathcal{F}}$ associated with $\mathcal{K}$, and then label edge (v, u) with c.
>
> **Rule Cr:** Suppose the two identical rows in the current $\tilde{M}$ are associated with the subsets of taxa $\mathcal{K}$ and $\mathcal{K}'$, and let v and v' be the most ancestral nodes in $\tilde{\mathcal{F}}$ associated with $\mathcal{K}$ and $\mathcal{K}'$ respectively. Then merge nodes v and v' into a single node.

Note that Rule **Cc** corresponds to creating a *mutation event* in the growing $\tilde{\mathcal{F}}$, and that Rule**Cr** corresponds to creating a *coalescent event* in the growing forest. Neither of these rules corresponds to creating a recombination event. Later, we will add a third rule that will do exactly that. Note also that if Rule

Dc removes a column that only contains zeros, then Rule **Cc** does not apply and no change to $\mathcal{F}$ is made.

Algorithm *Clean-Build*, shown in Figure 2.3 completely destroys M and builds a perfect-phylogeny T for M with all-zero ancestral sequence, if one exists. Otherwise, it constructs a forest of two or more trees.

```
ALGORITHM CLEAN-BUILD (M)

  while (Rule Dc or Dr applies) do
    if (Rule Dc applies) then
      execute Rules Dc and Cc.
    endif

    if (Rule Dr applies) then
      execute Rules Dr and Cr.
    endif
  endwhile

  return the resulting matrix M̃ and the forest (possibly a single tree) F̃.
```

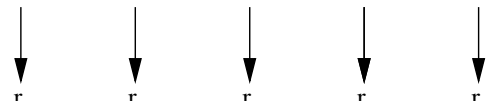
Figure 2.3: Algorithm *Clean-Build*

A complete example of the execution of Algorithm *Clean-Build* in the case that M has a perfect-phylogeny with all-zero ancestral sequence, is shown in Figures 2.4 through 2.13.

Another view of Algorithm *Clean-Build* There is another helpful way to view Algorithm *Clean-Build*. Since M has a perfect-phylogeny with all-zero ancestral sequence, any submatrix $\tilde{M}$ of M, and in particular any submatrix created by Algorithm *Clean-Build*, will also have a perfect-phylogeny with all-zero ancestral sequence. In general, let $\tilde{T}$ denote the unique perfect-phylogeny for $\tilde{M}$ with all-zero ancestral sequence. Since any row $\tilde{r}$ of $\tilde{M}$ is represented by

	1	2	3	4
r_1	0	0	1	0
r_2	0	0	1	0
r_3	1	1	0	1
r_4	1	1	0	0
r_5	1	0	0	0

(a) The original matrix M

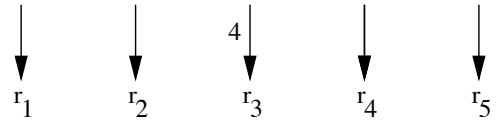


(b) The initial forest $\tilde{\mathcal{F}}$

Figure 2.4: Matrix M has a perfect-phylogeny. Site 4 has only one entry of 1 (in row r_3), so Rules **Dc** and **Cc** can be applied. See Figure 2.5 for the results.

	1	2	3
r_1	0	0	1
r_2	0	0	1
r_3	1	1	0
r_4	1	1	0
r_5	1	0	0

(a) $\tilde{M}$ after removal of site 4.

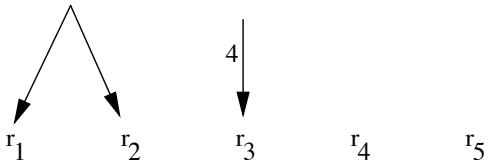


(b) The modified $\tilde{\mathcal{F}}$

Figure 2.5: Rule **Dc** removed site 4 from the matrix in Figure 2.4. Rule **Cc** added label 4 to the edge directed into leaf r_3. Now rows r_1 and r_2 are identical, so Rule **Dr** can be applied. See Figure 2.6 for the results.

	1	2	3
$\{r_1, r_2\}$	0	0	1
r_3	1	1	0
r_4	1	1	0
r_5	1	0	0

(a) The modified $\tilde{M}$

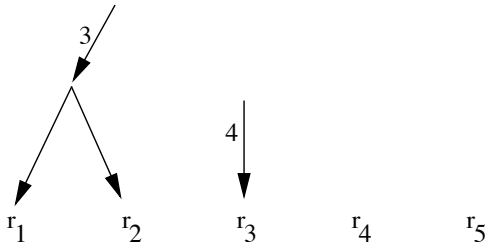


(b) The modified $\tilde{\mathcal{F}}$

Figure 2.6: Rule **Dr** merged rows r_1 and r_2 of the matrix in Figure 2.5 into one row labeled $\{r_1, r_2\}$. Rule **Cr** merged the parents of leaves r_1 and r_2 into single node associated with the set $\{r_1, r_2\}$. As a result, site 3 now has only a single entry with value 1 (in the row for $\{r_1, r_2\}$), so Rules **Dc** and **Cc** can be applied. See Figure 2.7 for the results.

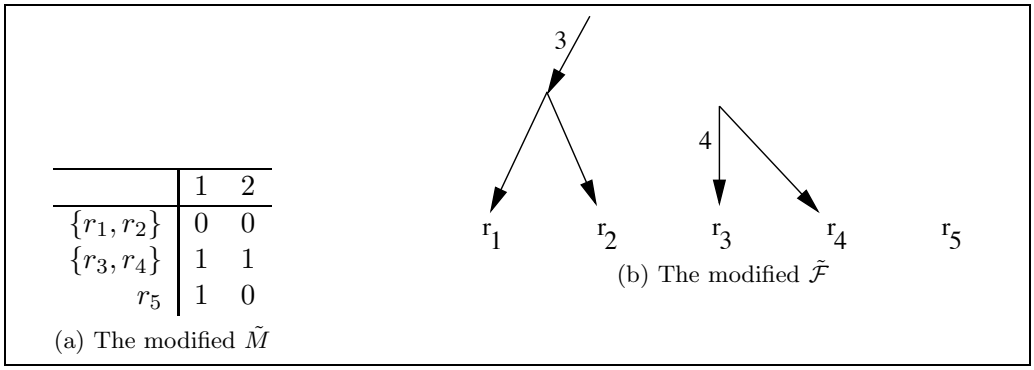
	1	2
$\{r_1, r_2\}$	0	0
r_3	1	1
r_4	1	1
r_5	1	0

(a) The modified $\tilde{M}$



(b) The modified $\tilde{\mathcal{F}}$

Figure 2.7: Rule **Dc** removed site 3 from the matrix in Figure 2.6. Rule **Cc** created an edge directed into the most ancestral node associated with set $\{r_1, r_2\}$, and labeled that edge with site 3. Note that the new node is now the most ancestral node associated with $\{r_1, r_2\}$. Rows r_3 and r_4 are now identical, so Rules **Dr** and **Cr** can be applied. See Figure 2.8 for the results.

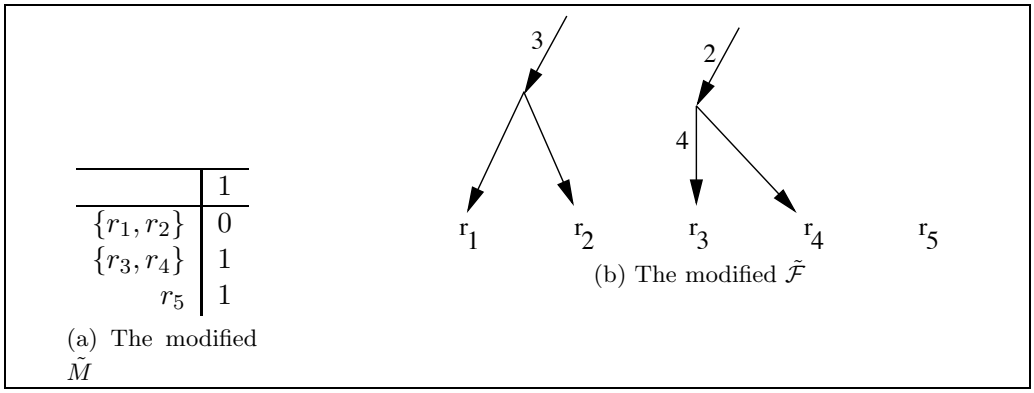


	1	2
$\{r_1, r_2\}$	0	0
$\{r_3, r_4\}$	1	1
r_5	1	0

(a) The modified $\tilde{M}$

(b) The modified $\tilde{\mathcal{F}}$

Figure 2.8: Rule **Dr** merged rows r_3 and r_4 in the matrix from Figure 2.7 into one row associated with $\{r_3, r_4\}$. Rule **Cr** merged the parents of r_3 and r_4 into a single node associated with the set $\{r_3, r_4\}$. Site 2 now only contains one entry with value 1, so Rules **Dc** and **Cc** can be applied. See Figure 2.9 for the results.

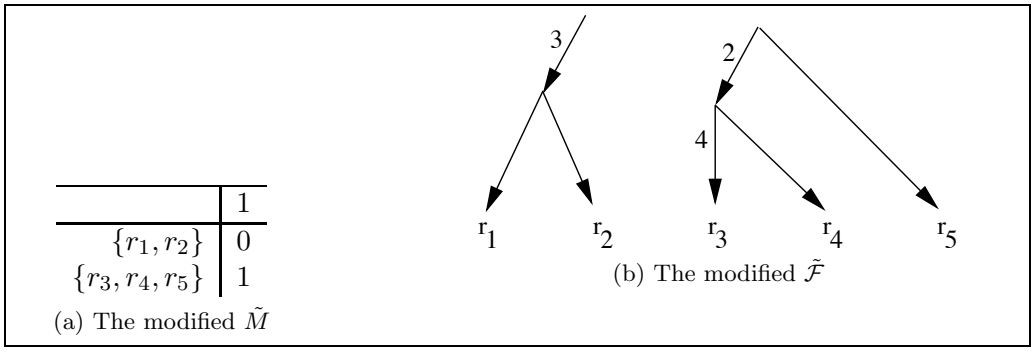


	1
$\{r_1, r_2\}$	0
$\{r_3, r_4\}$	1
r_5	1

(a) The modified $\tilde{M}$

(b) The modified $\tilde{\mathcal{F}}$

Figure 2.9: Rule **Dc** removed site 2 from the matrix in Figure 2.8. Rule **Cc** created a new node and an edge directed into the most ancestral node associated with the set $\{r_3, r_4\}$, and labeled that edge with site 2. The new node is now the most ancestral node associated with $\{r_3, r_4\}$. Rows $\{r_3, r_4\}$ and r_5 are now identical, so Rules **Dr** and **Cr** can be applied. See Figure 2.10 for the results.

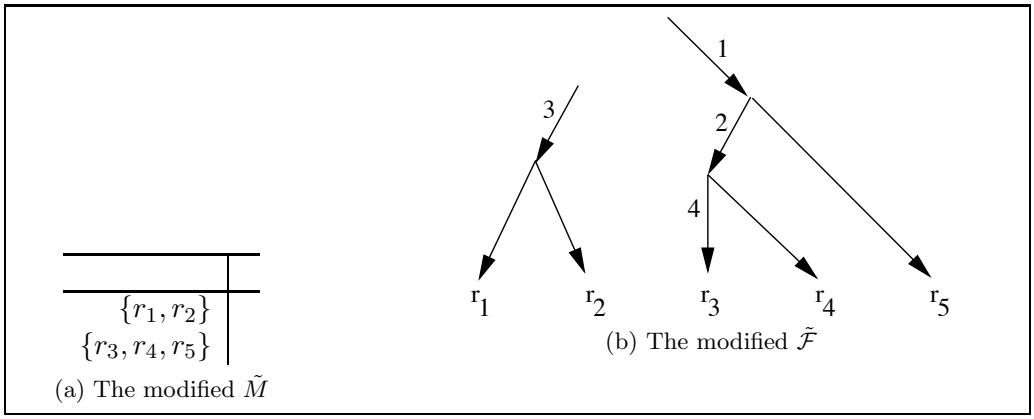


	1
$\{r_1, r_2\}$	0
$\{r_3, r_4, r_5\}$	1

(a) The modified $\tilde{M}$

(b) The modified $\tilde{\mathcal{F}}$

Figure 2.10: Rule **Cr** merged rows associated with $\{r_3, r_4\}$ and $\{r_5\}$ from Figure 2.9 into one row labeled $\{r_3, r_4, r_5\}$. Rule **Cr** merged the parents of leaf r_5 and the most ancestral node associated with $\{r_3, r_4\}$. Site 1 now has only a single entry with value 1, so Rules **Dc** and **Cc** can be applied. See Figure 2.11.

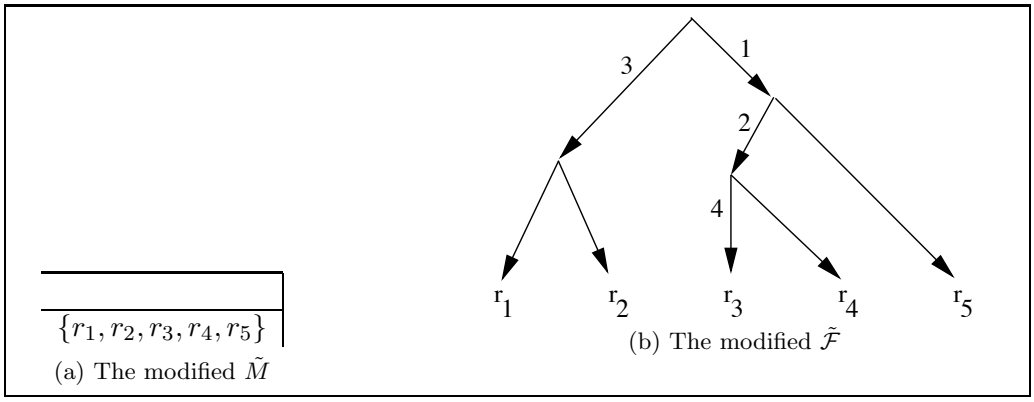


$\{r_1, r_2\}$	
$\{r_3, r_4, r_5\}$	

(a) The modified $\tilde{M}$

(b) The modified $\tilde{\mathcal{F}}$

Figure 2.11: Rule **Dc** removed site 1 from the matrix in Figure 2.11. Rule **Cc** created a new node and an edge directed into the most ancestral node associated with the $\{r_3, r_4, r_5\}$, and labeled the edge with site 1. All entries in M have now been removed, but two rows remain, labeled $\{r_1, r_2\}$, and $\{r_3, r_4, r_5\}$. These rows are identical, so Rules **Dr** and **Cr** can be applied. See Figure 2.12 for the results.



(a) The modified $\tilde{M}$

(b) The modified $\tilde{\mathcal{F}}$

Figure 2.12: Rule **Dr** merged the two rows of Figure 2.11 into a single row labeled $\{r_1, r_2, r_3, r_4, r_5\}$. Rule **Cr** merged the most ancestral node associated with $\{r_3, r_4, r_5\}$ and the most ancestral node associated with $\{r_1, r_2\}$. Matrix M has now been reduced to a single row, associated with all of the taxa, and with no sites. The construction now consists of a single tree whose root node is associated with all of the taxa. The tree is a perfect-phylogeny for M. See Figure 2.13.

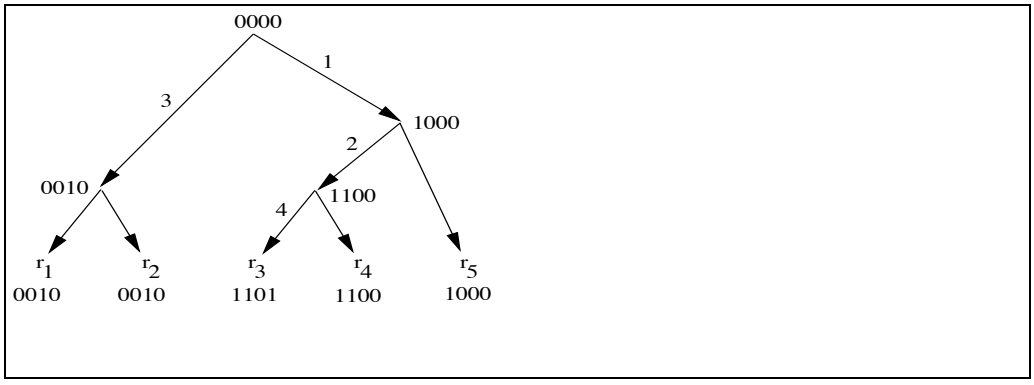


Figure 2.13: The perfect-phylogeny T for M with all of the node labels.

a leaf v in $\tilde{T}$, if $\tilde{r}$ is associated with a subset $\mathcal{K}$ of taxa of M, we also associate $\mathcal{K}$ leaf v. The following two lemmas are easy to establish by inducting on the number of rules applied in Algorithm *Clean-Build*.

Lemma 2.2.2 *In Algorithm Clean-Build when Rule* **Dc** *applies to a site c in $\tilde{M}$, the edge labeled c in $\tilde{T}$ must be directed into a leaf of $\tilde{T}$.*

Lemma 2.2.3 *In Algorithm Clean-Build, when Rule* **Dr** *merges two rows of $\tilde{M}$ that are associated with subsets of taxa $\mathcal{K}$ and $\mathcal{K}'$, there must be two sibling leaves in $\tilde{T}$ that are associated with $\mathcal{K}$ and $\mathcal{K}'$ respectively. Further, the directed edges into those two sibling leaves must be unlabeled.*

Given Lemmas 2.2.2 and 2.2.3, an insightful way to view the action of Algorithm *Clean-Build* is with the following thought experiment. Imagine that we know the unique perfect-phylogeny T for M, with all-zero ancestral sequence. Imagine also that for every destructive operation on M, we will execute a parallel *destructive* operation on T. At the start of an execution of Algorithm *Clean-Build*, each leaf in T will be associated with the taxon that labels it. Then, in the thought experiment, whenever Rule **Dc** applies to a site c in $\tilde{M}$, remove the label c from the edge (into a leaf) in $\tilde{T}$ labeled by c. That edge exists by Lemma 2.2.2. Also, whenever Rule **Dr** applies to two rows in $\tilde{M}$, let v and v' be the sibling leaves in $\tilde{T}$ associated with sets $\mathcal{K}$ and $\mathcal{K}'$. Those sibling leaves exist by Lemma 2.2.3. Then in the thought experiment, associate the parent node of leaves v and v' with $\mathcal{K} \cup \mathcal{K}'$, and remove the v and v' and the edges into them from $\tilde{T}$.

Figure 2.14 shows the seven steps of this thought experiment corresponding to the first seven steps in the destruction of M shown in Figures 2.4 to 2.11. The next step would remove the two leaves and label the one remaining node with the full set of taxa of M. Lemmas 2.2.2 and 2.2.3, also lead to the following

Theorem 2.2.1 *Let T be the unique perfect-phylogeny for M with all-zero ancestral sequence. Let $\tilde{M}$ be a submatrix of M created during an execution of Algorithm Clean-Build. Then $\tilde{T}$ is is a subtree of T, rooted at the root node of T.*

The point of the thought experiment is that the (assumed) known T is destroyed from the leaves upward, in parallel with the destruction of M, and in parallel with the construction of unknown T, by Algorithm *Clean-Build*. When M is reduced to a single row with no sites, T is reduced to a single node with no edges, and that node is associated with the set of all the taxa in M. Essentially, the forest construction part of Algorithm *Clean-Build* constructs edges and labels of the perfect-phylogeny corresponding to edges and labels of T that the destructive thought experiment removes. The actual correspondence is left to

the reader but the idea is simple: as columns and rows of are removed from M, edge labels and sibling edges are removed from the original T, and corresponding labels, sibling edges and labeled edges are added to the growing $\tilde{\mathcal{F}}$ in Algorithm *Clean-Build*.

Formal Correctness Now we can formally prove the correctness of Algorithm *Clean-Build*.

Theorem 2.2.2 *The set of sequences M can be derived on a perfect-phylogeny T with all-zero ancestral sequence if and only if Algorithm Clean-Build reduces M to a single row containing no sites; and if and only if the forest constructed by the algorithm is the perfect-phylogeny T for M.*

Proof We first prove the "only if" side of the theorem, so suppose that M can be derived on a perfect-phylogeny T with all-zero ancestral sequence. Clearly, the theorem holds if T contains only a single node, so that the corresponding M contains a single row with no entries; or if T contains only a single labeled edge, so M contains a single row and a single column with an entry of value 1.

More generally, note that if M has been reduced to a single row, but there are sites remaining, those sites will be removed by application of Rule **Dc**. So, if M is reduced to a single row, it can be reduced to a row containing no sites. Therefor, for contradiction, assume that there is a counter-example to the theorem, i.e., a matrix M that can be derived on a perfect-phylogeny T with all-zero ancestral sequence, but Algorithm *Clean-Build* does not reduce M to a single row. Then there is a counter-example with the minimum number of sites, and among such counter-examples, there is one with a minimum number of taxa. Let M be such a minimal counter-example. Now consider the deepest leaf node v in the perfect phylogeny T for M and suppose it is labeled by taxon f. Recall that all interior nodes of a perfect-phylogeny must have degree at least three, so the parent of v must have at least two children. Therefore, v has a sibling v' which is also a leaf node. Let f' be the taxon labeling v'.

If the edge into v or v' (say v) is labeled by a site c, then site c in M has only a single 1 entry and Rule **Dc** applies. Removing site c from M creates a matrix $\tilde{M}$, and removing label c from the edge in T into v creates a perfect-phylogeny $\tilde{T}$ for $\tilde{M}$. $\tilde{M}$ has fewer sites than M, so the theorem applies to $\tilde{M}$. Therefore, Algorithm *Clean-Build* reduces $\tilde{M}$ to a single row, showing that M also reduces to a single row.

If neither edge into v or v' is labeled by a site, then the two rows for taxa f and f' are identical and Rule **Dr** applies. Merging the rows for f and f' creates a matrix $\tilde{M}$; removing v' from T and the edge into v', and labeling v with $\{f, f'\}$ creates a perfect-phylogeny for $\tilde{M}$. $\tilde{M}$ has fewer rows than M, so the theorem applies to $\tilde{M}$, and so Algorithm *Clean-Build* reduces $\tilde{M}$, and M, to a single row.

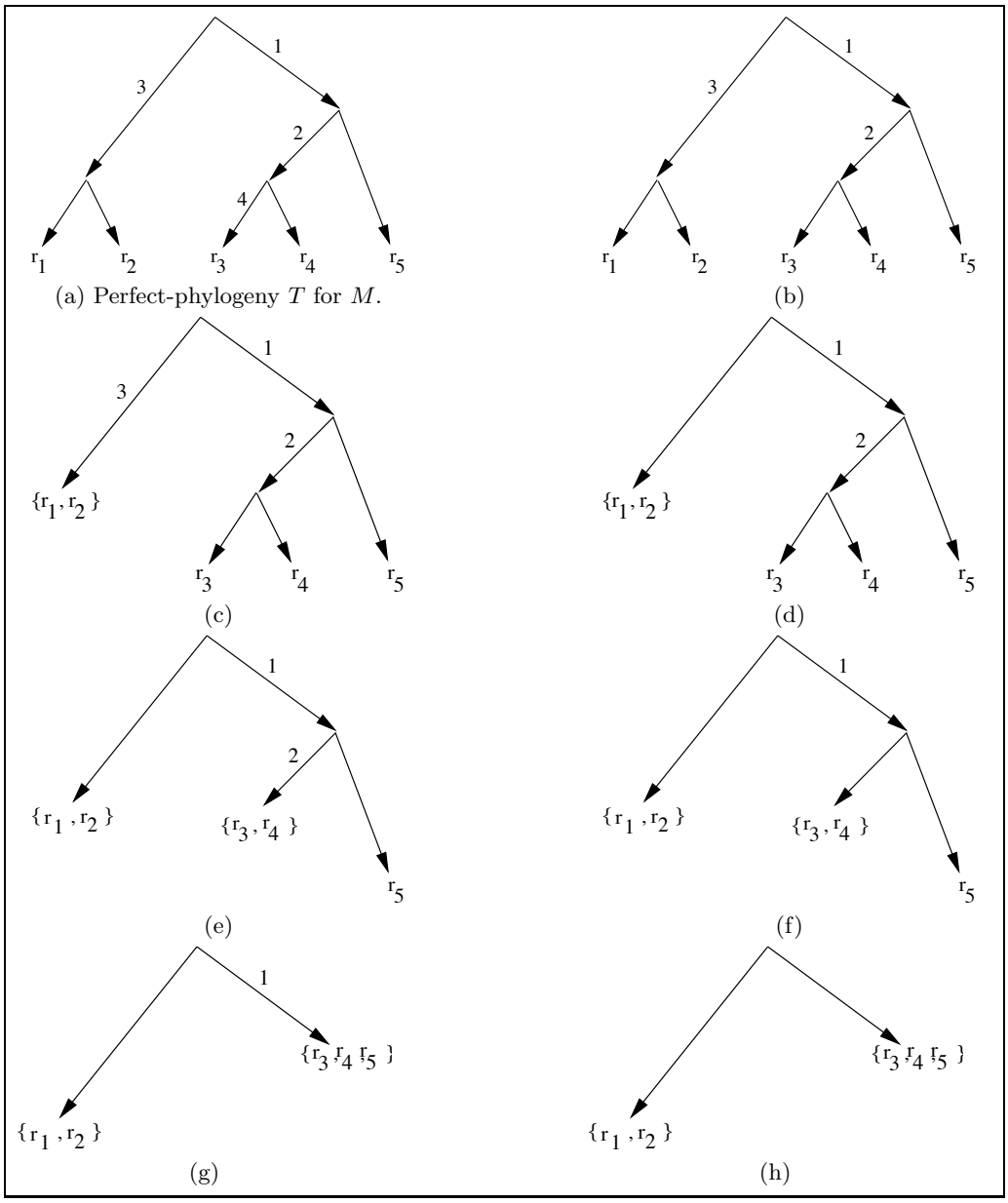


Figure 2.14: Seven steps of the thought experiment corresponding to the first seven destructive operations on M shown in Figures 2.4 through 2.12. The last step would merge the remaining two leaves into a single node associated with all the taxa in M.

We have therefore proved that if M can be derived on a perfect-phylogeny T then Algorithm *Clean-Build* reduces M to a single row with no sites.

In a similar way, we can prove by contradiction that if there is a perfect-phylogeny T for M, with all-zero ancestral sequence, then Algorithm *Clean-Build* will construct T. But it is more insightful to understand constructively that as M is destroyed, the algorithm builds a set of trees (a forest) that generate the sequences defined by submatrices of M of increasing size. In particular, at any point in the algorithm, the submatrix consists of those rows and those columns that have been involved in an application of Rule **Dc** or **Dr**. Therefore, when M is reduced to a single row with no sites, the set of trees consists of a single tree, i.e., the unique perfect-phylogeny T for M. In more detail, assuming that there is a perfect phylogeny T for M, with all-zero ancestral sequence, each application of Rule **Cc** or **Cr** identifies a *forced* feature (a new edge label, a new labeled edge, or the merging of two nodes) of the unique perfect-phylogeny T for M. That is what is established by Lemmas 2.2.2 and 2.2.3. For example, if site c in the current $\tilde{M}$ has only a single 1 (so Rule **Dc** applies) and the 1 is in row $\tilde{r}$ associated with the subset of taxa $\mathcal{K}$, then in T there must be an edge into a node associated with $\mathcal{K}$, and that edge must be labeled with site c. Hence Rule **Cc** is forced. Similarly, when Rule **Dr** applies, Rule **Dr** is forced. So, as M is destroyed, the algorithm constructs features of T that are forced, finishing the proof of the only-if side of the theorem.

To prove the "if" side of the theorem, suppose that Algorithm *Clean-Build* reduces M to a single row with no sites. As above, as M is destroyed, a perfect-phylogeny T for M is constructed, so certainly a perfect-phylogeny for M exists. ■

2.2.2 The case of the root-unknown perfect-phylogeny

Above we assumed that M can be derived on a perfect-phylogeny with all-zero ancestral sequence. It is easy to extend that to the case of a different *known* ancestral sequence in the same way that the perfect-phylogeny problem with any known ancestral sequence can be reduced to the case of the all-zero ancestral sequence (see Section **??**). But if no ancestral sequence is known, then we modify the approach as follows. First assume that no column in M contains only zeros or only ones. Next, modify Rule **Dc** so that column c is removed if it contains only a single 1 or only a single 0. Such a column is called *uninformative*. The parallel Rule **Cc** is not changed; the edge created is labeled with site c. However, the ancestral state of c is set whenever Rule **Cc** is applied: If column c contains only a single 1, the ancestral state of c is set to 0; if column c only contains a single 0, the ancestral state of c is set to 1. Rules **Dr** and **Cr** are not changed. When these variants of Rules **Dc** and **Cc** are used in Algorithms *Clean* and *Clean-Build*, the resulting algorithms will be called *RU-Clean* and *RU-Clean-Build*.

2.2.3 The general ARG case

We now move from the case of perfect-phylogeny to ARGs that must contain recombination nodes. We again assume that the ancestral sequence is required to be the all-zero sequence. By Theorem 2.2.2, if Algorithm *Clean-Build* reduces M to a single row with no sites, then M can be derived on a perfect-phylogeny with all-zero ancestral sequence. Hence if there is no such perfect-phylogeny for M, then any execution of Algorithm *Clean-Build* must reach a point where $\tilde{M}$ still contains entries, but neither rules **Dc** nor **Dr** apply. Let $\tilde{\mathcal{F}}$ be the forest constructed by Algorithm *Clean-Build* at that point; let D be the set of sites *removed* from M; and let R be the set of taxa involved in any application of Rule **Dr**, i.e., the set of rows in M that were part of some merge(s). $M(D)$ then represents the submatrix of M restricted to the sites in D. Application of Lemmas 2.2.2 and 2.2.3 imply the following

Lemma 2.2.4 *The set of sequences in $M(D)$ can be derived on the unique forest $\tilde{\mathcal{F}}$ of perfect-phylogenies, each with all-zero ancestral sequence.*

As an example, consider the matrix M and the ARG for it shown in Figure 2.15. Since sites r_2 and r_4 are incompatible (after adding the all-zero ancestral sequence to M), there is no perfect-phylogeny for M with all-zero ancestral sequence. Algorithm *Clean-Build* removes columns 1 and 3 and merges rows r_3 and r_4. The resulting matrix $\tilde{M}$ is shown in Figure 2.16 a); the forest $\tilde{\mathcal{F}}$ is shown in Figure 2.16 b); and the matrix $M(D)$ is shown in Figure 2.16 c). As stated in Lemma 2.2.4, $\tilde{\mathcal{F}}$ is a forest of three perfect-phylogenies, each with all-zero ancestral sequence, that generate the sequences $M(D)$. Also, as in the thought experiment done for the case of perfect-phylogenies, $\tilde{\mathcal{F}}$ is a forest of trees that *hangs off* the periphery of the ARG for the sequences in $\tilde{M}$ shown in Figure 2.15. That observation will be formalized below in Theorem 2.2.3.

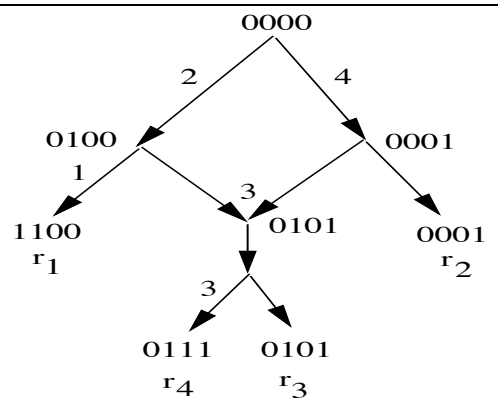
So, at the point in Algorithm *Clean-Build* where neither Rule **Dr** nor **Dc** applies, the algorithm has constructed a forest $\tilde{\mathcal{F}}$ of perfect-phylogenies whose leaves are labeled by the taxa in M, and whose edges are either unlabeled or are labeled by the sites in D. Intuitively, this forest $\tilde{\mathcal{F}}$ should be part of a larger ARG for M, and it should form part of the "bottom" of the ARG. We formalize and prove that next.

Theorem 2.2.3 *There exists an ARG $\mathcal{N}$ for the original M that contains the forest $\tilde{\mathcal{F}}$. More exactly, if $\tilde{\mathcal{F}}$ consists of k trees, then there are k nodes in $\mathcal{N}$ that root the trees in $\tilde{\mathcal{F}}$.*

Proof Let $\tilde{M}$, $\tilde{\mathcal{F}}$, D and R be as defined above. By construction, $\tilde{M}$ does not contain any of the sites in D, but for each tree in $\tilde{\mathcal{F}}$, $\tilde{M}$ does contain one row $\tilde{r}$ associated with the taxa labeling the leaves of that tree. Let $T_{\tilde{r}}$ denote that tree, and let $\mathcal{T}(\tilde{r})$ be the set of taxa of M labeling the leaves of $T_{\tilde{r}}$ (and

M	1	2	3	4
r_1	1	1	0	0
r_2	0	0	0	1
r_3	0	1	0	1
r_4	0	1	1	1

(a) The original matrix M



(b) An ARG for M

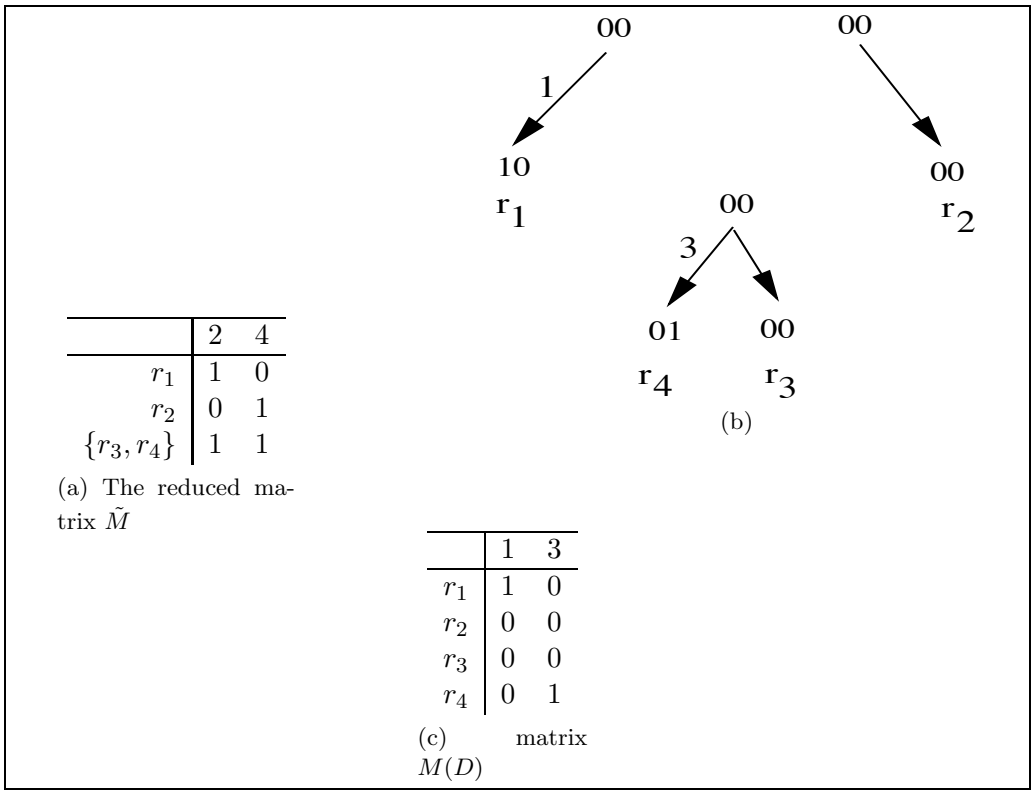
Figure 2.15: Matrix M does not have a perfect-phylogeny when the required ancestral sequence is the all-zero sequence.

hence associated with row $\tilde{r}$ of $\tilde{M}$). Let $S_{\tilde{r}}$ be the sequence in $\tilde{M}$ in row $\tilde{r}$. Two rows are merged in an execution of Algorithm *Clean-Build* only when the two rows are identical in $\tilde{M}$, so in the original M all of the taxa in $\mathcal{T}(\tilde{r})$ must be identical at all of the sites in $\tilde{M}$. More exactly, when restricted to the sites in $\tilde{M}$, each sequence in M for the taxa in $\mathcal{T}(\tilde{r})$ must be identical to the sequence $S_{\tilde{r}}$.

Now let $\tilde{N}$ be an ARG for $\tilde{M}$. Since $\tilde{r}$ is a taxon in $\tilde{M}$, there must be a node $\tilde{v}$ in $\tilde{\mathcal{N}}$ labeled by $S_{\tilde{r}}$. If we attach the root of tree $T_{\tilde{r}}$ at node $\tilde{v}$, the result is an ARG that generates the sequences in M for all the taxa in $\mathcal{T}(\tilde{r})$. Let $\mathcal{N}$ be the ARG with all-zero ancestral sequence created by repeating this operation for each tree in $\tilde{\mathcal{F}}$, i.e., attaching each tree to the appropriate node in $\tilde{\mathcal{N}}$. Clearly, the ARG $\mathcal{N}$ correctly generates all the sequences in R. We need to prove that it also correctly generates the sequences not in R. By assumption, $\tilde{N}$ correctly generates those sequences at all the sites not in D. We next show that each sequence not in R has a value of 0 at any site in D.

Every site in D must label an edge in some tree in $\tilde{\mathcal{F}}$. Consider a site c in D that labels an edge of tree $T_{\tilde{r}}$. Clearly, only taxa in R can label leaves of $T_{\tilde{r}}$, so we need to prove that any taxon not in R has a value of 0 at site c. That is true because at the point in Algorithm *Clean-Build* when a site c was removed, site c had only one entry with a value of l, so in M, no taxon outside of $\mathcal{T}(\tilde{r})$ can have a value of 1 at site c. ■

Given Theorem 2.2.3, an ARG for M can be built by first applying Algorithm *Clean-Build* until no further application of Rule **Dc** or **Dr** is possible,



(b)

	2	4
r_1	1	0
r_2	0	1
$\{r_3, r_4\}$	1	1

(a) The reduced matrix $\tilde{M}$

	1	3
r_1	1	0
r_2	0	0
r_3	0	0
r_4	0	1

(c) matrix $M(D)$

Figure 2.16: Sites 1 and 3 have been removed from the matrix M shown in Figure 2.15, and rows r_3 and r_4 have been merged. The set D of removed columns is $\{1, 3\}$. The forest $\tilde{\mathcal{F}}$, restricted to the sites in D, is shown in (b), and the matrix $M(D)$ is shown in (c). As stated in Lemma 2.2.4, forest $\tilde{\mathcal{F}}$ generates the sequences in $M(D)$.

creating the forest $\tilde{\mathcal{F}}$ and the resulting matrix $\tilde{M}$. Then, recursively, an ARG $\tilde{\mathcal{N}}$ for $\tilde{M}$ can be constructed; after that, $\tilde{\mathcal{F}}$ can be attached to $\tilde{\mathcal{N}}$ as in the proof of Theorem 2.2.3. Since there are no recombination nodes in $\tilde{\mathcal{F}}$, the number of recombination nodes for the resulting ARG $\mathcal{N}$ will be the number of recombination nodes in $\tilde{\mathcal{N}}$.

Now how can a good ARG for $\tilde{M}$ be constructed? We know that no application of Rule **Dc** or **Dr** on $\tilde{M}$ is possible. Instead, we introduce the *third* destructive rule that operates on $\tilde{M}$, and the third constructive rule that builds part of $\tilde{N}$, again "bottom up".

The Third Destructive Rule

> Rule **Dt:** If neither Rule **Dc** nor **Dr** can be applied, pick a row $\tilde{r}$ in the current $\tilde{M}$ (other than the all-zero row that corresponds to the ancestral sequence) and remove row $\tilde{r}$ from $\tilde{M}$.

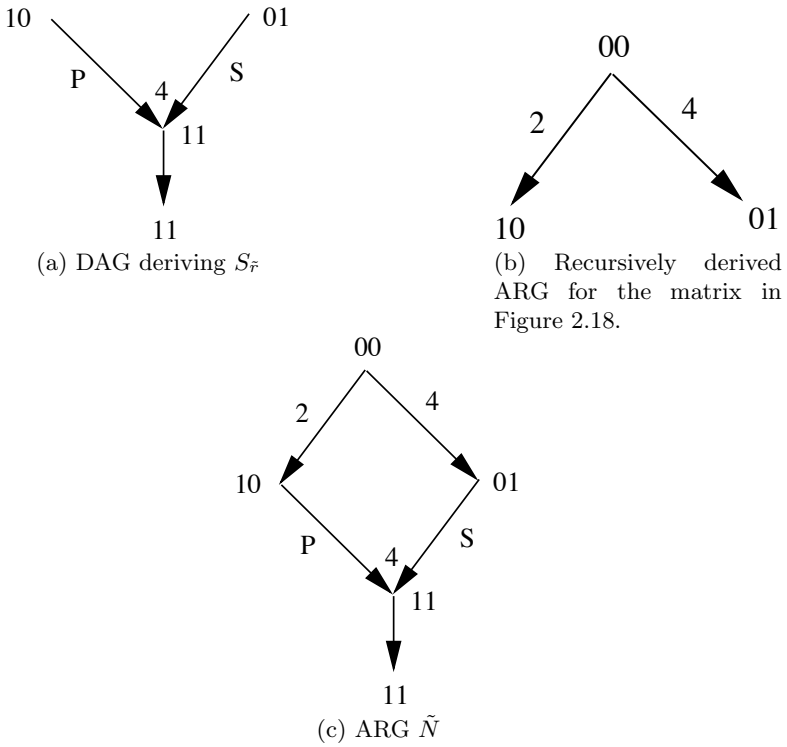
For clarity of the discussion below, we will use $\tilde{M}_1$ to refer to the matrix $\tilde{M}$ before an application of Rule **Dt**. $\tilde{M}$ will always refer to the current matrix incorporating all of the removals of columns and rows due to applications of Rules **Dc, Dr, Dt**. We use $S_{\tilde{r}}$ to denote the sequence removed from $\tilde{M}_1$.

The Third Constructive Rule *After* the application of Rule **Dt**, we use the third *constructive* Rule to begin the construction of $\tilde{N}$:

> Rule **Ct:** Find a series of recombinations (without mutations) of sequences in $\tilde{M} = \tilde{M}_1 - S_{\tilde{r}}$ that derive sequence $S_{\tilde{r}}$. (This will always be possible, as explained below.) Then construct a DAG containing one node for each sequence in $\tilde{M}$ and one node for $S_{\tilde{r}}$ and all needed edges and additional recombination nodes specified by the recombinations that derive $S_{\tilde{r}}$ from $\tilde{M}$.

The DAG constructed by application of Rule **Ct** forms a bottom or peripheral part of the desired ARG $\tilde{N}$ for $\tilde{M}_1$. The full ARG $\tilde{N}$ for $\tilde{M}_1$ is formed by adding that DAG to an ARG that derives $\tilde{M}$. Of course, the ARG for $\tilde{M}$ is found by recursive application of the entire method.

For example, consider the reduced matrix in Figure 2.16, and denote it $\tilde{M}_1$. If we pick $\tilde{r}$ to be the row labeled $\{r_3, r_4\}$, sequence $S_{\tilde{r}} = 11$ can be generated by recombining the two sequences in rows r_1 and r_2. The DAG representing the generation of $S_{\tilde{r}}$ is shown in Figure 2.17 a). Then, the algorithm must recursively find an ARG for the matrix $\tilde{M}$ shown in Figure 2.18. That ARG is the tree shown in Figure 2.17 b). The ARG $\tilde{\mathcal{N}}$ for $\tilde{M}_1$, shown in Figure 2.17 c), is formed by combining the DAG in Figure 2.17 a) with the ARG in Figure 2.17 b). The full ARG $\mathcal{N}$ for M, shown in Figure 2.15, is formed by adding the forest



(a) DAG deriving $S_{\tilde{r}}$

(b) Recursively derived ARG for the matrix in Figure 2.18.

(c) ARG $\tilde{N}$

Figure 2.17: ARG $\tilde{N}$ for $\tilde{M}_1$ is constructed by adding the DAG deriving $S_{\tilde{r}}$ to the ARG for the matrix in Figure 2.18.

$\tilde{\mathcal{F}}$ shown in Figure 2.16 b) to ARG $\tilde{\mathcal{N}}$. If instead of picking $\tilde{r}$ to be the row $\{r_3, r_4\}$, we pick row r_2, the resulting ARG for M is shown in Figure 2.19. In this example, both choices for $\tilde{r}$ result in an ARG with only one recombination node. In general, however, different choices for $\tilde{r}$ can result in different numbers of recombination nodes in the ARG for M. In this example, the DAG generating $S_{\tilde{r}}$ only has one recombination node, but that need not be true in general, and the recursively found ARG is a tree, but that also need not be true in general. For a more involved example, see Figure 1 in [27].

	2	4
r_1	1	0
r_2	0	1

Figure 2.18: The reduced matrix $\tilde{M}$ after Rule **Dt** removes row $\tilde{r}$, labeled $\{r_3, r_4\}$.

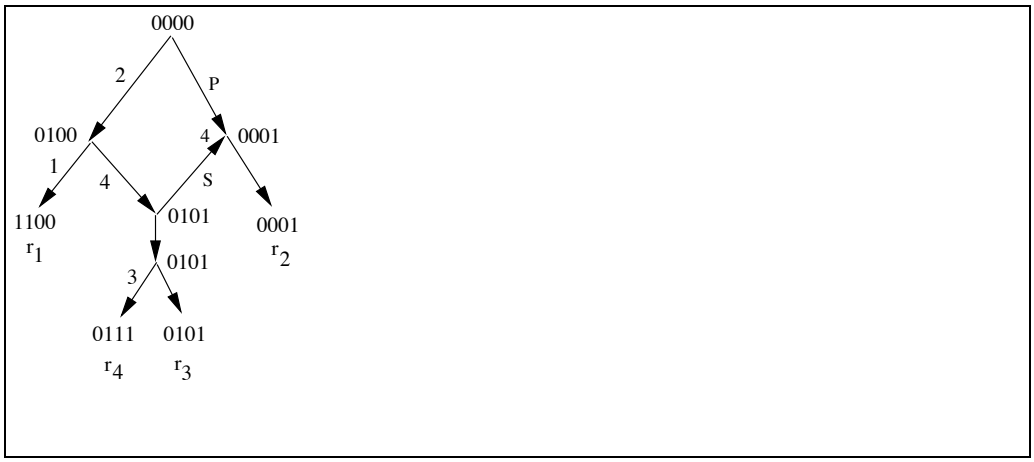


Figure 2.19: The ARG constructed by Algorithm *Clean-Build-with-Recombination* when row r_2 is selected for row $\tilde{r}$ in Rule **Dt**.

2.2.4 Summarizing the Full Algorithm

We reduce M to a single row with no sites, and build an ARG $\mathcal{N}$ for M, using Algorithm *Clean-Build-with-Recombination*, shown in Figure 2.20. As always, $\tilde{M}$ denotes the current matrix created during an execution of the algorithm, changing as the algorithm proceeds. The algorithm is initially called with input M.

It should be clear that Algorithm *Clean-Build-with-Recombination* does build an ARG for M, but we have not yet given any reason to believe that it will be an ARG where the number of recombination nodes is close to $Rmin_0(M)$. To achieve that goal, we will have to add criteria for the way we select the row $\tilde{r}$ in Rule **Dt**, and we will have to be explicit about how recombinations are found to derive $S_{\tilde{r}}$. We do that next.

2.2.4.1 Specifying and Reducing Recombinations

To reduce the number of recombination nodes created in Algorithm *Clean-Build-with-Recombination* we first examine how to derive $S_{\tilde{r}}$ from a set of sequences $\tilde{M}$ using the fewest number of (single-crossover) recombination nodes.

Definition Given a set of sequences $\tilde{M}$, each of length m, and an m-length sequence $S_{\tilde{r}}$ not in $\tilde{M}$, $Rmin(\tilde{M}, S_{\tilde{r}})$ is the *minimum* number of *single-crossover* recombinations needed to create $S_{\tilde{r}}$ from the sequences in $\tilde{M}$ without using any mutations. $Rmin(\tilde{M}, S_{\tilde{r}})$ is not defined if it is not possible to create $S_{\tilde{r}}$ from $\tilde{M}$ without mutations.

$Rmin(\tilde{M}, S_{\tilde{r}})$ will be defined if and only if, for every site c in $S_{\tilde{r}}$, the state of c in $S_{\tilde{r}}$ equals the state of c for some sequence in $\tilde{M}$. Clearly, after any execution

ALGORITHM CLEAN-BUILD-WITH-RECOMBINATION ($\tilde{M}_0$)

{The algorithm will return an ARG $\tilde{\mathcal{N}}$ for $\tilde{M}_0$.}

if $\tilde{M}_0$ contains more than one row or contains some sites **then**

Set $\tilde{M}$ to $\tilde{M}_0$ and run Algorithm *Clean-Build* on $\tilde{M}$
until no further applications of Rules **Dc** or **Dr** are possible.
Let $\tilde{\mathcal{F}}$ denote the forest created by Algorithm *Clean-Build*,
and let $\tilde{M}_1$ denote $\tilde{M}$ at this point.

Apply Rule **Dt** to $\tilde{M}$
{so $\tilde{M}$ now is $\tilde{M}_1 - S_{\tilde{r}}$.}

Apply Rule **Ct** to $\tilde{M}$, creating a DAG G_t that derives the sequence $S_{\tilde{r}}$
by recombinations of sequences in $\tilde{M}$.

(Recursively) Call *Clean-Build-with-Recombination* with input $\tilde{M}$.

endif

Add the DAG G_t to the ARG returned from the above recursive call,
creating an ARG $\tilde{\mathcal{N}}_1$ for $\tilde{M}_1$.

Add the forest $\tilde{\mathcal{F}}$ to $\tilde{\mathcal{N}}_1$, creating the ARG $\tilde{N}$ for $\tilde{M}_0$.

return $\tilde{N}$

Figure 2.20: Algorithm Clean-Build-with-Recombination

of Algorithm *Clean*, no column will contain exactly one entry with value 1, and if it has a column with exactly one entry with value 0, that entry will be in the all-zero ancestral sequence. Since the ancestral sequence cannot be chosen for $S_{\tilde{r}}$, the state of c in $S_{\tilde{r}}$ will always equal the state of c for some sequence in $\tilde{M}$, so in Algorithm *Clean-Build-with-Recombination*, $Rmin(\tilde{M}, S_{\tilde{r}})$ will always be defined. Algorithm *Min-Crossover*, shown in Figure 2.21, efficiently computes $Rmin(\tilde{M}, S_{\tilde{r}})$, or determines that it is not defined, for an arbitrary matrix $\tilde{M}$ and a sequence $S_{\tilde{r}}$. This algorithm is a generalization of Algorithm *Multiple-Crossover-Test* discussed in Section **??**.

Clearly, if $Rmin(\tilde{M}, S_{\tilde{r}})$ is defined, *Algorithm Min-Crossover* will return some value for cr^* and $S_{\tilde{r}}$ can be created from $\tilde{M}$ using cr^* single-crossover recombinations (whose crossover-indices are recorded in $\mathcal{C}$) and no mutations. More completely, we have the following

ALGORITHM MIN-CROSSOVER $(\tilde{M}, S_{\tilde{r}})$

Set $\mathcal{C}$ to the list containing the number 1.
Set c to 1, and cr to 0.

repeat

Over all the sequences in $\tilde{M}$, find the longest substring starting at site c that matches the substring in $S_{\tilde{r}}$ starting at site c.
Let k denote the length of that longest matching substring.

if $(k == 0)$ **then**
report that $Rmin(\tilde{M}, S_{\tilde{r}})$ is undefined, and exit.

else
set cr to $cr + 1$, and c to $c + k$.
Put c at the end of list $\mathcal{C}$.
endif
until $(c > m)$

Set $cr^* = cr$, and add $m + 1$ to the end of $\mathcal{C}$.
return cr^* and $\mathcal{C}$

Figure 2.21: Algorithm *Min-Crossover*, which generalized Algorithm *Multiple-Crossover-Test*.

Theorem 2.2.4 *Algorithm Min-Crossover correctly determines whether* $Rmin(\tilde{M}, S_{\tilde{r}})$ *is defined, and if defined,* $Rmin(\tilde{M}, S_{\tilde{r}})$ *equals* cr^*.

Proof At each iteration, the algorithm attempts to extend the length of the prefix of $S_{\tilde{r}}$ that can be generated from $\tilde{M}$ by single-crossover recombinations. In particular, the length of the prefix starts at zero and increases by exactly k characters in any iteration where $k > 0$. Therefore, unless $k = 0$ in some iteration, the algorithm constructively shows that $Rmin(\tilde{M}, S_{\tilde{r}})$ is defined, and $S_{\tilde{r}}$ can be created from exactly $cr^* + 1$ substrings of sequences in $\tilde{M}$.

Conversely, if $k = 0$ at some iteration, it means that there is a site c where all the sequences in $\tilde{M}$ have a state that is unequal to the state of c in $S_{\tilde{r}}$. Therefore, the algorithm is correct if it reports that $Rmin(\tilde{M}, S_{\tilde{r}})$ is not defined.

To show that $cr^* = Rmin(\tilde{M}, S_{\tilde{r}})$ (when it is defined), assume it does not. Since $S_{\tilde{r}}$ can be created from $cr^* + 1$ substrings in $\tilde{M}$, it must be that $cr^* > Rmin(\tilde{M}, S_{\tilde{r}})$. Let $\mathcal{C}_{min}$ be the list consisting of 1, followed by all the crossover-indices in some scenario that creates $S_{\tilde{r}}$ from $\tilde{M}$ using $Rmin(\tilde{M}, S_{\tilde{r}})$ single-crossover recombinations, followed by $m + 1$. If $cr^* > Rmin(\tilde{M}, S_{\tilde{r}})$, then there must be a *first* index c, such that $\mathcal{C}[c]$, the c'th entry in $\mathcal{C}$, is *strictly* less than $\mathcal{C}_{min}[c]$. Note that by the choice of c, $\mathcal{C}[c-1] \geq \mathcal{C}_{min}[c-1]$. But then, the substring of $S_{\tilde{r}}$ from site $\mathcal{C}_{min}[c-1]$ to site $\mathcal{C}_{min}[c]-1$ must match the substring

of some sequence S in $\tilde{M}$, in that same range of sites. And since $\mathcal{C}_{min}[c-1] \leq \mathcal{C}[c-1]$, the substring of $S_{\tilde{r}}$ from site $\mathcal{C}[c-1]$ to site $\mathcal{C}_{min}[c]-1$ must match S in that range of sites. But that would contradict the fact that the algorithm reported a crossover at site $\mathcal{C}[c] < \mathcal{C}_{min}[c]$. Hence there is no index c where $C[c] < C_{min}[c]$ and therefore $cr^* = Rmin(\tilde{M}, S_{\tilde{r}})$. ■

Given Theorem 2.2.4, the obvious first modification of Algorithm *Clean-Build-with-Recombination* is to use Algorithm *Min-Crossover* to determine how $S_{\tilde{r}}$ should be derived from $\tilde{M}$ in each application of constructive Rule **Ct**. We will assume that this is done in every application of Rule **Ct**. A more effective change is to modify destructive Rule **Dt**. Recall that $\tilde{M}_1$ is the matrix $\tilde{M}$ before the application of Rule **Dt**.

> **Modified Rule Dt:** In Rule **Dt**, choose the row $\tilde{r}$ in $\tilde{M}_1$ that minimizes $Rmin(\tilde{M}_1 - S_{\tilde{r}}, S_{\tilde{r}})$.

The uses of modified Rule **Dt**, and of Algorithm *Min-Crossover* to determine exactly how $S_{\tilde{r}}$ should be derived from $\tilde{M}$, are clearly good heuristics to *locally* (in each application of Rules **Dt, Ct**) reduce the number of recombination nodes in the resulting ARG $\mathcal{N}$ for M. But their use does not guarantee that the resulting ARG will minimize the number of recombination nodes over all possible executions of Algorithm *Clean-Build-with-Recombination*. We consider that goal next.

Definition For a matrix M, let $\mathcal{N}^*(M)$ be the ARG with the minimum number of recombination nodes over all possible executions of Algorithm *Clean-Build-with-Recombination* on input M, and let $RN^*(M)$ denote the number of recombination nodes in $\mathcal{N}^*(M)$.

It is not true that $\mathcal{N}^*(M)$ is necessarily found by applying the modified Rule **Dt** and using Algorithm *Min-Crossover* at every iteration of Algorithm *Clean-Build-with-Recombination*.

It is also not true that $\mathcal{N}^*(M)$ is necessarily a MinARG for M, or an ARG with all-zero ancestral sequence that uses $Rmin_0(M)$ recombination nodes. Still, it is desirable to build $\mathcal{N}^*(M)$ and compute $RN^*(M)$.

2.2.4.2 Computing $RN^*(M)$ and Building $\mathcal{N}^*(M)$

The conceptually simplest way to compute $RN^*(M)$ and to build $\mathcal{N}^*(M)$ is to *branch* on all choices of $\tilde{r}$ in each application of Rule **Dt**, building a search tree of choices[2]. Each path in that search tree defines an ARG for M and the ARG with the fewest recombination nodes defines $RN^*(M)$. The algorithm that conducts such an exhaustive branching over all choices of $\tilde{r}$ is called *Algorithm CBR-branch*.

[2]Note that the original Rule **Dt** is used here, not the modified Rule **Dt**.

It is simple to modify Algorithm *CBR-branch* so that each path of the tree only computes the *number* of recombination nodes that would be used in an ARG constructed along that path, rather than actually constructing the ARG. After the tree is built, the path that computed $RN^*(M)$ can be used to build $\mathcal{N}^*(M)$. Deferring the construction of $\mathcal{N}^*(M)$ until the full search tree is completed yields a small speedup, but the exhaustive branching in Algorithm *CBR-branch* still makes it computationally prohibitive except for small matrices[3]. However, there is considerable redundancy in any execution of Algorithm *CBR-branch* because the same submatrix $\tilde{M}$ of M can be formed many times along many different search paths. We can speed up the branching by avoiding such redundancy, and that will be done in Section 2.2.4.3. There, the addition of branch-and-bound ideas will make the branching approach practical for data of current interest. Moreover, understanding the redundancy in Algorithm *CBR-branch* leads to a *dynamic programming* algorithm to compute $RN^*(M)$, achieving a worst-case running time that is significantly less than the $\theta(n!)$ worst-case time for Algorithm *CBR-branch*.

Dynamic Programming computation of $RN^*(M)$ and $\mathcal{N}^*(M)$

Definition Given a matrix M, and a subset $\mathcal{K}$ of rows of M, let $M_{\mathcal{K}}$ denote the submatrix of M consisting of the rows in $\mathcal{K}$ and *all* of the columns of M.

The idea of the dynamic program is to compute the $RN^*(M_{\mathcal{K}})$, for each subset $\mathcal{K}$ of the rows of M.

In the dynamic program, the values will be computed in order of increasing cardinality of $\mathcal{K}$. $RN^*(M)$ is the extreme case that $\mathcal{K}$ is the entire set of rows of M. In the algorithm, we use the variable $rn^*(M)$ rather than $RN^*(M)$, but later prove that $rn^*(M) = RN^*(M)$. After $RN^*(M)$ is computed, the ARG $\mathcal{N}^*(M)$ can be constructed during a standard dynamic programming traceback. That traceback will specify a series of applications of Rules **Dc, Dr** and **Dt**, and $\mathcal{N}^*(M)$ can therefore be build by applying the corresponding Rules **Cc, Cr** and **Ct**, as discussed earlier in this chapter.

The key idea in the dynamic programming recurrence is to mimic the action of Algorithm *CBR-branch* on matrix $M_{\mathcal{K}}$, but avoid redundancy by enumerating each required matrix $\tilde{M}$ and computing each $RN^*(\tilde{M})$ only once. The non-trivial part of the recurrence is that it stores $rn^*(M_{(\tilde{\mathcal{K}}-\tilde{r})})$, where $M_{(\tilde{\mathcal{K}}-\tilde{r})}$ is the submatrix of M consisting of the rows in set $\tilde{\mathcal{K}} - \tilde{r}$, but *all* of the columns of M. In that way, it enumerates 2^n submatrices of M, each specified by a choice of rows, rather than enumerating $2^n 2^m$ submatrices specified by choices of both rows and columns. The correctness of this idea essentially follows from Lemma 2.2.1 (on page 36), and is formally proved next.

[3]The only established upper bound for this version of Algorithm *CBR-branch* is $\theta(n!)$.

ALGORITHM DP-$RN^*(M)$

if (M has less than three rows) **then**
  set $rn^*(M)$ to 0
  return $rn^*(M)$
endif

for ($k = 2, \ldots, n$) **do**
  {where n is the number of rows in M}

  for (each subset $\mathcal{K}$ of k rows of M) **do**
    Form the submatrix $M_{\mathcal{K}}$ of M, and run Algorithm *Clean* on $M_{\mathcal{K}}$.
    Let $\tilde{M}$ denote the resulting matrix, and let $\tilde{\mathcal{K}}$ denote
    the set of rows of $\tilde{M}$.

    Set $rn^*(M_{\mathcal{K}}) = \min_{S_{\tilde{r}} \in \tilde{M}}[Rmin(\tilde{M} - S_{\tilde{r}}, S_{\tilde{r}}) + rn^*(M_{(\tilde{\mathcal{K}}-\tilde{r})})]$
  endfor
endfor

return $rn^*(M)$

Theorem 2.2.5 *The value of $rn^*(M)$ computed by Algorithm DP-RN^* on input M is $RN^*(M)$.*

Proof The proof is by induction on $|\mathcal{K}|$. When $|\mathcal{K}| < 3$, matrix $M_{\mathcal{K}}$ cannot have any incompatible pair of sites, so it can be derived on a perfect-phylogeny with all-zero ancestral sequence. Therefore, by Theorem 2.2.2, Algorithm *Clean* reduces $M_{\mathcal{K}}$ to a matrix with one row and no sites. So $RN^*(M_{\mathcal{K}}) = rn^*(M_{\mathcal{K}}) = 0$, and the theorem holds when $|\mathcal{K}| < 3$. Now suppose the theorem holds up to some $k \geq 3$, and that $|\mathcal{K}| = k + 1$.

Consider the action of Algorithm *CBR-branch* on $M_{\mathcal{K}}$. It would first run Algorithm *Clean* on $M_{\mathcal{K}}$, resulting in the same matrix $\tilde{M}$ defined in Algorithm *DP-RN^**. At that point, it would branch on all choices of $S_{\tilde{r}} \in \tilde{M}$. Each path from that branching point would compute $Rmin(\tilde{M} - S_{\tilde{r}}, S_{\tilde{r}}) + RN^*(\tilde{M} - S_{\tilde{r}})$ for one of those $S_{\tilde{r}}$. So clearly,

$$RN^*(M_{\mathcal{K}}) = \min_{S_{\tilde{r}} \in \tilde{M}} [Rmin(\tilde{M} - S_{\tilde{r}}, S_{\tilde{r}}) + RN^*(\tilde{M} - S_{\tilde{r}})].$$

The right-hand side of that recurrence differs from the right-hand side of the recurrence given in Algorithm DP-RN^* in that the second term in the recurrence is $RN^*(\tilde{M} - S_{\tilde{r}})$ rather than $rn^*(M_{(\tilde{\mathcal{K}}-\tilde{r})})$. So if we can prove that those two terms are equal, then the recurrence for rn^* will be the same as for RN^* and the theorem will be proved. Now $rn^*(M_{(\tilde{\mathcal{K}}-\tilde{r})}) = RN^*(M_{(\tilde{\mathcal{K}}-\tilde{r})})$, by the induction hypothesis, so we want to prove that $RN^*(M_{(\tilde{\mathcal{K}}-\tilde{r})}) = RN^*(\tilde{M} - S_{\tilde{r}})$.

Note that $\tilde{\mathcal{K}}$ is the set of rows in $\tilde{M}$, so $\tilde{M} - S_{\tilde{r}}$ and $M_{(\tilde{\mathcal{K}}-\tilde{r})}$ have the same set of rows but a different set of columns. Recall that the computation of $RN^*(M_{(\tilde{\mathcal{K}}-\tilde{r})})$ begins with the application of Algorithm *Clean* to $M_{(\tilde{\mathcal{K}}-\tilde{r})}$. Consider a column c in $M_{\mathcal{K}}$ that is not in $\tilde{M}$. In reducing $M_{\mathcal{K}}$ to $\tilde{M}$, column c was removed at a point where it had at most one entry with value 1. Since the rows in $\tilde{\mathcal{K}}-\tilde{r}$ are a subset of the rows at that point, column c in $M_{(\tilde{\mathcal{K}}-\tilde{r})}$ will also have at most one entry with value 1. If we run Algorithm *Clean* on $M_{(\tilde{\mathcal{K}}-\tilde{r})}$ by first removing all the columns in $M_{\mathcal{K}}$ not in $\tilde{M}$ (which is permitted by Lemma 2.2.1), the result after removing those columns will be $\tilde{M}-S_{\tilde{r}}$. Therefore, the full execution of Algorithm *Clean* on $M_{(\tilde{\mathcal{K}}-\tilde{r})}$ will be the same as the result of running Algorithm *Clean* on $\tilde{M} - S_{\tilde{r}}$. Further, the destructive operations in Algorithm *Clean* induce constructive application of Rules **Cc** and **Cr** that build a forest containing *zero* recombination nodes. Therefore, $rn^*(M_{\tilde{\mathcal{K}}-\tilde{r}}) = RN^*(M_{(\tilde{\mathcal{K}}-\tilde{r})}) = RN^*(\tilde{M} - S_{\tilde{r}})$, and we conclude that the value $rn^*(M)$ returned by Algorithm *DP-RN** is $RN^*(M)$. ■

Time analysis There are 2^n subsets of rows of M, and for each one we examine at most n individual sequences $S_{\tilde{r}}$, running Algorithm *Min-Crossover* on each. With the appropriate data structures and string algorithm, each execution of Algorithm *Min-Crossover* takes $O(nm)$ time, so

Theorem 2.2.6 *Algorithm DP-RN* can be implemented to run in* $O(n^2m2^n)$ *time.*

We also need to keep a table to hold $rn^*(M_{\mathcal{K}})$ for each subset of rows $\mathcal{K}$.

2.2.4.3 Major Speedups of Algorithm *CBR-branch*

In addition to the dynamic-programming approach to computing $RN^*(M)$ (and then $\mathcal{N}^*(M)$), there are three ways to speed up Algorithm *CBR-branch*.

The first speedup is through the standard way that *memoization* converts a top-down, recursive branching algorithm into an algorithm whose worst-case running time is of the same order as a bottom-up dynamic programming solution [6]. In that approach, the search tree formed by Algorithm *CBR-branch* would be expanded *depth-first* and a table would be built that stores $RN^*(\tilde{M})$ for every submatrix of M formed during the search. When the depth-first search backs up to a node in the search tree where a recursive call to compute $RN^*(\tilde{M})$ was made, the value of $RN^*(\tilde{M})$ will be known and can be put in the table. Then, when another search path requires $RN^*(\tilde{M})$ it can retrieve the value from the table, rather than expanding the search tree to (re)compute it. This standard memoizing idea has the consequence that the number of internal nodes in the search tree is bounded by the number of *distinct* submatrices of M. Further, by

Lemma 2.2.1, the number of submatrices where any branching occurs is bounded by 2^n, and so we have

Theorem 2.2.7 *Algorithm CBR-branch can be implemented using memoization to run in $O(n^2 m 2^n)$ worst-case time.*

The second speedup occurs if the search-tree in Algorithm *CBR-branch* is *not* expanded in a strictly depth-first manner. In that case there can be two or more nodes in the expanded tree that require the computation of $RN^*(\tilde{M})$, which has not yet been computed. Because the order that rows are removed by Rule **Dt** differs along different search paths that create submatrix $\tilde{M}$, the total number of recombination nodes (in the implied ARGs that could be created) specified along those paths can differ. Clearly, among all the paths in the search tree that create the same $\tilde{M}$, only the one with the smallest total number of specified recombination nodes should be expanded. The other paths can be terminated.

The third speedup is through application of the classic idea of using lower bounds to cut off paths in the search tree. Suppose that some of the paths in the tree have been completed, and let w be the minimum number of recombination nodes in the ARGs for M specified along those paths. Consider an incomplete path in the search tree ending at a node v, creating submatrix $\tilde{M}$. Let $w(\tilde{M})$ be the total number of recombination nodes specified by that path, and let $L(\tilde{M})$ be a *lower-bound* on $Rmin_0(\tilde{M})$. If $w \leq w(\tilde{M}) + L(\tilde{M})$ then there is no need to expand the search tree at node v.

2.2.5 Program *SHRUB*

Algorithm *CBR-branch* with the second and third speedups discussed above has been implemented in a program called *SHRUB*, which stands for "simulated history recombination upper bound" [27]. *SHRUB* does not necessarily assume that the ancestral sequence is the all-zero sequence, but we will continue to use the notation $\mathcal{N}^*(M)$ for best possible ARG produced by *SHRUB*, and use $RN^*(M)$ for the number of recombination nodes in $\mathcal{N}^*(M)$.

With no additional modifications, when given an input matrix M, *SHRUB* will compute the *full upper bound* $RN^*(M)$ and build the ARG $\mathcal{N}^*(M)$. The lower bound used in the branch-and-bound is either the *HK* bound or the *haplotype* bound discussed in Chapter 1. $RN^*(M)$ is called an "upper bound" because $Rmin(M) \leq RN^*(M)$. This inequality is due to the fact that $\mathcal{N}^*(M)$ is an ARG with all-zero ancestral sequence that derives M and uses exactly $RN^*(M)$ recombination nodes.

The user can also select a number k, and specify that the search tree should branch at most k ways. In that case, when *SHRUB* is at a branch node of the search tree, it *randomly* selects k rows (or all rows if there are k or fewer rows) in the current $\tilde{M}$ and individually chooses each of those rows as $\tilde{r}$. Since the

rows are selected randomly, different executions of the computation can give different results, and it may be advantageous to repeat the computation several times to select the best result. In experiments reported in [27] the number of recombination nodes in the ARGs found with k between 3 and 5 was very close to $RN^*(M)$.

SHRUB can also be used to compute a *fast upper bound* which is obtained by an implementation of Algorithm *Clean-Build-with-Recombination* using the modified Rule **Dt**, i.e., always choosing the row $\tilde{r}$ to minimize $Rmin(\tilde{M}_1 - S_{\tilde{r}}, S_{\tilde{r}})$. The computation time for this approach is polynomially-bounded since there is no branching when the Modified Rule **Dt** is applied. The resulting number of recombination nodes may not be close to $RN^*(M)$, but it can be used as a first value w in the branch-and-bound version of Algorithm *CBR-branch*.

The program *SHRUB* not only computes $RN^*(M)$, but outputs the specifications for $\mathcal{N}^*(M)$ in a format that can then be processed by a graphical display program and drawn on the plane. The program and the ARG drawing that it specified was used to help identify a single gene that greatly influences the body size of dogs [28] (see the supplemental material to see the use of *SHRUB*).

Experimental studies of the accuracy of *SHRUB* are reported in [27] for both real SNP data and for simulated data. In each dataset, $RN^*(M)$ is compared to the lower bound on $Rmin(M)$ computed by program *HapBound*. Table 1.1 on page 31 shows both of the numbers, for real population data. The results there show that the observed upper and lower bounds are close, often equal, in which case both *HapBound* and *SHRUB* have computed $Rmin(M)$ exactly. Simulated data was produced by the coalescent simulation program *MS* [12] which is commonly used to generate test data for questions about ARGs. The critical parameters in that simulation are the sample size (number of taxa in the terminology of this book), the mutation rate θ (which determines the number of sites), and the recombination rate ρ. The rates were chosen to reflect a range of realistic biological situations. The results show that when θ and ρ are modest, and the sample size is 25, the upper and lower bounds agreed more than 95% of the time; as the sample size increases to 100, the agreement falls slowly to about 85%. In the cases where the upper and lower bounds were not equal, the lower bound was on average about 80% of the value of the upper bound. More details on these experiments can be found in [27].

SHRUB also efficiently, almost instantly, builds an ARG with 7 recombination nodes for Kreitman's classic SNP data (see Figure 1.5). In Section **??** we discussed that program *HapBound* produces a lower bound of 7 for this data, so we can now conclude that $Rmin(M)$ for Kreitman's data is in fact 7. This will be additionally confirmed in Section **??** when we discuss the program *Beagle*.

Bibliography

[1] V. Bafna and V. Bansal. Improved recombination lower bounds for haplotype data. Proceedings of RECOMB 2005.

[2] V. Bafna and V. Bansal. The number of recombination events in a sample history: conflict graph and lower bounds. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 1:78–90, 2004.

[3] V. Bafna and V. Bansal. Inference about recombination from haplotype data: Lower bounds and recombination hotspots. *J. of Comp. Biology*, 13:501–521, 2006.

[4] D.G. Brown and I.M. Harrower. Integer Programming Approaches to Haplotype Inference by Pure Parsimony. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 3(2):141–154, 2006.

[5] A. Clark, K. Weiss, and D. Nickerson et. al. Haplotype structure and population genetic inferences from nucleotide-sequence variation in human lipoprotein lipase. *Am. J. Human Genetics*, 63:595–612, 1998.

[6] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT press and McGraw Hill, Cambridge, MA., 1992.

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

[8] M. Garey and D. Johnson. *Computers and intractability*. Freeman, San Francisco, 1979.

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

[10] D. Gusfield, D. Hickerson, and S. Eddhu. An efficiently-computed lower bound on the number of recombinations in phylogenetic networks: Theory

and empirical study. *Discrete Applied Math, Special issue on Computational Biology, 2007*, 155:806–830, 2007.

[11] J. Hein, M. Schierup, and C. Wiuf. *Gene Genealogies, Variation and Evolution: A primer in coalescent theory*. Oxford University Press, UK, 2005.

[12] R. Hudson. Generating samples under the Wright-Fisher neutral model of genetic variation. *Bioinformatics*, 18(2):337–338, 2002.

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

[14] J. D. Kececioglu and D. Gusfield. Reconstructing a history of recombinations from a set of sequences. *Discrete Applied Math.*, 88:239–260, 1998.

[15] M. Kreitman. Nucleotide polymorphism at the alcohol dehydrogenase locus of drosophila melanogaster. *Nature*, 304:412–417, 1983.

[16] R. Lyngsø, Y.S. Song, and J. Hein. Minimum recombination histories by branch and bound. In *Proceedings of Workshop on Algorithm of Bioinformatics (WABI) 2005*, volume 3692, pages 239–250, Berlin, Germany, 2005. Springer-Verlag LNCS.

[17] M. Minichiello and R. Durbin. Mapping trait loci using inferred ancestral recombination graphs. *Am. J. Hum. Genet.*, 79:910–922, 2006.

[18] S. Myers. *The detection of recombination events using DNA sequence data*. PhD thesis, University of Oxford, Oxford England, Department of Statistics, 2003.

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

[20] D. Nickerson, S. Taylor, K. Weiss, and A. Clark et. al. DNA sequence diversity in a 9.7-kb region of the human lipoprotein lipase gene. *Nature Genetics*, 19:233–240, 1998.

[21] University of Oxford S. Myers, 2004. personal communication.

[22] Y. S. Song, Z. Ding, D. Gusfield, C. H. Langley, and Y. Wu. Algorithms to distinguish the role of gene-conversion from single-crossover recombination in the derivation of SNP sequences in populations. In *Proceedings of the 10'th Annual International Conference on Research in Computational Molecular Biology (RECOMB)*, volume 3909, pages 231–245. Springer LNBI, 2006.

[23] Y. S. Song, Z. Ding, D. Gusfield, C. H. Langley, and Y. Wu. Algorithms to distinguish the role of gene-conversion from single-crossover recombination in the derivation of SNP sequences in populations. *J. of Comp. Biology*, 14:1273–1286, 2007.

[24] Y. S. Song and J. Hein. Parsimonious reconstruction of sequence evolution and haplotype blocks: Finding the minimum number of recombination events. In *Proc. of 2003 Workshop on Algorithms in Bioinformatics*, Berlin, Germany, 2003. Springer-Verlag LNCS.

[25] Y. S. Song and J. Hein. On the minimum number of recombination events in the evolutionary history of DNA sequences. *Journal of Mathematical Biology*, 48:160–186, 2004.

[26] Y. S. Song and J. Hein. Constructing minimal ancestral recombination graphs. *J. of Comp. Biology*, 12:159–178, 2005.

[27] Y. S. Song, Y. Wu, and D. Gusfield. Efficient computation of close lower and upper bounds on the minimum number of needed recombinations in the evolution of biological sequences. *Bioinformatics*, 21:i413–i422, 2005. Bioinformatics Suppl. 1, Proceedings of ISMB 2005.

[28] N. Sutter and C. D. Bustemante et al. A single IGF1 allele is a major determinant of small size in dogs. *Science*, 213:112–115, 2007.

[29] J. Wakeley. *Coalescent Theory*. Roberts and Co., Greenwood Village, CO., 2009.

[30] J. D. Wall. A comparison of estimators of the population recombination rate. *Mol. Biol. Evol.*, 17:156–163, 2000.

[31] Y. Wu and D. Gusfield. Efficient computation of minimum recombination over genotypes (not haplotypes). In *Proceedings of Life Science Society Computational Systems Bioinformatics (CSB) 2006*, pages 145–156, 2006.

[32] Y. Wu and D. Gusfield. Efficient computation of minimum recombination with genotypes (not haplotypes). *Journal of Bioinformatics and Computational Biology*, pages 181–200, 2007.

[33] Y. Wu and D. Gusfield. A new recombination lower bound and the minimum perfect phylogenetic forest problem. In *Proceedings of 13'th Annual International Conference on Combinatorics and Computing*, pages 16–26. Springer-Verlag LNCS Vol. 4598, 2007.