Reconstructing a history of recombinations from a set of sequences*

John Kececioglu[†] Dan Gusfield[‡]

October 30, 1996; revised March 3, 1998

Abstract One of the classic problems in computational biology is the reconstruction of evolutionary history. A recent trend in the area is to increase the explanatory power of the models that are considered by incorporating higher-order evolutionary events that more accurately reflect the mechanisms of mutation at the level of the chromosome.

We take a step in this direction by considering the problem of reconstructing an evolutionary history for a set of genetic sequences that have evolved by *recombination*. Recombination is a non-tree-like event that produces a child sequence by crossing two parent sequences. We present polynomial-time algorithms for reconstructing a parsimonious history of such events for several models of recombination when all sequences, including those of ancestors, are present in the input. We also show that these models appear to be near the limit of what can be solved in polynomial time, in that several natural generalizations are NP-complete.

Keywords Computational biology, evolutionary trees, edit distance, recombination, directed hypergraphs, bottleneck optimality

1 Introduction

One of the classic problems of computational biology that is enjoying renewed interest is the reconstruction of evolutionary histories.

In the formulation most often considered, the input is a set of strings, usually representing the genetic sequences of a collection of organisms, and one seeks a tree over the set that explains the observed differences and optimizes an objective function. Such a tree describes how the collection of organisms could have evolved from a common ancestor by a process of

---

*An earlier version of this paper was presented at the 5th ACM-SIAM Symposium on Discrete Algorithms [23].

†Department of Computer Science, University of Georgia, Athens, GA 30602. Electronic mail: `kece@cs.uga.edu`. This research was performed at the University of California, Davis, Department of Computer Science, and supported by a postdoctoral fellowship from the National Science Foundation Program in Mathematics and Molecular Biology under Grant DMS-8720208, and a Department of Energy Human Genome Distinguished Postdoctoral Fellowship.

‡Department of Computer Science, University of California, Davis, CA 95616. Electronic mail: `gusfield@cs.ucdavis.edu`. Research supported in part by Department of Energy Grant DE-FG03-90ER60999, and National Science Foundation Grant CCR-9103937.

segregation, and a number of efficient algorithms for finding such trees have been designed in recent years [1, 8, 12, 20, 45].

However, the two assumptions inherent in a tree model—that the organisms descended from a common ancestor and that descendants arose by segregation—are not always well met by the data. In fact, several molecular mechanisms lead not to the segregation of genetic material, but to its merging. When merging of genetic material has taken place, the pattern of evolution need no longer be tree-like, and it has been observed in the literature that such evolution confounds tree reconstruction algorithms. Wareham [43] is an excellent source of references, among them [27, 32, 34, 38].

The fundamental mechanism by which genetic material is merged is *recombination* [39]. Informally, recombination produces a new sequence by crossing two parent sequences. The new sequence is formed in effect by starting at the left end of one parent sequence, copying a substring, crossing over to some location in the other parent sequence, again copying a substring, crossing back to the first parent, and so on, always moving to the right within a sequence and stopping only when at the right end of one of the parents. This mechanism of crossover leads to many of the more complex events considered by sequence comparison algorithms, including block insertions and deletions [33, 10] and tandem repeats [21, 26]. Recombination, in organisms with pairs of chromosomes, often occurs during *meiosis*, the stage at which a reproductive cell receives a representative from each chromosome pair. Crossover between the pair usually takes place in regions of high *homology*, or genetic similarity. However it does not occur only in organisms that have pairs of chromosomes, or that reproduce sexually: recombination also occurs in bacteria and viruses. As Watson et al. [46, p. 313] write,

> All DNA is recombinant DNA .... [The] natural processes of recombination and mutation ... have acted throughout evolution .... Genetic exchange works constantly to blend and rearrange chromosomes, most obviously during meiosis, when homologous chromosomes cross over in every generation.

In this paper, we consider the problem of reconstructing an evolutionary history when recombination has occurred. This follows a trend in computational biology to design algorithms that take into account macro-mutations at the chromosome level [25, 5, 6, 24, 13, 14], in contrast to micro-mutations at the gene level, now that data for whole genomes is becoming available.

We introduce two new problems. The first, *Recombination Cost*, is a generalization of sequence alignment where the operations of insertion, deletion, and substitution of characters are extended to include crossover. The problem is to convert two strings into a third by a minimum-cost series of such operations.

The second problem, *Bottleneck Recombination History*, is a generalization of evolutionary tree reconstruction where segregation of descendants is extended to include merging by recombination, and the notion of a single common ancestor of the organisms is extended to a pair of ancestors. The problem is to find a series of recombinations, called a *spanning history*, that generate precisely the sequences in a given set, starting from two of the sequences, called a *protopair*. In the spanning history, the parents in every recombination must be produced before the child, and the cost of every recombination must be within a given bound. The protopair in effect represents an Adam and an Eve who by mating can give rise to the recorded population in such a way that no child is too distant from its parents. The bottleneck constraint on recombination costs reflects the fact that in living systems the cost of a recombination cannot be too large if the offspring is to be viable.

It is worth pointing out that an analysis of recombination is more in the mode of population genetics, where data is gathered from a large number of individuals belonging to a group to reveal the fine structure of their similarities and differences, rather than traditional phylogenetics, where a few representative individuals are studied to reveal the slowly changing features common to an entire class of organisms. Even so, a major assumption in a spanning history formulation is that all individuals, including ancestors, are contained in, or at least well-represented by, the input. This is counter to the common tenet in phylogenetics that only existing organisms can be observed, not ancestral organisms. Interestingly, with new techniques for recovering ancestral DNA, for example by extracting DNA from ancient organisms encased in amber, the restriction on ancestral sequences may be too severe; as a recent article on ancient DNA reports [41], "Genetic information that had seemed lost forever turns out to linger in the remains of long-dead plants and animals. Evolutionary change can at last be observed directly." Furthermore, it is conceivable that one day we may be able to obtain in the laboratory a snapshot of the genetic material of an evolving population of organisms; for such data, the algorithms of this paper would provide a useful analysis. Finally, from the viewpoint of algorithm design, it is worth understanding the underlying spanning problem before tackling a more complex Steiner version.

For Bottleneck Recombination History, we design algorithms for several increasingly general models of recombination. In the simplest model, a recombination consists of one crossover on strings of equal length, and no insertions, deletions, or substitutions are allowed. For this case we give an algorithm that runs in $O(n + k^3)$ time for k sequences of total length n.

In the most general model, the number of crossovers is arbitrary, and insertion, deletion, and substitution of characters are allowed. For this model we give an algorithm that takes $O(n^3 + k^2 m)$ time, where $m = O(k^3)$ is the number of triples of sequences that have a recombination cost within the given bound. With an additional term of $O(k^3 \log k)$ in the running time, one can find a protopair whose bottleneck cost is minimum.

Bottleneck Recombination History appears to be near the limit of reconstruction problems with recombinations that can be solved in polynomial time, in that generalizing it in either of two natural directions leads to NP-completeness. If instead of a protopair we ask whether there is a *protoset* whose cardinality is within a given bound, the problem becomes NP-complete. Similarly if we relax the condition that every recombination has bounded cost and ask whether there is a protopair and a history with *total cost* within a given bound, the problem is also NP-complete.

In the remainder of the introduction we formally define the two basic problems and briefly survey related work. Section 2 examines the case with one crossover per recombination, and Section 3 treats multiple crossover. Section 4 considers the two generalizations. We conclude with some problems for further research.

1.1 Problem statement

Formally, given three strings A, B, and C over an alphabet Σ, we say C can be formed from A and B by pure recombination if A can be expressed as the concatenation of $c+1$ strings $v_1, \ldots, v_{c+1}$ where $c \geq 0$,

$$A = v_1 v_2 \cdots v_{c+1},$$

and B can be expressed as the concatenation of $c+1$ strings $w_1, \ldots, w_{c+1}$,

$$B \;=\; w_1 w_2 \,\cdots\, w_{c+1},$$

such that C is the concatenation of the $c+1$ strings shown below, which alternates strictly between the strings forming A and B, beginning with v_1 and ending with v_{c+1} or w_{c+1}:

$$C \;=\; v_1 w_2 v_3 w_4 \,\cdots\, v_{2i-1} w_{2i} \,\cdots\,.$$

If $c+1$ is odd, the last string in the concatenation forming C is v_{c+1}, while if $c+1$ is even, the last string forming C is w_{c+1}. In this definition, we require that the substrings forming C, namely v_1, v_3, v_5, $\ldots$ and w_2, w_4, w_6, $\ldots$, are all nonempty, though the other substrings of A and B can be empty. The decomposition of A, B, and C into strings $v_1, \ldots, v_{c+1}$ and $w_1, \ldots, w_{c+1}$ is called a *pure recombination.*

If we view C as being formed by copying symbols from A and B, starting with the first symbol of A and switching from A to B when following string v_{2i-1} with w_{2i} and from B to A when following w_{2i} with v_{2i+1}, it is natural to call each transition between A and B a crossover. Quantity c in the above, which counts the number of times C switches between A and B, is called the *number of crossovers* of the recombination.

In recombination with *equal-length crossover*, $|v_i| = |w_i|$ for all $1 \le i \le c+1$. In other words, the locations at which A and B are cut into the $c+1$ substrings, which are the points in A and B at which crossovers occur, are the same in A and B. In recombination with *unequal-length crossover*, it is possible to have $|v_i| \ne |w_i|$ for some i, so that the locations of the crossover points may differ in A and B.

We define *recombination with point mutation*, which differs from pure recombination in that errors are allowed in C, as follows. As in pure recombination, A is decomposed into $v_1 v_2 \,\cdots\, v_{c+1}$ and B is decomposed into $w_1 w_2 \,\cdots\, w_{c+1}$, but now C is the concatenation

$$C \;=\; \widetilde{v}_1 \widetilde{w}_2 \widetilde{v}_3 \widetilde{w}_4 \,\cdots\, \widetilde{v}_{2i-1} \widetilde{w}_{2i} \,\cdots\,,$$

where strings $\widetilde{v}_{2i-1}$ and $\widetilde{w}_{2i}$ differ from the corresponding v_{2i-1} and w_{2i} of A and B by the insertion, deletion, and substitution of characters. We call these edit operations *point mutations.*[1]

The cost of a recombination with point mutation is measured as follows. As is common when measuring the cost of transforming one string into another by an extended set of edit operations [42, 28], we assume that along with A, B, and C we are given a non-negative real number χ and a function δ from pairs of characters in $\Sigma \cup \{\epsilon\}$ to the non-negative reals, where $\epsilon \notin \Sigma$. The constant χ is the cost of a crossover, while for $a, b \in \Sigma$, $\delta(a, b)$ is the cost of substituting character b for character a, $\delta(\epsilon, a)$ is the cost of inserting character a, and $\delta(b, \epsilon)$ is the cost of deleting character b. For a fixed function δ, the *edit distance* between two strings x and y over Σ is the minimum total cost under δ of a series of point mutations that edit x into y, and is denoted by $d(x, y)$. The cost then of the recombination with point mutation given above is

$$d(v_1, \widetilde{v}_1) \;+\; d(w_2, \widetilde{w}_2) \;+\; \cdots \;+\; d(v_{2i-1}, \widetilde{v}_{2i-1}) \;+\; d(w_{2i}, \widetilde{w}_{2i}) \;+\; \cdots \;+\; c\chi,$$

[1] It is worth pointing out that point mutations occur by a process separate from recombination, and in reality may have occurred before or after the recombination event that produced child C from parents A and B.

where c is the number of crossovers. The *recombination cost* of C with respect to A and B is the minimum cost of a recombination with point mutation that produces C from A and B. Note that δ and χ together specify the recombination cost function.

Definition 1 The *Recombination Cost Problem* is the following. Given three strings A, B, and C together with a recombination cost function specified by a point mutation cost function δ and a crossover cost χ, find a recombination of minimum cost that produces C from A and B. □

A *spanning history* for a set $\mathcal{S}$ of sequences is a series of recombinations that, starting from a pair $A, B \in \mathcal{S}$, produces exactly the sequences in $\mathcal{S}$ in such a way that the parents of a recombination are always produced before the child.[2] Sequences A and B are called a *protopair* for $\mathcal{S}$. Such a history is analogous to a directed spanning tree (sometimes called a spanning arborescence in graph theory), in that all sequences except the protopair are produced by exactly one recombination, and the descendants relation does not contain cycles.

Definition 2 The *Bottleneck Recombination History Problem* is the following. Given a cost bound d and a set $\mathcal{S}$ of sequences, together with a recombination cost function specified by δ and χ, find a protopair $A, B \in \mathcal{S}$ and an associated spanning history such that every recombination in the history has cost at most d. □

Note that this models pure recombination when the cost of point mutation exceeds cost bound d.

Table 1 summarizes the time and space of our algorithms for Bottleneck Recombination History. The algorithms consist of a preprocessing phase, which analyzes the sequence data, followed by a search phase, which tests candidate protopairs. For multiple-crossover recombination, the search proceeds over a directed hypergraph [3] containing sequence triples. (Each triple represents a directed hyperedge. The first two elements of a triple are the parents in a recombination, and are unordered; the third element of a triple is the child of the recombination. A hyperedge is considered to be directed from the two parents to the child.) Throughout the paper, n denotes the total length of the sequences, k denotes the number of sequences, and m denotes the number of triples in the directed hypergraph constructed during preprocessing for multiple-crossover recombination. In the worst case, $m = O(k^3)$, but m may be much smaller.

1.2 Related work

Prior work on recombination has largely focused on statistical tests for population genetics studies [18, 19, 22, 36, 40]. Hein [15, 16] appears to be the first to consider the problem from the view of designing an algorithm to reconstruct a history. He presents a heuristic for a Steiner version of the problem in which the sequence data is aligned (in other words point mutation is restricted to substitution), each recombination involves one equal-length crossover, and at any position across the sequences at most one crossover has occurred in the history.

[2] Note that while it is convenient to refer to a *series* of recombinations, the time-ordering constraint between the parents and the child in a recombination actually imposes only a partial order on the recombinations in a spanning history, and not a total order.

Table 1 Worst-case time and space for Bottleneck Recombination History on k sequences of total length n. For multiple-crossover recombination, the number of hyperedges in the hypergraph constructed during preprocessing is m, the maximum length of a sequence is l, and the maximum number of crossovers allowed per recombination in multiple-crossover recombination without point mutation is c.

		Time		Space	
		Preprocessing	Search	Preprocessing	Search
Single crossover	Equal crossover without point mutation	$O(n+k^2)$	$O(k^3)$	$O(n+k^2)$	$O(k)$
	Unequal crossover without point mutation	$O(n+k^2)$	$O(k^4)$	$O(n+k^2)$	$O(k^2)$
	Unequal crossover with point mutation	$O(n^2)$	$O(k^3 n)$	$O(kn)$	$O(n)$
Multiple crossover	Unequal crossover without point mutation	$O(ckn^2)$	$O(k^2 m)$	$O(cl+m)$	$O(m)$
	Unequal crossover with point mutation	$O(n^3)$	$O(k^2 m)$	$O(l^2+m)$	$O(m)$

There is an operator in parallel programming theory that is similar to recombination. If a pair of strings is viewed as representing two parallel processes, where each character represents a task to be performed, a *shuffle* of the two strings is an arbitrary interleaving or serialization of the two processes, much like shuffling two decks of cards. Mansfield [30, 31] gives an $O(n^k)$ time algorithm for determining whether a particular string of length kn is the shuffle of k given strings of length n, and proves that the problem is in general NP-complete. Warmuth and Haussler [44] show, among other results, that the closure of a regular language with respect to the shuffle operator can be recognized in polynomial time.

2 Single crossover

In this section we consider the history problem with one crossover per recombination. We first consider pure recombination, and then extend our solution to include point mutation.

2.1 Pure recombination

Our algorithm for finding a protopair has two phases. The preprocessing phase analyses the sequences to speed up the search for a protopair, and the search phase tests candidate pairs.

For the preprocessing we compute two quantities for every pair A, B of sequences: $\mathrm{PrefLen}(A, B)$, which is the length of the longest common prefix of A and B, and $\mathrm{SufLen}(A, B)$, which is the length of the longest common suffix of A and B.

In the search phase we test a candidate protopair as follows. Recall that A, B is a protopair for $\mathcal{S}$ if and only if every member of $\mathcal{S}$ can be reached from A and B by a series of recombinations within $\mathcal{S}$. The set of sequences reachable from A and B can be determined by breadth-first search. During the search, we maintain two quantities for each unreached sequence C: $\mathrm{MaxPrefLen}(C)$, which is the maximum of $\mathrm{PrefLen}(A, C)$ over every sequence A

that has been reached; and MaxSufLen(C), which is the maximum of SufLen(B, C) over every B that has been reached. An unreached sequence C is then reachable by one single-crossover recombination involving already reached sequences if and only if

$$\text{MaxPrefLen}(C) \;+\; \text{MaxSufLen}(C) \;\geq\; |C|.$$

In this inequality it is possible for the same reached sequence to contribute both the suffix and the prefix of C. This is legitimate, and models the unequal crossing of a sequence with a copy of itself.[3]

We can compute PrefLen and SufLen for all pairs of sequences in $O(n + k^2)$ time, by building a suffix tree for the sequences and looking up nearest common ancestors in constant time per query [2, 37]. Finding the next reachable sequence, by testing the above inequality on $O(k)$ sequences, takes $O(k)$ time. Adding this sequence to the reached set and updating MaxPrefLen and MaxSufLen also takes $O(k)$ time. A search from a candidate protopair reaches $O(k)$ sequences, so a pair can be tested in $O(k^2)$ time. The only question is how many pairs we have to test.

Lemma 1 *A set of k sequences can have at most $k/2$ protopairs under equal-length crossover.*

Proof Let $\mathcal{S}$ be the set of sequences, and $\{A, B\}$ be a protopair for $\mathcal{S}$. With equal-length crossover, the ith character of a sequence C of $\mathcal{S}$ must be either the ith character of A or the ith character of B. Any position then at which more than one character appears can contain at most two different characters, and a protopair, in order to generate all the sequences, must contain both characters at such a position. Hence, if we ignore all positions at which only one character appears, A and B are *complementary sequences.* Since the sequence B that is complementary to a given A is unique, protopairs must be disjoint, which implies the bound. □

The proof shows we can find protopairs as follows. For each A of $\mathcal{S}$, we construct the complementary sequence B and look it up in the suffix tree. If B is in $\mathcal{S}$, then A, B is a candidate protopair. Identifying all candidates takes $O(n)$ time. Testing a candidate takes $O(k^2)$ time, and there are $O(k)$ candidates to test. Thus the time to find a protopair is $O(n + k^3)$. We have the following.

Theorem 1 *For equal-length, single-crossover recombination without point mutation, Bottleneck History can be solved in $O(n + k^3)$ time using $O(n + k^2)$ space.*

For unequal crossover we do not know of a nontrivial bound on the number of protopairs. However, we can avoid testing all $\binom{k}{2}$ pairs of sequences with the following observation.

Lemma 2 *Let $\mathcal{R}(A, B)$ be the set of sequences in $\mathcal{S}$ that are reachable from A and B. If A, B is not a protopair, then no C, D in $\mathcal{R}(A, B)$ is a protopair.*

Proof Immediate from the fact that $\mathcal{R}(C, D) \subseteq \mathcal{R}(A, B)$. □

[3] This is one of the mechanisms by which repeats are introduced in biological sequences.

Thus for unequal crossover our search proceeds as follows. During the search we maintain a set of candidate protopairs, which can be represented by an undirected graph whose vertex set is $\mathcal{S}$ and which is initially the complete graph. We repeatedly choose an edge $\{A, B\}$ and test the pair. If $\mathcal{R}(A, B)$ is all of $\mathcal{S}$ we report A, B and halt. Otherwise, we remove from the graph all edges between vertices in $\mathcal{R}(A, B)$, and repeat.

In practice we expect this to reduce the number of tests significantly. Nevertheless in the worst case this heuristic does not prevent us from testing $\Omega(k^2)$ candidates. A test takes $O(k^2)$ time, the same as for equal crossover, which gives the following bound.

Theorem 2 *For unequal-length, single-crossover recombination without point mutation, Bottleneck History can be solved in $O(n + k^4)$ time using $O(n + k^2)$ space.*

We remark that it is possible to write down a set of strings for which the algorithm, even with the above heuristic, tests $\Omega(k^2)$ candidates and spends $\Omega(k^4)$ time in total, which eliminates the possibility of a better amortized bound.

Note also that the algorithms in Theorems 1 and 2 actually find *all* protopairs in the quoted time bounds, which is more than we require. This recurs throughout our models, in that in the worst case we do not know how to find one protopair in less time than finding all protopairs.

2.2 Recombination with point mutation

We can modify our algorithm to accomodate point mutation as follows.

For the preprocessing, we compute two quantities for each ordered pair (A, B) of sequences: $\text{PrefCost}(A, B, i)$, which is the minimum cost to edit a prefix of A into the prefix of B ending at position i; and $\text{SufCost}(A, B, i)$, which is the minimum cost to edit a suffix of A into the suffix of B starting at position $i + 1$.

During the search, we maintain two quantities for each unreached sequence C: $\text{MinPrefCost}(C, i)$, which is the minimum of $\text{PrefCost}(A, C, i)$ over all A that have been reached; and $\text{MinSufCost}(C, i)$, which is the minimum of $\text{SufCost}(B, C, i)$ over all B that have been reached. Then an unreached sequence C is reachable if and only if, for some position i,

$$\text{MinPrefCost}(C, i) \ + \ \text{MinSufCost}(C, i) \ + \ \chi \ \leq \ d.$$

Notice that, as with pure recombination, the same sequence may contribute both the prefix and the suffix of C. Again this is legitimate, and models the crossing of a sequence with a copy of itself.

Computing PrefCost and SufCost for all pairs of sequences takes $O(n^2)$ time using standard sequence comparison methods [35]. Finding the next reachable sequence by evaluating the above inequality takes $O(n)$ time. Adding the sequence to the reached set and updating MinPrefCost and MinSufCost also takes $O(n)$ time. A search from a candidate protopair reaches $O(k)$ sequences, so the time to test a pair is $O(kn)$. At worst we test $O(k^2)$ pairs, which we summarize as follows.

Theorem 3 *For unequal-length, single-crossover recombination with point mutation, Bottleneck History can be solved in $O(n^2 + k^3 n)$ time using $O(kn)$ space.*

3 Multiple crossover

With single-crossover recombination we could find the next reachable sequence by separately considering sequences contributing a prefix and sequences contributing a suffix. In other words, we did not have to consider pairs of parents explicitly. With multiple-crossover recombination, however, we cannot separate the interaction of two parents in this way, so we explicity consider them paired together.

Accordingly, our preprocessing phase enumerates triples of sequences, and for each triple (A, B, C), determines the recombination cost of C with respect to A and B. Every triple whose recombination cost is within the cost bound is placed into a directed hypergraph. Our search phase then explores this hypergraph to find a protopair and a spanning history.

In the search phase we test a candidate protopair by breadth-first search over the above hypergraph. Edges of the directed hypergraph are stored on out-edge lists: for every A, we have a list of entries of the form (B, C), where (A, B, C) is a hyperedge in the graph. During the search we maintain two lists: Reached, which is the set of sequences that have been reached from the candidate protopair, and Reachable, which is the set of sequences not in Reached that may be formed from the reached set with one multiple-crossover recombination. To test a candidate $\{A, B\}$, the algorithm initializes Reached to the set $\{A, B\}$ and Reachable to the set of C such that (A, B, C) is in the graph. The algorithm then repeatedly removes a sequence C from Reachable, adds it to Reached, and examines all out-edges of C. For each sequence D such that (X, C, D) is in the graph where X is on Reached and D is not on Reached or Reachable, the algorithm adds D to Reachable, and repeats. The process halts when Reachable becomes empty.

Testing a protopair thus takes $O(m)$ time, where m is the number of edges in the hypergraph. We again employ the heuristic of Lemma 2. In the worst-case this tests $O(k^2)$ pairs, which gives the following.

Theorem 4 *Given a hypergraph of m recombination triples over k sequences, a protopair and an associated spanning history can be found, if they exist, in $O(k^2 m)$ time and $O(m)$ space.*

Note that in the worst case $m = O(k^3)$, though m can be much smaller, so this is at worst $O(k^5)$ time and $O(k^3)$ space.

Note also that in the time bound of Theorem 4 we could simply test all the pairs, so one might ask, is it possible to test *all* pairs faster than testing each pair individually? It appears not. Testing all pairs is essentially computing the transitive closure of a directed hypergraph, and the bound of Theorem 4 matches the best known bound for that problem [3, 4].

We also note that with a simple modification the algorithm can find a protopair of *minimum bottleneck cost*. We can maintain list Reachable as a heap, where vertices are prioritized by the weight of the least-cost in-edge that touches them from set Reached. Using a Fibonacci heap [9], removing a vertex of minimum priority takes $O(\log k)$ time, while updating the priority of a vertex when examining an out-edge takes $O(1)$ time. This adds $O(k \log k)$ time to the test for a candidate protopair, and finds a spanning history of minimum bottleneck cost for the given pair. Testing all pairs, and retaining the protopair with minimum bottleneck cost, takes a total of $O(k^2 m + k^3 \log k)$ time.

There may be many spanning histories that have minimum bottleneck cost. We remark that the spanning history found by this algorithm, as well as having minimum bottleneck cost, is *lexicographically minimum* for the given protopair in the following sense. A linearization of a spanning history is an ordering of its recombinations so that for any recombination (A, B, C) in the history, A and B are produced before C in the ordering. For the spanning history reported by the algorithm we take the linearization given by the order in which sequences are reached. Compare two linearizations by lexicographic order on the cost of their recombinations. Over all spanning histories from a given protopair, and all linearizations of these spanning histories, the spanning history found by the algorithm is lexicographically minimum.

Finally, we note that an alternate approach to finding a protopair of minimum bottleneck cost is to perform a binary search on the bottleneck values, assuming an integer cost-function. With this approach, we can run the algorithm of Theorem 4 at a given cost bound, which allows us to use the heuristic of Lemma 2. This takes a total of $O(k^2 m \log d)$ time, where integer d is the cost bound.

In the next two subsections we consider the preprocessing step, namely how to compute the recombination cost of a triple (A, B, C). We consider pure recombination first, then recombination with point mutation. Both are treated in the case of unequal crossover.

3.1 Pure recombination

For pure recombination the problem is to determine whether sequence C can be formed from A and B solely by crossover, and if it can be formed in this manner, to determine the minimum number of crossovers. When restricted to one crossover we could test for feasibility with a simple greedy strategy: match as much of C as possible first by a prefix of A, then by a suffix of B; if this matches all of C, it can be formed from A and B with one crossover.

When we allow multiple crossovers, this greedy strategy is no longer correct. For example, sequence $C =$ `abab` can be formed from $A =$ `ab` and $B =$ `ba` with two crossovers: take the first `a` from A, crossover to B to match `ba`, then return to A to match the final `b`. But greedily taking the longest initial match to A, the prefix `ab`, makes it impossible to pick up the final `b` when returning from the crossover.

We can determine the minimum number of crossovers, however, by dynamic programming. For a string S, let us write $S_{i,j}$ for the substring of S consisting of the ith through jth characters. Define $P_A(i, j, c)$ to be the smallest x such that $A_{1,x}$ and $B_{1,i}$ can form $C_{1,j}$ with exactly c crossovers, ending in a match to B. If there is no such x we take $P_A(i, j, c)$ to be infinite, or equivalently a value exceeding the length of A.

Similarly, define $P_B(i, j, c)$ to be the smallest x such that $A_{1,i}$ and $B_{1,x}$ can form $C_{1,j}$ with exactly c crossovers, ending in a match to A.

In terms of P_A and P_B, the minimum number of crossovers to form C from A and B is the smallest d such that

$$P_A(|B|, |C|, d) \ \leq \ |A| \quad \text{or} \quad P_B(|A|, |C|, d) \ \leq \ |B|.$$

We can compute P_A and P_B recursively as follows.

Associated with a recombination there is a natural alignment expressing how characters of C are alternately matched to characters in A and B. For a recombination, such an alignment specifies for each position i in C whether the character at i comes from A or B as well as the position of the corresponding character of A or B. Consider an alignment $\mathcal{A}$ associated with $P_A(i, j, c)$, namely an alignment ending in B that matches $C_{1,j}$ against $A_{1,x}$ and $B_{1,i}$, where in particular x, the length of the prefix of A, is as small as possible. We say such an alignment $\mathcal{A}$ is *leftmost* in A.

The useful property of a leftmost alignment is that it may be constructed from a smaller alignment that is also leftmost. By definition the last match in $\mathcal{A}$ aligns C_j to B_i, assuming these characters agree. (Of course if characters C_j and B_i are different, there is no corresponding alignment.) For the next-to-last match in $\mathcal{A}$ there are two cases. The next-to-last match either aligns C_{j-1} to B_{i-1} or, if a crossover occurs, to the character A_x that is as far to the left in A as possible.

In the first case $P_A(i,j,c)$ is given by $P_A(i-1,j-1,c)$. In the second case $P_A(i,j,c)$ is given by position x, where x is the smallest position such that $C_{1,j-1}$ can be formed from $A_{1,x}$ and $B_{1,i-1}$ with exactly $c-1$ crossovers, ending in a match to A. This position is simply the smallest x such that the alignment of $C_{1,j-1}$ against $A_{1,x}$ that is leftmost in B ends at or before position $i-1$. In terms of P_B this is

$$\min\Big\{x \;\Big|\; P_B(x, j{-}1, c{-}1) \leq i{-}1\Big\},$$

which we call $Q_A(i-1, j-1, c-1)$. This gives the following system of recurrences.

$$\begin{aligned}
P_A(i,j,c) &= \begin{cases} \min\Big\{P_A(i{-}1, j{-}1, c), \\ \quad Q_A(i{-}1, j{-}1, c{-}1)\Big\}, \\ \quad \text{if } B_i = C_j; \\ \infty, \text{ otherwise;} \end{cases} \\
Q_A(i,j,c) &= \min\Big\{Q_A(i{-}1, j, c),\; R_A(i,j,c)\Big\}, \\
R_A(i,j,c) &= \min\Big\{x \;\Big|\; P_B(x,j,c) = i\Big\}.
\end{aligned}$$

The recurrences for P_B, Q_B, and R_B are symmetric, and may be obtained by interchanging A and B.

The primary boundary condition is that $P_A(0,0,0) = 0$ and $P_B(0,0,0) = 0$. All other values are correctly determined from the recurrences as long as we take P, Q, and R to be infinite whenever $i \leq 0$, $j \leq 0$, or $c < 0$. For infinity we may use the value $\max(|A|, |B|) + 1$.

Figure 1 gives an algorithm that computes the recombination cost from these recurrences. The algorithm takes as input A, B, C, and a bound d on the number of crossovers. It accumulates P and Q using eight 2-dimensional arrays, and R using two 1-dimensional vectors. The indices of the P-arrays and the R-vectors both start from zero, while the second index of the Q-arrays, which is the crossover number, starts at -1. At the beginning of the jth iteration of the outermost loop, array P holds the values $P(\cdot, j-1, \cdot)$, while array P' will accumulate the values $P(\cdot, j, \cdot)$. Similarly, array Q holds the values $Q(\cdot, j{-}1, \cdot)$, while array Q' will accumulates the values $Q(\cdot, j, \cdot)$. In the cth iteration of the next outermost loop, vector R accumulates $R(\cdot, j, c)$.

The algorithm determines P' for a new value of j from the entries of P and Q for the previous value of j, using equation (1). Equation (1) effectively defines $R(\cdot, j, c)$ as the inverse of $P(\cdot, j, c)$. Once P' is known, the algorithm evaluates R by computing the inverse of a column of P'. Once R is known, the algorithm evaluates Q', in preparation for computing P' at the next value of j, using equation (1). This determines P and Q in $O(1)$ time per entry, and R in $O(1)$ amortized time per entry. A total of $O(d{\cdot}|A|{\cdot}|C| + d{\cdot}|B|{\cdot}|C|)$ entries are evaluated, which gives the running time of the algorithm. The space for all arrays is $O(d{\cdot}|A| + d{\cdot}|B|)$.

Thus the recurrences may be evaluated in the following time and space.

```
function RecombinationCost (A, B, C, d) begin

    Initialize arrays P_A, P'_A, P_B, P'_B, Q_A, Q'_A, Q_B, Q'_B to ∞.
    P_A(0,0) := 0
    P_B(0,0) := 0

    for j := 1 to |C| do begin
        for c := 0 to d do begin
            for i := 1 to |B| do
                if B_i = C_j then
                    P'_A(i,c) := min (P_A(i−1,c), Q_A(i−1,c−1))
                else
                    P'_A(i,c) := ∞
            for i := 1 to |A| do
                if A_i = C_j then
                    P'_B(i,c) := min (P_B(i−1,c), Q_B(i−1,c−1))
                else
                    P'_B(i,c) := ∞

            Initialize vectors R_A and R_B to ∞.
            for i := |A| down to 1 do
                R_A(P'_B(i,c)) := i
            for i := |B| down to 1 do
                R_B(P'_A(i,c)) := i

            for i := 1 to |B| do
                Q'_A(i,c) := min (Q'_A(i−1,c), R_A(i))
            for i := 1 to |A| do
                Q'_B(i,c) := min (Q'_B(i−1,c), R_B(i))
        end
        Swap arrays P_A and P'_A, P_B and P'_B, Q_A and Q'_A, Q_B and Q'_B.
    end

    for c := 0 to d do
        if P_A(|B|,c) ≤ |A|  or  P_B(|A|,c) ≤ |B| then
            return c
    return ∞
end
```

Figure 1 Algorithm for recombination cost without point mutation. Parameter d bounds the number of crossovers that are allowed.

Theorem 5 *For multiple-crossover recombination without point mutation, Recombination Cost can be solved in $O(cl^2)$ time and $O(cl)$ space, where c is the maximum number of crossovers that are allowed, and l is the length of the longest sequence.*

Combining this with Theorem 4 gives the following.

Corollary 1 *For multiple-crossover recombination without point mutation, Bottleneck History can be solved in $O(ckn^2 + k^2m)$ time and $O(cl + m)$ space.*

We remark that c is a small constant in practice, usually between 1 and 10, so the bounds of Theorem 5 are essentially quadratic time and linear space to determine the minimum number of crossovers. By applying the divide and conquer technique of Hirschberg [17], one can obtain the corresponding alignment, as well as the minimum number of crossovers, within the same time and space.

3.2 Recombination with point mutation

When point mutations are allowed, we do not know of a generalization of the recurrences for P_A and P_B that allows us to compute the recombination cost in quadratic time for a constant number of crossovers. Instead we develop a recurrence for recombination cost in terms of the lengths of prefixes of A, B, and C. This yields a cubic time algorithm.

Let $D_A(i, j, k)$ be the recombination cost of $C_{1,k}$ with respect to $A_{1,i}$ and $B_{1,j}$, given that the recombination ends by aligning a suffix of $A_{1,i}$ to a suffix of $C_{1,k}$. Let $D_B(j, i, k)$ be the same for a recombination that ends by aligning a suffix of $B_{1,j}$ to a suffix of $C_{1,k}$. Then the recombination cost of C with respect to A and B is

$$\min\Big\{D_A(|A|, |B|, |C|),\ D_B(|B|, |A|, |C|)\Big\}.$$

Consider $D_A(i, j, k)$. A minimum cost recombination that produces $C_{1,k}$ from $A_{1,i}$ and $B_{1,j}$ by aligning a suffix of $A_{1,i}$ to a suffix of $C_{1,k}$, ends in one of four ways. It either

- aligns a suffix of $A_{1,i-1}$ to a suffix of $C_{1,k-1}$ and *substitutes* C_k for A_i, or
- aligns a suffix of $A_{1,i-1}$ to a suffix of $C_{1,k}$ and *deletes* A_i, or
- aligns a suffix of $A_{1,i}$ to a suffix of $C_{1,k-1}$ and *inserts* C_k, or
- aligns a substring of $B_{1,j}$ to a suffix of $C_{1,k-1}$, *crosses over* into A, and substitutes C_k for A_i.

Notice we do not have to consider a crossover into A followed by the deletion of A_i, since crossing over to the next character in A will save the cost of the deletion. We also do not have to consider a crossover into A followed by the insertion of C_k, since the insertion will have been considered in the alignment of B to C before the crossover.

In each of the four cases, the recombination leading up to the last operation must be of minimum cost. For the case of a crossover, let $E_A(i, j, k)$ be the minimum cost of a recombination that produces $C_{1,k}$ from $A_{1,i}$ and $B_{1,j}$, given that the recombination ends by aligning a *substring* of $A_{1,i}$ to a suffix of $C_{1,k}$. Let $E_B(j, i, k)$ be the same for a recombination that ends by aligning a substring of $B_{1,j}$ to a suffix of $C_{1,k}$. We have the following recurrences.

$$D_A(i,j,k) \;=\; \min\begin{cases} D_A(i-1,j,k-1) + \delta(A_i, C_k), \\ D_A(i-1,j,k) \quad + \delta(A_i, \epsilon), \\ D_A(i,j,k-1) \quad + \delta(\epsilon, C_k), \\ E_B(j,i-1,k-1) + \delta(A_i, C_k) + \chi, \end{cases}$$

$$E_A(i,j,k) \;=\; \min\Big\{E_A(i-1,j,k),\; D_A(i,j,k)\Big\}.$$

The recurrences for D_B and E_B are symmetric, and may be obtained by interchanging A and B.

The primary boundary condition is that $D_A(0,x,0) = 0$ and $D_B(0,x,0) = 0$ for $x \geq 0$. If we take D and E to be infinite whenever $i < 0$, $j < 0$, or $k < 0$, all other values are correctly determined.

We can evaluate $D_A(|A|,|B|,|C|)$ and $D_B(|B|,|A|,|C|)$ using these recurrences in $O(|A| \cdot |B| \cdot |C|)$ time and $O(|A| \cdot |B|)$ space. Each entry of D and E can be obtained by examining a constant number of entries with smaller coordinates, and there are a total of $O(|A| \cdot |B| \cdot |C|)$ entries. The values of D and E can be accumulated using four arrays of size $O(|A| \cdot |B|)$, for example by storing $D(\cdot,\cdot,k-1)$ in order to evaluate $D(\cdot,\cdot,k)$. This gives the following.

Theorem 6 *For multiple-crossover recombination with point mutation, Recombination Cost can be solved in $O(l^3)$ time and $O(l^2)$ space, where l is the length of the longest sequence.*

Combining this with Theorem 4 gives the following.

Corollary 2 *For multiple-crossover recombination with point mutation, Bottleneck History can be solved in $O(n^3 + k^2 m)$ time and $O(l^2 + m)$ space.*

Since m, the number of edges in the hypergraph, can be $O(k^3)$, this is $O(n^3 + k^5)$ time in the worst case.

4 Generalizations and their complexity

We have treated Bottleneck Recombination History as two separate problems: a type of edit distance problem, in which we want to find a recombination of minimum cost for three sequences, and a type of directed spanning tree problem, in which we want to find a tree in a directed hypergraph where a root is generalized to a pair of vertices. Our focus in Sections 2 and 3 has been on the definition of recombination and how it affects the running time of our algorithms. In this section we study the definition of spanning history, and how it affects the complexity of the problem. We consider two generalizations.

Definition 3 The *Generalized Bottleneck History Problem* is the following. Given a set $\mathcal{S}$ of sequences, a cost function specified by δ and χ, a bottleneck bound d, and a protoset bound c, is there a *protoset* $\mathcal{P} \subseteq \mathcal{S}$ with $|\mathcal{P}| \leq c$ such that, starting from $\mathcal{P}$, there is a spanning history for $\mathcal{S}$ where every recombination has cost at most d? □

Definition 4 The *Minimum Recombination History Problem* is the following. Given a set $\mathcal{S}$ of sequences, a recombination cost function δ and χ, and a cost bound d, is there a protopair and associated spanning history for $\mathcal{S}$ of *total cost* at most d? □

In the following we show that both of these generalizations are NP-complete. For a set $\mathcal{S}$ of sequences, let $\mathcal{S}|_i$ denote $\{S_i \mid S \in \mathcal{S}\}$, the set of characters obtained by looking at the ith position in each sequence. We use the following simple property of a protoset.

Lemma 3 *Let $\mathcal{S}$ be a set of strings, all of the same length, and $\mathcal{P} \subseteq \mathcal{S}$ be a protoset under pure recombination. Then $\mathcal{P}|_i = \mathcal{S}|_i$ at every position i.*

Proof Since strings in $\mathcal{S}$ are all of the same length and no characters are inserted or deleted, all recombinations must involve equal-length crossover. Thus each character at position i in a string of $\mathcal{S}$ can be traced to a character at position i of some string in protoset $\mathcal{P}$. In the absence of substitutions, this implies $\mathcal{P}|_i \supseteq \mathcal{S}|_i$. Since clearly $\mathcal{P}|_i \subseteq \mathcal{S}|_i$, the lemma follows. □

We now prove that Generalized Bottleneck History is NP-complete. The proof in fact shows that the problem is NP-complete for the simplest model of recombination: equal-length, single-crossover recombination without point mutation.

Theorem 7 *Generalized Bottleneck History is NP-complete.*

Proof Since the feasiblity of a candidate protoset can be tested in polynomial time using the same directed hypergraph approach given in Section 3, but now starting the search from the candidate protoset, Generalized Bottleneck History is in NP. We prove NP-hardness by a reduction from Exact Cover by 3-Sets [11, p. 221]. Recall that an instance is a ground set X together with a family $\mathcal{F} \subseteq 2^X$ of 3-element subsets of X. The problem is to determine whether there is a subfamily $\mathcal{C} \subseteq \mathcal{F}$ that is a partition of X. Such a subfamily $\mathcal{C}$ is called an exact cover of X.

Given an instance $X, \mathcal{F}$ of Exact Cover by 3-Sets, we construct an instance $\mathcal{S}, \delta, \chi, c, d$ of Generalized Bottleneck History as follows. Denote the elements of X by $x_1, x_2, \ldots, x_n$, and the 3-sets in $\mathcal{F}$ by $F_1, F_2, \ldots, F_k$. We assume without loss of generality that each element of X is a member of some 3-set. Sequences in $\mathcal{S}$ are over the alphabet $\{\mathtt{0}, \mathtt{1}\}$, and essentially encode subsets of X. We describe $\mathcal{S}$ by the following disjoint union of four classes of sequences:

$$\mathcal{S} \;=\; \{S_1, S_2, \ldots, S_k\} \;\cup\; \mathcal{X}^{(2)} \;\cup\; \mathcal{X}^{(1)} \;\cup\; \mathcal{M},$$

where

- sequences S_1, S_2, …, S_k correspond to the 3-sets in $\mathcal{F}$,
- set $\mathcal{X}^{(2)}$ contains sequences corresponding to all 2-element subsets of X,
- set $\mathcal{X}^{(1)}$ contains sequences corresponding to all 1-element subsets of X, and
- set $\mathcal{M} = \{M_0, M_{n+1}\}$ contains two "mask sequences."

All sequences have $n + 2$ characters. We associate a sequence with a subset of X as follows. Character positions are numbered $0, 1, \ldots, n + 1$, and positions 1 through n correspond to elements x_1 through x_n, with position $1 \leq i \leq n$ containing character $\mathtt{1}$ if and only if element x_i is in the corresponding subset. In a sequence corresponding to a subset, positions 0 and $n + 1$ contain character $\mathtt{0}$. In the two mask sequences of $\mathcal{M}$, all positions contain character $\mathtt{0}$, except position 0 in sequence M_0 and position $n + 1$ in sequence M_{n+1}, which both contain character $\mathtt{1}$.

For the cost function, we set $\chi = 1$, $\delta(a,a) = 0$ for $a \in \{\mathtt{0},\mathtt{1}\}$, and $\delta(a,b) = 2$ for all distinct $a,b \in \{\epsilon,\mathtt{0},\mathtt{1}\}$. For the bottleneck bound we take $d = \chi$, which implies every recombination consists of one crossover with no insertions, deletions, or substitutions, and for the protoset bound we set

$$c \;=\; \frac{n}{3} + 2.$$

We claim that with this δ, χ, and d, set $\mathcal{S}$ has a protoset of size c if and only if $\mathcal{F}$ contains an exact cover of X.

For suppose $\mathcal{P} \subseteq \mathcal{S}$ is a protoset of size c. By Lemma 3, the characters in $\mathcal{P}$ cover the characters in $\mathcal{S}$ at every position. Since only M_0 contains a $\mathtt{1}$ at position 0, and only M_{n+1} contains a $\mathtt{1}$ at position $n+1$, $\mathcal{P}$ must contain M_0 and M_{n+1}. The remaining sequences of $\mathcal{P}$ must cover the $\mathtt{1}$'s in positions 1 through n. Since any sequence can cover at most three $\mathtt{1}$'s, and as there are only $\frac{n}{3}$ sequences remaining in $\mathcal{P}$, these sequences must have $\mathtt{1}$'s at disjoint positions, and must be a subset of $\{S_1,\ldots,S_k\}$. The corresponding subset of $\mathcal{F}$ is an exact cover of X.

Now suppose $\mathcal{C} \subseteq \mathcal{F}$ is an exact cover of X. Let $\mathcal{P} = \{S_i \mid F_i \in \mathcal{C}\} \cup \mathcal{M}$, which is a set of size c. From $\mathcal{P}$, we can form all strings in $\mathcal{S}$ with recombinations of cost d, in a history of four levels:

(1) Form all strings in $\mathcal{X}^{(1)}$ and $\mathcal{X}^{(2)}$ whose $\mathtt{1}$'s are a prefix or suffix of the $\mathtt{1}$'s in an S_i in $\mathcal{P}$ by crossing with the mask sequences in $\mathcal{M}$.
(2) Form the remainder of $\mathcal{X}^{(1)}$ by crossing the masks in $\mathcal{M}$ with the strings of $\mathcal{X}^{(2)}$ that have just been generated.
(3) Form the remainder of $\mathcal{X}^{(2)}$ by crossing strings from $\mathcal{X}^{(1)}$.
(4) Form the remaining strings in $\{S_1,\ldots,S_k\} - \mathcal{P}$ by crossing strings from $\mathcal{X}^{(1)}$ and $\mathcal{X}^{(2)}$.

Thus $\mathcal{P}$ is a protoset for $\mathcal{S}$ of size c. $\square$

Theorem 8 *Minimum Recombination History is NP-complete.*

Proof We again use a reduction from Exact Cover by 3-Sets. Given an instance $X,\mathcal{F}$ of Exact Cover by 3-Sets, we construct an instance $\mathcal{S},\delta,\chi,d$ of Minimum Recombination History as follows. Sequences in $\mathcal{S}$ are over the alphabet $\{\mathtt{0},\mathtt{1},-,*\}$ and are organized as follows:

$$\mathcal{S} \;=\; \{S_1,S_2,\ldots,S_k\} \,\cup\, \mathcal{X}^{(1)} \,\cup\, \mathcal{X}^{(2)} \,\cup\, \mathcal{M} \,\cup\, \mathcal{P},$$

where

- the sequences in sets $\{S_1,\ldots,S_k\}$, $\mathcal{X}^{(2)}$, and $\mathcal{X}^{(1)}$ correspond, as before, to the 3-sets in $\mathcal{F}$, all 2-element subsets of X, and all 1-element subsets of X,
- set $\mathcal{M} = \{M\}$ contains a mask sequence M, and
- set $\mathcal{P} = \{P,D\}$ contains what will effectively be a protosequence, P, and a dummy sequence, D.

All sequences in $\mathcal{S}$ except dummy sequence D have $n+1$ characters. Character positions are numbered $0,1,\ldots,n$, and positions 1 through n correspond to elements x_1 through x_n. Sequences corresponding to subsets are over the alphabet $\{\mathtt{0},\mathtt{1}\}$. They start with character $\mathtt{0}$, and have a $\mathtt{1}$ at position i if and only if element x_i is in the associated subset. Mask sequence M is $\mathtt{00}\cdots\mathtt{0}$. Sequence P has the form $*{-}{-}\cdots{-}$, and dummy sequence D is the character $\mathtt{*}$.

The cost function has the form,

- $\delta(*, \mathtt{0}) = \alpha$,
- $\delta(-, \mathtt{1}) = \beta$, and
- $\delta(-, \mathtt{0}) = 0$,

where it will suffice to have $\alpha = 1$, $\beta = 2$, and $\chi = 2$. All other values of δ exceed the cost bound, which is

$$d \;=\; \alpha(\tfrac{n}{3}+1) \;+\; \beta n \;+\; \chi(|\mathcal{S}| - (\tfrac{n}{3}+3)).$$

In particular, no insertions or deletions are allowed.

We claim that with this δ and χ, set $\mathcal{S}$ has a protopair and associated spanning history of total cost d if and only if $\mathcal{F}$ contains an exact cover of X.

For suppose $\mathcal{C} \subseteq \mathcal{F}$ is an exact cover of X. Let $\mathcal{Q}$ be the set $\{S_i \mid F_i \in \mathcal{C}\} \cup \mathcal{M}$. From $\mathcal{Q}$ we can generate $\mathcal{S} - \mathcal{P}$ with recombinations of one crossover, using a history analogous to the four-level history given in the proof of Theorem 8. This history has cost $\chi(|\mathcal{S}| - (\frac{n}{3}+3))$. We can generate $\mathcal{Q}$ from sequence P by pure substitution, which adds a top level to the history of cost $\alpha(\frac{n}{3}+1) + \beta n$. Thus $\mathcal{P} = \{P, D\}$ is a protopair for $\mathcal{S}$ of total cost d.

Now suppose $\mathcal{S}$ has a protopair of total cost d. No string of $\mathcal{S}$ can generate P or D within cost d, so the protopair must be $\mathcal{P} = \{P, D\}$. The cost function does not permit insertion or deletion, and all sequences except D are of the same length, so all recombinations must involve equal crossover. This being the case, the history must create an initial `1` in each of positions 1 through n, and each of these `1`'s must be created by a substitution from P. In the spanning history starting from $\{P, D\}$, there is an earliest sequence to contain the character `1` at any given position. Let $\mathcal{Q}$ be this set of sequences for positions 1 through n, together with the mask sequence M. We claim that $\mathcal{C} = \{F_i \mid S_i \in \mathcal{Q}\}$ is an exact cover of X. To prove this it suffices to show that $\mathcal{Q}$ contains exactly $\frac{n}{3}+1$ sequences.

Consider the cost of generating $\mathcal{S} - \{P, D\}$ from $\mathcal{P}$. All sequences in $\mathcal{Q}$ are produced by recombination with sequence P. These recombinations may or may not involve crossover. A recombination that involves crossover incurs at least cost χ plus the cost of turning dashes into `1`'s. A recombination that does not involve crossover must use pure substitution, which incurs cost α (to change the initial `*` into a `0`) plus the cost of turning dashes into `1`'s. Ignoring the cost of turning on `1`'s, which must be at least βn for all of set $\mathcal{Q}$, the cost of producing a sequence in $\mathcal{Q}$ by crossover, minus the cost of producing it by pure substitution, is at least $\chi - \alpha$. Notice this is a positive difference.

The remaining sequences of $\mathcal{S} - \mathcal{Q}$ can be produced either by pure substitution from P, or by pure crossover. Each of these sequences contains a `1`. Thus the cost of producing a remaining sequence by pure substitution, minus the cost of producing it by pure crossover, is at least $\beta + \alpha - \chi$. This is also a positive difference.

The five-level history described earlier first produces $\mathcal{Q}$ from $\mathcal{P}$ by pure substitution and then the remainder of $\mathcal{S}$ by pure crossover. A history of this form in which $\mathcal{Q}$ has size $\frac{n}{3}+1$ has a total cost of d, and the above analysis shows that any other history has cost exceeding d. In short, if $\mathcal{S}$ has a protoset of total cost d, then $\mathcal{Q}$ must contain exactly $\frac{n}{3}+1$ sequences. This implies $\mathcal{C}$ is an exact cover of X. □

We note that the alphabets in the reductions for Generalized Bottleneck History and Minimum Recombination History have four or fewer characters. This implies the problems are NP-complete for sequences over the DNA and protein alphabets.

The structure of function δ in the reduction for Minimum Recombination History is rather artificial, however. While the proof demonstrates that an algorithm that accepts any cost matrix for δ must solve an NP-complete problem, we do not know whether the problem remains NP-complete for a more realistic cost function. For instance it would be interesting to know whether Minimum Recombination History is NP-complete when δ is a metric. (It is worth keeping in mind however that the most frequently used cost matrix for protein sequences, the so-called PAM matrix, is not a metric.)

For a fixed bound on the size of a protoset, Generalized Bottleneck History can be solved in polynomial time by enumerating and testing protosets, as in Section 3. On the other hand, approximating the smallest protoset when its size is unbounded appears difficult. While we do not show it here, an algorithm that in polynomial time finds a protoset whose size is at most a constant factor larger than the minimum, for an arbitrary recombination hypergraph, would give a constant-factor approximation for Set Cover. Recent non-approximability results [29, 7] show that such an algorithm would imply P = NP. However, while the minimization problem of approximating the size of the smallest protoset to within a constant factor appears difficult, the maximization problem of approximating the size of the *complement* of a smallest protoset to within a factor of $\frac{1}{2}$ can be solved in polynomial time, as we have recently discovered.

5 Conclusion

In this paper we have defined two new problems, Recombination Cost and Bottleneck Recombination History, and shown that they can be solved in polynomial time. These are just two of the types of problems that arise with recombination, and even here many questions remain.

We suspect that the algorithm of Section 2 for reconstructing a history with pure, equal-length, single-crossover recombination is worst-case optimal. Is $\Omega(n+k^3)$ time a lower bound?

With respect to computing recombination costs, is there an $O(cl^2)$ time algorithm for recombination with point mutation, where l is the length of the longest sequence and c is the maximum number of crossovers? Is there an $O(cl)$ time algorithm for recombination without point mutation? Can recombination cost with point mutation be computed in less than $O(l^2)$ space?

With respect to finding protopairs, can one protopair be found in less time than it takes to find all protopairs? If not, can all protopairs be found in less time than it takes to test each pair individually? Failing this, is there an algorithm whose expected search time is less than its worst-case time?

Clearly there are many possibilities for further study.

Acknowledgements

The first author wishes to thank Jotun Hein for introducing him to the problem of reconstructing evolutionary trees with recombination, Joe Felsenstein and Todd Wareham for pointers to some of the literature, and R. Ravi for discussions on approximation algorithms. We also thank the referee for suggestions that improved the definitions of terms.

References

[1] Agarwala, Richa and David Fernandez-Baca. "A polynomial-time algorithm for the perfect phylogeny problem when the number of character states is fixed." Proceedings of the 34th IEEE *Symposium on Foundations of Computer Science*, 140–147, 1993.

[2] Apostolico, A. "The myriad virtues of subword trees." *Combinatorial Algorithms on Words*, A. Apostolico and Z. Galil, editors, NATO ASI series F, volume 12, Springer-Verlag, Berlin, 85–96, 1985.

[3] Ausiello, G., A. D'Atri, and D. Sacca. "Minimal representation of directed hypergraphs." *SIAM Journal on Computing* 15:2, 419–431, 1986.

[4] Ausiello, Giorgio, Umberto Nanni, and Giuseppe F. Italiano. "Dynamic maintenance of directed hypergraphs." *Theoretical Computer Science* 72, 97–117, 1990.

[5] Bafna, Vineet and Pavel A. Pevzner. "Genome rearrangements and sorting by reversals." Proceedings of the 34th IEEE *Symposium on Foundations of Computer Science*, 148–157, 1993.

[6] Bafna, Vineet and Pavel A. Pevzner. "Sorting by transpositions." Proceedings of the 6th ACM-SIAM *Symposium on Discrete Algorithms*, 614–621, 1995.

[7] Bellare, M., S. Goldwasser, C. Lund, and A. Russell. "Efficient probabilistically checkable proofs and applications to approximation." Proceedings of the 25th ACM *Symposium on the Theory of Computing*, 294–304, 1993.

[8] Farach, Martin, Sampath Kannan, and Tandy Warnow. "A robust model for finding optimal evolutionary trees." *Algorithmica* 13, 155–179, 1995.

[9] Fredman, Michael L. and Robert Endre Tarjan. "Fibonacci heaps and their uses in improved network optimization algorithms." *Journal of the ACM* 34:3, 596–615, 1987.

[10] Galil, Zvi and Raffaele Giancarlo. "Speeding up dynamic programming with applications to molecular biology." *Theoretical Computer Science* 64, 107–118, 1989.

[11] Garey, Michael R. and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. W.H. Freeman, New York, 1979.

[12] Gusfield, Dan. "Efficient algorithms for inferring evolutionary trees." *Networks* 21, 19–28, 1991.

[13] Hannenhalli, Sridhar and Pavel A. Pevzner. "Transforming cabbage into turnip: polynomial algorithm for sorting signed permutations by reversals." Proceedings of the 27th ACM *Symposium on the Theory of Computing*, 178–189, 1995.

[14] Hannenhalli, Sridhar and Pavel A. Pevzner. "Transforming men into mice: polynomial algorithm for genomic distance problem." Proceedings of the 36th IEEE *Symposium on Foundations of Computer Science*, 581–592, 1995.

[15] Hein, Jotun. "Reconstructing the evolution of sequences subject to recombination using parsimony." *Mathematical Biosciences* 98, 185–200, 1990.

[16] Hein, Jotun. "A heuristic method to reconstruct the history of sequences subject to recombination." *Journal of Molecular Evolution* 36, 396–405, 1993.

[17] Hirschberg, D.S. "A linear space algorithm for computing maximal common subsequences." *Communications of the ACM* 18:6, 341–343, 1975.

[18] Hudson, R.R. "Properties of a neutral model with intragenic recombination." *Theoretical Population Biology* 23, 183–201, 1983.

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

[20] Kannan, Sampath and Tandy Warnow. "Inferring evolutionary history from DNA sequences." Proceedings of the 31st IEEE *Symposium on Foundations of Computer Science*, 1990.

[21] Kannan, Sampath K. and Eugene W. Myers. "An algorithm for locating non-overlapping regions of maximum alignment score." Proceedings of the 4th *Symposium on Combinatorial Pattern Matching*, Springer-Verlag Lecture Notes in Computer Science 684, 74–86, 1993.

[22] Kaplan, N. and Hudson, R. "The use of sample genealogies for studying a selectively neutral M-loci model with recombination." *Theoretical Population Biology* 28, 382–396, 1985.

[23] Kececioglu, John and Dan Gusfield. "Reconstructing a history of recombinations from a set of sequences." Proceedings of the 5th ACM-SIAM *Symposium on Discrete Algorithms*, 471–480, 1994.

[24] Kececioglu, John D. and R. Ravi. "Of mice and men: Algorithms for evolutionary distances between genomes with translocation." Proceedings of the 6th ACM-SIAM *Symposium on Discrete Algorithms*, 604–613, 1995.

[25] Kececioglu, John and David Sankoff. "Exact and approximation algorithms for sorting by reversals, with application to genome rearrangement." *Algorithmica* 13:1/2, 180–210, 1995.

[26] Landau, Gad M. and Jeanette P. Schmidt. "An algorithm for approximate tandem repeats." Proceedings of the 4th *Symposium on Combinatorial Pattern Matching*, Springer-Verlag Lecture Notes in Computer Science 684, 120–133, 1993.

[27] Lathrop, G.M. "Evolutionary trees and admixture: phylogenetic inference when some populations are hybridized." *Annals of Human Genetics* 46, 245–255, 1982.

[28] Lowrance, R. and R.A. Wagner. "An extension of the string-to-string correction problem." *Journal of the ACM* 22, 177–183, 1975.

[29] Lund, Carsten and Mihalis Yannakakis. "On the hardness of approximating minimization problems." Proceedings of the 25th ACM *Symposium on the Theory of Computing*, 286–293, 1993.

[30] Mansfield, Anthony. "An algorithm for a merge recognition problem." *Discrete Applied Mathematics* 4, 193–197, 1982.

[31] Mansfield, Anthony. "On the computational complexity of a merge recognition problem." *Discrete Applied Mathematics* 5, 119–122, 1983.

[32] McDade, L. "Hybrids and phylogenetic systematics I. Patterns of character expression in hybrids and their implications for cladistic analysis." *Evolution* 44:6, 1685–1700, 1990.

[33] Miller, Webb and Eugene W. Myers. "Sequence comparison with concave weighting functions." *Bulletin of Mathematical Biology* 50:2, 97–120, 1988.

[34] Nelson, G. "Reticulation in cladograms." *Advances in Cladistics*, volume 2, N.I. Platnick and V.A. Funk, editors, Columbia University Press, New York, 105–111, 1983.

[35] Sankoff, David and Joseph B. Kruskal, editors. *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison.* Addison-Wesley, Reading, MA, 1983.

[36] Sawyer, S. "Statistical tests for detecting gene conversion." *Molecular Biology and Evolution* 6, 526–539, 1989.

[37] Schieber, B. and U. Vishkin. "On finding lowest common ancestors: simplification and parallelization." *SIAM Journal on Computing* 17:6, 1253–1262, 1988.

[38] Sneath, P.H.A. "Cladistic representation of reticulate evolution." *Systematic Zoology* 24, 360–368, 1975.

[39] Stahl, Franklin W. "Genetic recombination." *Scientific American*, 90–101, February 1987.

[40] Stephens, J.C. "Statistical methods of DNA sequence analysis: detection of intragenic recombination or gene conversion." *Molecular Biology and Evolution* 2:6, 539–556, 1985.

[41] Svante, Pääbo. "Ancient DNA." *Scientific American*, 86–92, November 1993.

[42] Wagner, R.A. "On the complexity of the extended string-to-string correction problem." Proceedings of the 7th ACM *Symposium on Theory of Computing*, 218–222, 1975.

[43] Wareham, Harold Todd. *On the Computational Complexity of Inferring Evolutionary Trees.* M.S. thesis, Technical Report 9301, Department of Computer Science, Memorial University of Newfoundland, St. John's, Newfoundland, Canada, 1993.

[44] Warmuth, Manfred K. and David Haussler. "On the complexity of iterated shuffle." *Journal of Computer and System Sciences* 28, 345–358, 1984.

[45] Warnow, Tandy J. "Tree compatibility and inferring evolutionary history." Proceedings of the 4th ACM-SIAM *Symposium on Discrete Algorithms*, 382–391, 1993.

[46] Watson, James D., Nancy H. Hopkins, Jeffrey W. Roberts, Jean Argetsinger Steitz, and Alan M. Weiner. *Molecular Biology of the Gene*, 4th edition, volume 1. Benjamin-Cummings, Menlo Park, CA, 1987.