Adrian-Horia Dediu
Francisco Hernández-Quiroz
Carlos Martín-Vide
David A. Rosenblueth (Eds.)

LNBI 9199

Algorithms for Computational Biology

Second International Conference, AlCoB 2015
Mexico City, Mexico, August 4–5, 2015
Proceedings

Lecture Notes in Bioinformatics 9199

Subseries of Lecture Notes in Computer Science

LNBI Series Editors

Sorin Istrail
Brown University, Providence, RI, USA
Pavel Pevzner
University of California, San Diego, CA, USA
Michael Waterman
University of Southern California, Los Angeles, CA, USA

LNBI Editorial Board

Alberto Apostolico
Georgia Institute of Technology, Atlanta, GA, USA
Søren Brunak
Technical University of Denmark Kongens Lyngby, Denmark
Mikhail S. Gelfand
IITP, Research and Training Center on Bioinformatics, Moscow, Russia
Thomas Lengauer
Max Planck Institute for Informatics, Saarbrücken, Germany
Satoru Miyano
University of Tokyo, Japan
Eugene Myers
Max Planck Institute of Molecular Cell Biology and Genetics Dresden, Germany
Marie-France Sagot
Université Lyon 1, Villeurbanne, France
David Sankoff
University of Ottawa, Canada
Ron Shamir
Tel Aviv University, Ramat Aviv, Tel Aviv, Israel
Terry Speed
Walter and Eliza Hall Institute of Medical Research Melbourne, VIC, Australia
Martin Vingron
Max Planck Institute for Molecular Genetics, Berlin, Germany
W. Eric Wong
University of Texas at Dallas, Richardson, TX, USA

More information about this series at http://www.springer.com/series/5381

Adrian-Horia Dediu · Francisco Hernández-Quiroz
Carlos Martín-Vide · David A. Rosenblueth (Eds.)

Algorithms for Computational Biology

Second International Conference, AlCoB 2015
Mexico City, Mexico, August 4–5, 2015
Proceedings

Springer

Editors
Adrian-Horia Dediu
Research Group on Mathematical Linguistics
Rovira i Virgili University
Tarragona
Spain

Carlos Martín-Vide
Research Group on Mathematical Linguistics
Rovira i Virgili University
Tarragona
Spain

Francisco Hernández-Quiroz
Faculty of Science
National Autonomous University of Mexico – UNAM
Mexico City
Mexico

David A. Rosenblueth
Institute for Research in Applied Mathematics and Systems – IIMAS
National Autonomous University of Mexico – UNAM
Mexico City
Mexico

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Bioinformatics
ISBN 978-3-319-21232-6 ISBN 978-3-319-21233-3 (eBook)
DOI 10.1007/978-3-319-21233-3

Library of Congress Control Number: 2015943051

LNCS Sublibrary: SL8 – Bioinformatics

Springer Cham Heidelberg New York Dordrecht London
© Springer International Publishing Switzerland 2015
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made.

Printed on acid-free paper

Springer International Publishing AG Switzerland is part of Springer Science+Business Media
(www.springer.com)

Preface

These proceedings contain the papers that were presented at the Second International Conference on Algorithms for Computational Biology (AlCoB 2015), held in Mexico City, Mexico, during August 4–5, 2015.

The scope of AlCoB includes topics of either theoretical or applied interest, namely:

- Exact sequence analysis
- Approximate sequence analysis
- Pairwise sequence alignment
- Multiple sequence alignment
- Sequence assembly
- Genome rearrangement
- Regulatory motif finding
- Phylogeny reconstruction
- Phylogeny comparison
- Structure prediction
- Compressive genomics
- Proteomics: molecular pathways, interaction networks
- Transcriptomics: splicing variants, isoform inference and quantification, differential analysis
- Next-generation sequencing: population genomics, metagenomics, metatranscriptomics
- Microbiome analysis
- Systems biology

AlCoB 2015 received 23 submissions. Most papers were reviewed by three and some by two Program Committee members. There were also two external reviewers consulted; we acknowledge all the reviewers in the next section. After a thorough and lively discussion phase, the committee decided to accept 11 papers (which represents an acceptance rate of 47.83 %). The conference program also included three invited talks and some presentations of work in progress.

The excellent facilities provided by EasyChair allowed us to successfully manage the conference submissions and to handle and check the proceedings preparation.

We would like to thank all invited speakers and authors for their contributions, the Program Committee and the reviewers for their cooperation, and Springer for its very professional publishing work.

April 2015

Adrian-Horia Dediu
Francisco Hernández-Quiroz
Carlos Martín-Vide
David A. Rosenblueth

Organization

AlCoB 2015 was organized by the Center for Complexity Sciences–C3, School of Sciences, Institute for Research in Applied Mathematics and Systems–IIMAS, Graduate Program in Computing Science and Engineering from National Autonomous University of Mexico–UNAM, and the Research Group on Mathematical Linguistics–GRLMC, from Rovira i Virgili University, Tarragona.

Program Committee

Stephen Altschul	National Center for Biotechnology Information, Bethesda, USA
Yurii Aulchenko	Russian Academy of Sciences, Novosibirsk, Russia
Pierre Baldi	University of California, Irvine, USA
Daniel G. Brown	University of Waterloo, Canada
Yuehui Chen	University of Jinan, China
Keith A. Crandall	George Washington University, Washington, USA
Joseph Felsenstein	University of Washington, Seattle, USA
Michael Galperin	National Center for Biotechnology Information, Bethesda, USA
Susumu Goto	Kyoto University, Japan
Igor Grigoriev	DOE Joint Genome Institute, Walnut Creek, USA
Martien Groenen	Wageningen University, The Netherlands
Yike Guo	Imperial College, London, UK
Javier Herrero	University College London, UK
Karsten Hokamp	Trinity College Dublin, Ireland
Hsuan-Cheng Huang	National Yang-Ming University, Taipei, Taiwan
Ian Korf	University of California, Davis, USA
Nikos Kyrpides	DOE Joint Genome Institute, Walnut Creek, USA
Mingyao Li	University of Pennsylvania, Philadelphia, USA
Jun Liu	Harvard University, Cambridge, USA
Rodrigo López	European Bioinformatics Institute, Hinxton, UK
Andrei N. Lupas	Max Planck Institute for Developmental Biology, Tübingen, Germany
B.S. Manjunath	University of California, Santa Barbara, USA
Carlos Martín-Vide (Chair)	Rovira i Virgili University, Tarragona, Spain
Tarjei Mikkelsen	Broad Institute, Cambridge, USA
Henrik Nielsen	Technical University of Denmark, Lyngby, Denmark
Zemin Ning	Wellcome Trust Sanger Institute, Hinxton, UK
Christine Orengo	University College London, UK
Modesto Orozco	Institute for Research in Biomedicine, Barcelona, Spain

Christos A. Ouzounis	Center for Research & Technology Hellas, Thessaloniki, Greece
Manuel Peitsch	Philip Morris International R&D, Neuchâtel, Switzerland
David A. Rosenblueth	National Autonomous University of Mexico, Mexico City, Mexico
Julio Rozas	University of Barcelona, Spain
Alessandro Sette	La Jolla Institute for Allergy and Immunology, USA
Peter F. Stadler	University of Leipzig, Germany
Guy Theraulaz	Paul Sabatier University, Toulouse, France
Alfonso Valencia	Spanish National Cancer Research Center, Madrid, Spain
Kai Wang	University of Southern California, Los Angeles, USA
Lusheng Wang	City University of Hong Kong, Hong Kong, SAR China
Zidong Wang	Brunel University, Uxbridge, UK
Harel Weinstein	Cornell University, New York, USA
Jennifer Wortman	Broad Institute, Cambridge, USA
Jun Yu	Chinese Academy of Sciences, Beijing, China
Mohammed J. Zaki	Rensselaer Polytechnic Institute, Troy, USA
Louxin Zhang	National University of Singapore, Singapore
Hongyu Zhao	Yale University, New Haven, USA

External Reviewers

Quan Chen
Sajeet Haridas

Organizing Committee

Adrian-Horia Dediu, Tarragona
Francisco Hernández-Quiroz, Mexico City
Carlos Martín-Vide, Tarragona (Co-chair)
David A. Rosenblueth, Mexico City (Co-chair)
Lilica Voicu, Tarragona

Contents

Genetic Processing

Generalized Hultman Numbers and the Distribution of Multi-break Distances 3
Nikita Alexeev, Anna Pologova, and Max A. Alekseyev

Implicit Transpositions in Shortest DCJ Scenarios 13
Shuai Jiang and Max A. Alekseyev

Constraint-Based Genetic Compilation 25
Christophe Ladroue and Sara Kalvala

Molecular Recognition/Prediction

P2RANK: Knowledge-Based Ligand Binding Site Prediction Using Aggregated Local Features 41
Radoslav Krivák and David Hoksza

StreaM - A Stream-Based Algorithm for Counting Motifs in Dynamic Graphs 53
Benjamin Schiller, Sven Jager, Kay Hamacher, and Thorsten Strufe

Convolutional LSTM Networks for Subcellular Localization of Proteins 68
Søren Kaae Sønderby, Casper Kaae Sønderby, Henrik Nielsen, and Ole Winther

Phylogenetics

Hybrid Genetic Algorithm and Lasso Test Approach for Inferring Well Supported Phylogenetic Trees Based on Subsets of Chloroplastic Core Genes 83
Bassam AlKindy, Christophe Guyeux, Jean-François Couchot, Michel Salomon, Christian Parisod, and Jacques M. Bahi

Constructing and Employing Tree Alignment Graphs for Phylogenetic Synthesis 97
Ruchi Chaudhary, David Fernández-Baca, and J. Gordon Burleigh

A More Practical Algorithm for the Rooted Triplet Distance 109
Jesper Jansson and Ramesh Rajaby

Likelihood-Based Inference of Phylogenetic Networks from Sequence Data by PhyloDAG 126
Quan Nguyen and Teemu Roos

Constructing Parsimonious Hybridization Networks from Multiple Phylogenetic Trees Using a SAT-Solver 141
Vladimir Ulyantsev and Mikhail Melnik

Author Index 155

Genetic Processing

Generalized Hultman Numbers and the Distribution of Multi-break Distances

Nikita Alexeev[1(✉)], Anna Pologova[2], and Max A. Alekseyev[1]

[1] George Washington University, Washington, DC, USA
{nikita_alexeev,maxal}@gwu.edu
[2] St. Petersburg State University, St. Petersburg, Russia

Abstract. Genome rearrangements can be modeled by k-breaks, which break a genome at k positions and glue the resulting fragments in a new order. In particular, reversals, translocations, fusions, and fissions are modeled as 2-breaks, and transpositions are modeled as 3-breaks. While k-break rearrangements for $k > 3$ have not been observed in evolution, they are used in cancer genomics to model chromothripsis, a catastrophic event of multiple breakages happening in a genome simultaneously.

It is known that the k-break distance between two genomes (i.e., the minimal number of k-breaks needed to transform one genome into the other) can be computed in terms of cycle lengths of the breakpoint graph of these genomes. In the current work, we address the combinatorial problem of enumeration of genomes at a given k-break distance from a fixed genome. More generally, we enumerate genome pairs, whose breakpoint graph has a fixed distribution of cycle lengths.

1 Introduction

Genome rearrangements are evolutionary events that change gene order along the genome. The genome rearrangements can be modeled as *k-breaks* [1], which break a genome at k positions and glue the resulting fragments in a new order. While most frequent genome rearrangements such as *reversals* (that flip segments of a chromosome), *translocations* (that exchange segments of two chromosomes), *fusions* (that merge two chromosomes into one), and *fissions* (that split a single chromosome into two) can be modeled as 2-breaks (also called *Double-Cut-and-Join*, *DCJ* [2]), more complex and rare genome rearrangements such as transpositions are modeled as 3-breaks. While k-break rearrangements for $k > 3$ were not observed in evolution, they are used in cancer genomics to model *chromothripsis*, a catastrophic event, when multiple breakages happen simultaneously [3,4]. The minimal number of k-breaks required to transform one genome into another is called *k-break distance*. In particular, the 2-break (DCJ) distance is often used in phylogenomic studies to estimate evolutionary remoteness of genomes.

In the current work, we address the combinatorial problem of enumeration of genomes at a given k-break distance from a fixed genome. There are two variations of this problem, depending on whether all genes in the two genomes have

© Springer International Publishing Switzerland 2015
A.-H. Dediu et al. (Eds.): AlCoB 2015, LNBI 9199, pp. 3–12, 2015.
DOI: 10.1007/978-3-319-21233-3_1

the same orientation, in which case they can be encoded by a permutation with positive (*unsigned*) elements, while in the general case the permutation elements may have various signs. Previous studies are mostly concerned with the case of 2-break distance between unichromosomal genomes. In the unsigned case, the number of such genomes is given by a Hultman number [5]. Doignon et al. [6] gave a closed formula for Hultman numbers, Bóna and Flynn [7] proved the relation between Hultman numbers and Stirling numbers of the first kind. The general case of 2-break distance was solved by Grusea and Labarre [8]. The asymptotic distribution of the 2-break distances is proved to be normal by Alexeev and Zograf [9]. The multichromosomal analog of Hultman numbers was recently studied by Feijão et al. [10].

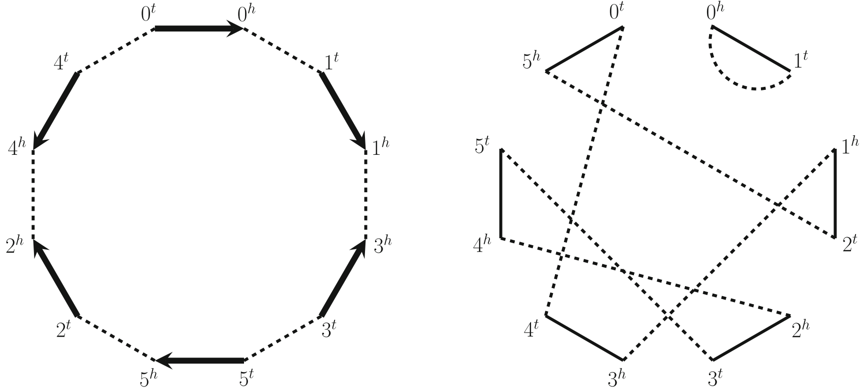
It is known that the k-break distance between two genomes can be computed in terms of cycles in the breakpoint graph of these genomes [1]. Namely, while the 2-break distance depends only on the number of cycles in this graph, the k-break distance for the general k depends on the distribution of the cycle lengths. In the current work, we enumerate unichromosomal genome pairs, whose breakpoint graph has a fixed distribution of cycle lengths.

2 Preliminary Definitions and Results

We restrict our attention to (linear) unichromosomal genomes and assume that we are given a fixed genome P with n genes. Then a unichromosomal genome Q on the same set of n genes can be encoded by a permutation π of length n. Namely, the genes are enumerated in the order defined by genome P, and their order in genome Q defines π. Each element of π may be positive or negative, depending on whether the orientation (strand) of the gene i in Q is opposite to its orientation in P. To obtain a natural generalization of Hultman numbers, we will separately address the case of the *unsigned permutation* π with all elements being positive. We denote by S_n and $S_n^{\pm}$ the set of all unsigned and signed, respectively, permutations of length n.

We represent a unichromosomal genome with n genes as a *genome graph*. This graph contains $2n+2$ vertices: for each gene $i \in \{1, 2, \dots, n\}$, there are the *tail* and *head* vertices i^t and i^h, and in addition there are two vertices 0^t, 0^h corresponding to a virtual gene 0. The graph has $n+1$ directed *gene edges*, encoding n genes and the virtual 0-gene, and $n+1$ undirected *adjacency edges* encoding gene adjacencies, where the virtual 0-gene is used to circularize the graph (Fig. 1a).

Let P and Q be a pair of genomes, encoded by a signed permutation π. We assume that in their genome graphs the adjacency edges of P are colored black and the adjacency edges of Q are colored gray. The *breakpoint graph* $G(\pi) = G(P, Q)$ is defined on the set of vertices $\{i^t, i^h | i \in \{0, 1, \dots, n\}\}$ with black and gray edges inherited from genome graphs of P and Q (Fig. 1b). Since each vertex in $G(P, Q)$ has degree 2, the black and gray edges form a collection of alternating black-gray cycles. We say that a black-gray cycle is an *ℓ-cycle* if it contains ℓ black edges (and ℓ gray edges) and denote the number of ℓ-cycles in $G(P, Q)$ by



(a) Genome graph of genome $Q = (1, -3, 5, 2, -4)$.

(b) Breakpoint graph $G(P, Q)$.

Fig. 1. Breakpoint graph $G(P, Q)$ corresponding to a signed permutation $\pi = (1, -3, 5, 2, -4)$. Adjacency edges of P are black (solid) and adjacency edges of Q are gray (dashed). It consists of one 1-cycle, one 2-cycle, and one 3-cycle.

$c_\ell(P, Q)$. We note that if π is a signed permutation of length n, then the total number of black edges equals to

$$\sum_{\ell=1}^{\infty} \ell \cdot c_\ell = n + 1 \,.$$

A k-break on genome Q can be viewed as an operation on its genome graph and the breakpoint graph $G(P, Q)$. Namely, a k-break replaces any k-tuple of gray edges with another k-tuple of gray edges, which forms a perfect matching on the same set of $2k$ vertices (Fig. 2). A transformation of genome Q into genome P with k-breaks then can be viewed as a transformation of the breakpoint graph $G(P, Q)$ into the breakpoint graph $G(P, P)$ with k-breaks on gray edges. The *k-break distance* $d_k(P, Q)$ between genomes P and Q is the minimum number of k-breaks in such a transformation.

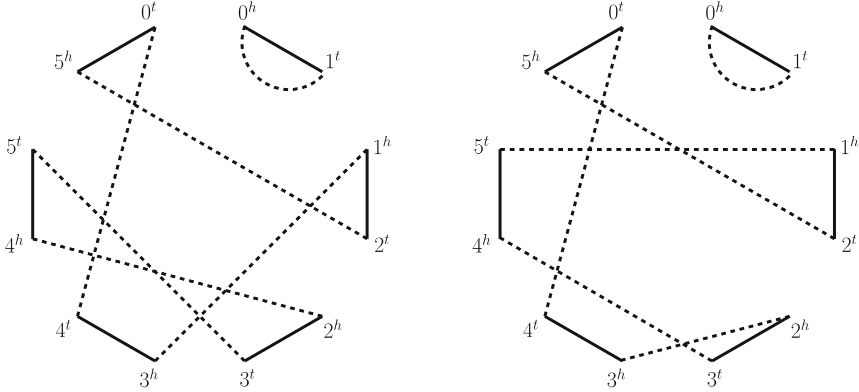
The 2-break distance between genomes P and Q is given by the following formula [2]:

$$d_2(P, Q) = n + 1 - c(P, Q) \,,$$

where $c(P, Q) = \sum_{\ell=1}^{n+1} c_\ell(P, Q)$ is the total number of cycles in $G(P, Q)$. Formulae for k-break distance for $k > 2$ is more sophisticated. In particular, $d_3(P, Q)$ and $d_4(P, Q)$ are given by the following formulae [1]:

$$d_3(P, Q) = \frac{n + 1 - c^{2,1}(P, Q)}{2} \,, \tag{1}$$

$$d_4(P, Q) = \left\lceil \frac{n + 1 - c^{3,1}(P, Q) - \lfloor c^{3,2}(P, Q)/2 \rfloor}{3} \right\rceil \,, \tag{2}$$



(a) Breakpoint graph $G(1, -3, 5, 2, -4)$ (b) Breakpoint graph $G(1, 5, 2, -3, -4)$

Fig. 2. The 3-break changes gray edges $(1^h, 3^h), (2^h, 4^h)$ and $(3^t, 5^t)$ in the breakpoint graph $G(1, -3, 5, 2, -4)$ to edges $(1^h, 5^t), (2^h, 3^h)$ and $(3^t, 4^h)$. The result is the breakpoint graph $G(1, 5, 2, -3, -4)$.

where

$$c^{m,i}(P, Q) = \sum_{\ell \equiv i \pmod m} c_\ell(P, Q) \,.$$

Our goal is to compute the number of permutations, whose breakpoint graphs consist of c_ℓ ℓ-cycles (for $\ell \in \{1, 2, \dots\}$). As an application, it will allow to find the distribution of k-break distances for $k > 2$.

3 Permutations with a Fixed Breakpoint Graph

Let $\mathbf{c} = (c_1, c_2, c_3, \dots)$ be sequence of nonnegative integers with a finite number of nonzero (i.e., strictly positive) terms. Then $L(\mathbf{c}) = \sum_{\ell=1}^{\infty} \ell \cdot c_\ell$ represents a finite integer. Whenever a breakpoint graph has c_ℓ ℓ-cycles for each $\ell \in \mathbb{N}$, we say that it has *cycle structure* $\mathbf{c}$.

We denote by $M_n(\mathbf{c})$ (respectively, $M_n^{\pm}(\mathbf{c})$) the number of unsigned (respectively, signed) permutations of length n, whose breakpoint graphs have cycle structure $\mathbf{c}$. Clearly, we have $M_n(\mathbf{c}) = M_n^{\pm}(\mathbf{c}) = 0$ unless $L(\mathbf{c}) = n + 1$. For each n, we define the generating function of numbers $M_n(\mathbf{c})$ by

$$F_n(s_1, s_2, \dots) = \sum_{\mathbf{c}: L(\mathbf{c}) = n+1} M_n(\mathbf{c}) \prod_{i=1}^{\infty} s_i^{c_i} \,.$$

Similarly, we define the generating function of numbers $M_n^{\pm}(\mathbf{c})$ by

$$F_n^{\pm}(s_1, s_2, \dots) = \sum_{\mathbf{c}: L(\mathbf{c}) = n+1} M_n^{\pm}(\mathbf{c}) \prod_{i=1}^{\infty} s_i^{c_i} \,.$$

It is easy to see that

$$F_n(1,1,\dots) = |S_n| = n!$$

and

$$F^{\pm}(1,1,\dots) = |S_n^{\pm}| = 2^n \cdot n!\,.$$

Furthermore, in the unsigned case by substituting $s_i = s$ for all $i = 1, 2, \dots,$ we obtain

$$F_n(s,s,\dots) = \sum_{m=1}^{n+1} H(n,m) s^m\,,$$

where $H(n,m)$ is the number of permutations in S_n, whose breakpoint graphs consist of m cycles. In particular, we have

$$H(n,m) = \sum_{\mathbf{c}\in\mathcal{C}_{n,m}} M_n(\mathbf{c})\,,$$

where $\mathcal{C}_{n,m} = \{\mathbf{c} : L(\mathbf{c}) = n+1 \text{ and } \sum_{i=1}^{n+1} c_i = m\}$.

The numbers $H(n,m)$ are known as Hultman numbers [6,7,9] and appear in the On-Line Encyclopedia of Integer Sequences (OEIS) [11] as the sequence A164652.

For the signed case a similar observation holds, namely,

$$F_n^{\pm}(s,s,\dots) = \sum_{m=1}^{n+1} H^{\pm}(n,m) s^m\,,$$

where $H^{\pm}(n,m)$ is the number of signed permutations in $S_n^{\pm}$, whose breakpoint graphs consist of m cycles. Similarly, we have

$$H^{\pm}(n,m) = \sum_{\mathbf{c}\in\mathcal{C}_{n,m}} M_n^{\pm}(\mathbf{c})\,. \tag{3}$$

The numbers $H^{\pm}(n,m)$ were introduced in [8] and form the sequence A189507 in the OEIS [11]. A few first numbers $H(n,m)$ and $H^{\pm}(n,m)$ are tabulated in Table 1.

For the general case, we prove recurrent formulae for $M_n(\mathbf{c})$.

Theorem 1. *Let $M_n(c_1, c_2, \dots)$ be the number of permutations in S_n, whose breakpoint graphs consist of c_ℓ ℓ-cycles (for $\ell \in \{1, 2, \dots\}$).*

Then $M_0(1, 0, 0, \dots) = 1$ (initial condition) and

$$nM_n(c_1,c_2,c_3,\dots) = \tag{4}$$

$$= \sum_{i=2}^{\infty}\sum_{j=1}^{i-1}(i-1)(c_{i-1}+1-\delta_{j,1}-\delta_{j,i-1})M_{n-1}(\mathbf{c}+\boldsymbol{e}_{i-1}-\boldsymbol{e}_j-\boldsymbol{e}_{i-j})+ \tag{5}$$

$$+\sum_{i=1}^{\infty}\sum_{j=1}^{i-1} j(i-j)(c_j+1)(c_{i-j}+1+\delta_{j,i-j})M_{n-1}(\mathbf{c}-\boldsymbol{e}_{i+1}+\boldsymbol{e}_j+\boldsymbol{e}_{i-j})\,, \tag{6}$$

Table 1. Hultman numbers

(a) Values of $H(n, m)$.

$n\backslash m$	1	2	3	4	5	6
0	1					
1	0	1				
2	1	0	1			
3	0	5	0	1		
4	8	0	15	0	1	
5	0	84	0	35	0	1

(b) Values of $H^{\pm}(n, m)$.

$n\backslash m$	1	2	3	4	5	6
0	1					
1	1	1				
2	4	3	1			
3	20	21	6	1		
4	148	160	65	10	1	
5	1348	1620	701	155	15	1

where δ_{ij} is Kronecker delta ($\delta_{ij} = 1$ if $i = j$ and $\delta_{ij} = 0$ otherwise), and $\boldsymbol{e}_i$ is the sequence $(0, \ldots, 1, 0, \ldots)$ with 1 in the ith place and 0 elsewhere.

Theorem 2. *Let $M_n^{\pm}(c_1, c_2, \ldots)$ be the number of permutations in $S_n^{\pm}$, whose breakpoint graphs consist of c_ℓ ℓ-cycles (for $\ell \in \{1, 2, \ldots\}$).*

Then $M_0^{\pm}(1, 0, 0, \ldots) = 1$ (initial condition) and

$$nM_n^{\pm}(c_1, c_2, c_3, \ldots) = \tag{7}$$

$$= \sum_{i=2}^{\infty} \sum_{j=1}^{i-1} (i-1)(c_{i-1} + 1 - \delta_{j,1} - \delta_{j,i-1}) M_{n-1}^{\pm}(\mathbf{c} + \boldsymbol{e}_{i-1} - \boldsymbol{e}_j - \boldsymbol{e}_{i-j}) + \tag{8}$$

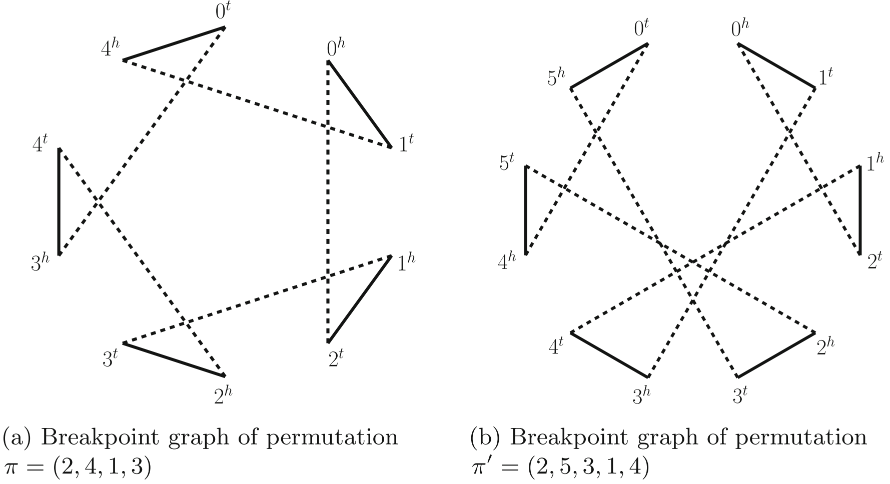
$$+ \sum_{i=2}^{\infty} (i-1)^2 (c_{i-1} + 1) M_{n-1}^{\pm}(\mathbf{c} + \boldsymbol{e}_{i-1} - \boldsymbol{e}_i) + \tag{9}$$

$$+2 \sum_{i=1}^{\infty} \sum_{j=1}^{i-1} j(i-j)(c_j + 1)(c_{i-j} + 1 + \delta_{j,i-j}) M_{n-1}^{\pm}(\mathbf{c} - \boldsymbol{e}_{i+1} + \boldsymbol{e}_j + \boldsymbol{e}_{i-j}) . \tag{10}$$

We prove both theorems using double counting (a similar technique for a different enumeration problem was used in [12]).

Proof. Let us first prove Theorem 1. We define a function Γ, which maps a triple (π, l, m), where $\pi \in S_{n-1}$ and $l, m \in \{0, 1, \ldots, n-1\}$, to a pair (π', t) with $\pi' \in S_n$ and $t \in \{1, 2, \ldots, n\}$ as follows. In the breakpoint graph $G(\pi)$, there is a black edge $e_b = (l^h, (l+1)^t)$ and a gray edge $e_g = (\pi(m)^h, \pi(m+1)^t)$ (here we assume $\pi(0) = 0$ and consider $l+1$ and $m+1$ modulo n). We delete these two edges, increase by 1 all labels greater than l, and add two new vertices $(l+1)^t$, $(l+1)^h$ along with two new black edges $(l^h, (l+1)^t)$, $((l+1)^h, (l+2)^t)$ and two new gray edges $(\pi(m)^h, (l+1)^t)$, $((l+1)^h, \pi(m+1)^t)$. The resulting graph represents the breakpoint graph of some permutation, which we call π', and we further let $t = l + 1$ (Fig. 3). It is easy to see that Γ is a bijection.

Let us analyze how the cycle structure of $G(\pi')$ may differ from the cycle structure of $G(\pi)$. There are two possibilities here. The first possibility is that edges e_b and e_g belong to the same alternating cycle in $G(\pi)$ (*splitting case*).



(a) Breakpoint graph of permutation $\pi = (2, 4, 1, 3)$

(b) Breakpoint graph of permutation $\pi' = (2, 5, 3, 1, 4)$

Fig. 3. Example: $\Gamma(\pi, 2, 2) = (\pi', 3)$

If this is a $(i-1)$-cycle, it splits into a j-cycle and a $(i-j)$-cycle for some $j \in \{1, \ldots, i-1\}$. The second possibility is that edges e_b and e_g belong to different alternating cycles in $G(\pi)$ (*merging case*). If these cycles are a j-cycle and a $(i-j)$-cycle, they merge into a single $(i+1)$-cycle.

Let us now compute the number of ways to obtain a permutation π', whose breakpoint graph $G(\pi')$ has a given cycle structure $\mathbf{c}$. The first case of splitting some $(i-1)$-cycle gives us $(i-1)(c_{i-1}+1-\delta_{j,1}-\delta_{j,i-1})M_{n-1}(\mathbf{c}+\boldsymbol{e}_{i-1}-\boldsymbol{e}_j-\boldsymbol{e}_{i-j})$, which is the first summand (5). Indeed, if $G(\pi')$ contains c_{i-1} $(i-1)$-cycles, then $G(\pi)$ contains exactly $c_{i-1}+1-\delta_{j,1}-\delta_{j,i-1}$ $(i-1)$-cycles. There are $c_{i-1}+1-\delta_{j,1}-\delta_{j,i-1}$ ways to choose an $(i-1)$-cycle and $i-1$ ways to choose a black edge in this cycle. Then the gray edge could be chosen in a unique way for each j. The second case of merging two cycles gives us the second summand (6). Indeed, there are $(c_j+1)(c_{i-j}+1+\delta_{j,i-j})$ ways to choose a j-cycle and a $i-j$-cycle in $G(\pi)$, j ways to choose a gray edge in the j-cycle, and $i-j$ ways to choose a black edge in the $(i-j)$-cycle. We remark that any permutation $\pi' \in S_n$ appears exactly n times in the image

$$\Gamma(S_{n-1} \times \{0, 1, \ldots, n-1\} \times \{0, 1, \ldots, n-1\}) = S_n \times \{1, \ldots, n\}\,,$$

which gives us the factor n in (4). Theorem 1 is proved.

Proof of Theorem 2 is similar. The only difference is that gray edges in $G(\pi)$ do not necessarily have the form $(\pi(m)^h, \pi(m+1)^t)$. This implies that when we define an analog of Γ, there are two ways to add gray edges. The merging case works equally well for both ways and so, one just has to count the summand (6) twice. In the splitting case (when e_b and e_g belong to the same cycle), the alternative way to add gray edges presents one new possibility, when the $(i-1)$-cycle does not split but just becomes an i-cycle. This gives us the summand (9). $\square$

Theorems 1 and 2 allow us to compute numbers $M_n(\mathbf{c})$. For instance, a few first values of $F_n(s_1, s_2, \dots)$ and of $F_n^{\pm}(s_1, s_2, \dots)$ are listed below:

$$\begin{aligned}
F_0(s_1, s_1, \dots) &= s_1, \\
F_1(s_1, s_2, \dots) &= s_1^2, \\
F_2(s_1, s_2, \dots) &= s_1^3 + s_3, \\
F_3(s_1, s_2, \dots) &= s_1^4 + (4s_1s_3 + s_2^2), \\
F_4(s_1, s_2, \dots) &= s_1^5 + (10s_1^2s_3 + 5s_1s_2^2) + 8s_5, \\
F_5(s_1, s_2, \dots) &= s_1^6 + (20s_1^3s_3 + 15s_1^2s_2^2) + (48s_1s_5 + 12s_3^2 + 24s_2s_4), \\
F_6(s_1, s_2, \dots) &= s_1^7 + (35s_1^3s_2^2 + 35s_1^4s_3) + \\
&\quad + (84s_1s_3^2 + 168s_1s_2s_4 + 168s_1^2s_5 + 49s_2^2s_3) + 180s_7\,.
\end{aligned}$$

$$\begin{aligned}
F_0^{\pm}(s_1, s_1, \dots) &= s_1, \\
F_1^{\pm}(s_1, s_2, \dots) &= s_1^2 + s_2, \\
F_2^{\pm}(s_1, s_2, \dots) &= s_1^3 + 3s_1s_2 + 4s_3, \\
F_3^{\pm}(s_1, s_2, \dots) &= s_1^4 + 6s_1^2s_2 + (5s_2^2 + 16s_1s_3) + 20s_4, \\
F_4^{\pm}(s_1, s_2, \dots) &= s_1^5 + 10s_1^3s_2 + (40s_1^2s_3 + 25s_1s_2^2) + (100s_1s_4 + 60s_2s_3) + 148s_5.
\end{aligned}$$

The terms are sorted in descending order of the monomial degree. According to (3), the sum of coefficients in each parenthesis is equal to the corresponding entry in Table 1. For example,

$$F_4^{\pm}(s, s, \dots) = s^5 + 10s^4 + 65s^3 + 160s^2 + 148s$$

corresponds to the row of $n = 4$ in Table 1b.

4 Generalized Hultman Numbers

Let us denote by $H_k(n, d)$ (respectively, $H_k^{\pm}(n, d)$) the number of unsigned (respectively, signed) unichromosomal genomes with n genes at the k-break distance d from a fixed genome. For $k = 2$, these numbers represent conventional and signed Hultman numbers: $H_2(n, d) = H(n, n + 1 - d)$ and $H_2^{\pm}(n, d) = H^{\pm}(n, n + 1 - d)$.

Using formulae (1) and (2), we can obtain $H_3(n, d)$ and $H_4(n, d)$. A few first numbers $H_3(n, d)$, $H_3^{\pm}(n, d)$, $H_4(n, d)$, and $H_4^{\pm}(n, d)$ are tabulated in Tables 2 and 3.

5 Discussion

In the current work, we address the problem of enumeration of genomes with n genes that are at a given k-break distance from a fixed genome. It is known

Table 2. Distribution of 3-break distance

(a) Values of $H_3(n,d)$.

$n\backslash d$	0	1	2	3	4
0	1				
1	1				
2	1	1			
3	1	4	1		
4	1	10	13		
5	1	20	75	24	
6	1	35	287	397	
7	1	56	854	3112	1017
8	1	84	2142	16196	21897

(b) Values of $H_3^{\pm}(n,d)$.

$n\backslash d$	0	1	2	3	4
0	1				
1	1	1			
2	1	7			
3	1	22	25		
4	1	50	333		
5	1	95	1851	1893	
6	1	161	6839	39079	
7	1	252	19782	323580	301505
8	1	372	48510	1706180	8566857

Table 3. Distribution of 4-break distance

(a) Values of $H_4(n,d)$.

$n\backslash d$	0	1	2	3
0	1			
1	1			
2	1	1		
3	1	5		
4	1	15	8	
5	1	35	84	
6	1	70	649	
7	1	126	3585	1328
8	1	210	14949	25160

(b) Values of $H_4^{\pm}(n,d)$.

$n\backslash d$	0	1	2	3
0	1			
1	1	1		
2	1	7		
3	1	47		
4	1	175	208	
5	1	470	3369	
6	1	1036	45043	
7	1	2002	315213	327904
8	1	3522	1472157	8846240

that the k-break distance can be computed in terms of the cycle structure of the breakpoint graph of the corresponding genomes [1]. We derive the recurrent formula for the numbers $M_n(\mathbf{c})$ and $M_n^{\pm}(\mathbf{c})$ of breakpoint graphs with a given cycle structure. Using this formula, we further compute numbers $H_k(n,d)$ of unichromosomal genomes with n genes at the k-break distance d from a fixed genome, which generalize Hultman numbers [5–9].

While 2-breaks and 3-breaks mimic most common genome rearrangements in comparative genomics, k-breaks for $k > 3$ model chromothripsis in cancer genomics. We recently used the breakpoint graph cycle structure to estimate the proportion of evolutionary 3-breaks (transpositions) between two genomes [13] and plan to develop a similar technique based on the present work to evaluate the distribution of values of k in chromothripsis k-breaks between reference and cancer genomes.

Acknowledgements. The work is supported by the National Science Foundation under the grant No. IIS-1462107. The work of NA is also partially supported by RFBR grant 13-01-12422-ofi-m.

References

1. Alekseyev, M., Pevzner, P.: Multi-break rearrangements and chromosomal evolution. Theoret. Comput. Sci. **395**(23), 193–202 (2008)
2. Yancopoulos, S., Attie, O., Friedberg, R.: Efficient sorting of genomic permutations by translocation, inversion and block interchange. Bioinformatics **21**(16), 3340–3346 (2005)
3. Stephens, P.J., Greenman, C.D., Fu, B., Yang, F., Bignell, G.R., Mudie, L.J., Pleasance, E.D., Lau, K.W., Beare, D., Stebbings, L.A., McLaren, S., Lin, M.L., McBride, D.J., Varela, I., Nik-Zainal, S., Leroy, C., Jia, M., Menzies, A., Butler, A.P., Teague, J.W., Quail, M.A., Burton, J., Swerdlow, H., Carter, N.P., Morsberger, L.A., Iacobuzio-Donahue, C., Follows, G.A., Green, A.R., Flanagan, A.M., Stratton, M.R., Futreal, P.A., Campbell, P.J.: Massive genomic rearrangement acquired in a single catastrophic event during cancer development. Cell **144**(1), 27–40 (2011)
4. Weinreb, C., Oesper, L., Raphael, B.: Open adjacencies and k-breaks: detecting simultaneous rearrangements in cancer genomes. BMC genomics **15**(Suppl 6), S4 (2014)
5. Hultman, A.: Toric permutations. Master's thesis, Department of Mathematics, KTH, Stockholm, Sweden (1999)
6. Doignon, J.P., Labarre, A.: On Hultman numbers. J. Integer Sequences **10**(6), 13 (2007)
7. Bóna, M., Flynn, R.: The average number of block interchanges needed to sort a permutation and a recent result of stanley. Inf. Process. Lett. **109**(16), 927–931 (2009)
8. Grusea, S., Labarre, A.: The distribution of cycles in breakpoint graphs of signed permutations. Discrete Appl. Math. **161**(10), 1448–1466 (2013)
9. Alexeev, N., Zograf, P.: Random matrix approach to the distribution of genomic distance. J. Comput. Biol. **21**(8), 622–631 (2014)
10. Feijão, P., Martinez, F.V., Thévenin, A.: On the multichromosomal hultman number. In: Campos, S. (ed.) BSB 2014. LNCS, vol. 8826, pp. 9–16. Springer, Heidelberg (2014)
11. The OEIS Foundation: The On-Line Encyclopedia of Integer Sequences. Published electronically at http://oeis.org (2015)
12. Alexeev, N., Andersen, J., Penner, R., Zograf, P.: Enumeration of chord diagrams on many intervals and their non-orientable analogs (2013). arXiv preprint arXiv:1307.0967
13. Alexeev, N., Aidagulov, R., Alekseyev, M.A.: A computational method for the rate estimation of evolutionary transpositions. In: Ortuño, F., Rojas, I. (eds.) IWBBIO 2015, Part I. LNCS, vol. 9043, pp. 471–480. Springer, Heidelberg (2015)

Implicit Transpositions in Shortest DCJ Scenarios

Shuai Jiang and Max A. Alekseyev(✉)

George Washington University, Washington, DC, USA
maxal@gwu.edu

Abstract. Genome rearrangements are large-scale evolutionary events that shuffle genomic architectures. The minimal number of such events between two genomes is often used in phylogenomic studies to measure the evolutionary distance between the genomes. Double-Cut-and-Join (DCJ) operations represent a convenient model of most common genome rearrangements (reversals, translocations, fissions, and fusions), while other genome rearrangements, such as transpositions, can be modeled by pairs of DCJs. Since the DCJ model does not directly account for transpositions, their impact on DCJ scenarios is unclear.

In the current work, we study implicit appearance of transpositions (as pairs of DCJs) in shortest DCJ scenarios and prove uniform lower and upper bounds for their proportion. Our results imply that implicit transpositions may be unavoidable and even appear in a significant proportion for some genomes. We estimate that in mammalian evolution transpositions constitute at least 17 % of genome rearrangements.

Keywords: Genome rearrangements · Transpositions · DCJ

1 Introduction

Genome rearrangements are dramatic evolutionary events that change genome structures. Since large-scale rearrangements are rare, it is natural to assume that the evolution history between two genomes corresponds to a shortest rearrangement scenario between them. The most common rearrangements are *reversals* that inverse contiguous segments of chromosomes, *translocations* that exchange tails of two chromosomes, and *fissions/fusions* that split/glue chromosomes. All these rearrangements can be conveniently modeled by the Double-Cut-and-Join (DCJ) operations [15], also known as 2-breaks [2], which make 2 "cuts" in a genome and "glues" the resulting genomic fragments in a new order.

Transpositions represent yet another type of genome rearrangements that relocate genomic segments across the genome. In contrast to reversal-like rearrangements modeled by DCJs (2-breaks), transpositions correspond to 3-breaks [2], which make 3 cuts and 3 gluings in a genome. Transpositions are more "powerful" than reversal-like rearrangements and in the model that includes both types of rearrangements (as 3-breaks and DCJs), transpositions tend to appear

© Springer International Publishing Switzerland 2015
A.-H. Dediu et al. (Eds.): AlCoB 2015, LNBI 9199, pp. 13–24, 2015.
DOI: 10.1007/978-3-319-21233-3_2

in shortest scenarios in a large proportion. However, in reality transpositions happen more rarely than reversals and typically appear in a small proportion in the course of evolution (e.g., in Drosophila evolution transpositions are estimated to constitute less than 10 % of genome rearrangements [13]). Earlier we showed that even the most promising model of *weighted genomic distance* [4,6,7] (where transpositions are assigned a higher weight) cannot bound the proportion of transpositions in the resulting rearrangement scenarios to biologically reasonable value [8]. This result emphasizes the need for a biologically adequate model for analysis of transpositions among other types of genome rearrangements.

While a transposition cannot be directly modeled by a DCJ, it can be modeled by a pair of DCJs. We refer to such pair of DCJs as an *implicit transposition*. We remark that DCJs forming an implicit transposition may not necessarily appear consecutively in a DCJ scenario. Furthermore, two implicit transpositions may share a DCJ and thus correspond to at most one actual transposition. We study appearance of implicit transpositions in shortest DCJ scenarios and derive a lower and an upper bounds for their proportion. Our results imply that implicit transpositions may be unavoidable in shortest DCJ scenarios between some genomes and even appear in a large proportion. In particular, we describe an extreme case where shortest DCJ scenarios entirely consist of implicit transpositions.

The paper is organized as follows. We describe graph-theoretical representation of genomes, rearrangement models (DCJs and k-breaks), and rearrangement scenarios in Sect. 2, and length-preserving modifications of DCJ scenarios along with dependency graphs capturing their combinatorial structure in Sect. 3. In Sect. 4, we study the appearance of implicit transpositions in shortest DCJ scenarios between two genomes and prove the uniform lower and upper bounds for their proportion (rate). In Sect. 5, we apply the obtained results for estimating the rate transpositions in mammalian evolution. We conclude the paper with discussion in Sect. 6.

2 Breakpoint Graphs and Rearrangement Scenarios

In the current study, we restrict our analysis to genomes with circular chromosomes (analysis of genomes with linear chromosomes will be published elsewhere). We represent a circular chromosome in genome P consisting of n genes as a cycle with n directed edges (encoding genes and their strands) alternating with n undirected edges connecting extremities of adjacent genes. A genome graph $G(P)$ is a collection of such cycles (Fig. 1a).

A DCJ [15] (also called a 2-break [2]) in genome P corresponds to a replacement of a pair of undirected edges with a different pair of undirected edges on the same set of four vertices in $G(P)$. Similarly, a 3-break [2] in genome P corresponds to a replacement of a triple of undirected edges with different triple of undirected edges on the same set of six vertices in $G(P)$.

For genomes P and Q composed of the same set of genes, the *breakpoint graph* $G(P, Q)$ is defined as the superposition of individual genome graphs $G(P)$ and $G(Q)$, and can be constructed by "gluing" the identically labeled directed edges in the graphs (Fig. 1b, c). From now on, we will ignore directed edges and assume that $G(P, Q)$ consists only of undirected edges, where the edges from genome P (P-*edges*) are colored black and the edges from genome Q (Q-*edges*) are colored red. Then the breakpoint graph $G(P, Q)$ represents a collection of cycles consisting of undirected edges alternating between black and red colors. We distinguish the following types of cycles with respect to their *length* ℓ (i.e., the number of black edges in a cycle): trivial cycles ($\ell = 1$) and odd cycles (ℓ is odd). We denote the number of cycles, trivial cycles, and odd cycles in $G(P, Q)$ as $c(P, Q)$, $c_1(P, Q)$, and $c_{odd}(P, Q)$, respectively.

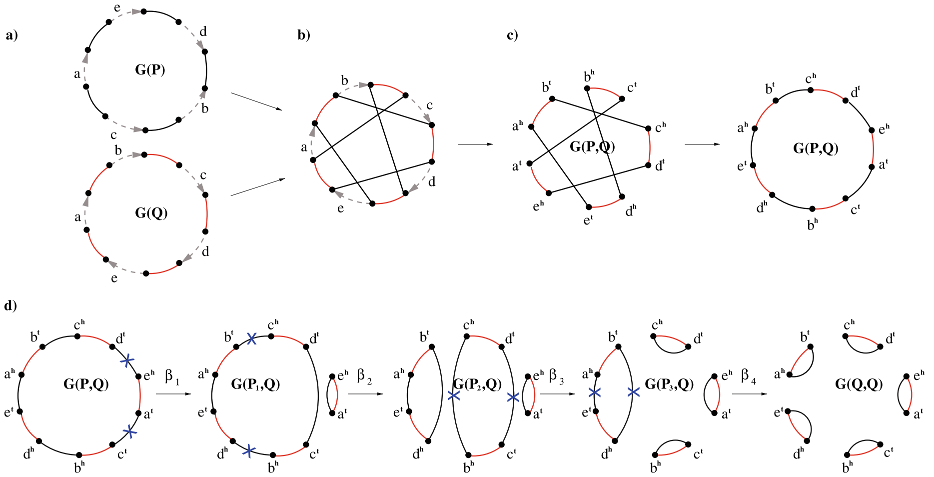


Fig. 1. (a) Genome graphs $G(P)$ and $G(Q)$ for unichromosomal circular genomes $P = (+a + e + d - b - c)$ and $Q = (+a + b + c + d + e)$, where undirected P-edges and Q-edges are colored black and red, respectively. (b) The superposition of genome graphs $G(P)$ and $G(Q)$. (c) The breakpoint graph $G(P, Q)$ is obtained from the superposition of $G(P)$ and $G(Q)$ with removal of directed edges. The graph $G(P, Q)$ is formed by a single black-red cycle, i.e., $c(P, Q) = 1$. (d) A transformation of the breakpoint graph $G(P, Q)$ into $G(Q, Q)$, which corresponds to a shortest DCJ scenario (of length $d_{\mathrm{DCJ}}(P, Q) = 4$) between genomes P and Q.

A DCJ scenario from genome P to genome Q corresponds to a transformation of the breakpoint graph $G(P, Q)$ into the breakpoint graph $G(Q, Q)$, which consists of trivial cycles (Fig. 1d).

Lemma 1 ([2,15]). *In a shortest DCJ scenario between two genomes, each DCJ splits some cycle in the corresponding breakpoint graph into two and thus increases the number of cycles by one.*

From Lemma 1, one can immediately get a formula for the DCJ distance:

Theorem 1 ([2,15]). *The DCJ distance between genomes P and Q on n genes is*

$$d_{\mathrm{DCJ}}(P, Q) = n - c(P, Q).$$

We remark that a DCJ (2-break) represents a particular case of a 3-break. A3-break that is not a DCJ is called *complete*. The following lemma describes the effect of DCJs and complete 3-breaks in shortest 3-break scenarios.

Lemma 2 ([2,8]). *In a shortest 3-break scenario between two genomes, each DCJ (2-break) splits an even cycle in the corresponding breakpoint graph into two odd cycles, while each complete 3-break splits an odd cycle into three odd cycles. Therefore, each rearrangement in such a scenario increases the number of odd cycles in the breakpoint graph by two.*

A 3-break scenario from genome P to genome Q increases the number of odd cycles from $c_{odd}(P, Q)$ in $G(P, Q)$ to $c_{odd}(Q, Q)$, which is the number of genes in Q. From Lemma 2, the following formula for the 3-break distance emerges:

Theorem 2 ([2]). *The 3-break distance between genomes P and Q on n genes is*

$$d_3(P, Q) = \frac{n - c_{odd}(P, Q)}{2}.$$

We will also need the following theorem that easily follows from Lemma 1.

Theorem 3. *Along a transformation of the breakpoint graphs from $G(P, Q)$ to $G(Q, Q)$ corresponding to a shortest DCJ scenario between genomes P and Q, any edge once removed is never recreated.*

Proof. Lemma 1 implies that after an edge (u, v) is removed by a DCJ from the breakpoint graph, the vertices u and v start to belong to distinct cycles and can never belong to the same cycle again (which would be the case if the edge (u, v) is ever re-created). □

3 Length-Preserving Operations and Dependency Graphs

We call rearrangement scenarios (in particular, single DCJs or pairs of DCJs) between the same two genomes *equivalent*.

Since each DCJ removes and adds some edges in a breakpoint graph, two adjacent DCJs α and β in a DCJ scenario are called *independent* if β removes edges that were not created by α. Otherwise, if β removes some edge(s) created by α, then β *depend*s on α. Furthermore, let $k \in \{1, 2\}$ be the number of edges created by α and removed by β. We say that β *strongly depends* on α if $k = 2$ and *weakly depends* on α if $k = 1$. We remark that adjacent pair of strongly dependent DCJs may not appear in shortest DCJ scenarios, since such pair can be replaced by an equivalent single DCJ, decreasing the scenario length.

As we mentioned above, we can change the order of two adjacent independent DCJs and obtain an equivalent scenario. For a pair of adjacent weakly dependent DCJs, there exist exactly two other equivalent pairs of weakly dependent DCJs [5,9]. We therefore consider the following two types of length-preserving operations, which can be applied to a pair of adjacent DCJs (α, β) in a DCJ scenario:

(T1) If α and β are independent, we replace (α, β) with (β, α).
(T2) If α and β are (weakly) dependent, we replace (α, β) with an equivalent pair of weakly dependent DCJs.

In was shown [5] that any shortest DCJ scenario can be obtained from any other shortest DCJ scenario between the same two genomes using only operations of types (T1) and (T2).

To better capture and analyze the combinatorial structure of DCJs in a shortest DCJ scenario t, we construct the *dependency digraph* $\mathrm{DG}(t)$ (also known as *overlap graph* [11,12]), whose vertices are labeled with DCJs from t and there is an arc (α, β) whenever β depends on α (Fig. 2).

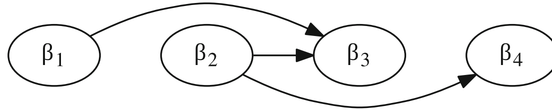


Fig. 2. The dependency graph $\mathrm{DG}(t)$ for DCJ scenario t defined in Fig. 1d.

Theorem 4. *Let t be a shortest DCJ scenario between genomes P and Q composed of the same n genes. Then*

(i) the number of arcs in $\mathrm{DG}(t)$ *is* $n - 2 \cdot c(P, Q) + c_1(P, Q)$*;*
(ii) both indegree and outdegree of each vertex in $\mathrm{DG}(t)$ *are at most 2;*
(iii) t represents a topological ordering of $\mathrm{DG}(t)$*;*
(iv) $\mathrm{DG}(t)$ *is acyclic.*

Proof. An arc (α, β) in $\mathrm{DG}(t)$ corresponds in the breakpoint graph transformation t to an edge that is created by DCJ α and removed by DCJ β. By Theorem 3 the removed edges are never recreated, implying that this correspondence is one-to-one.

There are $n - c_1(P, Q)$ P-edges in the non-trivial cycles in $G(P, Q)$ and they have to be removed by DCJs from t in order to form trivial cycles. The other edges removed by DCJs from t must have been created by earlier DCJs. Since the total number of removed edges by DCJs in t is $2 \cdot |t| = 2 \cdot d_{\mathrm{DCJ}}(P, Q) = 2n - 2c(P, Q)$ (by Theorem 1), the number of such earlier created and then removed edges is $2n - 2c(P, Q) - (n - c_1(P, Q)) = n - 2c(P, Q) + c_1(P, Q)$ and this gives the number of arcs in $\mathrm{DG}(t)$.

Since by Theorem 3 the edges in the breakpoint graph transformation are not recreated, any DCJ in t (which removes two edges and creates two edges) depends on at most two other DCJs and may have at most two dependent DCJs. That is, both indegree and outdegree of any vertex in $\mathrm{DG}(t)$ are bounded by 2.

If (α, β) is an arc in $\mathrm{DG}(t)$, then a DCJ β removes some edge e created by a DCJ α. By Theorem 3 no other DCJ besides α can create e, and thus β must follow α in t. So t represents a topological ordering for $\mathrm{DG}(t)$ and therefore $\mathrm{DG}(t)$ is acyclic. □

The following theorem refines the result of [5] with respect to sorting shortest DCJ scenarios with operations (T1) only.

Theorem 5. *Let t_1 and t_2 be shortest DCJ scenarios between the same two genomes. Scenario t_1 can be obtained from scenario t_2 with operations (T1) if and only if* $\mathrm{DG}(t_1) = \mathrm{DG}(t_2)$.

Proof. Suppose that t_1 and t_2 correspond to the same dependency graph, i.e., $\mathrm{DG}(t_1) = \mathrm{DG}(t_2) = G$, then by Theorem 4 they represent topological orderings of G. We will show that t_1 and t_2 can be obtained from each other with operations (T1). Let

$$t_1 = (\alpha_1, \alpha_2, \ldots, \alpha_k, \gamma, \ldots) \text{ and}$$

$$t_2 = (\alpha_1, \alpha_2, \ldots, \alpha_k, \beta_1, \beta_2, \ldots, \beta_m, \gamma, \ldots),$$

where $\gamma \neq \beta_1$ are the first different DCJs in the two scenarios. We will show that γ in t_2 can be moved to $(k+1)$-st position (i.e., its position in t_1) with operations (T1). Since β_m follows γ in t_1 but precedes γ in t_2, these vertices are not connected with an arc in G and we can apply operation (T1) to t_2 to obtain $(\alpha_1, \alpha_2, \ldots, \alpha_k, \beta_1, \beta_2, \ldots, \gamma, \beta_m, \ldots)$. After m such operations we get $(\alpha_1, \alpha_2, \ldots, \alpha_k, \gamma, \beta_1, \beta_2, \ldots, \beta_m, \ldots)$, where γ is at the same position as in t_1. Using induction on k, we conclude that t_1 can be obtained from t_2 with operations (T1), and vice versa.

Now, suppose that DCJ scenarios t_1 and t_2 can be obtained from each other with operations (T1). Since operations (T1) changes only the order of DCJs in the scenario but keeps the DCJs themselves intact, the dependency graph is not affected by such operations either. Therefore, $\mathrm{DG}(t_1) = \mathrm{DG}(t_2)$. □

For a directed graph G, we define $\overline{G}$ as the undirected graph obtained from G by making all arcs undirected. While Theorem 4 claims that $\mathrm{DG}(t)$ is acyclic, from the results of Shao *el al.* [14][1] it follows that $\overline{\mathrm{DG}(t)}$ is a forest (i.e., a graph with no cycles):

Theorem 6 ([14])**.** *Let t be a shortest DCJ scenario between two genomes composed of the same set of genes. Then the graph* $\overline{\mathrm{DG}(t)}$ *represents a forest.*

[1] Shao *el al.* [14] studies more general *trajectory graphs*, from which the dependency graphs can be obtained by contraction of edges.

4 Implicit Transpositions in Shortest DCJ Scenarios

While DCJs mimic most of common genome rearrangements (reversals, translocations, fissions, fusions), more complex rearrangements such as transpositions cannot be modeled by a single DCJ. A transposition, which cuts off a segment of a chromosome and inserts it into some other place in the genome, can be modeled by a pair of weakly dependent DCJs, replacing three undirected edges with three other undirected edges on the same six vertices in the genome graph. We remark that this operation is also known as a 3-break rearrangement [2].

Below we study how transpositions appearing in the course of evolution between two genomes may affect shortest DCJ scenarios between them. While a transposition constitutes a pair of consecutive DCJs, their positions in a DCJ scenario may not always be reconstructed correctly. In particular, the two DCJs forming a transposition may be interweaved with other independent DCJs that precede or follow the transposition in the evolutionary scenario, which inspires the following definition.

In a DCJ scenario $t = (\alpha_1, \alpha_2, \ldots, \alpha_n)$, a pair of DCJs (α_i, α_j) forms an *implicit transposition* if they can be made adjacent by applying a number of operations (T1) and form a pair of weakly dependent DCJs. When these DCJs become adjacent, they can be replaced by a single transposition. We refer to such a transposition as *recovered* from the DCJ scenario t. This poses us a question of how many transpositions can be *simultaneously* recovered from a given shortest DCJ scenario t.

Since two distinct implicit transpositions in a shortest DCJ scenario t may share a DCJ, the maximum number of transpositions that can be recovered from t may be smaller than the number of implicit transpositions in t. We therefore are interested in (pairwise) *disjoint* implicit transpositions, which do not share any DCJs between them. Furthermore, it is not immediately clear if existence of a set of m disjoint implicit transpositions in t implies that m transpositions can be simultaneously recovered from t, but we will prove below that this is indeed the case. We therefore define $\mathrm{DIT}(t)$ as the maximum number of disjoint implicit transpositions in t, which will be shown also equal to the maximum number of transpositions that can be simultaneously recovered from t.

4.1 Disjoint Implicit Transpositions as Matchings

It can be easily seen that an implicit transposition formed by a pair of DCJs (α, β) in a shortest DCJ scenario t corresponds to an arc in the dependency graph $\mathrm{DG}(t)$. However, it is not immediately clear if every arc (x, y) in $\mathrm{DG}(t)$ represents an implicit transposition, i.e., if DCJs x and y in t can be made adjacent with operations (T1). In this section, we prove that any matching M in $\mathrm{DG}(t)$ forms a disjoint collection of implicit transpositions that can be simultaneously recovered from t.

We call a graph G a *directed forest* if $\overline{G}$ is a forest. By Theorem 6, $\mathrm{DG}(t)$ represents a directed forest for any shortest DCJ scenario t.

Lemma 3. *Let G be a directed forest. Then for any arc (α_1, α_2) in G, there exists a topological ordering of G in which α_1 and α_2 are adjacent.*

Proof. Let G' be a graph obtained from G by removing the arc (α_1, α_2) and gluing vertices α_1, α_2 into a new single vertex β. That is, for any arc (α_i, γ) with $\gamma \neq \alpha_2$ in G, there is an arc (β, γ) in G'; and for any arc (γ, α_i) with $\gamma \neq \alpha_1$ in G, there is an arc (γ, β) in G'.

We claim that G' is a directed forest. Indeed, the $\overline{G'}$ can be viewed as the result of contraction of the edge (α_1, α_2) in $\overline{G}$ into a single vertex β. Such contraction cannot created a cycle, i.e., $\overline{G'}$ remains to be a forest.

Let t' be a topological ordering of G'. By replacing the vertex β in t' with pair of adjacent vertices α_1, α_2, we obtain the required topological ordering of G. □

Theorem 7. *Let G be a directed forest. Then for any matching M in G, there exists a topological ordering t of G such that for any arc $(\alpha_1, \alpha_2) \in M$, DCJs α_1 and α_2 are adjacent in t.*

Proof. We prove the theorem statement by induction on $|M|$. For the base case $|M| = 1$, the statement follows from Lemma 3. Assume now that the statement holds for $|M| = m$.

For $|M| = m + 1$, let (α_1, α_2) be an arc in M. We construct the graph G' as described in the proof of Lemma 3 and let $M' = M \setminus \{(\alpha_1, \alpha_2)\}$. Since G' is a directed forest and $|M'| = m$, by the induction assumption there is a topological ordering t' of G' such that for any arc $(\alpha_1, \alpha_2) \in M'$, the DCJs α_1 and α_2 are adjacent in t'.

We obtain t from t' by replacing the vertex β with the ordered pair of vertices α_1, α_2. It is easy to see that such t represents the required topological ordering for G. □

4.2 Bounds for the Rate of Implicit Transpositions

Theorem 8. *Let t be a shortest DCJ scenario between genomes P and Q composed of the same n genes. Then* $\mathrm{DIT}(t) \geq T_L(P, Q)$, *where*

$$T_L(P, Q) = \left\lceil \frac{n - 2 \cdot c(P, Q) + c_1(P, Q)}{4} \right\rceil.$$

Proof. We know that the graph $\overline{\mathrm{DG}(t)}$ is a forest (Theorem 6), where degree of each vertex is bounded by 4 (Theorem 4). Let us construct a matching M in $\overline{\mathrm{DG}(t)}$ iteratively. Initially we let $G = \overline{\mathrm{DG}(t)}$ and $M = \emptyset$.

If G contains at least one edge, it also contains a leaf (i.e., vertex of degree 1) α. We add its only incident edge (α, β) to M and remove from G all edges incident to the vertex β. Clearly, at most four such edges are deleted. We repeat this procedure until all edges of G are removed. By Theorem 4, the graph $\overline{\mathrm{DG}(t)}$ contains $n - 2 \cdot c(P, Q) + c_1(P, Q)$ edges and thus we perform at least $\left\lceil \frac{n-2 \cdot c(P,Q)+c_1(P,Q)}{4} \right\rceil = T_L(P, Q)$ iterations, implying that $|M| \geq T_L(P, Q)$.

By construction, it is clear that M forms a matching in $\overline{\mathrm{DG}(t)}$ and thus under a suitable orientation of the edges in M, it also forms a matching in $\mathrm{DG}(t)$. By Theorem 7, there exists a topological ordering t' of $\mathrm{DG}(t)$ such that the endpoints of all edges in M are adjacent in t'. By Theorem 5, topological ordering t' can be obtained from t with operations (T1), implying that we can simultaneously recover from t all elements of M. Therefore, $\mathrm{DIT}(t) \geq |M| \geq T_L(P,Q)$. □

Theorem 9. *Let t be a shortest DCJ scenario between genomes P and Q composed of the same n genes. Then* $\mathrm{DIT}(t) \leq T_U(P,Q)$*, where*

$$T_U(P,Q) = \frac{n - 2 \cdot c(P,Q) + c_{odd}(P,Q)}{2}.$$

Proof. There exist $\mathrm{DIT}(t)$ pairwise disjoint implicit transpositions in t, which after a number of operations (T1) can be made adjacent. We replace each pair of DCJs forming an implicit transposition with a 3-break to obtain a 3-break scenario ℓ. The length of ℓ is $|\ell| = |t| - \mathrm{DIT}(t) = n - c(P,Q) - \mathrm{DIT}(t)$. We remark that $|\ell|$ is no smaller than the 3-break distance between genomes P and Q:

$$n - c(P,Q) - \mathrm{DIT}(t) \geq d_3(P,Q) = \frac{n - c_{odd}(P,Q)}{2},$$

implying the required upper bound for $\mathrm{DIT}(t)$. □

From the perspective of the upper bound in Theorem 9, an extreme case for genomes P and Q on n genes is $c(P,Q) = c_{odd}(P,Q) = 1$, in which by Theorem 2 there exists a shortest 3-break scenario between P and Q of length $\frac{n-1}{2}$. According to Lemma 2, such scenario contains no DCJs but only complete 3-breaks. By replacing each 3-break in this scenario with an equivalent pair of DCJs (forming an implicit transposition), we can get a DCJ scenario t between P and Q with $2 \cdot \mathrm{DIT}(t) = n - 1$ DCJs. It is easy to see that in this case we have $\mathrm{DIT}(t) = T_U(P,Q) = \frac{n-1}{2}$, i.e., the upper bound for $\mathrm{DIT}(t)$ is tight.

Let t be a shortest DCJ scenario between genomes P and Q. There exist $\mathrm{DIT}(t)$ pairwise disjoint implicit transpositions in t. The pair of DCJs in each of these implicit transpositions can be made adjacent with operations (T1). If we replace each adjacent pair of DCJs forming an implicit transposition in the resulting scenario with an actual transposition (complete 3-break), then we obtain a scenario t' of length $d_{\mathrm{DCJ}}(P,Q) - \mathrm{DIT}(t)$ composed of $d_{\mathrm{DCJ}}(P,Q) - 2 \cdot \mathrm{DIT}(t)$ DCJs and $\mathrm{DIT}(t)$ transpositions. Thus the proportion of transpositions in t' is $\frac{\mathrm{DIT}(t)}{d_{\mathrm{DCJ}}(P,Q) - \mathrm{DIT}(t)}$. We refer to this proportion as the *rate of implicit transpositions* in t and denote it by $r(t)$. The following theorem gives uniform bounds for $r(t)$ that do not depend on a particular scenario t.

Theorem 10. *Let t be any shortest DCJ scenario between genomes P and Q. Then*

$$\frac{T_L(P,Q)}{d_{\mathrm{DCJ}}(P,Q) - T_L(P,Q)} \leq r(t) \leq \frac{T_U(P,Q)}{d_{\mathrm{DCJ}}(P,Q) - T_U(P,Q)}.$$

Proof. Since $r(t) = \frac{\mathrm{DIT}(t)}{d_{\mathrm{DCJ}}(P,Q)-\mathrm{DIT}(t)} = \frac{d_{\mathrm{DCJ}}(P,Q)}{d_{\mathrm{DCJ}}(P,Q)-\mathrm{DIT}(t)} - 1$, the value of $r(t)$ monotonically increases as $\mathrm{DIT}(t)$ grows. The stated bounds for $r(t)$ immediately follow from Theorems 8 and 9. □

5 Implicit Transpositions in Mammalian Evolution

Below we estimate the rate of implicit transpositions recovered from pairwise DCJ scenarios between 6 mammalian species: mouse, rat, dog, macaque, human, and chimpanzee. Their gene orders and pairwise orthology relationship were obtained from Ensembl BioMart tool [10] on the following genomes: *Mus musculus* (GRCm38.p1), *Rattus norvegicus* (Rnor_5.0), *Canis familiaris* (CanFam3.1), *Homo sapiens* (GRCh37.p12), *Macaca mulatta* (MMUL_1.0), and *Pan troglodytes* (CHIMP2.1.4). Since our approach is currently limited to circular chromosomes, we artificially circularized chromosomes in each genome (such circularization is expected to have a minor impact [1]). For each pair of genomes, we represented them as sequences of shared genes present in one copy in each of these genomes[2] and used Theorem 10 to compute the lower and upper bounds for the rate of implicit transpositions between them. The results are given in Table 1.

Table 1. Estimation for the rate of implicit transpositions between pairs of genomes among mouse (M), rat (R), dog (D), macaque (Q), human (H), and chimpanzee (C).

Genome pairs	Shared genes	DCJ distance	Lower bound	Upper bound
M & R	14312	832	0.18	0.79
M & D	13852	1131	0.20	0.86
M & H	14119	1173	0.20	0.86
M & C	12608	1004	0.21	0.85
M & Q	13537	947	0.20	0.84
R & D	13312	1310	0.20	0.83
R & H	13352	1110	0.20	0.81
R & C	11942	961	0.21	0.80
R & Q	13064	1181	0.20	0.81
D & H	13808	1123	0.18	0.85
D & C	12372	984	0.19	0.85
D & Q	13449	1139	0.18	0.86
H & C	15646	929	0.17	0.93
H & Q	14408	973	0.17	0.91
C & Q	12912	884	0.18	0.91

[2] We remark that since the set of shared genes varies across genome pairs, the DCJ distances between genomes in different pairs are incomparable.

Table 1 demonstrates that the rate of implicit transpositions in mammalian evolution is at least 0.17. This is consistent and close to the estimate of the transposition rate in mammalian evolution as 0.26 recently obtained with statistical methods [3]. While we showed that the upper bound may be tight for some pairs of genomes, we believe that this not the case for mammalian genomes and the upper bound values in Table 1 appear to be superfluous.

6 Discussion

We continue our study of the combinatorial structure of DCJ scenarios from the perspective of simple length-preserving transformations, each affecting only a pair of consecutive DCJs (first introduced in [5]). Earlier we showed [9] that any shortest DCJ scenario between a genome with $m \geq 1$ circular chromosomes and a linear genome (consisting of linear chromosomes) can be transformed into a shortest DCJ scenario, where circular chromosomes are eliminated by the first m DCJs and the rest represents a scenario between linear genomes. We further used this construction to obtain an approximate solution for the linear genome median problem.

In the current work, we study how evolutionary transpositions may implicitly appear in shortest DCJ scenarios and prove uniform bounds on their rate. Since transpositions are rather powerful rearrangements, it is not surprising that they may appear in a significant proportion that cannot be easily bounded in rearrangement scenarios between some genomes. Even though we do not yet have a recipe for limiting the effect of transpositions in the combined DCJ (2-break) and 3-break model (for which we earlier proved failure of the weighting approach [8]), our current study provides a step towards better understanding of the properties of transpositions and how they may affect reconstruction of the evolutionary history.

Our analysis of mammalian genomes demonstrates that the lower bound for the implicit transposition rate is close to the estimation obtained with statistical methods [3]. It is interesting to notice that the rate attains extreme values at pairs of primate genomes, which cannot be directly explained by the number of shared genes or DCJ distance between them and thus may indicate higher complexity of primate genomes with respect to the transposition analysis. While our approach is currently limited to circular chromosomes and applied for mammalian genomes with artificially circularized chromosomes, such circularization is expected to have a minor impact on the resulting estimates [1]. Extension of our approach to linear genomes will be published elsewhere.

Acknowledgments. The work was supported by the National Science Foundation under the grant No. IIS-1462107.

References

1. Alekseyev, M.A.: Multi-break rearrangements and breakpoint re-uses: from circular to linear genomes. J. Comput. Biol. **15**(8), 1117–1131 (2008)

2. Alekseyev, M.A., Pevzner, P.A.: Multi-break rearrangements and chromosomal evolution. Theor. Comput. Sci. **395**(2), 193–202 (2008)
3. Alexeev, N., Aidagulov, R., Alekseyev, M.A.: A computational method for the rate estimation of evolutionary transpositions. In: Ortuño, F., Rojas, I. (eds.) IWBBIO 2015, Part I. LNCS, vol. 9043, pp. 471–480. Springer, Heidelberg (2015)
4. Bader, M., Ohlebusch, E.: Sorting by weighted reversals, transpositions, and inverted transpositions. J. Comput. Biol. **14**(5), 615–636 (2007)
5. Braga, M.D., Stoye, J.: The solution space of sorting by DCJ. J. Comput. Biol. **17**(9), 1145–1165 (2010)
6. Eriksen, N.: $(1 + \epsilon)$-approximation of sorting by reversals and transpositions. In: Gascuel, O., Moret, B.M. (eds.) WABI 2001. LNCS, vol. 2149, pp. 227–237. Springer, Heidelberg (2001)
7. Fertin, G., Labarre, A., Rusu, I., Tannier, E., Vialette, S.: Combinatorics of Genome Rearrangements. MIT Press, Cambridge (2009)
8. Jiang, S., Alekseyev, M.A.: Weighted genomic distance can hardly impose a bound on the proportion of transpositions. In: Bafna, V., Sahinalp, S.C. (eds.) RECOMB 2011. LNCS, vol. 6577, pp. 124–133. Springer, Heidelberg (2011)
9. Jiang, S., Alekseyev, M.A.: Linearization of median genomes under DCJ. In: Brown, D., Morgenstern, B. (eds.) WABI 2014. LNCS, vol. 8701, pp. 97–106. Springer, Heidelberg (2014)
10. Kasprzyk, A.: BioMart: driving a paradigm change in biological data management. Database 2011, bar049 (2011)
11. Ouangraoua, A., Bergeron, A.: Combinatorial structure of genome rearrangements scenarios. J. Comput. Biol. **17**(9), 1129–1144 (2010)
12. Ozery-Flato, M., Shamir, R.: Sorting by translocations via reversals theory. In: Bourque, G., El-Mabrouk, N. (eds.) RECOMB-CG 2006. LNCS (LNBI), vol. 4205, pp. 87–98. Springer, Heidelberg (2006)
13. Ranz, J., González, J., Casals, F., Ruiz, A.: Low occurrence of gene transposition events during the evolution of the genus Drosophila. Evolution **57**(6), 1325–1335 (2003)
14. Shao, M., Lin, Y., Moret, B.: Sorting genomes with rearrangements and segmental duplications through trajectory graphs. BMC Bioinform. **14**(15), 1–8 (2013)
15. Yancopoulos, S., Attie, O., Friedberg, R.: Efficient sorting of genomic permutations by translocation, inversion and block interchange. Bioinformatics **21**(16), 3340–3346 (2005)

Constraint-Based Genetic Compilation

Christophe Ladroue and Sara Kalvala(✉)

Department of Computer Science, University of Warwick,
Coventry CV4 7AL, UK
Sara.Kalvala@warwick.ac.uk

Abstract. Synthetic biology aims at facilitating the design of new organisms via the standardization of biological parts and following engineering principles. We present ATGC (Assistant To Genetic Compilation), a software tool that automatically builds a functional sequence of DNA from a minimal set of requirements. Through a simple language, the user provides in-house knowledge about their construct (*e.g.* relative placement of parts, number of restriction enzymes). ATGC combines information from established biology, user knowledge and bioinformatics databases, and maps the problem to a constraint satisfaction setting. The solution is a functional DNA sequence ready to be assembled and transferred to a target organism.

Keywords: Synthetic biology · Biocompilation · Constraint satisfaction

1 Introduction

Synthetic biology [9] aims at reaching a level of control in biology that is traditionally found in mechanical or electrical engineering [5]. Its approach consists in considering biological *parts* and combining them to achieve a pre-defined task. Parts are fragments of DNA fulfilling a specific function: promoters, protein coding sequences (CDS), ribosome binding sites (RBS), terminators and so on. A *device* is an ordered collection of parts, that has a specific effect (*e.g.* producing a certain protein when sensing a signal). The resulting string of DNA can be thought of as a blueprint for a program to be run by the cell. The potential benefit for science and society is immense, from a better understanding of cellular processes [8], to drug manufacturing [16] to energy production [18].

Following this computer analogy, we present our approach to *biocompiling* [6]: deriving, from a high-level, human-readable encoding of desired behavior, the corresponding set of instructions at a machine (or a cell) level, in this case in terms of nucleotides (A, C, T and G) instead of zeroes and ones. In this paper we consider only simple forms of control, where the production of proteins is controlled by the choice of promoter that precedes the corresponding gene in the DNA sequence. Nature has evolved many more complex forms of control (such as the very powerful method known as RNA Interference [11]) but these are still beyond the scope of most of the projects developing practical design tools for synthetic biology.

© Springer International Publishing Switzerland 2015
A.-H. Dediu et al. (Eds.): AlCoB 2015, LNBI 9199, pp. 25–38, 2015.
DOI: 10.1007/978-3-319-21233-3_3

From a conceptual point of view biocompilation *should* be a simple process: one can look up parts from the many comprehensive databases of biological parts available to find ones that provide the required functionality, and string the corresponding nucleotide sequences together. However, in practice there are many obstacles to obtaining functional sequences: the quantitative nature of genetic translation means that there may be many initial designs but they provide unacceptable reaction rates. Furthermore, there are local interactions that can occur between specific nucleotide sequences and the underlying cellular mechanisms which need to be considered. And there are usually some pragmatic constraints from the specific lab or experimental setting in which the actual biological experiments are to be performed.

Biocompilation has been addressed by a number of projects in recent years. GenoCAD [2] uses a context-free grammar for constraining sequence building and also verifying existing sequences. PROTO BIOCOMPILER [1] is based on the spatial language PROTO. An abstract regulatory network is derived from the behavioral specifications and then simplified and instantiated to a smaller genetic circuit. Eugene [4] starts from a collection of parts and devices, and creates all possible combinations. It then prunes unwanted arrangements following simple rules. However, these rules are rarely under the control of the user, who is often unable to control the decisions made by the tools. Therefore, realistic automation of the biocompilation process has so far been quite elusive.

Our approach, described here and exemplified by our tool ATGC, also uses built-in rules informed from biology to construct a functional device from a collection of parts, but it also allows users to add simple, specific *directives* to control the decisions made. Potential inconsistencies are managed by using a constraint-based methodology that balances different constraints against each other. We make extensive use of a CSP solver, the Java library JaCoP [14]. The advantage of this is that our decision making can easily accommodate more heuristics and user design strategies, without needing to re-structure the whole process. Our tool automatically completes unfinished devices, lowering the requirement for genetic knowledge from the user while accommodate user-defined constraints.

2 General Workflow

In the general workflow of ATGC, a user declares a list of parts to be include in the design, using parts of different types (promoters, CDS, terminators, etc.), and they can also add extra constraints on the construction of the final sequence, such as the relative position of parts or the direction of the CDS. The set of constraints offered may be informed by users' daily practice. These extra constraints are encoded in terms of a cost function to be minimized. The task of ATGC consists in (1) taking stock of what parts and devices are used, (2) finding a realistic arrangement of the parts (3) instantiating their exact nucleotide sequences and (4) creating an output file that can be fed into a DNA assembly process.

In the simplest use-case, users only specify promoters and CDSs, which provide enough information to check the desired functionality via simulation.

But to be biologically functional, a device requires extra parts to be integrated into the design–such as terminator sequences, to ensure that DNA transcription stops after the gene in question has been transcribed. Furthermore, low-level conflicts at the nucleotide level can arise, which are difficult to check for and which can invalidate a design attempt which has been made.

The biocompiler automatically adds the missing necessary parts, such as RBS and terminators, as well as, if requested by the user, a number of cloning sites. Insertion of all these different components can raise several conflicts, and ATGC solves these conflicts in many cases. There are three main steps in the biocompilation process:

1. The set of parts specified by the user is expanded with other parts needed to complete the design and the various parts are placed in a coherent sequence, taking into consideration the fact that devices can be placed in *both* directions in the complementary DNA strands.
2. Ribosomal Binding Sites (RBSs) are tags that are translated along with CDSs and determine the translation rate for the CDSs. ATGC chooses the RBS sequences that implement the translation rate desired by the user.
3. Cloning sites are often desired by users and it is often very difficult to find appropriate cloning sites because these 4- and 6-mer sequences should not appear anywhere else in the DNA. ATGC has an extensive mechanism to try to find adequate cloning sites and their corresponding restriction enzymes, by re-generating nucleotide sequences for the rest of the DNA if required.

The solution found by ATGC can be exported in SBOL format [10], which can be simulated and which is compatible with a large a growing number of synthetic biology software, for example the J5 DNA assembly tool [12].

ATGC does not interpret the meaning of sequences it manipulates. It is thus possible to generate a nonsensical sequence if the input sequences are themselves nonsensical; e.g. a promoter consisting only of the nucleotide A, or a 'gene' with the sequence of a terminator. Biology is hugely pliable, and nature has evolved many clever ways of implementing control, so we do not over-constrain the types of designs that can be produced. Another point to be mentioned is that ATGC was designed with the assumption that each part has a separate function, following the efforts of the synthetic biology community to standardise parts. As a consequence, it cannot cater for situations where one would like to produce overlapping coding sequences, for example; the sequences will appear separately, one after the other. But we believe the infrastructure we provide is robust enough that further extensions that capture more interesting designs are possible.

In the next sections, we explain the various elements of user input and also explain the way that ATGC processes this information to produce viable designs at the nucleotide level.

3 User Input Language

The input language for ATGC is similar to that adopted by many other biocompiler tools, and consists mainly in ways in which parts can be accessed from

various databases, and host organisms can be identified and the environment (such as concentrations of signaling molecules, etc.) can be specified.

What is unique in ATGC is the way *directives* can be specified. These are interspersed into the input file and are all preceded by the **ATGC** keyword. User expertise is often difficult to formalize as it contains *ad-hoc* rules and rules of thumb. The various directives which have been implemented so far will be shown in the next section alongside the algorithms we developed to process them. Here we explain how parts, devices, and cells are specified by users.

Parts are declared by the user by specifying their types and sequences, as both information will be required for building the final sequence. That is, a biological part in ATGC is just an ordinary variable. Four types of parts are available: promoter, gene, RBS and terminator. Each of these types take one argument: either an explicit DNA sequence, or a pointer to a database entry.

Users need to specify the set of parts that they may want to use in their designs: we believe this is more sensible than hard-wiring *our* favorite parts, which may unnecessarily increase the search space for designs. Figure 1 shows the various ways in which parts can be defined: directly as sequences (PromD, GeneD), or as named parts in popular databases, for example PromB from BIOFAB [3], GeneP from the Parts Registry ([13], and PromV and GeneV from the Virtual Parts Repository [7], which we use in our running example. These repositories contain promoters, genes, RBSs and terminators that have been used extensively by the synthetic biology community.

```
PromD =
 PROMOTER(sequence = "TGTGCATGACAAATCAGATTAACAC")
GeneD =
 GENE(sequence = "ATGAGTCAGTTTCGATAATC")

PromB =
 PROMOTER(URI = "ATGC://biofab/part/PLTETo1")

PromP =
 PROMOTER(URI = "http://parts.igem.org/Part:BBa_I14033")
GeneP =
 GENE(URI = "http://parts.igem.org/Part:BBa_K592009")

PromV =
 PROMOTER(URI = "http://sbol.ncl.ac.uk:8081/part/BO_2689")
GeneV =
 GENE(URI = "http://sbol.ncl.ac.uk:8081/part/BO_28536")
```

Fig. 1. Declaring parts that can be used in the biocompiler

Genetic designs involve specifying the assembly of parts into devices and the placement of these devices into a context, such as a host cell. These structures are

specified as given in Fig. 2. *Devices* are understood in synthetic biology as composite genetic units with functional parts such as promoters and CDSs. Devices are declared analogously to functions in a typical programming language, with a signature specifying the parts used. The actual placement of the parts is the goal of biocompilation. Thus, initially, this section may be empty, but is filled in by the biocompiler. Devices are used in cells, which in turn are placed in a region. The information about the cell and the region provide important information that is used to simulate the dynamics of the cell and thus verify the results of the biocompiler.

```
DEVICE myDevice = new DEVICE(parts =
        [myPromoter, aGene, anotherGene])(){
                // body of the device    }

define myCell typeof CELL(){
  (...)  // declare parts and devices    }

define myRegion typeof REGION(){
  CELL strain2015d = new myCell()          }
```

Fig. 2. Declaring devices, cells and regions

An important aspect for ATGC is that descriptions may be (and in fact are expected to be) incomplete: the goal of biocompilation is to fill in the design to make a feasible complete specification. This is done by automatically searching through a large space of genetic parts and configurations and finding consistent and good solutions, not only in terms of syntax but also in terms of quantitative analysis and analysis of the actual nucleotide sequences.

4 Biocompilation Step: Arranging the Bio-Parts

The biocompiler will use the parts required for building the whole construct, following rules from known biology. There can be many equivalent ways to place the parts in the final piece of DNA, and some might be preferred by the user. For example, they might want to reproduce an existing construct that has been reported to work in previous studies.

ATGC finds the position of each part by using built-in biological knowledge as directives for placement. But some hard constraints, such as the notion that a device should start with a promoter and end with terminators, or that genes must be preceded by an RBS, cannot be violated. Extra constraints provided by the user, like relative positions of parts, are captured in a cost function.

The directive **ARRANGE**, followed by a list of parts, will force the compiler to favor an arrangement that matches the relative positions of the parts involved in the directive. This directive makes it easy to guide the biocompiler to generate solutions that fit a pre-determined, in-house design. For example, the directive:

```
ATGC ARRANGE nahR, Pnah, Psal, xysl2
```

directs the biocompiler to try to place these four parts in this particular order—for example, when such a component has already been produced in-house. **ARRANGE** is a 'soft' constraint, in that it favors solutions where it is satisfied, but not if it is in direct contradiction with hard-coded rules for genetic constructs.

The **DIRECTION** directive forces the direction of a device: forward or reverse. For example, the requirement that a device is to be read in a specific direction can be specified as:

```
ATGC myDevice DIRECTION: FORWARD
```

Mapping to a Constraint Satisfaction Problem. The corresponding constraint satisfaction problem (CSP) has 2 types of objects: parts $p \in \{P, R, G, T, C\}$ and devices $i \in i^1, i^2, \ldots$, each of which is associated with a set σ of parts. The goal is to assign locations $L() \in \{1, \ldots, \#parts\}$ to parts and to assign a direction $D() \in \{F(=0), B(=1)\}$ to devices.

The biocompiler adds a number of constraints in order to build a functional piece of DNA, such as the following:

- No two parts can be at the same location: $\forall p, p'.L(p) \neq L(p')$.
- A device in either direction should start with a promoter:
  $\forall i.D(i) = F \rightarrow \min_{L(p).p \in \sigma(i)} \rightarrow p = P$
  $\forall i.D(i) = R \rightarrow \max_{L(p).p \in \sigma(i)} \rightarrow p = P$.
- Each gene must be preceded by a RBS sequence:
  $\forall i.D(i) = F \rightarrow \forall G \in \sigma(i).\ \exists R \in \sigma(i).L(G) = L(R) + 1$
  $\forall i.D(i) = R \rightarrow \forall G \in \sigma(i).\ \exists R \in \sigma(i).L(R) = L(G) + 1$.
- Devices do not overlap: Given the relative order $\sigma_.$ of the devices
  $\forall i.\forall p.\ p \notin \sigma(i) \rightarrow L(p) < min_{L(p).p \in \sigma(i)} \ \vee \ L(p) > max_{L(p).p \in \sigma(i)}$.
- Terminators should be the last parts of a device:
  $\forall i.D(i) = F \rightarrow \max_{L(p).p \in \sigma(i)} \rightarrow p = T$
  $\forall i.D(i) = R \rightarrow \min_{L(p).p \in \sigma(i)} \rightarrow p = T$.

To these and other in-built numerical constraints, the biocompiler then adds any user-specified constraints:

- The directive **ATGC ARRANGE** $P_1, P_2, ..., P_k$ is translated into the constraints: $L(P_1) < L(P_2) < \cdots < L(P_k)$.
- The directive **ATGC** *Dev* **DIRECTION** = REVERSE is translated into the constraint: $D(Dev) = R$.

These conditions are expressed as a system of numerical constraints that can be solved by the JaCoP CSP solver. Once the program is run, a solution (if it exists) is found, with all constraints satisfied and the positions of each parts assigned a particular value. Currently the tool only produces one (the best) solution, but we plan to extend it to export an arbitrary number of solutions, which can then all be tested through parallel wet-lab experiments. This extension is not conceptually difficult, but we need to ensure we provide solutions in a style that is easily integrated into lab protocols.

5 Biocompilation Step: RBS Selection

An RBS sequence needs to precede each gene in order to initiate translation. It is tailored for individual contexts: it depends on the CDS, the pre-sequence, and the type of host organism. The RBS sequences are computed via a stochastic process and can be very different even given the same initial conditions. The Salis RBS calculator [17] is well established as providing acceptable RBS sequences, so ATGC simply calls this tool with the specific parameters to generate this sequence which is then inserted into the emerging design.

In the Salis RBS Calculator, translation rates are specified in an arbitrary unit, and the default rate is set to 1000. ATGC also uses this default rate, but also allows it to be overridden by the user. To specify a particular initiation translation rate, users specify the value they would like to achieve:

```
ATGC TRANSLATION RATE: 50000
```

The RBS calculated for this device will have an initiation translation rate 50 times higher than the rate for a default RBS. Note that the sequence generated in this way may result in a conflict in the next step, so the RBS Calculator may be called several times during the biocompilation process.

6 Biocompilation Step: Cloning Sites Selection

ATGC also allows the user to ask for a number of *cloning sites*, non-coding fragments of DNA that can be cleaved with a *restriction enzyme* specific to each nucleotide sequence. Cloning sites are useful for *in vivo* use, *e.g.* by allowing reporters to be inserted at these locations. However, it is often difficult to find restriction enzymes that can be used because the cleavage sequences can appear in other places in the DNA, and cleaving the DNA at these sites can be catastrophic.

Users can insert a request for a number of cloning sites (say 5) within the specification of devices with a simple directive:

```
ATGC CLONING SITES: 5
```

General Strategy. ATGC attempts to find a selection of restriction enzymes that cut only at the desired location in the final sequence. Since the restriction enzymes will cut the DNA string at any occurrence of their characteristic nucleotide sequence, they have to be chosen so as not to cut the DNA sequence anywhere else. Since restriction enzymes (and therefore cloning sequences) are in limited number, it might not be possible to find enough fitting restriction enzymes given a particular sequence. If this is the case, the algorithm is allowed to change the rest of the sequence: either RBS can be recalculated or codons can be changed to suit more restriction enzymes.

This work-flow is described in Fig. 3. From the whole sequence (with all devices) and the number of required cloning sites, we first build a list of potential restriction enzymes (line 5), *i.e.* with cloning sequences that do not appear in the

whole sequence. In a first pass, in case of conflict with calculated RBSs, the RBS are re-calculated to achieve the previously set translation rates but with different sequences. This is done at most twice (to avoid infinite loops). In a second pass, if there is a conflict with some coding sequences, the biocompiler proceeds to find the optimal codon change that will both: (1) free up previously conflicting restriction enzymes, and (2) minimize the disruption to the CDS (by applying a cost dependent on the codon usage). This is done through the mapping to a constraint satisfaction problem described in the next subsection.

Given a list of non-conflicting restriction enzymes and the whole sequence, the algorithm in Fig. 4 attempts to find a suitable selection. A restriction enzyme is selected from the list one at the time, and the corresponding cloning sequence is inserted into the sequence if there is no conflict. This ensures that the added sequences are not in conflict with the newly selected sequence. The algorithm either succeeds to fit the necessary number of cloning sites, in which case the search ends, or it fails, in which case the original CDS is modified if possible.

Mapping to a Constraint Satisfaction Problem. Figure 5 shows the algorithm used for updating the codons in CDSs in order to fit more restriction enzymes. It starts by considering the restriction enzymes that only have a con-

```
1  foreach device do
2  |   get the whole sequence (including all devices)
3  |   get the number of required restriction enzymes
4  |   while RBS have been updated at most twice do
5  |   |   build a list of potential restriction enzymes
6  |   |   attempt to find enough restriction enzymes (Fig. 4)
7  |   |   if successful then
8  |   |   |   replace the place holders by the actual sequences
9  |   |   |   exit
10 |   |   else
11 |   |   |   recompute the RBS's if it might free up some restriction enzymes
12 |   |   |   try again
13 |   end
14 |   if unsuccessful then
15 |   |   attempt codon optimization via CSP (Fig. 5)
16 end
```

Fig. 3. General workflow for finding restriction enzymes

```
1 use the list of non-cutting REs
2 repeat
3 |   replace a placeholder by an actual RE in the list
4 |   if it cuts more than once then
5 |   |   remove from the list and the placeholder
6 until the list of non-cutting RE is exhausted or enough RE's have been found
```

Fig. 4. Finding fitting restriction enzymes

1 Consider REs that cut in CDS only
2 Build the list of conflicting codons and their possible alternative forms
3 Create a reified Boolean variable for each restriction enzyme:
4 **foreach** RE_i **do**
5 $\mathrm{RE}_i := \bigvee_{\text{All fitting combinations}}(\mathrm{codon}_1 = \mathrm{form}_2 \wedge \cdots \wedge \mathrm{codon}_n = \mathrm{form}_4)$
6 **end**
7 Assign the cost of changing a codon to a particular form
8 $\mathrm{cost}(\mathrm{codon}_j, \mathrm{form}_k) =$
$$\begin{cases} -\log f_{j,k} & \text{with } f_{j,k} \text{ :natural frequency of } \mathrm{form}_k \text{ for } \mathrm{codon}_j \\ 0 & \text{if } \mathrm{codon}_j \text{ has the current form: } \mathrm{form}_k \end{cases}$$
9 Set that there must be at least enough fitting restriction enzymes:
10 $\sum \mathrm{RE}_i \geq \#$ requested RE
11 And that codons are changed with minimal cost:
12 Minimise $\sum_{\text{all codons}} \mathrm{codon}_j$ takes form k

Fig. 5. Fitting restriction enzymes with codon modifications

flict with the CDS, from the list of potential enzymes built in Fig. 3 (line 5). It then makes a list of the specific codons which cause conflicts, and for each such codon, finds possible alternatives according to the genetic code (*e.g.* Alanine can have 4 forms:GCT, GCC, GCA, GCG). A reified Boolean variable RE_i is created for each candidate restriction enzymes (lines 3–6) that is true if and only if the codons each take a form such that the final sequence does not conflict with the enzyme. In other words, they are all acceptable alternative codes for the aminoacids coded by the initial codons, but which do not give rise to forbidden sequences.

The program does not simply search for codon alternatives to free up at least a minimal number of enzymes (lines 9–10). It also does it in such a way that the original sequence is minimally disrupted: each codon change is associated with a cost (lines 7–8). If the original form is kept, the cost is 0. Otherwise, the cost is $-\log(f_k)$, where f_k is the natural codon bias for the species. Thus, not changing a codon does not cost anything and changing a codon to a less common form is more expensive. The codon usage frequencies are found from the Codon Usage Database (http://www.kazusa.or.jp/codon/ [15].

The overall goal of the algorithm is thus to change codons in order to free up a minimal number of restriction enzymes (lines 9–10) while minimizing the total cost of the changes (lines 11–12).

7 Example

In this section, we build a biological AND operator with two devices, following the approach taken in [19]. The aim is to make a cell produce a fluorescent protein (GFP) when two molecules IPTG and aTc are present. Figure 6 shows the regulatory network designed to achieve this goal. The first device produces the molecules lacI and tetR. The production of GFP by the second device is

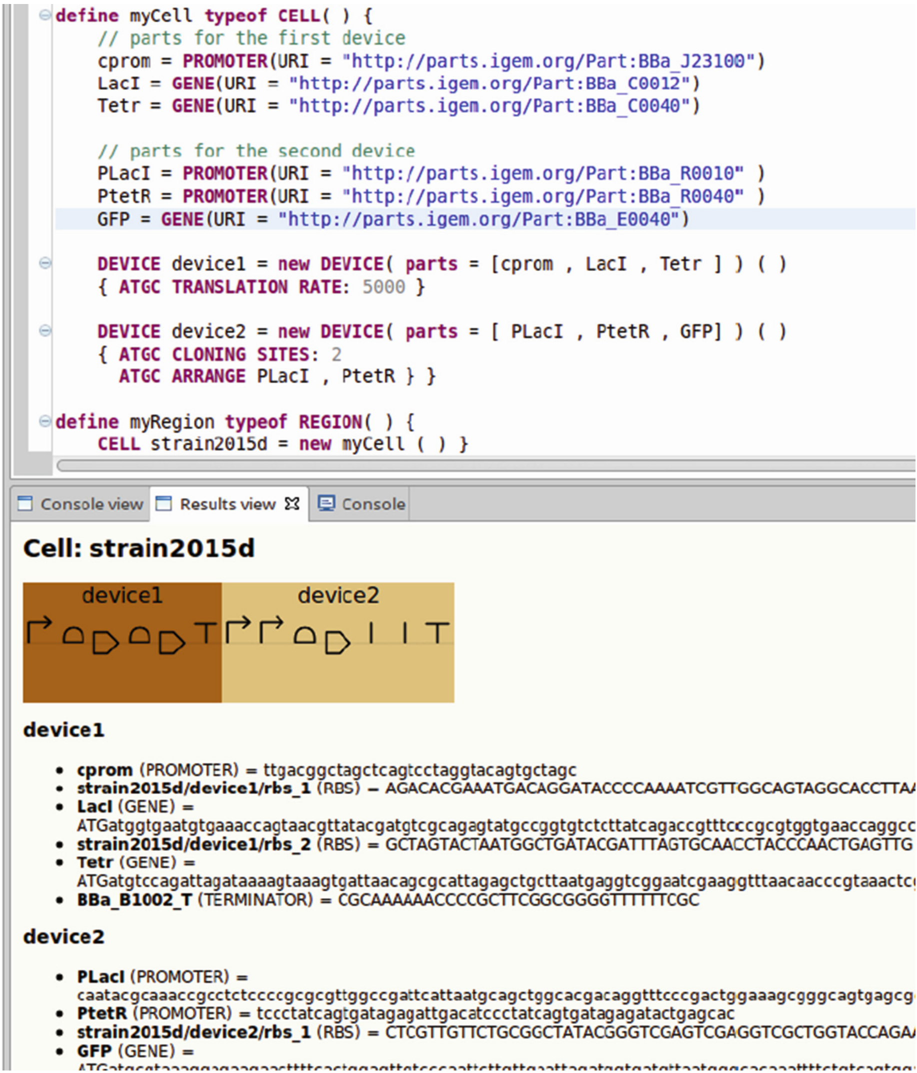


Fig. 6. Regulatory network and construct resulting from the biocompilation.

inhibited by lacI and tetR. IPTG inhibits laclI and aTc inhibits tetR. As a result, GFP will be produced only when aTc and IPTG are present. The inhibition for the second device is achieved with two promoters PLACI and PTER.

In Fig. 7 we show the directives coded by experimentalist colleagues in specifying some requirements. Typically, they wished to specify only the promoters and genes and leave other decisions to the tools. For experimental reasons, they wished to enforce the order of the two promoters. They also wanted to add a couple of cloning sites to the second device for testing purposes.

```
define myCell typeof CELL() {
      // parts for the first device
  cprom =
    PROMOTER(URI = "http://parts.igem.org/Part:BBa_J23100")
  LacI =
    GENE(URI = "http://parts.igem.org/Part:BBa_C0012")
  Tetr =
    GENE(URI = "http://parts.igem.org/Part:BBa_C0040")
      // parts for the second device
  PLacI =
    PROMOTER(URI = "http://parts.igem.org/Part:BBa_R0010")
  PtetR =
    PROMOTER(URI = "http://parts.igem.org/Part:BBa_R0040")
  GFP =
    GENE(URI = "http://parts.igem.org/Part:BBa_E0040")

  DEVICE device1 =
    new DEVICE(parts = [cprom, LacI, Tetr])()
   { ATGC_TRANSLATION_RATE: 5000 }

  DEVICE device2 =
    new DEVICE(parts = [PLacI, PtetR, GFP])()
   { ATGC_CLONING_SITES: 2
     ATGC_ARRANGE PLacI, PtetR    }        }

define myRegion typeof REGION(){
  CELL strain2015d = new myCell()          }
```

Fig. 7. User directives for example device

The biocompiler automatically completed the devices with RBS and terminators and found a functional arrangement for the parts. The sequences for the parts were obtained in the Biobricks online database. The compiler also selected 2 non-cutting restriction enzymes once the rest of the sequence had been decided. In this case, the compiler found the following two restriction enzymes: BstI (ggatcc) and Bst6I (ctcttc).

8 Interface

atgc's user interface is shown in Fig. 8. The central panel contains the editor. The left panel contains a project manager, where multiple files and folders can be created. The right hand-side panel shows the current understanding of the biocompiler for the current model. The user updates its content by clicking on Refresh model. The panel then shows an overview of the construct and enables a second button labeled Compile. Pushing this button starts the biocompiler. Every step, as well as warnings and errors, are shown in the console (bottom panel). If the compilation is successful, the final construct, with the parts arranged and

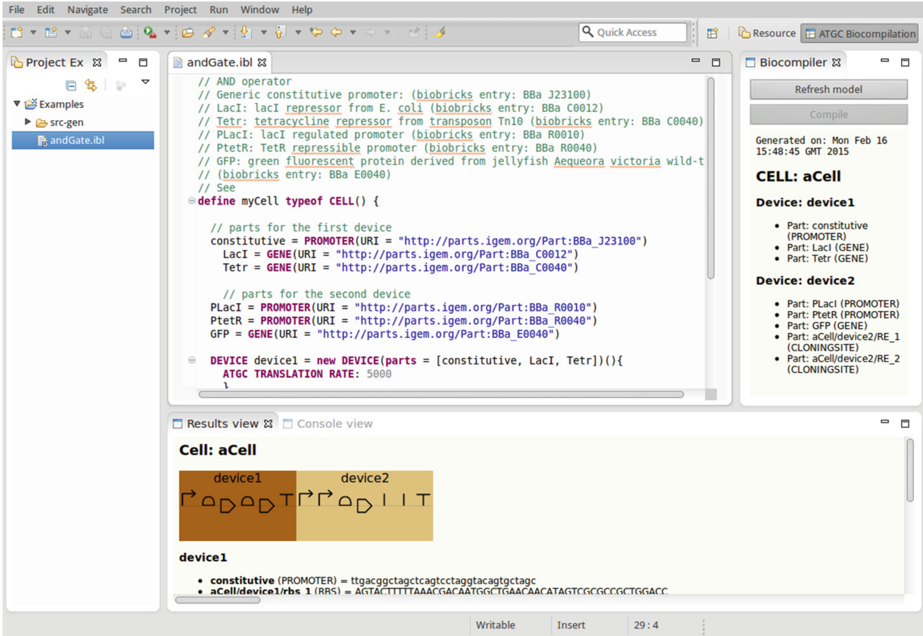


Fig. 8. User interface for ATGC with a fully-featured editor and project manager.

assigned a nucleotide sequence, is shown in the results panel. The corresponding SBOL file can be found the folder src-gen in the project manager.

ATGC's interface is built on the Eclipse platform, and is compatible with Windows, Mac OS, and Linux. The user interface language is a simple domain-specific language developed with XTEXT (http://www.eclipse.org/Xtext), which provides a text editor with keyword completion and syntax coloring.

9 Conclusion

ATGC is a biocompiler that facilitates the building of a functional string of DNA from an initial specification of functionality, an initial collection of parts, and heuristics expressed through constraints. The compilation is done in three stages: initial parts placement, RBS optimisation, and insertion of cloning sites. It uses constraint solving for dealing with conflicts and choices, through the powerful, stable and well-established CSP solver JACOP.

The approach taken here is one that favors readability of the code (by using a domain-specific language), design automation (to facilitate the access to synthetic biology for non-specialists) and extensibility. By mapping the searches and the user-specified requirements to a constraint satisfaction program, ATGC leverages the tools and techniques developed by the CSP community; adding new types of constraint or new assumptions will not require the development of new *ad-hoc* algorithms but simply the addition of extra rules to the CSP basis.

In our experience and through discussion with biologists, there is a reluctance to use specialized software tools to complete genetic designs, as users often feel hindered by the lack of control in guiding the decision making. By using a constraint-based approach where user directives are injected directly into the decision process, we leave users in the driving seat. On the other hand, ATGC is very helpful in how it handles the low-level book-keeping issues, such as finding restriction enzymes that are compatible with the rest of the design.

ATGC is also part of an upcoming platform for synthetic biology, which will integrate the modeling, verification and biocompilation into a unified language and system. All three aspects will be seamlessly intertwined to produce a one-stop shop for designing new organisms *in silico*.

Acknowledgments. This research was supported by EPSRC through grant EP/I03157X/1, *Towards Programmable Defensive Bacterial Coatings and Skins*. We are grateful for the collaboration within the Roadblock consortia.

References

1. Beal, J., Lu, T., Weiss, R.: Automatic compilation from high-level biologically-oriented programming language to genetic regulatory networks. PLoS ONE **6**(8), e22490 (2011). doi:10.1371/journal.pone.0022490
2. Bilitchenko, L., et al.: Eugene: a domain specific language for specifying and constraining synthetic biological parts, devices, and systems. PLoS ONE **6**(4), e18882 (2011). doi:10.1371/journal.pone.0018882
3. Biofab: Data Access Web Service (2015). http://biofab.synberc.org/data
4. Cai, Y., et al.: A syntactic model to design and verify synthetic genetic constructs derived from standard biological parts. Bioinformatics **23**(20), 2760–2767 (2007). doi:10.1093/bioinformatics/btm446
5. Church, G.M., Elowitz, M.B., Smolke, C.D., Voigt, C.A., Weiss, R.: Realizing the potential of synthetic biology. Nat. Rev. Mol. Cell Biol. **15**(4), 289–294 (2014). doi:10.1038/nrm3767
6. Clancy, K., Voigt, C.A.: Programming cells: towards an automated 'genetic compiler'. Curr. Opin. Biotechnol. **21**(4), 572–581 (2010). doi:10.1016/j.copbio.2010.07.005
7. Cooling, M.T., et al.: Standard virtual biological parts: a repository of modular modeling components for synthetic biology. Bioinformatics **26**(7), 925–931 (2010). http://bioinformatics.oxfordjournals.org/content/26/7/925.abstract
8. Elowitz, M., Lim, W.A.: Build life to understand it. Nature **468**(7326), 889–890 (2010). doi:10.1038/468889a
9. Freemont, P.S., Kitney, R.I., Baldwin, G., Bayer, T., Dickinson, R., Ellis, T., Polizzi, K., Stan, G.B., Kitney, R.I.: Synthetic Biology - A Primer. World Scientific Publishing, London (2012). http://www.worldcat.org/isbn/1848168632
10. Galdzicki, M., et al.: The synthetic biology open language (SBOL) provides a community standard for communicating designs in synthetic biology. Nat. Biotechnol. **32**(6), 545–550 (2014). doi:10.1038/nbt.2891
11. Hannon, G.: RNA interference. Nature **418**(6894), 244–251 (2002)
12. Hillson, N.J., et al.: j5 DNA assembly design automation software. ACS Synth. Biol. **1**(1), 14–21 (2011). doi:10.1021/sb2000116

13. iGem: Parts Registry (2015). http://partsregistry.org/
14. Kuchcinski, K.: Constraints-driven scheduling and resource assignment. ACM Trans. Des. Autom. Electron. Syst. **8**(3), 355–383 (2003). doi:10.1145/785411.785416
15. Nakamura, Y., et al.: Codon usage tabulated from the international DNA sequence databases. Nucleic Acids Res. **24**(1), 214–215 (1996). http://www.ncbi.nlm.nih.gov/pmc/articles/PMC145571/
16. Paddon, C.J., et al.: High-level semi-synthetic production of the potent antimalarial artemisinin. Nature **496**, 528–532 (2013). doi:10.1038/nature12051
17. Salis, H.M.: The ribosome binding site calculator. Methods Enzymol. **498**, 19–42 (2011). doi:10.1016/b978-0-12-385120-8.00002-4. Elsevier
18. Schirmer, A., Rude, M.A., Li, X., Popova, E., del Cardayre, S.B.: Microbial biosynthesis of alkanes. Science **329**(5991), 559–562 (2010). doi:10.1126/science.1187936
19. Tamsir, A., Tabor, J.J., Voigt, C.A.: Robust multicellular computing using genetically encoded NOR gates and chemical /'wires/'. Nature **469**(7329), 212–215 (2011). doi:10.1038/nature09565

Molecular Recognition/Prediction

P2RANK: Knowledge-Based Ligand Binding Site Prediction Using Aggregated Local Features

Radoslav Krivák(✉) and David Hoksza

FMP, Department of Software Engineering, Charles University in Prague,
Malostranské nám. 25, 118 00 Prague, Czech Republic
{krivak,hoksza}@ksi.mff.cuni.cz

Abstract. The knowledge of protein-ligand binding sites is vital prerequisite for any structure-based virtual screening campaign. If no prior knowledge about binding sites is available, the ligand-binding site prediction methods are the only way to obtain the necessary information. Here we introduce P2RANK, a novel machine learning-based method for prediction of ligand binding sites from protein structure. P2RANK uses Random Forests learner to infer ligandability of local chemical neighborhoods near the protein surface which are represented by specific near-surface points and described by aggregating physico-chemical features projected on those points from neighboring protein atoms. The points with high predicted ligandability are clustered and ranked to obtain the resulting list of binding site predictions. The new method was compared with a state-of-the-art binding site prediction method Fpocket on three representative datasets. The results show that P2RANK outperforms Fpocket by 10 to 20 % points on all the datasets. Moreover, since P2RANK does not rely on any external software for computation of various complex features, such as sequence conservation scores or binding energies, it represents an ideal tool for inclusion into future structural bioinformatics pipelines.

Keywords: Ligand-binding site prediction · Protein structure · Molecular recognition · Machine learning · Random forest

1 Introduction

1.1 Motivation

Prediction of ligand binding sites from protein structure has many applications, ranging from use in rational drug design [30,44], drug side-effects prediction [42] to elucidation of protein function [18]. Of special interest is the application in structure based virtual screening (SBVS) pipelines. In most types of SBVS, docking algorithms are used to predict possible ligand-binding interactions. It is recommended to focus docking to a protein cavity of interest to limit the search space of possible conformations. In the cases where there is no a priori

© Springer International Publishing Switzerland 2015
A.-H. Dediu et al. (Eds.): AlCoB 2015, LNBI 9199, pp. 41–52, 2015.
DOI: 10.1007/978-3-319-21233-3_4

information with regard to which protein regions to focus on (e.g. confirmed active sties), it may be necessary to perform blind docking which scans the whole protein surface. Compared to local docking it is generally less accurate and significantly more time consuming, which limits the size of compound libraries that is possible to screen [34]. Alternatively, ligand binding site prediction can be employed in such scenarios to generate and prioritize the locations on which to center subsequent docking procedure [23]. In a similar manner, binding site prediction could also be of great use in a related task of structure-based target prediction (or so called inverse virtual screening) [37]. As a result of the structural genomic efforts [28], many protein structures still lack functional annotation and even if it is present, it may not be complete. We believe that accurate ligand binding site prediction methods (in combination with validation via docking) can help to discover new and potentially useful allosteric binding sites.

1.2 Existing Methods

Many different ligand binding site prediction methods based on various strategies have been already developed. The first dedicated method was proposed in 1992 [26] and the recent increase of interest in the field, presumably due to rapid increase in number of available protein structures, is indicated by the number of recently published reviews [6,13,23,25,30]. Several categories of methods (or rather distinctive approaches) have been recognized. We present them together with their representative examples, although in reality the actual methods may use a combination of those approaches:

– **Geometrical Methods.** Methods focused mainly on the algorithmic side of the problem of finding concave pockets and clefts on the surface of a 3D structure [12,15,41], some of them incorporating additional physico-chemical information like polarity or charge [21,24].
– **Energetic Methods.** Methods that build on the approximations of free energy potentials by force fields, placing probes around the protein surface and calculating binding energies [1,10,22,27,36].
– **Evolutionary Methods.** Algorithms that make use of sequence conservation estimates (functional residues are more evolutionary conserved) [5,15], or protein threading (fold recognition from sequence) to find set of evolutionary related structures and determine their common binding sites [4,38].
– **Knowledge Based Methods.** Methods that try to capture and exploit the knowledge about protein-ligand binding that is implicitly stored in sequence and structural databases by means of statistical inference [34]. Although several residue-centric studies focused on classification of ligand binding residues have been published [7,16,32], there are not many examples of complete prediction methods using this approach that would produce putative binding sites as such [33].
– **Consensus Methods.** Those are meta approaches that combine the results of other methods [6,14,43].

In studies that introduced respective methods relatively high identification success rates have been reported (usually more than 70 % considering only the first

predicted binding site). However, the results of the only independent benchmarking study [6] suggest that accuracy of many of the methods may not be as good as previously believed (closer to 50 %). It showed that there is still a need for more accurate methods and thus an opportunity for improvement by examining new approaches to binding site prediction. On a practical side, only few of the methods are available for download as ready to use free software packages.

2 Materials and Methods

2.1 Method Outline

In this paper we are introducing a novel method for prediction of ligand binding sites form a protein structure. Method is named P2RANK and it represents an evolutionary improvement of our pocket ranking method PRANK [19] which could only be used to reorder output of other pocket prediction methods. By not relying on other methods to generate putative binding site locations and thus making it a full-fledged method that can generate predictions itself has led to a marked improvements in identification success rates.

The method takes a PDB structure as an input and outputs a ranked list of predicted ligand binding sites defined by a set of points The following list outlines the proposed method:

1. Generating a set of regularly spaced points lying on a protein's Connolly surface (referred to as *Connolly points*).
2. Calculating feature descriptors of Connolly points based on their local chemical neighborhood:
   (a) computing property vectors for protein's solvent exposed atoms,
   (b) projecting distance weighted properties of the adjacent protein atoms onto Connolly points,
   (c) computing additional features describing Connolly point neighborhood.
3. Predicting ligandability score of Connolly points by Random Forest classifier.
4. Clustering points with high ligandability score and thus forming pocket predictions.
5. Ranking predicted pockets by cumulative ligandability score of their points.

Individual steps are described in greater detail in following paragraphs. For visualization of Connolly points see Fig. 1. One possible way how to look at our protein surface representation is that protein solvent exposed atoms produce potential fields for every feature (e.g. hydrophobicity, aromaticity, ...), and Connolly points are sampling values from those fields at places near the protein surface, which are likely to harbor potential ligand atoms.

Connolly Points. (1) Position of Connolly points is generated by a fast numerical algorithm [9] implemented in CDK library [39]. The algorithm produces a set of a more or less regularly spaced points lying on a Connolly surface of the

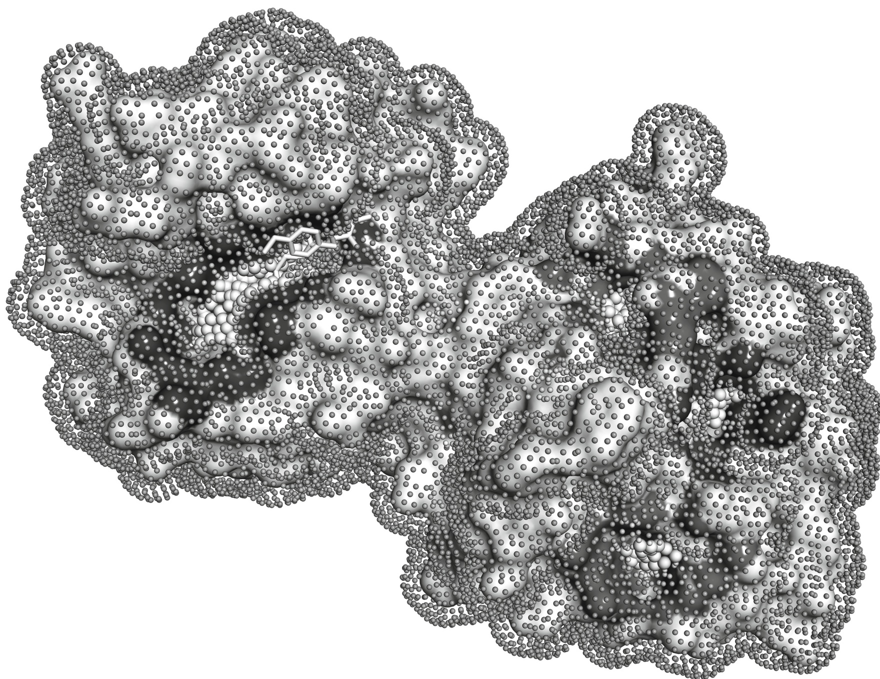


Fig. 1. Connolly Points. Protein (1FBL) is covered in a layer of points lying on a Connolly surface. Each point represents its local chemical neighborhood and is colored according to its predicted ligandability score (*dark=0/white=1*). Points deemed highly ligandable by a threshold (*displayed enlarged*) are clustered to form predicted pockets (*highlighted by coloring adjacent protein surface by darker colors*). In this case, the largest predicted pocket (*shown on the left*) is indeed a correctly identified true binding site that binds a ligand. Visible are three other smaller predicted pockets, or rather hotspots (*shown on the right*). Cumulative ligandability score of their respective points is lower, therefore they will be ranked lower than the true pocket on the resulting list of predicted binding sites.

protein. Solvent radius used is 1.6 Å(this value as well as other arbitrary parameters was optimized, see Results section). The density of the points depend on an integer parameter tessellation level with default value 2 that produces points with approximately 1.5 Å spacing.

Feature Representation. (2) For each Connolly point a feature vector (CFV) that describes its local physico-chemical neighborhood is calculated. Before calculating CFVs, each solvent exposed heavy atom of the protein is assigned atomic feature vector (AFV) that describes given atom. CFV of a given Connolly point is then calculated by aggregating AFVs of neighboring atoms and adding additional Connolly point features (XFV), i.e. extra features that are not defined for atoms.

Atomic neighborhood of Connolly point P is defined as:

$$\mathrm{A}(P) = \{\text{solvent exposed heavy protein atoms within } r{=}6\,\text{Å radius around } P\} \tag{1}$$

The following aggregation function is used to project AFVs onto the Connolly points and calculate CFV for point P:

$$\mathrm{CFV}(P) = \sum_{\mathrm{A}_i \in \mathrm{A}(P)} \mathrm{AFV}\,(\mathrm{A}_i) \cdot w(\mathrm{dist}(P, \mathrm{A}_i)) \quad || \quad \mathrm{XFV}(P), \tag{2}$$

where || is the operator of concatenation, XFV is a vector of additional features specific to Connolly points and w is a distance weight function:

$$w(d) = 1 - d/6. \tag{3}$$

AFV that describes protein atoms consists of two types of features: residue level features (inherited by all atoms of a given residue) and atomic level features. Residue level features include e.g. physico-chemical properties of standard amino acids or hydropathy index of amino acids [20]. Examples of atomic features are pharmacophore-related labels of atoms adopted from VolSite druggability prediction study [8] or statistical ligand-binding propensities of amino acid atoms [17]. Most of the features are table features defined either for 20 standard amino acids or their atom types (ALA.CA, ALA.CB,...). Exception to this is temperature factor, taken directly from PDB file, which can be different for each atom. Extra Connolly point features (XFV), which are not defined for atoms, include the number of neighboring H-bond donors and acceptors and protrusion index [31]. Protrusion is defined as a density of a protein atoms around the point and is calculated using larger neighborhood cutoff radius (10 Å). Altogether, CFV consists of 34 features. For their complete list and description we refer the reader to [19].

Classification. (3) Machine learning approach was used to classify Connolly points as ligandable/unligandable from their feature vectors. In general, output of a binary classifier is a number between 0 and 1 that represents certainty of a trained model that the classified instance belongs to the particular class (here $\text{class}_1 = \text{ligandable}$). Commonly, a threshold optimizing certain metric is chosen and applied to produce binary output. In our case we decided to work directly with output score (rather than binary output) which we refer to as a predicted ligandability score (LS).

In theory any classification algorithm can be employed at this stage. After preliminary experiments with several machine learning methods we decided to adopt Random Forests [3] as our predictive modelling tool of choice. Random Forests is an ensemble of trees created by using bootstrap samples of training data and random feature selection in tree induction [40]. In comparison with other machine learning approaches, Random Forests are characterized by an outstanding speed (both in learning and execution phase) and generalization ability [3]. Additionally, Random Forests is robust to the presence of a large

number of irrelevant variables; it does not require their prior scaling [29] and can cope with complex interaction structures as well as highly correlated variables [2].

Random Forests algorithm has 3 basic hyperparameters: number of trees, maximum tree depth and a number of random features used to construct each tree. To train the final model we used Random Forest with 100 trees of unlimited depth, each tree built considering 6 features. Model is trained on a dataset of ligandable and unligandable points that come from PDB structures with known ligand positions. To train our final model which we distribute with our software we used protein/ligand complexes from CHEN11 dataset (see Datasets section).

Clustering. (4) To prepare putative binding site predictions we first filter out Connolly points that have ligandability score lower than give threshold (default $t = 0.35$) and apply single linkage clustering procedure on the rest (default cutoff distance $d = 3$ Å). Predicted pocket is then defined by the set of Connolly points in a cluster. For each pocket we compute associated set of protein solvent exposed atoms that form putative ligand binding surface patch. We include into the output all pockets that are defined by 3 or more Connolly points. This is rather low threshold, which results to many small predicted pockets that are most probably not true binding sites. However, this was a deliberate choice as thanks to an efficient ranking algorithm those small pockets will always be ranked at the bottom of the list. Nevertheless, those small clusters might be still interesting for visual inspection (possibly forming hotspots for protein-protein interactions or peptide binding).

Ranking. (5) Each pocket is assigned a score calculated as the sum of squared ligandability scores of all of the Connolly points P_i that define the pocket:

$$\text{PScore} = \sum_i \left(\text{LS}(P_i)\right)^2 \tag{4}$$

Squaring of the ligandability scores puts more emphasis on the points with ligandability score closer to 1 (i.e. points that were classified as ligandable with higher certainty). Score defined in such way will roughly order pockets by size but will favor smaller pockets with strongly ligandable points before larger pockets with weakly ligandable points. The very last step of the algorithm involves reordering the putative pockets in the decreasing order of their *PScores*.

Ranking of the predicted pockets is important for prioritization of subsequent efforts, e.g. docking or visual inspection. Pocket ranking is also pivotal in the context of evaluation and comparison of different ligand-binding site prediction methods, where only pockets with highest ranks are considered (usually Top-1 and Top-3). If it was not so, then the simplistic and obviously useless method that returns many binding sites covering all of the protein surface (most of them false positives) would achieve 100 % identification success rate.

Table 1. Datasets.

Dataset	Proteins	Ligands	Avg. ligands	Avg. lig. atoms	Avg. prot. atoms
CHEN11	251	374	1.49	26.9	1836
JOINED	589	689	1.17	22.5	2400
HOLO4K	4543	11511	2.54	22.4	3888

2.2 Datasets

For the purpose of training an evaluation of our method we have used three datasets:

- **CHEN11** – dataset introduced in benchmarking study [6]. A non-redundant dataset constructed in a way so that each SCOP family has one typical representative.
- **JOINED** – consists of structures from several smaller datasets used in previous studies (48bound/unbound structures [15], ASTEX [11], 198 drug targets [43], 210 bound proteins [14]) joined into one larger dataset.
- **HOLO4K** – large dataset of protein-ligand complexes currently available in PDB based on the list published in [35].

Details of the datasets are presented in Table 1.

3 Experimental Evaluation

3.1 Evaluation Measures

To evaluate predictive performance of our method and compare it with Fpocket we have used methodology based on ligand-centric counting and D_{CA}(distance between the center of the pocket and any ligand atom) pocket identification criterion with 4 Å threshold. Ligand-centric counting means, that for every relevant ligand in the dataset, its binding site must be correctly predicted for a method to achieve 100 % identification success rate. Connected to this is the use of Top-n and Top-(n + 2) rank cutoffs where n is the number of ligands in a protein structure where evaluated ligand comes from (for proteins with only one ligand this corresponds to usual Top-1 and Top-3 cutoffs). This evaluation methodology is the same as used in benchmarking study [6].

Because of the great differences in evaluation protocols used to produce results of previously published methods, we are of the opinion that the only way how to accurately compare ligand binding site prediction methods is to preform experiments and compare methods side by side using the same methodology (as opposed to using results taken from literature even if experiments are performed on the same dataset). Aforementioned differences in protocols include: different identification criteria (D_{CA}/D_{CC}(center-center distance)/variously defined

volume overlap criteria), different counting strategies (ligand-centric/protein-centric), different valid ligand selection and different rank cutoffs considered (Top-1/Top-3/Top-4/Top-n). Experimentally comparing ligand binding site prediction methods is complicated and lengthy effort involving many technical hurdles. Nevertheless, instead of reporting here the results of our method side-by-side with results taken from literature we compare it thoroughly on large datasets with Fpocket and our previous ranking method (which improves results of Fpocket by reordering its output). The significance of our results with respect to other methods can be inferred by comparing our presented results with mentioned independent benchmarking study which includes Fpocket [6].

3.2 Results

We have evaluated our method on 3 different datasets and compared its predictive performance with a well-known Fpocket method. Results are summarized in Table 2. Our method comes with a pre-trained classification model that was trained on the CHEN11 dataset. On JOINED and HOLO4K datasets we report results achieved using this default model and on CHEN11 dataset averaged results from 10 runs of 5-fold cross-validation. Success rates (percentage of correctly predicted binding sites) are reported for Top-n and Top-(n + 2) cutoffs from the top of the ranked list.

It is apparent that P2RANK outperforms Fpockets on all dataset by a large margin. The difference is most visible comparing results for Top-1 cutoffs. For a difficult CHEN11 dataset a difference amounts to more than 10 % in nominal terms and almost 20 for other datasets. Detailed results on HOLO4K dataset are compared in Fig. 2.

3.3 Optimization and Tradeoffs

Extensive optimization of practically all arbitrary parameters of the algorithm (distance cutoffs, thresholds, ...) was performed to establish optimal default values. Parameters were optimized with respect to the success rate achieved on the JOINED dataset. This was done to avoid bias towards CHEN11 dataset so the

Table 2. Results: the numbers represent identification success rate [%] measured by D_{CA} criterion with 4 Å threshold considering only pockets ranked at the top of the list (n is the number of ligands in considered structure). *average results of 10 independent 5-fold cross-validation runs.

Dataset	Top-n			Top-(n + 2)		
	Fpocket	PRANK	P2RANK	Fpocket	PRANK	P2RANK
CHEN11	47.8	58.6*	**59.2***	61.5	**68.1***	65.9*
JOINED	51.1	64.7	**71.6**	68.9	76.1	**78.7**
HOLO4K	45.2	53.6	**63.9**	55.1	61.6	**69.8**

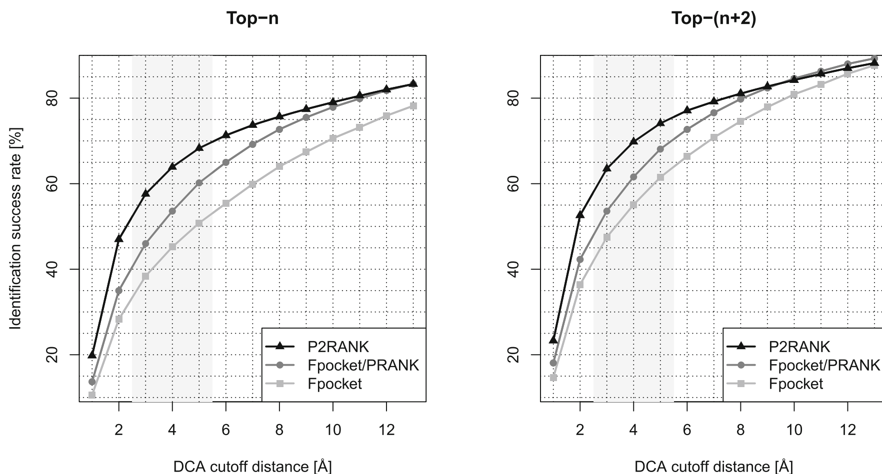


Fig. 2. Results on HOLO4K dataset for different D_{CA}cutoff distances.

cross-validation results on the CHEN11 could be compared with benchmarking study which used the same evaluation criteria [6]. We have also refrained from tweaking the parameters with respect to HOLO4K dataset, so that the results on this dataset present unbiased estimate of the algorithm's true identification success rate on unknown input.

Several parameters present a tradeoff between the time and space complexity of the algorithm and its accuracy. Among those is the number of trees in Random Forest model and tessellation level influencing density of the generated Connolly points. Ultimately we have decided to use 100 trees (using 10× more trees leads only to marginal improvements) and tessellation level of 2 (using higher levels leads to some improvements but also unproportionally longer running times).

4 Conclusion

In the present paper we have proposed P2RANK, a novel method for ligand-binding site prediction based on classification of points lying on a protein's Connolly surface. Each point represents potential location of a binding ligand atom and is described by a feature vector generated from its spatial neighborhood. Ligandability score is predicted for each point by a Random Forests classifier and points with higher score are clustered forming predicted pockets. Pockets are then ranked according to the cumulative ligandability score of their points.

To our knowledge this is a first time a machine learning approach was used in such a manner for ligand binding site prediction. Methods that applied machine learning in this context focused on classification of ligand-binding residues i.e. were residue-centric. Unfortunately, most of those residue-centric studies were focused on a successful classification of residues themselves and not on predicting ligand binding sites as such.

We showed on several datasets that P2RANK significantly improves identification success over the state of the art method Fpocket, while still being reasonably fast to be used on large datasets. Like Fpocket, P2RANK is a standalone program that is ready to be used as is, without depending on any external data or programs or secondary inputs such as pre-computed sequence conservation scores, forcefield calculations or threading template libraries. We believe that it is a viable method with a potential to become useful part of structural bioinformatics toolkit.

Acknowledgments. This work was supported by the Czech Science Foundation grant 14-29032P and by project SVV-2015-260222 and by the Charles University in Prague, project GA UK No. 174615.

References

1. An, J., Totrov, M., Abagyan, R.: Pocketome via comprehensive identification and classification of ligand binding envelopes. Mol. Cell. Proteomics **4**(6), 752–761 (2005)
2. Boulesteix, A.L., Janitza, S., Kruppa, J., K-nig, I.R.: Overview of random forest methodology and practical guidance with emphasis on computational biology and bioinformatics. Wiley Interdisc. Rev. Data Min. Knowl. Discov. **2**(6), 493–507 (2012)
3. Breiman, L.: Random forests. Mach. Learn. **45**(1), 5–32 (2001)
4. Brylinski, M., Skolnick, J.: A threading-based method (FINDSITE) for ligand-binding site prediction and functional annotation. Proc. Natl. Acad. Sci. U.S.A **105**(1), 129–134 (2008)
5. Capra, J.A., Laskowski, R.A., Thornton, J.M., Singh, M., Funkhouser, T.A.: Predicting protein ligand binding sites by combining evolutionary sequence conservation and 3d structure. PLoS Comput. Biol. **5**(12), e1000585 (2009)
6. Chen, K., Mizianty, M., Gao, J., Kurgan, L.: A critical comparative assessment of predictions of protein-binding sites for biologically relevant organic compounds. Structure (London, England: 1993) **19**(5), 613–621 (2011)
7. Chen, P., Huang, J.Z., Gao, X.: Ligandrfs: random forest ensemble to identify ligand-binding residues from sequence information alone. BMC Bioinform. **15**(15), S4 (2014)
8. Desaphy, J., Azdimousa, K., Kellenberger, E., Rognan, D.: Comparison and druggability prediction of protein-ligand binding sites from pharmacophore-annotated cavity shapes. J. Chem. Inf. Model. **52**(8), 2287–2299 (2012)
9. Eisenhaber, F., Lijnzaad, P., Argos, P., Sander, C., Scharf, M.: The double cubic lattice method: efficient approaches to numerical integration of surface area and volume and to dot surface contouring of molecular assemblies. J. Comput. Chem. **16**(3), 273–284 (1995)
10. Ghersi, D., Sanchez, R.: EasyMIFS and SiteHound: a toolkit for the identification of ligand-binding sites in protein structures. Bioinformatics (Oxford, England) **25**(23), 3185–3186 (2009)
11. Hartshorn, M., Verdonk, M., Chessari, G., Brewerton, S., Mooij, W., Mortenson, P., Murray, C.: Diverse, high-quality test set for the validation of protein-ligand docking performance. J. Med. Chem. **50**(4), 726–741 (2007)

12. Hendlich, M., Rippmann, F., Barnickel, G.: LIGSITE: automatic and efficient detection of potential small molecule-binding sites in proteins. J. Mol. Graph. Model. **15**(6), 359–363, 389 (1997)
13. Henrich, S., Outi, S., Huang, B., Rippmann, F., Cruciani, G., Wade, R.: Computational approaches to identifying and characterizing protein binding sites for ligand design. J. Mol. Recogn. (JMR) **23**(2), 209–219 (2010)
14. Huang, B.: MetaPocket: a meta approach to improve protein ligand binding site prediction. Omics J. Integr. Biol. **13**(4), 325–330 (2009)
15. Huang, B., Schroeder, M.: Ligsitecsc: predicting ligand binding sites using the connolly surface and degree of conservation. BMC Struct. Biol. **6**(1), 19 (2006). http://www.biomedcentral.com/1472-6807/6/19
16. Kauffman, C., Karypis, G.: Librus: combined machine learning and homology information for sequence-based ligand-binding residue prediction. Bioinformatics (Oxford, England) **25**(23), 3099–3107 (2009). http://bioinformatics.oxfordjournals.org/cgi/pmidlookup?view=long&pmid=19786483
17. Khazanov, N.A., Carlson, H.A.: Exploring the composition of protein-ligand binding sites on a large scale. PLoS Comput. Biol. **9**(11), e1003321 (2013)
18. Konc, J., Janei, D.: Binding site comparison for function prediction and pharmaceutical discovery. Curr. Opin. Struct. Biol. **25**, 34–39 (2014)
19. Krivak, R., Hoksza, D.: Improving protein-ligand binding site prediction accuracy by classification of inner pocket points using local features. J. Cheminformatics **7**(1), 12 (2015). http://www.jcheminf.com/content/7/1/12
20. Kyte, J., Doolittle, R.F.: A simple method for displaying the hydropathic character of a protein. J. Mol. Biol. **157**(1), 105–132 (1982). http://www.sciencedirect.com/science/article/pii/0022283682905150
21. Labute, P., Santavy, M.: Locating binding sites in protein structures (2001). http://www.chemcomp.com/journal/sitefind.htm. Accessed 16 April 2015
22. Laurie, A., Jackson, R.: Q-SiteFinder: an energy-based method for the prediction of protein-ligand binding sites. Bioinformatics (Oxford, England) **21**(9), 1908–1916 (2005)
23. Laurie, A., Jackson, R.: Methods for the prediction of protein-ligand binding sites for structure-based drug design and virtual ligand screening. Curr. Protein Pept. Sci. **7**(5), 395–406 (2006)
24. Le Guilloux, V., Schmidtke, P., Tuffery, P.: Fpocket: an open source platform for ligand pocket detection. BMC Bioinform. **10**(1), 168 (2009). http://www.biomedcentral.com/1471-2105/10/168
25. Leis, S., Schneider, S., Zacharias, M.: In silico prediction of binding sites on proteins. Curr. Med. Chem. **17**(15), 1550–1562 (2010)
26. Levitt, D.G., Banaszak, L.J.: Pocket: a computer graphies method for identifying and displaying protein cavities and their surrounding amino acids. J. Mol. Graph. **10**(4), 229–234 (1992). http://www.sciencedirect.com/science/article/pii/026378559280074N
27. Morita, M., Nakamura, S., Shimizu, K.: Highly accurate method for ligand-binding site prediction in unbound state (apo) protein structures. Proteins **73**(2), 468–479 (2008)
28. Nair, R., Liu, J., Soong, T.T., Acton, T., Everett, J., Kouranov, A., Fiser, A., Godzik, A., Jaroszewski, L., Orengo, C., et al.: Structural genomics is the largest contributor of novel structural leverage. J. Struct. Funct. Genom. **10**(2), 181–191 (2009)
29. Nayal, M., Honig, B.: On the nature of cavities on protein surfaces: application to the identification of drug-binding sites. Proteins **63**(4), 892–906 (2006)

30. Pérot, S., Sperandio, O., Miteva, M., Camproux, A., Villoutreix, B.: Druggable pockets and binding site centric chemical space: a paradigm shift in drug discovery. Drug Discovery Today **15**(15–16), 656–667 (2010)
31. Pintar, A., Carugo, O., Pongor, S.: Cx, an algorithm that identifies protruding atoms in proteins. Bioinformatics **18**(7), 980–984 (2002)
32. Qiu, Z., Qin, C., Jiu, M., Wang, X.: A simple iterative method to optimize protein-ligand-binding residue prediction. J. Theor. Biol. **317**, 219–223 (2013)
33. Qiu, Z., Wang, X.: Improved prediction of protein ligand-binding sites using random forests. Protein Pept. Lett. **18**(12), 1212–1218 (2011). http://www.ingentaconnect.com/content/ben/ppl/2011/00000018/00000012/art00005
34. Rognan, D.: Docking Methods for Virtual Screening: Principles and Recent Advances, pp. 153–176. Wiley, Weinheim (2011). http://dx.doi.org/10.1002/9783527633326.ch6
35. Schmidtke, P., Souaille, C., Estienne, F., Baurin, N., Kroemer, R.: Large-scale comparison of four binding site detection algorithms. J. Chem. Inf. Model. **50**(12), 2191–2200 (2010)
36. Schneider, S., Zacharias, M.: Combining geometric pocket detection and desolvation properties to detect putative ligand binding sites on proteins. J. Struct. Biol. **180**(3), 546–550 (2012)
37. Schomburg, K., Bietz, S., Briem, H., Henzler, A., Urbaczek, S., Rarey, M.: Facing the challenges of structure-based target prediction by inverse virtual screening. J. Chem. Inf. Model. **54**(6), 1676–1686 (2014)
38. Skolnick, J., Brylinski, M.: FINDSITE: a combined evolution/structure-based approach to protein function prediction. Briefings Bioinform. **10**(4), 378–391 (2009)
39. Steinbeck, C., Han, Y., Kuhn, S., Horlacher, O., Luttmann, E., Willighagen, E.: The chemistry development kit (CDK): an open-source java library for chemo- and bioinformatics. J. Chem. Inf. Comput. Sci. **43**(2), 493–500 (2003). pMID: 12653513
40. Svetnik, V., Liaw, A., Tong, C., Culberson, J.C., Sheridan, R.P., Feuston, B.P.: Random forest: a classification and regression tool for compound classification and qsar modeling. J. chem. Inf. Comput. Sci. **43**(6), 1947–1958 (2003)
41. Weisel, M., Proschak, E., Schneider, G.: Pocketpicker: analysis of ligand binding-sites with shape descriptors. Chem. Central J. **1**(1), 7 (2007). http://journal.chemistrycentral.com/content/1/1/7
42. Xie, L., Xie, L., Bourne, P.E.: Structure-based systems biology for analyzing off-target binding. Curr. Opin. Struct. Biol. **21**(2), 189–199 (2011)
43. Zhang, Z., Li, Y., Lin, B., Schroeder, M., Huang, B.: Identification of cavities on protein surface using multiple computational approaches for drug binding site prediction. Bioinformatics (Oxford, England) **27**(15), 2083–2088 (2011)
44. Zheng, X., Gan, L., Wang, E., Wang, J.: Pocket-based drug design: exploring pocket space. AAPS J. **15**, 228–241 (2012)

StreaM - A Stream-Based Algorithm for Counting Motifs in Dynamic Graphs

Benjamin Schiller[1(✉)], Sven Jager[2], Kay Hamacher[2,3], and Thorsten Strufe[1]

[1] Privacy and Data Security, Department of Computer Science,
TU Dresden, Dresden, Germany
{benjamin.schiller1,thorsten.strufe}@tu-dresden.de

[2] Computational Biology and Simulation, Department of Biology,
TU Darmstadt, Darmstadt, Germany
{jager,hamacher}@bio.tu-darmstadt.de

[3] Department of Physics, Department of Computer Science,
TU Darmstadt, Darmstadt, Germany

Abstract. Determining the occurrence of motifs yields profound insight for many biological systems, like metabolic, protein-protein interaction, and protein structure networks. Meaningful spatial protein-structure motifs include enzyme active sites and ligand-binding sites which are essential for function, shape, and performance of an enzyme. Analyzing their dynamics over time leads to a better understanding of underlying properties and processes. In this work, we present *StreaM*, a stream-based algorithm for counting undirected 4-vertex motifs in dynamic graphs. We evaluate *StreaM* against the four predominant approaches from the current state of the art on generated and real-world datasets, a simulation of a highly dynamic enzyme. For this case, we show that *StreaM* is capable to capture essential molecular protein dynamics and thereby provides a powerful method for evaluating large molecular dynamics trajectories. Compared to related work, our approach achieves speedups of up to 2,300 times on real-world datasets.

1 Introduction

Motifs, the basic building blocks of any complex network, have been widely studied in the past [31]. They are often used in the analysis of biological networks, most notably protein-protein interactions [1,8,9,27,36], DNA [18,40], cellular networks [21], and protein structure networks [22]. Because of their general applicability, they have also been studied and used in other fields, e.g., to understand patterns in real and generated languages [5], to analyze and improve Peer-to-Peer networks [15,26], and for the generation of Internet PoP maps [12,13]. Recently, temporal motifs that describe how interactions between components change over time have been investigated [17,23] as well as degree-based signatures [30].

The problem of counting motifs in a static graph has been widely studied. The first approaches like ProMotif [16], mfinder [20], MAVisto [39], and NeMoFinder [7] provided tools to count motifs of small sizes but performed

© Springer International Publishing Switzerland 2015
A.-H. Dediu et al. (Eds.): AlCoB 2015, LNBI 9199, pp. 53–67, 2015.
DOI: 10.1007/978-3-319-21233-3_5

rather poorly, especially on larger graphs. This changed with the development of Fanmod [43], a very efficient algorithm that all recent approaches have been compared to. New algorithms like Kavosh [19] improve the efficiency of enumerating all subgraphs. G-Tries [35] is based on the idea of creating dedicated representations of sub graphs and ACC [28] uses combinatorial techniques to speed-up the computation. Furthermore, parallelized approaches [37,44] have been developed as well as approximations [14,42].

While the analysis of static networks is important, time-dependent or dynamic networks have recently gained a lot of attention. Analyzing dynamics of biological processes and systems is important for synthetic as well as for computational biology [2]. For example, the analysis of enzyme dynamics helps to understand how it works, and thus, reveals opportunities for improving its functionality. Analyzing the dynamics of amino acids to identify spatial arrangements that correspond to active sites or other functionally relevant features is important for protein classification and structure prediction [6,22]. Due to spatial amino acid arrangements, some motifs occur only in stable structure elements like α-helices and β-sheets and some represent general interactions. Moreover, counting such motifs in dynamic graphs or motifs in any kind of biological network seems to be a promising approach to gain insight into various biological systems [3,33,41].

A common way to analyze the protein dynamics is the solution of Newtons equation of motion, i.e., molecular dynamics (*MD*). This method is used to quantify motions, mechanics, and spatial motifs within a single protein-structure as well as different molecular interactions. The MD approach approximates the time dependent behavior of a protein in its natural environment and results in a trajectory of atoms. One efficient way to analyze this trajectory is the use of graph-theoretic measures. Moreover, using dynamic graph measures like motif frequencies opens new opportunities for MD analysis. To this end, the trajectory has to be transformed into an amino acid contact map defined by distance cutoffs. Afterwards one can apply methods from graph theory to analyze the networks of transient contacts and identify flexible or rigid regions as well as important motifs. So far, only rough metrics like root mean square deviation (*RMSD*) of heavy atoms are utilized to capture protein dynamics. While the maximum number of contacts of an amino acid is approximately 6 [32], functional motifs are commonly considered on smaller sizes. For simplicity, we focus on 4-vertex motifs in this work.

The dynamics of time-dependent graphs are commonly modeled as a stream of updates. Each update describes a change to the graph as an atomic operation [34]. Stream-based algorithms use these updates to continuously update graph-theoretic properties of interest. While such an algorithm for counting triangles in dynamic, undirected graphs has been developed [11], there is no approach for counting undirected 4-vertex motifs in dynamic graphs. Using snapshot-based algorithms like Fanmod, Kavosh, G-Tries, or ACC is expensive, especially when performing an analysis with a high granularity. In this work, we close this gap by developing a stream-based algorithm for updating the motif count in dynamic graphs.

The remainder of this paper is structured as follows: In Sect. 2, we introduce our terminology and define the problem of counting motifs in a dynamic graph. We introduce *StreaM*, a stream-based algorithm for counting undirected 4-vertex motifs in dynamic graphs, in Sect. 3. We present an evaluation of our algorithm against existing approaches in Sect. 4 as well as an application scenario were we show a completely new approach of using the analytical power of *StreaM* to capture essential protein dynamics in a large MD trajectory. Finally, we summarize and conclude our work in Sect. 5.

2 Background and Terminology

In this Section, we introduce our terminology for graphs, dynamic graphs, and motifs. Then, we define the problem of counting motifs in dynamic graphs and discuss the general proceedings of analyzing dynamic graphs.

Graphs. An *undirected, unweighted graph* $G = (V, E)$ is described by a vertex set $V = \{v_1, v_2, \dots\}$ and an edge set $E \subseteq \{\{v, w\} : v, w \in V \wedge v \neq w\}$. We define the neighborhood of v as $n(v) := \{w : \{v, w\} \in E\}$ and its degree as $d(v) := |n(v)|$. Then, the maximum degree of a graph is defined by $d_{max} := \max_{v \in V} d(v)$.

Dynamic Graphs. As a *dynamic graph*, we consider a graph whose vertex and edge sets change over time. Each change of such a dynamic graph is then represented by an update of V or E that adds or removes an element. Hence, there are four different updates to a dynamic graph:

1. adding a new vertex $(\mathrm{a}dd(v), v \notin V)$,
2. adding a new edge $(\mathrm{a}dd(e), e \notin E)$,
3. removing an existing vertex $(\mathrm{r}m(v), v \in V)$, and
4. removing an existing edge $(\mathrm{r}m(e), e \in E)$.

Then, a dynamic graph is represented by its initial state $G_0 = (V_0, E_0)$ and an ordered list or stream of updates $u_1, u_2, u_3, \dots$. Their consecutive application transforms the graph over time:

$$G_0 \xrightarrow{u_1} G_1 \xrightarrow{u_2} G_2 \xrightarrow{u_3} G_3 \dots$$

Each state G_i can be seen as a separate snapshot at the respective point in time. We refer to a consecutive list of updates as a batch $B_{i,j} := \{u_{i+1}, \dots, u_j\}$ whose application transforms the dynamic graph G_i into G_j, i.e.,

$$G_i \xrightarrow{B_{i,j}} G_j$$

Motifs. In this work, we consider *undirected 4-vertex motifs* (cf. Fig. 1). They represent the 6 classes of isomorph, connected subgraphs of size 4. We denote them as $\mathcal{M} = \{m_1, m_2, \dots, m_6\}$.

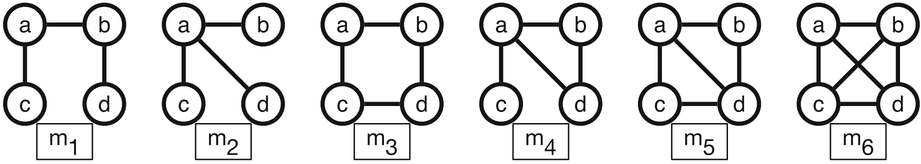


Fig. 1. The 6 undirected 4-vertex motifs $\mathcal{M} = \{m_1, m_2, m_3, m_4, m_5, m_6\}$

Problem Definition. Counting the motifs in a given graph G means to determine the number of occurrences $\mathcal{F}(m_i)$ of each motif $m_i \in \mathcal{M}$. Assume a dynamic graph described by its initial state G_0 and a list of updates $U = (u_1, u_2, \ldots, u_{|U|})$. Further assume a subset $S = (s_0, s_1, \ldots, s_t), 0 \leq s_0, s_i < s_{i+1}, s_t \leq |U|$ of its states which determines the granularity of the analysis. Then, the problem is to generate the motif count $\mathcal{F}_s$ for each state $s \in S$ of interest. Hence, the result of counting motifs in a dynamic graph is a list of motif frequencies $\mathcal{F}_{s_0}, \mathcal{F}_{s_1}, \ldots, \mathcal{F}_{s_t}$ which describes how they change over time.

Analysis of Dynamic Graphs. The properties of dynamic graphs can be analyzed at different granularities. At the highest granularity, properties like degree distribution, shortest-path lengths, or motif count are computed for each change, i.e., each possible state $G_0, G_1, G_2, \ldots$ is analyzed. Lowering this granularity means to only consider a subset of these states, e.g., every 10th state $G_0, G_{10}, G_{20}, \ldots$ or an arbitrary subset $G_0, G_{s_1}, G_{s_2}, \ldots, s_i < s_{i+1}$.

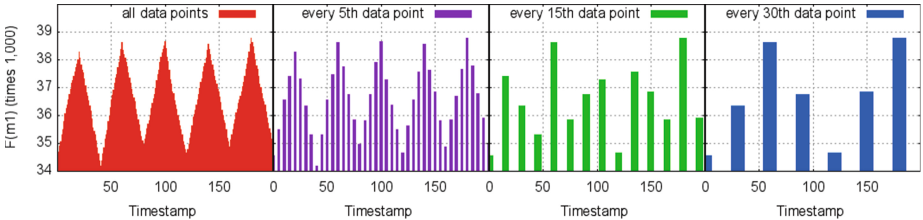


Fig. 2. $\mathcal{F}(m_1)$ in a random, dynamic graph analyzed with different granularities

As an example, take a random, dynamic graph $G_0, \ldots, G_{200}$ with 100 vertices and 500 edges where 20 random edge additions are always followed by 20 random edge removals. Figure 2 shows $\mathcal{F}(m_1)$, the number of occurrences of m_1 over time which fluctuates between 34,000 and 39,000. Performing an analysis at highest granularity shows the impact of each update, i.e., 201 data points. In case the granularity is lowered (only every 5th, 15th, or 30th state is considered), local maxima are missed and the appearance of the development changes. Therefore, it is desirable to determine the properties of a dynamic graph at a high granularity.

Snapshot-based algorithms are executed separately for each snapshot $G_{s \in S}$ to obtain $\mathcal{F}_s$. Hence, the total runtime grows roughly linearly with the number of analyzed snaphots. In contrast, using *stream-based analysis*, the granularity does not influence the runtime. After computing $\mathcal{F}_0$ for the initial graph G_0 using any snapshot-based algorithm, each count $\mathcal{F}_{s_{i+1}}$ is computed by taking G_{s_i}, $\mathcal{F}_{s_i}$, and $B_{s_i,s_{i+1}}$ as input. Since each update is applied separately, an increase in granularity only requires to output the results with a higher frequency. Therefore, stream-based analysis should outperform snapshot-based approaches in case a high granularity is desired and the cost of applying the updates between two states is less than a complete re-evaluation.

3 Counting Motifs in Dynamic Graphs

In this Secion, we describe basic insights regarding motifs in dynamic graphs. Then, we describe *StreaM*, a new stream-based algorithm for counting undirected 4-vertex motifs in dynamic graphs, and discuss its runtime complexity.

Basic Insights. Whenever an edge $e = \{a, b\}$ is added to a graph G_t, i.e., update $u_{t+1} = add(e)$, two things happen: existing motifs are changed and new motifs are formed. First, consider an existing motif m_i that consists of a, b, and 2 other vertices. The addition of e causes the motif to change into a different motif m_j which contains one more edge. We denote this operation as $(i \rightarrow j)$. Its execution decreases the occurrences of m_i and increases the occurrences of m_j, i.e.,

$$(i \rightarrow j) : \mathcal{F}_{t+1}(m_i) := \mathcal{F}_t(m_i) - 1,\ \mathcal{F}_{t+1}(m_j) := \mathcal{F}_t(m_j) + 1$$

Second, consider vertices c and d that do not form a connected component with a and b without e's existence. In case e connects the four vertices, a new motif m_k is formed. We denote this operation as $+(k)$. Its execution increases the occurrences of m_k, i.e.,

$$+(k) : \mathcal{F}_{t+1}(m_k) := \mathcal{F}_t(m_k) + 1$$

In case an existing edge is removed, i.e., $u_{t+1} = rm(e)$, the inverse happens: some motifs are changed and others are dissolved. We denote these operation as $(i \rightarrow j)^{-1}$ and $+(i)^{-1}$.

$$\begin{aligned}(i \rightarrow j)^{-1} :&\ \mathcal{F}_{t+1}(m_i) := \mathcal{F}_t(m_i) + 1,\ \mathcal{F}_{t+1}(m_j) := \mathcal{F}_t(m_j) - 1\\ +(k)^{-1} :&\ \qquad\qquad\qquad\qquad \mathcal{F}_{t+1}(m_k) := \mathcal{F}_t(m_k) - 1\end{aligned}$$

Adding or removing a vertex with degree 0 has no effect on the motif count.

Each motif $m_i \in \mathcal{M}$ contains at least 3 and at most 6 edges. The addition and removal of edges leads to transitions between them (cf. Fig. 3). For example, adding the missing edge to m_5 changes it to m_6 ($(5 \rightarrow 6)$) while removing

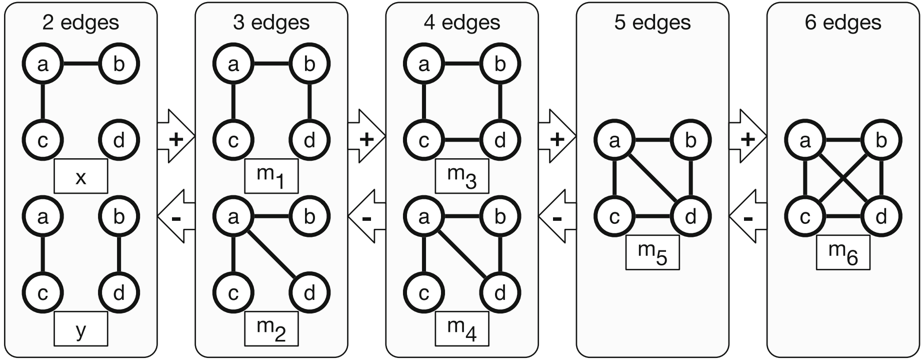


Fig. 3. Transitions between the motifs $m_i \in \mathcal{M}$ when adding and removing edges

any edge from m_6 changes it to m_5 ($(5 \rightarrow 6)^{-1}$). Adding edge $\{b, d\}$ to the disconnected set of nodes x creates a new motif m_1 ($+(1)$) which is dissolved by the removal of any of its 3 edges ($+(1)^{-1}$).

The main idea behind our new stream-based algorithm is to find and apply these operations to correctly update $\mathcal{F}$ for each edge addition and removal.

StreaM. Assume an update (addition or removal) of edge $e = \{a, b\}$. To correctly adapt $\mathcal{F}$, we need to consider all 2-vertex sets $\{c, d\} \in CD(a, b)$ such that a, b, c, and d form a motif if e exists. Either both vertices are connected to a or b directly or d is a neighbor of c which is connected to a or b. With

$$N(a, b) := (n(a) \cup n(b)) \backslash \{a, b\}, \tag{1}$$

we can define $CD(a, b)$ as follows:

$$CD(a, b) = \{\{c, d\} : (c, d \in N(a, b), c \neq d) \vee (c \in N(a, b), d \in n(c) \backslash \{a, b\})\}$$

Besides $\{a, b\}$, 5 edges are possible between a, b, c, and d. We denote their existence as a quintuple $\mathcal{S}(a, b, c, d) = (ac, ad, bc, bd, cd)$, called their *signature*. At least two distinct edges must exist, the first connecting c and the second connecting d. Therefore, there are $2^5 - 2 \cdot 2^2 = 24$ possible signatures.

Table 1. Operation mapping $\mathcal{O}$ from signatures $\mathcal{S}(a, b, c, d)$ to operations

$\mathcal{S}$	10010 01100	10001 01001 00101 00011	11000 00110	11001 00111	10011 01101	11100 11010 10110 01110	10101 01011	11110	11101 11011 10111 01111	11111
$\mathcal{O}$	$+(1)$	$+(1)$	$+(2)$	$+(4)$	$(1 \rightarrow 3)$	$(1 \rightarrow 4)$	$(2 \rightarrow 4)$	$(3 \rightarrow 5)$	$(4 \rightarrow 5)$	$(5 \rightarrow 6)$

Each signature corresponds to a specific operation that must be executed to update $\mathcal{F}$. We define a function $\mathcal{O}$ that maps a signature $\mathcal{S}$ on the corresponding operation. The complete assignment of signatures to operations is given in Table 1. In case the edge $\{a, b\}$ is removed instead of added, the inverse operation must be executed. As an example consider the signature (10010) which is isomorph to (01100). The addition of $\{a, b\}$ creates the motif m_1. Its removal dissolves the motif as a, b, c, and d are no longer connected.

Based on $\mathcal{S}$ and $\mathcal{O}$, we can now describe the stream-based algorithm *StreaM* for updating the motif frequency in an undirected graph (cf. Algorithm 1). For an edge $\{a, b\}$ that is added or removed (described by *type*), we first determine the set $CD(a, b)$ of all pairs of vertices connected to a and b. For each pair $\{c, d\} \in CD(a, b)$, the required operation $o = \mathcal{O}(\mathcal{S}(a, b, c, d))$ is determined from the signature of a, b, c, and d. If $\{a, b\}$ is added, the operation o is executed. Otherwise, the inverse operation o^{-1} is executed.

Data: $G, \{a, b\}, type \in \{add, rm\}$
begin
  for $\{c, d\} \in CD(a, b)$ **do**
    $o = \mathcal{O}(\mathcal{S}(a, b, c, d))$; /* operation */
    if $type = add$ **then**
      execute o ; /* edge is added */
    else if $type = rm$ **then**
      execute o^{-1} ; /* edge is removed */
  end
end

Algorithm 1. *StreaM* for maintaining $\mathcal{F}$ in dynamic graphs

Complexity Discussion. *StreaM* iterates over the $|CD(a, b)| \leq 5 \cdot (d_{max})^2$ elements of $CD(a, b)$. For each element $\{c, d\}$, it computes the signature which can be done in $5 \cdot O(1)$ time, assuming hash-based datastructures are used for adjacency lists. In addition, $\mathcal{F}$ is incremented or decremented which has time complexity of $O(1)$ as well. Therefore, processing a single edge addition or removal with *StreaM* has time complexity of

$$O((d_{max})^2) \cdot (O(1) + O(1)) = O((d_{max})^2)$$

4 Evaluation

In this Section, we evaluate the runtime performance *StreaM*. First, we briefly discuss our evaluation setup. Then, we evaluate the runtime dependence of the algorithm to batch size as well as vertex degree. We compare the runtime of our algorithms to related work in scenarios where the analysis is performed at high granularity and on dynamic graphs obtained from MD simulations consisting of 20,000 snapshots. Finally, we show that *StreaM* is a powerful and unique approach to capture essential molecular dynamics and gain insights on secondary structure focused amino acid interactions.

Evaluation Setup. All measurements are executed on an *HP ProLiant DL585 G7* server with 64 *AMD OpteronTM 6282SE* processors with 2.6 GHz each running a Debian operating system. We implemented *StreaM* in the Java-based framework DNA (Dynamic Network Analyzer) [38] for the analysis of dynamic graphs. The framework including our implementation of *StreaM* is available on the project's GitHub page[1]. We compare our approach with four popular snapshot-based approaches for counting motifs: Fanmod [43], Kavosh [19], G-Tries [35], and ACC [28]. For all approaches, we use the original programs provided by their authors[2,3,4,5]. The cmd-line version of Fanmod as well as G-Tries and Kavosh are implemented in C++. We compiled them from the original sources using GCC version 4.7.2. Like our approach, ACC is implemented in Java and executed using a 64-bit JVM with version 1.7.0_25.

Complexity of StreaM. Now, we validate the runtime complexity of *StreaM* discussed in Sect. 3. We generated undirected random graphs (R) as well as power-law graphs (PL) using the Barabási-Albert model with 500 vertices and 5,000, 10,000, and 15,000 edges. First, we generated 200 batches for each graph with a growing number of random edge exchanges. A random edge exchange is performed by selecting two random edges $e_1 = (a, b)$ and $e_2 = (c, d)$ and exchanging their end points, i.e., transforming the edges to $e'_1 = (a, d)$ and $e'_2 = (c, b)$. This implies 4 updates which are added to the respective batch: $rm(e_1)$, $rm(e_2)$, $add(e'_1)$, and $add(e'_2)$. The ith batch contains i edge exchanges, denoted as $E^x(i)$. Second, we created 200 batches for each graph, each containing 250 random edge addition, denoted as $E^+(250)$. The application of each batch leads to an increase of the average and maximum degree by 1, hence 200 over all. For all graphs and batch types, we recorded the average per-batch runtime of 20 repetitions while performing an analysis using *StreaM*. In the first case, we expect the runtime to grow linearly with the number of updates $|B|$ because $E^x(i)$ does not change d_{max} significantly during the application of each batch B. Furthermore, we expect the runtime of *StreaM* to grow quadratically with the batches $E^+(250)$) since the maximum degree is increased by 1 with each batch.

Figure 4a shows the per-batch runtimes for the analysis of random and power-law graphs for $E^x(i)$. For all graphs, the runtime grows linearly with each batch. The per-batch runtimes of applying $E^+(250)$ to both graph types is shown in Fig. 4b. For all datasets, the runtime appears to grow quadratically with average and maximum degree which are increased by approximately 1 with each batch. As expected, the runtime of *StreaM* increases linearly with the batch size (cf. Figs. 4a. Since the application of $E^x(i)$ does not alter the degree distribution, average and maximum degree stay constant which results in this linear increase of the runtime. Furthermore, the runtime of *StreaM*, in dependence of the

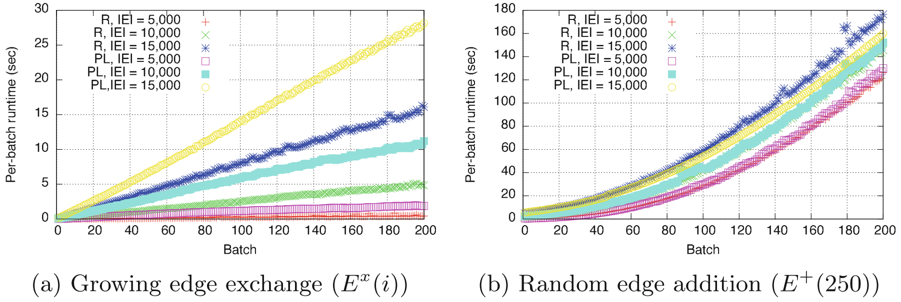
[1] https://github.com/BenjaminSchiller/DNA.

[2] http://theinf1.informatik.uni-jena.de/~wernicke/motifs/.

[3] http://lbb.ut.ac.ir/Download/LBBsoft/Kavosh/.

[4] http://www.dcc.fc.up.pt/gtries/.

[5] http://www.ft.unicamp.br/docentes/meira/accmotifs/.



(a) Growing edge exchange ($E^x(i)$) (b) Random edge addition ($E^+(250)$)

Fig. 4. Per-batch runtime for random (R) and power-law (PL) graphs of size $|V| = 500$ with different edge count $|E|$ depending on batch type

maximum vertex degree d_{max}, is bounded from above by a quadratic function as its algorithmic complexity of $O((d_{max})^2)$ implies. These results validate the complexity discussion. The runtime of *StreaM* grows linearly with the batch size and depends quadratically on the maximum degree. Therefore, *StreaM* can be executed without performance penalties for arbitrary granularity.

Analysis with High Granularity. Next, we show that *StreaM* outperforms snapshot-based approaches for analyses with high granularities. As initial graphs, we consider 12 different datasets that have already been used in the evaluation of ACC [28] and other snapshot-based approaches. The datasets originate from a wide range of areas including biological, social, and traffic networks (cf. Table 2). Their size ranges from 418 to 12,905 vertices with an average degree between 1.85 and 22.01. As dynamics, we created 1,000 batches each consisting of a single random edge exchange $E^x(1)$, i.e., $|B| = 4$. We measure the total time it takes each approach to determine the motif count of the resulting 1,001 states. In some cases, the execution of the snapshot-based approach did not finish after a whole week. We terminated these processes and extrapolated the runtime for the analysis of the 1,001 snapshots from the number of completed ones. The values for G-Tries on foldoc are excluded since it did not even process the first snapshot

Table 2. Properties of datasets used for evaluation with high granularity

	$\|V\|$	$\|E\|$	d_{avg}		$\|V\|$	$\|E\|$	d_{avg}
ecoli	418	519	2.48	odlis	2,900	16,377	11.29
yeast	688	1,078	3.13	epa	4,271	8,909	4.17
roget	1,010	3,648	7.22	pairsfsg	5,018	55,227	22.01
airport	1,574	17,215	21.87	california	6,175	15,969	5.17
csphd	1,882	1,740	1.85	wordsenglish	7,381	44,207	11.98
facebook	1,899	13,838	14.57	foldoc	12,905	83,101	12.88

during this time. Especially for larger graphs, the repeated re-computation of the snapshot-based algorithms should perform much slower than the stream-based application of small batches. Therefore, we expect *StreaM* to outperform the snapshot-based approaches for all graphs.

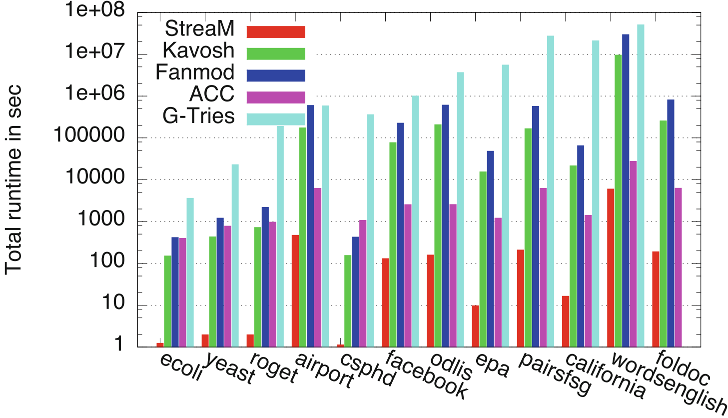


Fig. 5. Total runtime of analysis with high granularity (1,000 times $E^x(1)$)

Figure 5 depicts the resulting runtimes for each dynamic graph. For most datasets and approaches, *StreaM* performs between 10 and 1,000 times faster than the other approaches. The smallest speedup we observed was for the *wordsenglish* dataset, where *StreaM* is still 4.6 times faster than ACC, the best competitor. These results comply with our expectations. Since *StreaM* only computes the motif count of the complete graph once and then only updates the results for 4 updates between two states it performs much faster than all snapshot-based approaches. Therefore, it becomes clear that *StreaM* outperforms snapshot-based algorithms in case an analysis with high granularity is desired.

MD-simulations Case. Motifs are essential for the structural classification of proteins which can be observed from amino acid interactions during MD simulations. In proteins, some motifs, like helices, occur only within structurally stable elements while others occur more frequently. Counting such structural motifs during an MD simulation of a dynamic enzyme is an interesting approach and allows the evaluation of essential molecular dynamics. Therefore, we analyze an MD simulation using *StreaM* as well as snapshot-based algorithms to investigate their performance on such a realistic dynamic graph. In addition, we investigate the general applicability of *StreaM* to gain new insights of capturing molecular dynamics from amino acid interaction motifs.

We created a graph time series from a molecular dynamics simulation of an enzyme, the para Nitro Butyrate Esterase-13 (*pNB-Est13*), a large carboxilic esterase. It is used as an additive of cleansing agents and holds a big potential

towards plastic degradation [29]. *PNB-Est13* monomer consists of 491 residues with molecular weight of 35 to 40 kDa [29]. As a protein structure we used an homology model of *pNB-Est13*. We used *1C7J chain A*, *1QE3 chain B*, and *1C7I chain A* as templates for the homology model. MD simulations were performed with the *Yasara* software suite [24] using the *AMBER03* force-field [10] with constant temperature (313K), pressure (1 bar), and pH (7.4). At the beginning of the simulation, the box is filled with *Tip3* water and *NaCl* counter ions (0.9 %). Afterwards, the protein was energy-minimized utilizing the *AMBER03* force field until convergence was reached (0.01 kJ/mol per atom during 200 steps) [25]. We equilibrated the solvent for 500 ps. Then we simulated for 50 ns and took snapshots every 2.5 ps. From these 20,000 snapshots, we generated a dynamic graph with 491 vertices, each modeling an amino acid C_α. An undirected edge is created between two vertices in case their Euclidean distance is shorter than $d = 7$Å. In addition, we generated dynamic graphs using $d \in [8, 12]$Å to create denser graphs. All 6 dynamic graphs consist of 491 vertices connected by 1,904 to 7,398 edges on average, depending on the distance threshold d (cf. Table 3).

Table 3. Properties of the dynamic graphs generated during MD simulations

	7Å	8Å	9Å	10Å	11Å	12Å
$\|V\|$	491	491	491	491	491	491
$\|E\|$	1,904	2,413	3,248	4,370	5,877	7,398
d_{avg}	7.76	9.83	13.23	17.8	23.94	30.13
$\|B\|$	141.16	176.54	278.92	366.02	422.32	487.70
$\frac{\|B\|}{\|E\|}$	7.42	7.32	8.58	8.38	7.18	6.60

The average batch size ranges between 141.16 and 487.7 implying that 6.6 % to 8.58 % of all connections are changed between two snapshots. To compare the performance of *StreaM* to existing approaches, we analyzed these dynamic graphs consisting of an initial graph and 19,999 batches using our stream-based implementation and measured the total runtime of the execution. For existing approaches, we generated the 20,000 separate snapshots that represent the dynamic graph and analyzed each one separately. We added the execution times of all steps to obtain a single runtime.

The averages of 20 repetitions for all approaches and distance thresholds d are shown in Fig. 6. For the standard distance threshold of 7Å, our stream-based approach takes 173 sec while Kavosh, the fastest competitor, takes 2,365 sec. The analysis using Fanmod and ACC takes around 10,000 sec while G-Tries runs for 400,000 sec. Hence, the speedup of *StreaM* lies between 13.7 and 2,300 times when compared to snapshot-based approaches for the standard case of $d = 7$Å [4]. When increasing the distance threshold d, the runtimes of *StreaM*, Kavosh, and Fanmod increase in a similar way. Interestingly, the runtimes of ACC and G-Tries only increase slightly, indicating that they mainly depend on the number of vertices in a graph and not the number of edges.

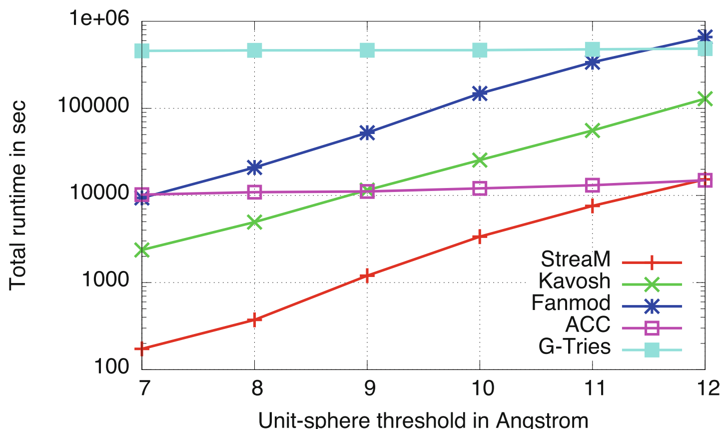
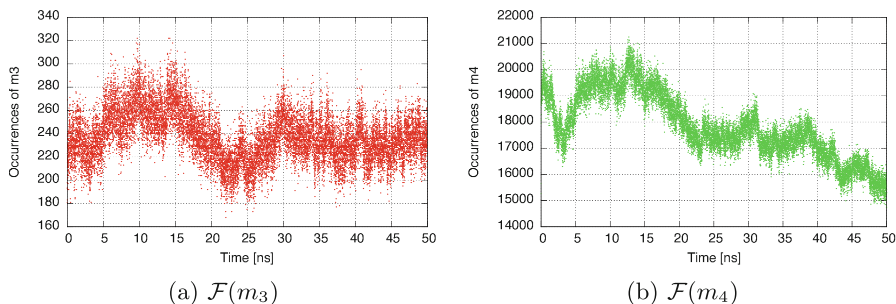


Fig. 6. Total runtime of MD graph analysis depending on distance threshold

As expected and indicated by our complexity discussion and evaluation before, the runtime of *StreaM* increases as the batch size and maximum vertex degree grows. Since the runtime of ACC does not increase as drastically with the distance threshold, the runtimes of both approaches are very close for the highest investigated threshold of 12Å. Notably, a distance threshold of 12Å is not realistic for amino acid contact prediction.

This performance evaluation shows that *StreaM* outperforms snapshot-based algorithms when analyzing realistic dynamic graphs, in our case from MD simulations. Except for unrealistically dense graphs, obtained with a distance threshold of 12Å, *StreaM* performs considerably faster than all other approaches. Hence, it allows a much faster analysis of dynamic biological networks such as the protein graphs obtained from MD simulations. For structural motifs during protein dynamics, a high granularity is very important to count transient interactions. Especially long term MD trajectories require fast and efficient algorithms to analyze transient amino acid interactions. To avoid unstable or unrealistic long term simulations, in case of protein unfolding or using incorrect force field parameter sets, *StreaM* indeed is powerful enough to monitor MD stability online in parallel to the execution of the MD simulation.

Interpretation of Analysis Results. For the quantification of *pNB-Est13*'s dynamic behavior we now investigate two meaningful motifs: The structure of m_4 is typical for stabilizing effects between structure elements or loops. It is capable to describe 3 amino acids which are covalently connected within the backbone and interact with a flexible loop due to electrostatic or hydrophobic interactions. In contrast, m_3, a circle/loop containing 4 edges, can only be found in robust structure elements like α-helices and β-sheet. In case of our MD simulation, we observe that the occurrences of m_4 decreases over time (cf. Fig. 7b). This means that the protein structure enlarges during simulation. In relation to



(a) $\mathcal{F}(m_3)$ (b) $\mathcal{F}(m_4)$

Fig. 7. Analysis results: motif counts for MD simulations over time

the *RMSD*, we observe a Pearson correlation of -0.67 (p-value $< 2.2 \cdot 10^{-16}$, 95 % conf. interval, -0.673 to -0.657). Similar for m_4 we observe a Pearson correlation of *RMSD* to m_3 with a value of -0.190 (p-value $< 2.2 \cdot 10^{-16}$, 95 % conf. interval: -0.204 to -0.177). Clearly, the number of m_3 motifs remain nearly constant over time as shown in Fig. 7a. This behavior agrees with the general assumption that m_3 can only be found in stable structure elements, such as α-helices, which is necessary for a constant *RMSD*. In case of MD graphs, a high granularity is indispensable to capture all transient amino acid interactions. In contrast, snapshot-based approaches do not allow for an analysis at such high granularity and therefore cannot generate similar insights. From these results, we can conclude that *StreaM* is capable of capturing essential molecular dynamics at high granularity - in particular important structural features based on secondary structure focused amino acid interactions. To this end, besides its outstanding performance, we showed that *StreaM* is a powerful new algorithm for the analysis of large MD trajectories.

5 Summary, Conclusion, and Future Work

As dynamic graphs have gained much attention in the recent past, not many approaches exist to efficiently analyze the time-dependent properties of such networks. In this work, we developed *StreaM*, a stream-based algorithm for counting undirected 4-vertex motifs in dynamic graphs. We evaluated the algorithm on generated datasets as well as realistic graphs obtained from MD simulations of *pNB-Est13*. We showed that using motifs for protein dynamic analysis helps to distinguish between structure elements and general interactions and might be a valuable, additional analysis procedure to assess local stability of MD trajectories whereas *RMSD* measures global stability. Our approach outperforms state-of-the-art by up to $2,300$ times on real-world datasets. Thereby, it enables the fine-grained analysis required to understand highly dynamic graphs over time.

Dynamic aspects are typically done on small motifs because the maximal number of contacts of an amino acid is approximately six. In the future, we will extend our work by generalizing the algorithm for arbitrary motif sizes

and developing rule sets for other motif types. Moreover we will dynamically annotate individual amino acids and the respective motifs participate in, during a simulation. This will open new possibilities for bimolecular engineering and in particular enzyme engineering.

References

1. Albert, I., Albert, R.: Conserved network motifs allow protein-protein interaction prediction. Bioinformatics **20**(18), 3346–52 (2004)
2. Alder, B.J., Wainwright, T.E.: Studies in molecular dynamics. J. Chem. Phys. **31**(2), 459–466 (1959)
3. Alon, N., et al.: Biomolecular network motif counting and discovery by color coding. Bioinformatics **24**(13), 241–249 (2008)
4. Atilgan, A.R., et al.: Anisotropy of fluctuation dynamics of proteins with an elastic network model. Biophys. J. **80**(1), 505–515 (2001)
5. Biemann, C., et al.: Quantifying semantics using complex network analysis. In: COLING (2012)
6. Chakraborty, S., Biswas, S.: Approximation algorithms for 3-D common substructure identification in drug and protein molecules. In: Dehne, F., Gupta, A., Sack, J.-R., Tamassia, R. (eds.) WADS 1999. LNCS, vol. 1663, pp. 253–264. Springer, Heidelberg (1999)
7. Chen, J., et al.: Nemofinder: Dissecting genome-wide protein-protein interactions with meso-scale network motifs. In: ACM SIGKDD (2006)
8. Chen, J., et al.: Labeling network motifs in protein interactomes for protein function prediction. In: IEEE ICDE (2007)
9. Colak, R., et al.: Dense graphlet statistics of protein interaction and random networks. In: Pacific Symposium on Biocomputing (2009)
10. Duan, Y., et al.: A point-charge force field for molecular mechanics simulations of proteins based on condensed-phase quantum mechanical calculations. J. Comput. Chem. **24**(16), 1999–2012 (2003)
11. Ediger, D., et al.: Massive streaming data analytics: a case study with clustering coefficients. In: IEEE IPDPSW (2010)
12. Feldman, D., Shavitt, Y.: Automatic large scale generation of internet pop level maps. In: IEEE GLOBECOM (2008)
13. Feldman, D., et al.: A structural approach for pop geo-location. Comput. Netw. **56**, 1029–1040 (2012)
14. Gonen, M., Shavitt, Y.: Approximating the number of network motifs. Internet Math. **6**(3), 349–372 (2009)
15. Hales, D., Arteconi, S.: Motifs in evolving cooperative networks look like protein structure networks. Netw. Heterogen. Media **3**(2), 239–249 (2008)
16. Hutchinson, E.G., Thornton, J.M.: Promotif– program to identify and analyze structural motifs in proteins. Protein Sci. **5**(2), 212–220 (1996)
17. Jurgens, D., Lu, T.: Temporal motifs reveal the dynamics of editor interactions in wikipedia. In: ICWSM (2012)
18. Kalir, S., et al.: Ordering genes in a flagella pathway by analysis of expression kinetics from living bacteria. Science **292**(5524), 2080–2083 (2001)
19. Kashani, Z.R.M., et al.: Kavosh: a new algorithm for finding network motifs. BMC Bioinformatics, 10(1) (2009)
20. Kashtan, N., et al.: Mfinder tool guide. Technical report (2002)

21. Kim, J., et al.: Coupled feedback loops form dynamic motifs of cellular networks. Biophys. J. **94**(2), 359–365 (2008)
22. Kleywegt, D.J.: Recognition of spatial motifs in protein structures. J. Mol. Biol. **285**(4), 1887–1897 (1999)
23. Kovanen, L., et al.: Temporal motifs in time-dependent networks. Journal of Statistical Mechanics: Theory and Experiment (2011)
24. Krieger, E., et al.: Increasing the precision of comparative models with YASARA NOVA-a self-parameterizing force field. Proteins **47**, 393–402 (2002)
25. Krieger, E., et al.: Fast empirical pKa prediction by Ewald summation. Journal of molecular graphics & modelling (2006)
26. Krumov, L., et al.: Leveraging network motifs for the adaptation of structured peer-to-peer-networks. In: IEEE GLOBECOM (2010)
27. Maslov, S., Sneppen, K.: Specificity and stability in topology of protein networks. Science **296**, 910–913 (2002)
28. Meira, L.A.A., et al.: acc-motif detection tool (2012). arXiv:1203.3415
29. Michels, A., et al.: Verwendung von esterasen zur spaltung von kunststoffen (2011)
30. Milenkoviæ, T., Pržulj, N.: Uncovering biological network function via graphlet degree signatures. Cancer Inform. **6**, 257–273 (2008)
31. Milo, R., et al.: Network motifs: simple building blocks of complex networks. Science **298**(5594), 824–827 (2002)
32. Miyazawa, S., Jernigan, R.L.: Residue-residue potentials with a favorable contact pair term and an unfavorable high packing density term, for simulation and threading. J. Mol. Biol. **256**, 623–644 (1996)
33. Panni, S., Rombo, S.E.: Searching for repetitions in biological networks: methods, resources and tools. Briefings Bioinform. **16**(1), 118–136 (2015)
34. Rauch, M., et al.: Computing on data streams. In: DIMACS Workshop External Memory and Visualization (1999)
35. Ribeiro, P., Silva, F.: G-tries: an efficient data structure for discovering network motifs. In: ACM Symposium on Applied Computing (2010)
36. Royer, L., et al.: Unraveling protein networks with power graph analysis. PLoS Comput. Biol. 4(7) (2008)
37. Schatz, M., et al.: Parallel network motif finding. University of Maryland, Technical report (2008)
38. Schiller, B., Strufe, T.: Dynamic network analyzer building a framework for the graph-theoretic analysis of dynamic networks. In: SummerSim (2013)
39. Schreiber, F., Schwöbbermeyer, H.: Mavisto: a tool for the exploration of network motifs. Bioinformatics, **21**(9), 2076–2082 (2005)
40. Shen-Orr, S.S., et al.: Network motifs in the transcriptional regulation network of escherichia coli. Nature Genet **31**, 64–68 (2002)
41. Tran, N., et al.: Counting motifs in the human interactome. Nature Communications (2013)
42. Wernicke, S.: Efficient detection of network motifs. IEEE ACM TCBB **3**(4), 321–322 (2006)
43. Wernicke, S., Rasche, F.: Fanmod: a tool for fast network motif detection. Bioinformatics **22**(9), 1152–1153 (2006)
44. Zhao, Z., et al.: Subgraph enumeration in large social contact networks using parallel color coding and streaming. In: ICPP (2010)

Convolutional LSTM Networks for Subcellular Localization of Proteins

Søren Kaae Sønderby[1(✉)], Casper Kaae Sønderby[1], Henrik Nielsen[2], and Ole Winther[1,3]

[1] Bioinformatics Centre, Department of Biology, University of Copenhagen, Copenhagen, Denmark
{skaaesonderby,casperkaae}@gmail.com
[2] Center for Biological Sequence Analysis, Department of Systems Biology, Technical University of Denmark, 2800 Kongens Lyngby, Denmark
hnielsen@cbs.dtu.dk
[3] Department for Applied Mathematics and Computer Science, Technical University of Denmark, 2800 Kongens Lyngby, Denmark
olwi@dtu.dk

Abstract. Machine learning is widely used to analyze biological sequence data. Non-sequential models such as SVMs or feed-forward neural networks are often used although they have no natural way of handling sequences of varying length. Recurrent neural networks such as the long short term memory (LSTM) model on the other hand are designed to handle sequences. In this study we demonstrate that LSTM networks predict the subcellular location of proteins given only the protein sequence with high accuracy (0.902) outperforming current state of the art algorithms. We further improve the performance by introducing convolutional filters and experiment with an attention mechanism which lets the LSTM focus on specific parts of the protein. Lastly we introduce new visualizations of both the convolutional filters and the attention mechanisms and show how they can be used to extract biologically relevant knowledge from the LSTM networks.

Keywords: Subcellular location · Machine learning · LSTM · RNN · Neural networks · Deep learning · Convolutional networks

1 Introduction

Deep neural networks have gained popularity for a wide range of classification tasks in image recognition and speech tagging [9,20] and recently also within biology for prediction of exon skipping events [30]. Furthermore a surge of interest in recurrent neural networks (RNN) has followed the recent impressive results shown on challenging sequential problems like machine translation and speech recognition [2,14,27]. Within biology, sequence analysis is a very common task

S.K. Sønderby and C.K. Sønderby—These authors contributed equally.

© Springer International Publishing Switzerland 2015
A.-H. Dediu et al. (Eds.): AlCoB 2015, LNBI 9199, pp. 68–80, 2015.
DOI: 10.1007/978-3-319-21233-3_6

used for prediction of features in protein or nucleic acid sequences. Current methods generally rely on neural networks and support vector machines (SVM), which have no natural way of handling sequences of varying length. Furthermore these systems rely on highly hand-engineered input features requiring a high degree of domain knowledge when designing the algorithms [11,24]. RNNs are a type of neural networks that naturally handles sequential data. In an RNN the input to the network's hidden layers is both the input features at the current timestep and the hidden activation from the previous time step. Hence an RNN corresponds to placing neural networks with shared identical weights at each timestep and letting information flow across the sequence by connecting the networks with (recurrent) weights between the hidden layers. In bioinformatics RNNs have previously been used for contact map prediction [10], and protein secondary structure predition [3,21]. However standard RNNs have been shown to be difficult to train with backpropagation through time due to both vanishing and exploding gradients [5]. To mitigate this problem, Hochreiter et al. [17] introduced the long short term memory (LSTM) that uses a gated memory cell instead of the standard sigmoid or hyperbolic tangent hidden units used in standard RNNs. In the LSTM the value of each memory cell is controlled with input, modulation, forget and output gates, which allow the LSTM network to store analog values for many time steps by controlling access to the memory cell. In practice this architecture have proven easier to train than standard RNN.

In this paper LSTMs are used to analyze biological sequences and predict to which subcellular compartment a protein belongs. This prediction task, known as protein sorting or subcellular localization, has attracted large interest in the bioinformatics field [11]. We show that an LSTM network, using only the protein sequence information, has significantly better performance than current state of the art SVMs and furthermore have nearly as good performance as large hand engineered systems relying on extensive metadata such as GO terms and evolutionary phylogeny, see Fig. 1 [6,7,18]. These results show that LSTM networks are efficient algorithms that can be trained even on relatively small datasets of around 6000 protein sequences. Secondly we investigate how an LSTM network recognizes the sequence. In image recognition, convolutional neural networks (CNN) have shown state of the art performance in several different tasks [8,20]. Here the lower layers of a CNN can often be interpreted as feature detectors recognizing simple geometric entities, see Fig. 2. We develop a simple visualization technique for convolutional filters trained on either DNA or amino acid sequences and show that in the biological setting filters can be interpreted as motif detectors, as visualized in Fig. 2. Thirdly, inspired by the work of Bahdanau et al. [2], we augment the LSTM network with an attention mechanism that learns to assign importance to specific parts of the protein sequence. Using the attention mechanism we can visualize where the LSTM assigns importance, and we show that the network focuses on regions that are biologically plausible. Lastly we show that the LSTM network learns a fixed length representation of amino acids sequences that, when visualized, separates the sequences into clusters with biological meaning. The contributions of this paper are:

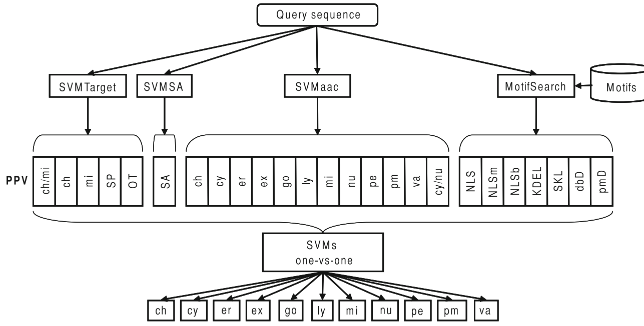


Fig. 1. Schematic showing how MultiLoc combines predictions from several sources to make predictions whereas the LSTM networks only rely on the sequence [18].

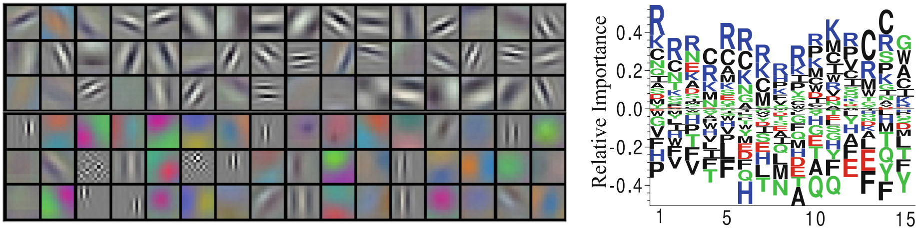


Fig. 2. *Left:* first layer convolutional filters learned in [20], note that many filters are edge detectors or color detectors. *Right:* example of learned filter on amino acid sequence data, note that this filter is sensitive to positively charged amino acids (Color figure online).

1. We show that LSTM networks combined with convolutions are efficient for predicting subcellular localization of proteins from sequence.
2. We show that convolutional filters can be used for amino acid sequence analysis and introduce a visualization technique.
3. We investigate an attention mechanism that lets us visualize where the LSTM network focuses.
4. We show that the LSTM network effectively extracts a fixed length representation of variable length proteins.

2 Materials and Methods

This section introduces the LSTM cell and then explains how a regular LSTM (R-LSTM) can produce a single output. We then introduce the LSTM with attention mechanism (A-LSTM), and describe how the attention mechanism is implemented.

2.1 LSTM NETWORKS

The memory cell of the LSTM networks is implemented as described in [15] except for peepholes, because recent papers have shown good performance without [27,32,33]. Figure 3 shows the LSTM cell. Equations (1-10) state the forward recursions for a single LSTM layer.

$$i_t = \sigma(D(x_t)W_{xi} + h_{t-1}W_{hi} + b_i) \tag{1}$$

$$f_t = \sigma(D(x_t)W_{xf} + h_{t-1}W_{hf} + b_f) \tag{2}$$

$$g_t = \tanh(D(x_t W_{xg}) + h_{t-1}W_{hg} + b_g) \tag{3}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t \tag{4}$$

$$o_t = \sigma(D(x_t)W_{xo} + h_{t-1}W_{ho} + b_o) \tag{5}$$

$$h_t = o_t \odot \tanh(c_t) \tag{6}$$

$$\sigma(z) = \frac{1}{1 + \exp(-z)} \tag{7}$$

$$\odot : \text{Elementwise multiplication} \tag{8}$$

$$D : \text{Dropout, set values to zero with probability } p \tag{9}$$

$$x_t : \text{input from the previous layer } h_t^{l-1}, \tag{10}$$

where all quantities are given as row-vectors and activation and dropout functions are applied element-wise. Note that for the first hidden layer h_t^1 the input x_t are the amino acid features. In the memory cell i_t, f_t and o_t can are gating functions controlling input, storage and output of the value c_t stored in each cell. Due to the logistic squashing function used for each gate, the value are always in the interval (0,1) and information can flow through the gate if the value is close to one. If dropout is used it is only applied to non-recurrent connections in the LSTM cell [31]. In a multilayer LSTM h_t is passed upwards to the next layer.

2.2 Regular LSTM Networks for Predicting Single Targets

When used for predicting a single target for each input sequence, one approach is to output the predicted target from the LSTM network at the last sequence position as shown in Fig. 5A. A problem with this approach is that the gradient has to flow from the last position to all previous positions and that the LSTM network has to store information about all previously seen data in the last hidden state. Furthermore a regular bidirectional LSTM (BLSTM, 5B)[26] is not useful in this setting because the backward LSTM will only have seen a single position, x_T, when the prediction has to be made. We instead combine two unidirectional LSTMs, as shown in Fig. 5C, where the backward LSTM has the input reversed. The output from the two LSTMs are combined before predictions.

2.3 Attention Mechanism LSTM Netowork

Bahdanau et al. [2] have introduced an attention mechanism for combining hidden state information from a encoder-decoder RNN approach to machine translation. The novelty in their approach is that they use an alignment function that

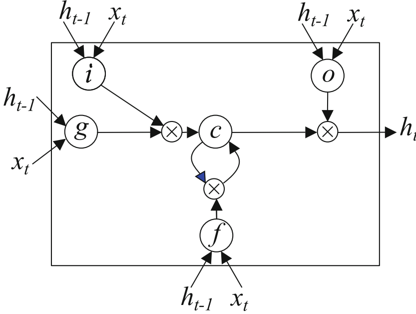


Fig. 3. LSTM memory cell. i: input gate, f: forget gate, o: output gate, g: input modulation gate, c: memory cell. The Blue arrow heads refers to c_{t-1}. The notation corresponds to Eqs. 1 to 10 such that W_{xo} denotes wights for x to output gate and W_{hf} denotes weights for h_{t-1} to forget gates etc. Adapted from [33].

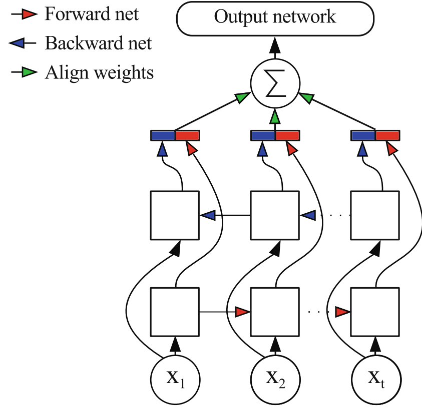


Fig. 4. A-LSTM network. Each state of the hidden units, h_t are weighted and summed before the output network calculates the predictions (Color figure online).

for each output word finds important input words, thus aligning and translating at the same time. We modify this alignment procedure such that only a single target is produced for each sequence. The developed attention mechanism can be seen as assigning importance to each position in the sequence with respect to the prediction task. We use a BLSTM to produce a hidden state at each position and then use an attention function to assign importance to each hidden state as illustrated in Fig. 4. The weighted sum of hidden states is used as a single representation of the entire sequence. This modification allows the BLSTM model to naturally handle tasks involving prediction of a single target per sequence. Conceptually this corresponds to adding weighted skip connections (green arrow heads Fig. 4) between any h_t and the output network, with the weight on each skip connection being determined by the attention function. Each hidden state h_t, $t = 1, \ldots, T$ is used as input to a Feed Forward Neural Network (FFN) attention function:

$$a_t = \tanh(h_t W_a) v_a^{\mathrm{T}} \ , \tag{11}$$

where W_a is an attention hidden weight matrix and v_a is an attention output vector. From the attention function we form softmax weights:

$$\alpha_t = \frac{\exp(a_t)}{\Sigma_{t'=1}^{T} \exp(a_{t'})} \tag{12}$$

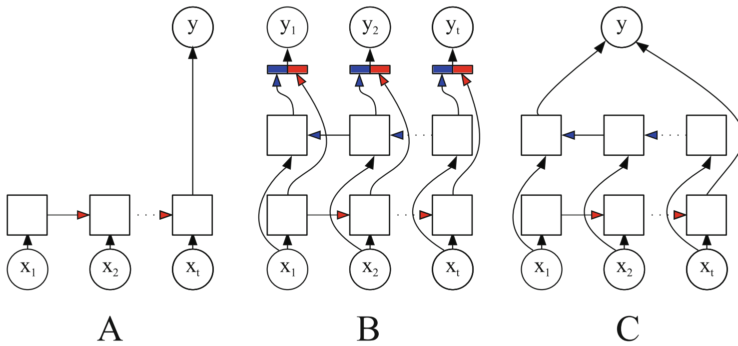


Fig. 5. *A:* Unidirectional LSTM predicting a single target. *B:* Unrolled single layer BLSTM. The forwards LSTM (red arrows) starts at time 1 and the backwards LSTM (blue arrows) starts at time T, then they go forwards and backwards respectively. The errors from the forward and backward nets are combined and a prediction is made for each sequence position. Adapted from [13]. *C:* Unidirectional LSTM for predicting a single target. Squares are LSTM layers (Color figure online).

that are used to produce a context vector c as a convex combination of T hidden states:

$$c = \Sigma_{t=1}^{T} h_t \alpha_t. \tag{13}$$

The context vector is then used is as input to the classification FFN $f(c)$. We define f as a single layer FFN with softmax outputs.

2.4 Subcellular Localization Data

The model was trained and evaluated on the dataset used to train the MultiLoc algorithm published by Höglund et al. [18][1]. The dataset contains 5959 proteins annotated to one of 11 different subcellular locations. To reduce computational time the protein sequences were truncated to a maximum length of 1000. We truncated by removing from the middle of the protein as both the N- and C-terminal regions are known to contain sorting signals [11]. Each amino acid was encoded using 1-of-K encoding, the BLOSUM80 [16] and HSDM [25] substitution matrices and sequence profiles, yielding 80 features per amino acid. Sequence profiles where created with PROFILpro [22][2] using 3 blastpgp [1][3] iterations on UNIREF50.

2.5 Visualizations

Convolutional filters for images can be visualized by plotting the convolutional weights as pixel intensities as shown in Fig. 2. However a similar approach does

[1] http://abi.inf.uni-tuebingen.de/Services/MultiLoc/multiloc_dataset.
[2] http://download.igb.uci.edu/.
[3] http://nebc.nox.ac.uk/bioinformatics/docs/blastpgp.html.

not make sense for amino acid inputs due to the 1-of-K vector encoding. Instead we view the 1D convolutions as a position specific scoring matrix (PSSM). The convolutional weights can be reshaped into a matrix of l_{filter}-by-l_{enc}, where the amino acid encoding length is is 20. Because the filters show relative importance we rescale all filters such that the height of the highest column is 1. Each filter can then be visualized as a PSSM logo, where the height of each column can be interpreted as position importance and the height of each letter is amino acid importance. We use Seq2Logo with the PSSM-logo setting to create the convolution filter logos [28].

We visualize the importance the A-LSTM network assigns to each position in the input by plotting α from Eq. 12. Lastly we extract and plot the hidden representation from the LSTM networks. For the A-LSTM network we use c from Eq. 13 and for the R-LSTM we use the last hidden state, h_t. Both c and h_t can be seen as fixed length representation of the amino acid sequences. We plot the representation using t-SNE [29].

2.6 Experimental Setup

All models were implemented in Theano [4] using a modified version of the Lasagne library[4] and trained with gradient descent. The learning rate was controlled with ADAM ($\alpha = 0.0002$, $\beta_1 = 0.1$, $\beta_2 = 0.001$, $\epsilon = 10^{-8}$ and $\lambda = 10^{-8}$) [19]. Initial weights were sampled uniformly from the interval $[-0.05, 0.05]$. The network architecture is a 1D convolutional layer followed by an LSTM layer, a fully connected layer and a final softmax layer. All layers use 50 % dropout. The 1D convolutional layer uses convolutions of sizes 1, 3, 5, 9, 15 and 21 with 10 filters of each size. Dense and convolutional layers use ReLU activation [23] and the LSTM layer uses hyperbolic tangent. For the A-LSTM model the size of the first dimension of W_a was 400. We used 4/5 of the data for training and the last 1/5 of the data for testing. The hyperparameters were optimized using 5-fold cross validation on the training data. The cross validation experiments showed that the model converged after 100 epochs. Using the established hyperparameters the models were retrained on the complete training data and the test performance were reported after epoch 100.

3 Results

Table 1 shows accuracy for the R-LSTM and A-LSTM models and several other models trained on the same dataset. Comparing the performance of the R-LSTM, A-LSTM and MultiLoc models, utilizing only the sequence information, the R-LSTM model (0.879 Acc.) performs better than the A-LSTM model (0.854 Acc.) whereas the MultiLoc model (0.767 Acc.) performs significantly worse. Furthermore the 10-ensemble R-LSTM model further increases the performance to 0.902 Acc. Comparing this performance with the other models, combining the

[4] https://github.com/skaae/nntools.

sequence predictions from the MultiLoc model with large amounts of metadata for the final predictions, only the Sherloc2 model (0.930 Acc.) performs better than the R-LSTM ensemble. Figure 6 shows a plot of the attention matrix from the A-LSTM model. Figure 8 shows examples of the learned convolutional filters. Figure 7 shows the hidden state of the R-LSTM and the A-LSTM model.

Table 1. Comparison of results for LSTM models and MultiLoc1/2. MultiLoc1/2 accuracies are reprinted from [12] and the SherLoc accuracy from [7]. Metadata refers to additional protein information such as GO-terms and phylogeny.

Model	Accuracy
Input: Protein Sequence	
R-LSTM	0.879
A-LSTM	0.854
R-LSTM ensemble	**0.902**
MultiLoc	0.767
Input: Protein Sequence + Metadata	
MultiLoc + PhyloLoc	0.842
MultiLoc + PhyloLoc + GOLoc	0.871
MultiLoc2	0.887
SherLoc2	**0.930**

4 Discussion and Conclusion

In this paper we have introduced LSTM networks with convolutions for prediction of subcellular localization. Table 1 shows that the LSTM networks perform much better than other methods that only rely on information from the sequence (LSTM ensemble 0.902 vs. MultiLoc 0.767). This difference is all the more remarkable given the simplicity of our method, only utilizing the sequences and their localization labels, while MultiLoc incorporates specific domain knowledge such as known motifs and signal anchors. One explanation for the performance difference is that the LSTM networks are able to look at both global and local sequence features whereas the SVM based models do not model global dependencies. The LSTM networks have nearly as good performance as methods that use information obtained from other sources than the sequence (LSTM ensemble 0.902 vs. SherLoc2 0.930). Incorporating these informations into the LSTM models could further improve the performance of these models. However, it is our opinion that using sequence alone yields the biologically most relevant prediction, while the incorporation of, e.g., GO terms limits the usability of the prediction requiring similar proteins to be already annotated to some degree. Furthermore, as we show below, a sequence-based method potentially allows for a de novo identification of sequence features essential for biological function.

Table 2. Confusion matrix with true labels shown by row and R-LSTM model predictions by column. E.g. the cell at row 4 column 3 means that the actual class was Cytoplasmic but the R-LSTM model predicted Chloroplast.

Confusion Matrix											
ER	26	1	0	0	8	1	0	0	0	3	0
Golgi	1	28	0	0	0	0	0	0	0	1	0
Chloroplast	0	0	82	3	0	0	5	0	0	0	0
Cytoplasmic	0	0	1	266	0	0	3	12	0	0	0
Extracellular	0	0	0	1	166	0	0	0	0	1	0
Lysosomal	0	0	0	0	5	12	0	0	0	3	0
Mitochondrial	0	0	2	5	0	0	94	1	0	0	0
Nuclear	0	0	0	27	1	0	3	137	0	0	0
Peroxisomal	0	1	0	10	0	0	0	1	18	2	0
Plasma membrane	0	0	0	0	5	0	1	1	0	241	0
Vacuolar	0	0	0	0	7	0	0	0	0	1	5

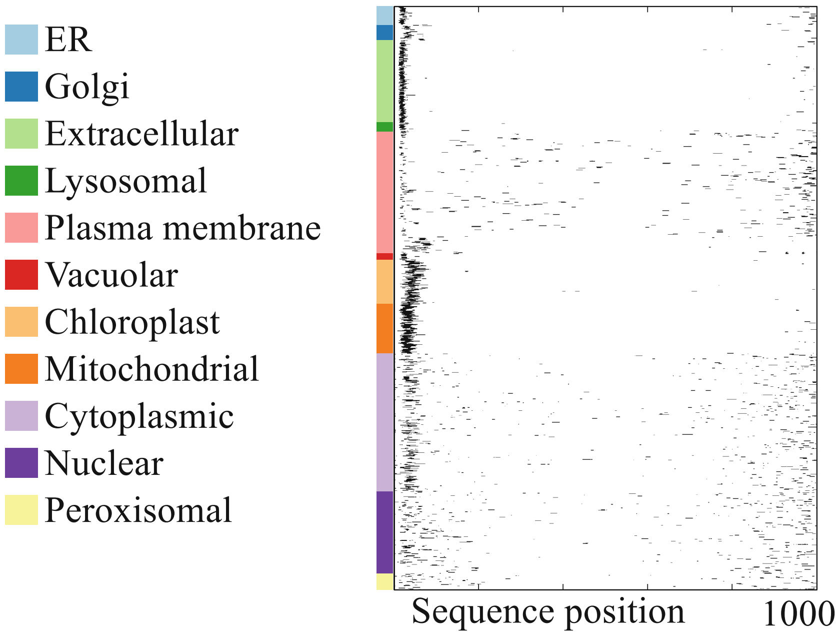


Fig. 6. Importance weights assigned to different regions of the proteins when making predictions. y-axis is true group and x-axis is the sequence positions. All proteins shorter than 1000 are zero padded from the middle such that the N and C terminals align.

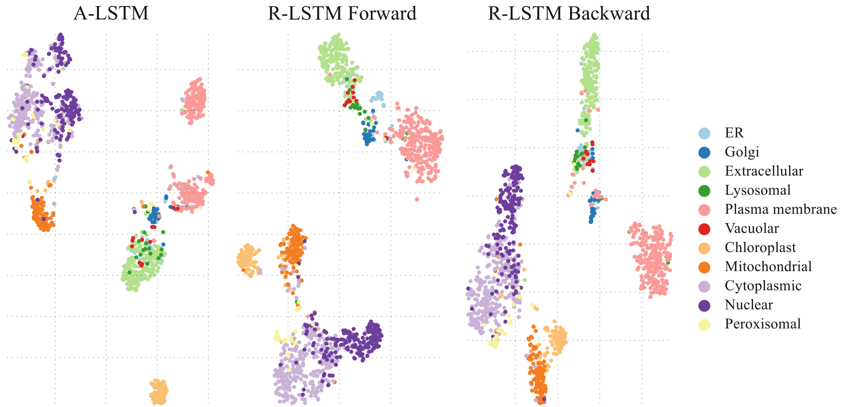


Fig. 7. t-SNE plot of hidden representation for forward and backward R-LSTM and A-LSTM.

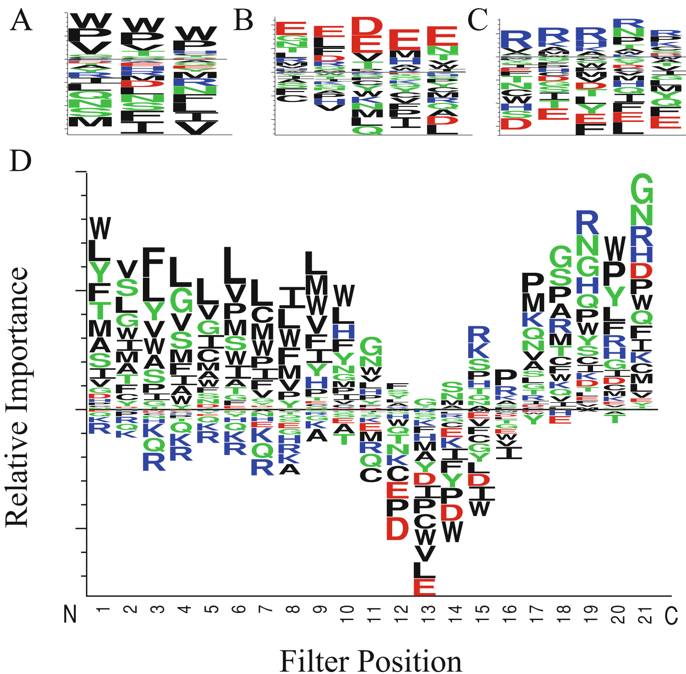


Fig. 8. Examples of learned filters. Filter A captures proline or trypthopan stretches, (B) and (C) are sensitive to positively and negatively charged regions, respectively. Note that for C, negative amino acids seems to suppress the output. Lastly we show a long filter which captures larger sequence motifs in the proteins.

Fig. 6 shows where in the sequence the A-LSTM network assigns importance. Sequences from the compartments ER, extracellular, lysosomal, and vacuolar all belong to the secretory pathway and contain N-terminal signal peptides, which are clearly seen as bars close to the left edge of the plot. Some of the ER proteins additionally have bars close to the right edge of the plot, presumably representing KDEL-type retention signals. Golgi proteins are special in this context, since they are type II transmembrane proteins with signal anchors, slightly further from the N-terminus than signal peptides [18]. Chloroplast and mitochondrial proteins also have N-terminal sorting signals, and it is apparent from the plot that chloroplast transit peptides are longer than mitochondrial transit peptides, which in turn are longer than signal peptides [11]. For the plasma membrane category we see that some proteins have signal peptides, while the model generally focuses on signals, presumably transmembrane helices, scattered across the rest of the sequence with some overabundance close to the C-terminus. Some of the attention focused near the C-terminus could also represent signals for glycosylphosphatidylinositol (GPI) anchors [11]. Cytoplasmic and nuclear proteins do not have N-terminal sorting signals, and we see that the attention is scattered over a broader region of the sequences. However, especially for the cytoplasmic proteins there is some attention focused close to the N-terminus, presumably in order to check for the absence of signal peptides. Finally, peroxisomal proteins are known to have either N-terminal or C-terminal sorting signals (PTS1 and PTS2) [11], but these do not seem to have been picked up by the attention mechanism.

In Fig. 8 we investigate what the convolutional filters in the model focus on. Notably the short filters focus on amino acids with specific characteristics, such as positively or negatively charged, whereas the longer filters seem to focus on distributions of amino acids across longer sequences. The arginine-rich motif in Fig. 7C could represent part of a nuclear localization signal (NLS), while the longer motif in Fig. 7D could represent the transition from transmembrane helix (hydrophobic) to cytoplasmic loop (in accordance with the "positive-inside" rule). We believe that the learned filters can be used to discover new sequence motifs for a large range of protein and genomic features.

In Fig. 7 we investigate whether the LSTM models are able to extract fixed length representations of variable length proteins. Using t-SNE we plot the LSTMs hidden representation of the sequences. It is apparent that proteins from the same compartment generally group together, while the cytoplasmic and nuclear categories tend to overlap. The corresponds with the fact that these two categories are relatively often confused, see Table 2. The categories form clusters which make biological sense; all the proteins with signal peptides (ER, extracellular, lysosomal, and vacuolar) lie close to each other in t-SNE space in all three plots, while the proteins with other N-terminal sorting signals (chloroplasts and mitochondria) are close in the R-LSTM plots (but not in the A-LSTM plot). Note that the lysosomal and vacuolar categories are very close to each other in the plots, this corresponds with the fact that these two compartments are considered homologous [18].

In summary we have introduced LSTM networks with convolutions for subcellular localization. By visualizing the learned filters we have shown that these can be interpreted as motif detectors, and lastly we have shown that the LSTM network can represent protein sequences as a fixed length vector in a representation that is biologically interpretable.

References

1. Altschul, S.F., Madden, T.L., Schäffer, A.A., Zhang, J., Zhang, Z., Miller, W., Lipman, D.J.: Gapped blast and psi-blast: a new generation of protein database search programs. Nucleic acids Res. **25**(17), 3389–3402 (1997)
2. Bahdanau, D., Cho, K., Bengio, Y.: Neural Machine Translation by Jointly Learning to Align and Translate. arXiv preprint arXiv:1409.0473 (Sep 2014)
3. Baldi, P., Brunak, S., Frasconi, P.: Exploiting the past and the future in protein secondary structure prediction. Bioinformatics **15**(11), 937–946 (1999)
4. Bastien, F., Lamblin, P., Pascanu, R., Bergstra, J., Goodfellow, I., Bergeron, A., Bouchard, N., Warde-Farley, D., Bengio, Y.: Theano: new features and speed improvements, November 2012. arXiv preprint arXiv:1211.5590
5. Bengio, Y., Simard, P., Frasconi, P.: Learning long-term dependencies with gradient descent is difficult. IEEE Trans. Neural Netw. **5**(2), 157–166 (1994)
6. Blum, T., Briesemeister, S., Kohlbacher, O.: MultiLoc2: integrating phylogeny and Gene Ontology terms improves subcellular protein localization prediction. BMC bioinform. **10**, 274 (2009)
7. Briesemeister, S., Blum, T., Brady, S., Lam, Y., Kohlbacher, O., Shatkay, H.: SherLoc2: a high-accuracy hybrid method for predicting subcellular localization of proteins. J. Proteome Res. **8**(11), 5363–5366 (2009)
8. Cunn, Y.L., Boser, B., Denker, J., Henderson, D., Howard, R., Hubbard, W., Jackel, L.: Handwritten digit recognition with a back-propagation network. In: Lippmann, R., Moody, J., Touretzky, D. (eds.) Advances in neural information processing systems. pp. 396–404 (1990)
9. Dahl, G., Yu, D., Deng, L., Acero, A.: Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. IEEE Trans. Audio Speech Lang. Process. **20**(1), 30–42 (2012)
10. Di Lena, P., Nagata, K., Baldi, P.: Deep architectures for protein contact map prediction. Bioinformatics **28**(19), 2449–2457 (2012)
11. Emanuelsson, O., Brunak, S., von Heijne, G., Nielsen, H.: Locating proteins in the cell using TargetP, SignalP and related tools. Nat. Protoc. **2**(4), 953–971 (2007)
12. Goldberg, T., Hamp, T., Rost, B.: LocTree2 predicts localization for all domains of life. Bioinformatics **28**(18), i458–i465 (2012)
13. Graves, A.: Supervised sequence labelling with recurrent neural networks. Springer, Heidelberg (2012)
14. Graves, A., Jaitly, N.: Towards end-to-end speech recognition with recurrent neural networks. In: Proceedings of the 31st International Conference on Machine Learning (ICML-14), pp. 1764–1772 (2014)
15. Graves, A.: Generating sequences with recurrent neural networks, (2013). arXiv preprint arXiv:1308.0850
16. Henikoff, S., Henikoff, J.G.: Amino acid substitution matrices from protein blocks. Proc. Natl. Acad. Sci. USA **89**, 10915–10919 (1992)

17. Hochreiter, S., Schmidhuber, J., Elvezia, C.: Long short-term memory. Neural Comput. **9**(8), 1735–1780 (1997)
18. Höglund, A., Dönnes, P., Blum, T., Adolph, H.W., Kohlbacher, O.: MultiLoc: prediction of protein subcellular localization using N-terminal targeting sequences, sequence motifs and amino acid composition. Bioinformatics **22**(10), 1158–1165 (2006)
19. Kingma, D., Ba, J.: Adam: a method for stochastic optimization, December 2014. arXiv preprint arXiv:1412.6980
20. Krizhevsky, A., Sutskever, I., Hinton, G.: Imagenet classification with deep convolutional neural networks. In: Pereira, F., Burges, C.J.C., Bottou, L., Weinberger, K. (eds.) Advances in neural information processing systems, pp. 1097–1105 (2012)
21. Magnan, C., Baldi, P.: SSpro/ACCpro 5: almost perfect prediction of protein secondary structure and relative solvent accessibility using profiles, machine learning, and structural similarity. Bioinformatics **30**(18), 1–6 (2014)
22. Magrane, M. et al.: UniProt Consortium: Uniprot knowledgebase: a hub of integrated protein data. Database 2011, bar009 (2011)
23. Nair, V., Hinton, G.E.: Rectified linear units improve restricted boltzmann machines. In: Proceedings of the 27th International Conference on Machine Learning (ICML-10), pp. 807–814 (2010)
24. Petersen, T., Brunak, S., von Heijne, G., Nielsen, H.: SignalP 4.0: discriminating signal peptides from transmembrane regions. Nat. Methods **8**(10), 785–786 (2011)
25. Prlić, A., Domingues, F.S., Sippl, M.J.: Structure-derived substitution matrices for alignment of distantly related sequences. Protein Eng. **13**, 545–550 (2000)
26. Schuster, M., Paliwal, K.: Bidirectional recurrent neural networks. Signal Process. **45**(11), 2673–2681 (1997)
27. Sutskever, I., Vinyals, O., Le, Q.: Sequence to sequence learning with neural networks. In: Advances in Neural Information Processing Systems, pp. 3104–3112 (2014)
28. Thomsen, M.C.F., Nielsen, M.: Seq2Logo: a method for construction and visualization of amino acid binding motifs and sequence profiles including sequence weighting, pseudo counts and two-sided representation of amino acid enrichment and depletion. Nucleic Acids Res. **40**, W281–W287 (2012)
29. Van Der Maaten, L.J.P., Hinton, G.E.: Visualizing high-dimensional data using t-sne. J. Mach. Learn. Res. **9**, 2579–2605 (2008)
30. Xiong, H.Y., Alipanahi, B., Lee, L.J., Bretschneider, H., Merico, D., Yuen, R.K.C., Hua, Y., Gueroussov, S., Najafabadi, H.S., Hughes, T.R., Morris, Q., Barash, Y., Krainer, A.R., Jojic, N., Scherer, S.W., Blencowe, B.J., Frey, B.J.: The human splicing code reveals new insights into the genetic determinants of disease. Science **347**, 1254806 (2014)
31. Zaremba, W., Sutskever, I., Vinyals, O.: Recurrent neural network regularization (2014). arXiv preprint arXiv:1409.2329
32. Zaremba, W., Kurach, K., Fergus, R.: Learning to Discover Efficient Mathematical Identities. In: Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N., Weinberger, K. (eds.) Advances in Neural Information Processing Systems, pp. 1278–1286, June 2014
33. Zaremba, W., Sutskever, I.: Learning to Execute, October 2014. arXiv preprint arXiv:1410.4615

Phylogenetics

Hybrid Genetic Algorithm and Lasso Test Approach for Inferring Well Supported Phylogenetic Trees Based on Subsets of Chloroplastic Core Genes

Bassam AlKindy[1,3](✉), Christophe Guyeux[1], Jean-François Couchot[1], Michel Salomon[1], Christian Parisod[2], and Jacques M. Bahi[1]

[1] FEMTO-ST Institute, UMR 6174 CNRS, DISC Computer Science Department, University of Franche-Comté, Besançon, France
{bassam.al-kindy,christophe.guyeux,jean-francois.couchot, michel.salomon,jacques.bahi}@univ-fcomte.fr

[2] Laboratory of Evolutionary Botany, University of Neuchâtel, Neuchâtel, Switzerland
christian.parisod@unine.ch

[3] Department of Computer Science, University of Mustansiriyah, Baghdad, Iraq

Abstract. The amount of completely sequenced chloroplast genomes increases rapidly every day, leading to the possibility to build large scale phylogenetic trees of plant species. Considering a subset of close plant species defined according to their chloroplasts, the phylogenetic tree that can be inferred by their core genes is not necessarily well supported, due to the possible occurrence of "problematic" genes (*i.e.*, homoplasy, incomplete lineage sorting, horizontal gene transfers, *etc.*) which may blur phylogenetic signal. However, a trustworthy phylogenetic tree can still be obtained if the number of problematic genes is low, the problem being to determine the largest subset of core genes that produces the best supported tree. To discard problematic genes and due to the overwhelming number of possible combinations, we propose an hybrid approach that embeds both genetic algorithms and statistical tests. Given a set of organisms, the result is a pipeline of many stages for the production of well supported phylogenetic trees. The proposal has been applied to different cases of plant families, leading to encouraging results for these families.

Keywords: Chloroplasts · Phylogeny · Genetic algorithms · Lasso test

1 Introduction

The multiplication of complete chloroplast genomes should normally lead to the ability to infer trustworthy phylogenetic trees for plant species. Indeed, the existence of trustworthy coding sequence prediction and annotation software specific

© Springer International Publishing Switzerland 2015
A.-H. Dediu et al. (Eds.): AlCoB 2015, LNBI 9199, pp. 83–96, 2015.
DOI: 10.1007/978-3-319-21233-3_7

to chloroplasts (like DOGMA [16]), together with the good control of sequence alignment and maximum likelihood or Bayesian inference phylogenetic reconstruction techniques, should imply that, given a set of close species, their core genome (the set of genes in common) can be as large and accurately detected as possible to finally obtain a well-supported phylogenetic tree. However, all genes of the core genome are not necessarily constrained in a similar way, some genes having a larger ability to evolve than other ones due to their lower importance. Such minority genes tell their own story instead of the species one, blurring so the phylogenetic information.

To obtain well-supported phylogenetic trees, the deletion of these problematic genes (which may result from homoplasy, stochastic errors, undetected paralogy, incomplete lineage sorting, horizontal gene transfers, or even hybridization) is needed. A solution is to construct the phylogenetic trees that correspond to all the combinations of core genes, and to finally consider the tree that is as supported as possible while considering as many genes as possible. The major drawback is its inhibitory computational cost, since testing all the possible combinations is totally intractable in practice (2^n phylogenetic tree reconstructions with $n \approx 100$ core genes of plants belonging to the same order). Thus we have to remove the problematic genes without exhaustively testing combinations of genes. Therefore, our proposal is to mix various approaches to extract promising subsets of core genes, encompassing systematic deletion of genes, random selection of large subsets, statistical evaluation of gene effects, and genetic algorithms (GAs) [3,4]. These latters are efficient, robust, and adaptive search techniques designed for solving optimization problems, which have the ability to produce semi-optimal solutions [7,10,14].

The contribution of this article can be summarized as follows. We focus on situations where a large number of genes are shared by a set of species so that, in theory, enough data are available to produce a well supported phylogenetic tree. However, a few genes tell a different evolutionary scenario than the majority of sequences, leading to phylogenetic noise blurring the phylogeny reconstruction. The pipeline that we propose attempts to solve such an issue by computing all phylogenetic trees which can be obtained by removing at most one core gene. In case where such a preliminary systematic approach does not solve the phylogeny, new investigation stages are added to the pipeline, namely a Monte-Carlo based random approach and two invocations of a genetic algorithm, separated by a Lasso test. The pipeline is finally tested on various sets of chloroplast genomes.

The remainder of this article is as follows. We start with Sect. 2 by giving a brief and global description of the problem. Genetic population initialization is discussed in Sect. 3, while the first optimization stage with genetic algorithm is fully detailed in Sect. 4. Targeting problematic genes using a Lasso test and the following second invocation of the genetic algorithm is detailed in Sect. 5. Then, in the next section, various plant families are tested as a case study. Finally this research work ends with a conclusion section in which the contributions are summarized and intended future work is outlined.

2 Presentation of the Problem

Let us consider a set of chloroplast genomes that have been annotated using DOGMA [16] (http://dogma.ccbb.utexas.edu/). We have then access to the core genome [1] (genes present everywhere) of these species, whose size is about one hundred genes when the species are close enough. For further information on how we found the core genome, see [1,2]. Sequences are further aligned with MUSCLE [5] and the RAxML [13] tool infers the corresponding phylogenetic tree. If this resulting tree is well-supported, then the process is stopped without further investigations. Indeed, if all bootstrap values are larger than 95, then we can reasonably consider that the phylogeny of these species is resolved, as the largest possible number of genes has led to a very well supported tree.

In case where some branches are not supported well, we can wonder whether a few genes can be incriminated in this lack of support. If so, we face an optimization problem: *find the most supported tree using the largest subset of core genes.* Obviously, a brute force approach investigating all possible combinations of genes is intractable, as it leads to 2^n phylogenetic tree inferences for a core genome of size n. To solve this optimization problem, we have proposed an hybrid approach mixing a genetic algorithm with the use of some statistical tests for discovering problematic genes. The initial population for the genetic algorithm is built by both systematic and random pre-GA investigations. These considerations led to a pipeline detailed in Fig. 1, whose stages will be developed thereafter.

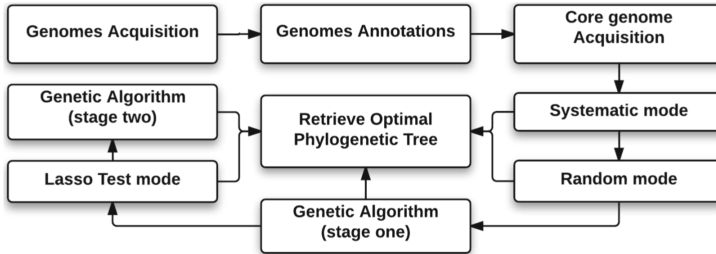
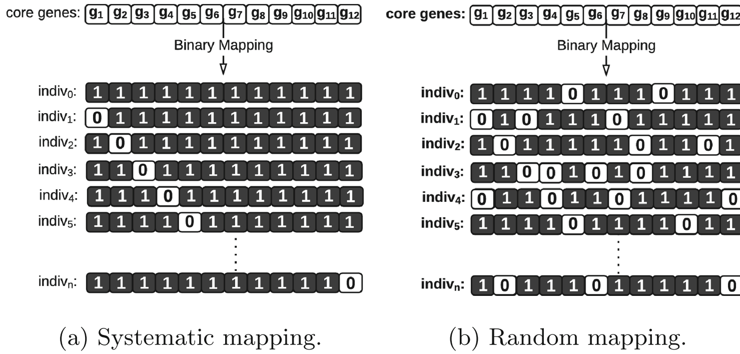


Fig. 1. Overview of the proposed pipeline for phylogenies based on chloroplasts.

3 Generation of the Initial Population

The objective is to obtain a well-supported phylogenetic tree by using the largest possible subset of genes. If this goal cannot be reached by taking all core genes, the first thing to investigate is to check whether one particular gene is responsible of this problem. Therefore we systematically compute all the trees we can obtain by removing exactly one gene from the core genome, leading to n new phylogenetic trees, where n is the core size (see Fig. 2a).

If, during this systematic approach, one well-supported tree is obtained, then it is returned as the phylogeny of the species under consideration. Conversely,



(a) Systematic mapping. (b) Random mapping.

Fig. 2. Binary mapping operation and genetic algorithm overview. (a) Initial individuals obtained in systematic mode stage. Two kinds of individuals are generated. First, by considering all genes in the core genome. Second, by omitting one gene sequentially depending on the core length. (b) Initial individuals are generated randomly in random stage (random mode) by omitting 2–10 genes randomly.

if all trees obtained have at least one problematic branch, then deeper investigations are required. However the systematic approach has reached its limits which is preliminary to GA, as investigating all phylogenetic trees that can be obtained by removing randomly 2 genes among a core genome of size n leads to $\frac{n(n-1)}{2}$ tree inferences. Obviously, the number of cases explodes, and it is illusory to hope to investigate all reachable trees by discarding 10 % of a core genome having 100 genes. This is why a genetic algorithm has been proposed.

Using the $n + 1$ computed trees to initialize the population of the genetic algorithm results in a population which remains too small and too homogeneous. Indeed, all these trees have been computed in the same way, each inference being produced using 99 % of the core genome (we have removed at most 1 gene in a core genome having approximately 100 genes). Thus, in order to increase the diversity of the initial population a second stage (random stage) as shown in Fig. 2b, which extracts large random subsets of core genome for inference, is applied.

More precisely, there are indeed two random stages. The first one operates during 200 iterations: at each iteration, an integer k between 2 and 10 is first randomly picked. Then k genes are randomly removed from the core genome, and a phylogeny is inferred using the remaining genes. If during these iterations, by chance, a very well supported tree is obtained, a stop signal is sent to the master process and the obtained tree is returned. If not, we now have enough data to build a relevant initial population for the genetic algorithm. And the second random stage is indeed included in this genetic algorithm.

4 Genetic Algorithm

A genetic algorithm is a well-known metaheuristic which has been described by a rich body of literature since its introduction in the mid-seventies [8,9].

In the following, we will only discuss the choices we made regarding operators and parameters. For further information and applications regarding the genetic algorithm, see, *e.g.*, [3,4,6,12].

4.1 Genotype and Fitness Value

Genes of the core genome are supposed to be ordered lexicographically. At each subset s of the core genome corresponds thus a unique binary word w of length n: for each i lower than n, w_i is 1 if the i-th core gene is in s, else w_i is equal to 0. At each binary word w of length n, we can associate its percentage p of 1's and the lowest bootstrap b of the phylogenetic tree we obtain when considering the subset of genes associated to w. At each word w we can thus associate as fitness value the score $b + p$, which must be as large as possible. We currently consider that bootstrap b and the number of genes p have the same importance in the scoring function. However, changing the weight of each parameter may be interesting in deeper investigations.

4.2 Genetic Process

Until now, binary words (genotypes) of length n that have been investigated are:

1. the word having only 1's (systematic mode);
2. all words having exactly one 0 (systematic mode);
3. 200 words having between 2 and 10 0's randomly located (random mode).

To each of these words is attached a score which is used to select the 50 best words, or fittest individuals, in order to build the initial population. After that, the genetic algorithm will loop during 200 iterations or until an offspring word such that $b \geqslant 95$ is obtained. During an iteration the algorithm will apply the following steps to produce a new population P' given a population P (see Fig. 3).

- Repeat 5 times a random pickup of a couple of words and mix them using a crossover approach. The obtained words are added to the population P, as described in Sect. 4.3, resulting in population P_c.
- Mutate 5 words of the population P_c, the mutated words being added too to P_c, as detailed in Sect. 4.4, leading to population P_m.
- Add 5 new random binary words having less than 10 % of 0's (see Sect. 4.5) to P_m producing population P_r.
- Select the 50 best words in population P_r to form the new population P'.

Let us now explain with more details each step of this genetic algorithm.

4.3 Crossover Step

Given two words w^1 and w^2, the idea of the crossover operation is to mix them, hoping by doing so to generate a new word w having a better score (see Fig. 4a). For instance, if we consider a one-point crossover located at the middle of the

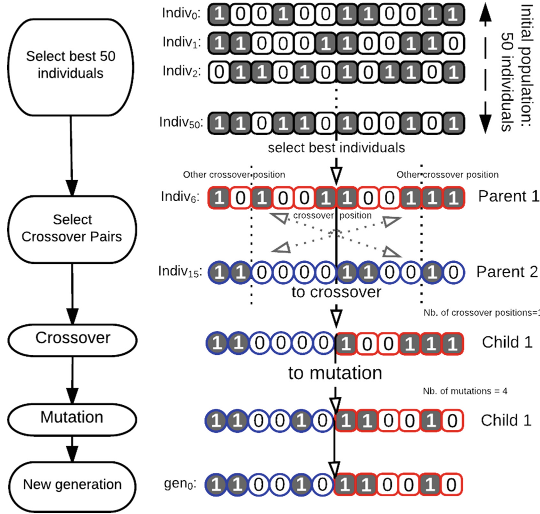


Fig. 3. Outlines of genetic algorithm.

words, for $i < \frac{n}{2}$, $w_i = w_i^1$, while for $i \geqslant \frac{n}{2}$, $w_i = w_i^2$: in that case, for the first core genes, the choice (to take them or not for phylogenetic construction) in w is the same than in w^1, while the subset of considered genes in w corresponds to the one of w^2 for the last 50 % of core genes.

More precisely, at each crossover step, we first pick randomly an integer k lower than $\frac{n}{2}$, and randomly again k different integers $i_1, \dots i_k$ such that $1 < i_1 < i_2 < \dots < i_k < n$. Then w^1 and w^2 are randomly selected from the population P, and a new word w is computed as follows:

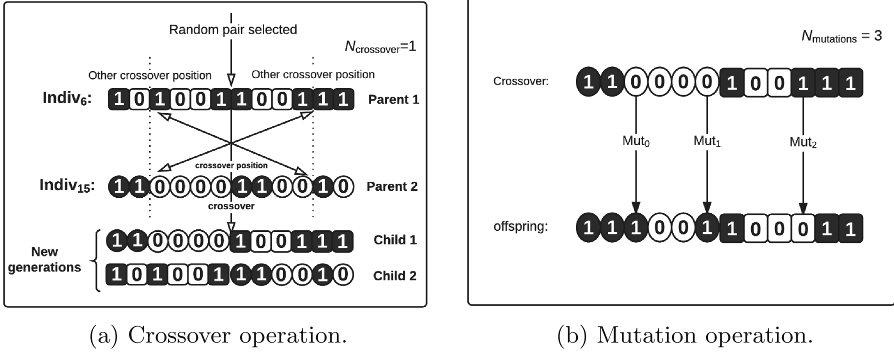
- $w_i = w_i^1$ for $i = 1, ..., i_1$,
- $w_i = w_i^2$ for $i = i_1 + 1, ..., i_2$,
- $w_i = w_i^1$ for $i = i_2 + 1, ..., i_3$,
- etc.

Then the phylogenetic tree based on the subset of core genes labeled by w is computed, the score s of w is deduced, and w is added to the population with the fitness value s attached to it.

4.4 Mutation Step

In this step, we ask whether changing a little a given subset of genes, by removing a few genes and adding a few other ones, may by chance improve the support of the associated tree. Similarly speaking, we try here to improve the score of a given word by replacing a few 0's by 1 and a few 1's by 0 (Fig. 4b).

In practice, an integer $k \leqslant \frac{n}{4}$ corresponding to the number of changes, or "mutations", is randomly picked. Then k different integers $i_1, \dots, i_k$ lower than



(a) Crossover operation. (b) Mutation operation.

Fig. 4. (a) Two individuals are selected from given population. First portion from determined crossover position in the first individual is switched with the first portion of the second individual. The number of crossover positions is determined by $N_{crossover}$. (b) Random mutations are applied depending on the value of $N_{mutation}$, changing randomly gene state from 1 to 0 or vice versa. New offsprings generated from this stage are predicted w.r.t natural evolution scenario.

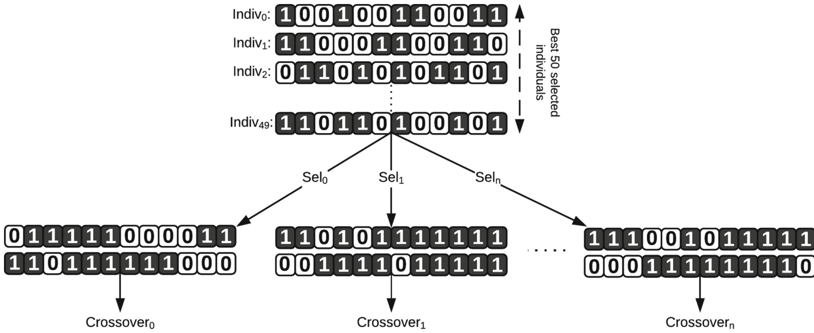


Fig. 5. Random pair selections from given population.

n are randomly chosen and a word w is randomly extracted from the current population. A new word w' is then constructed as follows: for each $i = 1, ..., n$,

- if i in $\{i_1, \ldots, i_k\}$, then $w'_i = w_i + 1 \mod 2$ (the bit is mutated),
- else $w'_i = w_i$ (no modification).

Again, the phylogenetic tree corresponding to the subset of core genes associated to w' is computed, and w' is added to the population together with its score.

4.5 Random Step

In this step, new words having a large amount of 1's are added to the population. Each new word is obtained by starting from the word having n 1s, followed by k random selection of 1s which are changed to 0, where k is an integer randomly

chosen between 1 and 10. The new word is added to the population after having computed its score thanks to a phylogenetic tree inference.

5 Targeting Problematic Genes Using Statistical Tests

5.1 The Lasso Test

After having carried out 200 iterations of the genetic algorithm detailed above, it may occur that no well-supported tree has been produced. Various reasons may explain this failure, like a lazy convergence speed, a large number of problematic genes (*e.g.*, homoplasic ones, or due to stochastic errors, undetected paralogy, incomplete lineage sorting, horizontal gene transfers, or hybridization), or close divergence species leading to very small branch lengths between two internal nodes. However, we now have computed enough word scores to determine the effects of each gene in topologies and bootstraps, and to remove the few genes that break supports.

The idea is then to investigate each topology that has appeared enough times during previous computations. In this study, we only consider topologies having a frequency of occurrence larger than 10 %. Remark that this 10 % is convenient for the given case study, but it must depend in fact on the number of obtained topologies. Then for each best word of these best topologies, and for each problematic bootstrap in its associated tree, we apply a Lasso approach as follows.

The Lasso (Least Absolute Shrinkage and Selection Operator) test [15] is an estimator that takes place in the category of least-squares regression analysis. Like all the algorithms in this group, it estimates a linear model which minimizes a residual sum with respect to a variable λ. Let us explain how this variable can be used to order genes with respect to their ability to modify the bootstrap support.

Let X be a $m \times p$ matrix where each line $X_i = (X_{i1}, \ldots, X_{ij}, \ldots X_{ip})$, $1 \leq i \leq m$, is a configuration where X_{ij} is 1 if gene number j is present inside the configuration i and X_{ij} is 0 otherwise. For each X_i, let Y_i be the real positive support value for each problematic bootstrap b per topology and per gene. According to [15], the Lasso test $\beta = (\beta_1, \ldots, \beta_i, \ldots \beta_p)$ is defined by

$$\beta = \text{argmin} \left\{ \sum_{i=1}^{m} \left(Y_i - \sum_{j=1}^{p} \beta_j X_{ij} \right)^2 + \lambda \sum_{j=1}^{p} |\beta_j| \right\}. \quad (1)$$

When λ has high value, all the β_j are null. It is thus sufficient to decrease the value of λ to observe that some β_j become not null. Moreover, the sign of β_j is positive (resp. negative) if the bootstrap support increases (resp. decreases) with respect to j.

5.2 Second Stage of Genetic Algorithm

Targeting problematic genes using Lasso approach can solve the issue of badly supported values in some cases, especially when only one support is lower than the predefined threshold. In cases where at least two branches are not well supported, removing genes that break the first support may or may not has an effect on the second problematic support. In other words, each of the two problematic supports can be separately solved using Lasso investigations, but not necessarily both together.

However, the population has been improved, receiving very interesting words for each problematic branch. Then a last genetic algorithm phase is launched on the updated population, in order to mix these promising words by crossover operations, hoping by doing so to solve in parallel all of the badly supported values. This last stage runs until either the resolution of all problematic bootstraps or the reach of iterations limit (set to 1000 in our simulations).

6 Case Studies

6.1 Pipeline Evaluation on Various Groups of Plant Species

In this section, the proposed pipeline is tested on various sets of close plant species. An example of 50 subgroups (ranging on average from 12 to 15 chloroplasts species) encompassing 356 plant species is presented in Table 1. The *Stage* column contains the termination step for each subgroup, namely: the systematic (code 1), random (2), or optimization stages (3) using genetic algorithm and/or Lasso test. A large occurrence in this table means that the associated group and/or subgroups has its computation terminated in either penultimate or last pipeline stage. An occurrence of 31 is frequent due to the fact that 32 MPI threads (one master plus 31 slaves) have been launched on our supercomputer facilities. Notice that the Table 1 is divided into four parts: groups of species stopped in systematic stage with weak bootstrap values (which is due to the fact that a upper time limit has been set for each group and/or subgroups, while each computed tree in these remarkable groups needed a lot of times for computations), subgroups terminated during systematic stage with desired bootstrap value, groups or subgroups terminated in random stage with desired bootstrap value, and finally, groups or subgroups terminated during optimization stages. The majority of subgroups has its phylogeny satisfactorily resolved, as can be seen on all obtained trees which are downloadable at http://meso.univ-fcomte.fr/peg/phylo. In what follows, an example of problematic group, namely the *Apiales*, is more deeply investigated as a case study.

6.2 Investigating *Apiales* Order

In our study *Apiales* chloroplasts consist of two sets, as detailed in Table 2: two species belong to the *Apiaceae* family set (namely *Daucus carota* and *Anthriscus cerefolium*), while the remaining seven species are in the *Araliaceae* family set.

Table 1. Families applied on pipeline stages

Group or subgroup	Occurrences	Core genes	# Species	L.Bootstrap	Pip. Stage	Likelihood	Outgroup
Gossypium_group_0	85	84	12	26	1	−84187.03	*Theo_cacao*
Ericales	674	84	9	67	3	−86819.86	*Dauc_carota*
Eucalyptus_group_1	83	82	12	48	1	−62898.18	*Cory_gummifera*
Caryophyllales	75	74	10	52	1	−145296.95	*Goss_capitis-viridis*
Brassicaceae_group_0	78	77	13	64	1	−101056.76	*Cari_papaya*
Orobanchaceae	26	25	7	69	1	−19365.69	*Olea_maroccana*
Eucalyptus_group_2	87	86	11	71	1	−72840.23	*Stoc_quadrifida*
Malpighiales	1183	78	12	80	3	−95077.52	*Mill_pinnata*
Pinaceae_group_0	76	75	6	80	1	−76813.22	*Juni_virginiana*
Pinus	80	79	11	80	1	−69688.94	*Pice_sitchensis*
Bambusoideae	83	81	11	80	3	−60431.89	*Oryz_nivara*
Chlorophyta_group_0	231	24	8	81	3	−22983.83	*Olea_europaea*
Marchantiophyta	65	64	5	82	1	−117881.12	*Pice_abies*
Lamiales_group_0	78	77	8	83	1	−109528.47	*Caps_annuum*
Rosales	81	80	10	88	1	−108449.4	*Glyc_soja*
Eucalyptus_group_0	2254	85	11	90	3	−57607.06	*Allo_ternata*
Prasinophyceae	39	43	4	97	1	−66458.26	*Oltm_viridis*
Asparagales	32	73	11	98	1	−88067.37	*Acor_americanus*
Magnoliidae_group_0	326	79	4	98	3	−85319.31	*Sacc_SP80-3280*
Gossypium_group_1	66	83	11	98	1	−81027.85	*Theo_cacao*
Triticeae	40	80	10	98	1	−72822.71	*Loli_perenne*
Corymbia	90	85	5	98	2	−65712.51	*Euca_salmonophloia*
Moniliformopses	60	59	13	100	1	−187044.23	*Prax_clematidea*
Magnoliophyta_group_0	31	81	7	100	1	−136306.99	*Taxu_mairei*
Liliopsida_group_0	31	73	7	100	1	−119953.04	*Drim_granadensis*
basal_Magnoliophyta	31	83	5	100	1	−117094.87	*Ascl_nivea*
Araucariales	31	89	5	100	1	−112285.58	*Taxu_mairei*
Araceae	31	75	6	100	1	−110245.74	*Arun_gigantea*
Embryophyta_group_0	31	77	4	100	1	−106803.89	*Stau_punctulatum*
Cupressales	87	78	11	100	2	−101871.03	*Podo_totara*
Ranunculales	31	71	5	100	1	−100882.34	*Cruc_wallichii*
Saxifragales	31	84	4	100	1	−100376.12	*Aral_undulata*
Spermatophyta_group_0	31	79	4	100	1	−94718.95	*Mars_crenata*
Proteales	31	85	4	100	1	−92357.77	*Trig_doichangensis*
Poaceae_group_0	31	74	5	100	1	−89665.65	*Typh_latifolia*
Oleaceae	36	82	6	100	1	−84357.82	*Boea_hygrometrica*
Arecaceae	31	79	4	100	1	−81649.52	*Aegi_geniculata*
PACMAD_clade	31	79	9	100	1	−80549.79	*Bamb_emeiensis*
eudicotyledons_group_0	31	73	4	100	1	−80237.7	*Eryc_pusilla*
Poeae	31	80	4	100	1	−78164.34	*Trit_aestivum*
Trebouxiophyceae	31	41	7	100	1	−77826.4	*Ostr_tauri*
Myrtaceae_group_0	31	80	5	100	1	−76080.59	*Oeno_glazioviana*
Onagraceae	31	81	5	100	1	−75131.08	*Euca_cloeziana*
Geraniales	31	33	6	100	1	−73472.77	*Ango_floribunda*
Ehrhartoideae	31	81	5	100	1	−72192.88	*Phyl_henonis*
Picea	31	85	4	100	1	−68947.4	*Pinu_massoniana*
Streptophyta_group_0	31	35	7	100	1	−68373.57	*Oedo_cardiacum*
Gnetidae	31	53	5	100	1	−61403.83	*Cusc_exaltata*
Euglenozoa	29	26	4	100	3	−8889.56	*Lath_sativus*

Table 2. Genomes information of Apiales

Organism name	Accession	Genome Id	Sequence length	Number of genes	Lineage
Daucus carota	NC_008325.1	114107112	155911 bp	138	Apiaceae
Anthriscus cerefolium	NC_015113.1	323149061	154719 bp	132	Apiaceae
Panax ginseng	NC_006290.1	52220789	156318 bp	132	Araliaceae
Eleutherococcus senticosus	NC_016430.1	359422122	156768 bp	134	Araliaceae
Aralia undulata	NC_022810.1	563940258	156333 bp	135	Araliaceae
Brassaiopsis hainla	NC_022811.1	558602891	156459 bp	134	Araliaceae
Metapanax delavayi	NC_022812.1	558602979	156343 bp	134	Araliaceae
Schefflera delavayi	NC_022813.1	558603067	156341 bp	134	Araliaceae
Kalopanax septemlobus	NC_022814.1	563940364	156413 bp	134	Araliaceae

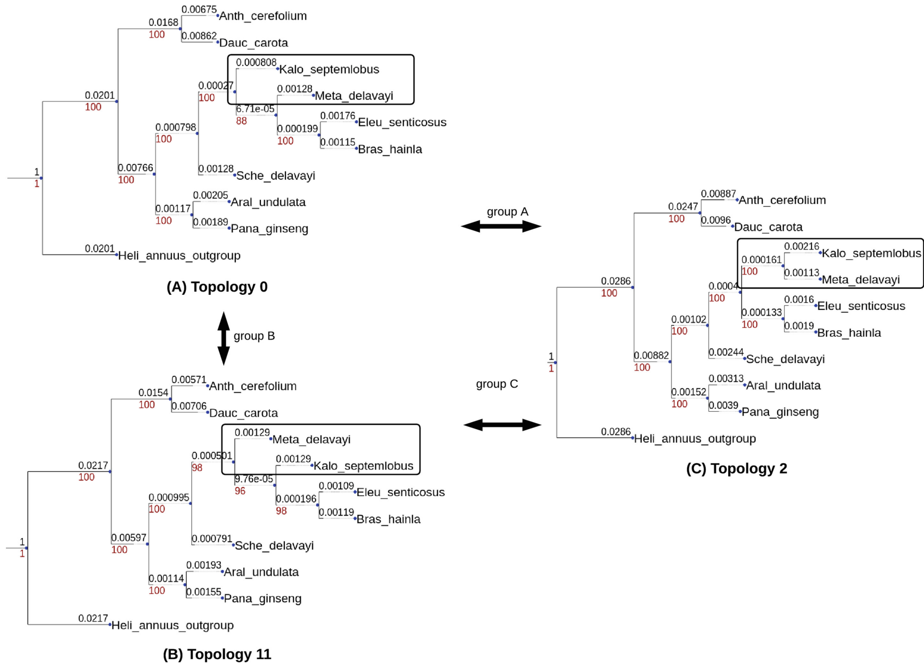


Fig. 6. Best trees of topologies 0, 11, and 2.

These latter are: *Panax ginseng, Eleutherococcus senticosus, Aralia undulata, Brassaiopsis hainla, Metapanax delavayi, Schefflera delavayi*, and *Kalopanax septemlobus*. Chloroplasts of *Apiales* are characterized by having highly conserved gene content and order [11].

Method to Select Best Topologies. We define $T = [t_0, t_1, ..., t_m]$ as a list of $m = 9,053$ obtained trees from given pipeline. By comparing each tree t_i in T with the other trees in T, a set of topologies is then numbered and defined as $W = \{w_0, w_1, w_2, ..., w_n\}$, where w_i is the topology of number i. Let $f(x)$ be a function on W which represents the number of trees having x for their topology.

We say that a given topology w_i is selected as the best topology if and only if $f(w_i) \geq lb$ where lb is the lower bound threshold computed by the following formula

$$lb = \frac{m * \gamma}{100}$$

γ is a constant value between $1 - 10$ and m is the size of T. Then x is stored as best topology.

Pratical Results. In our case, $\gamma = 8$, meaning that we exclude as noise the topologies representing less than 8 % from the given trees. Three from 43 identified tree topologies are selected, with a number of occurrences $f(x)$ above $lb = 724$, as the best topologies as shown in Table 3. In this table, topologies 0 and 11 are delivered from optimization stages when the desired bootstrap value is set to 96, and topology 2 is obtained from systematic stage when we increase the desired bootstrap to 100. The best obtained phylogenetic trees from selected topologies are provided in Table 3: in this table *Min.Bootstrap* is higher than *Avg.Bootstrap*, as the former represents the lowest bootstrap value of the best tree in the given topology, while *Avg.Bootstrap* consists of the average lowest bootstrap in all trees having this topology.

As it can be noted, only 3 of the 43 obtained topologies contain trees whose lowest bootstrap is larger than 87, namely 0, 11, and 2. It is not so easy to make the decision, since all selected trees are very closed to each other with small differences. A new question needs to be answered: which genes are responsible for changing the tree from topology$_0$ to topology$_{11}$, or to topology$_2$? Deep investigations are needed in future work to answer this new question and to discover the set of genes in $group_A$, $group_B$, $andgroup_C$ that change one tree topology to another one (see Fig. 6).

The only notable difference between topologies 0 and 11 is the taxa position of *Kalo_septemlobus* and *Meta_delavayi*. In the same way, there is only one difference between topologies 0 and 11 with 2: grouping the same two taxa of *Kalo_septemlobus* and *Meta_delavayi*. Different comparisons on trees provided with selected topologies are summarized in Fig. 7.

Table 3. Information regarding obtained topologies

Topology	Min.Bootstrap	Avg.Bootstrap	Occurrences ($f(x)$)	Gene rate (%)
0	88	56	5422	64.7
11	96	76	2579	44.8
2	100	68	787	99.1
8	72	50	89	44.8
9	49	29	48	35.3
14	61	48	31	25
5	80	48	21	34.5
20	63	53	11	53.4
10	62	50	8	68.1

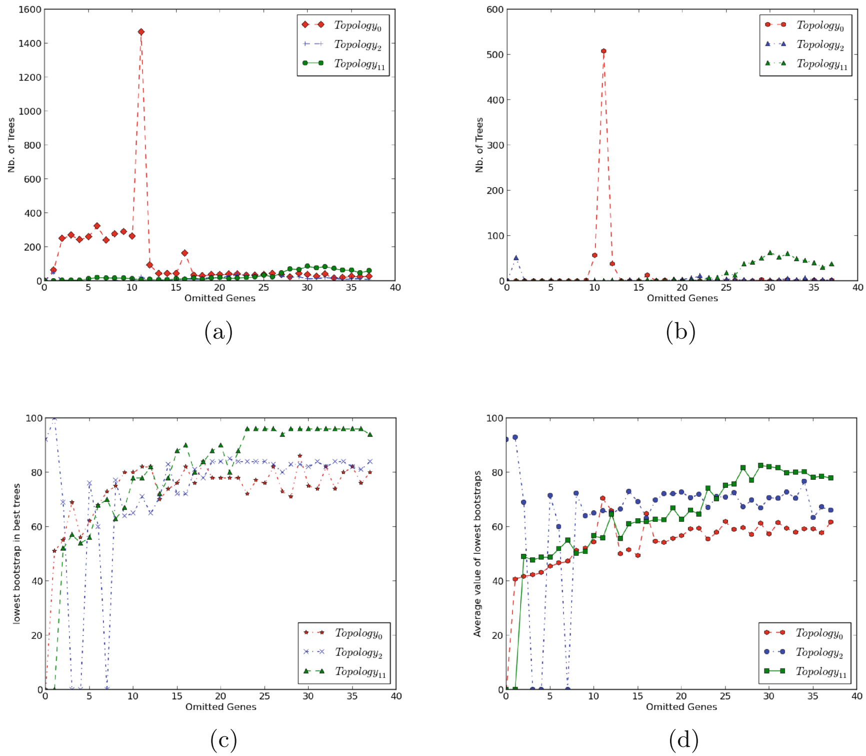


Fig. 7. Different comparisons of the topologies w.r.t the amount of removed genes: the number of disregarded genes in these figures is specified by $\frac{n}{3}$ where n is the number of core genes. (a) Number of trees per topology, (b) number of trees whose lowest bootstrap is larger than or equal to 80, (c) lowest bootstrap in best trees, and (d) the average of lowest bootstraps.

7 Conclusion

In this study, an many stages pipeline have been applied (namely: systematic mode, random mode, GA stage one, Lasso test mode, and GA stage two) for inferring trustworthy phylogenetic trees from various plant groups. We have verified that inferring a phylogenetic tree based on either the full set or some subsets of common core genes does not always lead to good support of the phylogenetic reconstruction. In both systematic and random stages, many trees have been generated based on omitting randomly some genes. When the desired score was not reached, a genetic algorithm has then been applied inside two specific stages using previously generated trees, to find new optimized solutions after realizing crossover and mutation operations. Furthermore, we applied a Lasso test for identifying and removing systematically blurring genes, discarding so those which have the worst impact on supports. We tested this pipeline on 322 different plant groups, where 63 of them are base families while the remaining ones are random trees, these latter playing the rule of skeletons when reconstructing

the supertree. A case study regarding *Apiales* order is analyzed and three "best" topologies stand out from the 43 obtained. Deep investigation will be needed in future work, in order to discover which genes change the topology, and to deeply investigate the sequences of the genes that blur the signal, to find the reasons of such effects.

Acknowledgement. *Computations have been performed on the supercomputer facilities of the Mésocentre de calcul de Franche-Comté.*

References

1. Alkindy, B., Couchot, J.F., Guyeux, C., Mouly, A., Salomon, M., Bahi, J.M.: Finding the core-genes of chloroplasts. J. Biosci. Biochem. Bioinform. **4**(5), 357–364 (2014)
2. Alkindy, B., Guyeux, C., Couchot, J.-F., Salomon, M., Bahi, J.M.: Gene similarity-based approaches for determining core-genes of chloroplasts. In: 2014 IEEE International Conference on Bioinformatics and Biomedicine (BIBM), pp. 71–74. IEEE (2014)
3. Bhandari, D., Murthy, C., Pal, S.K.: Genetic algorithm with elitist model and its convergence. Int. J. Pattern Recogn. Artif. Intell. **10**(06), 731–747 (1996)
4. Booker, L.B., Goldberg, D.E., Holland, J.H.: Classifier systems and genetic algorithms. Artif. Intell. **40**(1), 235–282 (1989)
5. Edgar, R.C.: Muscle: multiple sequence alignment with high accuracy and high throughput. Nucleic Acids Res. **32**(5), 1792–1797 (2004)
6. Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley, Reading (1993)
7. Gupta, M., Singh, S.: A novel genetic algorithm based approach for optimization of distance matrix for phylogenetic tree construction. Int. J. Comput. Appl. **52**(9), 14–18 (2012)
8. Holland, J.H.: Adaptation in Natural and Artificial Systems. University of Michigan Press, Ann Arbor (1975)
9. Holland, J.H.: Adaptation in Natural and Artificial Systems, 2nd edn. MIT Press, Cambridge (1992)
10. Matsuda, H.: Construction of phylogenetic trees from amino acid sequences using a genetic algorithm. In: Proceedings of Genome Informatics Workshop, vol. 6, pp. 19–28 (1995)
11. Palmer, J.D.: Plastid chromosomes: structure and evolution. Mol. Biol. Plastids **7**, 5–53 (1991)
12. Prebys, E.K.: The genetic algorithm in computer science. MIT Undergrad. J. Math **2007**, 165–170 (2007)
13. Stamatakis, A., Ludwig, T., Meier, H.: Raxml-iii: a fast program for maximum likelihood-based inference of large phylogenetic trees. Bioinformatics **21**(4), 456–463 (2005)
14. Tate, S.I., Yoshihara, I., Yamamori, K., Yasunaga, M.: A parallel hybrid genetic algorithm for multiple protein sequence alignment. In: Proceedings of the World on Congress on Computational Intelligence, vol. 1, pp. 309–314. IEEE (2002)
15. Tibshirani, R.: Regression shrinkage and selection via the lasso. J. Roy. Stat. Soc. (Ser. B) **58**, 267–288 (1996)
16. Wyman, S.K., Jansen, R.K., Boore, J.L.: Automatic annotation of organellar genomes with dogma. Bioinformatics **20**(17), 3252–3255 (2004). Oxford Press

Constructing and Employing Tree Alignment Graphs for Phylogenetic Synthesis

Ruchi Chaudhary[1](✉), David Fernández-Baca[2], and J. Gordon Burleigh[1]

[1] Department of Biology, University of Florida, Gainesville, FL 32611, USA
ruchic@ufl.edu

[2] Department of Computer Science, Iowa State University, Ames, IA 50011, USA

Abstract. Tree alignment graphs (TAGs) provide an intuitive data structure for storing phylogenetic trees that exhibits the relationships of the individual input trees and can potentially account for nested taxonomic relationships. This paper provides a theoretical foundation for the use of TAGs in phylogenetics. We provide a formal definition of TAG that — unlike previous definition — does not depend on the order in which input trees are provided. In the consensus case, when all input trees have the same leaf labels, we describe algorithms for constructing majority-rule and strict consensus trees using the TAG. When the input trees do not have identical sets of leaf labels, we describe how to determine if the input trees are compatible and, if they are compatible, to construct a supertree that contains the input trees.

1 Introduction

Phylogenetic trees are graphs depicting the evolutionary relationships among species; thus, they are powerful tools for examining fundamental biological questions and understanding biodiversity (e.g., [1]). The wealth of available genetic sequences has rapidly increased the number of phylogenetic studies from across the tree of life (e.g., [2]). For example, STBase contains a million species trees generated from sequence data in GenBank [3]. New next-generation sequencing technologies and sequence capture methods (e.g., [4–6]) will further increase the rate in which phylogenetic data is generated in the coming years. This continuous flow of new phylogenetic data necessitates new approaches to store, evaluate, and synthesize existing phylogenetic trees.

Recently Smith et al. introduced tree alignment graphs (TAGs) as a way to analyze large collections of phylogenetic trees [7]. TAGs preserve the structure of the input trees and thus provide an intuitive, interpretable representation of the input trees, which enables users to visually assess patterns of agreement and conflict. The TAG structure also makes it possible to combine trees whose tips include nested taxa (e.g., the tips of one tree contain species in taxonomic families, while the tips of another tree contain the families), which was true of only a few previous synthesis approaches [8–10]. Indeed, a TAG was used to merge a taxonomy of all ∼2.3 million named species with ∼500 published phylogenetic trees to obtain an estimate of the tree of life [11].

© Springer International Publishing Switzerland 2015
A.-H. Dediu et al. (Eds.): AlCoB 2015, LNBI 9199, pp. 97–108, 2015.
DOI: 10.1007/978-3-319-21233-3_8

The original TAG definition of Smith et al. [7] depends on the order of the input trees, which can be problematic. Further, the several details of the synthesis process were not specified. Our aim in this paper is to lay the theoretical foundations for further research on TAGs. To this end, we first provide a mathematically precise definition of TAGs which is independent of the order of the input trees (Sect. 2), and develop an algorithm for constructing TAGs (Sect. 3). We also describe algorithms to build strict and majority-rule consensus trees using TAGs (Sect. 4). We show how to check the compatibility among input trees and construct a supertree from compatible phylogenetic trees using a TAG (Sect. 5). Finally, we discuss the future applications and problems associated with using TAGs for assessing and synthesizing the enormous and rapidly growing number of available phylogenetic trees in the future (Sect. 6).

Related work. TAGs are part of a long history of using graph structures to synthesize the relationships among phylogenetic trees with partial taxonomic overlap. The classic example is the Build algorithm [12,13] and its later variations (e.g., [9,10,14,15]). These methods yield polynomial-time algorithms to determine whether a collection of input trees is compatible, and, if so, output the parent tree(s). Other graph-based algorithms, such as the MinCutSupertree [16], the Modified MinCutSupertree [17], or the MultiLevelSupertree [8] algorithm allow users to synthesize collections of conflicting phylogenetic trees. Although TAGs share important features with these earlier graph-theoretic approaches, TAGs display more directly the phylogenetic relationships exhibited by the input trees and therefore provide a more intuitive framework to examine patterns of conflict among trees [7]. TAGs also potentially summarize the information in the input trees with fewer nodes than previous graphs for semi-labeled trees [9,10].

2 Preliminaries

2.1 Notation

Let T be a rooted tree. Then, $\mathrm{rt}(T)$ and $\mathcal{L}(T)$ denote, respectively, the root and the leaf set of T, and $V(T)$ and $E(T)$ denote, respectively, the set of vertices and the set of edges of T. The set of all internal vertices of T is $I(T) := V(T) \backslash \mathcal{L}(T)$. We define $\leq_T$ to be the partial order on $V(T)$ where $x \leq_T y$ if y is a vertex on the path from $\mathrm{rt}(T)$ to x. If $\{x, y\} \in E(T)$ and $x \leq_T y$, then y is the *parent* of x and x is a *child* of y. Two vertices in T are *siblings* if they share a parent.

Let X be a finite set of *labels*. A *phylogenetic tree on* X is a pair $\mathcal{T} = (T, \varphi)$ where 1) T is a rooted tree in which every internal vertex has degree at least three, except $\mathrm{rt}(T)$, which has degree at least two, and 2) φ is a bijection from $\mathcal{L}(T)$ to X [13]. Tree T is called the *underlying tree* of $\mathcal{T}$ and φ is called the *labeling map* of $\mathcal{T}$. For convenience, we will often assume that the set of labels X of $\mathcal{T}$ is simply $\mathcal{L}(\mathcal{T})$. The *size* of $\mathcal{T}$ is the cardinality of $\mathcal{L}(T)$. $\mathcal{T}$ is *binary* (or *fully resolved*) if every vertex $v \in I(T) \setminus \mathrm{rt}(T)$ has degree three and $\mathrm{rt}(T)$ has degree two.

Let $\mathcal{T} = (T, \varphi)$ be a phylogenetic tree on X and let v be any vertex in $V(T)$. The subtree of T rooted at vertex $v \in V(T)$, denoted by T_v, is the tree induced by $\{u \in V(T) : u \leq v\}$. The *cluster at* v, denoted $C_{\mathcal{T}}(v)$, is the set of leaf labels $\{\varphi(u) \in X : u \in \mathcal{L}(T_v)\}$. We write $\mathcal{H}(\mathcal{T})$ to denote the set of all clusters of $\mathcal{T}$. Note that $\mathcal{H}(\mathcal{T})$ includes trivial clusters; i.e., clusters of size one or $|X|$.

2.2 Tree Alignment Graphs

Here we define the tree alignment graph (TAG). Our definition is somewhat different from that of Smith et al. [7]. We explain these differences later.

We first need an auxiliary notion. A *directed multi-graph* is a directed graph that is allowed to have multiple edges between the same two vertices. More formally, a directed multi-graph is a pair (V, E) of disjoint sets (of vertices and edges) together with two maps $\text{init} : E \to V$ and $\text{ter} : E \to V$ assigning to each edge e an *initial vertex* $\text{init}(e)$ and a *terminal vertex* $\text{ter}(e)$ [18]. Edge e is said to be *directed* from $\text{init}(e)$ to $\text{ter}(e)$. The *in-degree* of a node v is the number of edges e such that $\text{ter}(e) = v$; the *out-degree* of v is the number of edges e such that $\text{init}(e) = v$. We call a node with out-degree zero a *leaf node*; all non-leaf nodes are called *internal nodes*.

A *directed acyclic (multi-) graph*, *DAG* for short, is a directed multi-graph with no cycles.

Definition 1 (Tree Alignment Graph (TAG)). *Let $\mathcal{P}$ be a collection of phylogenetic trees and $S = \bigcup_{\mathcal{T} \in \mathcal{P}} \mathcal{L}(\mathcal{T})$. The* tree alignment graph *of $\mathcal{P}$ is a directed graph $D = (U, E)$ along with an injective function $f : U \to 2^S$, called the* vertex-labeling function, *such that*

- *for each $v \in U$, $f(v) \in \mathcal{H}(\mathcal{T})$ for some $\mathcal{T} \in \mathcal{P}$, and*
- *for each $\mathcal{T} := (T, \varphi) \in \mathcal{P}$ and each $e := \{x, y\} \in E(T)$ where $x <_T y$, there exists a unique $e' \in E$ such that $C_{\mathcal{T}}(x) = f(\text{ter}(e'))$ and $C_{\mathcal{T}}(y) = f(\text{init}(e'))$.*

Figure 1 illustrates Definition 1.

Remarks:

1. Note that we only use the vertex-labeling function f to facilitate the definition and to label the leaf nodes of the TAG. We do not actually label the internal vertices of the TAG, since, as the TAG gets bigger, assigning labels using f becomes impractical.
2. Having a unique edge in the TAG for each input tree edge enables systematically annotating the TAG for each individual input tree, as the TAG also provides a means for storing phylogenetic trees.

Lemma 1. *The TAG is acyclic.*

Proof. Stems from the fact that for each edge $e \in E$, $f(\text{ter}(e)) \subset f(\text{init}(e))$. $\square$

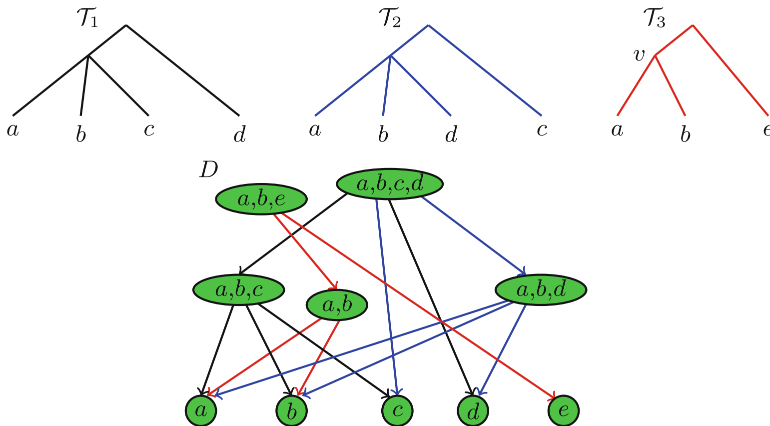


Fig. 1. A collection $\mathcal{P}$ of phylogenetic trees $\mathcal{T}_1$, $\mathcal{T}_2$, and $\mathcal{T}_3$ and the TAG D of $\mathcal{P}$. The edges of D are colored so as to correspond to the input trees; the internal vertices are labeled for clarity (Color figure online).

Comparison with Smith et al.'s TAG. In [7], Smith et al. define their TAG procedurally, as follows. Let $\mathcal{P}$ be a collection of phylogenetic trees and $S = \bigcup_{\mathcal{T} \in \mathcal{P}} \mathcal{L}(\mathcal{T})$. Let $D = (U, E)$ be a directed graph along with an injective vertex-labeling function $f : U \rightarrow 2^S$. Initially, D has a vertex $f^{-1}(S)$, and $|S|$ vertices $f^{-1}(\{s\})$, for each $s \in S$. Next, Smith et al.'s method process each input tree $\mathcal{T} = (T, \varphi) \in \mathcal{P}$, in some order, and does the following:

1. Map each vertex $v \in \mathcal{L}(T)$ to vertex $u \in U$ where $\varphi(v) = f(u)$.
2. Map each vertex $v \in I(T)$ to the vertex $u \in U$, where $C_{\mathcal{T}}(v) \cap f(u) \neq \phi$, $\mathcal{L}(\mathcal{T}) \setminus C_{\mathcal{T}}(v) \cap f(u) = \phi$, and $C_{\mathcal{T}}(v) \cap S \setminus f(u) = \phi$. If no such u exists, then add new vertex u with $f(u) := C_{\mathcal{T}}(v)$ in D.
3. In the case of a vertex $v \in I(T)$ mapping to multiple vertices in D, for each such t vertices $u_1, ..., u_t \in U$, where for each $j \in \{1, \ldots, t-1\}$ there exists $e' \in E$ such that $\text{ter}(e') = u_j$ and $\text{init}(e') = u_{j+1}$, discard all mapping of v to $u_2, ..., u_t$, except u_1. Note that $v \in V(T)$ can still be mapped to multiple vertices in D. For example, vertex v of T_3 in Fig. 1 is mapped to vertices $f^{-1}(\{a, b, c\})$ and $f^{-1}(\{a, b, d\})$ of D_1 in Fig. 2.
4. For edge $e = \{x, y\} \in E(T)$, add directed edges in D from all mappings of x to all mappings of y.

Observe that Smith et al.'s definition of the TAG coincides with Definition 1 when the input trees have completely overlapping leaf label sets; however, it differs when the input trees have partially overlapping leaf label sets, as we discuss next.

Notice that Step 2 first tries to map a vertex $v \in I(T)$ to a vertex $u \in U$ for which $C_{\mathcal{T}}(v) \subseteq f(u)$, and $f(u)$ does not have any label of $\mathcal{L}(T)$ other than that of $C_{\mathcal{T}}(v)$. If no suitable match exists, then a new vertex $f^{-1}(C_{\mathcal{T}}(v))$ is added to D.

As a result, the set of vertices in the TAG that this procedure creates can depend on the order of input trees. Figure 2 illustrates how changing the order of input trees can lead to different TAGs for the input trees of Fig. 1. When $\mathcal{T}_3$ is processed after $\mathcal{T}_1$ and $\mathcal{T}_2$, the vertex v of $\mathcal{T}_3$ maps to $f^{-1}(\{a,b,c\})$ and $f^{-1}(\{a,b,d\})$ in Step 2. On the other hand, processing $\mathcal{T}_3$ before $\mathcal{T}_1$ and $\mathcal{T}_2$, necessitates the creation of $f^{-1}(\{a,b\})$ in the resulting TAG.

Smith et al. discussed the possibility of order dependence of their TAG and addressed it through a post-processing procedure [7]. For the given collection of input trees and the TAG that results from the first round of processing, the post-processing procedure recomputes the mapping of each internal vertex of the input tree following Step 2. If the new mapping of an internal vertex of the input tree differs from the old mapping, then the mapping is updated. Edges of the resulting TAG that correspond to the outdated mapping are removed and edges for the new mapping are added. For example, there will be no change in TAG D_1 after applying post-processing procedure. On the contrary, post-processing will map $v \in I(\mathcal{T}_3)$ to $f^{-1}(\{a,b,c\})$ and $f^{-1}(\{a,b,d\})$ in D_2. This new mapping will cause adding directed edges, 1) from $f^{-1}(\{a,b,c,d,e\})$ to $f^{-1}(\{a,b,c\})$ and $f^{-1}(\{a,b,d\})$, 2) from $f^{-1}(\{a,b,c\})$ to $f^{-1}(\{a\})$ and $f^{-1}(\{b\})$, and 3) from $f^{-1}(\{a,b,d\})$ to $f^{-1}(\{a\})$ and $f^{-1}(\{b\})$ in D_2. Let D_2' be the resulting TAG after applying post-processing on D_2. Clearly, D_1 and D_2' are different. We note that since the post-processing is inadequate for overcoming order-dependence, an algorithm for pre-processing of input trees is in development[1]. In contrast to Smith et al.'s TAG [7], our TAG (in Definition 1) is independent of the order of input trees.

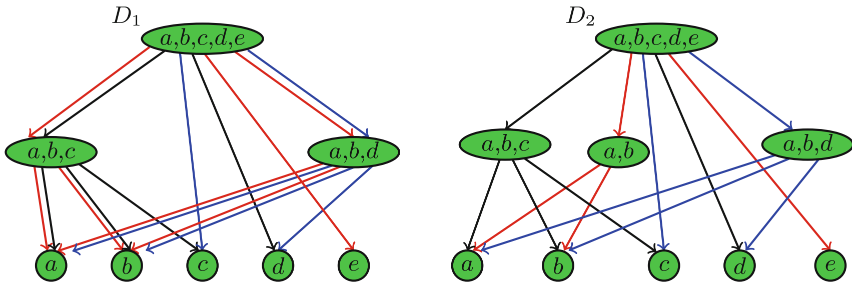


Fig. 2. Following [7], if the input trees from Fig. 1 are processed in the order $\mathcal{T}_1$, $\mathcal{T}_2$, and $\mathcal{T}_3$, the resulting TAG is D_1; changing the order of input trees, and processing them as $\mathcal{T}_3$, $\mathcal{T}_1$, and $\mathcal{T}_2$, results into D_2, which is different from D_1. Again, the edges of both TAGs are colored so to correspond to the input trees; the internal vertices are labeled for clarity.

The input trees in [7] always include a taxonomy tree, which contains all of the leaf labels from the input trees and could be a star tree (i.e., all the leaf

[1] S. A. Smith, J. W. Brown, and C. E. Hinchliff (Department of Ecology and Evolutionary Biology, University of Michigan, Ann Arbor), personal communication.

nodes connecting to a root node) if no taxonomic classification is available. Here we study the TAG in the standard supertree framework, so we do not assume that a taxonomy tree is included.

3 Constructing a TAG

We now present an algorithm for constructing the TAG for a collection of phylogenetic trees. The algorithm first collects clusters by reading through the input trees and making each unique cluster a node in the TAG. It then adds edges between nodes in the TAG.

Let $\mathcal{P}$ be the input collection of k phylogenetic trees. Let $S = \bigcup_{\mathcal{T} \in \mathcal{P}} \mathcal{L}(\mathcal{T})$ and $n = |S|$.

3.1 Building TAG Nodes

We define a bijection g that maps each taxon in S to a unique number in $\{1, 2, \ldots, n\}$. For each tree $\mathcal{T} = (T, \varphi)$ in $\mathcal{P}$, the *bit-string* of $v \in V(T)$ is a binary string of length n, where the ith bit is 1, if $g^{-1}(i) \in C_{\mathcal{T}}(v)$, or 0, otherwise. We collect bit-strings from the input trees in a list $\mathcal{A}$ and construct a TAG node for each unique bit-string.

Collecting bit-strings: The algorithm starts by traversing each input tree in post-order. When, after traversing its subtree, a vertex v is visited, we compute v's bit-string as follows. If v is a leaf with label $s \in S$, the bit string for v is simply the string of length n with a 1 at position $g(s)$ and 0 s everywhere else. If v is an internal node, its bit-string is the *OR* of the bit-strings of v's children. After each bit-string is computed, it is stored in $\mathcal{A}$. When the traversals of all k input trees are complete, $\mathcal{A}$ has $O(nk)$ bit-strings.

Filtering unique bit-strings: To remove duplicates from $\mathcal{A}$, we first sort it using *radix sort* [19, Chapter8]. Given N b-bit numbers and any positive integer $r \leq b$, radix sort sorts these numbers in $O((b/r)(N + 2^r))$ time. In our case, $b = n$ and $N = nk$, giving a running time of $O((n/r)(nk + 2^r))$. This quantity is minimized when $r = \log(nk)$, giving a running time of $O(n^2k/\log(nk))$.

After sorting $\mathcal{A}$, we remove its duplicate bit-strings in a single linear scan. This can be done in $O(n^2k)$ time through standard methods.

TAG nodes: We construct a node in the TAG for each unique bit-string in $\mathcal{A}$. For bit-strings corresponding to the leaf nodes, we also associate the appropriate label from S using function g.

3.2 Adding Edges to the TAG

Once the vertices of input trees have been mapped to the vertices of TAG, we add directed edges to the TAG. We traverse each input tree in post-order. When

a tree traversal visits an internal vertex v of an input tree, we find the bit-strings of v and v's children in $\mathcal{A}$ and locate the nodes corresponding to them in the TAG. We then add directed edges from the TAG node corresponding to v to the TAG nodes for v's children.

Theorem 1. *For a given collection of k phylogenetic trees on n labels, the TAG can be built in $O(n^2k)$ time.*

Proof. Collecting bit-strings and then sorting them requires $O(n^2k)$ time. The remaining steps take time linear in the size of the input. □

4 Finding Consensus Trees Using the TAG

Let $\mathcal{P}$ be a collection of k input phylogenetic trees with completely overlapping leaf label set of size n. The *strict consensus tree* for $\mathcal{P}$ is the tree whose clusters are precisely those that appear in all the trees in $\mathcal{P}$. The *majority-rule consensus tree* for $\mathcal{P}$ is the tree whose clusters are precisely those that appear in more than half (i.e., the majority) of the trees in $\mathcal{P}$. Here we show how to build the majority-rule consensus trees for $\mathcal{P}$ from the TAG for $\mathcal{P}$. We then outline the modifications needed to compute the strict consensus tree.

Algorithm MajorityRuleTree (Algorithm 1) builds the majority-rule tree for $\mathcal{P}$ by traversing the TAG D for $\mathcal{P}$. Let $D = (U, E)$ and f be the vertex-labeling function. We assume that each vertex v in D stores the *cardinality* of v — i.e., the number of taxa in $f(v)$ — along with count(v), the number of times cluster $f(v)$ appears in a tree in $\mathcal{P}$. We also assume that multiple edges between the same two vertices are replaced by a single edge. The next observation follows from the fact that the input trees have completely overlapping leaf label sets.

Observation 1. *D has precisely one vertex s with in-degree zero.*

Let v be a node in D. Then, v is a *majority node* if count(v) $> k/2$. The clusters associated with majority nodes are precisely the clusters of the majority-rule tree. Let the nodes of the majority-rule tree correspond to the majority nodes of D. Next we develop an approach to hook up these nodes to actually build the majority-rule tree.

Let u and v be nodes of D. Then, v is a *majority ancestor* of u if v is a majority node, and there is directed path from v to u in D. Algorithm 1 is based on the following observation (parts of which were noted in [20]).

Observation 2. *Let u and v be majority nodes in D. Then,*

(i) if there is a directed path from a majority node u to a majority node v in D, then $f(v) \subset f(u)$, and
(ii) if v is the parent of u in the majority-rule tree for $\mathcal{P}$, then v is the (unique) minimum-cardinality majority ancestor of u; further, $f(u) \subset f(v)$.

Let u be a node in D. The *most recent majority ancestor of* u is the unique minimum-cardinality majority ancestor u. For each vertex $u \in U$, our algorithm maintains two variables: $p(u)$, a reference to the smallest cardinality majority ancestor of u seen thus far, and $m(u)$, the cardinality of $p(u)$. Initially, every node u, except the node s of in-degree zero, has $p(u) = s$, representing initial best estimate of the most recent majority ancestor of u. The algorithm revises this estimate repeatedly until it converges on the correct value. After this process is complete, it is now a simple matter to assemble the majority-rule tree, since, for each majority node u, $p(u)$ points to u's parent in that tree.

Algorithm 1 processes the nodes of D according to topological order — this ordering exists because D is acyclic (from Lemma 1). When the algorithm visits a node u, it examines each successor v, and considers two possibilities. If u is a majority node, then u may become the new value of $p(v)$, while if u is not a majority node, $p(u)$ may become the new value of $p(v)$. By Observation 2, the decision depends solely on node cardinalities.

Input: The TAG $D = (U, E)$ for a collection $\mathcal{P}$ of trees over the same leaf set.
Output: The majority-rule tree for $\mathcal{P}$.

```
1  Let s be the unique vertex in D with in-degree 0
2  foreach u ∈ V − s do
3  |  m(u) = n ; p(u) = s
4  Perform a topological sort of D − s
5  Let M ⊆ U be the set of majority nodes in D
6  foreach u ∈ U − s in topological order do
7  |  if u ∈ M then
8  |  |  μ = |f(u)| ; π = u
9  |  else
10 |  |  μ = m(u) ; π = p(u)
11 |  foreach v ∈ U such that (u, v) ∈ E do
12 |  |  if m(v) > μ then
13 |  |  |  m(v) = μ ; p(v) = π
14 Let T be the tree with vertex set M, where, for every u ∈ M, the parent
   of u in T is p(u)
15 Let φ be the function that maps each leaf u of T to f(u)
16 return (T, φ)
```

Algorithm 1. MajorityRuleTree(D)

Theorem 2. *Given the TAG D for a collection $\mathcal{P}$ of k phylogenetic trees on the same n leaves, the majority-rule consensus tree of $\mathcal{P}$ can be computed in $O(nk)$ time.*

Proof (Sketch). Correctness can be proved using Observation 2. To bound the running time, note that topological sort takes time linear in the size of D, and the main loop (Lines 6–13) examines each node and each edge once. Since the number of edges and nodes in D is $O(nk)$, the claimed time bound follows. □

The algorithm for strict consensus tree is similar to Algorithm 1, with only one significant difference: instead of dealing with majority nodes, it focuses on *strict* nodes, that is, TAG nodes u such that $\text{count}(u) = k$. We omit the details, and simply summarize the result.

Theorem 3. *Given the TAG for a collection $\mathcal{P}$ of k phylogenetic trees on the same n leaves, the strict consensus tree of $\mathcal{P}$ can be computed in $O(nk)$ time.*

5 Testing Compatibility Using the TAG

Let $\mathcal{T}$ and $\mathcal{T}'$ be two phylogenetic trees on X and X', respectively, where $X \subseteq X'$. We say that $\mathcal{T}'$ *displays* $\mathcal{T}$ if, up to suppressing non-root nodes of degree two, the minimum rooted subtree of $\mathcal{T}'$ that connects the elements of X *refines* $\mathcal{T}$, i.e., $\mathcal{T}$ can be obtained from it by contracting internal edges. *Suppressing a node of degree two* means replacing that node and its incident edges by an edge.

Let $\mathcal{P}$ be the input collection of rooted phylogenetic trees. We say that $\mathcal{P}$ is *compatible* if there exists a phylogenetic tree $\mathcal{T}$, called a *compatible supertree for* $\mathcal{T}$, that simultaneously displays every tree in $\mathcal{P}$. A classic result in phylogenetics is that compatibility can be tested in polynomial time [12,13]. In this section, we show that compatibility can be tested directly from the TAG for $\mathcal{P}$.

We need some definitions. As before, we assume that multiple edges between the same two TAG vertices are replaced by a single edge. The *extended TAG* is the graph D^* obtained from D by adding undirected edges between every two vertices $u, v \in U$ such that $f(u)$ and $f(v)$ are clusters corresponding to sibling vertices in some tree in $\mathcal{P}$. D^* is a *mixed* graph, i.e., a graph that contains both directed and undirected edges. See Fig. 3.

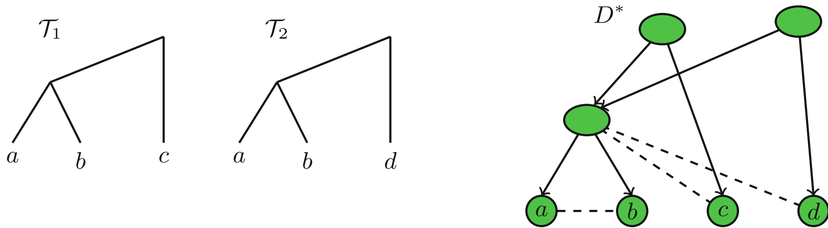


Fig. 3. Two phylogenetic trees $\mathcal{T}_1$ and $\mathcal{T}_2$ with their extended TAG, D^*; undirected edges are shown with dashed lines.

Let $D' = (U', E')$ be a mixed graph. An *arc component* of D' is a maximal sub-mixed graph W of D' such that for every two nodes u and v in W there is a path from u to v which consists only directed edges, irrespective of their directions. Let v be a node of D'. The mixed graph obtained by deleting v and its incident directed and undirected edges is denoted by $D' \setminus v$. Let V be a subset of U'. We write $D' \setminus V$ to denote the (mixed) graph obtained from D' by deleting

each node in V from D'. The *restriction of* D' *to* V, denoted by $D'|V$, is the subgraph of D' obtained by deleting each node in $U' \setminus V$ from D'.

Input: The extended TAG D^* for a collection $\mathcal{P}$ of phylogenetic trees.
Output: A phylogenetic tree $\mathcal{T}$ that displays each tree in $\mathcal{P}$ or the statement *not compatible*.

```
1  Let S_0 be the set of nodes in D* that have in-degree zero and no incident
   edges.
2  if S_0 is empty then
3  |  return not compatible
4  if S_0 contains exactly one node with out-degree zero and label ℓ then
5  |  return the tree composed of singleton node with label ℓ.
6  Find the node sets S_1, S_2, ..., S_m of the arc components of D* \ S_0.
7  Delete all undirected edges of D* \ S_0 whose endpoints are in distinct arc
   components.
8  foreach i ∈ {1, 2, ..., m} do
9  |  Call DESCENDANT(D*|S_i)
10 |  if this call returns not compatible then
11 |  |  return not compatible
12 |  else
13 |  |  Let T_i be the phylogeny returned by this call
14 return a tree with a root node and T_1, T_2, ..., T_m as its subtrees.
```

Algorithm 2. DESCENDANT(D^*)

The extended TAG is closely related to the *restricted descendancy graph* (RDG) [9,10]. The RDG has a unique node for each internal input tree node along with $|S|$ leaf nodes. Let u and v be two input tree nodes, and let u' and v' be the corresponding nodes in the RDG. If u is a parent of v, then there is a directed edge from u' to v' in the RDG. If u and v are siblings, then there is an undirected edge between u' and v' in the RDG. Otherwise, there is no edge between u' and v'.

The extended TAG can be viewed as a compact version of the RDG of $\mathcal{P}$. Thus, a slight adaptation of the DESCENDANT algorithm [9,10] enables us to determine whether $\mathcal{P}$ is compatible given its extended TAG D^*. The details of this adaptation are shown in Algorithm 2. The algorithm first attempts to decompose the problem into subproblems, each of which corresponds to one of the subtrees of the compatible supertree. If no such decomposition exists, then $\mathcal{P}$ is incompatible. Otherwise, the algorithm identifies a collection of subproblems, each associated with a different arc component, and recursively tests compatibility for each subproblem.

Theorem 4. *Let* D^* *be the extended TAG for a collection* $\mathcal{P}$ *of phylogenetic trees. If* $\mathcal{P}$ *is compatible, then* DESCENDANT(D^*) *returns a compatible supertree for* $\mathcal{P}$*; otherwise,* DESCENDANT(D^*) *returns the statement* not compatible.

Proof (Sketch). Let $\mathcal{P}'$ be the collection of phylogenetic trees after labeling the internal nodes of the input trees in $\mathcal{P}$ by their clusters. The order of labels in a cluster does not matter, that is, we assume two labels identical if their respective clusters are identical sets. Now the extended TAG of $\mathcal{P}$ is the same as the RDG of $\mathcal{P}'$. The correctness of Algorithm 2 now follows from the proof of [9, Preposition 4] for $\mathcal{P}'$. We omit the details for lack of space. □

Running time: Following [9, Preposition3]), we can show that if $\mathcal{P}$ consists of k *fully resolved* phylogenetic trees on the leaf set of size n, then the DESCENDANT subroutine runs in time $O(n^2k^2)$. We conjecture that the running time can be reduced to $O(nk \log^2 n)$ using the approach discussed in [9]. If, however, the input trees are not fully resolved, the running time increases by a factor of n.

Remark. As we mention earlier, there are considerable similarities between the extended TAG and the RDG. Nevertheless, the former has some advantages in practice. While every internal node of a tree in $\mathcal{P}$ gives rise to a distinct node in the RDG, the extended TAG has one node for each unique cluster. For instance, in the extreme case when $\mathcal{P}$ contains k identical phylogenetic trees on S, the RDG has $O(nk)$ nodes, while the extended TAG contains only $O(n)$ nodes. More typically, the trees in $\mathcal{P}$ will share many clusters, and the likelihood of this being the case is especially high when k is much larger than n.

6 Discussion

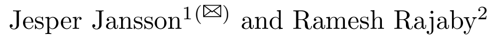
We have presented a formal definition of the TAG that does not depend on the order of the input trees. We have also presented a procedure for building TAGs, and described how to use TAGs to find consensus trees and to determine whether a collection of phylogenetic trees is compatible.

Extending TAGs to include potentially thousands of input trees from across the tree of life leads to several future challenges; two major ones are incorporating trees at different taxonomic levels and finding ways to synthesize conflicting phylogenetic input trees. It may be possible to address the second of these challenges using ideas from the ANCESTRALBUILD algorithm [9,10]. Dealing with conflict among the input trees is also essential for processing large-scale phylogenetic data sets. Although a visual inspection of the TAG provides some insight into the areas of conflict, approaches to quantify phylogenetic conflict within the TAG may provide valuable insight into mechanisms causing phylogenetic incongruence among biological datasets and help guide future phylogenetic research. The synthesis approach of Smith et al. [11] relies on a subjective ranking of the input trees. Potentially, a MINCUTSUPERTREE approach, like the MULTILEVEL-SUPERTREE algorithm [8], could be applied to a TAG to provide an efficient and effective approach for synthesizing a tree of life.

References

1. Baum, D., Smith, S.: Tree Thinking: An Introduction to Phylogenetic, 1st edn. Roberts and Company, Englewood (2012)
2. Goldman, N., Yang, Z.: Introduction. statistical and computational challenges in molecular phylogenetics and evolution. Philos. Trans. Royal Soc. B Biol. Sci. **363**(1512), 3889–3892 (2008)
3. McMahon, M., Deepak, A., Fernández-Baca, D., Boss, D., Sanderson, M.: STBase: one million species trees for comparative biology. PLoS One **10**(2), e0117987 (2015)
4. Faircloth, B.C., McCormack, J.E., Crawford, N.G., Harvey, M.G., Brumfield, R.T., Glenn, T.C.: Ultraconserved elements anchor thousands of genetic markers spanning multiple evolutionary timescales. Syst. Biol. **61**(5), 716–726 (2012)
5. Lemmon, A.R., Emme, S.A., Lemmon, E.M.: Anchored hybrid enrichment for massively high-throughput phylogenomics. Syst. Biol. **61**(5), 727–744 (2012)
6. McCormack, J.E., Faircloth, B.C., Crawford, N.G., Gowaty, P.A., Brumfield, R.T., Glenn, T.C.: Ultraconserved elements are novel phylogenomic markers that resolve placental mammal phylogeny when combined with species-tree analysis. Genome Res. **22**, 746–754 (2012)
7. Smith, S.A., Brown, J.W., Hinchliff, C.E.: Analyzing and synthesizing phylogenies using tree alignment graphs. PLoS Comput. Biol. **9**(9), e1003223 (2013)
8. Berry, V., Bininda-Emonds, O., Semple, C.: Amalgamating source trees with different taxonomic levels. Syst. Biol. **62**(2), 231–249 (2013)
9. Berry, V., Semple, C.: Fast computation of supertrees for compatible phylogenies with nested taxa. Syst. Biol. **55**(2), 270–288 (2006)
10. Daniel, P., Semple, C.: A class of general supertree methods for nested taxa. SIAM J. Discrete Methods **19**, 463–480 (2005)
11. Smith, S.A., Cranston, K.A., Allman, J.F., Brown, J.W., Burleigh, G., Chaudhary, R., Coghill, L., Crandall, K.A., Deng, J., Drew, B., Gazis, R., Gude, K., Hibbett, D.S., Hinchliff, C., Katz, L.A., IV, H.D.L. , McTavish, E.J., Owen, C.L., Ree, R., Rees, J.A., Soltis, D.E., Williams, T.: Synthesis of phylogeny and taxonomy into a comprehensive tree of life. (Under review)
12. Aho, A.V., Sagiv, Y., Szymanski, T.G., Ullman, J.D.: Inferring a tree from lowest common ancestors with an application to the optimization of relational expressions. SIAM J. Comput. **10**(3), 405–421 (1981)
13. Semple, C., Steel, M.: Phylogenetics. Oxford University Press, Oxford (2003)
14. Constantinescu, M., Sankoff, D.: An efficient algorithm for supertrees. J. Classif. **12**, 101–112 (1995)
15. Ng, M.P., Wormald, N.C.: Reconstruction of rooted trees from subtrees. Discrete Appl. Math. **69**(1–2), 19–31 (1996)
16. Semple, C., Steel, M.A.: A supertree method for rooted trees. Discrete Appl. Math. **105**, 147–158 (2000)
17. Page, R.D.M.: Modified Mincut Supertrees. In: Guigó, R., Gusfield, D. (eds.) WABI 2002. LNCS, vol. 2452, pp. 537–551. Springer, Heidelberg (2002)
18. Diestel, R.: Graph Theory. Springer, Heidelberg (2000)
19. Cormen, T.H., Leiserson, C.E., Rivest, R.E.: Introduction to Algorithms. MIT Press, Cambridge (1996)
20. Amenta, N., Clarke, F., St. John, K.: A linear-time majority tree algorithm. In: Benson, G., Page, R.D.M. (eds.) WABI 2003. LNCS (LNBI), vol. 2812, pp. 216–227. Springer, Heidelberg (2003)

A More Practical Algorithm for the Rooted Triplet Distance



Jesper Jansson[1(✉)] and Ramesh Rajaby[2]

[1] Laboratory of Mathematical Bioinformatics, Institute for Chemical Research, Kyoto University, Gokasho, Uji, Kyoto 611-0011, Japan
jj@kuicr.kyoto-u.ac.jp

[2] University of Milano-Bicocca, Milano, Italy
r.rajaby@campus.unimib.it, ramesh.rajaby@gmail.com

Abstract. The *rooted triplet distance* is a measure of the dissimilarity of two phylogenetic trees with identical leaf label sets. An algorithm by Brodal *et al.* [2] that computes it in $O(n \log n)$ time, where n is the number of leaf labels, has recently been implemented in the software package tqDist [14]. In this paper, we show that replacing the hierarchical decomposition tree used in Brodal *et al.*'s algorithm by a centroid paths-based data structure yields an $O(n \log^3 n)$-time algorithm that, although slower in theory, is easier to implement and apparently faster in practice. Simulations for values of n up to $1,000,000$ support our claims experimentally.

Keywords: Bioinformatics · Phylogenetic tree comparison · Rooted triplet distance · Centroid path decomposition tree

1 Introduction

Over the years, many alternative methods for inferring phylogenetic trees have been developed; see, e.g., [7]. Due to errors in experimentally obtained data or the inherent instability of classifications, applying the same tree inference method to different datasets, applying different tree inference methods to the same dataset, or changing the assumed model of evolution may result in trees with different branching patterns. In this case, in order to identify parts of the trees that look alike or to reconcile all the trees into a single tree, methods for measuring the similarity between phylogenetic trees are needed. Measuring the similarity between two phylogenetic trees may also be useful for supporting queries in phylogenetic databases in the future [1] or for evaluating the performance of a newly proposed tree inference method by doing simulations and comparing the inferred trees to the corresponding known correct trees.

Several measures of the (dis-)similarity of two phylogenetic trees with identical leaf label sets have been suggested in the literature (see [1]). One such measure is the *rooted triplet distance* [5], which counts how many of the subtrees

J. Jansson—Funded by The Hakubi Project and KAKENHI grant number 26330014.
R. Rajaby—Funded by the EXTRA Project at the University of Milano-Bicocca.

© Springer International Publishing Switzerland 2015

A.-H. Dediu et al. (Eds.): AlCoB 2015, LNBI 9199, pp. 109–125, 2015.
DOI: 10.1007/978-3-319-21233-3_9

induced by cardinality-3 subsets of the leaves that differ between the two trees. Intuitively, this measure considers two phylogenetic trees that share many small embedded subtrees to be similar. This paper presents a practical algorithm for computing the rooted triplet distance, based on the framework introduced in an algorithm by Brodal *et al.* [2], along with its implementation.

1.1 Basic Definitions

In this paper, a *phylogenetic tree* is a rooted, unordered tree whose leaves are distinctly labeled and whose internal nodes have degree at least 2. From here on, phylogenetic trees are referred to as "trees" for short. The set of all nodes and the set of all leaf labels in a tree T are denoted by $V(T)$ and $\Lambda(T)$, respectively. For any $x \in V(T)$, $T(x)$ is the subtree of T rooted at x, i.e., the subgraph of T induced by the node x and all of its proper descendants in T. For any $x, y \in V(T)$, $lca^T(x, y)$ is the lowest common ancestor in T of x and y. Also, for any $x, y \in V(T)$, if x is a proper descendant of y then we write $x \prec y$.

A *rooted triplet* is a tree with exactly three leaves. Suppose t is a rooted triplet with leaf label set $\Lambda(t) = \{a, b, c\}$. There are two possibilities. If t has a single internal node then t is called a *fan triplet* and is denoted by $a|b|c$. Observe that in this case, t is a non-binary tree and $lca^t(a, b) = lca^t(a, c) = lca^t(b, c)$ holds. Otherwise, t has two internal nodes and is a binary tree; in this case, t is called a *resolved triplet* and is denoted by $xy|z$, where $\{x, y, z\} = \{a, b, c\}$ and $lca^t(x, y) \prec lca^t(x, z) = lca^t(y, z)$.

For any tree T and $\{a, b, c\} \subseteq \Lambda(T)$, the fan triplet $a|b|c$ is said to be *consistent with* T if $lca^T(a, b) = lca^T(a, c) = lca^T(b, c)$. Similarly, the resolved triplet $ab|c$ is *consistent with* T if $lca^T(a, b) \prec lca^T(a, c) = lca^T(b, c)$. Let $rt(T)$ be the set of all rooted triplets consistent with the tree T. (Thus, $|rt(T)| = \binom{|\Lambda(T)|}{3}$).

For any two trees T_1, T_2 with $\Lambda(T_1) = \Lambda(T_2)$, the *rooted triplet distance* $d_{rt}(T_1, T_2)$ is defined as $|rt(T_1) \triangle rt(T_2)|$, i.e., the number of rooted triplets that are consistent with one of the two trees but not the other. Note that dividing $d_{rt}(T_1, T_2)$ by $\binom{n}{3}$, where $n = |\Lambda(T_1)| = |\Lambda(T_2)|$, yields a dissimilarity coefficient between 0 and 1 that may be more informative than d_{rt} in some applications.

Below, we consider the problem of computing $d_{rt}(T_1, T_2)$ for two input trees T_1, T_2 with identical leaf label sets. To simplify the notation, write $L = \Lambda(T_1)$ $(= \Lambda(T_2))$ and $n = |L|$ for the given trees.

1.2 Previous Results and Related Work

The rooted triplet distance was proposed by Dobson [5] in 1975. Given two trees T_1, T_2 with identical leaf label sets, $d_{rt}(T_1, T_2)$ can be computed in $O(n^3)$ time by a straightforward algorithm. Critchlow *et al.* [4] gave an $O(n^2)$-time algorithm for the special case where T_1 and T_2 are *binary*, and Bansal *et al.* [1] showed how to compute $d_{rt}(T_1, T_2)$ in $O(n^2)$ time for two trees of *arbitrary* degrees. Recently, Brodal *et al.* [2] achieved a time complexity of $O(n \log n)$ for two trees

of arbitrary degrees. An implementation of the latter algorithm, written in C++, is available in the free software package tqDist [14].

The counterpart of the rooted triplet distance for *unrooted* trees is the *unrooted quartet distance* [6]. The currently fastest algorithm for computing the unrooted quartet distance [2] runs in $O(dn \log n)$ time, where n is the number of leaf labels and d is the maximum degree of any node in the two input trees.

An extension of the rooted triplet distance to *phylogenetic networks* has been studied in [11]. For two *galled trees* [9] (networks whose cycles are disjoint) with n leaves each, the rooted triplet distance can be computed in $o(n^{2.687})$ time [11].

1.3 Our Contributions

We present some non-trivial modifications to Brodal *et al.*'s algorithm [2] for computing $d_{rt}(T_1, T_2)$ for two trees of arbitrary degrees that make it easier to implement and more efficient in practice. The theoretical time complexity of the resulting algorithm is $O(n \log^3 n)$, which is slightly worse than that of the original version, but we show experimentally that a direct C++-implementation of the new algorithm gives a faster and more memory-efficient method than tqDist [14] (the publicly available implementation of Brodal *et al.*'s algorithm) for various types of large inputs consisting of two trees with up to $1,000,000$ leaves each.

The paper is organized as follows. Brodal *et al.*'s algorithm [2] is reviewed in Sect. 2. Section 3 describes the new algorithm, Sect. 4 discusses some implementation issues, and Sect. 5 presents the experimental results. Finally, Sect. 6 gives some concluding remarks.

2 Summary of Brodal *et al*'s Algorithm [2]

On a high level, the algorithm of Brodal *et al.* [2] works as follows. Each rooted triplet t in $rt(T_1)$ is implicitly assigned to the lowest common ancestor in T_1 of the three leaves in $\Lambda(t)$. For each internal node u in T_1, the algorithm counts how many of its assigned rooted triplets that also appear in $rt(T_2)$ by first coloring the leaves of T_2 in such a way that two leaves receive the same color if and only if they are descendants of the same child of u in T_1, and then finding the number of elements in $rt(T_2)$ compatible with this particular coloring by making a query to a special data structure called a *hierarchical decomposition tree* (HDT) that represents T_2.

To avoid unnecessary leaf recolorings, a simple, recursive recoloring scheme is used that visits all nodes of T_1 in order and generates the corresponding leaf colorings in T_2. It is reviewed in Sect. 3.2 (a) below. Constructing the HDT, augmenting it with auxiliary information to support the relevant queries, and updating this information when the leaves of T_2 are recolored are somewhat complicated; see [2] for details.

As proved in [2], the algorithm's time complexity is $O(n \log n)$.

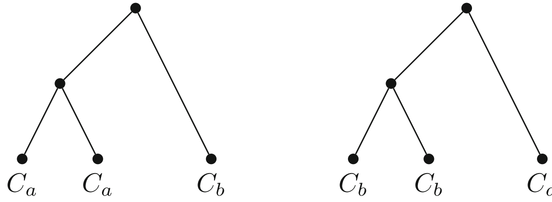


Fig. 1. Topologies induced by good triplets in T_2.

3 The New Algorithm

The new algorithm, described below, uses the same framework as Brodal *et al.*'s algorithm [2]. To be precise, we also implicitly assign each rooted triplet in $rt(T_1)$ to an internal node in T_1 and count, for each node in T_1, how many of its rooted triplets that appear in $rt(T_2)$. To handle all of T_1's nodes efficiently, we apply Brodal *et al.*'s recoloring scheme with a minor modification. The main difference between the old algorithm and the new algorithm is how the rooted triplets in $rt(T_2)$ assigned to each node in $V(T_1)$ are counted. Whereas Brodal *et al.*'s algorithm uses the (in our opinion) cumbersome HDT data structure, we use the conceptually simpler centroid path decomposition technique [3]. This makes the new algorithm a little slower in theory but easier to implement and faster in practice.

3.1 Preliminaries

For convenience, we make T_1 and T_2 into *ordered* trees by imposing the following left-to-right ordering: for every non-leaf node v in a tree, the leftmost child of v is always a child of v having the most leaf descendants (with ties broken arbitrarily), and the other children of v are ordered arbitrarily. Also, let a *triplet* be any subset of L of cardinality three; each triplet x induces a rooted triplet in T_1 (or T_2), namely the rooted triplet belonging to $rt(T_1)$ (or $rt(T_2)$) whose leaf label set equals x.

We first introduce some notation related to the leaf colorings induced by the internal nodes of T_1. Let d be the number of children of the highest degree node in T_1. Define a set of $d+1$ *colors* $\{C_0, C_1, .., C_d\}$. We usually refer to C_1 as *RED* and C_0 as *WHITE*, and sometimes call a WHITE leaf *non-colored* and a leaf that is neither RED nor WHITE *NON-RED*. Colors will be assigned to the leaf label set L, and we say that we are coloring a leaf when we are coloring its label.

Let $\{v_1, v_2, \dots\}$ be the children of an internal node $v \in V(T_1)$. Then T_1 and T_2 are *colored according to* v if and only if, for every $l \in L$, it holds that:

- l is colored by C_i if and only if $T_1(l)$ is a descendant of v_i; and
- l is WHITE if and only if $T_1(l)$ is not a descendant of v.

Suppose that T_1 and T_2 are colored according to some internal node $v \in T_1$. Let t be a triplet. We call t a *good triplet* if it induces one of the two (unordered)

topologies shown in Fig. 1 in T_2, where C_a and C_b are colors in $\{C_1, C_2, .., C_d\}$ and $a < b$. Similarly, we call t a *good fan* if its three leaves all have different colors from the set $\{C_1, C_2, .., C_d\}$. (This corresponds to the concept of a triplet being "compatible with a coloring" in [2].)

For a given color C_c, $C_c(S)$ is the number of leaves in S colored by C_c, where S can be either a single subtree or a set of subtrees (which is generally clear from the context). $C_1(S)$ will usually be referred to as $Red(S)$. Also define $C_{\bar{a}}(S) = \sum_{i=2, i \neq a}^{d} C_i(S)$ as the number of NON-RED leaves in S which are not colored by C_a, and $C_{R\bar{a}}(S) = \sum_{i=1, i \neq a}^{d} C_i(S)$ as the number of non-WHITE leaves in S which are not colored by C_a. Finally, define $Red^{(2)}(S) = \sum_{i=1}^{k} \binom{Red(S_i)}{2}$, where $\{S_1..S_k\}$ are the subtrees rooted at the root of S. We will sometimes use a node as an argument, meaning the subtree rooted at it.

Next, we recall the concept of a *centroid path* [3]. A *centroid path* starting at some node v in a tree T is a heaviest path from v to a leaf of T, i.e., a path starting at v and always choosing a child with the largest number of leaf descendants until a leaf is reached. Let *the centroid path of* T, $cp(T)$, be a centroid path starting at the root of T. In our case, the centroid path of T starts from the root and always selects the left subtree until it reaches a leaf.

The *centroid path decomposition tree* of T, denoted by $CPDT(T)$, is an ordered tree of unbounded degree defined as follows. One node u represents $cp(T)$; u is the root of $CPDT(T)$. We traverse $cp(T)$ from its lowest node to its highest; for each node r_j in T that we encounter, we add a single node v_j to the ordered set of children of u (making v_j the rightmost child so far), and then define the children of v_j as $\{CPDT(T_2^j), ..., CPDT(T_k^j)\}$, where k is the degree of r_j and T_i^j is the i-th subtree of r_j (remember that the root of T_1^j lies on the centroid path). We call u a *CP-node* (*centroid path node*), since it represents a whole centroid path in T, and v_j a *SN-node* (*single-node node*), since it represents a single node in T. If r_j is binary then v_j will have a single child.

Note that $CPDT(T)$ has height $O(\log n)$, where $n = |\Lambda(T)|$. Moreover, we immediately have:

Lemma 1. *An SN-node is always the child of a CP-node, and the only CP-node which is not the child of an SN-node is the root.*

3.2 Description of the New Algorithm

First, the algorithm constructs $CPDT(T_2)$. Then, for each internal node v of T_1 in depth-first order, the algorithm: (a) colors the trees according to v, and (b) counts the resulting number of good triplets and good fans by using $CPDT(T_2)$. Finally, it returns the value $\binom{n}{3}$ minus the total number of good triplets and good fans found.

(a) Colorings: To obtain all the colorings of the trees efficiently, the algorithm uses a slightly modified variant of Brodal *et al.*'s recursive recoloring scheme from [2]. The latter does a depth-first traversal of T_1 while maintaining two invariants:

(i) when entering a node v, all leaves in $T_1(v)$ are RED and all other leaves are WHITE;
(ii) when exiting a node v, all leaves in $T_1(v)$ are WHITE.

Initially, all leaves are colored RED. This way, invariant i) holds when the transversal starts at the root of T_1.

During the transversal, whenever an internal node $v \in V(T_1)$ is reached, the leaves in the i-th subtree of v are colored by the color C_i for $i \in [2..k]$, where k is the degree of v. At this point, the trees are colored according to v. After this, the $k-1$ subtrees that were just colored are recolored by the color WHITE, and the scheme recurses on the leftmost subtree of v, which is still RED. Observe that the first invariant holds when entering the root of this subtree. After returning from the recursive call, the leftmost subtree of v is WHITE by invariant (ii), and the other subtrees of v are treated one by one; for each such subtree, the scheme colors its leaves RED and then recurses on it. Again, invariant (i) holds at each recursive call since the subtree is colored RED and everything else is WHITE. After handling all k subtrees of v, all leaf descendants of v will be WHITE, so invariant (ii) holds when exiting from v.

The base case of the recursion is when the reached node $v \in V(T_1)$ is a leaf. In this case, the scheme colors v WHITE and exits, so that invariant (ii) holds.

Our algorithm makes the following modification to Brodal *et al.*'s recursive recoloring scheme above: We color the leaves by colors $\{C_2..C_k\}$, not by their order in T_1, but according to their left-to-right order in $CPDT(T_2)$. (Recall that by definition, the CPDT is an ordered tree).

(b) Counting Good Triplets and Good Fans: By definition, good triplets and good fans are created *only* when leaves are colored by NON-RED colors. Therefore, we let the algorithm compute the number of newly created good triplets and good fans whenever the recursive coloring scheme colors a leaf l by a NON-RED color. To do this, the algorithm traverses the leaf-to-root path starting at $CPDT(l)$, and for each node v on the path, it counts the number of good triplets and good fans that include l and whose lca in the CPDT equals v by applying Lemmas 2 and 3 below.

From here on, we denote by C_b a color different from C_a in the set $\{C_1, .., C_k\}$, but in the formulas, RED and NON-RED will usually be given separate cases. (This is the reason why sums over colors in formulas sometimes start from 2.)

Lemma 2. *Given an internal CP-node u of the CPDT and some child u_i of u, let S_i be the subtree rooted at u_i. Also, let u'_j be the j-th child of u_i and S'_j the subtree rooted at it. If there are no NON-RED leaves in $S_{>i}$ nor in $S'_{>j}$, the*

number of good triplets introduced by coloring a leaf by a color C_a *in* S'_j, $a \geq 2$, *such that their lca is* u, *is:*

$$\binom{Red(S_{<i})}{2} + \sum_{b=2,b\neq a}^{d} \binom{C_b(S_{<i})}{2} + \sum_{h>i} Red^{(2)}(S_h) + C_a(S_{\leq i}) \cdot Red(S_{>i}) + C_a(S'_j) \cdot Red(S_{<i}) + C_a(S'_j) \cdot C_{\bar{a}}(S_{<i})$$

while the number of good fans, whose lca is u, *is:*

$$C_{R\bar{a}}(S_{<i}) \cdot C_{R\bar{a}}(S'_{<j}) - \sum_{b=1,b\neq a}^{d} C_b(S_{<i}) \cdot C_b(S'_{<j}) + C_{\bar{a}}(S_{<i}) \cdot RED(S'_{>j})$$

Lemma 3. *Given an SN-node* v *of the CPDT and some child* v_i *of* v, *let* S_i *be the subtree rooted at* v_i. *If there are no NON-RED leaves in* $S_{>i}$, *the number of good triplets introduced by coloring a leaf by a color* C_a *in* S_i, $a \geq 2$, *such that their lca is* v, *is:*

$$\sum_{j=1}^{i-1} \binom{Red(S_j)}{2} + \sum_{j>i} \binom{Red(S_j)}{2} + \sum_{b=2,b\neq a}^{d} \sum_{j=1}^{i-1} \binom{C_b(S_j)}{2} + C_a(S_i) \cdot (Red(S_{<i}) + Red(S_{>i})) + C_a(S_i) \cdot C_{\overline{a}}(S_{<i})$$

while the number of good fans, whose lca is v, *is:*

$$\binom{C_{R\bar{a}}(S_{<i})}{2} - \sum_{b=1,b\neq a}^{d} \binom{C_b(S_{<i})}{2} - \sum_{h=1}^{i-1} \binom{C_{R\bar{a}}(S_h)}{2} + \sum_{h=1}^{i-1} \sum_{b=1,b\neq a}^{d} \binom{C_b(S_h)}{2} + C_{\bar{a}}(S_{<i}) \cdot RED(S_{>i})$$

3.3 Time Complexity Analysis

The values in Lemmas 2 and 3 for any specified node in the CPDT can be obtained by a direct method in $O(n)$ time. This will be too slow for our purposes, so we first reduce it to $O(\log n)$ time (Lemmas 5 and 6). The solution uses the *range sum query data structure* (RSQ), a data structure for representing an array of non-negative integers $A[1..n]$ so that it is possible to:

1. given an index $i \in [1..n]$, change the value of $A[i]$;
2. given two positions $s, t \in [1..n]$, where $s \leq t$, return the value $\sum_{i=s}^{t} A[i]$.

Given an RSQ R, we refer to the array of numbers over which R supports queries as $R.A$.

We shall rely on the following result from the literature:

Lemma 4. *An RSQ supporting operations 1 and 2 in* $O(\log n)$ *time can be implemented in* $O(n)$ *space and* $O(n)$ *preprocessing time using a* Fenwick tree [8].

Now, for each node v in the CPDT, define and store the following set of counters, where $\{v_1, v_2, .., v_k\}$ denotes the set of children of v:

$$\begin{cases} C_c(v),\ \forall c \in 2..d, \text{as defined in Sect. 3.1} \\ C(v) = \sum_{c=2}^{d} C_c(v) \\ C_c^2(v) = \binom{C_c(v)}{2},\ \forall c \in 2..d \\ C^2(v) = \sum_{c=2}^{d} C_c^2(v) \\ C_c^{(2)}(v) = \sum_{i=1}^{k} \binom{C_c(v_i)}{2},\ \forall c \in 2..d \\ C^{(2)}(v) = \sum_{c=2}^{d} C_c^{(2)}(v) \\ SS(v) = \sum_{i=1}^{k} \binom{\sum_{b=2}^{d} C_b(v_i)}{2} + \sum_{b=2}^{d} C_b(v_i) \cdot Red(v_i) \\ SS_c(v) = \sum_{i=1}^{k} C_c(v_i) \cdot (C_{\overline{c}}(v_i) + Red(v_i)) + \binom{C_c(v_i)}{2},\ \forall c \in 2..d \end{cases}$$

Also store three RSQs, named $R_1(v)$, $R_2(v)$, and $R_3(v)$, such that $R_1(v).A[i] = Red(v_i)$, $R_2(v).A[i] = \binom{Red(v_i)}{2}$, and $R_3(v).A[i] = Red^{(2)}(v_i)$.

Lemma 5. *The values in Lemma 2 can be found in $O(\log n)$ time.*

Proof. (Refer to the notation introduced back in Lemma 2). By the definition of the coloring scheme, no NON-RED leaf is in $S_{>i}$. Therefore, $C_c(u) = C_c(S_{\leq i})$. Since $C_c(u_i) = C_c(S_i)$, it is straightforward to compute $C_c(S_{<i})$. A similar argument works for every counter: starting from its values for u and u_i, we can compute its value for $S_{<i}$ easily.

Next, when coloring leaves in S'_j, all leaves in $S_{<i}$ have already been colored. Thus, the values $C_c(S_{<i})$, $\forall c \in 2..d$, are fixed. We keep track of the current value of $C_c(S_{<i}) \cdot C_c(S'_{<j})$, $\forall c \in 2..d$, in S'_j as we color leaves in it, plus the value $\sum_{b=2}^{d} C_b(S_{<i}) \cdot C_b(S'_{<j})$. Then: $\sum_{b=1, b\neq a}^{d} C_b(S_{<i}) \cdot C_b(S'_{<j}) = \sum_{b=2}^{d} C_b(S_{<i}) \cdot C_b(S'_{<j}) - C_a(S_{<i}) \cdot C_a(S'_{<j}) + Red(S_{<i}) \cdot Red(S'_{<j})$.

The other quantities can be deduced using the counters and the RSQs defined above. When making range queries on them, we apply Lemma 4, which gives a time complexity of $O(\log n)$. □

Lemma 6. *The values in Lemma 3 can be found in $O(\log n)$ time.*

Proof. The only non-trivial quantity here is $\sum_{h=1}^{i-1} \binom{\sum_{b=1, b\neq a}^{d} C_b(S_h)}{2} = SS(S_{<i}) + \sum_{h=1}^{i-1} \binom{Red(S_h)}{2} - SS_a(S_{<i})$, which we explain now. We need to compute the number of pairs of colored leaves such that both leaves are in the same subtree S_h, $h < i$, and none of the leaves is colored by C_a. $SS(S_{<i})$ is the number of pairs of colored leaves such that both leaves are in the same subtree S_h, $h < i$, but the two leaves are not both colored RED. Adding $\sum_{h=1}^{i-1} \binom{Red(S_h)}{2}$, we remove the

latter restriction, thus getting the number of pairs of colored leaves such that both leaves are in the same subtree S_h, $h < i$. Finally, we remove $SS_a(S_{<i})$, which is the number of pairs of colored leaves in the same subtree such that at least one leaf is colored by C_a.

As in the proof of Lemma 5, the other quantities can be obtained in $O(\log n)$ time using Lemma 4 and the counters and RSQs above. □

During the execution of the algorithm, the number of good triplets and good fans created when coloring a leaf NON-RED can be obtained by applying Lemmas 2 and 3 to the leaf and all its ancestors in the CPDT; Lemmas 5 and 6 provide these values for any specified node of the CPDT in $O(\log n)$ time.

To ensure that Lemmas 5 and 6 can still be applied after leaves are recolored, the counters and RSQs for certain nodes need to be updated. More precisely, we extend the algorithm so that:

- Whenever a leaf is colored RED or WHITE, we traverse its leaf-to-root path in the CPDT and update every RSQ on it, taking $O(\log n)$ time per node by Lemma 4.
- Whenever a leaf is colored NON-RED, we traverse its leaf-to-root path in the CPDT and, after applying Lemmas 5 and 6 to each node on the path, we update its counters in $O(1)$ time.

In summary, each node in the CPDT that is visited after a leaf recoloring can be taken care of in $O(\log n)$ time. This gives:

Theorem 1. *The time complexity of the new algorithm is $O(n \log^3 n)$.*

Proof. Constructing $CPDT(T_2)$ in the first step takes $O(n)$ time. The construction follows directly from the definition of $CPDT$.

By Brodal *et al.*'s analysis in [2], a total of $O(n \log n)$ leaf colorings occur. Whenever a leaf is colored, we visit all nodes on its leaf-to-root path in the CPDT; its length is $O(\log n)$, leading to a total of $O(n \log^2 n)$ node visits in the CPDT. (Observe that although some nodes such as the root may be visited $\Omega(n \log n)$ times, the total number of node visits is bounded by $O(n \log^2 n)$.)

By the comments after Lemma 6, $O(\log n)$ time is used for each node visit in the CPDT. Hence, the total running time of the algorithm is $O(n \log^3 n)$. □

4 Implementation

We have implemented the new algorithm for the rooted triplet distance in two versions: a special binary trees-only optimized version, and one for general trees. The importance of the special case where both trees are binary justifies a dedicated implementation. Only plain standard C++ was used, expect for an (optional) single feature from C++11, mentioned below. The source code can be downloaded from:
http://sunflower.kuicr.kyoto-u.ac.jp/~jj/Software/CPDT-dist.html

A few points and optimizations that improve the implementation's running time in practice are discussed next.

4.1 Representation of the Counters

The first issue is how to efficiently represent the counters defined in Sect. 3.3. Let d be the number of children of the highest degree node in T_1: as noted in [10], if we naively represent the counters using d-long vectors for each node in the CPDT, we may end up with quadratic memory (and thus quadratic time) when d is close to n, e.g., if all the leaves are directly attached to the root of T_1. However, we do *not* actually need d counters in each node. At any time, the number of colors used in any subtree of the CPDT cannot be larger than the number of its leaves; therefore, $O(n \log n)$ counters are needed in total.

We implemented the counters as follows. In any given node v in the CPDT, for each set of counters, we allocate an array of length $min\{d, leaves(v)\}$, where $leaves(v) = |L(CPDT(v))|$ is the number of leaves in the subtree of the CPDT rooted at v. Note that when the length of the arrays is $<d$, counters for color $k \in 1..d$ may not be stored in position k in the arrays, so we use a map int-to-int to maintain an association between the (used) colors in $1..d$ and their actual position in the arrays.

We used a hashmap to implement the maps, which allows constant-time insertion and retrieval. We tested two different implementation of the hashmap: the C++11 unordered_map [16] included in the standard library of our test system, requiring C++11 support, and the dense_hash_map class in the (former) Google project sparsehash [15]. We can choose which one to use at compile-time. Some experimental results for both libraries are reported in Sect. 5.

4.2 Two-Step Coloring

In the theoretical version of the algorithm, for simplicity, when coloring leaves by NON-RED colors in left-to-right order in the CPDT, we take each leaf in order and count the good triplets rooted at each of its ancestors up to the root; it is evident that some nodes are considered many times.

In the implementation, we do it slightly differently: First, we mark all of the nodes in the CPDT we need to consider, i.e., all nodes that are ancestors of at least one leaf being colored: for each leaf, we start at it and go up until we either end at the root or at an already marked node, marking all the nodes we traverse. Second, we visit the marked subtree in post-order; when we actually consider a given node, we already know how many leaves we are coloring for each color; modifying the formulas in Lemmas 2 and 3 to count all of the good triplets introduced by such leaves, for each color, in one go is trivial.

4.3 The Coloring Scheme

A few optimizations were made to the coloring scheme.

First, consider what happens when the coloring scheme begins. We start by coloring all the leaves RED, only to immediately recolor leaves not in the biggest subtree of the root, first WHITE and then by NON-RED colors. This happens many times, i.e., every time we color a subtree RED and immediately recurse

on it. We save some unnecessary operations by only coloring RED leaves in the biggest subtree. Since coloring a leaf RED and WHITE requires updating a number of RSQs and is a fairly expensive operation, this saves us some time.

Next, cherry nodes (i.e., internal nodes with exactly two leaves attached) do not need to be colored, since they can not yield any good triplet. If we do not consider them, we can save a lot of RED and WHITE colorings. Cherry nodes are fairly frequent, especially in binary/low branching trees.

Finally, when we color a single leaf (or even a sufficiently low number of leaves), the two-step coloring can actually be a burden. Thus, we made a few special routines that directly update the leaf-to-root path of a given leaf, without the need of marking it first.

5 Experiments

We compared the running time and memory usage in practice of the new algorithm to that of tqDist [14] by a series of experiments, as described in this section.

5.1 Experimental Setup

The experiments were performed on a computer running Ubuntu 12.04, with an Intel Xeon W3530 (quad-core, 2.8 GHz) and 16 GB of RAM. The system C++ compiler was `g++`, version 4.6.3.

We used the C++-implementations of our algorithm presented in Sect. 4: one for general trees and a special binary trees-only optimized version. As mentioned in Sect. 4, the algorithm can be compiled using two different hashmaps: C++11 unordered_map [16] (we named this CPDT) and sparsehash [15] (named CPDTg). We will improperly refer to CPDT and CPDTg as "implementations" of our algorithm, but they are actually a single implementation linked against two different hashmap libraries. tqDist [14] was built from its source code using `cmake`, as instructed by the authors. We had to disable the HDT dynamic contraction [10] of tqDist as it was making the tool run tens of times slower; built with the default parameters, it would usually take more than an hour for two random 1 million leaves trees.

Running times were measured using the `time` command, which gives the sum of system and user times and includes the time spent parsing the trees from a file; the average over 50 runs was taken. Memory usage was measured with Valgrind [13] and its heap profiling tool Massif. Due to the slowdown caused by Valgrind, we took the average over 20 runs.

5.2 Input Trees

Our implementations and tqDist were applied to pairs of trees with values of n up to $1,000,000$. Arbitrary-degree input trees were generated as follows:

> First, generate a binary tree with n leaves in the uniform model [12]. Then, for each non-root, internal node v in the tree, contract it (i.e., make the children of v become children of v's parent, and remove v) with some fixed probability p.

Below, we let p_i for $i \in \{1, 2\}$ be the chosen value of p when generating T_i. We used three values of p_1 and p_2: 0.2, 0.5, and 0.8, calling the generated trees *lowly-branching*, *moderately-branching*, and *highly-branching*, respectively. This gave 9 sets of benchmarks. In addition, we created a set where both trees are binary (equivalent to the case $p_1 = p_2 = 0$) and two extra sets where p_1 is 0.95 (resp. 0.2) and p_2 is 0.2 (resp. 0.95) in order to test the behavior of the algorithms when dealing with *extremely-branching* trees, i.e., flat trees with very high degree. Also, all sets of benchmarks were executed on pairs of *unrelated* as well as *related* trees, where unrelated trees were generated independently of each other and related trees were generated from the same binary tree.

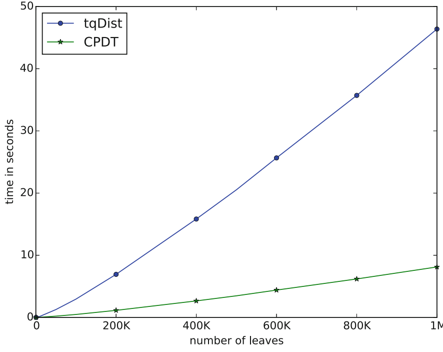
5.3 Results

Table 1 reports the average running times of the three implementations tested on pairs of trees with 1 million leaves, along with the relative speed-ups over tqDist. Figure 2 plots the average running times as a function of n for binary trees as well as for non-binary trees obtained using some representative (p_1, p_2)-values.

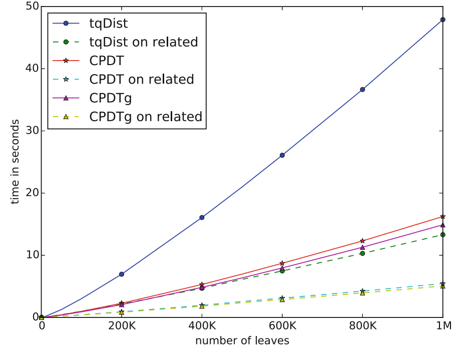
The binary case benefits greatly from having a special implementation, being faster than the more general implementation for arbitrary-degree trees, and

Table 1. Average running times, in seconds, on two 1M leaves trees, and relative speed-ups over tqDist.

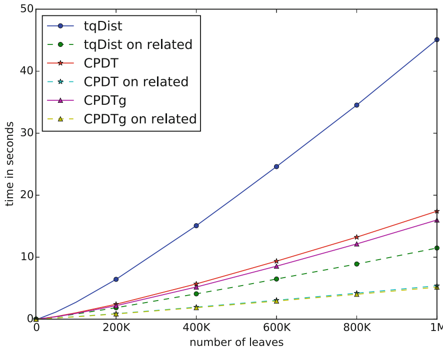
		Unrelated trees						Related trees					
p_1	p_2	tqDist		CPDT		CPDTg		tqDist		CPDT		CPDTg	
0.0	0.0	46.38	1.00x	8.12	5.71x	-	-	-	-	-	-	-	-
0.2	0.2	47.90	1.00x	16.23	2.95x	14.90	3.21x	13.31	1.00x	5.45	2.44x	5.06	2.63x
0.2	0.5	46.71	1.00x	17.89	2.61x	16.33	2.86x	12.04	1.00x	5.57	2.16x	5.08	2.37x
0.2	0.8	40.26	1.00x	15.67	2.57x	14.19	2.84x	10.29	1.00x	4.76	2.16x	4.28	2.40x
0.2	0.95	30.87	1.00x	11.26	2.74x	10.21	3.02x	9.09	1.00x	4.00	2.27x	3.59	2.53x
0.5	0.2	46.25	1.00x	15.78	2.93x	14.58	3.17x	12.75	1.00x	5.28	2.41x	5.11	2.50x
0.5	0.5	45.09	1.00x	17.42	2.59x	16.00	2.82x	11.48	1.00x	5.40	2.13x	5.17	2.22x
0.5	0.8	38.80	1.00x	15.18	2.56x	13.87	2.80x	9.79	1.00x	4.57	2.14x	4.35	2.25x
0.8	0.2	43.18	1.00x	14.67	2.94x	13.62	3.17x	11.91	1.00x	4.95	2.41x	4.70	2.53x
0.8	0.5	42.21	1.00x	16.36	2.58x	15.12	2.79x	10.61	1.00x	5.02	2.11x	4.70	2.26x
0.8	0.8	35.88	1.00x	14.08	2.55x	12.95	2.77x	8.96	1.00x	4.20	2.13x	3.86	2.32x
0.95	0.2	37.14	1.00x	12.72	2.92x	11.75	3.16x	11.15	1.00x	4.79	2.33x	4.19	2.66x



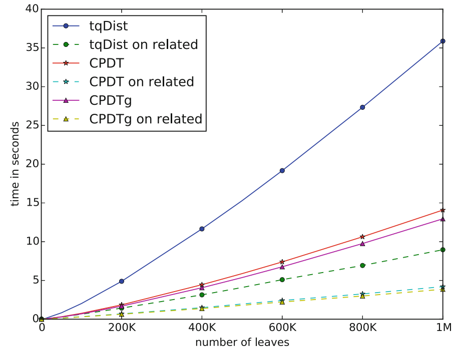
(a) Binary trees



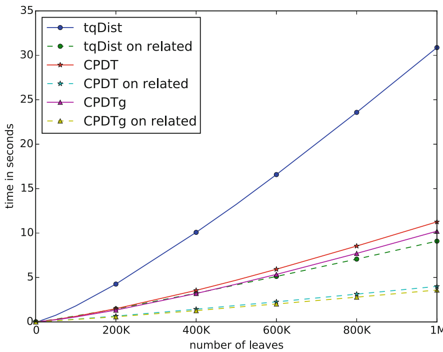
(b) $p_1 = 0.2$ $p_2 = 0.2$



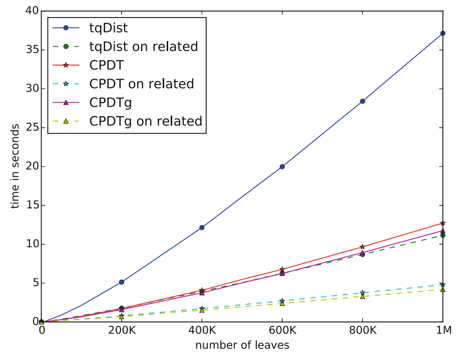
(c) $p_1 = 0.5$ $p_2 = 0.5$



(d) $p_1 = 0.8$ $p_2 = 0.8$

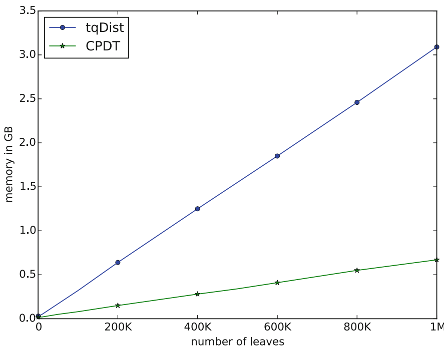


(e) $p_1 = 0.2$ $p_2 = 0.95$

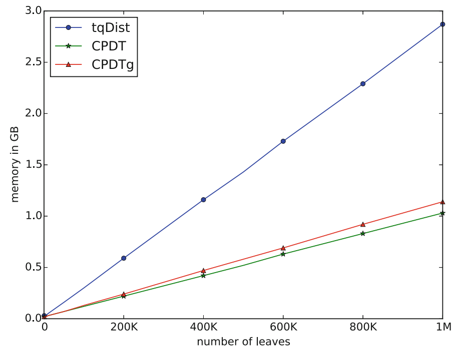


(f) $p_1 = 0.95$ $p_2 = 0.2$

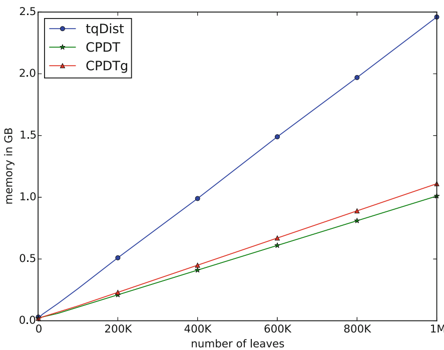
Fig. 2. Plots of the average running time in seconds (y-axis) against n (x-axis), for binary trees and for some representative values of (p_1, p_2). Solid lines represent values on unrelated trees, while dashed lines represent values on related trees; the notion of related trees is not applicable to binary trees.



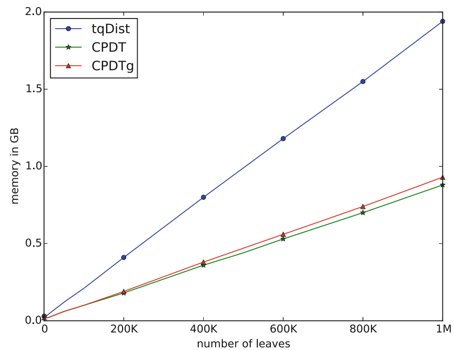
(a) Binary trees



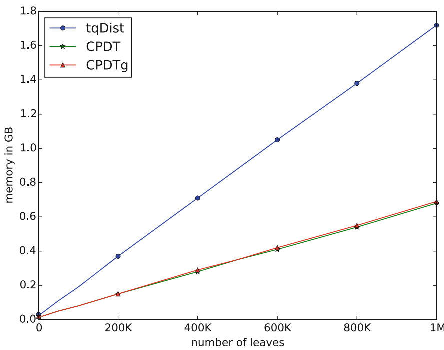
(b) $p_1 = 0.2$ $p_2 = 0.2$



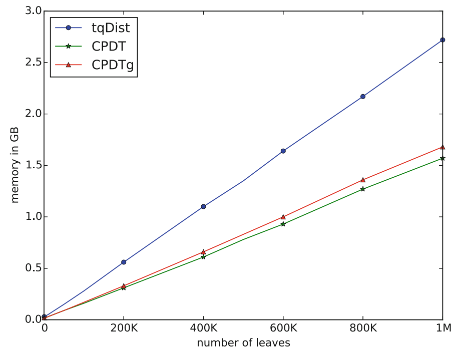
(c) $p_1 = 0.5$ $p_2 = 0.5$



(d) $p_1 = 0.8$ $p_2 = 0.8$



(e) $p_1 = 0.2$ $p_2 = 0.95$



(f) $p_1 = 0.95$ $p_2 = 0.2$

Fig. 3. Plots of the memory usage in gigabytes (y-axis) against n (x-axis), for binary trees and for some representative values of (p_1, p_2).

Table 2. Average memory usage, in GBs, on two 1M leaves trees, and the relative memory usage decrease over tqDist.

p_1	p_2	tqDist		CPDT		CPDTg	
0.0	0.0	3.09	1.00x	0.67	4.61x	-	-
0.2	0.2	2.87	1.00x	1.03	2.79x	1.14	2.52x
0.2	0.5	2.53	1.00x	0.99	2.56x	1.08	2.34x
0.2	0.8	2.05	1.00x	0.81	2.53x	0.87	2.36x
0.2	0.95	1.72	1.00x	0.68	2.53x	0.69	2.49x
0.5	0.2	2.77	1.00x	1.04	2.66x	1.15	2.41x
0.5	0.5	2.46	1.00x	1.01	2.44x	1.11	2.22x
0.5	0.8	2.00	1.00x	0.81	2.47x	0.87	2.30x
0.8	0.2	2.73	1.00x	1.13	2.42x	1.24	2.20x
0.8	0.5	2.37	1.00x	1.14	2.08x	1.24	1.91x
0.8	0.8	1.94	1.00x	0.88	2.20x	0.93	2.09x
0.95	0.2	2.72	1.00x	1.57	1.73x	1.68	1.62x

showing nearly six-fold improvement over tqDist. Note that as it does not rely on hashmaps, we have a single implementation of the CPDT.

For unrelated arbitrary-degree trees, CPDTg is clearly the fastest implementation, consistently being around three times faster than tqDist and showing noticeable improvements over CPDT. While tqDist performs better as we increase the values of p_1 and p_2, CPDT and CPDTg only show this trend with p_1, performing worse when p_2 is 0.5 and getting better as it moves away from it. All of the implementations perform very well when T_1 is extremely-branching, proving them to be able to easily handle a huge number of colors.

For related trees, all three implementations become much faster. CPDTg still has an obvious advantage, although speed-ups here are around 2.5x. This can be at least partially explained by overheads, such as tree parsing from files, becoming more significant as the running time of the actual algorithms decrease.

The average memory usage of the three implementations for pairs of unrelated trees with 1 million leaves are reported in Table 2 and Fig. 3. CPDT is the least memory-hungry, showing an improvement over tqDist of 4.61x on binary trees and up to 2.79x on arbitrary-degree ones; CPDTg is a close second. They all benefit from increasing p_2 (meaning less internal nodes in T_2) and suffer from increasing p_1 (meaning more colors), although tqDist proves to be less sensitive to it, as the advantage of CPDT is brought down to a still very respectable 1.73x in the extreme case where T_1 is extremely-branching.

6 Concluding Remarks

Some questions for future research are: Can the theoretical or practical running times of the CPDT-based algorithm be reduced? In particular, the CPDT

respects the definition of *locally balanced* in [2], so can the analysis be refined to prove that the time complexity of the new algorithm is in fact $O(n \log^2 n)$ using the technique in Sect. 5 of [2]? To make the algorithm faster in practice, one might try to parallelize it; unfortunately, this may be difficult due to possible imbalance in the trees and the intrinsic data-dependencies of the algorithm.

Computing the quartet distance for unrooted trees seems more difficult than computing the rooted triplet distance for rooted trees. It would be interesting to see if the CPDT can be adapted to get an efficient algorithm for this variant.

We remark that an unsolved open problem is whether or not the rooted triplet distance can be computed in $O(n)$ time. A linear-time algorithm would require a set of totally different techniques than the ones used here since Brodal *et al.*'s recursive recoloring scheme already introduces $\Omega(n \log n)$ work.

References

1. Bansal, M.S., Dong, J., Fernández-Baca, D.: Comparing and aggregating partially resolved trees. Theor. Comput. Sci. **412**(48), 6634–6652 (2011)
2. Brodal, G.S., Fagerberg, R., Mailund, T., Pedersen, C.N.S., Sand, A.: Efficient algorithms for computing the triplet and quartet distance between trees of arbitrary degree. In: Proceedings of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2013), pp. 1814–1832. SIAM (2013)
3. Cole, R., Farach-Colton, M., Hariharan, R., Przytycka, T., Thorup, M.: An $O(n \log n)$ algorithm for the maximum agreement subtree problem for binary trees. SIAM J. Comput. **30**(5), 1385–1404 (2000)
4. Critchlow, D.E., Pearl, D.K., Qian, C.: The triples distance for rooted bifurcating phylogenetic trees. Syst. Biol. **45**(3), 323–334 (1996)
5. Dobson, A.J.: Comparing the shapes of trees. In: Street, A.P., Wallis, W.D. (eds.) Combinatorial Mathematics III. LNM, vol. 452, pp. 95–100. Springer-Verlag, Heidelberg (1975)
6. Estabrook, G.F., McMorris, F.R., Meacham, C.A.: Comparison of undirected phylogenetic trees based on subtrees of four evolutionary units. Syst. Zool. **34**(2), 193–200 (1985)
7. Felsenstein, J.: Inferring Phylogenies. Sinauer Associates Inc, Sunderland (2004)
8. Fenwick, P.M.: A new data structure for cumulative frequency tables. Softw.: Pract. Experience **24**(3), 327–336 (1994)
9. Gusfield, D., Eddhu, S., Langley, C.: Optimal, efficient reconstruction of phylogenetic networks with constrained recombination. J. Bioinform. Comput. Biol. **2**(1), 173–213 (2004)
10. Holt, M.K., Johansen, J., Brodal, G.S.: On the scalability of computing triplet and quartet distances. In: Proceedings of the 16th Workshop on Algorithm Engineering and Experiments (ALENEX 2014), pp. 9–19. SIAM (2014)
11. Jansson, J., Lingas, A.: Computing the rooted triplet distance between galled trees by counting triangles. J. Discrete Algorithms **25**, 66–78 (2014)
12. McKenzie, A., Steel, M.: Distributions of cherries for two models of trees. Math. Biosci. **164**(1), 81–92 (2000)
13. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007), pp. 89–100. ACM (2007)

14. Sand, A., Holt, M.K., Johansen, J., Brodal, G.S., Mailund, T., Pedersen, C.N.S.: tqDist: a library for computing the quartet and triplet distances between binary or general trees. Bioinformatics **30**(14), 2079–2080 (2014)
15. sparsehash project webpage. https://code.google.com/p/sparsehash/
16. Documentation for unordered_map. http://www.cplusplus.com/reference/unordered_map/unordered_map/

Likelihood-Based Inference of Phylogenetic Networks from Sequence Data by PhyloDAG

Quan Nguyen and Teemu Roos(✉)

Department of Computer Science, Helsinki Institute for Information Technology HIIT, University of Helsinki, PO Box 68, 00014 Helsinki, Finland
{quan.nguyen,teemu.roos}@cs.helsinki.fi

Abstract. Processes such as hybridization, horizontal gene transfer, and recombination result in reticulation which can be modeled by phylogenetic networks. Earlier likelihood-based methods for inferring phylogenetic networks from sequence data have been encumbered by the computational challenges related to likelihood evaluations. Consequently, they have required that the possible network hypotheses be given explicitly or implicitly in terms of a backbone tree to which reticulation edges are added. To achieve speed required for unrestricted network search instead of only adding reticulation edges to an initial tree structure, we employ several fast approximate inference techniques. Preliminary numerical and real data experiments demonstrate that the proposed method, PhyloDAG, is able to learn accurate phylogenetic networks based on limited amounts of data using moderate amounts of computational resources.

Keywords: Phylogenetic networks · Likelihood-based inference · Phylogenetics · Probabilistic graphical models

1 Introduction

Phylogenetic trees are widely used for modeling the evolution of a group of organisms. However, trees are not able to represent reticulation events due to processes such as hybridization, horizontal gene transfer, and recombination. If reticulation is thought to be present, a phylogenetic network is a more useful model. For this reason, researchers in quantititive biology have been interested in representing evolutionary processes using network models since as early as the 1970s [20]. Even though various computational techniques have been proposed to deal with the challenges caused by network-like models, inferring the network structure from data remains a problem.

We propose a combination of solutions for speeding up the required computations in a likelyhood-based framework. These include a stochastic expectation-maximization (EM) algorithm for dealing with unobserved ancestral sequences. As a subroutine of the EM algorithm, we apply an approximate inference method known as loopy belief propagation [16], which provides dramatic computational

© Springer International Publishing Switzerland 2015
A.-H. Dediu et al. (Eds.): AlCoB 2015, LNBI 9199, pp. 126–140, 2015.
DOI: 10.1007/978-3-319-21233-3_10

savings when computing the required sampling distributions while avoiding any unwarranted independence assumptions (see e.g., [6]).

We describe a stand-alone method, which we call PhyloDAG[1], which can learn phylogenetic networks from data. The present implementation assumes a generic mixture model of the reticulation process but the model can be extended to handle more specific kinds of processes as well. Preliminary numerical and real world experiments demonstrate the utility of the method. For an application of PhyloDAG to the analysis of non-biological data, see [23].

The rest of the paper is organized as follows. In Sect. 2, we review some of the relevant prior work on likelihood-based phylogenetic networks. In Sect. 3, we describe our model in detail. We introduce the PhyloDAG method in Sect. 4, and present experimental results in Sect. 5. A summary and pointers for future work are given in Sect. 6.

2 Related Work

Likelihood-based inference has become a popular approach in phylogenetics since it was first proposed by Felsenstein [5]. Likelihood-based methods are widely considered to be the state-of-the-art in molecular phylogenetics [4,26].

The first framework for likelihood-based inference of phylogenetic networks was proposed by Haeseler and Churchill [8]. Based on their work, Strimmer and Moulton [21] proposed to use directed graphical models, or Bayesian networks, as a represention of explicit likelihood-based phylogenetic networks. Their framework was first applied to split networks, but it can be easily applied to evolutionary networks [22]. However, networks pose major computational challenges for likelihood-based inference. Computations involving unobserved ancestral sequences are in general intractable. The solution applied in [21] is to approximate the likelihood by method similar to Gibbs sampling.

Strimmer et al. [22] model reticulation events by introducing a random variable that indicates which one of the possible ancestral taxa is active and using the same mechanism as in tree-structured models as if the active taxon were the only immediate ancestor. The random choice of the ancestor taxon is repeated independently at each site according to fixed but unknown weight parameters. The authors referred to this as the mixture model. In this work, we adopt the mixture model and develop novel efficient algorithms that can be used for inferring the network structure and parameters from data.

Jin et al. [10] point out the importance of allowing different evolutionary mechanisms for different genomic sites. However, despite their emphasis on the differences between their approach and that of Strimmer et al., the existence of a separate edge length parameter for each site, which significantly increases the model complexity but simplifies the computations, turns out to be the distinctive feature of their model. In follow-up work, Park and Nakleh [15] consider given

[1] The implementation is available for download at http://phylomemetic.wordpress.com/2015/04/17/phylodag/.

genomic regions inside which a fixed ancestor taxon and edge length value is used.

There are also other sophisticated ways to relax the mixture model assumption. Husmeier and Wright [9] and Webb et al. [25] assume each site to be generated from an unknown phylogenetic tree which is a hidden state in a hidden Markov model (HMM). Transitions between the states of the HMM constitute breakpoints from one phylogenetic structure to another. This approach is likely to be more realistic under recombination scenarios, but it is very computationally expensive since it introduces complex dependencies between the sites and the state space of the HMM grows exponentially in the number of taxa.

In all of the aforementioned work, due to the said computational challenges, network search is either restricted to a small set of possible networks given explicitly by the user or more implicitly to networks obtained by adding reticulation edges to a fixed backbone tree structure obtained by standard tree methods such as MrBayes [17]. A key assumption behind the use of a backbone tree is that even when the actual phylogenetic process involves reticulation events, a tree structure estimated from the data comprises a part of the true network that represents the phylogenetic history. If this is the case, the true network can be obtained by adding reticulation edges. Unfortunately, in our experience this assumption is unlikely to hold in practice. In Sect. 5.3 we demonstrate simple cases where a violation of the assumption leads to suboptimal outcomes.

Apart from horizontal gene transfer and other processes discussed above, deep coalescence arising from incomplete lineage sorting is another source of incompatibility of gene trees for individual sites or genes of a given same set of taxa, see e.g., [13]. Since deep coalescence tends to occur even when the organisms' evolution is completely tree-like, it is usually not considered to be a type of reticulation. The models used to handle deep coalescence are also somewhat distinct from those used to handle reticulation. Recently, there have been several attempts to incorporate reticulation into models for deep coalescence [12,27].

3 Likelihood-Based Inference in Phylogenetic Trees and Networks

We adopt the standard likelihood-based framework in phylogenetics and let each node (either leaf or internal) of a phylogenetic tree correspond to a taxon. Leaf nodes are assumed to be extant taxa whose genomic sequences are observed. In this work we focus on DNA sequences although for example protein sequences can in principle be handled in the same fashion.

We denote the probability that a DNA sequence associated to node X_i in a phylogenetic tree evolves from the sequence in its immediate ancestor, called its *parent*, Pa_i in time proportional to branch length τ_i by $P_{\tau_i}(X_i \mid \mathrm{Pa}_i)$. These local probabilities are specified explicitly by a sequence evolution model such as the Jukes-Cantor (JC) model [11] as a function of τ_i. In the following, we denote random variables and sequences like X_i by upper case letters and their values, such as x_i, by lower case letters.

The above kind of probabilistic model describes the following evolution scenario. The nucleotide sequence at the root X_r is drawn independently from a stationary distribution π obtained as the limit $\pi(X) = \lim_{\tau\to\infty} P_\tau(X \mid y)$ for any sequence y. The sequence evolves independently along the edges of the tree. Assuming a fully observed tree T with p nodes (taxa), the likelihood of a single site at all taxa is factorized as

$$P_{(T,\tau)}(X_1 = x_1, ..., X_p = x_p) = \pi(x_r) \prod_{i\neq r} P_{\tau_i}(x_i \mid pa_i), \tag{1}$$

where pa_i denotes the nucleotide at the site in question in the parent of taxon X_i in tree T. However, since we assume that only the sequences in the leaf nodes are observed, the internal nodes, including the root node, represent ancestral taxa whose biological sequences are unavailable, and hence they become latent (unobserved) variables in the model.

Following and extending the convention familiar from phylogenetic trees, we assume that any node in a phylogenetic network is classified in one of three categories based on the number of its parents. First, the unique root node has no parents and two children (immediate descendants). Second, *tree nodes* have a single parent and either zero or two children. For both of these classes of nodes, the evolutionary model coincides with the model commonly used for likelihood-based phylogenetic trees. The third class of nodes are the *reticulation nodes* which have two parents and either zero or two children. For a given nucleotide $x_i \in \{A, C, G, T\}$ in reticulation node X_i, we have the conditional probability given its parents' states $pa_i = (y_i, z_i)$ as the weighted sum of its conditional probability given a single parent:

$$P_{(w_i,\tau_i)}(x_i \mid pa_i) = w_i P_{\eta_i}(x_i \mid y_i) + (1 - w_i) P_{\zeta_i}(x_i \mid z_i), \tag{2}$$

where the probabilities on the right side of the equation are the same as in the case of tree models, and the weight parameter w_i as well as the edge length parameters $\tau_i = (\eta_i, \zeta_i)$ are parameters whose values need to be given in order to make the model fully specified. Plugging the above terms in the factorization (1) provides a complete probability model for reticulate evolution.

The model in Eq. (2) is the mixture model of Strimmer and Moulton [21]. If genomic regions that follow a fixed ancestry are given like in [15], they can be incorporated in the model by treating sites within a given region as a sample of data from the same source. In this work, we focus on the case where the sites are independent.

From a computational point of view, most of the complications arise from the fact that the observed-data likelihood involves a summation over the possible values of the latent variables. In tree topologies, well-known techniques exist for carrying out the summation in linear time with respect to the size of the tree [5]. These techniques are known in probabilistic graphical models more generally as *variable elimination*. Felsenstein [5] uses the expectation–maximization (EM) algorithm [3] to estimate branch length parameters in tree-structured models.

In the following, we introduce methods for approximating the computations in the case where the phylogenetic hypothesis involves reticulation nodes.

4 The PhyloDAG Method

We propose an efficient method for likelihood-based inference of phylogenetic networks. The key novelties of the PhyloDAG method include a stochastic EM algorithm for learning the structure and parameters of the network as well as a fast loopy belief propagation (LBP) algorithm which is used to accelerate the required computations involving the latent variables in the model.

The outer loop of the algorithm is a stochastic structural EM (SSEM) algorithm. Similar to regular EM, SSEM repeats iterations consisting of an expectation (E) step followed by a maximization (M) step. Slightly different from regular EM, SSEM is based on stochastic sampling of latent variables in the E step in order to obtain (pseudo-)complete data. The word 'structural' refers to the fact that the M step involves a maximization not only over model parameters (parent weights and edge lengths) but also over the model structure (network topology). Inside the E step, an inner loop based on LBP replaces the variable elimination algorithm commonly used in dealing with latent variables in tree-structured phylogenies.

We initialize the structure as a phylogenetic tree obtained from by Neighbor-Joining algorithm [18], which is used for sampling of the latent variables in the first E step. After this the initial tree is discarded and in particular, it is not used to restrict the structure search in any way. The E and M steps are repeated until the objective function converges.

Let o denote all the observed data, and let L denote the latent variables. The model structure and parameters on each iteration, t, of the algorithm are denoted as $G^{(t)}$ and $\theta^{(t)} = (w^{(t)}, \tau^{(t)})$ respectively.

4.1 Stochastic E Step

Recall that in the E step, regular EM computes the expected complete data log-likelihood with respect to the latent variables

$$E_{L|o,G^{(t)},\theta^{(t)}} \left[\log P_{(G,\theta)}(o, L)\right], \tag{3}$$

where the structure and parameters $G, \theta = (w, \tau)$ are allowed to differ from $G^{(t)}, \theta^{(t)}$. The above quantity is then maximized with respect to G and θ in the M step. For complete data, under the i.i.d. assumption, the log-likelihood for a set of sequences of length N becomes a sum with N terms. We group them based on the configurations of a node and its parents:

$$\log P_{(G,\theta)}(o, l) = \sum_{\substack{i=1,\ldots,p \\ x\in\{A,C,G,T\} \\ pa\in\{A,C,G,T\}^{q_i}}} N_{ixpa} \log P_{\theta_i}(X_i = x \mid \mathrm{Pa}_i = pa), \tag{4}$$

where q_i denotes the number of parents for variable X_i, and the count N_{ixpa} indicates the number of sites where X_i takes value x and its parents take values pa. The counts N_{ixpa} are called the *sufficient statistics* since given the model parameters they uniquely determine the likelihood. For each combination of values x, pa, the conditional probability $P_{\theta_i}(X_i \mid \mathrm{Pa}_i)$ can be considered a constant, and the log-likelihood is a linear function of the sufficient statistics. Hence computing the *expected* log-likelihood only requires the computation of the expectation of the sufficient statistics, which can be done by summing over all the independent sites

$$E_{L|o,\theta^{(t)}}\left[N_{ixpa}\right] = \sum_{j=1}^{N} P_{(G^{(t)},\theta^{(t)})}(X_i^j = x, \mathrm{Pa}_i^j = pa \mid o^j),$$

where superscript j indicates the site. The required computations may easily become infeasible since the conditional probabilities in the above formula may require complex inference procedures in case the network structure is not of a very specific kind (such as a tree).

Friedman et al. [6] suggest an approximation of the form

$$P_{(G^{(t)},\theta^{(t)})}(L \mid o) \approx \prod_{i=1}^{|L|} P_{(G^{(t)},\theta^{(t)})}(L_i \mid o), \tag{5}$$

where $|L|$ denotes the number of latent variables. This amounts to treating each node and its (potential) parent(s) as conditionally independent given the observed data. We suggest a different approximation which avoids the above drastic conditional independence assumption by sampling the latent variables from their conditional distribution $P_{(G^{(t)},\theta^{(t)})}(L \mid o)$. To do so, we exploit the chain rule

$$P_{(G^{(t)},\theta^{(t)})}(L \mid o) = \prod_{i=1}^{|L|} P_{(G^{(t)},\theta^{(t)})}(L_i \mid L_{1:i-1}, o) \tag{6}$$

We will sample a value l_1 for the latent variable L_1 from its LBP-approximated conditional distribution given the observed data o, after which we include the value l_1 in the set of (pseudo-)observed variables, and proceed recursively to sample all the remaining latent variables. The procedure is outlined as Algorithm 1 below.

Data: o: vector of observed data (at a single site)
Result: l: vector of sampled data for latent variables (at the same site)
for $i \in \{1, ..., |L|\}$ **do**
  Perform LBP to approximate $P_{(G^{(t)},\theta^{(t)})}(L_i \mid l_1, \ldots, l_{i-1}, o)$
  Draw value l_i from the obtained distribution.
end
return $l_1, \ldots, l_{|L|}$

Algorithm 1. Sampling latent variables from their joint conditional distribution approximated by loopy belief propagation (LBP).

In practice, drawing a single sample vector, l, per site appears to be sufficient to obtain sufficiently accurate approximations of the expected counts unless the number of sites is very small. This strategy is more generally called *stochastic* EM [2]. Theoretical and numerical results backing up its validity are presented in [14].

4.2 Structural M Step

Having sampled the latent variables in the E step to obtain the pseudo-complete data $(o, \tilde{l})$, the M step is used to estimate a phylogenetic hypothesis, i.e., the network structure, G, the weight parameters associated with possible reticulations, w, and the edge lengths, τ. All of them are estimated by maximizing the following objective function:

$$(G^{(t+1)}, \theta^{(t+1)}) = \arg\max_{(G,\theta)} \log P_{(G,\theta)}(o, \tilde{l}), \tag{7}$$

where $\tilde{l}$ denotes the sampled values for all hidden variables obtained in the stochastic E step, and $\theta = (w, \tau)$. Any Bayesian network learning algorithm can be applied with the pseudo-complete data. We start with an empty network and apply local modifications including edge deletions, additions, and reversals until the likelihood score cannot be improved. Further heuristics including a tabu search to escape local optima are detailed in the next section.

The parameters can be estimated in a relatively straightforward manner under the JC model, which we use in our implementation, as well as other commonly used sequence evolution models.

4.3 Avoiding Local Optima and Overfitting

As is typical to EM-based algorithms, it is beneficial to implement some modifications that help to avoid the search from getting stuck to local optima. Since the method is based on maximizing the likelihood, it is also prone to overfitting unless some complexity regularization is performed.

First, to escape local optima in the structure search within the M step, we apply so called *tabu search* heuristic [7] where structure modifications that reduce the likelihood score are accepted in case there are no available local modifications that improve the score. To do so, we maintain a *tabu list* wherein we record recently visited graph structures in order to prevent repeatedly visiting the same structures. The search is terminated after a maximum number of iterations is reached or when no improvement in the best structure occurs in several steps, after which the overall best structure is returned.

Moreover, even if the M step finds the globally optimal structure given the pseudo-complete data $(o, \tilde{l})$, the EM iterations may end up in a local optimum of the incomplete-data likelihood, where the pseudo-observations sampled in the E step reinforce the current (locally optimal) structure hypothesis; see [6]. The stochastic EM algorithm is less prone to this problem than regular EM (see [14])

but when the sequence length is large enough, the problem persists. We therefore adopt the perturbation method in *deterministic annealing EM* by Ueda and Nakano [24]. This means that the sampling distributions in Algorithm 1 is raised to power $\beta \leq 1$ and normalized after it has been inferred by LBP so that the pseudo-observations are drawn from a distribution proportional to

$$P_{(G^{(t)},\theta^{(t)})}(L_i \mid l_{1:i-1}, o)^{\beta}$$

where $1/\beta$ acts like a temperature parameter. The inverse temperature β should be small at the beginning, so that the sampling distribution is close to uniform. When β is increased, the distribution is perturbed less and it will approach the unperturbed distribution as $\beta \to 1$. Currently we heuristically set $\beta^{(1)} = 0.6$ and $\beta^{(t+1)} = \min\{1.0, 1.05\,\beta^{(t)}\}$.

Finally, to avoid overfitting due to the increased flexibility allowed by the reticulation nodes, we use the Bayesian information criterion (BIC) [19] to penalize the score function, which becomes

$$\text{BIC}(G, \theta \mid o, \tilde{l}) = \log P_{(G,\theta)}(o, \tilde{l}) - \frac{k}{2}\log N, \tag{8}$$

where k is number of free parameters in model G (including both the weights and the edge lengths), and N is the sequence length. The second term in the BIC score can be seen as a complexity penalty reducing the tendency to overfit. Because the penalizing term is the same in both complete and incomplete data, when BIC is used instead of ML as the scoring function in Eq. (7), the validity of the EM algorithm is maintained; see [3]. The good performance of BIC in preventing overfitting in phylogenetic networks has been observed by Park and Nakleh [15].

4.4 Postprocessing of the Networks

From the point of view of network search, the properties required from phylogenetic networks can be a problem since they might restrict the exploration of promising structures. Therefore, we perform the SSEM algorithm using unconstrained network structures, and apply the following sequence of postprocessing steps only after the algorithm has converged:

1. Recursively remove all unlabeled leaves.
2. Remove unlabeled nodes with in-degree and out-degree of 1.
3. Edge from two labeled nodes (A, B) with length τ_{AB} is replaced by (x, A) with $\tau_{xA} = \epsilon$ and (x, B) with $\tau_{xB} = \tau_{AB}$, where x is an internal node and $\epsilon \approx 0$.
4. An internal node x with more than two children, $x_1, x_2, ...$, is replaced by a new internal node y with children x_1 and x, and x_1 is removed from the children of x. This rule is applied recursively until x has at most two children.

We refer the reader to [6] for detailed illustrations and the proof why these alterations do not change the score.

5 Experiments

To demonstrate the practical utility of the proposed method, we perform experiments on both simulated and real data. We first illustrate the accuracy of the likelihood evaluation based on loopy belief propagation which we use in the E step to show that model comparison based on approximated likelihood is reliable. We then demonstrate the PhyloDAG method by applying it to both simulated and real data.

5.1 Exact vs Approximated Likelihood: An Illustration

We apply a procedure where we simulate DNA data with increasing sequence length, $N = 50, 100, 150, \ldots, 500$ for the leaf nodes of an arbitrary *tree* structure following the JC model and no insertions or deletions (indels). To create a hybrid node, we pick two leaf nodes and produce a hybrid sequence by randomly copying the character at each site from either one of chosen the leaf nodes according to some fixed weights. The two leaf nodes are then removed and replaced by the hybrid node whose parents are those of the removed leaf nodes. The resulting phylogenetic network is shown in Fig. 1.

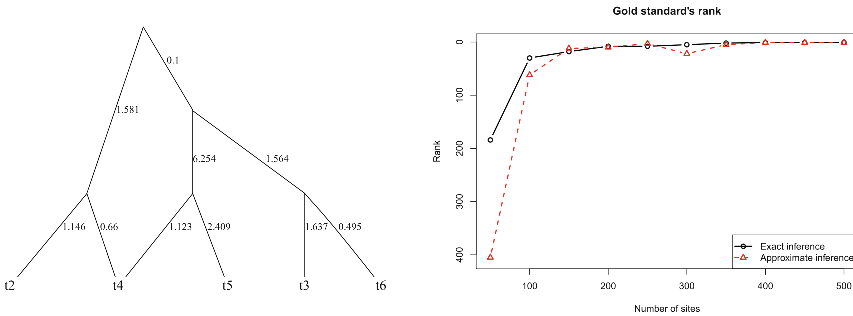


Fig. 1. *Left:* The true phylogenetic network. Edge lengths (shown along the edges) are drawn from an exponential distribution. *Right:* Ranks of the true network as a function of sequences length using exact and approximate computations. Both curves tend to increase which means that as the sample size grows, they eventually rank the true network first.

We then modify the true structure by adding and removing edges to obtain a sample of 1000 incorrect topologies (including some duplicates). We rank these 1001 phylogenies by their BIC scores where instead of the pseudo-complete likelihood $P(o, \tilde{l})$ we use the incomplete-data likelihood $P(o)$ so that the scores are comparable across different networks. We compare the ranking performance obtained by using an exact brute-force computation of the incomplete-data likelihood and the LBP approximation. Since the problem size is small, even the exact computation takes less than three seconds for samples up to $N = 500$

using an efficient implementation. In the case of LBP, we use the identity $P(o) = P(\tilde{l}, o)/P(\tilde{l} \mid o)$ which holds for all $\tilde{l}$. The LBP approximation takes less than 0.5 seconds. Since the exact computation takes exponential time in the number of latent variables, it quickly becomes useless in practice as the problem size is increased, whereas the LBP method scales to much bigger problems.

In Fig. 1, the ranks of the true phylogenetic network by both exact and approximate inference are plotted against the sample size. In both methods the rank of the true structure tends to improve as the sequence length grows. The brute-force method ranks the true structure higher for sequences up to 100 nucleotides but for longer sequences the differences are generally very small.

5.2 Structure Search on Synthetic Data

Following the experimental procedure described above, we generate a data set with 15 taxa and sequence length 2000. The true underlying phylogenetic network is shown in Fig. 2. We apply PhyloDAG as well as PhyloNet[2], a recent method proposed by Yu et al. [27].

Figure 3 shows the result of PhyloDAG. In order to make it easier to compare the structure inferred by PhyloDAG to the correct network, four groups of taxa are shaded and labelled as A–D. Except some minor differences like the position of group C (taxa $t6$ and $t17$), PhyloDAG infers the structure almost correctly. In particular, the two reticulation events at $t7$ and $t9$ are inferred correctly. Note that the BIC criterion was used to decide the number of reticulate edges in the model based on the data without user intervention.

In PhyloNet, we apply the maximum likelihood phylogenetic network method. PhyloNet requires a backbone tree, and as suggested by Yu et al. [27], we use a backbone obtained by MrBayes [17]. PhyloNet also requires that the number of reticulations be specified, and we provide the correct number, two. Other settings of PhyloNet are set to default values. By default, the algorithm is repeated 10 times and the network that maximizes the likelihood as computed by PhyloNet is produced as the output.

Figure 4 shows the structure inferred by PhyloNet. The solid edges are from the backbone tree by MrBayes and dotted edges are the added reticulation edges. In this experiment, despite the good backbone tree, the two reticulation edges suggested by PhyloNet are incorrect. The reticulation edge near $t11$ may correspond to an actual reticulation (see Fig. 2) between the immediate ancestors of $t11$ and $t4$ which results in the sequence at $t7$ but it is still relatively far from correct. It will be interesting to analyse in detail why PhyloNet produces reticulate edges between neighboring nodes only. The experiments presented by Yu et al. [27] do not test whether this behavior occurs generally: they involve only 4 or 5 taxa so that reticulation between more distant branches cannot be investigated. Another possible explanation for the poor result is a different sequence evolution model employed in PhyloNet whereas PhyloDAG may benefit from the

[2] http://bioinfo.cs.rice.edu/phylonet.

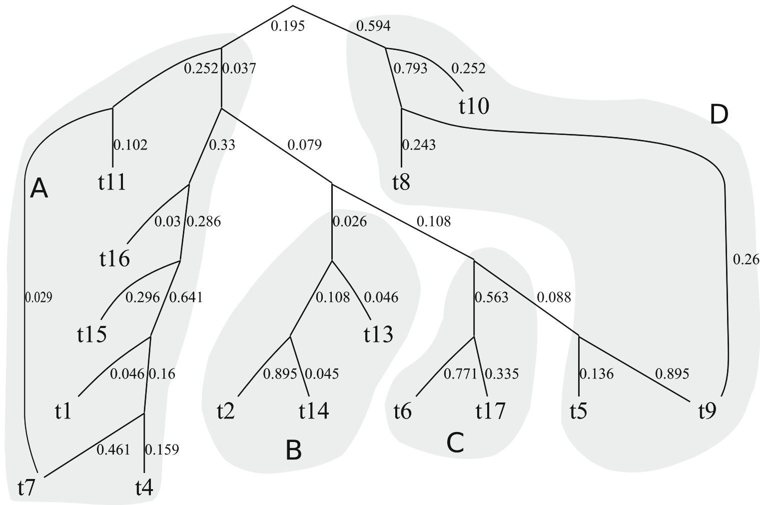


Fig. 2. The true phylogenetic network used to simulate 15 sequences, including two reticulations (taxa $t7$ and $t9$). Numbers indicate edge lengths. The groups A–D are shaded for clarity.

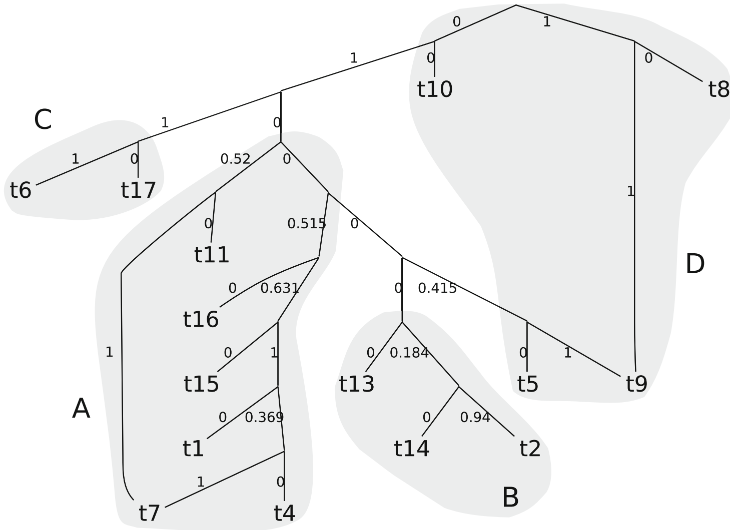


Fig. 3. Result of PhyloDAG for data simulated from the network in Fig. 2. Numbers indicate estimated edge lengths.

fact that it is based on the JC model which is also used to simulate the sequences – however, see the results on real data in the next subsection.

The test is done on an 3.4 GHz 8 core CPU computer with 16 GB of memory. For this data set, PhyloDAG runs 12 iterations of the SSEM procedure which

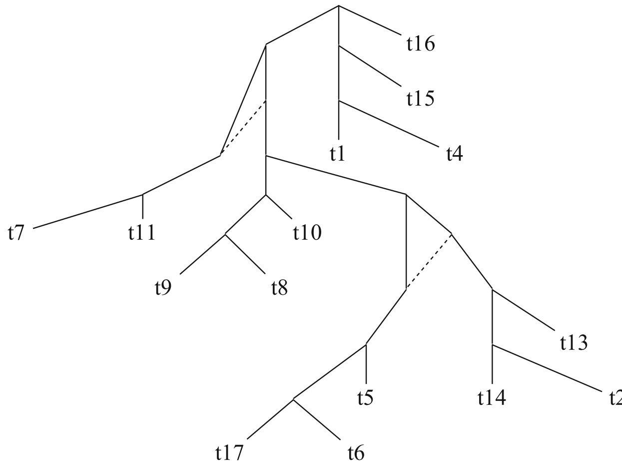


Fig. 4. Result of PhyloNet [27] for data simulated from the network in Fig. 2.

takes less than three minutes. On the same setup, PhyloNet runs for about five hours (excluding the running time of MrBayes).

5.3 Real Data Experiment

We test PhyloDAG on a real data set "Feliner"[3]. This is one of a few data sets where the underlying phylogenetic network is at least partially known since they result from an artificial hybridization of *Armeria* plants in a greenhouse [1].

The data contains a number of *Armeria villosa ssp. longiaristata* (VIL) and *Armeria colorata* (COL) plants. The specimens VIL#58/120 and COL#11/12 were crossed to create a hybrid generation labeled F1. We select a subset of the original data set that includes hybrid taxa and their ancestors, so that the relationships between the taxa are known from the experiment and the results are easy to interpret. We expand heterozygous sites as pairs of nucleotides following the encoding of Aguilar et al. [1] (for example, W in the sequence is expanded as nucleotides AT). The total sequence length is 626 nucleotides after the preprocessing. The problem is complicated by the fact that all the sequences are very similar to each other: they differ at not more than 10 sites.

Figure 5 shows the results of PhyloDAG on the subset of seven sequences from the Feliner data. PhyloDAG groups the COL and VIL families correctly and includes a reticulation edge correctly identifying the hybrid ancestry of the F1 family. The edge lengths are compatible with the observation that the F1 sequences are very close to the COL sequences (about 4–5 differences) and somewhat less close to the VIL sequences (about 7–10 differences).

Figure 6 shows the PhyloNet result, obtained using default settings. The backbone tree (solid lines) obtained by MrBayes places the hybrid F1 species

[3] http://www.rjr-productions.org/Database.html.

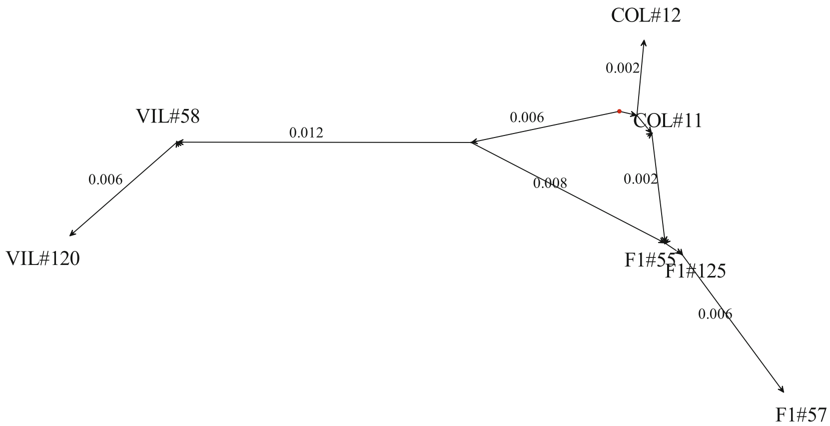


Fig. 5. PhyloDAG result for the Feliner data. In this case, edge lengths are drawn proportional to their estimates.

between the ancestor groups COL and VIL. The PhyloNet method was repeated twice: first, setting the number of reticulations to one, and another time, setting it to two. The network show in the figure includes all the reticulate edges (dotted lines) appearing in either of the resulting networks. Similar to the simulation experiment, the reticulate edges by PhyloNet are near the hybrid taxa but their end points are too close to each other to provide useful information about the ancestry of the hybrids.

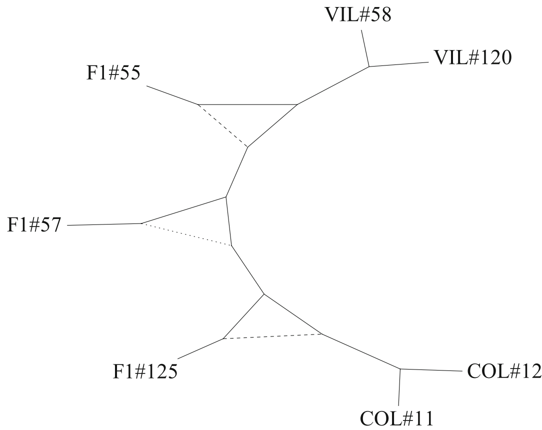


Fig. 6. PhyloNet result for the Feliner data. (Edge lengths not drawn proportional to their estimates.)

6 Conclusions

We propose a new method, PhyloDAG, for constructing likelihood based phylogenetic networks from sequence data. The method is based on (*i*) structural EM which treats the graph structure as a parameter to be optimized in the M step (*ii*) an efficient stochastic implementation of the E step based on loopy belief propagation. The key difference in the procedure compared to earlier likelihood-based approaches is that whereas earlier methods tend to involve an EM or Monte Carlo type algorithm as an inner loop of a structure learning process, we put the structure learning procedure inside the M step of an EM-type algorithm. This significantly speeds up the structure learning process since it avoids costly iterative likelihood evaluations, and allows an unrestricted structure search without a fixed backbone tree.

We presented simulations and a real data experiment to demonstrate the accuracy of the method. Compared to another recent likelihood-based method, PhyloDAG was orders of magnitude faster and produced much more accurate network structures. Variations of our method can be constructed where different models of reticulation are applied. Additional large scale experiments with real and simulated data will be required to assess the benefits of our approach.

Acknowledgments. This work was supported in part by the Academy of Finland (Center-of-Excellence COIN). We are grateful to Vincent Moulton for insightful comments. The anonymous reviewers suggested a comparison to the PhyloNet method and made several other suggestions that significantly improved the paper.

References

1. Aguilar, J.F., Rosselló, J., Feliner, G.N.: Nuclear ribosomal DNA (nrDNA) concerted evolution in natural and artificial hybrids of Armeria (Plumbaginaceae). Mol. Ecol. **8**(8), 1341–1346 (1999)
2. Celeux, G., Diebolt, J.: The SEM algorithm: a probabilistic teacher algorithm derived from the EM algorithm for the mixture problem. Comput. Stat. Q. **2**(1), 73–82 (1985)
3. Dempster, A.P., Laird, N.M., Rubin, D.B.: Maximum likelihood from incomplete data via the EM algorithm. J. Roy. Stat. Soc. Ser. B **39**(1), 1–38 (1977)
4. Durbin, R., Eddy, S.R., Krogh, A., Mitchison, G.: Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids. Cambridge University Press, Cambridge (1998)
5. Felsenstein, J.: Evolutionary trees from DNA sequences: a maximum likelihood approach. J. Mol. Evol. **17**, 368–376 (1981)
6. Friedman, N., Ninio, M., Pe'er, I., Pupko, T.: A structural EM algorithm for phylogenetic inference. J. Comput. Biol. **9**(2), 331–353 (2002)
7. Glover, F., Laguna, M.: Tabu Search. Kluwer Academic Publishers, Norwell, MA (1997)
8. Haeseler, A., Churchill, G.A.: Network models for sequence evolution. J. Mol. Evol. **37**(1), 77–85 (1993)

9. Husmeier, D., Wright, F.: Detection of recombination in DNA multiple alignments with hidden Markov models. J. Comput. Biol. **8**(4), 401–427 (2001)
10. Jin, G., Nakhleh, G., Snir, S., Tuller, T.: Maximum likelihood of phylogenetic networks. Bioinformatics **22**, 2604–2611 (2006)
11. Jukes, T.H., Cantor, C.R.: Evolution of protein molecules. Mamm. Protein Metab. **3**, 21–132 (1969)
12. Meng, C., Kubatko, L.S.: Detecting hybrid speciation in the presence of incomplete lineage sorting using gene tree incongruence: a model. Theor. Popul. Biol. **75**(1), 35–45 (2009)
13. Morrison, D.: Introduction to Phylogenetic Networks. RJR Productions, Uppsala (2011)
14. Nielsen, S.F.: The stochastic EM algorithm: estimation and asymptotic results. Bernoulli **6**, 457–489 (2000)
15. Park, H.J., Nakhleh, L.: Inference of reticulate evolutionary histories by maximum likelihood: the performance of information criteria. BMC Bioinf. **13**(Suppl 19), S12 (2012)
16. Pearl, J.: Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1988)
17. Ronquist, F., Huelsenbeck, J.P.: MrBayes 3: Bayesian phylogenetic inference under mixed models. Bioinformatics **19**(12), 1572–1574 (2003)
18. Saitou, N., Nei, M.: The neighbor-joining method: a new method for reconstructing phylogenetic trees. Mol. Biol. Evol. **4**, 406–425 (1987)
19. Schwarz, G.: Estimating the dimension of a model. Ann. Stat. **6**(2), 461–464 (1978)
20. Sneath, P.H.A.: Cladistic representation of reticulate evolution. Syst. Zool. **24**, 360–368 (1975)
21. Strimmer, K., Moulton, V.: Likelihood analysis of phylogenetic networks using directed graphical models. Mol. Biol. Evol. **17**(6), 875–881 (2000)
22. Strimmer, K., Wiuf, C., Moulton, V.: Recombination analysis using directed graphical models. Mol. Biol. Evol. **18**(1), 97–99 (2001)
23. Tehrani, J., Nguyen, Q., Roos, T.: Oral fairy tale or literary fake? Investigating the origins of Little Red Riding Hood using phylogenetic network analysis. Digital Scholarship in the Humanities (2015, to appear)
24. Ueda, N., Nakano, R.: Deterministic annealing EM algorithm. Neural Netw. **11**(2), 271–282 (1998)
25. Webb, A., Hancock, J.M., Holmes, C.C.: Phylogenetic inference under recombination using Bayesian stochastic topology selection. Bioinformatics **25**(2), 197–203 (2009)
26. Whelan, S., Lio, P., Goldman, N.: Molecular phylogenetics: state-of-the-art methods for looking into the past. Trends Genet. **17**(5), 262–272 (2001)
27. Yu, Y., Dong, J., Liu, K.J., Nakhleh, L.: Maximum likelihood inference of reticulate evolutionary histories. Proc. Nat. Acad. Sci. **111**(46), 16448–16453 (2014)

Constructing Parsimonious Hybridization Networks from Multiple Phylogenetic Trees Using a SAT-Solver

Vladimir Ulyantsev(✉) and Mikhail Melnik

ITMO University, Saint Petersburg, Russia
{ulyantsev,melnik}@rain.ifmo.ru

Abstract. We present an exact algorithm for constructing minimal hybridization networks from multiple trees which is based on reducing the problem to the Boolean satisfiability problem. The main idea of our algorithm is to iterate over possible hybridization numbers and to construct a Boolean formula for each of them that is satisfiable iff there exists a network with such hybridization number. The proposed algorithm is implemented in a software tool PhyloSAT. The experimental evaluation of our algorithm on biological data shows that our method is as far as we know the fastest exact algorithm for the minimal hybridization network construction problem.

Keywords: Phylogenetic networks · Boolean satisfiability · SAT · Bioinformatics · Genetics

1 Introduction

A phylogenetic network is a powerful model for reticulate evolutionary processes (such as horizontal gene transfer and hybrid specification). Briefly, a phylogenetic network is the directed acyclic graph which has nodes (called reticulation nodes) with more than one incoming edge. Phylogenetic networks have been studied by several researchers [7–9]. There are several formulations of phylogenetic network construction problem with various modelling assumptions and different types of input data. In this paper we focus on the specific type of phylogenetic networks called hybridization networks [4,12].

We consider a set of gene trees on the same set of taxa as input data for hybridization network construction. Each gene tree models the evolutionary history of some gene. Due to reticulate evolutionary events, these trees can have different topologies. The aim is to construct a hybridization network containing the smallest possible number of reticulation nodes and displaying each of the input trees.

Most of the algorithms for hybridization network construction are heuristic [11,14] and usually deal with only two trees. However, the exact algorithm $\mathrm{PIRN_C}$ which is able to process more than two input trees has been introduced recently [14].

© Springer International Publishing Switzerland 2015
A.-H. Dediu et al. (Eds.): AlCoB 2015, LNBI 9199, pp. 141–153, 2015.
DOI: 10.1007/978-3-319-21233-3_11

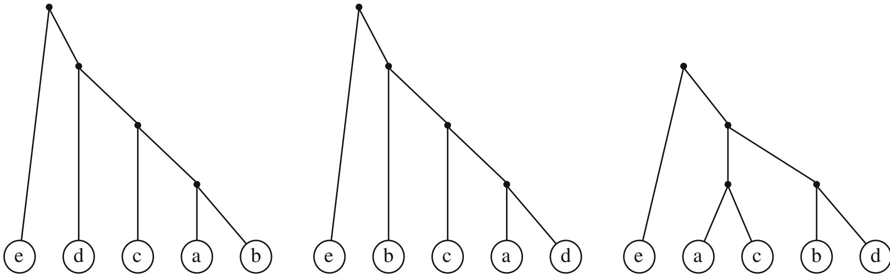


Fig. 1. Example of three gene trees over the set of taxa {a, b, c, d, e}.

In this paper we introduce a new approach to exact parsimonious hybridization network construction from multiple input trees based on satisfiability (SAT) solvers. The SAT problem is known to be NP-complete [3], but state-of-the-art SAT-solvers running on modern hardware are able to solve SAT instances having tens of thousands of variables and hundreds of thousands of clauses in several minutes.

SAT-solver based algorithms have been successfully applied to efficiently calculate evolutionary tree measures [2] as well as to solve problems in other domains such as finite-state machine induction [5], software verification [1].

The general outline of our approach is to convert an instance of hybridization network construction problem to an instance of the SAT problem (a Boolean formula), solve it using a SAT-solver and then if the solution exists convert the satisfying assignment into the hybridization network. Our approach leads to the exact algorithm that outperforms PIRN_C on our tests.

The paper is structured as follows. Section 2 gives the formal definitions, Sect. 3 describes the Boolean formula construction process and Sect. 4 gives the experimental results. The paper is concluded in Sect. 5.

2 Definitions and Background

We define a *phylogenetic tree* as a leaf-labelled tree constructed over a set of taxa. Throughout this paper we assume that trees are rooted and binary.

For any node v let $d^-(v)$ be the in-degree of v and $d^+(v)$ be the out-degree of v. A *hybridization network* on a set of taxa X is a directed acyclic graph with a single root ρ and leaves bijectively mapped to the set of taxa X. If $d^-(v) > 1$ then node v is called a *reticulation node*. In this paper we assume that $d^-(v) = 2$ and $d^+(v) = 1$ is true for every reticulation node v. We make this assumption by noting that we can convert a reticulation node with in-degree of three or more to a sequence of reticulation nodes with in-degrees of two [13]. Other nodes are regular tree nodes.

Every hybridization network N can be reduced to a tree. To do this we firstly keep only one of the incoming edges for each reticulation node in N. Secondly, we contract edges to remove any node v such that $d^-(v) = d^+(v) = 1$. By these two steps we reduce the network N to the tree T'.

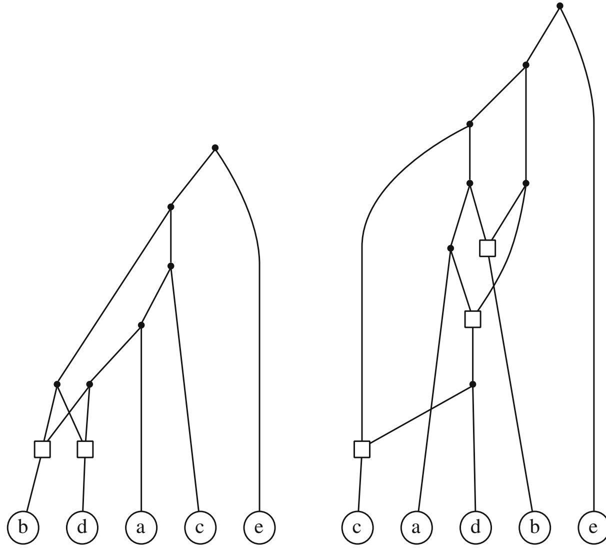


Fig. 2. Possible hybridization networks for the trees from Fig. 1 with two and three hybridization events respectively. Reticulation nodes are shown as boxes.

We say that hybridization network N *displays* phylogenetic tree T if we can choose the edges of reticulation nodes in such a way that after edge-contraction the obtained tree T' will be isomorphic to T. In Fig. 2 each of the three trees from Fig. 1 is displayed in both hybridization networks.

A *hybridization number* of network N with root ρ is commonly defined as $h(N) = \sum_{v \neq \rho} (d^-(v) - 1)$. Note that under our assumptions $h(N)$ simply equals the number of reticulation nodes.

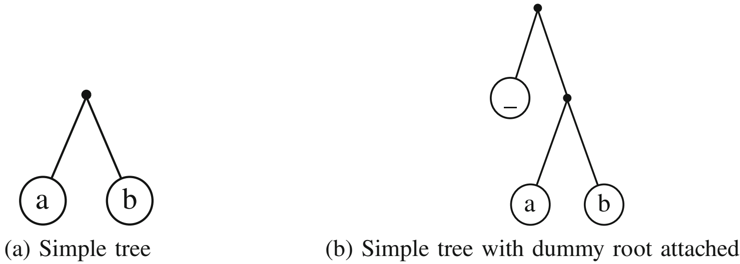
Suppose we are given a set of K phylogenetic trees $T_1, T_2, \ldots, T_k$ over the same set of taxa. The *minimal hybridization network* for that set of trees is a network $N_{\min}$ that displays each tree and has the smallest hybridization number possible. Note that there can be several networks with equal hybridization number.

The most parsimonious hybridization network problem is defined as follows: given a set of phylogenetic trees $T_1, T_2, \ldots, T_k$ over the same set of taxa, construct the minimal hybridization network for this set of trees.

It has been shown that even for the case of $k = 2$ the construction of such a network is an NP-complete problem [3]. As far as we know there exists only one algorithm for the construction of the most parsimonious hybridization network [14].

3 Algorithm

The main idea of our algorithm is to iterate over possible values of the hybridization number and to construct and solve a Boolean formula that represents a



(a) Simple tree (b) Simple tree with dummy root attached

Fig. 3. An illustration of an attachment of the dummy root to the tree.

hybridization network with this hybridization number. We implemented our algorithm in a software tool PhyloSAT which is available for download at GitHub[1].

3.1 Pre-processing

Before the actual Boolean formula encoding we modify the input and split it into several tasks to reduce the size and the complexity of the problem. We do this according to the rules from [2]. To define these rules we first need to define the term *cluster*: a set of taxa A is a cluster in trees $T_1, T_2, \ldots, T_k$ if there exists a node in each tree with the set of leaves of its subtree equal to the set A. The reduction rules are as follows:

1. **Subtree reduction rule:** replace every subtree which is present in all input trees with a leaf with a new label.
2. **Cluster reduction rule:** for each cluster A replace the subtrees containing it with a leaf with a new label and add a new task for processing which consists of deleted subtrees $T'_1, T'_2, \ldots, T'_k$ with leaf set A.

After splitting the task into a set of simpler tasks, we add a dummy root to each tree of each task along with the new dummy leaf for tree consistency. Figure 3 illustrates the procedure. This is done to ensure that all the trees in the input share a common root. This dummy root will be deleted on the post-processing stage. At this stage we have a set of tasks that will be solved separately and then their results will be merged at the post-processing stage.

3.2 Search of the Minimal Hybridization Number

To solve a subtask we need to find the lowest hybridization number k such that there exists a hybridization network with this hybridization number. To do this we use downwards search, i.e. we iterate through possible values of k starting from the highest and construct a Boolean formula corresponding to the current k.

[1] https://github.com/ctlab/PhyloSAT.

We decrease k until the solver cannot satisfy the formula, and this means that the previous value of k was the lowest possible.

There are other strategies of searching the minimal value of k. For example, we can start from zero and increase the value of k until the solver will be able to satisfy the formula, or we can use the binary search. The results of the binary search were close to the ones of downwards method, but in some cases, when the binary search tried to satisfy formulae with values of k less than minimal possible hybridization number, its results were also poor. This can be explained by an experimental observation that it is usually easier for the solver to produce an answer if the formula is satisfiable than if it is not. In case of an unsatisfiable formula the solver must check every possible answer to ensure that there is no solution; this is not needed if the formula is satisfiable. Because of this the results of the upwards search were poor. An obvious method for reducing the search time is to limit the range of possible values of k by using different heuristics to find close upper and lower bounds for k. Possible candidates are PIRN_{CH} [14], RIATA-HGT [10] and MURPAR [11]. We do not consider such optimizations in this paper.

3.3 Encoding the Boolean Formula

Having a set of trees T over a set of taxa A and a fixed hybridization number k, we will construct a Boolean formula which is satisfiable iff there exists a hybridization network that displays each tree in T and its hybridization number equals to k. To do this we first notice that a network over the set of taxa of size n with hybridization number k will have $2(n+k)-1$ nodes. As we add a dummy root and a dummy leaf, we finally have $2(n+k)+1$ nodes, k of which are reticulation nodes, $n+1$ are leaves and others are usual tree nodes.

We enumerate all the nodes in such a way that leaves are numbered in range $[0, n]$, regular nodes have numbers in range $[n+1, 2n+k]$ and all reticulation nodes have numbers in range $[2n+k+1, 2(n+k)]$. We also assume that the number of any leaf or regular node is less than the number of its parent. This is done to avoid consideration of isomorphic networks during SAT solving. Such enumeration allows us to define the following sets of nodes for each node v: $PC(v)$ is the set of possible children of v, $PP(v)$ is the set of possible parents of v and $PU(v)$ is the set of possible ancestors of v. Also let R be the set of reticulation nodes, L be the set of leaves and V be the set of regular nodes. Now we will describe variables and clauses required to construct the Boolean formula.

Network Structure Encoding. First, we encode the structure of the network. We introduce the following literals.

1. $l_{v,u}$ and $r_{v,u}$ for each $v \in V, u \in PC(v) : l_{v,u}$ (or $r_{v,u}$) is true iff regular node v has node u as its left (right) child.
2. $p_{v,u}$ for each $v \in L \cup V \backslash \{\rho\}, u \in PP(v) : p_{v,u}$ is true iff u is the parent of a regular node v.

3. $p^l_{v,u}$ and $p^r_{v,u}$ for each $v \in R, u \in PP(v) : p^l_{v,u}$ (or $p^r_{v,u}$) is true iff u is the left (right) parent of a reticulation node v.
4. $c_{v,u}$ for each $v \in R, u \in PC(v) : c_{v,u}$ is true iff u is a child of a reticulation node v.

This takes $O((n + k)^2)$ literals for specifying the network structure, and by noticing that $k < n$ we have $O(n^2)$ literals.

To encode the uniqueness of parents and children we tried to use an obvious pairwise encoding that requires $O(n^2)$ clauses for each node as well as more efficient Bimander encoding [6]. However, the Bimander encoding gave no speed boost because of the existence of other constraints that require $O(n^2)$ clauses per node. Thus we will describe the pairwise encoding for simplicity. For example consider parent variables. We state that a node can have at least one parent and at most one parent. For parents of the regular node v this can be expressed in the following way:

$$\left(\bigvee_{u \in PP(v)} p_{v,u}\right) \wedge \left(\bigwedge_{i,j \in PP(v); i<j} (p_{v,i} \rightarrow \neg p_{v,j})\right).$$

Using this pattern, we add the uniqueness constraints for literals l, r, p, p^l, p^r and c. These clauses are defined in Sects. 1–4 of Table 1. We also add constraints to order the number of children of regular nodes and parents of reticulation nodes. They are listed in Sect. 5 of Table 1.

A network is consistent if for every pair of nodes the parent relation implies the child relation and vice versa. Thus we add constraints that connect parent literals with children literals for all the types of nodes. See Sects. 6–9 of Table 1 for these clauses. The last step of the network construction is to deal with the enumeration around reticulation nodes. To do this, we add constraints to fix relative numbers of children and parents of reticulation nodes. They are defined in Sect. 10 of Table 1.

Since we need $O(n^2)$ clauses for each node to represent the uniqueness of its parents and children and $O(n)$ clauses for each node to represent the parents-children relation, we finally get $O(n^3)$ clauses in total to represent the network structure.

Mapping Trees to the Network. To express that the network contains all the input trees we add literals that represent the mapping of the tree nodes to the network nodes.

1. $x_{t,v_t,v}$ for each $t \in T, v_t \in V(t), v \in V : x_{t,s,v}$ is true iff regular node v represents node v_t from tree t, i.e. x literals represent injective mapping of network nodes to tree nodes. An example of such mapping is shown in Fig. 4. Note that leaves of the trees are bijectively mapped to leaves of the network thus there is no need to introduce x variables for them.

Table 1. Clauses for network structure encoding.

	Clause	Range
1.1	$p_{v,u_1} \vee \cdots \vee p_{v,u_k}$	$v \in V; u_1 \ldots u_k \in PP(v)$
1.2	$p_{v,u} \rightarrow \neg p_{v,w}$	$v \in V; u, w \in PP(v)$
2.1	$l_{v,u_1} \vee \cdots \vee l_{v,u_k}$	$v \in V; u_1 \ldots u_k \in PC(v)$
2.2	$r_{v,u_1} \vee \cdots \vee r_{v,u_k}$	
2.3	$l_{v,u} \rightarrow \neg l_{v,w}$	$v \in V; u, w \in PC(v)$
2.4	$r_{v,u} \rightarrow \neg r_{v,w}$	
3.1	$c_{v,u_1} \vee \cdots \vee c_{v,u_k}$	$v \in R; u_1 \ldots u_k \in PC(v)$
3.2	$c_{v,u} \rightarrow \neg c_{v,w}$	$v \in R; u, w \in PC(v)$
4.1	$p^l_{v,u_1} \vee \cdots \vee p^l_{v,u_k}$	$v \in R; u_1 \ldots u_k \in PP(v)$
4.2	$p^r_{v,u_1} \vee \cdots \vee p^r_{v,u_k}$	
4.3	$p^l_{v,u} \rightarrow \neg p^l_{v,w}$	$v \in R; u, w \in PP(v)$
4.4	$p^r_{v,u} \rightarrow \neg p^r_{v,w}$	
5.1	$l_{v,u} \rightarrow \neg r_{v,w}$	$v \in V; u, w \in PC(v) : u \geq w$
5.2	$p^l_{v,u} \rightarrow \neg p^r_{v,w}$	$v \in R; u, w \in PP(v) : u \geq w$
6.1	$l_{v,u} \rightarrow p_{u,v}$	$v \in V; u \in V \cap PC(v)$
6.2	$r_{v,u} \rightarrow p_{u,v}$	
6.3	$p_{u,v} \rightarrow (l_{v,u} \vee r_{v,u})$	
7.1	$l_{v,u} \rightarrow (p^l_{u,v} \vee p^r_{u,v})$	$v \in V; u \in R \cap PC(v)$
7.2	$r_{v,u} \rightarrow (p^l_{u,v} \vee p^r_{u,v})$	
7.3	$p^l_{u,v} \rightarrow (l_{v,u} \vee r_{v,u})$	
7.4	$p^r_{u,v} \rightarrow (l_{v,u} \vee r_{v,u})$	
8.1	$c_{v,u} \rightarrow p_{u,v}$	$v \in R; u \in V \cap PC(v)$
8.2	$p_{u,v} \rightarrow c_{v,u}$	
9.1	$c_{v,u} \rightarrow (p^l_{u,v} \vee p^r_{u,v})$	$v \in R; u \in R \cap PC(v)$
9.2	$p^l_{u,v} \rightarrow c_{v,u}$	
9.3	$p^r_{u,v} \rightarrow c_{v,u}$	
10.1	$c_{v,u} \rightarrow \neg p^l_{v,w}$	$v \in R; u \in PC(v); w \in PP(v) : u \geq w$
10.2	$c_{v,u} \rightarrow \neg p^r_{v,w}$	

2. $d_{t,v}$ for each $t \in T, v \in R : d_{t,v}$ is true iff the left parent edge of reticulation node v is used to display tree t, i.e. it specifies direction of necessary parent to display current tree.
3. $u^r_{t,v}$ for each $t \in T, v \in R : u^r_{t,v}$ is true iff the child of reticulation node v is used to display tree t.
4. $u_{t,v}$ for each $t \in T, v \in V : u_{t,v}$ is true iff regular node v is used to display tree t.
5. $a_{t,v,u}$ for each $t \in T, v \in V, u \in PU(v) : a_{t,v,u}$ is true iff regular node u is an ancestor of node v and node u corresponds to such node u_t from tree t that all the edges on the path from u to v are contracted to a single edge of tree t, i.e. node u is the first node on the path from v to the root that is mapped to some node in tree t. We will say that node u is the direct parent of node v considering tree t. Note that node v is not necessary present in tree t. In Fig. 4 node u is the parent of all the nodes starting from node v considering tree t.

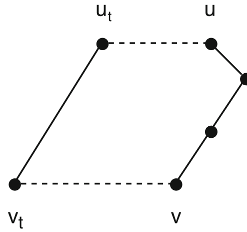


Fig. 4. An illustration of a piece of mapping of the nodes from the tree (on the left) to the nodes from the network (on the right). Nodes that are injectively mapped are connected by a dotted line. When displaying the tree all the edges in the path from u to v will be contracted to the single edge form u_t to v_t.

We have extra $O(tn^2)$ literals to match input trees to the network. Hence, we have $O(tn^2)$ literals in total.

We specify constraints for relations between network nodes and tree nodes. First, we define at-least-one and at-most-one constraints for literals x and a. Note that in case of x literals we have a restriction that at most one node from the tree corresponds to the network node, and that at most one node from the network corresponds to the tree node. These clauses are defined in Sects. 1–2 of Table 2. Because of dummy roots we know that roots of input trees are mapped to the root of the network so we have $x_{t,\rho_t,\rho}$ set to be true for every tree. Also note that $x_{t,v_t,v}$ implies $u_{t,v}$ by definition. See Sect. 3 of Table 2 for these clauses.

Furthermore, if we know that node v is a leaf and its parent u corresponds to node u_t from tree t, then we can conclude that $a_{t,v,u}$ is true i.e. u is the direct parent of v in tree t. The same observation can be done for non-leaves, i.e. if we know that v corresponds to v_t and another node u corresponds to u_t and u_t is parent of v_t then u is the direct parent of v considering tree t. On the other hand if we know that v corresponds to v_t and we know that u is the direct parent of v considering tree t, then u should correspond to parent of v_t. We also should take care of enumeration, so we add a constraint stating that if node u_t is the parent of the node v_t, then the number of the corresponding node u should be greater than the one of the node v, i.e. $x_{t,v_t,v} \to \neg x_{t,u_t,u}$ for each u and v such that $u < v$. These clauses are presented in Sect. 4 of Table 2.

We also add some heuristic constraints related to the trees' structure. Notice that the number of the node in the network cannot be less than the size of the subtree of the corresponding node v_t in tree t and also it cannot be greater than the size of the tree minus depth of v_t. Also if node v_t in tree t and node $v_{t'}$ in tree t' have disjoint sets of taxa in their subtrees then they cannot be mapped to the same node in the network. These clauses can be found in Sect. 5 of Table 2.

Next, we add constraints that connect child-parent relations in trees with indirect child-parent relations in the network. First, consider the regular node u that is the direct parent of the node v and u is used to display tree t then u is also the parent of v considering tree t. Vice versa, if we know that u is the parent of v in tree t, then u should be used for displaying that tree. If we do not use u for displaying tree t then we should share the information stored in the a

Table 2. Clauses for the mapping of the tree nodes to the network nodes.

	Clause	Range
1.1	$a_{t,v,u_1} \vee \cdots \vee a_{t,v,u_k}$	$v \in V \cup L \cup R; u_1 \ldots u_k \in PU(v)$
1.2	$a_{t,v,u} \rightarrow \neg a_{t,v,w}$	$v \in V \cup L \cup R; u, w \in PU(v)$
2.1	$x_{t,t_v,v_1} \vee \cdots \vee x_{t,t_v,v_k}$	$t \in T; t_v \in V(t); v_1 \ldots v_k \in V$
2.2	$x_{t,t_v,v} \rightarrow \neg x_{t,t_v,w}$	$t \in T; t_v \in V(t); v, w \in V$
2.3	$x_{t,t_v,v} \rightarrow \neg x_{t,t_w,v}$	$t \in T; t_v, t_w \in V(t); v \in V$
3.1	$x_{t,v_t,v} \rightarrow u_{t,v}$	$t \in T; v \in V; v_t \in V(t)$
3.2	$x_{t,\rho_t,\rho}$	$t \in T; \rho_t = \rho(t)$
4.1	$x_{t,u_t,u} \rightarrow a_{t,v,u}$	$t \in T; v \in L; u \in PP(v); u_t \in V(t)$
4.2	$(x_{t,v_t,v} \wedge x_{t,u_t,u}) \rightarrow a_{t,v,u}$	$t \in T; v \in V; u \in PP(v); v_t \in V(t) : u_t = p(v_t)$
4.3	$(x_{t,v_t,v} \wedge a_{t,v,u}) \rightarrow x_{t,u_t,u}$	
4.4	$x_{t,v_t,v} \rightarrow \neg x_{t,u_t,u}$	$t \in T; v \in V; u \in V; v_t \in V(t); u_t = p(v_t) : u < v$
5.1	$\neg x_{t,v_t,v}$	$t \in T; v \in V; v_t \in V(t) : v_t < \text{size}(\text{subtree}(v_t))$
5.2	$\neg x_{t,v_t,v}$	$t \in T; v \in V; v_t \in V(t) : v_t > \text{size}(t) - \text{depth}(v_t)$
5.3	$\neg x_{t,v_t,v} \vee \neg x_{t',v_{t'},v}$	$t, t' \in T; v \in V; v_t \in V(t); v_{t'} \in V(t')$: subtrees of t and t' have disjoint sets of taxa

variables between v and u because they will have the same parent considering tree t. We do this with the following constraints:

$$((p_{v,u} \wedge \neg u_{t,u} \wedge a_{t,u,w}) \rightarrow a_{t,v,w}) \wedge ((p_{v,u} \wedge \neg u_{t,u} \wedge a_{t,v,w}) \rightarrow a_{t,u,w}).$$

These clauses are listed in Sect. 1 of Table 3.

Now consider the reticulation node v and its parent u. If u is a reticulation node then we should share information about its parent considering tree t but only when direction of parent u matches direction specified in v. If u is a regular node and u is used for displaying tree t, then u is the direct parent of v considering tree t. On the other hand, if u is not used then we should share information about its parent considering tree t also only when the direction of u matches the direction specified in v. These constraints are presented in Sect. 2 of Table 3.

In cases when the direction of u does not match direction specified in v we should not use it. See Sect. 3 of Table 3 for these clauses. If u is a reticulation node, its direction matches direction specified in v and v is used, then u should also be used. Note that if we do not use the child of node v for displaying then we also should not use its parents. And the crucial point is that if the child of node v is a regular node then we should use v for displaying. These clauses are defined in Sect. 4 of Table 3.

We also add clauses to forbid incorrect numeration, if node v has a reticulation parent u then there cannot exist a node w that its number is less than the number of v, and $a_{t,u,w}$ is true. And for all w with numbers greater than v we add clauses to share information of a variables. See Sect. 5 of Table 3 for these clauses.

Again the most expensive clauses are clauses that represent uniqueness of variables a and x. We have $O(n^2)$ clauses for each node for each tree, so we have

Table 3. Clauses for translating child-parent relations from the trees to the network.

	Clause	Range
1.1	$(p_{v,u} \wedge u_{t,u}) \rightarrow a_{t,v,u}$	$t \in T; v \in V \cup L; u \in V \cap PP(v)$
1.2	$(p_{v,u} \wedge a_{t,v,u}) \rightarrow u_{t,u}$	
1.3	$(p_{v,u} \wedge \neg u_{t,u} \wedge a_{t,u,w}) \rightarrow a_{t,v,w}$	$t \in T; v \in V \cup L; u \in V \cap PP(v); w \in PP(u)$
1.4	$(p_{v,u} \wedge \neg u_{t,u} \wedge a_{t,v,w}) \rightarrow a_{t,u,w}$	
2.1	$(p^l_{v,u} \wedge d_{t,v} \wedge a_{t,u,w}) \rightarrow a_{t,v,w}$	$t \in T; v \in R; u \in R \cap PP(v); w \in PU(u)$
2.2	$(p^l_{v,u} \wedge d_{t,v} \wedge a_{t,v,w}) \rightarrow a_{t,u,w}$	
2.3	$(p^r_{v,u} \wedge \neg d_{t,v} \wedge a_{t,u,w}) \rightarrow a_{t,v,w}$	
2.4	$(p^r_{v,u} \wedge \neg d_{t,v} \wedge a_{t,v,w}) \rightarrow a_{t,u,w}$	
2.5	$(p^l_{v,u} \wedge d_{t,v} \wedge u_{t,u}) \rightarrow a_{t,v,u}$	$t \in T; v \in R; u \in V \cap PP(v)$
2.6	$(p^r_{v,u} \wedge \neg d_{t,v} \wedge u_{t,u}) \rightarrow a_{t,v,u}$	
2.7	$(p^l_{v,u} \wedge d_{t,v} \wedge \neg u_{t,u} \wedge a_{t,u,w}) \rightarrow a_{t,v,w}$	$t \in T; v \in R; u \in V \cap PP(v); w \in PU(u)$
2.8	$(p^l_{v,u} \wedge d_{t,v} \wedge \neg u_{t,u} \wedge a_{t,v,w}) \rightarrow a_{t,u,w}$	
2.9	$(p^r_{v,u} \wedge \neg d_{t,v} \wedge \neg u_{t,u} \wedge a_{t,u,w}) \rightarrow a_{t,v,w}$	
2.10	$(p^r_{v,u} \wedge \neg d_{t,v} \wedge \neg u_{t,u} \wedge a_{t,v,w}) \rightarrow a_{t,u,w}$	
3.1	$(p^l_{v,u} \wedge \neg d_{t,v}) \rightarrow \neg u^r_{t,u}$	$t \in T; v \in R; u \in R \cap PP(v)$
3.2	$(p^r_{v,u} \wedge d_{t,v}) \rightarrow \neg u^r_{t,u}$	
3.3	$(p^l_{v,u} \wedge \neg d_{t,v}) \rightarrow \neg u_{t,u}$	$t \in T; v \in R; u \in V \cap PP(v)$
3.4	$(p^r_{v,u} \wedge d_{t,v}) \rightarrow \neg u_{t,u}$	
4.1	$(p^l_{v,u} \wedge d_{t,v} \wedge u^r_{t,v}) \rightarrow u^r_{t,u}$	$t \in T; v \in R; u \in R \cap PP(v)$
4.2	$(p^r_{v,u} \wedge \neg d_{t,v} \wedge u^r_{t,v}) \rightarrow u^r_{t,u}$	
4.3	$\neg u^r_{t,v} \rightarrow \neg u^r_{t,u}$	
4.4	$\neg u^r_{t,v} \rightarrow \neg u_{t,u}$	$t \in T; v \in R; u \in V \cap PP(v)$
4.5	$c_{v,u} \rightarrow u^r_{t,v}$	$t \in T; v \in R; u \in (V \cup L) \cap PC(v)$
5.1	$p_{v,u} \rightarrow \neg a_{t,u,w}$	$t \in T; v \in V \cup L; u \in R \cap PP(v);$ $w \in PU(u) : w \leq v$
5.2	$(p_{v,u} \wedge a_{t,u,w}) \rightarrow a_{t,v,w}$	$t \in T; v \in V \cup L; u \in R \cap PP(v);$ $w \in PU(u) : w > v$
5.3	$(p_{v,u} \wedge a_{t,v,w}) \rightarrow a_{t,u,w}$	$t \in T; v \in V \cup L; u \in R \cap PP(v);$ $w \in PU(u) : w > v$

$O(tn^3)$ clauses to map trees to the network. This sums up to $O(tn^3)$ clauses in total.

3.4 Solving the Boolean Formula and Post-processing

To solve the generated Boolean formula we use a SAT-solver CryptoMiniSat[2]. Choosing the most appropriate solver is not considered in this paper, however, several experimentations were made by M. Bonet and K. John [2] so it is one of the topics of further research.

After solving a task we reconstruct the network from the SAT-solver output. After that we delete the dummy root and the corresponding leaf from the network. And when all the subtasks of the original task are solved, we merge their networks into a single hybridization network corresponding to the original task.

[2] http://www.msoos.org/cryptominisat4/.

4 Experiments

To test the performance of our algorithm we evaluated PhyloSAT on a grass (Poaceae) dataset provided by the Grass Phylogeny Working Group (2001). There are 57 test cases in the dataset with up to 47 taxa. All experiments were performed using a machine with an AMD Phenom II X6 1090 T 3.2 GHz processor on Ubuntu 14.04. All tests were run with a time limit of 1000 s. For comparison we also ran PIRN_C and PIRN_CH on the same test cases.

Out of 57 test cases, 9 were not solved even by heuristic algorithms in time. PhyloSAT was able to produce the optimal answer for 28 test cases. From these

Table 4. Experimental results.

Algorithm	Solved cases	Optimal solutions
PhyloSAT	48	39
PIRN_C	29	29
PIRN_CH	43	36

Table 5. Comparison of results of PIRN_CH and PhyloSAT on test cases with big hybridisation number. Runtimes in seconds are reported in brackets.

Test instance	PhyloSAT	PIRN_CH	Optimal solution
2NdhfPhyt	6 (9)	6 (6)	6
3NdhfPhytRpoc	8 (1000)	8 (28)	6
3PhytRbclRpoc	6 (11)	6 (3)	6
3RbclWaxyIts	6 (1000)	7 (4)	6
4NdhfRbclWaxyIts	7 (1000)	7 (35)	≥ 6
4PhytRbclRpocIts	9 (1000)	8 (377)	≥ 6
2RbclRpoc	7 (1000)	7 (42)	7
3NdhfWaxyIts	8 (1000)	8 (90)	≥ 7
3PhytRbclIts	11 (1000)	8 (120)	≥ 7
3PhytRpocIts	7 (1000)	7 (59)	7
4NdhfPhytRbclRpoc	10 (1000)	10 (287)	≥ 7
4NdhfPhytRpocIts	10 (1000)	-	≥ 7
2NdhfPhyt	8 (12)	-	8
2NdhfRbcl	8 (1)	8 (851)	8
2PhytIts	8 (41)	8 (372)	8
3NdhfPhytRbcl	9 (123)	-	9
2NdhfRpoc	9 (954)	9 (484)	9
3NdhfRbclRpoc	13 (1000)	-	≥ 10
3NdhfPhytIts	13 (1000)	-	≥ 11

28 test cases $PIRN_C$ was able to solve only 21 and in all of them hybridization number was less than 6 which shows that $PIRN_C$ is not capable of building hybridization networks with large hybridization number. In all test cases $PIRN_C$ was slower than PhyloSAT. Even $PIRN_{CH}$ did not solve 2 of these 28 test cases in time. $PIRN_{CH}$ was faster than PhyloSAT only on 5 test cases and was significantly slower on 2 test cases. Twelve more test cases were not solved by $PIRN_C$, but PhyloSAT was able to produce some (possibly non-optimal) network. $PIRN_{CH}$ did not solve 3 of these 12 cases in time, in 3 cases produced less optimal network than PhyloSAT, in 2 cases more optimal and in the rest 4 cases results were equal. From these 12 cases $PIRN_{CH}$ produced an optimal network for only 2, and PhyloSAT found an optimal network for 3 cases but was unable to prove their optimality. 8 more test cases had isomorphic trees in input, so they had an obvious answer of zero. Results of the simulation are summarized in Table 4. Also, the more precise results of comparison are in Table 5. We included only cases with hybridization number more than 6, because on less complicated tests results of all the algorithms were similar. The results of $PIRN_C$ are not included because it did not manage to solve any of these cases.

Experiments showed that in some cases PhyloSAT is able to find an optimal network but then it spends considerable time trying to prove its optimality and that time is much higher than reasonable limit. We can avoid wasting time on useless computation in cases when we find a network with hybridization number equal to the heuristic lower bound (like $PIRN_{CH}$ does). Besides, we found that it also costs much time to build a network with a big hybridization number when the minimal hybridization number is small. Thus, a close upper bound for the minimal hybridization number will also be useful and will save much computation time.

5 Conclusion

We have proposed an algorithm for constructing an exact parsimonious hybridization network from multiple phylogenetic trees. Experiments showed that PhyloSAT outperforms $PIRN_C$ in all cases and performs reasonably well comparing to the heuristic $PIRN_{CH}$. However, in the cases of large hybridization numbers search and construction of optimal network is still a very challenging problem. In the future we plan to use existing heuristics and estimations on lower and upper bounds of the hybridization number to limit search bounds and thus reduce the running time of our algorithm.

Acknowledgements. This work was financially supported by the Government of Russian Federation, Grant 074-U01. Authors would like to thank Igor Buzhinsky, Daniil Chivilikhin and Fedor Tsarev for helpful comments and conversations.

References

1. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Adv. Comput. **58**, 117–148 (2003)

2. Bonet, M.L., John, K.S.: Efficiently calculating evolutionary tree measures using SAT. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 4–17. Springer, Heidelberg (2009)
3. Bordewich, M., Semple, C.: Computing the minimum number of hybridization events for a consistent evolutionary history. Discret. Appl. Math. **155**(8), 914–928 (2007)
4. Chen, Z.Z., Wang, L.: Hybridnet: a tool for constructing hybridization networks. Bioinform. **26**(22), 2912–2913 (2010)
5. Heule, M.J.H., Verwer, S.: Exact DFA identification using SAT solvers. In: Sempere, J.M., García, P. (eds.) ICGI 2010. LNCS, vol. 6339, pp. 66–79. Springer, Heidelberg (2010)
6. Hölldobler, S., Nguyen, V.: An efficient encoding of the at-most-one constraint. Technical report, KRR Group 2013–04, Technische Universität Dresden, 01062 Dresden, Germany (2013)
7. Huson, D.H., Rupp, R., Scornavacca, C.: Phylogenetic Networks: Concepts, Algorithms and Applications. Cambridge University Press, New York (2010)
8. Morrison, D.A.: Introduction to Phylogenetic Networks. RJR Productions, Uppsala (2011)
9. Nakhleh, L.: Evolutionary phylogenetic networks: models and issues. In: Heath, L.S., Ramakrishnan, H. (eds.) Problem Solving Handbook in Computational Biology and Bioinformatics, pp. 125–158. Springer, Berlin (2011)
10. Nakhleh, L., Ruths, D., Wang, L.-S.: RIATA-HGT: a fast and accurate heuristic for reconstructing horizontal gene transfer. In: Wang, L. (ed.) COCOON 2005. LNCS, vol. 3595, pp. 84–93. Springer, Heidelberg (2005)
11. Park, H.J., Nakhleh, L.: MURPAR: a fast heuristic for inferring parsimonious phylogenetic networks from multiple gene trees. In: Bleris, L., Măndoiu, I., Schwartz, R., Wang, J. (eds.) ISBRA 2012. LNCS, vol. 7292, pp. 213–224. Springer, Heidelberg (2012)
12. Semple, C.: Hybridization Networks. Department of Mathematics and Statistics, University of Canterbury, New York (2006)
13. Wu, Y.: Close lower and upper bounds for the minimum reticulate network of multiple phylogenetic trees. Bioinformat. **26**(12), i140–i148 (2010)
14. Wu, Y.: An algorithm for constructing parsimonious hybridization networks with multiple phylogenetic trees. J. Comput. Biol. **20**(10), 792–804 (2013)

Author Index

Alekseyev, Max A. 3, 13
Alexeev, Nikita 3
AlKindy, Bassam 83

Bahi, Jacques M. 83
Burleigh, J. Gordon 97

Chaudhary, Ruchi 97
Couchot, Jean-François 83

Fernández-Baca, David 97

Guyeux, Christophe 83

Hamacher, Kay 53
Hoksza, David 41

Jager, Sven 53
Jansson, Jesper 109
Jiang, Shuai 13

Kalvala, Sara 25
Kriválk, Radoslav 41

Ladroue, Christophe 25

Melnik, Mikhail 141

Nguyen, Quan 126
Nielsen, Henrik 68

Parisod, Christian 83
Pologova, Anna 3

Rajaby, Ramesh 109
Roos, Teemu 126

Salomon, Michel 83
Schiller, Benjamin 53
Sønderby, Casper Kaae 68
Sønderby, Søren Kaae 68
Strufe, Thorsten 53

Ulyantsev, Vladimir 141

Winther, Ole 68