Efficient algorithms for analyzing segmental duplications with deletions and inversions in genomes

Crystal L Kahn1*, Shay Mozes1*, Benjamin J Raphael1,2*

Abstract

Background: Segmental duplications, or low-copy repeats, are common in mammalian genomes. In the human genome, most segmental duplications are mosaics comprised of multiple duplicated fragments. This complex genomic organization complicates analysis of the evolutionary history of these sequences. One model proposed to explain this mosaic patterns is a model of repeated aggregation and subsequent duplication of genomic sequences.

Results: We describe a polynomial-time exact algorithm to compute duplication distance, a genomic distance defined as the most parsimonious way to build a target string by repeatedly copying substrings of a fixed source string. This distance models the process of repeated aggregation and duplication. We also describe extensions of this distance to include certain types of substring deletions and inversions. Finally, we provide a description of a sequence of duplication events as a context-free grammar (CFG).

Conclusion: These new genomic distances will permit more biologically realistic analyses of segmental duplications in genomes.

Introduction

Genomes evolve via many types of mutations ranging in scale from single nucleotide mutations to large genome rearrangements. Computational models of these mutational processes allow researchers to derive similarity measures between genome sequences and to reconstruct evolutionary relationships between genomes. For example, considering chromosomal inversions as the only type of mutation leads to the so-called reversal distance problem of finding the minimum number of inversions/reversals that transform one genome into another [1]. Several elegant polynomial-time algorithms have been found to solve this problem (cf. [2] and references therein). Developing genome rearrangement models that are both biologically realistic and computationally tractable remains an active area of research.

Duplicated sequences in genomes present a particular challenge for genome rearrangement analysis and often make the underlying computational problems more difficult. For instance, computing reversal distance in genomes with duplicated segments is NP-hard [3]. Models that include both duplications and other types of mutations - such as inversions - often result in similarity measures that cannot be computed efficiently. Thus, most current approaches for duplication analysis rely on heuristics, approximation algorithms, or restricted models of duplication [3-7]. For example, there are efficient algorithms for computing tandem duplication histories [8-11] and whole-genome duplication histories [12,13]. Here we consider another class of duplications: large segmental duplications (also known as low-copy repeats) that are common in many mammalian genomes [14]. These segmental duplications can be quite large (up to hundreds of kilobases), but their evolutionary history remains poorly understood, particularly in primates. The mystery surrounding them is due in part to their complex organization; many segmental duplications are found within contiguous regions of the genome called duplication blocks that contain mosaic patterns of smaller repeated segments, or duplicons [15]. Duplication
blocks that are located on different chromosomes, or that are separated by large physical distances on a chromosome, often share sequences of duplicons [16]. These conserved sequences suggest that these duplicons were copied together across large genomic distances. One hypothesis proposed to explain these conserved mosaic patterns is a two-step model of duplication [14]. In this model, a first phase of duplications copies duplicons from the ancestral genome and aggregates these copies into primary duplication blocks. Then in a second phase, portions of these primary duplication blocks are copied and reinserted into the genome at disparate loci forming secondary duplication blocks. In [17], we introduced a measure called duplication distance that models the duplication of contiguous substrings over large genomic distances. We used duplication distance in [18] to find the most parsimonious duplication scenario consistent with the two-step model of segmental duplication. The duplication distance from an ancestral genome and aggregates these copies into primary duplication blocks. Then in a second phase, portions of these primary duplication blocks are copied and reinserted into the genome at disparate loci forming secondary duplication blocks.

In [17], we introduced a measure called duplication distance that models the duplication of contiguous substrings over large genomic distances. We used duplication distance in [18] to find the most parsimonious duplication scenario consistent with the two-step model of segmental duplication. The duplication distance from an ancestral genome and aggregates these copies into primary duplication blocks. Then in a second phase, portions of these primary duplication blocks are copied and reinserted into the genome at disparate loci forming secondary duplication blocks.

In [17], we introduced a measure called duplication distance that models the duplication of contiguous substrings over large genomic distances. We used duplication distance in [18] to find the most parsimonious duplication scenario consistent with the two-step model of segmental duplication. The duplication distance from an ancestral genome and aggregates these copies into primary duplication blocks. Then in a second phase, portions of these primary duplication blocks are copied and reinserted into the genome at disparate loci forming secondary duplication blocks.

In [17], we introduced a measure called duplication distance that models the duplication of contiguous substrings over large genomic distances. We used duplication distance in [18] to find the most parsimonious duplication scenario consistent with the two-step model of segmental duplication. The duplication distance from an ancestral genome and aggregates these copies into primary duplication blocks. Then in a second phase, portions of these primary duplication blocks are copied and reinserted into the genome at disparate loci forming secondary duplication blocks.

In [17], we introduced a measure called duplication distance that models the duplication of contiguous substrings over large genomic distances. We used duplication distance in [18] to find the most parsimonious duplication scenario consistent with the two-step model of segmental duplication. The duplication distance from an ancestral genome and aggregates these copies into primary duplication blocks. Then in a second phase, portions of these primary duplication blocks are copied and reinserted into the genome at disparate loci forming secondary duplication blocks.

In [17], we introduced a measure called duplication distance that models the duplication of contiguous substrings over large genomic distances. We used duplication distance in [18] to find the most parsimonious duplication scenario consistent with the two-step model of segmental duplication. The duplication distance from an ancestral genome and aggregates these copies into primary duplication blocks. Then in a second phase, portions of these primary duplication blocks are copied and reinserted into the genome at disparate loci forming secondary duplication blocks.

In [17], we introduced a measure called duplication distance that models the duplication of contiguous substrings over large genomic distances. We used duplication distance in [18] to find the most parsimonious duplication scenario consistent with the two-step model of segmental duplication. The duplication distance from an ancestral genome and aggregates these copies into primary duplication blocks. Then in a second phase, portions of these primary duplication blocks are copied and reinserted into the genome at disparate loci forming secondary duplication blocks.
**Definition 5.** The duplication distance from a source string \( x \) to a target string \( y \) is the minimum number of duplicate operations from \( x \) that generates \( y \) from an initially empty target string. That is, \( y = \emptyset \circ \delta_x(s_1, t_1, p_1) \circ \delta_x(s_2, t_2, p_2) \circ \ldots \circ \delta_x(s_i, t_i, p_i) \).

To compute the duplication distance from \( x \) to \( y \), we assume that every character in \( y \) appears at least once in \( x \). Otherwise, the duplication distance is undefined.

**Duplication Distance**

In this section we review the basic recurrence for computing duplication distance that was introduced in [18]. The recurrence examines the characters of the target string, \( y \), and considers the sets of characters of \( y \) that could have been generated, or copied from the source string in a single duplicate operation. Such a set of characters of \( y \) necessarily correspond to a substring of the source \( x \) (see Def. 4). Moreover, these characters must be a subsequence of \( y \). This is because, in a sequence of duplicate operations, once a string is copied and inserted into the target string, subsequent duplicate operations do not affect the order of the characters in the previously inserted string. Because every character of \( y \) is generated by exactly one duplicate operation, a sequence of duplicate operations that generates \( y \) partitions the characters of \( y \) into disjoint subsequences, each of which is generated in a single duplicate operation. A more interesting observation is that these subsequences are mutually non-overlapping. We formalize this property as follows.

**Lemma 1 (Non-overlapping Property).** Consider a source string \( x \) and a sequence of duplicate operations of the form \( \delta_x(s_1, t_1, p_1) \circ \ldots \circ \delta_x(s_i, t_i, p_i) \) that generates the final target string \( y \) from an initially empty target string. The subsequences \( x_{s_i:t_i} \) of \( x \) that are duplicated during the construction of \( y \) appear as mutually non-overlapping subsequences of \( y \).

**Proof.** Consider a sequence of duplicate operations \( \delta_x(s_1, t_1, p_1), \ldots, \delta_x(s_i, t_i, p_i) \) that generates \( y \) from an initially empty target string. For \( 1 \leq i \leq k \), let \( z^i \) be the intermediate target string that results from \( \delta_x(s_1, t_1, p_1) \circ \ldots \circ \delta_x(s_i, t_i, p_i) \). Note that \( z^k = y \). For \( j \leq i \), let \( S^i_j \) be the subsequence of \( z^i \) that corresponds to the characters duplicated by the \( j^{th} \) operation. We shall show by induction on the length \( i \) of the sequence that \( S^i_1, S^i_2, \ldots, S^i_i \) are pairwise non-overlapping subsequences of \( z^i \). For the base case, when there is a single duplicate operation, there is no non-overlap property to show. Assume now that \( S^i_{1-1}, \ldots, S^i_{l-1} \) are mutually non-overlapping subsequences in \( z^i \). For the induction step note that, by the definition of a duplicate operation, \( S^i_1 \) is inserted as a contiguous substring into \( z^{i-1} \) at location \( p_i \) to form \( z^i \). Therefore, for any \( j, j' < i \), if \( S^i_{j-1} \) and \( S^i_{j'-1} \) are non-overlapping in \( z^{i-1} \) then \( S^i_j \) and \( S^i_{j'} \) are non-overlapping in \( z^i \). It remains to show that for any \( j < i \), \( S^i_j \) and \( S^i_i \) are non-overlapping in \( z^i \). There are two cases: (1) the elements of \( S^i_j \) are either all smaller or all greater than the elements of \( S^i_i \) or (2) \( S^i_j \) is inside of \( S^i_i \) in \( z^i \).
(Definition 3). In either case, $S_j^i$ and $S_j^i$ are not overlapping in $z'$ as required.

The non-overlapping property leads to an efficient recurrence that computes duplication distance. When considering subsequences of the final target string $y$ that might have been generated in a single duplicate operation, we rely on the non-overlapping property to identify sub-strings of $y$ that can be treated as independent subproblems. If we assume that some subsequence $S$ of $y$ is produced in a single duplicate operation, then we know that all other subsequences of $y$ that correspond to duplicate operations cannot overlap the characters in $S$. Therefore, the substrings of $y$ in between successive characters of $S$ define subproblems that are computed independently.

In order to find the optimal (i.e. minimum) sequence of duplicate operations that generate $y$, we must consider all subsequences of $y$ that could have been generated by a single duplicate operation. The recurrence is based on the observation that $y_1$ must be the first (i.e. leftmost) character to be copied from $x$ in some duplicate operation. There are then two cases to consider: either (1) $y_1$ was the last (or rightmost) character in the substring that was copied to produce $y_1$, or (2) $y_1$ was not the last character in the substring that was duplicated from $x$ to generate $y_1$.

The recurrence defines two quantities: $d(x, y)$ and $d_i(x, y)$. We shall show, by induction, that for a pair of strings, $x$ and $y$, the value $d(x, y)$ is equal to the duplication distance from $x$ to $y$ and that $d_i(x, y)$ is equal to the duplication distance from $x$ to $y$ under the restriction that the character $y_1$ is copied from index $i$ in $x$, i.e. $x_i$ generates $y_1$. $d(x, y)$ is found by considering the minimum among all characters $x_i$ of $x$ that can generate $y_1$, see Eq. 1.

As described above, we must consider two possibilities in order to compute $d(x, y)$. Either:

Case 1: $y_1$ was the last (or rightmost) character in the substring of $x$ that was copied to produce $y_1$, (see Fig. 4), or

Case 2: $x_{i+1}$ is also copied in the same duplicate operation as $x_i$, possibly along with other characters as well (see Fig. 5).

For case one, the minimum number of duplicate operations is one - for the duplicate that generates $y_1$ - plus the minimum number of duplicate operations to generate the suffix of $y$, giving a total of $1 + d(x, y_2, |y|)$ (Fig. 4). For case two, Lemma 1 implies that the minimum number of duplicate operations is the sum of the optimal numbers of operations for two independent subproblems. Specifically, for each $j > 1$ such that $x_{j+1} = y_j$ we compute: (i) the minimum number of duplicate operations needed to build the substring $y_2, j-1$, namely $d(x, y_2, j-1)$, and (ii) the minimum number of duplicate operations needed to build the string $y_1 y_{j+1}, |y|$, given that $y_1$ is generated by $x_i$ and $y_j$ is generated by $x_{i+1}$. To compute the latter, recall that since $x_i$ and $x_{i+1}$ are copied in the same duplicate operation, the number of duplicates necessary to generate $y_1 y_{j+1}, |y|$ using $x_i$ and $x_{i+1}$ is equal to the number of

---

**Figure 4** Recurrence: Case 1. $y_1$ is generated from $x_i$ in a duplicate operation where $y_1$ is the last (rightmost) character in the copied substring (Case 1). The total duplication distance is one plus the duplication distance for the suffix $y_2, |y|$. 

---
Let \( y \) be a sequence of duplicate operations that generate \( y \). Consider a minimum-length sequence of duplicates \( x_i \) generated by \( x_i \) starting with \( x_i \) such that \( y_i \) is generated by \( x_i \). Consider \( y_j \) generated by \( x_j \) in the optimal sequence \( x_j \). We first show that \( y_j \) is generated by \( x_j \).

\[ d(y_j) = 0 \]

For \( |y_j| = 1 \), since we assume there is at least one \( i \) for which \( x_i = y_j \), \( d(y_j) = OPT(x, y_j) \). By definition, the recurrence also evaluates to 1. For the inductive step, assume that \( OPT(x, y^*) = d(x, y^*) \) and \( OPT(x, y) = d(x, y^*) \) for any string \( y^* \) shorter than \( y \). We first show that \( OPT(x, y_j) \leq d_i(x, y_j) \). Since \( OPT(x, y) = \min_i OPT_i(x, y) \), this also implies \( OPT(x, y) \leq d_i(x, y) \). We describe different sequences of duplicate operations that generate \( y_j \) using \( x_i \) starting with \( x_i \) to generate \( y_i \):

1. Consider a minimum-length sequence of duplicates that generates \( y_j \). By the inductive hypothesis its length is \( d(x, y_j) \). By duplicating \( y_j \), we obtain a sequence of duplicates that generates \( y_j \) whose length is \( 1 + d(x, y_j) \).

2. For every \( j : y_j = x_{i+1} \), consider a minimum-length sequence of duplicates that generates \( y_j \) using \( x_{i+1} \) to produce \( y_j \) and a minimum-length sequence of duplicates that generates \( y_j \).

By the inductive hypothesis their lengths are \( d_i(x, y_j) \) and \( d(x, y_{j+1}) \) respectively. By extending the start index \( s \) of the duplicate operation that starts with \( x_i \) to produce \( y_j \) to start with \( x_i \) and produce \( y_j \) as we produce \( y \) with the same number of duplicate operations.

Since \( OPT_i(x, y) \) is at most the length of any of these options, it is also at most their minimum. Hence,

\[ OPT_i(x, y) \leq \min \left\{ 1 + d(x, y_j), \sum_{\ell \geq 1} \left( d(x, y_{j+\ell}) + d_i(x, y_{j+\ell}) \right) \right\} \]

To show the other direction (i.e. that \( d(x, y) \leq OPT(x, y) \) and \( d(x, y) \leq OPT(x, y) \)), consider a minimum-length sequence of duplicate operations that generate \( y \) from \( x \). Using \( x_i \) to generate \( y_i \). There are a few cases:

- If \( y_i \) is generated by a duplicate operation that only duplicates \( x_i \), then \( OPT_i(x, y) = 1 + OPT(x, y) \). By the inductive hypothesis this equals \( 1 + d(x, y_{2,y}) \) which is at least \( d_i(x, y) \).
- Otherwise, \( y_i \) is generated by a duplicate operation that copies \( x_i \) and also duplicates \( x_{i+1} \) to generate some character \( y_j \). In this case the sequence \( \Delta \) of duplicates that generates \( y_{j+1} \) must appear after the duplicate operation that generates \( y_i \) and \( y_j \) because \( y_{j+1} \) is inside (Definition 3) of \( y_i \) and \( y_j \). Without loss of generality, suppose \( \Delta \) is ordered after all the other duplicates so that first \( y_1 \), \( y_2 \) is generated, and then \( \Delta \) generates \( y_2 \) \( y_j \) between \( y_i \). Hence, \( OPT_i(x, y) = OPT_i(x, y_{2,y}) + OPT(x, y_{j+1}) \). Since in the optimal sequence \( x_i \) generates \( y_i \) in the same
duplicate operation that generates \( y_i \) from \( x_{i+1} \), we have \( OPT_i(x, y_i| y) = OPT_{i+1}(x, y_{i+1}| y) \). By the inductive hypothesis, \( OPT_i(x, y_{z,j}) + OPT_{i+1}(x, y_{j+1}| y) = d_i(x, y_{z,j+1}) + d_{i+1}(x, y_{j+1}| y) \) which is at least \( d_i(x, y) \). \( \square \)

This recurrence naturally translates into a dynamic programming algorithm that computes the values of \( d(x, y) \) and \( d_{i+1}(x, y) \) for various target strings. To analyze the running time of this algorithm, note that both \( x_{z,j} \) and \( y_{j+1| y} \) are substrings of \( y \). Since the set of substrings of \( y \) is closed under taking substrings, we only encounter substrings of \( y \). Also note that since \( i \) is chosen from the set \( \{ i : x_i = y_1 \} \), there are \( O(\mu(x)) \) choices for \( i \), where \( \mu(x) \) is the maximal multiplicity of a character in \( x \). Thus, there are \( O(\mu(x)|y|^2) \) different values to compute. Each value is computed by considering the minimization over at most \( \mu(y) \) previously computed values, so the total running time is bounded by \( O(|y|^2\mu(x)|y|) \), which is \( O(|y|^3|y|) \) in the worst case. As with most dynamic programming approaches, this algorithm (and all others presented in subsequent sections) can be extended through trace-back to reconstruct the optimal sequence of operations needed to build \( y \). We omit the details.

**Extending to Affine Duplication Cost**

It is easy to extend the recurrence relations in Eqs. (1), (2) to handle costs for duplicate operations. In the above discussion, the cost of each duplicate operation is 1, so the sum of costs of the operations in a sequence that generates a string \( y \) is just the length of that sequence. We next consider a more general cost model for duplication in which the cost of a duplicate operation \( \delta_i(s, t, p) \) is \( \Delta_1 + (t - s + 1) \Delta_2 \) (i.e., the cost is affine in the number of duplicated characters). Here \( \Delta_1, \Delta_2 \) are some non-negative constants. This extension is obtained by assigning a cost of \( \Delta_2 \) to each duplicated character, except for the last character in the duplicated string, which is assigned a cost of \( \Delta_1 + \Delta_2 \). We do that by adding a cost term to each of the cases in Eq. 2. If \( x_i \) is the last character in the duplicated string (case 1), we add \( \Delta_1 + \Delta_2 \) to the cost. Otherwise \( x_i \) is not the last duplicated character (case 2), so we add just \( \Delta_2 \) to the cost. Eq. (2) thus becomes

\[
d_i(x, y) = \min \left\{ \Delta_1 + \Delta_2 + d_i(x, y_{z,j+1}), \min_{\{ j : x_{z,j} \neq y_{j+1} \}} [d_i(x, y_{z,j+1}) + d_{i+1}(x, y_{j+1}) + \Delta_2] \right\}
\]

The running time analysis for this recurrence is the same as for the one with unit duplication cost.

**Duplication-Deletion Distance**

In this section we generalize the model to include deletions. Consider the intermediate string \( z \) generated after some number of duplicate operations. A deletion operation removes a contiguous substring \( z_{r_i} \ldots z_j \) of \( z \), and subsequent duplicate and deletion operations are applied to the resulting string.

**Definition 6.** A delete operation, \( \tau(s, t) \), deletes a substring \( z_{r_i} \ldots z_j \) of the target string \( z \), thus making \( z \) shorter. Specifically, if \( z = z_1 \ldots z_{r_i} \ldots z_j \ldots z_m \), then \( z * \tau(s, t) = z_1 \ldots z_{r_i} z_{r_i+1} \ldots z_m \). See Figure 6.

The cost associated with \( \tau(s, t) \) depends on the number \( t - s + 1 \) of characters deleted and is denoted \( \Phi(t - s + 1) \).

**Definition 7.** The duplication-deletion distance from a source string \( x \) to a target string \( y \) is the cost of a minimum sequence of duplicate operations from \( x \) and deletion operations, in any order, that generates \( y \).

We now show that although we allow arbitrary deletions from the intermediate string, it suffices to consider deletions from the duplicated strings before they are pasted into the intermediate string, provided that the cost function for deletion, \( \Phi(\cdot) \) is non-decreasing and obeys the triangle inequality.

**Definition 8.** A duplicate-delete operation from \( x, \eta_k \) \((t_1, j_1, t_2, j_2, \ldots, t_k, j_k, p)\), for \( i_1 < j_1 < i_2 < j_2 < \ldots < i_k < j_k \) copies the subsequence \( x_{i_1} \ldots x_{j_1} x_{i_2} \ldots x_{j_2} \ldots x_{i_k} \ldots x_{j_k} \) of the source string \( x \) and pastes it into a target string at position \( p \). Specifically, if \( x = x_1 \ldots x_m \) and \( z = z_1 \ldots z_m \), then \( z * \eta_k(t_1, j_1, \ldots, i_k, j_k, p) = z_1 \ldots z_{p-1} x_{i_1} \ldots x_{j_1}, x_{i_2} \ldots x_{j_2} \ldots x_{i_k} \ldots x_{j_k} x_p \ldots z_m \).

The cost associated with such a duplication-deletion is \( \Delta_1 + (j_k - i_k + 1) \Delta_2 + \sum_{i_{k+1}}^{j_k} \Phi(i_{k+1} - j_k - 1) \). The first two terms in the cost reflect the affine cost of duplicating an entire substring of length \( j_k - i_k + 1 \), and the second term reflects the cost of deletions made to that substrings.

**Lemma 2.** If the affine cost for duplications is non-decreasing and \( \Phi(\cdot) \) is non-decreasing and obeys the triangle inequality then the cost of a minimum sequence of duplicate and delete operations that generates a target string \( y \) from a source string \( x \) is equal to the cost of a minimum sequence of duplicate-delete operations that generates \( y \) from \( x \).

**Proof.** Since duplicate operations are a special case of duplicate-delete operations, the cost of a minimal sequence of duplicate-delete operations and delete operations that generates \( y \) from \( x \) is the sum of the affine costs of the duplicate operations. Let \( \tau(s, t) \) be a sequence of deletion operations. Then the cost of the sequence \( \eta_k(t_1, j_1, \ldots, i_k, j_k, p) \) is equal to the cost of the sequence \( \tau(s, t) * \eta_k(t_1, j_1, \ldots, i_k, j_k, p) \). Since the cost is affine in the number of duplicated characters, the cost of the sequence \( \tau(s, t) * \eta_k(t_1, j_1, \ldots, i_k, j_k, p) \) is equal to the cost of the sequence \( \eta_k(t_1, j_1, \ldots, i_k, j_k, p) \). Therefore, the cost of the sequence \( \eta_k(t_1, j_1, \ldots, i_k, j_k, p) \) is equal to the cost of the sequence \( \tau(s, t) \).

![Figure 6 A delete operation](image)
operations that generates $y$ cannot be more than that of a sequence of just duplicate operations and delete operations. We show the (stronger) claim that an arbitrary sequence of duplicate-delete and delete operations that produces a string $y$ with cost $c$ can be transformed into a sequence of just duplicate-delete operations that generates $y$ with cost at most $c$ by induction on the number of delete operations. The base case, where the number of deletions is zero, is trivial. Consider the first delete operation, $\tau$. Let $k$ denote the number of duplicate-delete operations that precede $\tau$, and let $z$ be the intermediate string produced by these $k$ operations. For $i = 1, \ldots, k$, let $S_i$ be the subsequence of $x$ that was used in the $i$th duplicate-delete operation. By lemma 1, $S_1, \ldots, S_k$ form a partition of $z$ into disjoint, non-overlapping subsequences of $z$. Let $d$ denote the substring of $z$ to be deleted. Since $d$ is a contiguous substring, $S_i \cap d$ is a (possibly empty) substring of $S_i$ for each $i$. There are several cases:

1. $S_i \cap d = \emptyset$. In this case we do not change any operation.
2. $S_i \cap d = S_i$. In this case all characters produced by the $i$th duplicate-delete operation are deleted, so we may omit the $i$th operation altogether and decrease the number of characters deleted by $\tau$. Since $\Phi (\cdot)$ is non-decreasing, this does not increase the cost of generating $z$ (and hence $y$).
3. $S_i \cap d$ is a prefix (or suffix) of $S_i$. Assume it is a prefix. The case of suffix is similar. Instead of deleting the characters $S_i \cap d$ we can avoid generating them in the first place. Let $r$ be the smallest index in $S_i \setminus d$ (that is, the first character in $S_i$ that is not deleted by $\tau$). We change the $i$th duplicate-delete operation to start at $r$ and decrease the number of characters deleted by $\tau$. Since the affine cost for duplications is non-decreasing and $\Phi (\cdot)$ is non-decreasing, the cost of generating $z$ does not increase.
4. $S_i \cap d$ is a non-empty substring of $S_i$ that is neither a prefix nor a suffix of $S_i$. We claim that this case applies to at most one value of $i$. This implies that after taking care of all the other cases $\tau$ only deletes characters in $S_i$. Then we change the $i$th duplicate-delete operation to also delete the characters deleted by $\tau$, and omit $\tau$. Since $\Phi (\cdot)$ obeys the triangle inequality, this will not increase the total cost of deletion. By the inductive hypothesis, the rest of $y$ can be generated by just duplicate-delete operations with at most the same cost. It remains to prove the claim. Recall that the set $\{ S_i \}$ is comprised of mutually non-overlapping subsequences of $z$. Suppose that there exist indices $i \neq j$ such that $S_i \cap d$ is a non-prefix/suffix substring of $S_i$ and $S_j \cap d$ is a non-prefix/suffix substring of $S_j$. There must exist indices of both $S_i$ and $S_j$ in $z$ that precede $d$, are contained in $d$, and succeed $d$. Let $i_p < i_c < j_i$ be three such indices of $S_i$ and let $j_p < j_c < i_p$. Without loss of generality, suppose $i_p < j_p$. It follows that $(i_p, i_c)$ and $(j_p, j_c)$ are alternating in $z$. So, $S_i$ and $S_j$ are overlapping which contradicts lemma 1.

To extend the recurrence from the previous section to duplication-deletion distance, we must observe that because we allow deletions in the string that is duplicated from $x$, if we assume character $x_i$ is copied to produce $y_1$, it may not be the case that the character $x_{i+1}$ also appears in $y$; the character $x_{i+1}$ may have been deleted. Therefore, we minimize over all possible locations $k > i$ for the next character in the duplicated string that is not deleted. The extension of the recurrence from the previous section to duplication-deletion distance is:

\[
\hat{d}(x, \emptyset) = 0, \quad \hat{d}(x, y) = \min_{\{i : x_i = y_j\}} \hat{d}(x, y),
\]

(4)

\[
\hat{d}_i(x, \emptyset) = 0, \quad \hat{d}_i(x, y) = \min \left\{ \Delta_i + \Delta_i + \hat{d}(x, y), \hat{d}(x, y_i), \hat{d}(x, y_{i+1}) \right\}.
\]

(5)

**Theorem 2.** $\hat{d}(x, y)$ is the duplication-deletion distance from $x$ to $y$. For $i : x_i = y_1$, $\hat{d}_i(x, y)$ is the duplication-deletion distance from $x$ to $y$ under the additional restriction that $y_1$ is generated by $x_i$.

The proof of Theorem 2 is almost identical to that of Theorem 1 in the previous section and is omitted. However, the running time increases; while the number of entries in the dynamic programming table does not change, the time to compute each entry is multiplied by the possible values of $k$ in the recurrence, which is $O(|x|)$.

The running time is $O(|x|^2|x|\mu(x)\mu(y))$, which is $O(|x|^3|x|^2)$ in the worst case. We conclude this section by showing, in the following lemma, that if both the duplicate and delete cost functions are the identity function (i.e. one per operation), then the duplication-deletion distance is equal to duplication distance without deletions.

**Lemma 3.** Given a source string $x$, a target string $y$. If the cost of duplication is 1 per duplicate operation, and the cost of deletion is 1 per delete operation, then $\hat{d}(x, y) = d(x, y)$.

**Proof.** First we note that if a target string $y$ can be built from $x$ in $d(x, y)$ duplicate operations, then the same sequence of duplicate operations is a valid sequence of duplicate and delete operations as well, so $d(x, y) = \hat{d}(x, y)$.

We claim that every sequence of duplicate and delete operations can be transformed into a sequence of
duplicate operations of the same length. The proof of this claim is similar to that of Lemma 2. In that proof we showed how to transform a sequence of duplicate and delete operations into a sequence of duplicate-delete operations of at most the same cost. We follow the same steps, but transform the sequence into an a sequence that consists of just duplicate operations without increasing the number of operations. Recall the four cases in the proof of Lemma 2. In the the first three cases we eliminate the delete operation without increasing the number of duplicate operations. Therefore we only need to consider the last case (Si ∩ d is a non-empty substring of Si that is neither a prefix nor a suffix of Si). Recall that this case applies to at most one value of i. Deleting Si ∩ d from Si leaves a prefix and a suffix of Si. We can therefore replace the ith duplicate operation and the delete operation with two duplicate operations, one generating the appropriate prefix of Si and the other generating the appropriate suffix of Si. This eliminates the delete operation without changing the number of operations in the sequence. Therefore, for any string y that results from a sequence of duplicate and delete operations, we can construct the same string using only duplicate operations (without deletes) using at most the same number of operations. So, d(x, y) is no greater than Δ(x, y).

### Definition 9
A signed string of length m over an alphabet Σ is an element of \((+,-) \times Σ^m\).

For example, \((+b -c -a +d)\) is a signed string of length 4. An inversion of a signed string reverses the order of the characters as well as their signs. Formally,

### Definition 10
The inverse of a signed string \(x = x_1 \ldots x_m\) is a signed string \(\overline{x} = -x_m \ldots -x_1\).

For example, the inverse of \((+b -c -a +d)\) is \((-d +a +c -b)\).

In a duplicate-invert operation a substring is copied from x and inverted before being inserted into the target string y. We allow the cost of inversion to be an affine function in the length \(ℓ\) of the duplicated inverted string, which we denote \(Θ_1 + ℓΘ_2\), where \(Θ_1, Θ_2 ≥ 0\). We still allow for normal duplicate operations.

### Definition 11
A duplicate-invert operation from x, \(\overline{θ}(s, t, p)\), copies an inverted substring \(-x_{s-1} \ldots -x_s\) of the source string x and pastes it into a target string at position p. Specifically, if \(x = x_1 \ldots x_m\) and \(z = z_1 \ldots z_m\) then \(z \oplus \overline{θ}(s, t, p) = z_1 \ldots z_{s-1} \overline{x}_{t-1} \ldots \overline{x}_p z_{p+1} \ldots z_m\).

The cost associated with each duplicate-invert operation is \(Θ_1 + (t - s + 1)Θ_2\).

### Definition 12
The duplication-inversion distance from a source string x to a target string y is the cost of a minimum sequence of duplicate and duplicate-invert operations from x, in any order, that generates y.

The recurrence for duplication distance (Eqs. 1, 3) can be extended to compute the duplication-inversion distance. This is done by introducing a term for inverted duplications whose form is very similar to that of the term for regular duplication (Eq. 3). Specifically, when considering the possible characters to generate \(y_i\), we consider characters in x that match either \(y_i\) or its inverse, \(-y_i\). In the former case, then, we use \(d^+_i(x, y)\) to denote the duplication-inversion distance with the additional restriction that \(y_i\) is generated by \(x_i\) without an inversion. The recurrence for \(d^+_i\) is the same as for \(d_i\) in Eq. 3. In the latter case, we consider an inverted duplicate in which \(y_i\) is generated by \(-x_i\). This is denoted by \(d^-_i\), which follows a similar recurrence. In this recurrence, since an inversion occurs, \(x_i\) is the last character of the duplicated string, rather than the first one. Therefore, the next character in x to be used in this operation is \(-x_{i+1}\) rather than \(x_{i+1}\). The recurrence for \(d^-_i\) also differs in the cost term, where we use the affine cost of the duplicate-invert operation. The extension of the recurrence to duplication-inversion distance is therefore:

\[
\begin{align*}
\delta(x, \varnothing) &= 0, \\
\delta_i(x, y) &= \min \left\{ \min_{\{y_i, -y_i\}} \delta^+_i(x, y), \min_{\{x_i, -x_i\}} \delta_i(x, y) \right\}, \\
\delta^-_i(x, y) &= \min \left\{ \min_{\{y_i, -y_i\}} \delta^+_i(x, y), \min_{\{x_i, -x_i\}} \delta_i(x, y) \right\}.
\end{align*}
\]

### Theorem 3
\(\overline{δ}(x, y)\) is the duplication-inversion distance from x to y. For \(i : x_i = y_i\), \(\delta_i^+(x, y)\) is the duplication-inversion distance from x to y under the additional restriction that \(y_i\) is generated by \(x_i\). For \(i : x_i = -y_i\), \(\delta_i^-(x, y)\) is the duplication-inversion distance from x to y under the additional restriction that \(y_i\) is generated by \(-x_i\).

The correctness proof is very similar to that of Theorem 1, only requiring an additional case for handling the case of a duplicate invert operation which is symmetric to the case of regular duplication. The asymptotic running time of the corresponding dynamic programming algorithm is \(O(|y|^2μ(x)μ(y))\). The analysis is identical to the one in section 3. The fact that we now consider either a duplicate or a duplicate-invert operation does not change the asymptotic running time.
Duplication-Inversion-Deletion Distance

In this section we extend the distance measure to include delete operations as well as duplicate and duplicate-invert operations. Note that we only handle deletions after inversions of the same substring. The order of operations might be important, at least in terms of costs. The cost of inverting \( (+a + b + c) \) and then deleting \(-b\) may be different than the cost of first deleting \(+b\) from \( (+a + b + c) \) and then inverting \( (+a + c) \).

Definition 13. The duplication-inversion-deletion distance from a source string \( x \) to a target string \( y \) is the cost of a minimum sequence of duplicate and duplicate-invert operations from \( x \) and deletion operations, in any order, that generates \( y \).

Definition 14. A duplicate-invert-delete operation from \( x \),

\[ \overline{\mathcal{T}}_z(z_1, z_2, z_3, \ldots, z_k, p) \]

pastes the string \(-x_{z_1} - x_{z_1-1} \ldots - x_{z_2} - x_{z_2-1} \ldots - x_{z_k} - x_{z_k-1} \ldots - x_{j_1} \ldots - x_{j_2} - x_{j_2-1} \ldots - x_i\) into a target string at position \( p \). Specifically, if \( x = x_1 \ldots x_m \) and \( z = z_1 \ldots z_k \), then \( z \cdot \overline{\mathcal{T}}_z(z_1, z_2, z_3, \ldots, z_k, p) = x_1 \ldots x_{z_1} - x_{z_1-1} \ldots - x_{z_2} - x_{z_2-1} \ldots - x_{z_k} - x_{z_k-1} \ldots - x_{j_1} \ldots - x_{j_2} - x_{j_2-1} \ldots - x_i \).

The cost of such an operation is \( \Theta_1 + (j_1 - i_1 + 1) \Theta_2 + \sum_{i=2}^{k-1} \Phi(i_{k-1} - j_{k-1} - 1) \). Similar to the previous section, it suffices to consider just duplicate-invert-delete and duplicate-delete operations, rather than duplicate, duplicate-invert and delete operations.

Lemma 4. If \( \Phi(\cdot) \) is non-decreasing and obeys the triangle inequality and if the cost of inversion is an affine non-decreasing function as defined above, then the cost of a minimum sequence of duplicate, duplicate-invert and delete operations that generates a target string \( y \) from a source string \( x \) is equal to the cost of a minimum sequence of duplicate-delete and duplicate-invert-delete operations that generates \( y \) from \( x \).

The proof of the lemma is essentially the same as that of Lemma 2. Note that in that proof we did not require all duplicate operations to be from the same string \( x \). Therefore, the arguments in that proof apply to our case, where we can regard some of the duplicates from \( x \) and some from the inverse of \( x \).

The recurrence for duplication-inversion-deletion distance is obtained by combining the recurrences for duplication-deletion (Eq. 5) and for duplication-inversion distance (Eq. 6). We use separate terms for duplicate-delete operations (\( \overline{d}^+ \)) and for duplicate-invert-delete operations (\( \overline{d}^- \)). Those terms differ from the terms in Eq. 6 in the same way Eq. 5 differs from Eq. 2; Because of the possible deletion we do not know that \( x_{i_1 + 1} \) (\( x_{i_1} \)) is the next duplicated character. Instead we minimize over all characters later (earlier) than \( x_i \).

The recurrence for duplication-inversion-deletion distance is therefore:

\[
\hat{d}(x, y) = \begin{cases} \hat{d}(x, y) = 0, & \hat{d}(x, y) = \min \left( \hat{d}^+(x, y), \hat{d}^-(x, y) \right), \\
\hat{d}(x, y) = 0, & \hat{d}(x, y) = \min \left( \hat{d}^+(x, y) + \hat{d}(x, y), \hat{d}^-(x, y) + \hat{d}(x, y) \right), \\
\hat{d}(x, y) = \min \left( \hat{d}^+(x, y) + \hat{d}(x, y), \hat{d}^-(x, y) + \hat{d}(x, y) \right), & \hat{d}(x, y) = \min \left( \hat{d}^+(x, y) + \hat{d}(x, y), \hat{d}^-(x, y) + \hat{d}(x, y) \right). 
\end{cases}
\]

Theorem 4. \( \overline{d}(x, y) \) is the duplication-inversion-deletion distance from \( x \) to \( y \). For \( i : x_i = y_1 \), \( \overline{d}^+(x, y) \) is the duplication-inversion-deletion distance from \( x \) to \( y \) under the additional restriction that \( y_1 \) is generated by \( x_1 \). For \( i : x_i = y_1 \), \( \overline{d}^-(x, y) \) is the duplication-inversion-deletion distance from \( x \) to \( y \) under the additional restriction that \( y_1 \) is generated by \(-x_i \).

The proof, again, is very similar to the proofs in the previous sections. The running time of the corresponding dynamic programming algorithm is the same (asymptotically) as that of duplication-deletion distance. It is \( O(|y|^3 \bigr|x\bigr| \mu(y) \mu(x)) \), where the multiplicity \( \mu(y) \) (or \( \mu(x) \)) is the number of times a character appears in the string \( y \) (or \( x \)), regardless of its sign.

In comparing the models of the previous section and the current one, we note that restricting the model of rearrangement to allow only duplicate and duplicate-invert operations (Section 5) instead of duplicate-invert-delete operations may be desirable from a biological perspective because each duplicate and duplicate-invert requires only three breakpoints in the genome, whereas a duplicate-invert-delete operation may be significantly more complicated, requiring more breakpoints.

Variants of Duplication-Inversion-Deletion Distance

It is possible to extend the model even further. We give here one detailed example which demonstrates how such extensions might be achieved. Other extensions are also possible. In the previous section we handled the model where the duplicated substring of \( x \) may be inverted in its entirety before being inserted into the target string. In the generalized model a substring of the duplicated string may be inverted before the string is inserted into \( y \). For example, we allow \( (+a + b + c + d + e + f) \) to become \( (+a + b + c + d - e - f) \) before being inserted into \( y \). In this model, the cost of duplicating a string of length \( m \) with an inversion of a substring of length \( \ell \) is \( \Delta_1 + m \Delta_2 + \Theta(\ell) \), for some non-negative monotonically increasing cost function \( \Theta \).

The way we extend the recurrence is by considering all possible substring inversions to the original string \( x \). For \( 1 \leq s \leq t \leq |x| \), let \( \tilde{x}^{s:t} \) be the string \( x_1 \ldots x_{s-1}x_{t+1} \ldots x_m \).
Kahn et al. define also make much sense unless the distance from the duplicated string, then it is possible fixing or a suffix of the duplicated string, only if in every substring of the duplicated string is not used in the duplication. Howver, restricting the inversion cost to be non-negative and monotonically increasing makes sure that those cases will not contribute to the minimization since inverting a character that is not duplicated will only increase the cost. The recurrence for duplication-deletion with arbitrary-substring-duplicate-inversions distance is given below.

\[
\begin{align*}
\hat{d}(x, y) & = 0, \\
\hat{d}_i(x, y) & = 0, \\
\hat{d}_j^i(x, y) & = \min_{k=0, \ldots, |y|} \min_{\ell=1, \ldots, |x_i|} \min_{\gamma = 1} \left[ \Delta_1 + \Delta_2 + O(t - s + 1) + \hat{d}(x, y_{2,\gamma}) + \hat{d}_\ell(x, y_{\ell,\gamma}) \right].
\end{align*}
\]

The running time is \( O(|y|^3|x|^5\mu(x)\mu(y)) \).

**Figure 7 Example parse tree.** An optimal parse tree \( T \) for \( y = \text{bloccd} \) where \( x = \text{abcd} \). The root production duplicates \( x_{5t} = \text{bcd} \). \( x_t \) generates \( y_t \) and \( x_5 \) generates \( y_5 \). The trees \( T_1 \) and \( T_2 \) are indicated. \( T_1 \) is an optimal parse tree for \( y_{j+1} = \text{bc} \). \( T_2 \) is an optimal parse tree for \( y_{j+1} = \text{cd} \).

**Duplication Distance as a Context-Free Grammar**

The process of generating a string \( y \) by repeatedly copying substrings of a source string \( x \) and pasting them into an initially empty target string is naturally described by a context-free grammar (CFG). This alternative view might be useful in understanding our algorithms and their correctness. Thus, we provide the basic idea behind this connection for the most simple variant of duplication distance: no inversions or deletions and the cost of each duplicate operation is 1. For a fixed source string \( x \), we construct a grammar \( G_x \) in which for every \( i, j \) such that \( 1 \leq i \leq j \leq |x| \), there is a production rule \( S \rightarrow Sx_Sx_{i+1}S \ldots Sx_jS \).

These production rules correspond to duplicating the substring \( x_{i,j} \). In addition there is a trivial production rule \( S \rightarrow \varepsilon \), where \( \varepsilon \) denotes the empty string. It is easy to see that the language described by this grammar is exactly the set of strings that can be duplicated from \( x \). The non-overlapping property (Lemma 1) is an immediate consequence of the structure of parse trees of CFGs. Finding the duplication distance from \( x \) to \( y \) is equivalent to finding a parse tree with a minimal number of non-trivial productions among all possible parse trees for \( y \).

Consider now the slightly different grammar obtained by removing the leading \( S \) to the left of \( x_t \) from each of the production rules, so that the new rules are of the form \( S \rightarrow x_tSx_{t+1}S \ldots Sx_jS \). It is not difficult to see that both grammars produce the same language and have the same minimal size parse tree for every string \( y \). The change only...
Figure 8 Example parse tree. An optimal parse tree $T$ for $y = dab$ where $x = abcd$. The root production duplicates just $x_4 = d$. The tree $T_1$ is indicated. $T_2$ is empty (not indicated). The root production is not part of $T_1$.

restricts the order in which rules are applied. For example, $y_1$ is always produced by the first production rule.

The recurrence for $d(x, y)$ naturally arises by observing that if $T$ is an optimal parse tree for $y$ in which the first production rule generates $y_1$ by $x_i$ and $y_j$ by $x_{i+1}$, then the subtree $T_1$ of $T$ that generates $y_2, j-1$ is a valid parse tree which is optimal for $y_2, j-1$. Similarly, the tree $T_2$ obtained by deleting $x_i$ and $T_1$ from $T$ is a valid parse tree which is optimal for $y_j, |y|$ under the restriction that $y_j$ must be generated by $x_{i+1}$ (see Fig. 7). Moreover, $T_1$ and $T_2$ are disjoint trees which contain all non trivial productions in $T$. This explains the term $d(x, y_2, j-1) + d_{i+1}(x, y_j, |y|)$ in Eq. 2, which is the heart of the recursion. The minimization over $y_j, y_i = x_{i+1}, j > 1$ simply enumerates all of the possibilities for constructing $T$. The term $1 + d(x, y_2, |y|)$ handles the possibility that $y_1$ is generated by a duplicate operation that ends with $x_i$. In this case the tree $T_2$ is empty, so we only consider $T_1$. We add one to account for the production rule at the root of $T$ which is not part of $T_1$. This is illustrated in Fig. 8.

Conclusion
We have shown how to generalize duplication distance to derive phylogenetic relationships between human segmental duplications. We plan to apply the generalized distances introduced here to the same data to determine if these richer computational models yield new biological insights.

Acknowledgements
SM was supported by NSF Grant CCF-0635089. BJR is supported by a Career Award at the Scientific Interface from the Burroughs Wellcome Fund and by funding from the ADVANCE Program at Brown University, under NSF Grant No. 0548311.

Author details
1Department of Computer Science, Brown University, Providence, RI 02912, USA. 2Center for Computational Molecular Biology, Brown University, Providence, RI 02912, USA.

Authors’ contributions
CLK, SM, and BJR all designed and analyzed the algorithms and drafted the manuscript. All authors read and approved the final manuscript.

Competing interests
The authors declare that they have no competing interests.

Received: 11 August 2009
Accepted: 4 January 2010 Published: 4 January 2010

References
1. Sankoff D, Leduc G, Antoine N, Paquin B, Lang B, Cedergren R: Gene Order Comparisons for Phylogenetic Inference: Evolution of the Mitochondrial Genome. Proc Natl Acad Sci USA 1992, 89(14):6575-6579.
2. Pevzner P: Computational molecular biology: an algorithmic approach. Cambridge, Mass.: MIT Press 2000.
3. Chen X, Zheng J, Fu Z, Nan P, Zhong Y, Lonardi S, Jiang T: Assignment of Orthologous Genes via Genome Rearrangement. IEEE/ACM Trans Comp Biol Bioinformatics 2005, 2(4):302-315.
4. Marriott M, Svensson KM, Moret BME: Genomic Distances Under Deletions and Insertions. TCS 2004, 325(3):347-360.
5. El-Mabrouk N: Genome Rearrangement by Reversals and Insertions/Deletions of Contiguous Segments. Proc 11th Ann Symp Combin Pattern Matching (ICPM00) Berlin: Springer-Verlag 2000, 1848:222-234.
6. Zhang Y, Song G, Vinar T, Green ED, Siepel A, Miller W: Reconstructing the Evolutionary History of Complex Human Gene Clusters. Proc 12th Int'l Conf on Research in Computational Molecular Biology (RECOMB) 2008, 29-49.
7. Ma J, Ratan A, Raney BJ, Suh BB, Zhang L, Miller W, Haussler D: DUPCAR: Reconstructing Contiguous Ancestral Regions with Duplications. Journal of Computational Biology 2008, 15(8):1007-1027.
8. Bertrand D, Lajoie M, El-Mabrouk N: Inferring Ancestral Gene Orders for a Family of Tandemly Arrayed Genes. J Comp Biol 2008, 15(8):1063-1077.
9. Chaudhuri K, Chen K, Mihalcsu R, Rao S: On the Tandem Duplication- Random Loss Model of Genome Rearrangement. Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA) New York, NY, USA: ACM 2006, 564-570.
10. El-Mabrouk N, Gascuel O, Lefranc MP: Reconstructing the Duplication History of Tandemly Repeated Genes. Mol Biol Evol 2002, 19(3):278-288.
11. Lajoie M, Bertrand D, El-Mabrouk N, Gascuel O: Duplication and Inversion History of a Tandemly Repeated Genes Family. J Comp Bio 2007, 14(4):462-478.
12. El-Mabrouk N, Sankoff D: The Reconstruction of Doubled Genomes. SIAM J Comput 2003, 32(3):754-792.
13. Alekseyev MA, Pevzner PA: Whole Genome Duplications and Contracted Breakpoint Graphs. SODA 2007, 36(8):1748-1763.
14. Bailey J, Eichler EE: Primate Segmental Duplications: Crucibles of Evolution, Diversity and Disease. Nat Rev Genet 2006, 7:552-564.
15. Jiang Z, Tang H, Ventura M, Cardone MF, Marques-Bonet T, She X, Pevzner PA, Eichler EE: Ancestral reconstruction of segmental duplications reveals punctuated cores of human genome evolution. Nature Genetics 2007, 39:1361-1368.
16. Johnson M, Cheng Z, Morrison V, Scherer S, Ventura M, Gibbs R, Green E, Eichler E: Recurrent duplication-driven transposition of DNA during hominoid evolution. Proc Natl Acad Sci USA 2006, 103:17626-17631.

17. Kahn CL, Raphael BJ: Analysis of Segmental Duplications via Duplication Distance. Bioinformatics 2008, 24:i133-i38.

18. Kahn CL, Raphael BJ: A Parsimony Approach to Analysis of Human Segmental Duplications. Pacific Symposium on Bioinformatics 2009, 126-137.

doi:10.1186/1748-7188-5-11

Cite this article as: Kahn et al. Efficient algorithms for analyzing segmental duplications with deletions and inversions in genomes. Algorithms for Molecular Biology 2010 5:11.