Comenius University, Bratislava
Faculty of Mathematics, Physics and Informatics

An improved algorithm for ancestral gene order reconstruction

Master's Thesis

2014

Bc. Albert Herencsár

Comenius University, Bratislava
Faculty of Mathematics, Physics and Informatics

An improved algorithm for ancestral gene order reconstruction

Master's Thesis

Study Program:	Computer Science
Branch of Study:	2508 Computer Science
Department:	Department of Computer Science
Supervisor:	Mgr. Bronislava Brejová, PhD.

Bratislava, 2014

Bc. Albert Herencsár

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Albert Herencsár
Študijný program: informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: 9.2.1. informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: An Improved Algorithm for Ancestral Gene Order Reconstruction
Vylepšenie algoritmu pre rekonštrukciu ancestrálnych poradí génov

Cieľ: Cieľom práce je vyvinúť lepšiu metódu na rekonštrukciu ancestrálnych poradí génov pre danú sadu poradí genóv v niekoľkých súčasných organizmoch. Táto úloha sa dá sformulovať ako optimalizačný problém, ktorý je ale NP úplný a zvyčajne sa v praxi rieši heuristickými algoritmami. Cieľom je zlepšiť tieto heuristické algoritmy, čo môže viesť k nájdeniu histórií bližších k optimálnej.

Vedúci: Mgr. Bronislava Brejová, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: doc. RNDr. Daniel Olejár, PhD.

Spôsob sprístupnenia elektronickej verzie práce:
bez obmedzenia

Dátum zadania: 23.09.2013

Dátum schválenia: 30.09.2013

prof. RNDr. Branislav Rovan, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Bc. Albert Herencsár
Study programme:	Computer Science (Single degree study, master II. deg., full time form)
Field of Study:	9.2.1. Computer Science, Informatics
Type of Thesis:	Diploma Thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: An Improved Algorithm for Ancestral Gene Order Reconstruction

Aim: The goal of the thesis is to develop a better method for reconstructing ancestral gene orders given gene orders in several extant species. This task can be formulated as an optimization problem, which is NP-complete and usually adressed by heuristic algorithms. The goal is to improve these heuristic algorithms, which may lead to finding histories closer to optimum.

Supervisor:	Mgr. Bronislava Brejová, PhD.
Department:	FMFI.KI - Department of Computer Science
Head of department:	doc. RNDr. Daniel Olejár, PhD.
Assigned:	23.09.2013

Approved: 30.09.2013 prof. RNDr. Branislav Rovan, PhD.
Guarantor of Study Programme

.....
Student

.....
Supervisor

I hereby declare that I wrote this thesis by myself, only with the help of the referenced literature, under the careful supervision of my thesis advisor.

. . . . . .

I am deeply grateful to my supervisor Mgr. Broňa Brejová, PhD. for her invaluable help and guidance.

Bc. Albert Herencsár

Abstrakt

Genómové preusporiadania sú rozsiahle mutácie, ktoré menia poradie a orientáciu génov v genómoch. Rekonštrukcia ancestrálnych poradí génov fylogenetického stromu je známa pod pojmom malý fylogenetický problém. Na vstupe máme fylogenetický strom, čo je binárny strom popisujúci evolučnú históriu, a taktiež poradia génov súčasných druhov. Úlohou je rekonštruovať ancestrálne poradia génov a súčasne minimalizovať celkový počet preusporiadavacích operácií, ktoré museli nastať počas evolúcie. Malý fylogenetický problém je NP-ťažký pre väčšinu modelov genómového preusporiadania. Na riešenie malého fylogenetického problému vyvinul Kováč a kol. univerzálnu metódu (PIVO), ktorá používa procedúru iteratívnej lokálnej optimalizácie [1]. Vytvorili sme nový algoritmus, nazvaný PIVO2, ktorý vylepšuje pôvodný algoritmus PIVO v rôznych oblastiach. K nim patrí vylepšená inicializácia, generovanie kandidátov a výber kandidátov. Algoritmus PIVO2 obsahuje niekoľko pridaných voliteľných rozšírení. V PIVO2 sme taktiež vylepšili rýchlosť pôvodného algoritmu a to vytvorením novej a efektívnejšej metódy výpočtu genómovej vzdialenosti. V tejto práci tiež prezentujeme praktické experimenty a porovnania pôvodného algoritmu PIVO a algoritmu PIVO2. Tieto experimenty ukazujú, že algoritmus PIVO2 je skutočne lepší než pôvodný algoritmus PIVO v nájdení evolučných historií s nižším skóre. Naviac, na nájdenie histórie s dobrým skóre algoritmus PIVO2 potrebuje podstatne nižší počet behov. Experimenty takisto potvrdzujú, že nová metóda výpočtu genómovej vzdialenosti skutočne zvyšuje rýchlosť výpočtu.

Kľúčové slová: malý fylogenetický problém, ancestrálne poradie génov, genómové preusporiadanie, breakpoint vzdialenosť, DCJ vzdialenosť

Abstract

Genome rearrangements are large scale mutations, which change the order and orientation of genes in genomes. Reconstruction of ancestral gene orders on a phylogeny is known as the small phylogeny problem. On input, we have a phylogenetic tree, which is a binary tree describing the evolutionary history, as well as gene orders in present day species. The task is to reconstruct the gene orders of ancestors, 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. A universal heuristic method (PIVO) was developed by Kováč et al. for solving the small phylogeny problem using an iterative local optimization procedure [1]. We created a new algorithm, called PIVO2, which improves the original PIVO algorithm in various areas. These include improved initialization, candidate generation, and candidate selection. The PIVO2 algorithm contains several added optional extensions. In PIVO2, we also improved the speed of the original PIVO algorithm by creating a new and more efficient method for genome distance calculation. In this thesis, we also present practical experiments and comparisons of the original PIVO and the PIVO2 algorithms. The experiments show that the PIVO2 algorithm is indeed better than the original PIVO algorithm in finding evolutionary histories with lower score. Moreover, to find a history with a good score, the PIVO2 algorithm requires a significantly lower number of runs. The experiments also confirm that the new method of genome distance calculation really increases the computational speed.

Key words: small phylogeny problem, ancestral gene order, genome rearrangement, breakpoint distance, DCJ distance

Contents

List of Figures

List of Tables

List of Algorithms

Introduction

In evolution, genome rearrangements are rare genomic events. They are large scale mutations, which change the order and orientation of genes in genomes. Genome rearrangements are thought to play a significant role in speciation [3]. Various branches of biology require computational tools which contribute to understanding of evolution of genome organisation. Using algorithmic approaches in the analysis of rearrangement data helps to resolve difficult questions e.g. about branching patterns in evolution.

Reconstruction of ancestral gene orders on a phylogeny is known as the small phylogeny problem. On input, we have a phylogenetic tree, which is a binary tree describing an evolutionary history. The leaves of the tree are the extant species, and the internal nodes are their ancestors. We also know the order of the genes in the genomes of the extant species. The task is to reconstruct the gene orders of ancestors, while minimizing the overall number of rearrangement operations that had to occur during the evolution.

Many genome rearrangement models have been developed [4]. These models enable the measurement of the distance of two genomes, which is defined as the minimal number of rearrangement operations needed to transform one genome into the other. In this thesis, we use the Breakpoint model [5] and the DCJ model [6].

The small phylogeny problem is NP-hard for most genome rearrangement models. A popular heuristic method for solving this problem uses the steinerization method [7].

To handle multiple chromosome structures and provide more precise solutions than the software based on the steinerization method, a universal heuristic method was developed by Kováč et al. for solving the small phylogeny problem using an iterative local optimization procedure (PIVO) [1]. In every iteration, a set of candidate genomes is generated for every ancestral node of the phylogenetic tree. Then, by dynamic programming, the genomes in the phylogenetic tree are replaced with the best combination of the candidates.

In this thesis, we improved the PIVO algorithm in various areas. These include

improved initialization, candidate generation, and candidate selection. Several optional extensions were added to the algorithm. We also improved the speed of the PIVO algorithm by creating a new and more efficient distance calculation method. We named the reimplemented and improved version of the original algorithm as PIVO2.

We present practical experiments and comparisons, which evaluate the implemented improvements and confirm their benefit.

Chapter 1

Background and Related Work

1.1 Basic Biological Terms

The **Genome** is all the genetic material of an organism. It is a set of instructions for creating, keeping alive, and reproducing an organism. In most living things, the genome is made of a chemical called DNA.

The **DNA** is a long molecule made up of smaller building blocks called **nucleotides**. These nucleotides are: adenine, cytosine, guanine, and thymine, often abbreviated by letters A, C, G, T. The DNA molecule has a double helix structure.

The Genome of an organism is packaged into smaller chunks called **chromosomes**. The chromosomes can have linear or circular structure.

A **gene** is a distinct sequence of nucleotides forming part of a chromosome. Genes describe the characteristics of an organism.

The DNA of an organism can change, these changes are called **mutations**. Mutations range in size from a single nucleotide to a large segment of a chromosome.

We can imagine the genome as a book, in which chromosomes are the chapters. The sentences in the book are the genes, and the nucleotides are the letters of the book.

1.2 Genome Rearrangements

If we compare genomes of various species, we can often find very similar segments of DNA. However, these segments are usually ordered differently, have different orientation and they can be inside different chromosomes. For example, humans and mice have similar genomes. In figure 1.1, we can see a chromosome-level comparison of the human and mouse genomes [2].

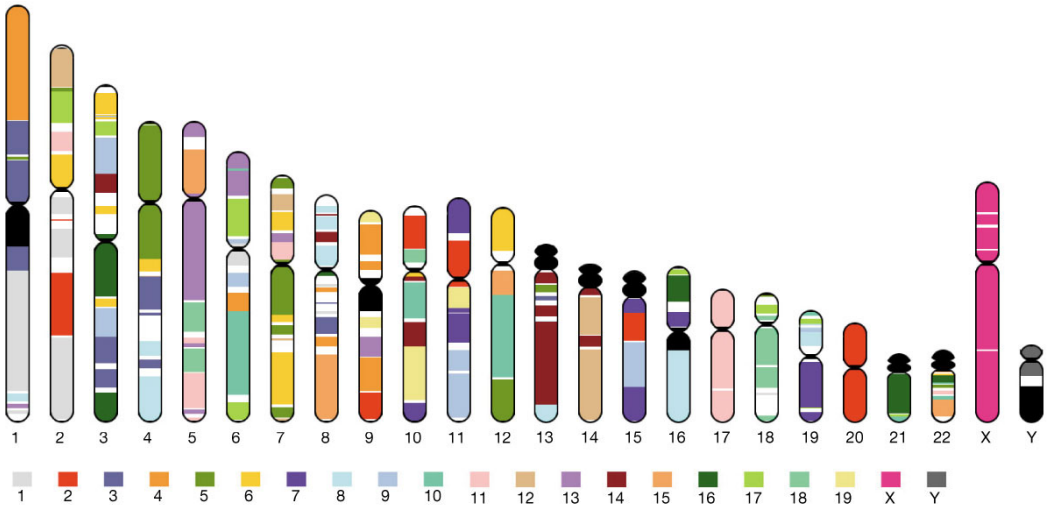


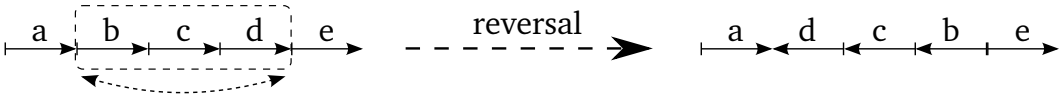
Figure 1.1: Human chromosomes, with segments containing at least two genes whose order is conserved in the mouse genome as color blocks. Each color corresponds to a particular mouse chromosome. Source [2].

This similarity between genomes exists because both species had a common ancestor in the past. During evolution, large-scale mutations rearranging the genome occurred, and the order of genes was changed.

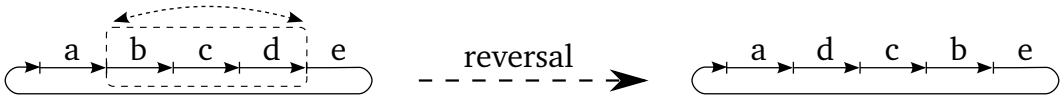
Genome rearrangements are evolutionary events that change the order and the orientation of genes in genomes. Studies suggest that genome rearrangements play a significant role in speciation [3].

Genome rearrangements occur less often than nucleotide mutations. Due to their low rate, gene order data are a valuable source of information about early evolution [8].

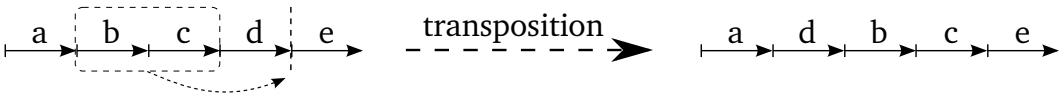
Genome rearrangements include inversions, transpositions, chromosome fusions and fissions (see figure 1.2). Reversal (or inversion) is the most common rearrangement (figure 1.2a). It happens when the double helix breaks at two points and the middle part is joined back in the opposite direction. Reversal can happen in circular chromosomes too (figure 1.2b). Transposition (figure 1.2c) happens when the chromosome breaks at 3 places and the pieces are joined in a wrong order. If a genome consists of several chromosomes and two different linear chromosomes break, we can end up with a translocation (figure 1.2d). Fusion and fission may change the number chromosomes (figure 1.2e and figure 1.2f), or they may change a linear chromosome to a circular one and vice versa (figure 1.2g). By fusion and fission, a circular segment may be excised or incorporated into a linear chromosome (figure 1.2h).



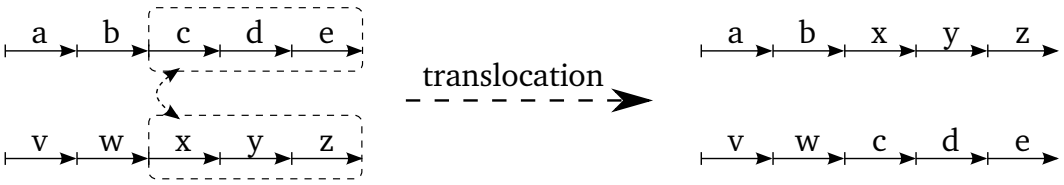
(a) Reversal



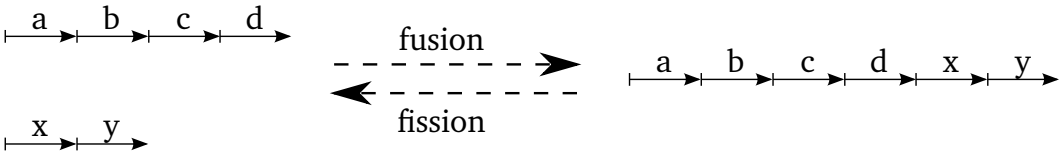
(b) Reversal in a circular chromosome



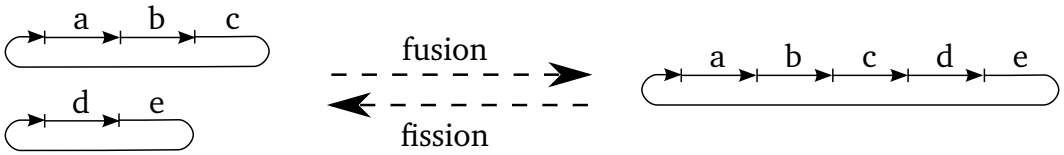
(c) Transposition



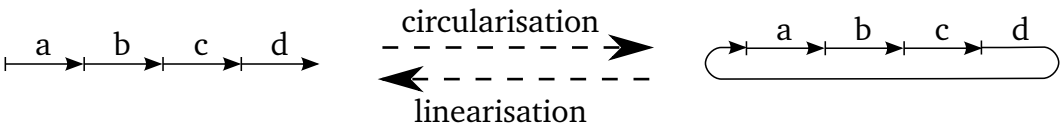
(d) Translocation



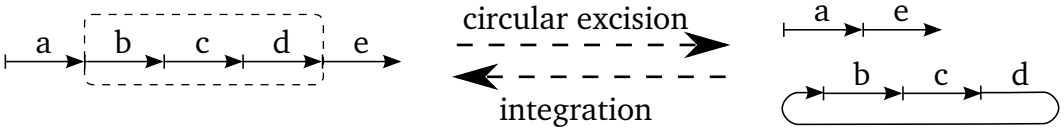
(e) Fusion and fission of linear chromosomes



(f) Fusion and fission of circular chromosomes



(g) Circularisation and linearisation



(h) Circular excision and the reverse process

Figure 1.2: Types of rearrangements. Each gene is shown as an arrow to indicate its orientation.

1.3 Genome Representation

A gene is an oriented sequence of nucleotides. It starts with a tail, and it ends with a head. The tail and head are the extremities of the gene. We number the genes with positive numbers, or sometimes, we mark them with letters. The tail of a gene, marked as a, is denoted as $a-$ and the head as $a+$. In figure 1.3, we can see a gene with its tail and head.

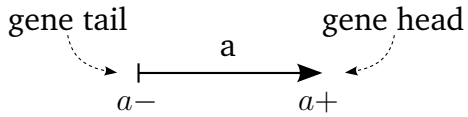


Figure 1.3: Gene with its tail and head

If extremities p and q are next to each other in a genome, they form an adjacency $\{p, q\}$. Two consecutive genes do not need to have the same orientation. The adjacency between two consecutive genes a and b, depending on their respective orientation, can be of four different types (see figure 1.4): $\{a+, b-\}, \{a+, b+\}, \{a-, b-\}, \{a-, b+\}$. If an extremity is not adjacent to any other gene, the extremity is called a telomere. We represent it by a singleton set $\{a-\}$ or $\{a+\}$.

Figure 1.4: Possible orientations of two neighbouring genes

A genome is a set of adjacencies and telomeres in which the tail and head of every gene appear in exactly one adjacency or telomere.

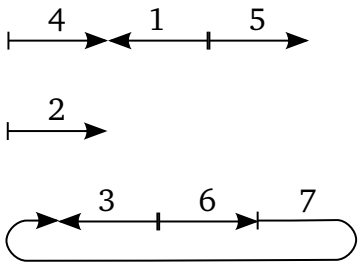
We can reconstruct the chromosomes of a genome by creating the genome graph (figure 1.5c). The extremities are represented by vertices. The tail and the head of each gene is joined by an edge. Adjacent extremities are then connected together (green edges in figure 1.5c). A genome graph has vertices of degree one or two, and so the graph is a set of disjoint paths and cycles. These paths and cycles are the linear and circular chromosomes of the genome. Linear chromosomes start and end with telomeres.

Sometimes it can be useful to add special telomere vertices into the genome graph. For each telomere x, we add a special vertex T_x and connect them with a green edge (figure 1.5d). We say that $\{x, T_x\}$ is a telometric adjacency.

Chromosomes can be represented by lists of gene labels (figure 1.5e). These lists are obtained by choosing a telomere in a linear chromosome or an arbitrary gene in a circular chromosome, and then by enumerating the gene labels along the component. We use positive signs to indicate genes that are read from tail to head and negative

signs to indicate genes that are read from head to tail. For linear chromosomes, we put a $ character at the end of the list. Circular chromosomes end with an @ character. Positive signs may be omitted where convenient.

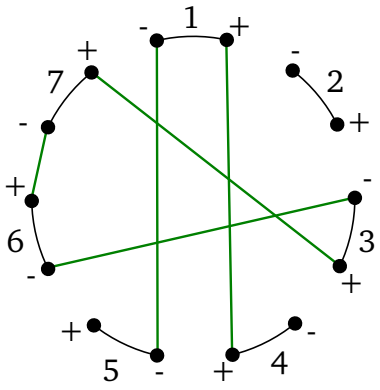
In figure 1.5a, we can see a genome made up of 7 genes, which consists of two linear and of one circular chromosome.



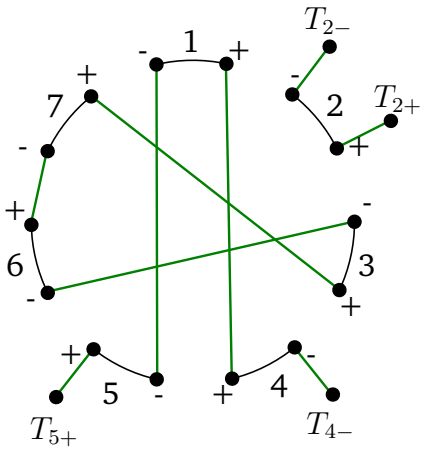
(a) Genome

$\{4-\}, \{4+, 1+\}, \{1-, 5-\}, \{5+\},$
$\{2-\}, \{2+\},$
$\{3+, 7+\}, \{3-, 6-\}, \{6+, 7-\}$

(b) Adjacencies and telomeres



(c) Genome graph



(d) Genome graph with special telomere vertices

4 -1 5 $
2 $
-3 6 7 @

(e) List of gene labels

Figure 1.5: Different ways of representing a genome

1.4 Distances Between Genomes

Working with genome rearrangements in a biologically plausible way can be quite complex. For this reason, many genome rearrangement models were developed. These models differ, for example, in the allowed karyotypes (allowed chromosome structure),

in the subset of allowed rearrangement operations, or in taking into account gene orientation.

We can measure the distance of two genomes in every genome rearrangement model.

Definition 1 (Distance of genomes)**.** *The distance $dist(\pi, \sigma)$ between two genomes π and σ is the minimal number of rearrangement operations needed to transform π into σ (or vice versa), while adhering to the restrictions of the given model. (Note: We consider only genomes with the same set of genes.)*

The distance between two genomes can be used, for example, to determine the number of rearrangement mutations which occurred between an organism and its ancestor during evolution.

In this thesis, we use the Breakpoint model and the DCJ model.

1.4.1 Breakpoint Distance

The Breakpoint model does not define which rearrangement operations are allowed, it only defines the distance between genomes. This distance measure is probably one of the simplest. It was introduced by Sankoff and Blanchette [9]. The breakpoint distance has been well-studied for permutations, i.e., unichromosomal genomes.

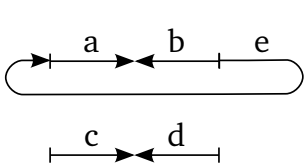
In this thesis, we work with more general, multichromosomal genomes, and so the definition in [9] is not sufficient. Instead, we calculate the breakpoint distance as it was described by Tannier [5]. The breakpoint distance between two genomes depends on the number of common adjacencies and the number of common telomeres.

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

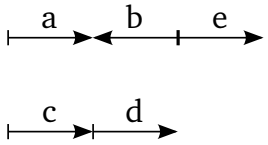
$$dist_{BP}(\pi, \sigma) = g - a(\pi, \sigma) - \frac{e(\pi, \sigma)}{2}$$

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 1.6 have 5 genes each, 2 common adjacencies and 1 common telomere. So, their breakpoint distance is $5 - 2 - 1/2 = 2.5$.



(a) Genome 1: $\{a-, e+\}$, $\{a+, b+\}$, $\{b-, e-\}$, $\{c-\}$, $\{c+, d+\}$, $\{d-\}$



(b) Genome 2: $\{a-\}$, $\{a+, b+\}$, $\{b-, e-\}$, $\{e+\}$, $\{c-\}$, $\{c+, d-\}$, $\{d+\}$

Figure 1.6: Breakpoint distance of the genomes is $5 - 2 - 1/2 = 2.5$

1.4.2 Double Cut and Join Distance

The Double Cut and Join (DCJ) model was introduced by Yancopoulos et al. [10] and revised by Bergeron et al. [6].

In this model, there are no restrictions on the karyotype, and so there can be arbitrarily many linear and circular chromosomes in the genome simultaneously. The genome rearrangement operations are modelled by a single operation, which is called the double cut and join operation.

The Double Cut and Join (DCJ) Operation

If we imagine our genome as a genome graph with special telometric vertices, the DCJ operation can be described as follows (see figure 1.5d): We break two green edges at most and then we rejoin the created endpoints in a different way. The DCJ operation is explained more formally in the following definition:

Definition 3 (The double cut and join operation)**.** *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 an 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. A reversal can be done by cutting the interval boundaries and joining the

created endpoints as in figure 1.7a. Translocation can be done by cutting and joining the adjacencies of two different linear chromosomes (figure 1.7b). By a DCJ operation, we can do fusion and fission on linear or circular chromosomes (figure 1.7c and figure 1.7d). Similarly, circularisation/linearisation, circular excision and integration can be done. Translocation is possible in two DCJ operations only: by doing a circular excision followed by an integration.

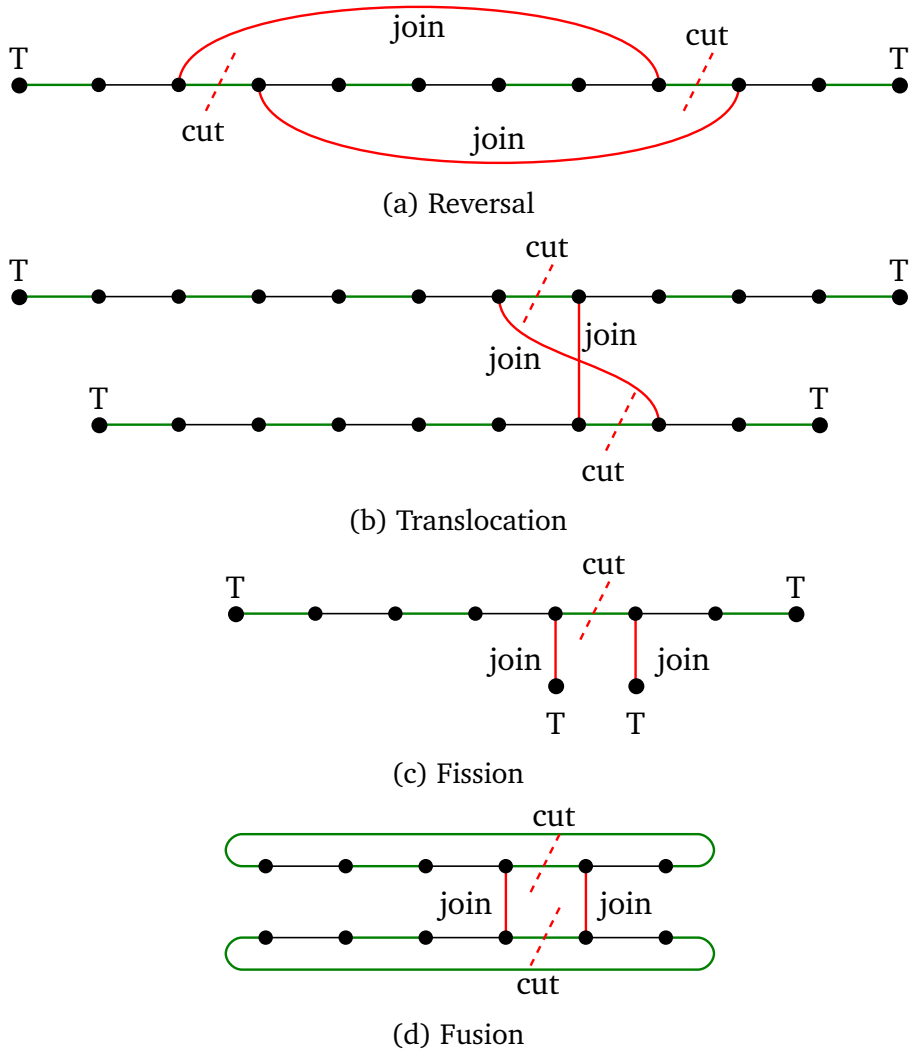


Figure 1.7: Simulating some genome rearrangements using DCJ operations

The Double Cut and Join (DCJ) Distance

The distance of two genomes π and σ, i.e. the minimal number of DCJ operations that transform π into σ, can be computed using the adjacency graph $AG(\pi, \sigma)$.

Definition 4 (Adjacency graph $AG(\pi, \sigma)$)**.** *The adjacency graph is a bipartite multigraph, in which the vertices of the graph are the adjacencies and telomeres of the genomes π and σ. Every vertex $v \in \pi$ and $w \in \sigma$ is connected by $|A_v \cap A_w|$ edges, where A_v and*

A_w are the adjacencies (or telomeres) represented by vertices v and w. So, if A_v and A_w have two common extremities, then v and w are connected by two edges. If A_v and A_w have one common extremity, then v and w are connected by one edge. Otherwise, there is no edge between v and w.

In the adjacency graph, every vertex has degree 1 (if it is a telomere) or 2 (if it is an adjacency), and so the adjacency graph consists of cycles and paths. If the two genomes are equal, the adjacency graph consists of cycles of length 2 and paths of length 1.

We can see an example of the adjacency graph in figure 1.8. The adjacency graph was built for genomes π and σ:
$\pi = \{\{1-\}, \{1+, 3-\}, \{3+, 4+\}, \{4-\}, \{5+, 2-\}, \{2+, 5-\}, \{6-\}, \{6+, 7-\}, \{7+\}\}$
$\sigma = \{\{1+, 2-\}, \{2+, 1-\}, \{3-\}, \{3+, 4-\}, \{4+\}, \{5-\}, \{5+\}, \{6 + 7-\}, \{7+, 6-\}\}$

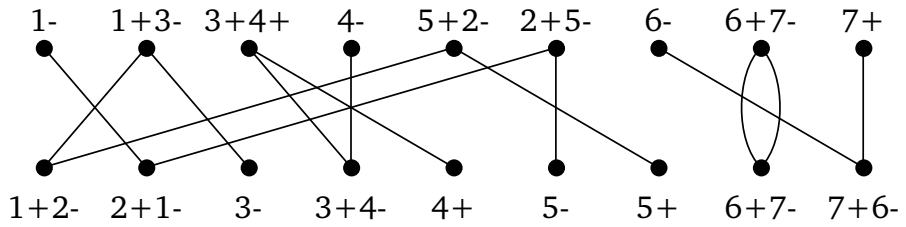


Figure 1.8: Adjacency graph $AG(\pi, \sigma)$

The following observations can be made [6]:

Lemma 1. *The application of a single DCJ operation changes the number of circular or linear components by at most one.*

Lemma 2. *Let π and σ be two genomes defined on the same set of g genes. Let the adjacency graph $AG(\pi, \sigma)$ have c cycles and p_O odd paths. Then $\pi = \sigma$ if and only if $g - (c + p_O/2) = 0$.*

Lemma 3. *The application of a single DCJ operation changes the number of odd paths in the adjacency graph by +2, 0, or -2.*

Corollary 1. *Let π and σ be two genomes defined on the same set of g genes. Let the adjacency graph $AG(\pi, \sigma)$ have c cycles and p_O odd paths. Then $dist_{DCJ}(\pi, \sigma) \geq g - (c + p_O/2)$.*

Corollary 1 gives us a lower bound on the distance, i.e. a lower bound on the number of DCJ operations needed to transform π into σ. On the other hand, if π and σ are not identical, we can always find a DCJ operation that decreases the distance by 1. Therefore, the following holds for the DCJ distance [6]:

Theorem 1 (The double cut and join distance)**.** *Let π and σ be two genomes defined on the same set of g genes. Let the adjacency graph $AG(\pi, \sigma)$ have c cycles and p_O odd paths. Then the DCJ distance between π and σ is:*

$$dist_{DCJ}(\pi, \sigma) = g - (c + p_O/2)$$

1.4.3 Other Distances

There are several other interesting genome rearrangement models and distance measures (see [4]). In this section, we only mention the Reversal model and the Reversal-Translocation model.

Reversal Distance

The Reversal model allows only the reversal rearrangement operation. If we compare this model with the DCJ model, we can imagine the Reversal model as a restricted DCJ model, where only those operations are allowed which do not create new chromosomes. The DCJ model allows reversals, and so it is a lower bound on the Reversal distance: $dist_{DCJ}(\pi, \sigma) \leq dist_{rev}(\pi, \sigma)$.

Reversal-Translocation Distance

In the Reversal-Translocation model, only the reversal and translocation operations are allowed.

1.5 The Median Problem

Definition 5 (The median problem)**.** *Let G be the set of all possible genomes on a particular set of genes that are allowed in a given rearrangement model, and let $dist$ be a distance measure on G. If we consider three genomes π, σ, γ, then the median problem is to find the genome $m \in G$, called a median that minimizes the sum of distances:*

$$dist(m, \pi) + dist(m, \sigma) + dist(m, \gamma)$$

The median problem can be used to compute the genome of an ancestral organism A of two species S_1 and S_2, and an outgroup species S_3 (see figure 1.9). According to the parsimony principle, we prefer evolutionary histories that explain present-day genomes with the smallest number of operations.

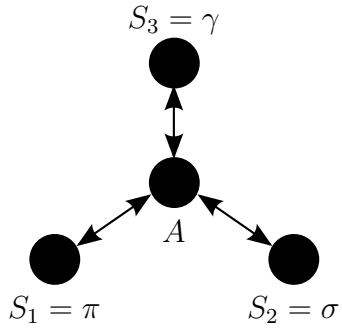


Figure 1.9: The preferred genome for A is the median of π, σ, and γ

Calculating the median is a hard problem; for most genome rearrangement models it was shown to be NP-hard. The NP-hardness of the median problem for multichromosomal DCJ model was shown in [5]. One interesting exception is the breakpoint distance on genomes with mixed chromosomes, where the median can be computed in polynomial time [5].

1.6 Small Phylogeny Problem

In the small phylogeny problem, we are given a phylogenetic tree and the genomes of the extant species (see figure 1.10). The task is to compute the genomes of ancestors, while minimizing the number of required genome rearrangement operations during evolution.

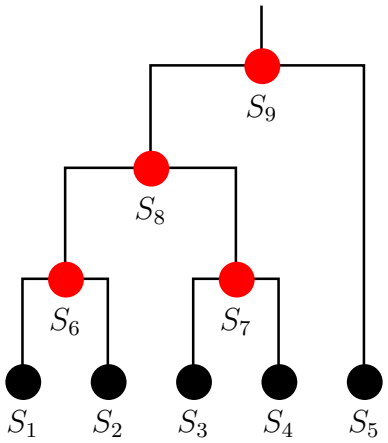


Figure 1.10: Small phylogeny problem: We know the phylogeny tree structure and the genomes of the extant species $S_1, ..., S_5$ (black nodes). The task is to compute the genomes of ancestors $S_6, ..., S_9$ (red nodes).

Because the median problem is a special case of the small phylogeny problem, it is obvious, that the small phylogeny problem is NP-hard for the majority of genome rear-

rangement models and distance measures. The NP-hardness of the small phylogeny problem for the breakpoint distance is shown in [11].

1.6.1 Solving the Small Phylogeny Problem

Probably the most popular method for solving the small phylogeny problem is the Steinerization method [7]. In this method, the algorithm iteratively tries to improve the evolutionary history until a local optimum is found. In each iteration, the algorithm cycles through the internal nodes of the tree (through the ancestors). For every internal node v, we take the genomes φ_a, φ_b, φ_c from the neighbouring nodes and calculate their median $\pi_M \in median(\varphi_a, \varphi_b, \varphi_c)$. We replace the genome inside node v with genome π_M, if the new tree has a better (lower) score (see figure 1.11). When none of the internal nodes can be improved, the algorithm has found a local optimum. The Steinerization method is described in Algorithm 1.

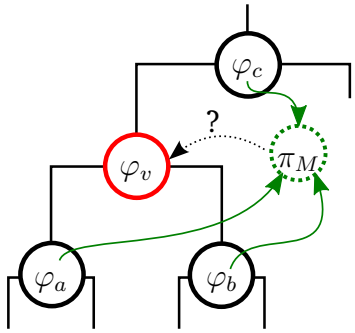


Figure 1.11: Steinerization method: the genome of vertex v is replaced by the median π_M, if it improves the score of the phylogenetic tree

Although the median problem is NP-hard in most rearrangement models, several solvers have been developed which work with acceptable time complexity. The Steinerization method was used in BPAnalysis software [12] [9] for the breakpoint model. The same method was used in GRAPPA software [13] [14] [15] for both the breakpoint and reversal models. The Steinerization method for the DCJ model was implemented by Adam and Sankoff [16].

MGR [17] 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.

A new algorithm used in PIVO software [1] encompasses and extends all these existing approaches. This new method is described in detail in chapter 2.

Data: phylogenetic tree, genomes of extant species
Result: local optimum
initialize evolutonary history h:
begin
  leaves: assign the genomes of extant species;
  internal nodes: assign a random genome;
repeat
  for $v \in V_{internal}$ **do**
    $\pi_M \in median(\varphi_a, \varphi_b, \varphi_c)$; // nodes a,b,c are the neighbours of v
    $h' = h, h'(v) = \pi_M$;
    if $score(h') < score(h)$ **then**
      replace φ_v with π_M in h;
until *no improvement*;
return h

Algorithm 1: Steinerization method

1.6.2 Definition of the Small Phylogeny Problem from a Computer Scientist's Perspective

The small phylogeny problem and its related terms are defined in this section in a more formal way:

Definition 6 (Phylogenetic tree)**.** *A phylogenetic tree is a binary tree $T = (V, E)$ rooted at node r, describing the evolutionary relationships between species. The leaves of the tree T are the extant species, and an internal node represents the most recent common ancestor of the node's children.*

Definition 7 (Evolutionary history)**.** *Let G be the set of all possible genomes on a particular set of genes, which is allowed in a rearrangement model. An evolutionary history h is a function, which assigns a genome from G to every node of a tree $T = (V, E)$: $h : V \to G$.*

In some cases in the following text, when we refer to the genome which is assigned to node v, we use φ_v instead of $h(v)$.

Definition 8 (Score of an evolutionary history)**.** *Let $dist$ be the distance measure of a rearrangement model. The score of the evolutionary history h on the phylogenetic tree $T = (V, E)$ is:*

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

Definition 9 (The small phylogeny problem)**.** *Let G be the set of all possible genomes on a particular set of genes, which is allowed in a rearrangement model, and let $dist$ be the distance measure on G. In the small phylogeny problem, we are given a phylogenetic tree $T = (V, E)$, with root r and leaves $L \subset V$. We know the genomes of the extant species, i.e. we are 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.*

Chapter 2

The PIVO Software

2.1 Introduction

The PIVO (Phylogeny by IteratiVe Optimization) software is a small phylogeny solver, which was created by Jakub Kováč [1]. PIVO was used to study the mitochondrial genomes of yeasts from the CTG clade of Hemiascomycetes [18]. This clade is interesting, because some species have genomes which consist of a single linear chromosome, some other species have two linear chromosomes, and the rest have a single circular chromosome. The first reason for creating the PIVO software was that, the existing small phylogeny solvers did not support multiple chromosome structures. The second reason was that the small phylogeny solvers mainly used the steinerization method and they applied various heuristics to enable the computation of data with thousands of genes. However, in the yeast study, only short mitochondrial genomes were analysed with a lower number of genes, and more accurate results were needed.

The original PIVO software was written in Python. In this thesis, the original software was rewritten in Java and several improvements were designed to get even better results and faster computation. We named our improved version of the original PIVO algorithm as PIVO2.

In this chapter, we describe both the original PIVO as well as our improved PIVO2 algorithm. In the description, we use the common name "PIVO" when the description applies to both PIVO and PIVO2. The name "original PIVO" is used, when the description applies to the original PIVO algorithm only.

2.2 The Algorithm

The PIVO algorithm uses a local search to find a good evolutionary history. The evolutionary history is initialized with some genomes, then the PIVO algorithm

iteratively tries to change the genomes in the evolutionary history to get a better score. The iterations are repeated until a local optimum is found.

In every iteration, a set of candidate genomes C_v is generated for every internal node v of the tree T. The candidates can be, for example, genomes within distance 1 of the current genome φ_v. More details on candidate generation can be found in section 2.4. The local search selects the best new evolutionary history from the neighbourhood of the current evolutionary history (see figure 2.1).

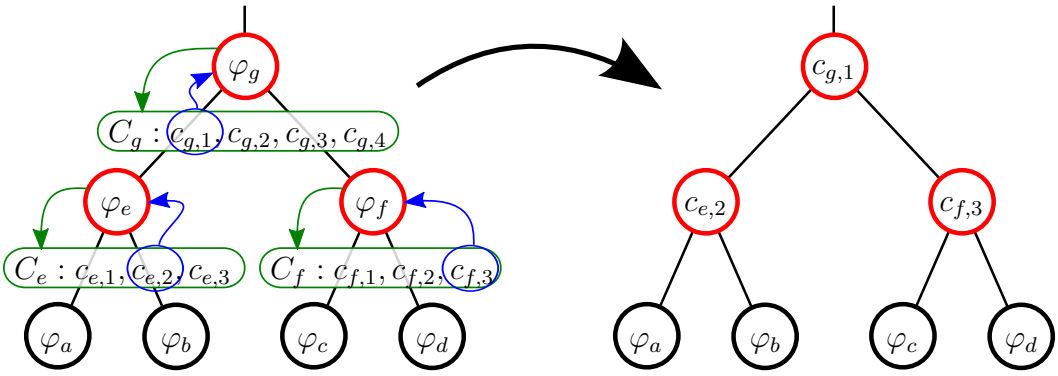


Figure 2.1: One iteration of the algorithm: For every internal node (red) the candidate sets are generated (green). Then the PIVO algorithm selects the best combination of the candidates (blue).

We get the neighbourhood of the current evolutionary history by selecting a candidate $c_v \in C_v$ for every internal node v and replacing the currently assigned genome φ_v of node v with c_v.

Definition 10 (Neighbourhood of the evolutionary history)**.** *The neighbourhood $N(h)$ of the evolutionary history h is a set of evolutionary histories:*

$$N(h) = \{h' \mid \forall v \in V_{internal} : h'(v) \in C_v\}$$

The size of the neighbourhood increases exponentially with the growing number of internal nodes: there are $\prod_v |C_v|$ neighbours. Fortunately, we do not have to enumerate the neighbours, and the neighbour with the lowest score can be computed efficiently by dynamic programming.

In the dynamic programming, we calculate the score of every candidate. Let us denote the i-th candidate of node v as $c_{v,i}$. The score $score(c_{v,i})$ of candidate $c_{v,i}$ is the lowest possible score we can get for the subtree rooted at v, if $h'(v) = c_{v,i}$. The score of a leaf is 0. Let us save the score of $c_{v,i}$ in the dynamic programming into the array denoted as $M[v, i]$. If v is an internal node with children u and w, first, we compute the candidate scores of nodes u and w. Then, we can compute the scores of

candidates from C_v:

$$M[v,i] = \min_j\{M[u,j] + dist(c_{v,i}, c_{u,j})\} + \min_k\{M[w,k] + dist(c_{v,i}, c_{w,k})\}$$

For the candidate $c_{v,i}$, we simply select which candidate genome from C_u should be assigned to node u to get the lowest score, and, independently, we select the best candidate for node w from C_w.

After we calculated the score for every candidate, we select the candidate in the root node of the phylogenetic tree with the lowest score. Then, we select the candidates for each child node: If u is the parent of v, we select the candidate $c_{v,i}$ for which the sum of $score(c_{v,i}) + dist(h(u), h(v))$ is minimal. The selection of the best candidate is detailed in algorithm 2.

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).

```
1  Function selectBestCandidate()
       // r is the root node
2      selected ← c_{r,1}, score ← M[r,1];
3      for c_{r,i} ∈ C_r do
4          if M[r,i] < score then selected ← c_{r,i}, score ← M[r,i];
5      h(r) ← selected;
       // u and v are the children of r
6      selectBestCandidateChild(u);
7      selectBestCandidateChild(v);
8  Function selectBestCandidateChild(v :node)
9      if v is leaf then return ();
10     selected ← c_{v,1}, score ← M[r,1] + dist(c_{v,1}, h(parent));
11     for c_{v,i} ∈ C_v do
12         if M[v,i] + dist(c_{v,i}, h(parent)) < score then
13             selected ← c_{v,i}, score ← M[v,i] + dist(c_{v,i}, h(parent));
14     h(v) ← selected;
       // u and w are the children of v
15     selectBestCandidateChild(u);
16     selectBestCandidateChild(w);
```

Algorithm 2: Selecting the candidates

The PIVO algorithm is flexible, and it can work with several genome rearrangement

models and distances. The DCJ model is used, in which the distance calculation can be done in $O(g)$ time. Therefore, the best evolutionary history in $N(h)$ can be found in $O(ngk^2)$ time.

The outline of the PIVO algorithm can be found in Algorithm 3.

Data: phylogenetic tree, genomes of extant species
Result: locally optimal evolutionary history
initialize evolutonary history h:
begin
    leaves: set the genomes of extant species;
    internal nodes: initialize internal nodes;
$s' \leftarrow score(h), s \leftarrow \infty$;
while $s' < s$ **do**
    for $v \in V_{internal}$ **do**
        generate candidates C_v;
    calculate score of the candidates;
    $h \leftarrow$ select best neighbourhood of h;
    $s \leftarrow s', s' \leftarrow score(h)$;
return h

Algorithm 3: The structure of the PIVO algorithm

2.3 The Usage of the PIVO Algorithm

The PIVO algorithm finds an evolutionary history, which is only a local optimum, and therefore, better histories may exist. It means, that the PIVO algorithm has to be run multiple times to find, with higher probability, a history that is near to the global optimum, see algorithm 4.

2.4 Improved Initialization and Candidate Generation in PIVO2

2.4.1 Initialization

At the beginning of the PIVO algorithm, the evolutionary history has to be initialized: We have to assign some genomes to the internal nodes of the phylogenetic tree.

```
1 T ← load phylogenetic tree;
2 S ← load the genomes of extant species;
3 R ← required number of runs;
4 i ← 0;
5 for i<R do
6 |   h ← Pivo(T,S);
7 |   save(h);
8 |   i ← i+1;
```

Algorithm 4: Repeated runs of the PIVO algorithm

Several strategies can be used for initialization. In the original PIVO algorithm, the internal nodes were initialized with completely random genomes.

Since there are at least $g!$ random genomes (actually more, if we consider gene orientation and chromosomes), the possibility that the randomly initialized internal nodes will be meaningful, is very low. Therefore, in the PIVO2 algorithm, we have chosen a different initialization method, in which we initialize the internal nodes with the genomes of the extant species. Every internal node v with children u and w is initialized randomly with φ_u or φ_w. Algorithm 5 describes this new initialization method.

```
1 Function initializeWithChild(v :node)
2 |   if v is internal then
  |   |   // u and w are the children of v
3 |   |   initializeWithChild(u);
4 |   |   initializeWithChild(w);
5 |   |   σ ← selectRandomly(φ_u, φ_w);
6 |   |   h(v) ← copy(σ)
7 |   return
```

Algorithm 5: Initialization with the genomes of extant species in PIVO2

2.4.2 Candidate Generation

Candidate generation is a very important part of the PIVO algorithm, because the algorithm is improving the existing evolutionary history by picking a new one from the neighbourhood $N(h)$, which is generated by selecting candidate genomes from the candidate sets. The time complexity of the algorithm depends quadratically on

the number of candidates, and so generating too many candidates may slow down the algorithm.

2.4.3 Strategies for Candidate Generation

The following candidate generation methods were proposed in [1].

Extant Species

The genome of every extant species is inserted into every candidate set.

$$cand_{extant}(v) = \{h(u) \mid u \in V_{leaves}\}$$

Neighbours

For every internal node v, the neighbourhood of genome φ_v is inserted into the candidate list C_v. By neighbourhood, we refer to the genomes which are allowed in the used rearrangement model and are at most in distance 1 from φ_v:

$$cand_{neigh}(v) = \{\sigma \mid \sigma \in G, dist(\varphi_v, \sigma) \leq 1\}$$

In the DCJ model, and also in most other models, the size of the neighbourhood is $O(g^2)$. For a bigger g, this number can be rather big, so a useful extension of this method is to generate only those neighbours which do not increase the distance from the genomes of the three neighbouring nodes too much. The maximum increase in distance can be 3. Therefore, a reasonable limit l for the distance increase is 0, 1, or 2.

$$cand_{neighLimited}(v) = \{\sigma \mid \sigma \in G, dist(\varphi_v, \sigma) \leq 1, \sum_{(u,v)\in E} (dist(\varphi_v, \varphi_u) - dist(\sigma, \varphi_u)) \leq l\}$$

Medians

Similarly, as in the steinerization method, the medians of the three neighbouring nodes could be inserted into the candidate set. Many medians may exist, but the algorithm would not need to choose only one of them as in the steinerization method. The algorithm could add all the medians into the candidate set.

$$cand_{median}(v) = \{\sigma \mid \sigma \in median(\varphi_a, \varphi_b, \varphi_c)\}$$

Intermediates

If a node v is adjacent to nodes u and w, the algorithm could add to C_v the genomes σ, for which $dist(\varphi_u, \sigma) + dist(\sigma, \varphi_w) = dist(\varphi_u, \varphi_w)$.

2.4.4 Candidate Generation Strategies in PIVO2

From the strategies described in section 2.4.3, in the PIVO2 algorithm, we opted for the use of the "Neighbours" candidate generation method. In the "Neighbours" method, we can optionally turn on the extension which limits the generated neighbour genomes on the basis of the distance increase.

We proposed and added to the PIVO2 algorithm a new strategy for candidate generation called "Tree".

Tree

The genome from every node of the tree is inserted into every candidate set. This method is useful, because genomes can "jump" from one node to another, if the "jump" yields a better evolutionary history.

$$cand_{tree}(v) = \{h(u) \mid u \in V\}$$

2.5 Alternative Genomes in Leaves

The PIVO algorithm has an extension which enables to assign multiple alternative genomes for the leaves of the evolutionary history. This is useful, because sometimes, we do not know exactly what is the correct genome of an extant species, and in this case, we can define multiple possible genomes for the algorithm. Another case is, when the genome of an extant species contains a duplicated gene. In this situation, we can delete one copy of the gene and define all the possible results after the deletion as an alternative genome.

During every iteration of the PIVO algorithm, the algorithm selects the best genome from the alternatives. We can simply imagine these alternative genomes as if they were candidates for the leaves. So, our dynamic programming can be used to select the best alternative genome, in the same way as it selects the best candidate for internal nodes.

2.6 Randomization of Candidate Selection in the PIVO2 Algorithm

As we can see in algorithm 2, the original PIVO software always selects the first best candidate during candidate selection. However, this is not optimal, because often the same decision is made in repeated searches. In the PIVO2 algorithm, we introduced

randomized candidate selection. If there are multiple evolutionary histories in the neighbourhood of the current history, which are equally good, 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.

Counting the Solutions in PIVO2

During the dynamic programming, the PIVO2 algorithm not only calculates the score of each candidate $c_{v,i}$, but it also counts how many solutions with minimal score exist in the subtree rooted at v, if $c_{v,i}$ is selected. The PIVO2 algorithm stores the numbers of solutions for candidate $c_{v,i}$ in $Q[v,i]$. If node v has children u and w, the numbers of solutions for every candidate of nodes u and w have to be first calculated. Then, the numbers of solutions for the candidates of node v are calculated: For each candidate $c_{v,i}$, the PIVO2 algorithm calculates the sets $BESTLEFT[v,i]$ and $BESTRIGHT[v,i]$, containing those candidates from the left and right child, for which the minimal score $M[v,i]$ can be achieved.

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

$$BESTRIGHT[v,i] = \{c_{w,j} \in C_w \mid M[w,j] + dist(c_{w,j}, c_{v,i}) = M[v,i]\}$$

Knowing the $BESTLEFT[v,i]$ and $BESTRIGHT[v,i]$, the computation of $Q[v,i]$ is easy. The number of solutions for the left subtree is summed, similarly, the number of solutions for the right subtree is summed, then the two sums are multiplied.

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

If node v is a leaf, its number of solutions is 1, or, if alternative genomes are defined in the leaf, then, for alternative p, the number of solutions is $Q[v,p] = 1$.

Randomized Selection of Candidates in PIVO2

To select the candidate at the root node r, we consider the set of candidates which have minimal score:

$$GOODCAND = \{c_{r,i} \mid c_{r,i} \in C_r, M[r,i] \; is \; minimal\}$$

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

$$Q[r] = \sum_{i:\, M[r,i] \; is \; minimal} Q[r,i]$$

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

$$Q[v] = \sum_{i:\, c_{v,i} \in BESTLEFT[p,k]} Q[v,i]$$

If the node v is the right child of p, then the selection is made similarly, but, instead of using set $BESTLEFT[p,k]$, the set $BESTRIGHT[p,k]$ is used.

This randomized selection can be implemented without increasing the time complexity of the score calculation and candidate selection algorithms.

2.7 Tabu Search in the PIVO2 Algorithm

The randomized selection of candidates in PIVO2 (see section 2.6) prevents making the same decision in repeated searches. Additionally, in the PIVO2 algorithm, a different approach, called Tabu search, is implemented.

The Tabu search [19] [20] is a metaheuristic search method, which is often used in local searches to avoid getting always the same local optimum. The Tabu search uses a data structure called a tabu list. During local searches, those neighbours which are in the tabu list, are penalized, and are less likely to be chosen.

The algorithm uses a tabu list T_v for every internal node v of the phylogenetic tree. The list T_v contains genomes π which were assigned to the node in those previous iterations that resulted in $h(v) = \pi$.

When the dynamic programming calculates the score of a candidate $c_{v,i}$, it adds a penalty p to the score, if the candidate $c_{v,i}$ is inside the tabu list T_v of the node v. So, if there are other good candidates which were not present in a previous solution, the candidate $c_{v,i}$ is not selected. The penalty p should not be too high, because getting an evolutionary history with a lower score is more important than getting an evolutionary history with a higher score and a lower number of candidates included in the tabu lists. A reasonable value for the penalty is $p = 1/(|V| + 1)$. With this penalty, even if the next evolutionary history consists only of penalized candidates, the sum of the penalties is lower than 1. This penalized evolutionary history will then be preferred, even if there exists a worse history with a lower number of tabu penalties.

The tabu lists are kept during multiple runs of the PIVO2 algorithm.

A tabu list is implemented as a hashset of genomes, and so, it can be checked efficiently whether a genome is inside a tabu list.

2.8 Combining Previous Solutions in PIVO2

Based on a proposal in [1], we implemented another heuristic method in the PIVO2 algorithm, which can be optionally turned on. It is a type of candidate generation. Using this heuristic, the algorithm takes the genomes from previous solutions, and adds them into the candidate lists. The motivation is that, by combining different solutions, we may get a better evolutionary history. If H is the set of evolutionary histories found by the PIVO2 algorithm in the previous runs, the heuristic adds the genomes $\{h(v) \mid h \in H\}$ into candidate set C_v of node v.

2.9 Preferred Chromosome Types in PIVO2

In the DCJ model, a genome can have an arbitrary structure: it can be a mix of linear and circular chromosomes. However, in the real world, the organisms usually have one of the following two types of genome structures: i) one circular chromosome, ii) one or more linear chromosomes. In the PIVO algorithm, the preferred type of genome structure can be prioritised by penalizing genomes which do not have the preferred structure. The penalty is added to the score of the candidates in the dynamic programming. In the original PIVO algorithm, the penalty setting was not solved optimally.

A more optimal penalty setting is implemented in PIVO2, which is described here. Choosing a suitable value for the penalty is important. If the penalty is too low, this penalizing method will not work. If the penalty is too high, the local search may not function satisfactorily.

From section 1.4.2, we know that with one DCJ operation, we can do circularization and linearization, fusion and fission, circular excision and integration. So, if we repair the genome φ_v (i.e. make the required rearrangement operations to get a preferred genome structure), the score of the subtree rooted at v increases at most by $2 \cdot r$, where r is the number of repair operations. We have defined the penalty to be $p = 2 \cdot r$. With this penalty setting, the PIVO2 algorithm works quite well, and the genomes mostly have the preferred structure.

If a genome consists of l linear and c circular chromosomes, then, depending on the preferred genome structure, the number of required repair operations is as follows:

- **Preferred genome structure: one circular chromosome**
  If there are more circular chromosomes, all of them need to be fused into one circular chromosome, and so we need $c - 1$ fusions. All the linear chromosomes need to be integrated into the circular chromosome; we need l operations for this. The total number of repair operations is: $r = c + l - 1$.

- **Preferred genome structure: one or more linear chromosomes**
  Every circular chromosome needs to be linearized; we need c operations. The total number of repair operations is: $r = c$.

- **Preferred genome structure: one circular, or one or more linear chromosomes**
  This is the combination of the first two cases. The algorithm selects the case, in which fewer repair operations are needed. The total number of repair operations is: $r = min(c + l - 1, c)$.

2.10 Implementation Details of the PIVO2 algorithm

The original PIVO algorithm is written in Python. The source code of the original PIVO software does not contain detailed comments, and so it would be rather time consuming to understand it fully. For this reason we completely reimplemented the original software by writing PIVO2. Instead of Python, we used Java for PIVO2. Details of the implementation of the PIVO2 algorithm are discussed in this section.

2.10.1 Genome Representation

The PIVO2 algorithm stores the adjacencies in an array G (we call it the genome array). For every extremity, the algorithm stores its adjacent extremity in G. If an extremity e is a telomere, then it is saved into the array as $G[e] = e$. An example of a genome array can be seen in figure 2.2.

In a genome array, the head of gene a is represented by index $2 \cdot a - 1$ and its tail by index $2 \cdot a - 2$. This array representation is good, because we can quickly tell which extremity is adjacent to another given extremity, and we can easily save the changes after a rearrangement operation. In a genome array, every adjacency is stored twice and the telomeres are stored once.

In PIVO2, we sometimes need to compute a hash value of a genome. To do so, we simply calculate the hash value of its genome array.

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	+3	+5	+6	-4	+1	-3	+4	-6	-2	-5	+2

Figure 2.2: The genome array of genome: 1 -3 4 $ -2 -5 6 @

2.10.2 Rearrangement Model

In the PIVO2 software, the DCJ rearrangement model is implemented. The model includes methods for applying a DCJ operation to a genome, distance calculation, and neighbour generation. The PIVO algorithm is general, and it would work with other rearrangement models too. For this reason, in PIVO2 we access the DCJ model implementation through a rearrangement model interface. Using this interface, other rearrangement models can be easily added.

Chapter 3

Efficient Distance Calculation

The PIVO algorithm makes a lot of distance calculations between pairs of candidates in the dynamic programming. When it calculates the scores of candidates of node v (with children u and w), it calculates distances between every pair of candidates from C_v and C_u, and C_v and C_w respectively.

In the PIVO2 algorithm, the "Neighbour" and the "Tree" candidate generation methods are used. 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. More efficient distance calculation in these cases could significantly increase the speed of the PIVO algorithm. We have designed a faster method of calculating the breakpoint and DCJ distances in such cases. This new and more efficient distance calculation method, which is the one of the main improvements of the PIVO2 algorithm, is described in this chapter (see sections 3.1 and 3.2).

3.1 Breakpoint Distance

The breakpoint distance of genomes π and σ can be simply calculated (see section 1.4.1). If g is the number of genes, $a(\pi, \sigma)$ is the number of common adjacencies, and $e(\pi, \sigma)$ is the number of common telomeres, then the breakpoint distance between π and σ is:

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

In the PIVO2 algorithm, we represent every genome as a genome array, and in this representation, every adjacency is stored twice and the telomeres are stored only once. So, the distance can be computed by simply counting the number of indexes where the array values differ (see figure 3.1). If $diff(\pi, \sigma)$ is the number of indexes with different values in the arrays of the genomes, the breakpoint distance of π and σ is:

$$dist_{BP}(\pi, \sigma) = \frac{diff(\pi, \sigma)}{2}$$

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

(a) 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

(b) Genome 2: 1 2 3 4 @ 5 6 $

Figure 3.1: The differing columns are marked in red, the breakpoint distance is 1.5

The breakpoint distance can be easily computed in $O(g)$ time. If we have two sets of genomes A and B, each of size $\Theta(g^2)$, the distance between every pair of genomes from sets A and B can be computed in time $O(g^5)$. But, if we know that the set A consists of genomes π_i which are in DCJ distance 1 from a genome π, and similarly, the set B consists of genomes σ_j which are in DCJ distance 1 from a genome σ, we can calculate the distances more efficiently, as we show next.

Connection Between the DCJ Operation and the Breakpoint Distance

A DCJ operation cuts at most two adjacencies or telomeres, and from their extremities it creates at most two new adjacencies or telomeres. So, a DCJ operation changes the genome array in the worst case at 4 indexes. Therefore, a DCJ operation can be described as a set of pairs $(index_i, extremity_i)$, where $index_i$ means which index was updated in the array G, and $extremity_i$ is the new value $G[index_i] = extremity_i$. Let us call this pair an "update pair", and the set of these pairs an "update set". An update set can describe complex rearrangement operations, but if an update set U contains information about a single DCJ operation, then its size is $|U| \leq 4$. If we know the update set U that transforms a genome π into a genome σ, then the distance between

π and σ can be calculated in constant time:

$$dist_{BP}(\pi, \sigma) = \frac{|U|}{2}$$

If we do not know the update set which transforms π into σ, then we do not improve the time complexity of a single distance computation, because computing the update set takes $\Theta(g)$ time. However, during "neighbour" candidate generation, we know for every candidate which DCJ operation was performed, and we can save this additional information as an update set without increasing the time complexity. As it is explained in the next sections, with this information we can compute the distances more efficiently.

3.1.1 Efficient Distance Calculation Between a Genome and a Set of Genomes

Let us first discuss a simplified problem, when our task is to compute the breakpoint distances between a single genome π and each genome σ_j from set B, where each σ_j is a genome in DCJ distance 1 from a genome σ. We also know for every genome σ_j the update set U_j, which transforms σ into σ_j. The trivial (less efficient) way of calculating the distances would take $O(g \cdot |B|)$ time. However, it is more efficient to compute the distance between π and σ and then to check for each σ_j, how the update set U_j changes the distance (see figure 3.2).

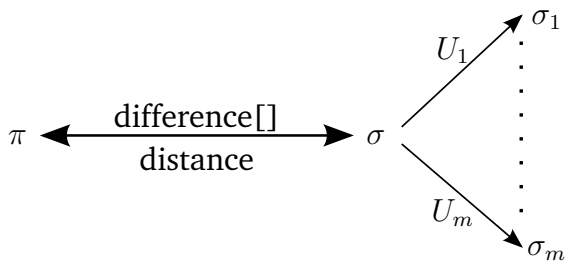


Figure 3.2: Efficient breakpoint distance calculation between a genome and a set of genomes

We calculate a boolean array called $difference$, which marks differences between genome arrays π and σ. If $\pi[k] \neq \sigma[k]$, then $difference[k] = true$, otherwise it is $false$. If the number of $true$ values in the whole array is t, then the distance is $dist_{BP}(\pi, \sigma) = t/2$. To get the distance of genomes π and σ_j, we must analyse the update pairs in set U_j, and update the distance. For every update pair $(index_i, extremity_i)$ in U_j, three cases may occur:

1. The update pair defines a value which was present in the array of π:

$$\pi[index_i] = extremity_i$$

This means that the difference between π and σ_j gets smaller by 1, so we update the distance by subtracting $1/2$.

2. The update pair redefines a value at an index, at which the values in arrays of π and σ are different, and the redefined value differs from the value in the array of π:

$$difference[index_i] = true \curlywedge \pi[index_i] \neq extremity_i$$

In this case, the difference between the genomes does not change, so the distance remains the same, and no update is needed.

3. The update pair changes the value at an index, at which the values in arrays of π and σ are the same:

$$difference[index_i] = false$$

The difference between π and σ_j increases by 1, so the distance is increased by 1/2.

All the three cases can be checked in constant time, so the distances can be calculated in time $O(g+|B|\cdot u)$, where u is the size of update sets U_j (see algorithm 6). Because every σ_j was derived from σ using a single DCJ operation, the size of u is at most 4. So, the distances can be calculated in $O(g + |B|)$ time.

3.1.2 Efficient Distance Calculation Between Two Sets of Genomes

Let us extend the previous method, so that we calculate the 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. We know for every genome $\pi_i \in A$ the update set U_i^A which transforms π into π_i, and also know for every genome $\sigma_j \in B$ the update set U_j^B which transforms σ into σ_j. The trivial (less efficient) solution has $O(g \cdot |A| \cdot |B|)$ time complexity. Our more efficient solution, similar to the previous section, calculates the distance of π and σ and then it analyses, how the update pairs change the distance (see figure 3.3).

As in the previous section, we calculate the $difference$ array of genomes π and σ. If the number of $true$ values is t, then the distance is $dist_{BP}(\pi, \sigma) = t/2$. Next, we check, how the update set U_j^B changes the distance, using the method from the previous section. The result is the distance of π and σ_j. Then we analyse the update

```
1  Function breakpointDistance(π,σ: genome, B: set of genomes)
2      difference[] ← calculateDifference(π, σ);
3      distance ← count the number of true values in difference[];
4      for σ_j ∈ B do
5          distance[π,σ_j] ← distance;
6          for (index_i, extremity_i) ∈ U_j do
7              if π[index_i] = extremity_i then
8                  distance[π,σ_j] ← distance[π,σ_j]-1/2
9              else if difference[index_i]=true then
10                 nothing
11             else
12                 distance[π,σ_j] ← distance[π,σ_j]+1/2
13     return distance[]
```

Algorithm 6: Efficient breakpoint distance calculation between a genome and a set of genomes

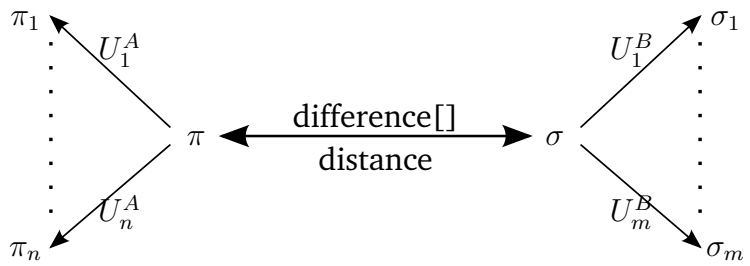


Figure 3.3: Efficient breakpoint distance calculation of a set versus a set

set U_i^A to get the distance of π_i and σ_j. Here, more cases exist, because several conflicting situations can occur between the update pairs of U_i^A and U_j^B. For every $(index_k, extremity_k) \in U_i^A$ we do the following:

1. The distance is lowered by $1/2$ if one of the following holds:

   - If the update pair is also present in U_j^B $(index_k, extremity_k) \in U_j^B$:
     It means that, in the array representation of π_i and σ_j, the value at index $index_k$ is the same. When we were analysing the update pair $(index_k, extremity_k)$, the distance was increased by $1/2$, and so, now we have to undo this update by subtracting $1/2$.
   - If in the array representation of σ, we have $\sigma[index_k] = extremity_k$, and the value at index $index_k$ was not changed by the update set U_j^B:
     It means that $\sigma_j[index_k] = extremity_k$, too. Therefore, at index $index_k$,

both π_i and σ_j have the same value, and the difference is lowered by 1. So we have to subtract $1/2$ from the distance.

2. Otherwise, if one of the following holds, the distance is not changed:

   - If there is difference at index $index_k$ between π and σ, and there is no update pair in U_j^B which sets the value at $index_k$ to $\pi[index_k]$:
     It means that the difference at index $index_k$ was already calculated into the distance between π_i and σ_j, and so no distance update is needed.
   - If there is an update pair in U_j^B that changes the value at $index_k$, but it does not set the value to $\pi[index_k]$:
     Again, the difference at $index_k$ was already calculated into the distance, and so, no update is needed.

3. Otherwise, the update pair $(index_k, extremity_k)$ increases the distance, and so, 1/2 is added to the distance.

After these updates, we get the distance of π_i and σ_j. If u is the maximum size of an update set, then the time complexity is $O(g+|A|\cdot|B|\cdot u^2)$. Algorithm 7 summarizes the efficient breakpoint calculation.

The u^2 in the time complexity stems from the fact that for every update pair in U_i^A, we have to answer the following questions

- "Was the value at index x changed in U_j^B?"
- "Is there an update pair in U_j^B which sets the value at index x to $\pi[x]$?"
- "Is there an update pair in U_j^B which sets the value at index x to a value not equal to $\pi[x]$?"

The answer to these questions can be found in time $O(u)$. However, we can precalculate the answers to all of these questions in a total time $O(|B|\cdot u)$: We cycle through every update pair in U_j^B. If there is an update pair $(index_k, extremity_k) \in U_j^B$ for which we get a "YES" answer to a question for which $x = index_k$, we save, that for U_j^B for index $x = index_k$ the answer is "YES", into a hash table. After this precalculation, answering a question becomes a simple lookup operation in the hash table, and so the answer can be found in time $O(1)$. After this improvement, the distance calculations are done in time $O(g + |A|\cdot|B|\cdot u)$.

Since every π_i and σ_j are in DCJ distance 1 from π and σ, the size of the update sets is at most 4. In this special case, the distances can be calculated in $O(g + |A|\cdot|B|)$ time.

```
1  Function breakpointDistance(π,σ: genome, A,B: set of genomes)
2      difference[] ← calculateDifference(π, σ);
3      distance ← count the number of true values in difference[];
4      for π_i ∈ A do
5          for σ_j ∈ B do
6              distance[π_i,σ_j] ← distance;
7              for (index_k, extremity_k) ∈ U_j^B do
8                  if π[index_k] = extremity_k then
9                      distance[π_i,σ_j] ← distance[π_i,σ_j]-1/2
10                 else if difference[index_k]=true then
11                     nothing
12                 else
13                     distance[π_i,σ_j] ← distance[π_i,σ_j]+1/2
14             for (index_k, extremity_k) ∈ U_i^A do
15                 if (index_k, extremity_k) ∈ U_j^B ⊻
                   (σ[index_k] = extremity_k ∧ ∄y : (index_k, y) ∈ U_j^B) then
16                     distance[π_i,σ_j] ← distance[π_i,σ_j]-1/2
17                 else if (difference[index_k] = true ∧ (index_k, π[index_k]) ∉ U_j^B) ⊻
                   (∃y : (index_k, y) ∈ U_j^B ∧ y ≠ π[index_k]) then
18                     nothing
19                 else
20                     distance[π_i,σ_j] ← distance[π_i,σ_j]+1/2
21     return distance[]
```

Algorithm 7: Efficient breakpoint distance calculation of a set versus a set

3.2 Double Cut and Join Distance

The Double Cut and Join distance of two genomes π and σ can be calculated by creating the adjacency graph (see section 1.4.2). If g is the number of genes, c is the number of cycles, and p_O is the number of odd paths in the adjacency graph, the DCJ distance is:

$$dist_{DCJ}(\pi, \sigma) = g - (c + p_O/2)$$

The creation of the adjacency graph and counting of the components can be done in time $O(g)$, and so the DCJ distance can be calculated in $O(g)$ time [6]. If we want to calculate the distance between every pair of genomes $\pi_i \in A$ and $\sigma_j \in B$, it can be done simply in $O(g \cdot |A| \cdot |B|)$ time.

3.2.1 Efficient Distance Calculation Between Two Sets of Genomes

If we know that every genome $\pi_i \in A$ and every genome $\sigma_j \in B$ are only in DCJ distance 1 from some genomes π and σ respectively, and if we have information about the relationship of π_i to π and σ_j to σ, then we can calculate the distances more efficiently.

In a DCJ operation, we break at most 2 adjacencies or telomeres, and then create at most 2 new adjacencies or telomeres. So, we can describe a DCJ operation by two lists: a disconnect list containing pairs of extremities which must be disconnected, and a connect list containing pairs of extremities which must be connected.

When we generate the neighbour candidates in the PIVO2 algorithm, we can create these lists for each candidate without increasing the time complexity.

Our approach for efficient DCJ distance calculation is similar to the one we used in breakpoint distance calculation: We calculate the adjacency graph for the genomes π and σ. Then, for every pair π_i and σ_j, we check how the operations from $disconnect_i^A$, $connect_i^A$, $disconnect_j^B$, and $connect_j^B$ change the components of the adjacency graph, i.e. how they change the distance (see figure 3.4).

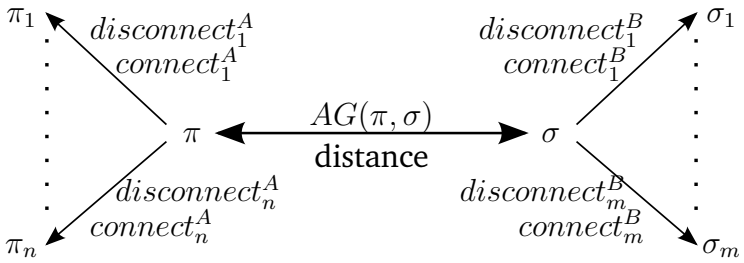


Figure 3.4: Efficient DCJ distance calculation of a set versus a set

Example 1 (Adjacency graph)**.** *Figure 3.5 shows the adjacency graph of genomes* $\pi =$ *a -d -e $ b -c -f @ and* $\sigma =$ *a -e d $ b c -f @. We assigned a different color to each component.*

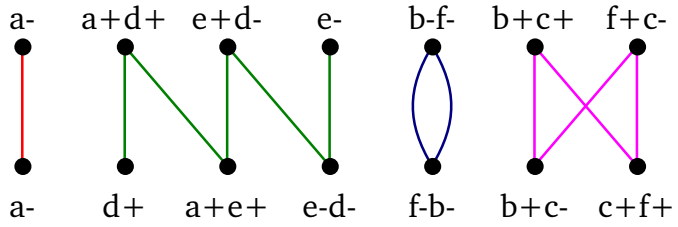


Figure 3.5: Adjacency graph $AG(\pi, \sigma)$

Data Structures

For tracking the changes in the adjacency graph, we will use several data structures. First, we have to save the information about the components of the adjacency graph $AG(\pi, \sigma)$. An adjacency graph consists of vertices of degree one or two, and so, an adjacency graph consists of paths and cycles. Let us imagine the paths as a list of extremities. In this list, the extremities are ordered according to their position along the path. Similarly, let us imagine cycles as "cyclic" paths (the first and last element of the extremity list is connected). We assign specific numbers to these paths.

Now, we can generate the following arrays in $O(g)$ time:

- $extremityLocation$ - this array maps every extremity to the number of the path in which the extremity is located
- $extremityPosition$ - this array maps every extremity to the position within the list of extremities
- $componentSize$ - maps every path to its extremity list size
- $componentCircular$ - is a boolean array, which stores whether a path is cyclic or not

Example 2 (Paths of the adjacency graph)**.** *In figure 3.6, we listed the paths of the adjacency graph from example 1. The extremities of genome σ are written with uppercase letters.*

Path 1: a-, A-
Path 2: D+, d+, a+, A+, E+, e+, d-, D-, E-, e-
Path 3: b-, B-, F-, f-, cyclic
Path 4: c+, C+, F+, f+, c-, C-, B+, b+, cyclic

Figure 3.6: Paths of the adjacency graph $AG(\pi, \sigma)$

Example 3 (Arrays for storing paths)**.** *In figure 3.7, we can see the arrays storing the paths from example 2.*

As we apply the DCJ operations (disconnecting and connecting extremities), the components of the adjacency graph change. Paths can split and join; they can change from linear to cyclic, and vice versa. To track the changes of components, we use a data structure called $Comp$. A $Comp$ object saves the information about the original paths' pieces which form the component. In the $Comp$ object, we also save whether

a-	a+	b-	b+	c-	c+	d-	d+	e-	e+	f-	f+
1	2	3	4	4	4	2	2	2	2	3	4

(a) Extremity location

a-	a+	b-	b+	c-	c+	d-	d+	e-	e+	f-	f+
0	2	0	7	4	0	6	1	9	5	3	3

A-	A+	B-	B+	C-	C+	D-	D+	E-	E+	F-	F+
1	3	1	6	5	1	7	0	8	4	2	2

(b) Extremity position

1	2	3	4
2	10	4	8

(c) Component size

1	2	3	4
N	N	Y	Y

(d) Component circular

Figure 3.7: The arrays for storing the paths

the component is a linear path or a cyclic path (cycle). A $Comp$ object CO_i also has pointers to other $Comp$ objects. The objects to which CO_i points, are components which came into being after CO_i was changed by a connect or disconnect operation. The $Comp$ objects form a forest.

To summarize, a $Comp$ object holds the following information about a component of the adjacency graph:

- list of original paths with start and end indexes
- circularity of the component
- pointers to child $Comp$ objects

Now, we have the necessary data structures for tracking the changes in the adjacency graph.

Preparation for Tracking the Changes

Before starting to track the changes, we create a $Comp$ object for every component in the adjacency graph. Every $Comp$ object holds a single path. Then, we create an array $extremityComp$ which will map every extremity into the appropriate $Comp$ object (see figure 3.12). The $extremityComp$ array is similar to the $extremityLocation$ array.

Tracking the Changes

Now, for every pair $\pi_i \in A$ and $\sigma_j \in B$, we look at the lists $disconnect_i^A$, $connect_i^A$, $disconnect_j^B$, $connect_j^B$ and analyse, how the disconnect and connect operations change the adjacency graph, and how the number of cycles and odd paths change.

Disconnecting. If we have to disconnect extremities ext_1 and ext_2, we must find the $Comp$ object in which we have these extremities. Finding it is easy, we just start with the array $extremityComp$, and follow the pointers to $Comp$ objects. If a $Comp$ object has two children, we can easily decide which child contains the searched extremity using the arrays $extremityLocation$ and $extremityPosition$. We follow the pointers until we get to a $Comp$ object which has no children.

If the component is circular, we create a new linear component and make a pointer to it, as seen in figure 3.8.

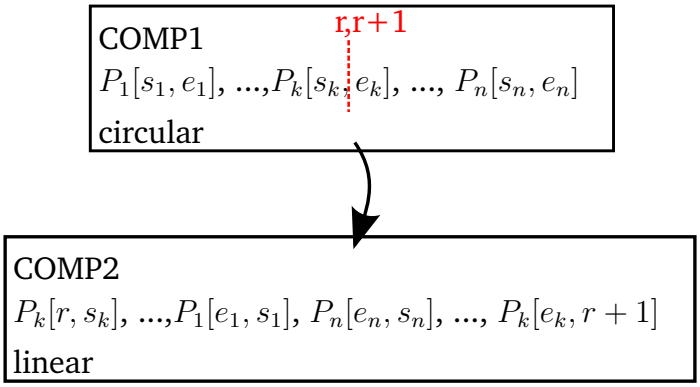


Figure 3.8: Disconnecting a circular component: ext_1 is in $P_k[r]$ and ext_2 is in $P_k[r+1]$

If the found component is linear, we will get two new linear components after disconnection (figure 3.9).

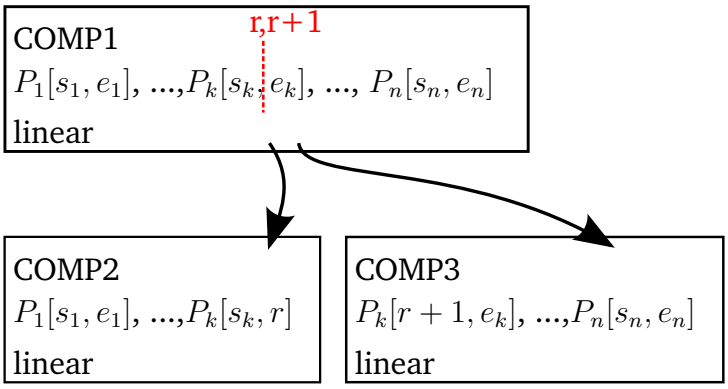


Figure 3.9: Disconnecting a linear component: ext_1 is in $P_k[r]$ and ext_2 is in $P_k[r+1]$

Connecting. Connecting extremities ext_1 and ext_2 is done similarly. We just find the components $comp1$ and $comp2$ which contain the extremities ext_1 and ext_2.

If $comp1 \neq comp2$, then we connect two linear components into a single linear component. We create a new component $comp3$, and make pointers to it from $comp1$ and $comp2$, as seen in figure 3.10.

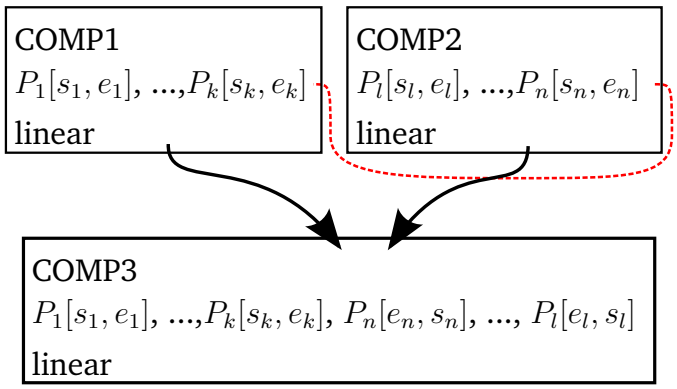


Figure 3.10: Connecting two linear components: ext_1 is in $P_k[e_k]$, and ext_2 is in $P_n[e_n]$.

If $comp1 = comp2$, then we connect the two ends of a linear component. We get a new circular component $comp3$, and make a pointer to it from $comp1$, as seen in figure 3.11.

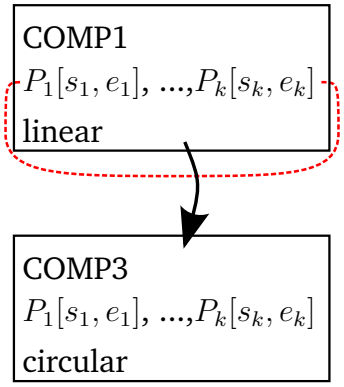


Figure 3.11: ext_1 is in $P_1[s_1]$, and ext_2 is in $P_k[e_k]$. So, we are connecting the two ends of a linear path.

When we finished processing every connect and disconnect operation, we successfully got the components of the adjacency graph of π_i and σ_j. After each connect and disconnect operation, the number of cycles and odd paths was updated, and so we know the distance of π_i and σ_j.

Before calculating the distance of the next pair, we delete the $Comp$ objects which were created during tracking.

Example 4 (Tracking the changes in the adjacency graph)**.** *In this example, we track the changes of the adjacency graph from example 1. We make two disconnect and two connect operations. The tracking is illustrated in figure 3.12.*

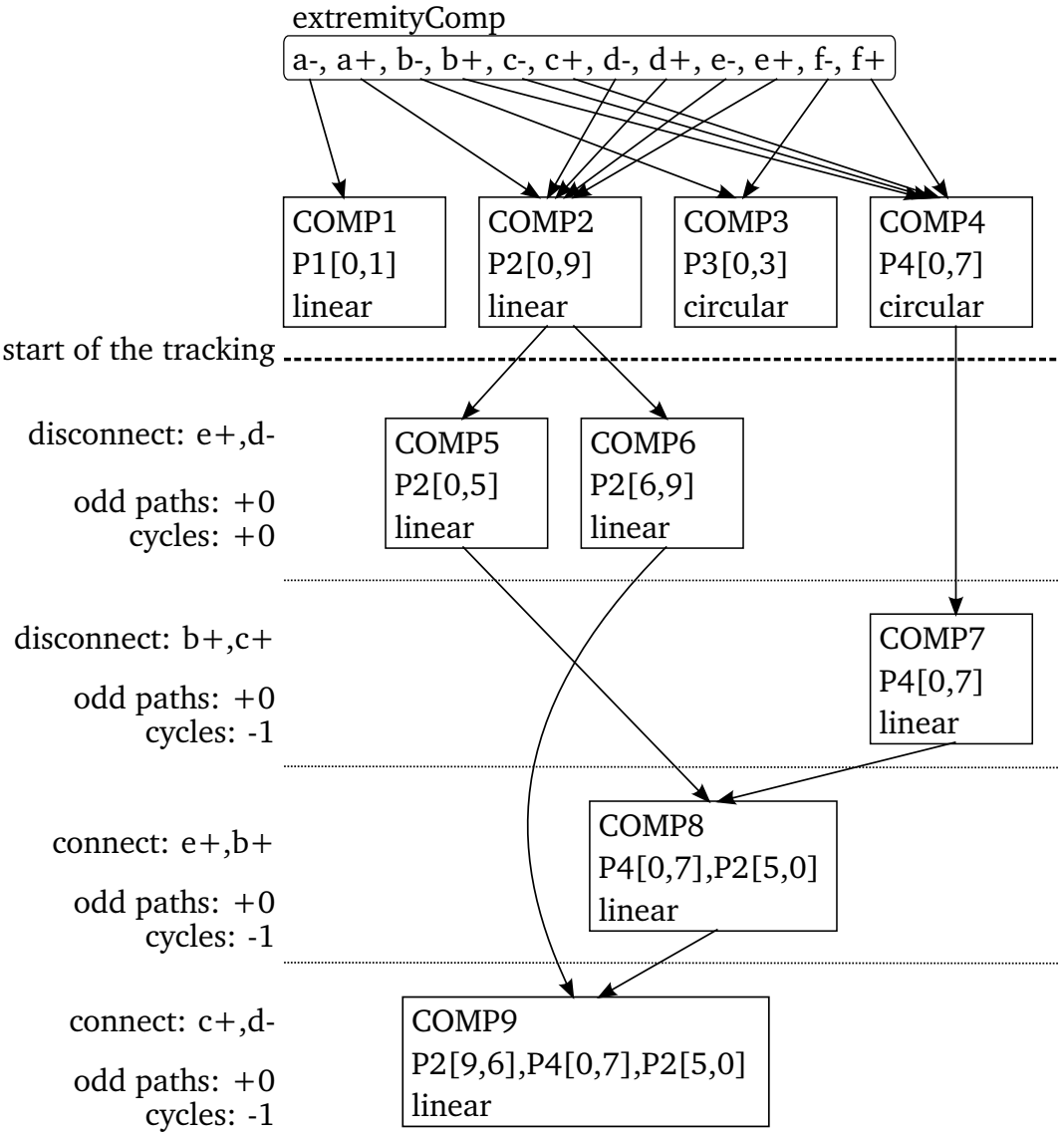


Figure 3.12: Tracking the changes

Time Complexity

The initialization of the tracking takes O(g) time, but it has to be done only once.

Every disconnect and connect operation increases the depth of the components by 1. Every disconnect and connect operation increases the maximal possible size of the path list in the $Comp$ object by 1. If o is the maximum number of operations in a connect/disconnect list, then the computation of the distance between a pair of candidates takes $O(o^3)$ time: For $O(o)$ operations, we have to find the component which can be in depth $O(o)$, and during search, when we are choosing the child component, we have to check $O(o)$ path list items. Generating a new child component needs $O(o)$ time.

So, calculating the distance between every pair has $O(g + |A| \cdot |B| \cdot o^3)$ time

complexity. Because every $\pi_i \in A$ and $\sigma_j \in B$ is only in DCJ distance 1 from some genome π and σ respectively, the size of o is at most 2. So, in our special case, calculating the distance between every pair takes $O(g + |A| \cdot |B|)$ time.

We implemented this efficient distance calculation in the PIVO2 algorithm. This way, in comparison with the original PIVO, the time complexity of the PIVO2 algorithm has decreased from $O(ngk^2)$ to $O(n(g+k^2))$, where k is the maximal size of a candidate set, $n = |V|$ is the tree size, and g is the number of genes. Since $g^2 \gg n$, therefore $k = O(g^2)$ and so the time complexity is decreased from $O(ng^5)$ to $O(ng^4)$.

Chapter 4

Experiments

This chapter contains the comparison of the original PIVO and PIVO2 algorithms. Here, it is ascertained that the PIVO2 algorithm gives better results and runs more efficiently.

4.1 Comparing the Results

In this section, we evaluate the evolutionary histories found by the original PIVO and PIVO2 algorithms. The scores of the evolutionary histories are compared. An evolutionary history is better, if its score is lower. Both algorithms were run on real as well as simulated data.

4.1.1 Real Data

The Campanulaceae cpDNA dataset

This is a well studied dataset which consists of 13 Campanulaceae chloroplast genomes [21]. Each genome has 105 genes and consists of a single circular chromosome. The phylogenetic tree of these genomes was reconstructed by Bourque and Pevzner using the MGR software [17], see figure 4.1a.

The reconstruction of ancestral genomes of these species, based on genome rearrangements using the DCJ model, was studied in several earlier works, see table 4.1.

In 2008, Adam and Sankoff found an evolutionary history with score 64 in which each ancestral genome consisted of a single circular chromosome. In case of arbitrary ancestral genomes, they could find an evolutionary history with score 59. [16]

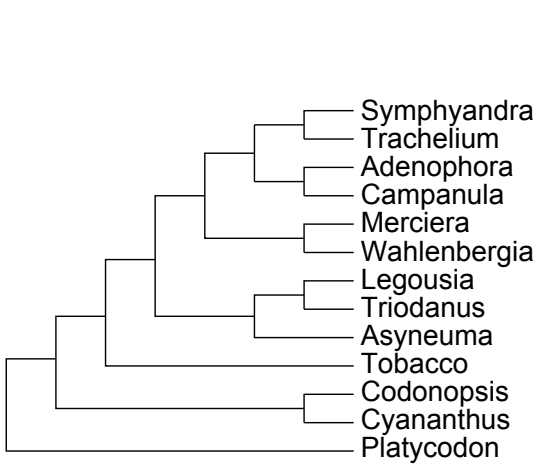
Histories with lower scores were found by Xu and Moret in 2011. Using their software GASTS, they found an evolutionary history with score 63, in which each ancestral genome consisted of a single circular chromosome. [22]

Using the original PIVO algorithm, Jakub Kováč has lowered this score to 59. [1] [23]

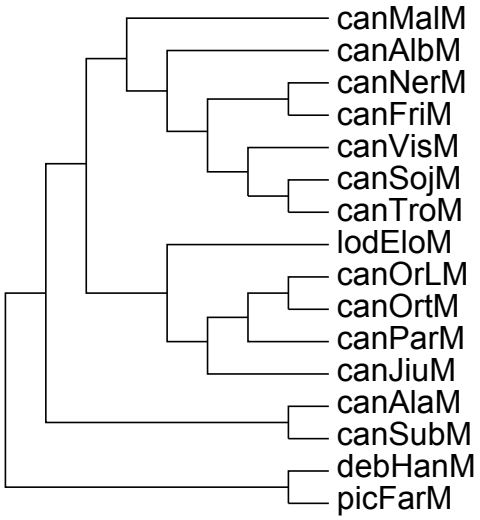
With single chromosomal ancestral genomes, the PIVO2 algorithm also found an evolutionary history with score 59. In the case of arbitrary ancestral genomes, the PIVO2 algorithm found an evolutionary history with score 56.

software	unichromosomal genome	arbitrary genome
ABC (Adam and Sankoff, 2008) [16]	64	59
GASTS (Xu and Moret, 2011) [22]	63	-
original PIVO (Kovac et al., 2011a) [1] [23]	59	59
The PIVO2 algorithm	**59**	**56**

Table 4.1: Scores for the Campanulaceae dataset



(a) Phylogenetic tree of Campanulaceae



(b) Phylogenetic tree of Hemiascomycetes

Figure 4.1: Phylogenetic trees used in the tests

The Hemiascomycetes mtDNA dataset

This dataset contains 16 mitochondrial genomes of pathogenic yeasts from the CTG clade of Hemiascomycetes [18]. Using software MrBayes [24], the phylogenetic tree of these genomes was calculated from protein sequences, see figure 4.1b. Each genome has 25 genes, and the genomes have various structures: some consist of a single linear chromosome, some other genomes consist of two linear chromosomes, and the rest of the genomes consist of a single circular chromosome. The genomes for

some of the species were not known exactly - several possibilities existed. Therefore, for these species, alternative genomes were used in the tests.

Due to this variability in genome structure 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 [1].

The PIVO2 algorithm found a better evolutionary history with score 77. In the case of ancestral genomes with arbitrary structure, an evolutionary history with score 75 was found by the PIVO2 algorithm (no result was published for the original PIVO algorithm).

software	restricted genome	arbitrary genome
original PIVO (Kovac et al., 2011a) [1]	78	-
The PIVO2 algorithm	**77**	**75**

Table 4.2: Scores for the Hemiascomycetes dataset

4.1.2 Experimental Data

Using experimental data, we also tested both the original and the PIVO2 algorithms. The experimental data were created for the phylogenetic tree of the Campanulaceae (figure 4.1a). A random genome consisting of 25 genes was generated. This genome was put into the root of the tree and a random evolution on the tree was simulated. 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 the random evolution, were used as the input for both algorithms.

The tests were made under various circumstances. In the simulations, we changed the parameter which limited the number of rearrangement operations which could happen between a parent and its child. In those simulations, for which a high number of mutations was allowed, the simulated evolution was probably not parsimonious. We expect, that in these cases, even better evolutionary histories will be found in our tests than the simulated evolutionary history.

We launched the original and the PIVO2 algorithms many times, and calculated the distribution of scores of the found evolutionary histories. We illustrated the results in several charts. The results of the PIVO2 algorithm are in red, the results of the original PIVO algorithm are in blue.

In the first test case, the number of allowed mutations on an edge was limited to 3. The score of the simulated history was 51. In all runs the PIVO2 algorithm

found histories with score 50, which is better than the score of the simulated history. The original PIVO algorithm produced worse results, the best score was achieved in only a few from many runs, and the best history found had a score of 52, as seen in figure 4.2.

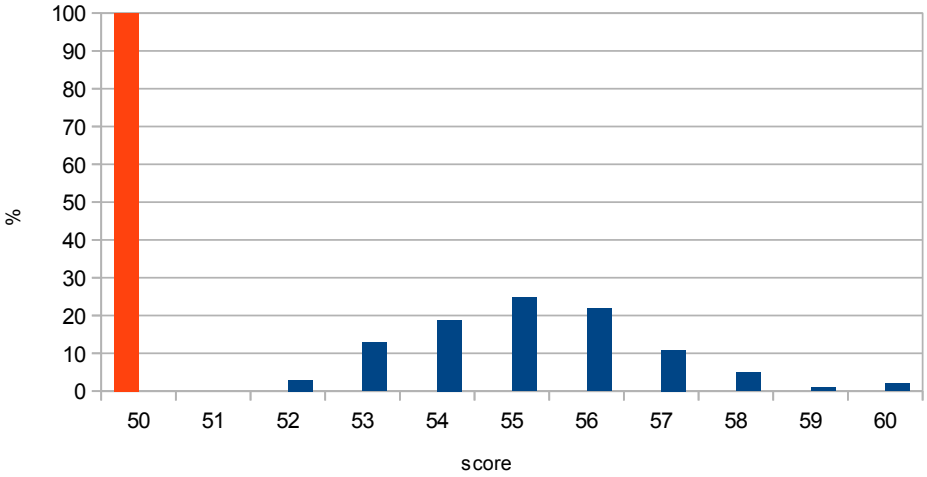


Figure 4.2: Score distribution in test case 1. (blue: original PIVO, red: PIVO2)

In the second test case, the number of allowed mutations on an edge was again limited to a maximum of 3, the score of the simulated history was 45. Both the original PIVO and the PIVO2 algorithms found histories with score 45. However, as seen in figure 4.3, the PIVO2 algorithm found a history with score 45 in every run, while the original PIVO algorithm was successful only in a very low number of runs.

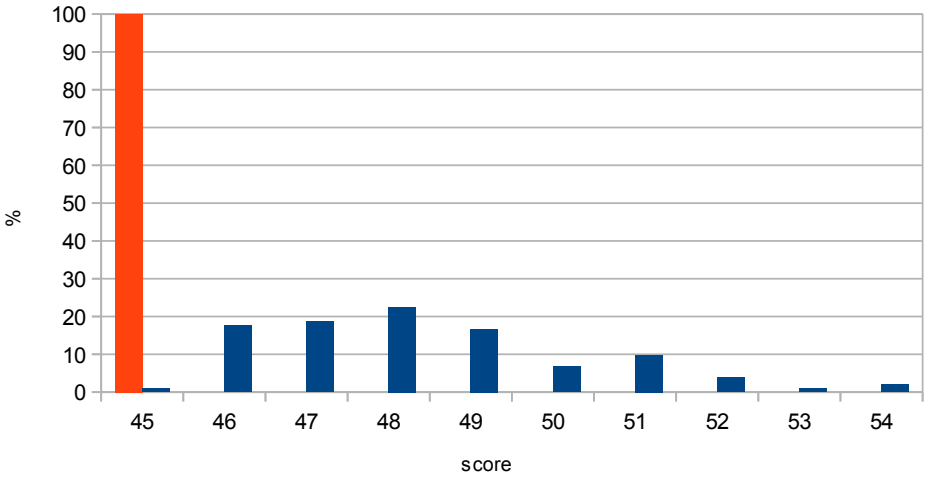


Figure 4.3: Score distribution in test case 2. (blue: original PIVO, red: PIVO2)

In the third test case, a maximum of 8 mutations was allowed on an edge, the simulated history had a score of 77. The PIVO2 algorithm found a more parsimonious

history with a score of 73, whereas the best history found by the original PIVO algorithm had a score of 78, as seen in figure 4.4.

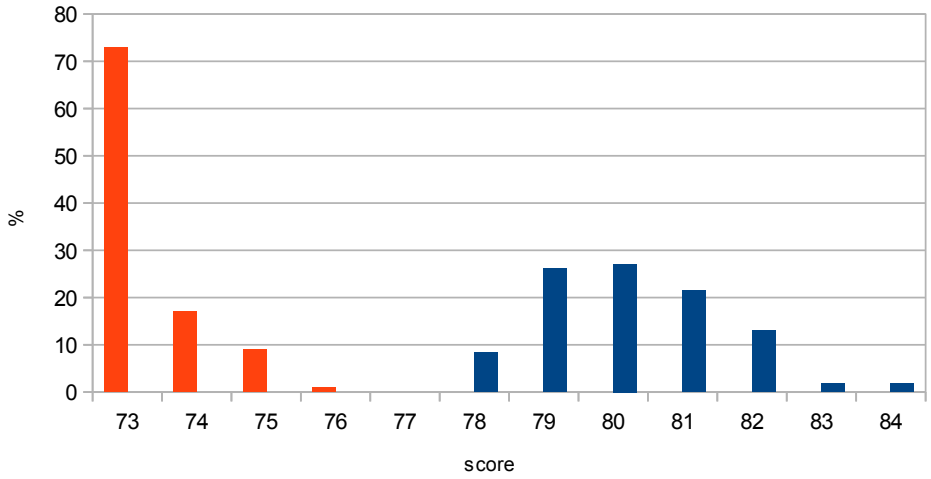


Figure 4.4: Score distribution in test case 3. (blue: original PIVO, red: PIVO2)

The fourth test case was similar: the score of the simulated evolution was 87. The PIVO2 algorithm was again better than the original PIVO algorithm, see figure 4.5.

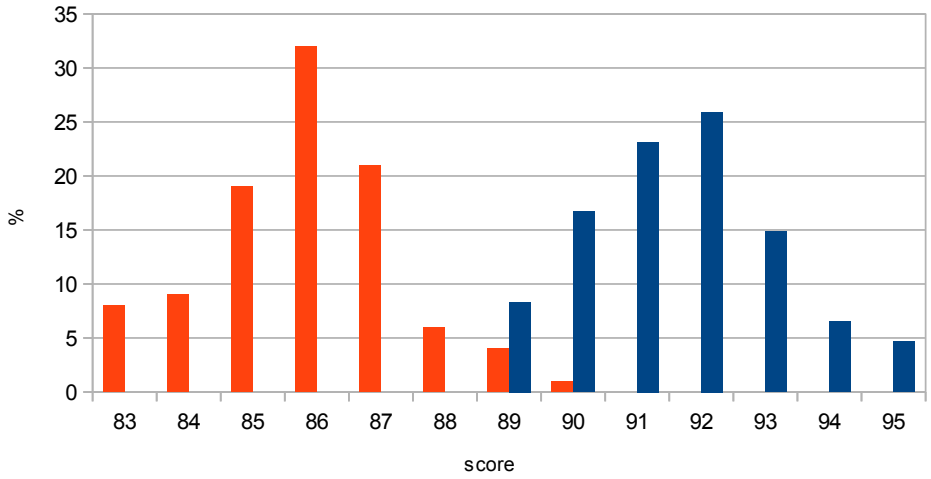


Figure 4.5: Score distribution in test case 4. (blue: original PIVO, red: PIVO2)

Finally, in the fifth test case, the simulated evolution contained even more mutations, and it had a score of 103. The comparison of the algorithms is in figure 4.6.

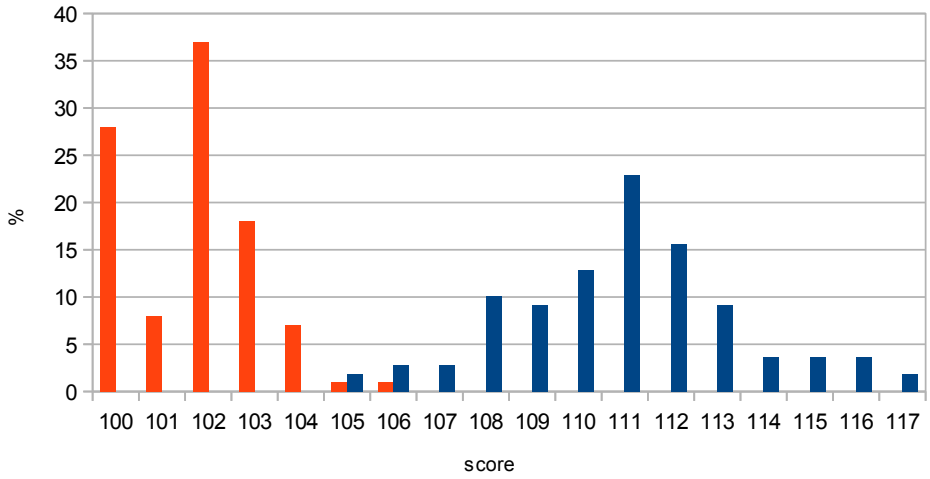


Figure 4.6: Score distribution in test case 5. (blue: original PIVO, red: PIVO2)

Test summary: In every test case, the results achieved by the PIVO2 algorithm were better than those of the original PIVO algorithm.

4.2 Number of iterations

In this section, we examine, how many iterations are needed until a local optimum is found by the original PIVO and PIVO2 algorithms. If many iterations are needed, the score of the history is only slowly improving towards a local optimum, and so the tested algorithm is slow. We used the same test cases as given in section 4.1.2 and computed the average number of iterations needed for both algorithms. The results can be seen in table 4.3.

	original PIVO	PIVO2
test case 1	5.2	6.49
test case 2	5.34	5.82
test case 3	4.4	6.7
test case 4	5.14	7.57
test case 5	5.29	8.58

Table 4.3: Comparison of the average number of iterations needed to reach a local optimum

We can see, that the PIVO2 algorithm needs a slightly higher number of iterations. However, this is amply justified by the fact, that the PIVO2 algorithm gives better results, than the original PIVO. For test case 5, we have created a chart which illustrates

the decrease of score with the increasing number of iterations. We calculated the average score, the minimum score, and the maximum score, which were achieved in each iteration, see figure 4.7. The red line symbolizes the average score and the light red dotted lines symbolize the minimum and maximum scores of the PIVO2 algorithm. The interval between the red minimum and maximum lines illustrates what scores could be achieved in the given iteration. Similarly, for the original PIVO algorithm, the blue line represents the average score, and the light blue dotted lines are the minimum and maximum scores.

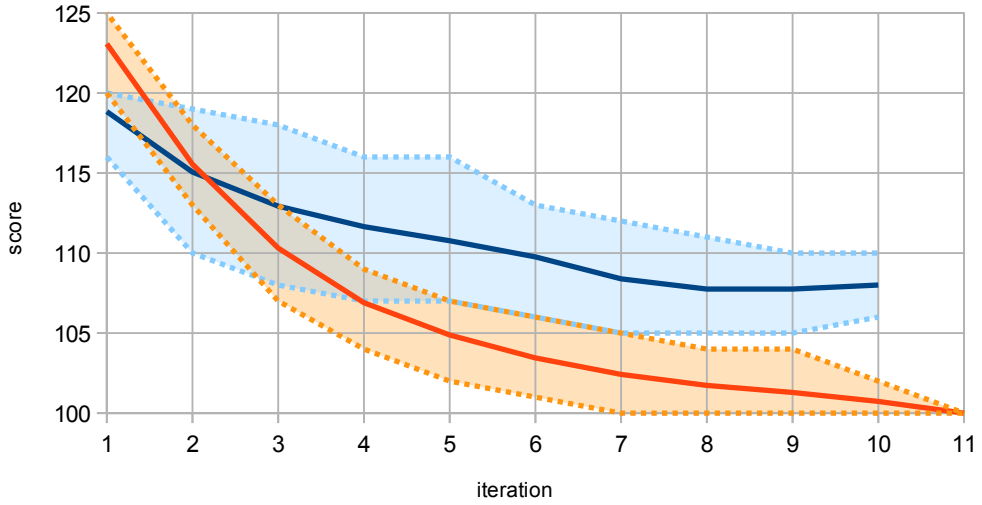


Figure 4.7: Decrease of score with the increasing number of iterations. (blue: original PIVO, red: PIVO2)

4.3 Speed comparison

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 algorithms meaningfully. According to our measurements, PIVO2 algorithm (with efficient distance calculation mode switched off) is approximately 6-7 times faster than the original PIVO algorithm. As we will show next, PIVO2 runs even faster in the efficient distance calculation mode.

Similarly to section 4.1.2, some test cases were created, but, in these test cases, 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 inner nodes. The results are in table 4.4. The chart in figure 4.8 illustrates, how many times is the computation faster with the efficient distance calculation mode switched on.

number of genes	average candidate set size	t_{OFF} efficient distance calculation off	t_{ON} efficient distance calculation on	$r = \frac{t_{OFF}}{t_{ON}}$
15	340	293 ms	304 ms	0.96
20	600	1 259 ms	913 ms	1.38
25	840	3 136 ms	1 991 ms	1.58
30	1 280	7 713 ms	4 293 ms	1.80
35	1 830	20 406 ms	7924 ms	2.58
40	2 300	36 067 ms	13 117 ms	2.75
45	3 210	84 007 ms	21 981 ms	3.82
50	3 770	120 737 ms	30 998 ms	3.89
55	4 750	220 329 ms	47 126 ms	4.68
60	5 170	284 145 ms	63 659 ms	4.46
65	7 000	549 698 ms	96 794 ms	5.68

Table 4.4: Comparison of the speeds of the PIVO2 algorithm with efficient distance calculation mode off and on

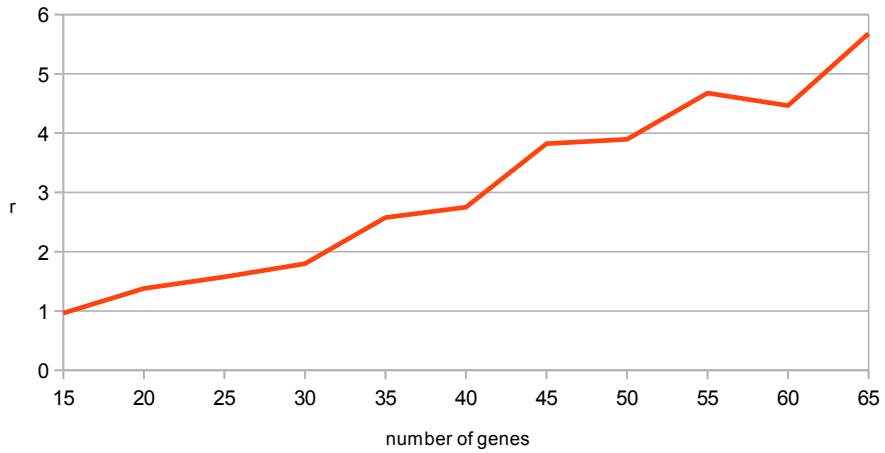


Figure 4.8: PIVO2: The increase of $r = \frac{t_{OFF}}{t_{ON}}$ with the increasing number of genes

Conclusion

In this thesis, we studied the reconstruction of ancestral gene orders by the PIVO algorithm, which uses an iterative local optimization procedure [1].

We have identified various areas of possible improvements and created an improved algorithm, called PIVO2. The improvements in PIVO2 are based on using different initialization and candidate generation methods. However, randomization of candidate selection is perhaps the most significant area of improvement. The PIVO2 software contains several added extensions, which can be optionally turned on.

To increase the speed of the original PIVO algorithm, we have also designed and implemented a new method of breakpoint and DCJ distance calculation, applicable when we calculate distances for many 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.

The original PIVO algorithm was written in Python; the reimplmented PIVO2 is in Java.

To evaluate the improvements, we carried out some practical tests and comparisons. The tests show that the PIVO2 algorithm is indeed better than the original PIVO algorithm in finding evolutionary histories with lower score. Moreover, to find a history with a good score, the PIVO2 algorithm requires a significantly lower number of runs. The tests also confirm that the new distance calculation method applied in PIVO2 really increases the computational speed.

Recommendations for further research

When using the efficient distance calculation method between two sets of genomes A and B (see chapter 3), we are given two genomes - π and σ. For each $\pi_i \in A$ and each $\sigma_j \in B$, we know the operations which transform π into π_i and σ into σ_j respectively (this information was saved during candidate generation). Enabling distance calculation on arbitrary sets of genomes A and B can be a useful extension of the efficient distance calculation method. The extension should make the following precalculation steps:

- automatically find groups of similar genomes in the sets A and B,
- select a representative genome for each group,
- calculate the transforming operations which regenerate each member of the group from the representative genome.

Based on these precalculation steps, the efficient distance calculation method should work for arbitrary sets of genomes A and B. The precalculation procedure should be further researched in the future.

Bibliography

[1] J Kováč, B Brejová, and T Vinař. A practical algorithm for ancestral rearrangement reconstruction. *Algorithms in Bioinformatics*, 2011.

[2] International Human Sequencing Genome. Initial sequencing and analysis of the human genome. *Nature*, 409(6822):860–921, 2001.

[3] Kirsten M Brown, Lisa M Burk, Loren M Henagan, and Mohamed A F Noor. A test of the chromosomal rearrangement model of speciation in Drosophila pseudoobscura. *Evolution; international journal of organic evolution*, 58:1856–1860, 2004.

[4] Guillaume Fertin, Anthony Labarre, Irena Rusu, Eric Tannier, and Stéphane Vialette. *Combinatorics of genome rearrangements*. 2009.

[5] Eric Tannier, Chunfang Zheng, and David Sankoff. Multichromosomal median and halving problems under different genomic distances. *BMC bioinformatics*, 10:120, January 2009.

[6] Anne Bergeron, Julia Mixtacki, and Jens Stoye. A Unifying View of Genome Rearrangements. *Wabi*, pages 163–173, 2006.

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

[8] Li-San Wang, Tandy Warnow, Bernard M E Moret, Robert K Jansen, and Linda A Raubeson. Distance-based genome rearrangement phylogeny. *Journal of molecular evolution*, 63:473–483, 2006.

[9] David Sankoff and Mathieu Blanchette. The median problem for breakpoints in comparative genomics. In *Proceedings of the Third Annual International Conference on Computing and Combinatorics (COCOON 97)*, volume 1276, pages 251–263, 1997.

[10] Sophia Yancopoulos, Oliver Attie, and Richard Friedberg. Efficient sorting of genomic permutations by translocation, inversion and block interchange. *Bioinformatics (Oxford, England)*, 21:3340–3346, 2005.

[11] Jakub Kováč. On the Complexity of Rearrangement Problems under the Breakpoint Distance. *Journal of computational biology : a journal of computational molecular cell biology*, 21:1–15, 2014.

[12] M Blanchette, G Bourque, and D Sankoff. Breakpoint Phylogenies. *Genome informatics. Workshop on Genome Informatics*, 8:25–34, 1997.

[13] Stacia Wyman, David A Bader, and Tandy Warnow. A New Implementation and Detailed Study of Breakpoint Analysis Bernard M.E. Moret.

[14] B M Moret, L S Wang, T Warnow, and S K Wyman. New approaches for reconstructing phylogenies from gene order data. *Bioinformatics (Oxford, England)*, 17 Suppl 1:S165–73, January 2001.

[15] Bernard M.E. Moret, Jijun Tang, Li-San Wang, and Tandy Warnow. Steps toward accurate reconstructions of phylogenies from gene-order data. *Journal of Computer and System Sciences*, 65(3):508–525, November 2002.

[16] Zaky Adam and David Sankoff. The ABCs of MGR with DCJ. *Evolutionary bioinformatics online*, 4:69–74, 2008.

[17] Guillaume Bourque and Pavel a Pevzner. Genome-scale evolution: reconstructing gene orders in the ancestral species. *Genome research*, 12(1):26–36, January 2002.

[18] Matus Valach, Zoltan Farkas, Dominika Fricova, Jakub Kovac, Brona Brejova, Tomas Vinar, Ilona Pfeiffer, Judit Kucsera, Lubomir Tomaska, B Franz Lang, and Jozef Nosek. Evolution of linear chromosomes and multipartite genomes in yeast mitochondria. *Nucleic acids research*, 39(10):4202–4219, May 2011.

[19] Fred Glover. Tabu Search - Part I. *ORSA Journal on Computing*, 1:190–206, 1989.

[20] Fred Glover. Tabu Search - Part II. *ORSA journal on Computing*, 2 1:4–32, 1989.

[21] M.E. Cosner, R.K. Jansen, B.M.E. Moret, L.A. Raubeson, L.S. Wang, T Warnow, and SK Wyman. An empirical comparison of phylogenetic methods on chloroplast gene order data in Campanulaceae. In *Comparative Genomics: Empirical and*

Analytical Approaches to Gene Order Dynamics, Map Alignment, and the Evolution of Gene Families, pages 99–121. 2000.

[22] Andrew Wei Xu and Bernard M E Moret. GASTS : Parsimony Scoring under Rearrangements. pages 351–363, 2011.

[23] Jakub Kováč, Broňa Brejová, and Tomáš Vinař. A New Approach to the Small Phylogeny Problem. December 2010.

[24] Fredrik Ronquist and John P Huelsenbeck. MrBayes 3: Bayesian phylogenetic inference under mixed models. *Bioinformatics (Oxford, England)*, 19:1572–1574, 2003.