V. Kůrková et al. (Eds.): ITAT 2014 with selected papers from Znalosti 2014, CEUR Workshop Proceedings Vol. 1214, pp. 46–53
http://ceur-ws.org/Vol-1214, Series ISSN 1613-0073, © 2014 A. Herencsár B. Brejová

An Improved Algorithm for Ancestral Gene Order Reconstruction

Albert Herencsár, Broňa Brejová

Faculty of Mathematics, Physics, and Informatics,
Comenius University, Mlynská Dolina, 842 48 Bratislava, Slovakia

Abstract: Genome rearrangements are large-scale mutations that change the order and orientation of genes in genomes. In the small phylogeny problem, we are given gene orders in several current species and a phylogenetic tree representing their evolutionary history. Our goal is to reconstruct gene orders in the ancestral species, while minimizing the overall number of rearrangement operations that had to occur during the evolution.

The small phylogeny problem is NP-hard for most genome rearrangement models. We have designed a heuristic method for solving this problem, building on ideas from an earlier tool PIVO by Kováč et al 2011. Our tool, PIVO2, contains several improvements, including randomization to select among potentially many equally good candidates in a hill-climbing search and a more efficient calculation of distances between gene orders. Using PIVO2, we were able to discover better histories for two biological data sets previously discussed in research literature. The software can be found at `http://compbio.fmph.uniba.sk/pivo/`.

1 Introduction

In evolution, genome rearrangements are rare genomic events. They are large-scale mutations that change the order and orientation of genes in genomes. The goal of our work is to reconstruct likely gene orders in ancestral extinct species given gene orders in present-day genomes. Such history reconstruction can help biologists to study evolutionary processes shaping genomes of living organisms.

In particular, we study the small phylogeny problem. On input, we have a phylogenetic tree describing an evolutionary history of a set of species. The leaves of the tree are the extant (current) species, and the internal nodes are their ancestors. We also know the order of genes in the genomes of the extant species. The task is to reconstruct the gene orders of the ancestors, while minimizing the overall number of rearrangement operations that had to occur during the evolution (Fig. 1).

The exact definition of the problem depends on the considered model of genome rearrangement. We use the breakpoint model (Tannier et al., 2009) and the double-cut-and-join (DCJ) model (Bergeron et al., 2006), which we describe in detail in the next section.

The small phylogeny problem is NP-hard for most genome rearrangement models, and it is in practice addressed by heuristic algorithms (Sankoff et al., 1976; Moret et al., 2001; Adam and Sankoff, 2008; Kováč et al., 2011). In this paper, we describe a series of improvements to a recent software called PIVO by Kováč et al. (2011). Our new implementation, PIVO2, runs faster and is able to find histories closer to the optimum, as we demonstrate in a series of experiments on both real and simulated data.

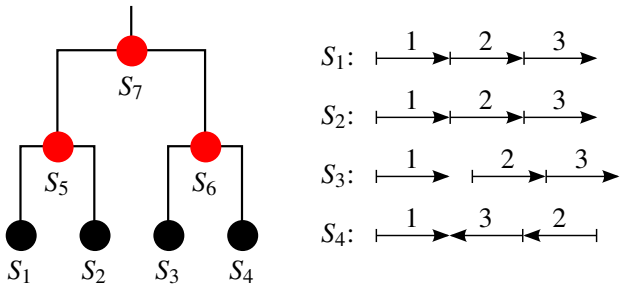

Figure 1: An example of the small phylogeny problem. We are given a tree as well as the gene orders in the present day species $S_1,\ldots,S_4$. The task is to infer the gene orders in ancestors S_5, S_6, S_7.

2 Background and Related Work

Genes, chromosomes and genomes. To study genome rearrangements, genomes are typically represented as sequences of markers called *genes*. A gene is thus in this context an identifier of a certain region in the genome. Genes have an orientation, going from left to right or from right to left. A sequence of genes is called a *chromosome*. We will consider both *linear chromosomes*, which have two ends called *telomeres*, and *circular chromosomes*, which do not have a distinct start or end. A *genome* is a set of chromosomes. In our work, we will consider a set of genomes over a set of genes $\{1,2,\ldots,g\}$ such that each genome contains exactly one copy of each gene.

To represent genomes more formally, we will call the two ends (*extremities*) of a gene its *head* and *tail* and denote them for gene a as $a+$ and $a-$ respectively. A pair of extremities located next to each other in a genome is called an *adjacency*. Two consecutive genes do not need to have the same orientation, and thus an adjacency between genes a and b can be of four different types: $\{a+,b-\},\{a+,b+\},\{a-,b-\},\{a-,b+\}$. If an extremity is not adjacent to any other gene, it is called a telomere. We represent it by a singleton set $\{a-\}$ or $\{a+\}$. A genome is then a set of adjacencies and telomeres in

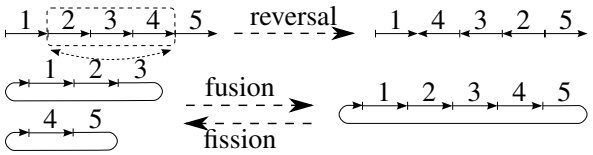

Figure 2: Examples of three rearrangement operations: reversal, chromosome fusion and chromosome fission

which the tail and head of every gene appear in exactly one adjacency or telomere.

Models of genome rearrangement. Genome rearrangements are large-scale mutations that move one or several genes to a different position or change the number or character of the chromosomes (Figure 2). The most common rearrangement is a reversal (or inversion), which happens when a chromosome breaks at two points and the middle part is joined back in the opposite direction.

In literature, several genome models were developed (Fertin et al., 2009), which allow us to measure distance between two genomes by counting the minimum number of rearrangements needed to transform one genome to another.

In this work, we concentrate on the double cut and join model introduced by Yancopoulos et al. (2005) and revised by Bergeron et al. (2006). This model allows a single general rearrangement operation called *double cut and join* (DCJ). Informally, we break chromosomes at at most two positions and rejoin them in a different way. The DCJ operation is explained more formally in the following definition.

Definition 1. *The double cut and join operation acts on two adjacencies or telomeres A_1 and A_2 in one of the following three ways:*

- *If $A_1 = \{p,q\}$ and $A_2 = \{r,s\}$, they are replaced by adjacencies $\{p,r\}$ and $\{q,s\}$ or by adjacencies $\{p,s\}$ and $\{q,r\}$.*
- *If $A_1 = \{p,q\}$ and $A_2 = \{r\}$ (A_2 is a telomere), they are replaced by $\{p,r\}$ and $\{q\}$ or by $\{q,r\}$ and $\{p\}$.*
- *If $A_1 = \{p\}$ and $A_2 = \{q\}$ (both are telomeres), they are replaced by adjacency $\{p,q\}$.*

In addition, as the inverse of the third case, an adjacency $\{p,q\}$ can be replaced by two telomeres $\{p\}$ and $\{q\}$.

Using DCJ operations, we can simulate all the common genome rearrangement operations, including reversal, chromosome fusion and fission, as well as circularisation and linearisation of chromosomes.

The distance between two genomes π and σ is in the DCJ model defined as the minimal number of DCJ operations that transform π into σ. It can be efficiently computed using the *adjacency graph* $AG(\pi,\sigma)$ (Figure 3). This graph is a bipartite multigraph, with one partition for each genome. The vertices correspond to the adjacencies and telomeres of the two genomes. Every vertex $v \in \pi$ and $w \in \sigma$ is connected by $|v \cap w|$ edges. So, if v and w represent the same adjacency in both genomes, they are connected by two edges. If v and w have one common extremity, they are connected by a single edge, and if they have no overlap, they are not connected by any edge.

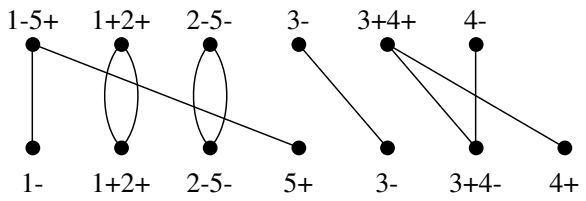

Figure 3: An example of the adjacency graph for genomes shown in Figure 4.

In the adjacency graph, every telomere has degree one and every adjacency has degree two. If the two genomes are equal, the graph consists of cycles of length two and paths of length one. Bergeron et al. (2006) have proved that the DCJ distance of genomes π and σ can be computed as follows. Let g be the number of genes, c the number of cycles in $AG(\pi,\sigma)$ and p_O the number of paths of odd length. Then the DCJ distance between π and σ is $dist_{DCJ}(\pi,\sigma) = g - (c + p_O/2)$. This quantity can be easily computed in $O(g)$ time.

We will also consider a simple breakpoint model (Sankoff and Blanchette, 1997), which does not describe particular rearrangement operations, only defines the distance between two genomes. Informally, this distance counts the number of breaks we need to introduce to the chromosomes of one genome so that by joining them in a different way we obtain the other genome. For multichromosomal genomes, it was defined by Tannier et al. (2009) as follows.

Definition 2 (Breakpoint distance)**.** *The breakpoint distance of genomes π and σ is:*

$$dist_{BP}(\pi,\sigma) = g - a(\pi,\sigma) - \frac{e(\pi,\sigma)}{2}$$

where g is the number of genes, $a(\pi,\sigma)$ is the number of common adjacencies in genomes π and σ, and $e(\pi,\sigma)$ is the number of common telomeres in genomes π and σ.

For example, the genomes in Figure 4 have five genes each, two common adjacencies and one common telomere. Their breakpoint distance is then $5 - 2 - 1/2 = 2.5$.

The Small Phylogeny Problem. In the small phylogeny problem, we are given a phylogenetic tree and the genomes of the extant species (see Figure 1). The task is to compute the genomes of ancestors, while minimizing the number of genome rearrangement operations required during evolution. We now define individual terms more formally.

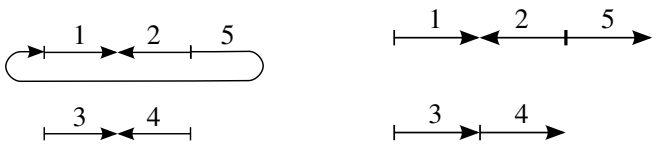

Figure 4: Breakpoint distance of the genomes is $5-2-1/2=2.5$

A *phylogenetic tree* is a binary tree $T=(V,E)$ rooted at node r, describing evolutionary relationships among a set of species. The leaves of the tree T are the extant species, and each internal node represents the most recent common ancestor of its children.

Let G be the set of all possible genomes on a particular set of genes. An *evolutionary history* h is a function that assigns a genome from G to every node of tree $T=(V,E)$: $h:V\to G$. The *score* of history h is

$$score(h,T)=\sum_{(u,v)\in E} dist(h(u),h(v)),$$

where *dist* is the distance measure of the chosen rearrangement model.

In the small phylogeny problem, we are given a phylogenetic tree $T=(V,E)$ with leaves $L\subset V$. We are also given a function $g:L\to G$, which assigns a genome to every leaf node. The task is to compute the evolutionary history h, which extends function g to cover all the nodes of the tree, while having the lowest possible score.

The *median problem* is a special case of the small phylogeny problem for three species. We are given three genomes and the goal is to compute the genome minimizing the sum of distances to the three input genomes. The median problem was shown to be NP-hard for most rearrangement models, including multichromosomal DCJ model (Tannier et al., 2009). One interesting exception is the breakpoint distance, where the median can be computed in polynomial time (Tannier et al., 2009). Being more general, the small phylogeny is also NP-hard in most models. The NP hardness of the small phylogeny problem for the breakpoint distance was shown by Kováč (2014).

Solving the Small Phylogeny Problem. Probably the most popular method for solving the small phylogeny problem is the Steinerization method (Sankoff et al., 1976). In this method, the algorithm iteratively improves the evolutionary history until a local optimum is found. In each iteration, the algorithm chooses an internal node v and calculates median π_M of the genomes in its three neighbouring nodes φ_a, φ_b, φ_c. We replace the genome inside node v with genome π_M, if the new tree has a lower score. When none of the internal nodes can be improved, the algorithm has found a local optimum.

Although the median problem is NP-hard in most rearrangement models, several solvers have been developed which in practice work in acceptable running time. The

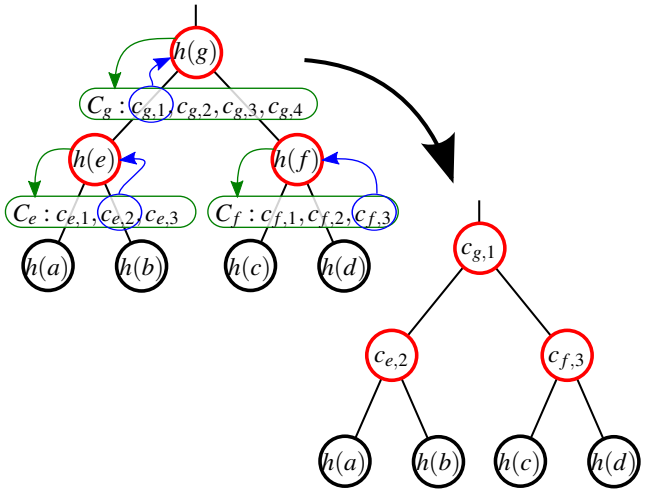

Figure 5: One iteration of the algorithm: For every internal node (red) the candidate sets are generated (green). Then the PIVO algorithm selects the best combination of the candidates (blue).

Steinerization method was used for example in BPAnalysis software for the breakpoint model (Blanchette et al., 1997) and in GRAPPA software (Moret et al., 2001) for both the breakpoint and reversal models. The Steinerization method for the DCJ model was implemented by Adam and Sankoff (2008). MGR (Bourque and Pevzner, 2002) is another small phylogeny solver for the reversal model. It uses a simple heuristic based on operations which bring genomes closer to other genomes in the tree.

Our work is however based on a different algorithm called PIVO, which encompasses and extends Steinerization approaches (Kováč et al., 2011). Next, we describe this algorithm in more details.

The PIVO Software. The PIVO (Phylogeny by IteratiVe Optimization) (Kováč et al., 2011) is a small phylogeny solver, which similarly to previous approaches uses a local search to iteratively improve initial history until a local optimum is found. While Steinerization methods update one internal node at a time, PIVO can potentially update values in all internal nodes together. This sometimes allows it to escape from situations where no internal node can be improved on its own.

In every iteration, PIVO generates a set of candidate genomes C_v for every internal node v of the tree. Although Kováč et al. (2011) describe several strategies for generating these candidates, the PIVO software typically uses as candidates for node v all genomes within DCJ distance one from the current genome $h(v)$.

The local search then computes a new history h' by selecting one genome from each candidate list C_v. The new history is chosen so that it has the smallest score out of all histories that can be produced by selecting values from candidate lists (see Figure 5).

The new history h' is computed by dynamic programming as follows. Let us denote the i-th candidate of node v

as $c_{v,i}$. Let $M[v,i]$ be the lowest possible score we can get for the subtree rooted at v, if $h'(v) = c_{v,i}$. For an internal node v with set of children X, we can compute value $M[v,i]$ using the following recurrence:

$$M[v,i] = \sum_{u \in X} \min_j \{M[u,j] + dist(c_{v,i}, c_{u,j})\}.$$

In the first phase of the dynamic programming, the values $M[v,i]$ are computed from leaves up (we set $M[v,0] = 0$ if v is a leaf). In the second phase, we choose candidates from root down. In particular, in the root, we can select any candidate with the lowest score. For each node u with parent v, we select the candidate $c_{u,j}$ for which the sum of $M[u,j] + dist(h'(v), c_{u,j})$ is minimal. If n is the number of species, g is the number of genes in every genome, and k is the number of candidates in every internal node, then the best candidates can be selected in $O(ngk^2)$ time (we suppose that the distance can be calculated in $O(g)$ time, which is the case for both DCJ and breakpoint distances). The PIVO algorithm is flexible, and it can work with several genome rearrangement models and distances.

In practice, the local search is run several times from different starting histories to increase the chance that a near-optimal or optimal history will be found.

3 PIVO2 Algorithm

We have implemented a new version of the PIVO algorithm, which we call PIVO2. Our main contributions are randomization of candidate selection and a more efficient algorithm for distance calculations in dynamic programming. Before describing these two topics in more detail, we briefly outline other changes made to the algorithm.

- The original PIVO software was written in Python; we have reimplemented it in Java to improve the running time.
- In PIVO, the local search starts from a history in which each internal node is initialized with a completely random genome. However, such histories are usually very far from the optimum. In PIVO2, we instead use the genomes of the extant species. In particular, every internal node v with children u and w is initialized randomly with $h(u)$ or $h(w)$.
- Kováč et al. (2011) discuss several strategies for generating candidate list C_v in every iteration of the algorithm. In PIVO2, we adapt the strategy, where the candidate list C_v contains all histories within DCJ distance 1 from $h(v)$. Optionally, we prune from this list all histories that increase the sum of distances to current histories in the three neighbours of v too much.

  In PIVO2, we also add all genomes $h(u)$ from the old history to the candidate list of every node v. This method allows genomes to "jump" from one node to another.

  Based on a proposal in Kováč et al. (2011), we optionally also add the genomes from the previous solutions to the candidate lists. The motivation is that we may get a better evolutionary history by combining different solutions.
- To avoid making the same decision in repeated searches, we have implemented a Tabu search metaheuristic (Glover, 1989a,b). The Tabu search tries to avoid getting always the same local optimum by keeping a tabu list of already visited configurations and penalizing configurations that are already on the list.

  PIVO2 keeps a tabu list T_v for every internal node v. This list contains genomes which were assigned to v in one of the previously considered histories.

  When the dynamic programming calculates the score of a candidate $c_{v,i}$, it adds a penalty $1/(|V|+1)$ to the score, if the candidate $c_{v,i}$ is in the tabu list T_v. Thus, if there are good candidates which were not present in previous solutions, the candidate $c_{v,i}$ is not selected. However, histories with a smaller rearrangement score will always be preferred, even if they incur the tabu penalty in every node.
- In the DCJ model, a genome can be a mix of linear and circular chromosomes. However, in the real world, the organisms usually have either one circular or several linear chromosomes. PIVO was designed to be flexible with respect to allowed chromosome architectures, because it was used to study the mitochondrial genomes of yeasts from a group containing both linear and circular genomes (Valach et al., 2011). The preferred type of genome architecture (circular or linear) can be prioritised by penalizing genomes which do not have the preferred structure. In PIVO2 we have implemented more refined penalties. In particular, if the minimum number of DCJ operations to transform a given genome $c_{v,i}$ to a preferred genome architecture is r, we add penalty $2r$ to $M[v,i]$ in the dynamic programming algorithm. With this penalty setting, the genomes in ancestral nodes mostly have the preferred architecture.

3.1 Randomization of Candidate Selection in the PIVO2 Algorithm

In the second phase of the dynamic programming algorithm for candidate selection, the original PIVO software always selects the first best candidate in each node. However, often there are multiple solutions with the optimal score. By choosing the first one, PIVO makes the same decision in repeated searches. In the PIVO2 algorithm, we have introduced randomized candidate selection, in which we choose a random history out of all combinations of candidates with the best score. The randomization ensures that each of these evolutionary histories will be picked

with equal probability. To do so, we need to extend the dynamic programming algorithm as outlined below.

In the first phase of the dynamic programming, the PIVO2 algorithm calculates the score $M[v,i]$ of each candidate $c_{v,i}$, but it also counts how many solutions with this score exist in the subtree rooted at v, if $c_{v,i}$ is selected. This value is denoted $Q[v,i]$. If node v has a child u, we first find the set $B[u,i]$ of candidates in C_u for which optimal value for $c_{v,i}$ is achieved. This set is computed as follows:

$$B[u,i] = \{c_{u,j} \in C_u \mid M[u,j] + dist(c_{u,j}, c_{v,i}) = M[v,i]\}$$

If node v has children u and w, we can then use sets $B[u,i]$ and $B[w,i]$ to compute $Q[v,i]$. First, the number of solutions is summed in each subtree separately, and the two sums are then multiplied.

$$Q[v,i] = \left(\sum_{j \in B[u,i]} Q[u,j]\right) \cdot \left(\sum_{j \in B[w,i]} Q[w,j]\right)$$

If node v is a leaf, we set $Q[v,0] = 1$.

In the second phase of the dynamic programming algorithm, we start at the root r and consider the set of candidates with minimal score:

$$B^* = \{c_{r,i} \mid c_{r,i} \in C_r, M[r,i] \textit{ is minimal}\}$$

The algorithm then selects candidate $c_{r,i} \in B^*$ with probability $Q[r,i]/Q[r]$, where $Q[r]$ is the total number of solutions with minimal score:

$$Q[r] = \sum_{j:\, c_{r,j} \in B^*} Q[r,j]$$

Choosing the candidate in other internal nodes is even easier, because it is already known, which candidate was selected in the parent node. If node v is a child of node p and if $c_{p,k}$ was selected in p, then the algorithm selects a candidate from $B[v,k]$. The probability of selecting candidate $c_{v,i} \in B[v,k]$ is $Q[v,i]/Q[v]$, where $Q[v]$ is the number of solutions with minimal score in the subtree rooted at v:

$$Q[v] = \sum_{j:\, c_{v,j} \in B[v,k]} Q[v,j]$$

This randomized selection can be implemented without increasing the overall time complexity of the dynamic programming algorithm.

3.2 Efficient Distance Computation in PIVO2

The PIVO algorithm computes distances between many pairs of candidates. When it calculates the scores $M[v,i]$ of candidates of node v (with children u and w), it computes distances between every pair of candidates from C_v and C_u, and from C_v and C_w, respectively.

The PIVO2 algorithm uses the "Neighbour" and the "Tree" strategies to generate candidate sets C_v. For every node v, the "Neighbour" method generates $\Theta(g^2)$ candidates, where g is the number of genes. These candidates are in DCJ distance 1 from the genome assigned to node v in the previous iteration. The "Tree" method generates $\Theta(|V|)$ candidates. Usually, the number of genes is larger than the tree size, and so $g^2 \gg |V|$. Therefore, the majority of distance calculations is made on pairs of genomes π_i and σ_j, such that all π_i have distance 1 from a genome π, and all σ_j have distance 1 from a genome σ (see Figure 6). We have designed a faster method of calculating the breakpoint and DCJ distances in such cases.

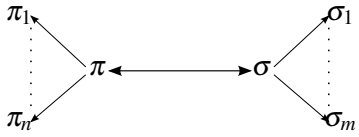

Figure 6: Efficient breakpoint distance calculation of a set versus a set.

We first describe the algorithm for efficient breakpoint distance calculation. To represent adjacencies in a genome, we use array G, which for every extremity stores its adjacent extremity (see Figure 7). We call G the genome array. Telomere extremity e is stored as $G[e] = e$. The head of gene a is represented at index $2a-1$ and its tail at index $2a-2$. Every adjacency is thus stored twice and telomeres are stored once.

Let $D_{\pi,\sigma}$ be the set of all indices where genome arrays of genomes π and σ differ. We will call this set the difference set between π and σ. The breakpoint distance of π and σ can be calculated as $|D_{\pi,\sigma}|/2$, and is thus related to the Hamming distance between the genome arrays representing the two genomes.

We will first describe a subroutine `update`(π,σ,σ',d,D), which gets genome arrays of genomes π, σ and σ', the breakpoint distance d between π and σ, and the difference set D between σ and σ'. It calculates the breakpoint distance between π and σ' in $O(|D|)$ time. We will initialize the distance with value d and update it for every index $i \in D$ as follows:

- If $\pi[i] = \sigma'[i]$, we subtract $1/2$ from the distance. This is because $\sigma[i] \neq \pi[i]$, and thus index i contributes value 1/2 to the distance between π and σ, but it does not contribute anything to the distance between π and σ'.
- Otherwise if $\pi[i] = \sigma[i]$, we add $1/2$ to the distance. This is because index i does not contribute anything to the distance between π and σ, but contributes 1/2 to the distance between π and σ'.
- Otherwise the distance does not change, because $\pi[i] \neq \sigma[i]$ and $\pi[i] \neq \sigma'[i]$, and thus index i contributes 1/2 to both distances.

A DCJ operation cuts at most two adjacencies or telomeres and creates at most two new adjacencies or telomeres

Index	0	1	2	3	4	5	6	7	8	9	10	11
Extremity	-1	+1	-2	+2	-3	+3	-4	+4	-5	+5	-6	+6
Adjacent extremity	-1	-2	+1	-3	+2	-4	+3	-5	+4	-6	+5	+6

Genome 1: 1 2 3 4 5 6 $

Index	0	1	2	3	4	5	6	7	8	9	10	11
Extremity	-1	+1	-2	+2	-3	+3	-4	+4	-5	+5	-6	+6
Adjacent extremity	+4	-2	+1	-3	+2	-4	+3	-1	-5	-6	+5	+6

Genome 2: 1 2 3 4 @ 5 6 $

Figure 7: Genome arrays of two genomes with breakpoint distance 1.5.

using the same extremities. Thus the difference set between the original and the new genomes will have size at most four. During "Neighbour" candidate generation, we know for every candidate which DCJ operation was performed, and we can save this additional information as a difference set without increasing the time complexity.

Let us now describe a method for calculating the breakpoint distance between every pair of genomes $\pi_i \in A$ and $\sigma_j \in B$, where every $\pi_i \in A$ is a genome in DCJ distance 1 from a genome π, and every $\sigma_j \in B$ is a genome in DCJ distance 1 from a genome σ (see Figure 6). We know for every genome $\pi_i \in A$ the difference set D_{π,π_i} and for every genome $\sigma_j \in B$ the difference set D_{σ,σ_j}. Our algorithm first calculates the distance d of π and σ in $O(g)$ time. The computation of distance between π_i and σ_j then proceeds in two update steps:

- $d' = \texttt{update}(\pi, \sigma, \sigma_j, d, D_{\sigma,\sigma_j})$
- $dist_{BP}(\pi_i, \sigma_j) = \texttt{update}(\sigma_j, \pi, \pi_i, d', D_{\pi,\pi_i})$

Our algorithm reduces the time complexity of computing distances between all pairs of genomes $\pi_i \in A$ and $\sigma_j \in B$ from $O(g \cdot |A| \cdot |B|)$ to $O(g + u \cdot |A| \cdot |B|)$, where u is the size of the difference sets. Note that in our case $u \leq 4$, and thus it can be considered a constant.

Similar, but a more complex method also works for DCJ distance calculation between all pairs of genomes $\pi_i \in A$ and $\sigma_j \in B$. Recall that to compute the DCJ distance, we represent the two genomes as an adjacency graph, in which each connected component is a cycle or a path. We can compute the DCJ distance if we know the number of cycles and paths of odd length in this graph.

For our algorithm, it is more convenient to divide each vertex containing two extremities into two vertices connected by an auxiliary edge. For example the vertex 1- 5+ in the top partition of Figure 3 is divided into vertices 1- and 5+ connected by an auxiliary edge. Regular edges connect the new vertex 1- to its counterpart 1- in the other genome and similarly for 5+. Each vertex then represents one extremity in one of the genomes and cycles and paths are simply lists of extremities. Note that auxiliary edges are not counted when distinguishing paths of odd and even lengths.

Instead of difference sets, we will now characterize the differences between a pair of genomes by a set of disconnect and connect operations. Each disconnect operation removes one auxiliary edge corresponding to a removed adjacency between two extremities, and each connect operation creates a new auxiliary edge, corresponding to a new adjacency. A DCJ operation corresponds to at most two disconnect and two connect operations.

We spend $O(g)$ time to compare genomes π and σ. We compute their adjacency graph, find its components and compute the DCJ distance. We can then compute the DCJ distance between $\pi_i \in A$ and $\sigma_j \in B$ in $O(o^3)$ time, where o is the size of the connect and disconnect sets between π and π_i and σ and σ_j respectively. To do so, we store the positions of individual extremities in the components of the original graph. We maintain each modified component of the graph after updates as a list of intervals from the original components (which are paths or cycles). By appropriately linking these structure together, we can process each update (connect or disconnect) in $O(o^2)$ time. Details are omitted due to space constraints and can be found in the thesis by Herencsár (2014). The overall running time to compute distances between all pairs is thus $O(g + o^3 \cdot |A| \cdot |B|)$.

4 Experiments

In this section, we present the results of an experimental comparison of PIVO2 to the original PIVO on real and simulated data sets.

Real data. We have first run PIVO2 on two real biological data sets previously considered in literature. The first data set consists of 13 chloroplast genomes from plants of the Campanulaceae family (Cosner et al., 2000), each genome consisting of a single circular chromosome. The reconstruction of ancestral genomes of these species under the DCJ model was studied in several earlier works, as outlined in Table 1. We consider two cases: when ancestors are restricted to have a single circular chromosome and when their genome architecture is unrestricted. In the unrestricted case, PIVO2 was able to achieve better results than previous approaches, including PIVO. In the restricted case, we match the score obtained by PIVO.

The second data set contains 16 mitochondrial genomes of pathogenic yeasts from the CTG clade of Hemias-

Table 1: Scores obtained by various algorithms for two real data sets. All values except for PIVO2 were obtained from literature.

Software	Genome architecture unichr.	restr.	any
Campanulaceae data set (13 genomes, 105 genes each)			
ABC (Adam and Sankoff, 2008)	64		59
GASTS (Xu and Moret, 2011)	63		-
PIVO (Kováč et al., 2011)	**59**		59
PIVO2	**59**		**56**
Hemiascomycetes data set (16 genomes, 25 genes each)			
PIVO (Kováč et al., 2011)		78	-
PIVO2		**77**	**75**

comycetes (Valach et al., 2011). As we have already discussed, these genomes have various architectures: some consist of a single linear chromosome, one consists of two linear chromosomes, and the rest consist of a single circular chromosome.

Due to this variability in the genome architecture of extant species, the ancestral genomes could have a single circular chromosome, or one or more linear chromosomes. The original PIVO algorithm found an evolutionary history with score 78 (Kováč et al., 2011), and the PIVO2 algorithm found a better evolutionary history with score 77. If we allow ancestral genomes with an arbitrary architecture (including a mixture of circular and linear chromosomes in the same ancestor), an evolutionary history with score 75 was found by the PIVO2 algorithm (no result was published for the original PIVO algorithm).

Results on simulated data. We have also tested our algorithm on simulated data and compared its speed and accuracy to the original PIVO software. To generate data, we have used the phylogenetic tree of the Campanulaceae. A random genome consisting of 25 genes was generated for the root, and random evolution consisting of DCJ operations was simulated along branches of the tree. No restrictions were applied on the karyotype, i.e. an arbitrary mix of linear and circular chromosomes was allowed. The extant genomes, which were produced by simulating random evolution, were used as the input for both algorithms.

Overall, we have generated 5 such test cases, which differed by the number of mutations allowed along each edge, ranging from 3 to 8. These histories have a relatively high number of events in a short genome, and as a result, it is often possible to find a lower-scoring history than the one actually generating the data. For each input, we have run both PIVO and PIVO2 local searches many times and recorded the local optimum from each run. For each input, the overall best history found by PIVO2 was better or the same as the one for PIVO. In addition, the distribution of local optima was shifted towards lower values for PIVO2, as illustrated for one test case in Figure 8.

On each input, PIVO2 performed on average more iterations of candidate selection than PIVO in each run. For example on the input considered in Figure 8, the average number of iterations for PIVO was 5.3 and for PIVO2 it was 8.6. However, this increase is amply justified by the fact, that the PIVO2 algorithm gives better results than the original PIVO.

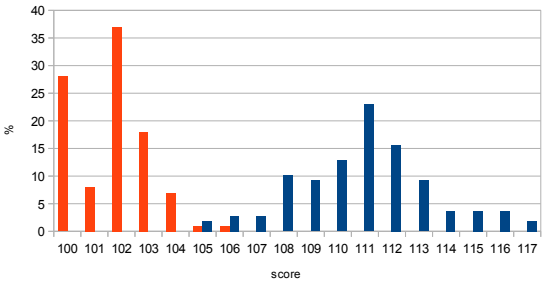

Figure 8: Histogram of scores of individual runs of local search in PIVO2 (red) and PIVO (blue) on a simulated test. The simulated history has score 103, the best result by PIVO2 has score 100 and the best result by PIVO2 has score 105.

Finally, we have compared the speed of individual candidate selection runs, to asses the impact of our faster distance calculation. The original PIVO algorithm was written in Python, and the PIVO2 algorithm was reimplemented in Java. Inherently, Python runs slower than Java, and therefore, it is difficult to compare the speed of the two implementations meaningfully. According to our measurements, PIVO2 (with efficient distance calculation mode switched off) is approximately 6-7 times faster than the original PIVO. As we will show next, PIVO2 runs even faster in the efficient distance calculation mode.

We have again created several simulated test cases, but in these tests we gradually increased the number of genes. The PIVO2 algorithm was run with the efficient distance calculation mode enabled or disabled, and we measured the average time needed to compute the distances between each pair of candidates of two neighbouring internal nodes. The results are shown in Figure 9. As we would expect, the speedup increases with the number of genes, because we have lowered the asymptotic complexity.

5 Conclusion

In this work, we have introduced several improvements to the PIVO algorithm for reconstruction of ancestral gene orders. The resulting software, PIVO2, was able to find histories with better score and has a potential to be useful for evolutionary studies on real biological data sets.

One possibility for further research is in the area of efficient distance computation. Our algorithm could be extended to sets of candidates which are likely to contain clusters of very similar genomes, but these clusters are not given explicitly in advance. The algorithm would have to

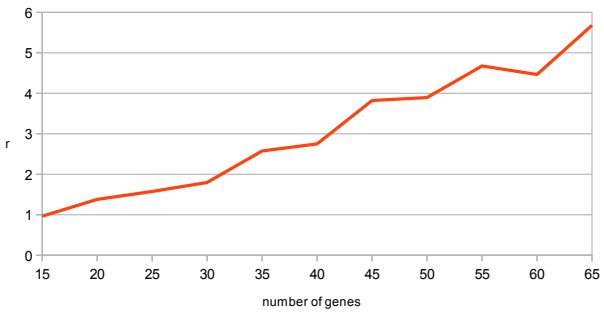

Figure 9: Ratio of the time needed for basic and fast distance calculation for one pair of nodes. As the number of genes in the genome increases, the speedup grows as well.

discover such clusters automatically, choose a representative of each cluster, compute distances between representatives and then adjust the distances for other members of the clusters.

Acknowledgments. This research was supported by VEGA grants 1/1085/12 and 1/0719/14.

References

Adam, Z. and Sankoff, D. (2008). The ABCs of MGR with DCJ. *Evolutionary Bioinformatics Online*, 4:69–74.

Bergeron, A., Mixtacki, J., and Stoye, J. (2006). A unifying view of genome rearrangements. In *Workhop on Algorithms in Bioinformatics (WABI)*, pages 163–173.

Blanchette, M., Bourque, G., and Sankoff, D. (1997). Breakpoint phylogenies. In *Workshop on Genome Informatics*, pages 25–34.

Bourque, G. and Pevzner, P. (2002). Genome-scale evolution: reconstructing gene orders in the ancestral species. *Genome Research*, 12(1):26–36.

Cosner, M., Jansen, R., Moret, B., Raubeson, L., Wang, L., Warnow, T., and Wyman, S. (2000). An empirical comparison of phylogenetic methods on chloroplast gene order data in Campanulaceae. In Sankoff, D. and Nadeau, J. H., editors, *Comparative Genomics*, pages 99–121. Springer.

Fertin, G., Labarre, A., Rusu, I., Tannier, E., and Vialette, S. (2009). *Combinatorics of genome rearrangements*. MIT Press.

Glover, F. (1989a). Tabu Search - Part I. *ORSA Journal on Computing*, 1:190–206.

Glover, F. (1989b). Tabu Search - Part II. *ORSA journal on Computing*, 2 1:4–32.

Herencsár, A. (2014). An improved algorithm for ancestral gene order reconstruction. Master's thesis, Comenius University in Bratislava.

Kováč, J. (2014). On the complexity of rearrangement problems under the breakpoint distance. *Journal of Computational Biology*, 21:1–15.

Kováč, J., Brejová, B., and Vinař, T. (2011). A practical algorithm for ancestral rearrangement reconstruction. In *Workshop on Algorithms in Bioinformatics (WABI)*, pages 163–174.

Moret, B., Wyman, S., Bader, D. A., Warnow, T., and M., Y. (2001). A new implementation and detailed study of breakpoint analysis. In *Pacific Symposium on Biocomputing*, pages 583–94.

Sankoff, D. and Blanchette, M. (1997). The median problem for breakpoints in comparative genomics. In *Conference on Computing and Combinatorics (COCOON)*, pages 251–263.

Sankoff, D., Cedergren, R. J., and Lapalme, G. (1976). Frequency of insertion-deletion, transversion, and transition in the evolution of 5S ribosomal RNA. *Journal of Molecular Evolution*, 7:133–149.

Tannier, E., Zheng, C., and Sankoff, D. (2009). Multichromosomal median and halving problems under different genomic distances. *BMC Bioinformatics*, 10:120.

Valach, M., Farkas, Z., Fricova, D., Kovac, J., Brejova, B., Vinar, T., Pfeiffer, I., Kucsera, J., Tomaska, L., Lang, B. F., and Nosek, J. (2011). Evolution of linear chromosomes and multipartite genomes in yeast mitochondria. *Nucleic Acids Research*, 39(10):4202–4219.

Xu, A. W. and Moret, B. M. E. (2011). GASTS: Parsimony scoring under rearrangements. In *Workshop on Algorithm in Bioinformatics (WABI)*, pages 351–363.

Yancopoulos, S., Attie, O., and Friedberg, R. (2005). Efficient sorting of genomic permutations by translocation, inversion and block interchange. *Bioinformatics*, 21:3340–3346.