Combinatorial optimization algorithms for (pseudo)alignment in bioinformatics

Borozan, Luka

University of Zagreb

FACULTY OF SCIENCE
DEPARTMENT OF MATHEMATICS

Luka Borozan

Combinatorial optimization algorithms for (pseudo)alignment in bioinformatics

DOCTORAL THESIS

Osijek, 2021

University of Zagreb

FACULTY OF SCIENCE
DEPARTMENT OF MATHEMATICS

Luka Borozan

Combinatorial optimization algorithms for (pseudo)alignment in bioinformatics

DOCTORAL THESIS

Supervisors:
izv.prof.dr.sc. Domagoj Matijević
dr.sc. Stefan Canzar

Osijek, 2021

Sveučilište u Zagrebu

PRIRODOSLOVNO - MATEMATIČKI FAKULTET
MATEMATIČKI ODSJEK

Luka Borozan

Algoritmi kombinatorne optimizacije za (pseudo)poravnavanje u bioinformatici

DOKTORSKI RAD

Osijek, 2021.

Sveučilište u Zagrebu

PRIRODOSLOVNO - MATEMATIČKI FAKULTET
MATEMATIČKI ODSJEK

Luka Borozan

Algoritmi kombinatorne optimizacije za (pseudo)poravnavanje u bioinformatici

DOKTORSKI RAD

Mentori:
izv.prof.dr.sc. Domagoj Matijević
dr.sc. Stefan Canzar

Osijek, 2021.

No moon is there, no voice, no sound
of beating heart; a sigh profound
once in each age as each age dies
alone is heard. Far, far it lies...

- J. R. R. Tolkien, The Lay of Leithian, Epilogue

Acknowledgements

First and foremost, I hereby express my deepest gratitude to my mentors - Domagoj Matijević and Stefan Canzar for their never ending patience and support. The amount of knowledge that they have passed on to me has proven instrumental in both tackling the challenges of scientific research and completing my PhD studies. Whenever I was in the need of support and encouragement related to both private and personal matters they were there for me. For that they have my warmest thanks.

Special thanks go to Maria Spletter who did the laboratory work to acquire the data used in the experiments related to *Drosophila* and to experimentally verify some of our findings.

Next, I would like to thank my parents - Danko and Đula for providing me love and the means to go through my entire education up to and including the PhD studies. My siblings - Bartol and Paula deserve a special mention as well for they have offered much needed respite from my work. I would also like to extend my genuine gratitude to my friends Arijan, Mario, Krasimir and Gunars for being there when it was needed the most and for sharing so many great moments with me. My thanks also go to colleagues - Domagoj Ševerdija and Slobodan Jelić for sharing with me their experience and offering me help at every turn. It has made me become a better researcher, lecturer and person.

Finally, I hope that everyone who helped me along the way even a small amount feels I was a worthy investment.

Summary

The field of bioinformatics is a fast growing interdisciplinary field with a strong contribution from mathematics and computer science. This thesis will deal with mathematical problems and algorithmic challenges from that field. Its first focus will be the comparison of hierarchic structures, mainly phylogenetic trees, which is used to explain various biological processes such as the evolution of the species. We will study mathematical models and algorithmic techniques which quantify the distance between such structures as means of determining the similarities or dissimilarities between them. The focus will be given to formulating the problem based on matching in the context of integer linear programming. Our goal will be to find a novel solution which respects the ancestry relations defined by those hierarchical structures and is often overlooked in the current research. Our main result will be given in a form of a software tool - Trajan, which will be tested on both the real world and simulated data. The second focus of the thesis will come from the problem of sequencing the RNA molecule. It is a combinatorial process of reconstruction of the RNA molecule from short nucleotide sequences which is used to analyze the transcriptome of a biological sample. Many recent studies consider a problem of quantification and classification of unannotated splicing events which often occur due to the mutations caused by abnormal state of the organism, e.g. cancer. We will present another software tool, called fortuna, which brings together high accuracy and fast running times to the analysis of the alternative splicing events unlike any of the well established competitor tools.

Keywords: phylogenetic trees, Trajan, distance, integer linear programming, branch-and-cut method, clique constraints, RNA-Seq, fortuna, alternative splicing, alignment, quantification

Prošireni sažetak

Bioinformatika je interdisciplinarno područje koje spaja matematiku, računalnu znanost, biologiju, medicinu i inženjerske discipline s ciljem razvijanja matematičkih modela i algoritamskih tehnika koje pružaju uvid u mnoge biološke procese kao što su transkripcija i sinteza proteina unutar stanice ili evolucija, ali i genetske osnove bolesti i adaptacija, razlike i interakcija među jedinkama i populacijama i sl. Počelo se razvijati ranih 50-tih godina prošlog stoljeća uvođenjem računala u obradu podataka dobivenih sekvenciranjem proteina [28] koje su po prvi puta prikupili [88][89]. U ovoj disertaciji, bavimo se problematikom iz tog područja.

Naš prvi fokus je usporedba hijerarhijskih struktura, najviše filogenetskih stabala koja organiziraju biološke vrste u stablastu strukturu baziranu na evoluciji. Njihovi čvorovi mogu predstavljati i druge podatke kao što su podklonovi tumora nastali prilikom evolocije tumora [54]. Također, protein-protein interakcijske (PPI) mreže implicitno sadrže hijerarhijske strukture koje je moguće rekonstruirati koristeći se hijerarhijskim metodama klasteriranja [37]. Uspoređivanje filogenetskih stabala dobivenih različitim metodama rekonstrukcije može kvantificirati njihove sličnosti i pružiti uvid u simbiozu parazita i domaćina [51]. Najpopularnija udaljenost među stablima je Robinson-Fouldsova udaljenost [85] u pozadini koje leži sparivanje vrhova dva stabla čija su podstabla topološki identična. Moguće ju je efikasno izračunati u polinomnom vremenu, no ona pruža ograničen uvid "niske razlučivosti" u razlike između dva stabla. Nadalje, često nije u mogućnosti identificirati topološki slične strukture te je izrazito osjetljiva na vrlo male promjene u ulaznim podacima [17][71]. Naše istraživanje direktno se nadovezuje na [7] u kojem je predstavljena generalizacija Robinson-Fouldsove udaljenosti čiji je glavni cilj otklanjanje njezinih loših svojstava putem izračuna bijektivnog preslikavanja vrhova iz jednog stabla u drugo koje poštuje roditeljske odnose. Postoje i druge udaljenosti definirane među stablima od kojih neke [24][71][63][11][12] imaju loša svojstva ili su u praksi teške za izračunati [2]. Unatoč tome što je u [7] dokazano da je izračun generalizirane Robinson-Fouldsove udaljenosti NP-težak problem, u [45] je pokazano kako postoji efikasno rješenje za njezin izračun koje se bazira na paradigmi cjelobrojnog linearnog programiranja. Naš glavni znanstveni doprinos je definicija uvjeta koji uvelike smanjuju poliedar u kojem rješavač Trajan metodom grananja-i-rezanja traži optimalno rješenje. Ideja na kojoj se temelje naši uvjeti je pronalaženje skupa bridova između dva stabla koji maksimalno narušavaju roditeljske odnose metodom dinamičkog programiranja čiju dinamičku tablicu efikasno popunjavamo prolaženjem vrhovima stabala. Smatramo da dva brida $(x_1, y_1), (x_2, y_2)$ ne narušavaju roditeljske odnose ukoliko vrijedi da je x_1 predak od x_2 u

prvom stablu ako i samo ako je y_1 predak od y_2 u drugom stablu. Trajan smo testirali na simuliranim stablima iz uniformnog i Yuleovog modela [9], te na stvarnim filogenetskim stablima kojima je predočena evolucija zelenih algi [69] i biljka cvijetnjača [92].

U drugom dijelu ove disertacije bavimo se problemima koji dolaze iz područja sekvenciranja molekule RNA (RNA-Seq). To je postupak čitanja strukture molekule RNA u obliku kratkih lanaca nukleotida sastavljenih od molekula adenina, citozina, gvanina i timina u svrhu određivanja svojstava stanične molekule DNA koja sadrži genetske informacije instrumentalne za proces nasljeđivanja. Tijekom posljednjih dvadesetak godina, tehnologija za sekvenciranje molekule RNA se razvijala iznimno brzo. Metode koje sekvenciraju čitav ljudski genom unutar jednoga dana česta su pojava. Podatke dobivene sekvenciranjem (u obliku kratkih lanaca nukleotida) potrebno je poravnati s referentnim genomom, tj. odrediti mjesto u genomu s kojega je pročitan podatak, a za što se koriste specijalizirani računalni programi kao što su [32][66][67][14][78][94][96][39][53]. Kvantifikacija količine podataka ovisno o njihovoj lokaciji u genomu je važan proces koji nam daje uvid u stanje organizma čiji smo genetski materijal sekvencirali. Za istraživanja bolesti poput raka [48] ili autizma [38] od iznimne su važnosti oni lanci nukleotida koji su sekvencirani s mutiranih područja. Identifikacija i kvantifikacija tih podataka najčešće se vrši nakon poravnjavanja na referentni genom pomoću specijaliziranog softvera kao što je [62], čije je izvršavanje dugotrajno u praksi, ili pomoću heurističkih metoda niske preciznosti [96]. Naš doprinos u ovome području je efikasan i precizan program: fortuna. On pridružuje kratke lance nukleotida klasama ekvivalencije konstruirane na temelju proširene reference koja omogućuje identifikaciju i klasifikaciju do sada nepoznatih izrezivanja (alternativnih načina prepisivanja molekule DNA koji prethode sintezi proteina). Proces koji fortuna izvršava može se podijeliti u tri koraka: gradnja indeksa, poravnavanje i naknadna obrada. U prvom koraku fortuna nadopunjuje referentni genom koristeći jedan od tri dobro definirana skupa mogućih izrezivanja. Potom slijedi proces poravnavanja podataka dobivenih sekvenciranjem na prošireni referentni genom. U koraku naknadne obrade vrši se najbitniji proces dodjeljivanja podataka klasama ekvivalencije. Rezultate na simuliranim i stvarnim podacima usporedili smo s onima dobivenim pomoću nekoliko konkurentnih programa.

Ključne riječi: filogenetska stabla, Trajan, udaljenost, cjelobrojno linearno programiranje, metoda grananja i rezanja, RNA sekvenciranje, fortuna, alternativno izrezivanje, poravnavanje, kvantifikacija

Contents

Nomenclature

Throughout the thesis, we use the following notation:

$\wedge, \vee, \veebar$	Logical conjunction, disjunction and exclusive disjunction.
$(s), s_i$	Sequences and their indexed elements.
$\mathbb{R}_+$	As set of non-negative real numbers.
capital latin letters $A, B, \ldots$	Matrices and data structures.
$[a, \ldots, b]$	A segment of numbers ranging from a to b, or a path from vertex a to the vertex b in a graph.
lg	Logarithm with basis 2.

CHAPTER 1

Introduction

Bioinformatics, often referred to as computational molecular biology, is an interdisciplinary field bridging mathematics, computer science, biology, medicine and engineering with a common goal of developing mathematical models and algorithmic techniques which give an insight into various biological processes such as transcription and protein synthesis, evolution, genetic basis of disease and adaptation or the differences between organisms and populations. It begun its development in the late fifties of the last century by introducing computers into processing of the protein sequencing data [28] first acquired by [88][89].

Deoxyribonucleic acid (DNA) molecule was not believed to be the carrier of encoded genetic information before the result [56] published in 1952. At that time, not much was known about the structure of DNA and proteins were considered to be responsible for hereditary traits. A first gigantic step towards the understanding of DNA was taken in 1953 when the Nobel prize awarded work of Watson and Crick was published [101]. Though, it wasn't until 1968 that the genetic code was deciphered [102]. This was the spark that ignited the field of DNA sequencing, which was first achieved in 1977 [87] [74], albeit on a modest level.

Rapid advancement in both fields of computer science and molecular biology has made sequencing available to an increasing number of scientists. With the dawn of next-generation sequencing [41] at the beginning of the 21st century, sequencing data has become broadly available at a cost which was rapidly decreasing by orders of magnitude. Next-generation sequencing techniques sequence complementary DNA synthesized from a transcribed ribonucleic acid (RNA) molecule, instead of directly sequencing DNA. This procedure, often referred to as RNA-Seq, yields more information than the traditional DNA sequencing. Suddenly, the amount of data at our disposal overcame our ability to process it, thus shifting the bottleneck to the area of computer science. Through the course of the last decade, we have witnessed numerous algorithmic solutions which aim to decrease the time required for processing of the biological samples such as [30][14], some of which are the main topic of this thesis.

The merits of using computers in processing biological data have led to their adoption in

many biological disciplines, one of which was computational phylogenetics [46]. A problem of comparing evolutionary paths, by computing the distance between them, was formulated as a graph theory problem [85] and, as such, its algorithmic solution has been implemented on a computer. Since then, better metrics with more favorable properties have been introduced, e.g. [7], and the notion of computing a distance between various hierarchical structures has been applied to a multitude of problems stemming from the field of biology such as [82][3].

A more detailed overview of the history of the field of bioinformatics can be be found in [49].

1.1 Motivation and high level problem formulation

Edit distance [100] was introduced as a means of quantification of the dissimilarities between two sequences of characters (string) over some alphabet. The more operations it takes to transform one string, the greater the distance between them. Those operations allow us to insert an element, to remove it or to substitute it with another. Sequencing information of a biological sample, especially that procured by the means of the next-generation sequencing, often comes in a form of a large amount of relatively short strings which are being compared against a set of reference strings in order to obtain information on their origin and properties. That is, each sampled string is mapped to a substring of the reference to which its distance is the least. The configuration of the sampled strings and their distribution over the reference carry information which is of great importance for researchers such as those involved in cancer [48] or autism research [38]. A high rate of mutation of the genetic material, often associated to the abnormal conditions related to various diseases, makes determining the origin of the short strings using a reference a challenging computational task. Many software tools which process sequencing information exist, e.g. [14][30][53][96][94][78][67][66][39]. Their most common downside is that they are unable to identify novel genomic features sometimes caused by the mutations [14][30][53][94][78][67][66]. Some of those [30][67][66] take a costly postprocessing step done by tools such as [62] in order to identify novel features, while others, such as [96], suffer from a high degree of inaccuracy due to their heuristic approach. The aim of this thesis is to provide an algorithmic solution to this problem and to present an implementation of a tool which is able to identify the aforementioned novel features in a shorter amount of time than any of the aforementioned tools.

A generalization of the edit distance problem to the hierarchical tree structures has been given in [97][105]. Slightly adapted edit operations have been supplemented with a constraint which ensures the consistency with the ancestry relations implied by the tree structure. Thus, a distance which honors the tree topology has been defined. The impracticality of its computation, due to its time complexity, has provoked the definition of more distance functions such

as [85][24][71][63][11][12][2] all of which exhibit certain negative traits, some of which we will cover in the following chapter. In [7], a problem formulation, equivalent to that of tree edit distance but using integer linear programming, has been given. It was an important theoretical result, but still impractical to use for anything other than very small problem instances. The second goal in this thesis is to formulate a novel solution to the problem of tree edit distance and to present its implementation which can be used to solve medium-to-large problem instances in a reasonable amount of time.

1.2 Structure of the thesis

This thesis will be structured as follows. The next chapter, titled **Basic definitions**, will give general notions of the concepts which are necessary for the understanding of the problems introduced in this thesis. Specifically, we will briefly clarify the matter concerning asymptotic analysis, basic data structures, hashing, graph theory, linear and integer programming, dynamic programming, greedy algorithms and genome and phylogeny related terms stemming from the field of biology.

In the chapter **Trajan**, we introduce an algorithm which solves tree alignment problem and its implementation in a form of a software tool - Trajan. A brief section with additional definitions is given first in order to introduce tree alignment problem in more detail. Afterwards, a literature review section is given which gives an overview of the research area and the research gaps which we enclose. Afterwards, a notion of arboreal matching is introduced and an integer linear programming formulation which is used to solve it is given. The experiments section in which we test various theoretical and empirical aspects of Trajan is preceded by a section in which Trajan's algorithms and implementation details are listed and analyzed.

The next chapter deals with **fortuna**, a software tool used to classify and quantify various genomic features of a living organism. As was the case with Trajan, we give some additional definitions and a literature review prior to explaining the methodology used by fortuna. The requirements imposed by the downstream analysis procedures and the limitations of competitor tools have played a large role in fortuna's development. In the method section, we carefully explain the concepts behind fortuna with a large focus on meeting the aforementioned requirements. We present the concepts from the method section in the form of algorithms and give some implementation specific details in the implementation section. The final section of this chapter, experiments, presents the series of tests we have performed in order to asses the accuracy and speed of our software.

Finally, we summarize our results and contribution in the conclusion section.

CHAPTER 2

Basic definitions

In this chapter, we give definitions which are required for the understanding of the topics covered by this thesis.

Let S be a set, $D \subseteq \mathbb{N}$ and $s : D \to S$ a function which assigns an element of S to each element in D. Then, such s is called a **sequence** and D is called an **index set**. An element of S which is mapped to by $s(i), i \in D$ is denoted as s_i and is called an **element** of the sequence (s) with an **index** i. In this thesis, we will mostly deal with the sequences which are indexed by a finite set $\{1,\ldots,n\},\ n \in \mathbb{N}$. The length of a finite sequence s will be denoted as $|(s)|$. In most programming languages, such finite sequences are often represented by a sequence of contiguous memory locations called **arrays**. Next, we give a definition of a distance function.

Definition 1. Let S be a set and $f : S \times S \to \mathbb{R}_+$ a non-negative function. If f satisfies

- $\forall x \in S,\ f(x,x) = 0$,
- $\forall x,y \in S,\ f(x,y) = f(y,x)$,
- $\forall x,y,z \in S, f(x,z) \leq f(x,y) + f(y,z)$,

then we call f a **distance function** (**metrics**).

Let $a \in \mathbb{R}^n$ be a non-zero vector and $c \in \mathbb{R}$ a scalar. Set $\left\{x \in \mathbb{R}^n : a^T x = c\right\}$ is called a **hyperplane**, while the set $\left\{x \in \mathbb{R}^n : a^T x \geq c\right\}$ is called a **halfspace**. A **polyhedron** in $\mathbb{R}^n$ is an intersection of a finite number of halfspaces defined as $\{x \in \mathbb{R}^n : Ax \geq b\}$ where $A \in \mathbb{R}^{n \times m}$ is a matrix and $b \in \mathbb{R}^m$ is a vector. Rows of A correspond to vectors a in the definition of a hyperplane, while the components of b correspond to the scalars c. Finally, we say that a set S is **bounded** in $\mathbb{R}^n$ if there exists a point $z \in \mathbb{R}^n$ and a constant $k > 0$ such that for every $x \in S$ it holds that $|x - z| \leq k$.

A very important notion for us will be that of a convex set. We say that $S \subseteq \mathbb{R}^n$ is a **convex set** if for any $x,y \in S$ and any $\lambda \in [0,1]$ set S contains $\lambda x + (1 - \lambda y)$. Let $x_1,\ldots,x_k \in \mathbb{R}^n$ be

vectors and $\lambda_1, \ldots, \lambda_k \in \mathbb{R}_+$ be scalars such that their sum is equal to 1. Then the vector

$$x = \sum_{i=1}^{k} \lambda_i x_i$$

is a **convex combinations** of the vectors x_i with scalars λ_i. A set of all convex combinations of vectors x_i is called a **convex hull**. The following theorem gives some basic results connecting convexity and polyhedra, whose proof can be found in [8].

Theorem 2.1 The following holds.

(a) Let (S) be a sequence of convex sets in $\mathbb{R}^n$. Then the intersection $\cap_{S\in(S)} S$ is also a convex set.

(b) Every polyhedron is a convex set.

(c) Let $S \subseteq \mathbb{R}^n$ be a convex set and (x) a finite sequence of its elements. Then any convex combination of elements in (x) is also in S.

(d) Let (x) be a finite sequence of vectors in $\mathbb{R}^n$. Then the convex hull of (x) is a convex set.

Knowing that a polyhedron P is a convex set, we may define its **extreme points** as vectors $x \in P$ such that there do not exist vectors $y, z \in P,\ y \neq z$ and a scalar $\lambda \in [0,1]$ such that $x = \lambda y + (1-\lambda)z$.

A sequence of vectors $x_1, \ldots, x_k \in \mathbb{R}^n$ is said to be linearly independent if for all scalars $\lambda_1, \ldots, \lambda_k$ the expression $\sum_i \lambda_i x_i = 0$ implies that all $\lambda_i = 0$, that is, no vector x_j can be represented as a linear combination of the rest of the vectors x_i.

Similarly to a convex set, a function $f : D \subseteq \mathbb{R}^n \to \mathbb{R}$ is a **convex function** if

$$\forall x, y \in D, \forall \lambda \in [0,1], \quad f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda) f(y).$$

2.1 Asymptotic analysis

Two algorithms solving the same problem may not run for the same amount of time on a computer. Consider an example in which our goal is to calculate the n-th number of the Fibonacci sequence given recursively as $F_1 = F_2 = 1$ and $F_n = F_{n-1} + F_{n-2}$. First solution would be to directly implement the recursive formula. In order to compute F_n for some fairly large n (e.g. $n = 10000$), this procedure takes a long time even on the modern processors. On the other, hand a procedure which computes F_n using the following procedure, although seemingly more complicated, terminates in no time.

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}$$

In order to determine how does the input size (n in our example) determine the running time of an algorithm and to compare different algorithms, we will be considering their **asymptotic** efficiency. Running time will be presented as a function $T(n)$ which considers the worst case scenario for the algorithm.

In order to illustrate the worst case scenario, lets consider a problem of searching an integer a in a sequence s of random integers such that $|(s)| = n$. Since (s) contains random integers, we have to traverse through its elements starting from the first index and compare them to a. In the best case scenario, first element we inspect will be equal to a and our search will end. In the worst case scenario where $a \notin (s)$, we will have to inspect all n elements of (s) to conclude that a is not present in it. Therefore, for an input size of n, we would say that the running time of that algorithm is $T(n) = n$ in the worst case.

In many cases, for a given algorithm, we will not be able or will not need to determine its exact running time. It is more common to consider input sizes large enough such that the running time is dominated by its asymptotically largest terms. For example, the dominant term of the expression $n^2 + 10^3 n + 10^5$ is n^2. If $n \geq 10^4$, it is larger than other terms by, at least, an order of magnitude.

If we wish to describe the running time of an algorithm with regards to the input size but regardless of its properties, it is common to use the **Θ-notation**. Let $g(n)$ be a function. We define a set of functions $\Theta(g(n))$ as

$$\{f(n) \ : \ \exists c_1, c_2, n_0 > 0 \ s.t. \ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \ \forall n \geq n_0\}.$$

If a function $f(n)$ belongs to the set $\Theta(g(n))$, as a slight abuse of the notation, we write $f(n) = \Theta(g(n))$. For example, let $f(n) = n^2 + 2n + 2$. There exist constants $c_1 = 1, c_2 = 5, n_0 = 1$ such that

$$0 \leq 1 \cdot n^2 \leq f(n) \leq 5 \cdot n^2$$

for all $n \geq 1$, so we say that $f(n) = \Theta(n^2)$. By using the Θ-notation, we have asymptotically bounded a function both from above and below. In many cases it is useful to provide only the lower or upper bound using, respectively, **Ω- or $\mathcal{O}$-notation**.

Similar to the Θ-notation, we say that $f(n) = \Omega(g(n))$ if $\exists c, n_0 > 0$ such that

$$0 \leq c \cdot g(n) \leq f(n), \ \forall n \geq n_0$$

and $f(n) = \mathcal{O}(g(n))$ if $\exists c, n_0 > 0$ such that

$$0 \leq f(n) \leq c \cdot g(n),\ \forall n \geq n_0.$$

Usually, using the Ω-notation we express our running time in the terms of the best case scenario, while $\mathcal{O}$-notation is used to express the worst case, even thought both sets contain a broader class of functions.

Matrix power operation in our Fibonacci sequence example can be done in $\mathcal{O}(\lg\ n)$ time because the multiplication of a constant sized matrix is considered to be $\mathcal{O}(1)$ (an atomary operation). Thus, the total complexity of the matrix power Fibonacci algorithm would be $\mathcal{O}(\lg\ n)$. On the other hand, a classic recursive implementation has a running time of $\mathcal{O}(2^n)$ which makes it asymptotically inferior.

Asymptotic analysis can be generalized to functions of multiple integer (or real variables) by considering the asymptotically fastest growing terms for each of them. More about the asymptotic analysis of the time complexity and its intricacies can be found in [23].

There are a few important classes of algorithms based on their running time. For example, polynomial algorithms are those that run in $\mathcal{O}(n^k)$ time, where k is a positive constant, and are considered faster than the algorithms with an exponential running time $\mathcal{O}(2^n)$. A class of algorithms which can be solved in polynomial time is often abbreviated to as **P**. Another important class of problems, abbreviated to as **NP** (non-deterministic polynomial time), is the class of problems for which a potential solution's correctness can be verified in polynomial time. To this date, the relation between classes P and NP is unknown, although it is speculated that P $\neq$ NP. A problem is considered to be **NP-complete** if any other NP problem can be reduced to (transformed into) it in polynomial time. An example of the reduction can be found in [77]. An important class of problems for us is the **NP-hard** class ("harder than NP"). It contains problems to which any NP problem can be reduced to in polynomial time, but their potential solutions can not be verified in polynomial time.

2.2 Data Structures

In this section, we are going to briefly introduce several data structures which were essential to the main algorithms used in this thesis.

An **array** is the most basic data structure, often implemented on a hardware level. It represents a contiguous sequence of registers in the random access memory (RAM) which may be read or written to by providing an address of its first element and an offset using which we can

reference its subsequent elements. Reading from or writing to an array element is a constant-time operation ($\mathcal{O}(1)$).

A **stack** is an abstract data structure that supports two operations - **push** which places an item on its "top" and **pop** which removes the top element from the stack and returns it. We are going to need it to introduce the concepts graph traversal in Section 2.4 and to keep track of the problems Trajan is solving presented in Section 3.4.2. The principle on which it operates is often called LIFO (last in, first out) due to its ability to pop an item which has been pushed last. The usual implementation of a stack uses an array (sometimes a vector or a linked list) as an underlying data structure. A possible implementation is given in Algorithm 1. All stack operations presented in the algorithm have $\mathcal{O}(1)$ complexity.

Algorithm 1 A simple stack implementation with an array A as its underlying structure. Let n be the maximum capacity of A and k an integer pointing to the current top of the stack. Initially, k is set to the first element of the array with an index 0. Procedure peek returns the top element of the stack without removing it.

```
1:  procedure PUSH(S, e)
2:      if k = n − 1 then
3:          error "overflow"
4:      end if
5:      A[k] = e
6:      k = k + 1
7:  end procedure
8:
9:  procedure POP(S)
10:     if k = 0 then
11:         error "underflow"
12:     end if
13:     k = k − 1
14:     return A[k]
15: end procedure
16:
17: procedure PEEK(S)
18:     if k = 0 then
19:         error "empty"
20:     end if
21:     return A[k − 1]
22: end procedure
```

Another abstract data structure we are going to consider is a **queue** which will be solely used in section 2.4 as an underlying structure of a graph traversal algorithm. It provides the user with two operations - **enqueue** and **dequeue** which follow the FIFO (first in, first out) principle. Queue places items to its "bottom", while it removes them from its "top". Similar to

that of a stack, queue is implemented with the same underlying data structures and a possible implementation is given in Algorithm 2. Due to the usage of modulo operation and two pointers, f and b, queue can maintain its maximum theoretical capacity after an arbitrary amount of operations without increasing their complexity beyond $\mathcal{O}(1)$.

Algorithm 2 A simple queue implementation with an array A as its underlying structure. Let n be the maximum capacity of A, f an integer pointing to the current front of the queue and b to its back. Initially, $f = b = 0$. Procedure front returns the front element of the queue without removing it, while the procedure back returns the last element in the queue without removing it.

```
1:  procedure ENQUEUE(Q, e)
2:      if (f = (b+1)%n) then
3:          error "overflow"
4:      end if
5:      A[b] = e
6:      b = (b+1)%n
7:  end procedure
8:
9:  procedure DEQUEUE(Q)
10:     if (f = b) then
11:         error "underflow"
12:     end if
13:     f = (f+1)%n
14:     return A[(f-1)%n]
15: end procedure
16:
17: procedure FRONT(Q)
18:     if (f = b) then
19:         error "empty"
20:     end if
21:     return A[f]
22: end procedure
23:
24: procedure BACK(Q)
25:     if (f = b) then
26:         error "empty"
27:     end if
28:     return A[(b-1)%n]
29: end procedure
```

2.3 Hashing

Hashing is a basic technique in computer science which allows us to compare character sequences faster than separately querying each of their consisting characters. It is widely used throughout the Chapter 4 to assist relatively complex procedures and as an integral part of several data structures.

A sequence of characters is commonly known as a **string**. Let (a) and (b) be strings. In order to compare two strings we would have to iterate through their characters (elements) and compare their distinct values. That operation takes $\mathcal{O}(min(|(a)|,|(b)|))$ time. Now lets consider a set of strings $S = \{(s_1),\ldots,(s_m)\}$ all of length n. We want to do a pairwise comparison of all strings in S. Basic combinatorics suggest that we have to do approximately m^2 comparisons with a complexity of $\mathcal{O}(n)$. Total complexity of the procedure would then be $\mathcal{O}(m^2 n)$. If we were to efficiently assign a unique integer for every different string, each comparison could be done in constant ($\mathcal{O}(1)$) time. The total complexity of the procedure then would be $\mathcal{O}(m^2 + m \cdot k)$, where k is the complexity of "converting" each of the m strings into integers. As long as k is asymptotically less complex than $m \cdot n$, our procedure would be better, in terms of running time, than the original one.

A function which maps a string into an integer is called a **string hash function**. Technically, a constant function would make for a valid hash function, but it wouldn't be usable in practice. Let us give an example of a valid hash function. We start by selecting a integer constant c and p a prime number. Then we define a hash function f which codomain is a segment $[0, c-1]$ with

$$f((s)) = \sum_{i=1}^{|(s)|} (s_i \cdot p^i) \bmod c.$$

Since string hashing using the function f can be done in $O(|(s)|)$ time, the total time complexity in our previous example would be $\mathcal{O}(m^2 + m \cdot n)$.

Note that we are mapping an infinite set onto a finite one. In practice, the choice of large enough c and p reduces the probability of collisions. Coupled with an array structure, a hash function may be used to form a data structure known as **hash table** which has different ways of solving the collisions. More about hash tables may be found in [23].

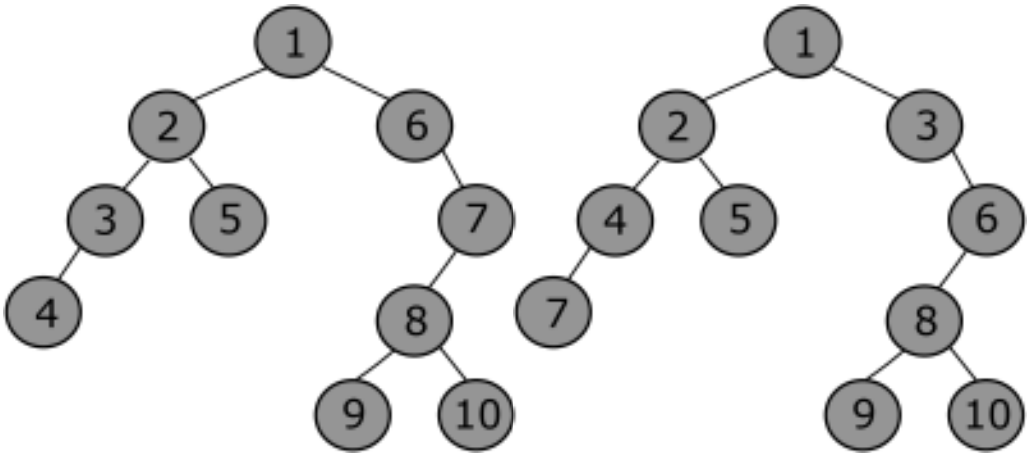
2.4 Graph theory

A plethora of problems in computer science may be formulated as problems in graph theory where graphs often represent relationships amongst the data, the stages of the computation, etc. We are going to heavily rely on it in both of the major topics in this thesis.

A **graph** G is considered to be an ordered pair of sets of **vertices** V (nodes) and **edges** $E \subseteq V \times V$. An edge $e = (s,t) \in E$ connects vertex s to vertex t. In that case, we say that t is **adjacent** to s. A set of all adjacent vertices of some vertex is called a **neighborhood**. The number of edges that are incident to a vertex is called a **degree** of that vertex. If for every $(s,t) \in E$, (t,s) is also in E, we call graph G **undirected**. Otherwise, we call G a **directed** graph. We say

that vertex t is **reachable** from s if there exists a sequence of vertices $(s) = s_1, \ldots, s_n$ such that $s_1 = s, s_n = t$ and $(s_i, s_{i+1}) \in E, \forall i$. Such (s) is called a **path** from s to t in G and is denoted as $[s,t]$. If, for every two vertices $s,t \in V$, there exists a path from s to t or from t to s, we say that G is **connected**. A path starting and ending in the same vertex is called a **cycle**. Graphs which do not contain cycles are called **acyclic**. A **clique** is a graph in which every two vertices have an edge between them.

A graph in which any two vertices are connected by exactly one path is called a **tree**. If one vertex of a tree is designated as a **root**, the tree is then called a **rooted tree**. Tree nodes which are of a degree 1 are called **leaves**, while other non-root vertices are designated **internal**. Any vertex of a tree induces a **subtree** consisting of all its ancestors, in which it has a role of a root node. In an **ordered tree** all children vertices have specified ordering. An example of a rooted tree can be seen in Figure 2.1. A tree is called a **binary tree** if all of its vertices have at most two children, while all non-leaf vertices of a **complete binary tree** have exactly two children.



(a) DFS **(b)** BFS

Figure 2.1: A rooted tree with 10 vertices traversed by DFS and BFS algrithms in a, so called, preorder fashion.

In the following few paragraphs we are going to explain two important graph search algorithms - depth first search (**DFS**) and breadth first search (**BFS**). These algorithms traverse all vertices of a graph and may perform arbitrary operations on them. The algorithmic solutions to both problems presented in this thesis use them to some extent. They play an important role in the design of more complex algorithms. Let G be a graph with its vertices V which are arranged in an ordering $\sigma = (s_1, \ldots, s_n)$ and edges E.

DFS algorithm, as presented in Algorithm 3, traverses the graph one path at a time. We say that σ is a **depth first ordering** if it was produced by the DFS algorithm. It can be characterized

by:

$$\forall 1 \leq i < j < k \leq n, \quad s_i \in N(s_k) \setminus N(s_j) \;\Rightarrow\; \exists s_m \in N(s_j) \text{ s.t. } i < m < j,$$

where $N(s)$ is the neighborhood of s.

Algorithm 3 Iterative DFS. G is a graph and r is an arbitrary root vertex.

```
1:  procedure DFS(G, r)
2:      S = new stack
3:      push r into S
4:      while S ≠ ∅ do
5:          pop v from S
6:          if v not discovered then
7:              set v as discovered
8:              for w ∈ N(v) in G do
9:                  push w to S
10:             end for
11:         end if
12:     end while
13: end procedure
```

Similarly, BFS algorithm which can be seen in Algorithm 4, traverses the graph by exploring nodes one depth level at a time. It will first explore all nodes adjacent to the arbitrary root node, then the nodes which are two edges away from it, etc. We say that σ is a **breadth first ordering** if it was produced by the BFS algorithm. It can be characterized by:

$$\forall 1 \leq i < j < k \leq n, \quad s_i \in N(s_k) \setminus N(s_j) \;\Rightarrow\; \exists s_m \in N(s_j) \text{ s.t. } m < i.$$

Both DFS and BFS algorithms have a complexity of $\mathcal{O}(|V| + |E|)$ which can be easily deducted from their pseudo codes. The main difference between the two algorithms is the usage of the underlying data structure: stack for DFS and queue for BFS. Also, in BFS vertices are set as discovered prior to them being placed in the queue, while the DFS vertices are set as discovered after they were removed from the stack. Figure 2.1 contains one possible DFS and BFS ordering on a rooted tree. Note that there exists a recursive variant of the DFS algorithm, given in Algorithm 5, which is commonly used in practice. When traversing trees using DFS or BFS, tracking vertex discovery is not needed since, due to the specific structure of a tree, each vertex will be placed in the data structure exactly once. If a graph is not connected, searching it from an arbitrary root will yield an ordering only for a subset of all vertices.

Algorithm 4 Iterative BFS. G is a graph and r is an arbitrary root vertex.

```
1:  procedure DFS(G, r)
2:      Q = new queue
3:      set r as discovered
4:      enqueue r into Q
5:      while Q ≠ ∅ do
6:          dequeue v from Q
7:          for w ∈ N(v) in G do
8:              if w not discovered then
9:                  set w as discovered
10:                 enqueue w to Q
11:             end if
12:         end for
13:     end while
14: end procedure
```

Algorithm 5 Recursive DFS. G is a graph and r is an arbitrary root vertex.

```
1: procedure DFS(G, r)
2:     set r as discovered
3:     for v ∈ N(r) in G do
4:         if v not discovered then
5:             DFS(G, v)
6:         end if
7:     end for
8: end procedure
```

2.5 Linear and integer programming

In this section we define linear programming (LP) and integer linear programming (ILP) problems and give a brief overview of the underlying theory used to obtain their solutions. The proofs of theorems and the definitions given in this section may be found in [8] and [23]. This topic is of utmost importance for us because in Chapter 3 we are going to use it to formulate and ultimately solve our problem.

Linear programming problems consider optimization (minimization of maximization) of a linear function, called the **objective function**, with the respect to a series of linear equalities or inequalities called **constraints**. They are very common in practice and have a wide variety of applications. A problem in which we maximize the objective function subject to a finite constraint set is called a **maximization problem**. Analogously, by minimizing the objective function, one may define a **minimization problem**. **Integer linear programming** problems further constrain the variable set to a subset of $\mathbb{Z}^n$.

Let $c \in \mathbb{R}^n$ be a constant parameter vector, called a **cost vector**. Objective function may be defined as a linear function $f : D \subseteq \mathbb{R}^n \to \mathbb{R}$ with $f(x) = c^T x$. The value of f in the point x is called the **objective value**. Let m be an integer. Constraints to which we subject our problem to are of the form $a_i^T x = b_i$, $a_i^T x \leq b_i$ or $a_i^T x \geq b_i$, where $i = 1, \ldots, m$, b_i is a scalar and a_i is a vector. Any x which satisfies all of the constraints is called a **feasible solution**, while the set of feasible solutions defined by the constraints is called a **feasible region**. A feasible solution such that either $f(x') \leq f(x)$ in a minimization problem or $f(x) \leq f(x')$ in a maximization problem for all feasible x' is called an **optimal solution**, while $f(x)$ is called an **optimal value**. A linear program which optimal objective function value is not finite, but the feasible region is non-empty, is called **unbounded**. If the feasible region is empty, then the linear program itself is called **infeasible**.

Note that all constraints which we introduced are either hyperplanes or halfspaces and that their intersection, feasible region, is a polyhedron. If a vector x satisfies $a_i^T x = b_i$ for some i, we say that the corresponding constraint is active at x. The constraints are said to be linearly independent if their corresponding vectors a_i are linearly independent. The following theorem presents the results which bring us closer to identifying the solution to the problem of linear programming.

Theorem 2.2 Let $x \in \mathbb{R}^n$ and I be a set of indices of active constraints at x. The following is then equivalent.

(a) There exist n linearly independent vectors in the set $\{a_i : i \in I\}$.

(b) Every element of $\mathbb{R}^n$ can be expressed as a linear combination of the vectors from the set

$\{a_i : i \in I\}$.

(c) The system of equations $a_i^T x = b_i,\ i \in I$ has a unique solution.

In order to give a characterization for the optimal solution, we will require the following definition which we will later relate to the polyhedra. Let P be a polyhedron defined by the constraints of a linear programming problem and $x \in \mathbb{R}^n$. Vector x is a **basic solution if** all equality constraints are active, and out of all active constraints, n of them are linearly independent. A basic solution which satisfies all constraints is called a **basic feasible solution**. The following theorem bridges the gap between the polyhedron defined by the constraints and the solution to the linear programming problem.

Theorem 2.3 Let P be a polyhedron and $x \in \mathbb{R}^n$. Then the following statements are equivalent.

(a) x is a vertex.

(a) x is an extreme point.

(a) x is a basic feasible solution.

In [8], it is concluded that, since the definition of an extreme point of a polyhedron does not depend on any of it's particular representations, the property of being a basic feasible solution should not depend on the representation of the linear programming problem. They also note that, given a finite number of linear constraints, the amount of basic feasible solutions is finite as well. As we will see in the later results, the existence of an optimal solution is closely tied to the existence of the extreme points of the polyhedron corresponding to the constraints. We say that a polyhedron $P \subset \mathbb{R}^n$ contains a line if there exists a vector $x \in P$ and a non-zero vector $d \in \mathbb{R}^n$ such that $x + \lambda d \in P,\ \forall \lambda$. Then, it is possible to prove the following theorem.

Theorem 2.4 Let polyhedron $P = \left\{x \in \mathbb{R}^n : a_i^T x \geq b,\ i = 1, \ldots, m\right\}$ be non-empty. Following statements are equivalent.

(a) P has at least one extreme point.

(b) P doesn't contain a line.

(c) There exist n vectors in the sequence $a_1, \ldots, a_m$ that are linearly independent.

Finally, we will give the results which, if possible, put the optimal solution in the place of one of the extreme points of P.

Theorem 2.5 Consider the minimization linear programming problem over a polyhedron P which has at least one extreme point and that there exists an optimal solution. Then, there exists an optimal solution which is an extreme point of P.

The previous theorem reduces the space in which we search our optimal solution to a set of finitely many extreme points. Finally, an even stronger result is given.

Theorem 2.6 Consider the minimization linear programming problem over a polyhedron P which has at least one extreme point. Then, there either exists an optimal solution which is an extreme point of P or the optimal cost is equal to $-\infty$.

Probably the most famous algorithm for solving linear programs is the simplex algorithm [25]. It runs in, theoretically, exponential time, while its implementations have been proven to be efficient in the state-of-the art solvers such as CPLEX or GUROBI. A very important part of the history of the field was the result from [61] in which a theoretical algorithm is constructed which solves linear programs in polynomial time. Contrary to the result for linear programming, integer linear programming problems are proven to be NP-complete [64].

2.6 Dynamic programming

Dynamic programming is a programming paradigm which solves optimization problems by dividing the problem into overlapping subproblems and combining their solutions. In Chapter 3, we are going to use this technique in order to tighten the polyhedron in which we are looking for a solution to our ILP problem. When developing a dynamical programming algorithm, as suggested by [23], we consider the following four points.

(1) Characterization of the optimal solution.

(2) Recursive definition of the optimal solution.

(3) Computation of the optimal value.

(4) Reconstruction of the optimal solution by backtracking through computed information.

The problems to which this paradigm is generally applied are the ones which exhibit the property of the optimal substructure, i.e., when the optimal solution contains optimal solutions to its subproblems. Usually, the solution to the problem solved by dynamical programming algorithms is the one which involves a choice in each of its steps. In each step a field in the data structure, called the **dynamic table**, is altered so that its new state reflects all choices we could have made while holding the information about all steps we have previously taken. Problems such as [72] [29] are examples of the problems which can be efficiently solved using dynamic programming. We will present, as an example, a simple solution to the lowest common subsequence problem.

Let (a) and (b) be strings representing a DNA sequence whose respective elements are characters A, C, G or T. We want to find a maximum-length subsequence (s) of both (a) and (b)

in order to quantify the similarity between the sequences. In [23], the optimal substructure of the problem has been proven, as has been formally given as the following theorem.

Theorem 2.7 Let (a) and (b) be the sequences such that $|(a)| = n, |(b)| = m$ and let (s) be the longest common sequence of (a) and (b) of length k.

(a) If $a_n = b_m$, then $a_n = b_m = s_k$ and $s_1, \ldots, s_{k-1}$ is the longest common subsequence of $a_1, \ldots, a_{n-1}$ and $b_1, \ldots, b_{m-1}$.

(b) If $a_n \neq b_m$, then $z_n \neq s_k$ and (s) is the longest common subsequence of $a_1, \ldots, a_{n-1}$ and (b).

(c) If $a_n \neq b_m$, then $b_m \neq s_k$ and (s) is the longest common subsequence of (a) and $b_1, \ldots, b_{m-1}$.

Now we may formulate a dynamic table D of size $(n+1) \times (m+1)$ which will contain the solution. Each element may be computed as

$$D[i,j] = \begin{cases} 0, & i = 0 \ \vee \ j = 0 \\ 1 + D[i-1, j-1], & a_i = b_j \ \wedge \ i, j \neq 0 \\ \max(D[i, j-1], D[i-1, j]), & a_i \neq b_j \ \wedge \ i, j \neq 0 \end{cases} .$$

Rows of D correspond to the characters in (a), while its columns correspond to the characters in (b). As we propagate through the table in a bottom-up manner, we are considering the longest common sequences found until the current element. If the characters in the current element match, we continue the subsequence. Otherwise, we extend the maximal length of the subsequence found so far consistently with Theorem 2.7. The length of the longest common subsequence will then be contained in the element $D[n,m]$. The algorithmic solution is presented in Algorithm 6.

Algorithm 6 Dynamic programming solution to the longest common subsequence problem.

```
1:  procedure LCS((a), (b))
2:      initialize table D[|(n)|+1, |(m)|+1]
3:      set all D[i,0] and D[0,j] to 0
4:      for i = 1,...,|(a)| do
5:          for j = 1,...,|(b)| do
6:              if a_i == b_j then
7:                  D[i,j] = 1 + D[i-1, j-1]
8:              else
9:                  D[i,j] = max(D[i-1, j], D[i, j-1])
10:             end if
11:         end for
12:     end for
13:     return D
14: end procedure
```

Table P in Algorithm 6 contains the information needed to backtrack through D and reconstruct the optimal solution. The backtracking algorithm is presented in Algorithm 7.

Algorithm 7 Backtracking the dynamic table of the longest common subsequence problem in order to print the solution.

```
1:  procedure LCS-B(D, (a), i, j)
2:      m = max(D[i−1, j−1], D[i−1, j], D[i, j−1])
3:      if i = j = 0 then
4:          return []
5:      else if m = D[i−1, j−1] then
6:          return append (a_i) to LCS−B(D, (a), i−1, j−1)
7:      else if m = D[i−1, j] then
8:          return LCS−B(D, (a), i−1, j)
9:      else
10:         return LCS−B(D, (a), i−1, j)
11:     end if
12: end procedure
```

It is easy to show, by tracking the amount of nested loops, that *LCS* procedure runs in $\mathcal{O}(n^2)$ time, while the backtracking algorithm's complexity is $\mathcal{O}(n)$.

2.7 Greedy algorithms

Greedy algorithms exploit the same overlapping supbroblem structure as the dynamic programming algorithms with one major difference. When presented with a choice, a greedy algorithm will not consider every possible option, but it will select one which is locally optimal. For that reason, it does not require a dynamic table as it propagates through states without any regard to the previous ones. As a result, greedy algorithms are more efficient both in terms of running time and memory consumption than their dynamic programming counterparts. Unfortunately, optimization problems which are optimally solvable using dynamical programming paradigm are not optimally solvable using greedy strategy. One such example is the lowest common subsequence problem presented in the previous section. Despite the aforementioned shortcomings, greedy algorithms will be used in Chapter 3 in order to speed up the computation towards the optimal solution of the problem. A class of problems which can be optimally solved using a greedy algorithm have a, so called, matroid structure and are explained in [23].

2.8 Genome related terms

The instructions for every feature of a living organism are contained within immensely complex molecules called DNA and RNA which are composed of smaller molecules called

nucleotides, a sugar called deoxyribose and a multitude of phosphate groups. It is one of the four types of macromolecules which are found in all life forms.

A great number of common terms in the field of bioinformatics are related to different genomic features shared across the vast majority of living organisms. A **genome** represented by a sequence of nucleotides encompassing the entirety of organism's genetic material is split into multiple DNA molecules called **chromosomes**. Common **nucleotides**, out of which DNA is comprised, are adenine (A), cytosine (C), guanine (G) and thymine (T). In RNA molecules, uracil (U) replaces thymine. On a double helix DNA, nucleotides residing on the two opposing strands are paired into **complementary pairs** A-T and C-G. Complementarity is essential for the replication of the DNA molecule (during cellular duplication) and transcription of DNA (decoding) into RNA. For a more detailed description of the underlying matter, refer to [1]. Chromosomal sequences for different species are usually stored in **FASTA files** which separate nucleotide sequences (belonging to different chromosomes) by their names. An example of a fasta file can be seen in Listing 2.1.

Listing 2.1: FASTA file example representing two chromosomial sequences - *N* and *M*

```
1 >chrN
2 ACTGATTACATATAA
3 CATAGAATGCTCAGC
4 ATGCATCTAGCTAGC
5 ...
6 ATCGATCGATCGATC
7 >chrM
8 ATGCATGCATGCAGT
9 ...
```

Transcription of RNA is a process which happens within individual cells that copies genomic information from a segment of a DNA molecule into a RNA molecule. It is initiated by the enzyme called RNA polymerase. In a case which is of particular interest for us, transcribed RNA molecules, often called **mRNA** or "messenger RNA", are used as templates for protein synthesis. Therefore, RNA molecules play an integral part in every living cell. More about the process may be found in [91].

Each chromosome is separated into regions called **genes** which encode all information necessary for transcription. Contiguous parts of genes which are directly involved in transcription are called **exons**, an acronym derived from the term "expressed region" [50]. It is considered that exons are the coding parts of DNA. A mechanism called RNA splicing separates exons from non-coding parts called **introns** or "intragenic regions" [50] and joins them together. A location on the gene where an intron was cut out is called a **splice site**. By convention, one strand of

DNA is called a **forward strand**, while the other is called a **reverse strand**. Genes starts and ends are represented with respect to their coding strand and their direction is either left-to-right on the forward strand or right-to-left on the reverse strand. When moving from the gene starting point to its end, every intron defines two **splice sites - donor and acceptor**, in that particular order.

Genes may contain a code for multiple mRNA molecules in the form of potentially overlapping regions called **transcripts** as can be seen in Figure 2.2. Similar proteins coded from those regions are called **isoforms**. Overlapping transcripts are a result of mechanisms that regulate **alternative splicing**, which defines exons and introns in multiple ways in the same genomic region, described in [15]. An alternative splicing event is considered to be **novel** if it is not a part of any annotated transcript. The abundance of different proteins in a cell is a direct consequence of transcript expression within it, that is, the quantity of alternatively spliced transcripts coding mRNA for those specific proteins. Analysis of transcript expression provides an insight into the state of the cell and organism alike.

Figure 2.2: An example of the alternative splicing found in gene CD1E on human chromosome 1. Transcripts are arranged horizontally and thick rectangular shapes represent exons. Image was produced by Integrated Genome Viewer.

The first step to determining the abundance of mRNA in a given sample is to employ the technology of **RNA sequencing**. It uses fast RNA sequencing techniques in oder to supply the evidence of the existence of different transcripts. Strands of RNA in the sample are sequenced into **short reads** - nucleotide sequences between 50 and 400 base pair long. In a process called **alignment**, short reads are then mapped to the **reference genome** or **reference transcriptome**, which can be obtained through the process of (”de novo”) **assembly**.

Assembly is a procedure which reconstructs the genome or transcriptome based on the short reads obtained from a sample as indicated by [58]. One of the challenges of this procedure is to determine the exact **genomic coordinates**, relative to the chromosome reference, from which short reads were sequenced. Afterwards, using the fact that only exonic regions are sequenced from mRNA, the problem is to determine their boundaries and supplement the gaps with introns in a way such that the transcript or gene boundaries are defined. A high amount of alternative

transcripts, overlaping exonic regions (such as ones seen in Figure 2.2) and possible sequencing errors make that step even more complicated. Some of the software tools used for assembly are [84], [52], [90], [80], [103], [21].

Once the reference has been assembled, it can be **annotated** using a **GTF file** which format can be found at [1]. An example of the GTF file, relevant for this thesis can be seen in the Listing 2.2. The annotation is used by alignment software such as [14][30] in order to rapidly map a large amount of short reads to the annotated genomic features. Alignment information has multiple uses in studying different stages in organism's development, its responses to external stimuli, detection of certain anomalies such as diseases and mutations, evolutionary paths of species, etc.

Listing 2.2: A part of a GTF file which annotates a single transcript, tab characters were replaced with spaces for readability. Transcript "XR_001737556.1" is located on the gene "WASH9P" on chromosome one, between the coordinates 184878 and 199860 on the reverse strand. It is composed of 6 exons whose information is listed below.

```
1 chr1  . transcript 184878 199860 . - . gene_id "WASH9P";
    transcript_id "XR_001737556.1";
2 chr1  . exon 184878 185350 . - . gene_id "WASH9P";
    transcript_id "XR_001737556.1"; exon_number "1"; exon_id "
    XR_001737556.1.1";
3 chr1  . exon 185491 185559 . - . gene_id "WASH9P";
    transcript_id "XR_001737556.1"; exon_number "2"; exon_id "
    XR_001737556.1.2";
4 chr1  . exon 186317 186469 . - . gene_id "WASH9P";
    transcript_id "XR_001737556.1"; exon_number "3"; exon_id "
    XR_001737556.1.3";
5 chr1  . exon 187129 187287 . - . gene_id "WASH9P";
    transcript_id "XR_001737556.1"; exon_number "4"; exon_id "
    XR_001737556.1.4";
6 chr1  . exon 187380 187577 . - . gene_id "WASH9P";
    transcript_id "XR_001737556.1"; exon_number "5"; exon_id "
    XR_001737556.1.5";
7 chr1  . exon 187755 187890 . - . gene_id "WASH9P";
    transcript_id "XR_001737556.1"; exon_number "6"; exon_id "
    XR_001737556.1.6";
```

Given a reference file and a FASTA or a **FASTQ file** (FASTA sequences supplemented with quality scores) containing short read sequences, an aligner software produces an output in a form of a **SAM file** which lists locations to which the reads have been mapped to. Note that a single read can be mapped to multiple locations or **multi-mapped**. An alignment which is

[1] https://mblab.wustl.edu/GTF22.html

considered the best by some scoring scheme is called a **primary alignment**, while all others are called **secondary alignments**. A complete reference to SAM file format can be found at [2]. A large number of aligners may also output a binary encoded SAM file called the **BAM file** which reduces the required storage space as plain text SAM files often exceed 100GB in size.

The increasing need to analyze large amounts of samples in short periods of time has pushed the computer scientists to consider new algorithmic approaches. A most notable class of aligners which often employ advanced data structures, hashing and other state of the art algorithms in order to speed up the read processing are called **pseudoaligners**. The most important property of a pseudoaligning software is that it (pseudo)aligns reads to the reference much faster than the traditional algorithms at the cost of the reduced accuracy. Such pseudoaligners are [14] [94] [96] [78].

An example of an alignment of a multimapped read can be found in the Listing 2.3. Read "ERR2902089.252943611" has been mapped to (human) chromosome 2 with a starting coordinate 227258887. Flag 16 indicates that the first alignment is considered to be primary on the reverse strand. Alignment with a flag 272 is a secondary alignment on the reverse strand. An interesting part of every alignment is the CIGAR string such as "119M2I30M". It indicates that 119 base pairs, starting from the starting coordinate were aligned to the reference, 2 base pairs were inserted and the next 30 bases were, again, a match. Letter "I" preceded by an integer n encodes insertion operation of n bases, "M" encodes match, "N" encodes splice junction of length n, "D" deletion of n bases and S encodes soft clip operation which removes n base pairs from either ends of the read.

Listing 2.3: A part of a SAM file with 2 alignments of a multimapped read "ERR2902089.252943611". Read sequences and quality strings have been removed and tabs have been replaced with spaces for readability.

```
1 ERR2902089.252943611 16 chr2 227258887 3 119M2I30M * 0 0 NH:i:2
    HI:i:1 AS:i:123 nM:i:9
2 ERR2902089.252943611 272 chr2 227258887 3 121M2I28M * 0 0 NH:i
    :2 HI:i:2 AS:i:123 nM:i:9
```

2.9 Phylogeny related terms

Phylogenetics is a part of systematics with studies evolutionary relationships between species, or broader terms such as taxa. A **taxon** is a group of organisms which form a unit according to a common physical or genomic trait. The relations between taxa are often presented in a hierarchical tree structure called a **phylogenetic** or an **evolutionary tree**. Since all life on Earth

[2] https://dash.harvard.edu/bitstream/handle/1/10246875/2723002.pdf?sequence=1

shares common ancestry, it can be represented as a single phylogenetic tree [26] [95]. The comparison of evolutionary paths of different species, represented by **clades** - subtrees sharing a common ancestor, is a common problem in biology. A common solution to this problem is the definition of a distance function between the phylogenetic trees.

Phylogenetic trees are often encoded in a newick file format. The tree is represented in a planar way such that every node encapsulates taxon of the species it refers to, along with all of its subtrees. The left tree in Figure 2.1 is encoded in Listing 2.4.

Listing 2.4: An example of a tree encoded in the newick format.

```
1 (((4, )3, 5)2, (, ((9, 10)8, )7)6)1;
```

CHAPTER 3

Trajan

Trajan is a framework which, among others, implements a novel algorithm [45] which computes the distance between phylogenetic trees with a strong attention to honoring hierarchical relationships imposed by the problem structure. It is designed as a flexible framework capable of defining a solver which may exploit the unique properties of a given class of problems based upon the quantification of their similarities or dissimilarities. This chapter is structured so that it follows a natural progression from a naive solution to an efficient ILP solver. First, we are going to present a few definitions related to the problem we are trying to solve. Afterwards, the current research on the topic will be presented and briefly discussed in the literature review section. In the section about arboreal matching, we are going to formulate an ILP problem which Trajan will be solving. The implementation section contains brief explanation of the routines Trajan is using, as well as their capabilities and modularity. Finally, we present our experimental results in the experiments section.

3.1 Additional definitions

Let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be two rooted, labeled, unordered trees with node labels over an alphabet Σ. Furthermore, let C_1 and C_2 be the sets of all subtrees (clades) of T_1 and T_2, respectively. We define a matching $M \subseteq C_1 \times C_2$ such that $\forall (X,Y) \in M$ if $\exists X' \in C_1, (X',Y) \in M$ then $X = X'$ and if $\exists Y' \in C_2, (X,Y') \in M$ then $Y = Y'$. Given a scoring function $\delta : C_1 \times C_2 \to [0,\infty)$ that measures dissimilarity between subtrees, we can define the distance d between T_1 and T_2 as the minimum cost of a matching M between subtrees in C_1 and C_2 as follows.

$$d(T_1,T_2) = \min_{M \text{ matching}} \left\{ \sum_{(X,Y)\in M} \delta(X,Y) + \sum_{\substack{X\in C_1,\\ \text{unmatched}}} \delta(X,-) + \sum_{\substack{Y\in C_2,\\ \text{unmatched}}} \delta(-,Y) \right\} \tag{3.1}$$

The cost of leaving X unmatched is $\delta(X,-)$, while the cost of leaving Y unmatched is $\delta(-,Y)$. By selecting different scoring functions, we are define different distance functions. Several of such distance functions will be covered in the literature review section.

A naive approach to solving this optimization problem, without looking into the properties of δ, would result in a difficult computational problem. The following lemma gives an estimate of the running time of such procedure.

Lemma 3.1 Let T_1, T_2 be trees with n and m nodes, respectively. The number of matchings between T_1 and T_2 is

$$\sum_{i=0}^{\min(n,m)} \binom{n}{i}\binom{m}{i} i!.$$

Proof. Matching between T_1 and T_2 may be formed by selecting i nodes in each of the trees and forming valid pairs. We offer a constructive proof by iterating over i and counting valid matchings.

$i = 0$: There exists a trivial matching in which none of the nodes are matched.

$i = 1$: We select a single node from each of the trees. That can be done in $\binom{n}{1}\binom{m}{1}$ distinct ways.

$i \geq 2$: We select i nodes from both trees in $\binom{n}{i}\binom{m}{i}$ distinct ways. Those nodes may be matched in $i!$ different ways, thus the total amount of matchings is $\binom{n}{i}\binom{m}{i} i!$.

By summing all of the steps above, we prove our claim. □

Let x and y be nodes in a graph, such as tree, in which there exists a partial ordering of the nodes based on the ancestor-descendant relation. If node x is an ancestor of y, we write $x > y$. If nodes x and y can not be compared, we write $x <> y$.

3.2 Literature review

Phylogenetic trees organize biological species in a hierarchical relationship. Their nodes can also represent other entities like tumor subclones that have formed during tumor evolution [54]. Even more, protein-protein interaction (PPI) networks embed a hierarchical structure that can be reconstructed by hierarchical clustering methods [37]. Comparing phylogenetic trees can quantify their similarity under different reconstruction methods, and provide valuable insights into the symbiosis between the evolution of a parasite and its host, for example [51]. The most popular measure of phylogenetic tree similarity is the Robinson-Foulds (RF) metric [85] whose main idea is to match identical nodes (or clades) of one tree to another. It can be efficiently computed but provides a very conservative and "low resolution" dissimilarity measure that is unable to discern between similar structures and is not robust against minor tree changes [17][71]. In [7] a generalization of the RF metric has been proposed that aims to alleviate some of its shortcomings by enforcing a bijective mapping between the tree nodes which preserves the ancestral relationships. Alternative metrics for trees either exhibit unfavorable properties

[24][71][63][11][12] or are hard to compute in practice [2]. Even though the computation of the generalized RF metric is NP-hard, it has been demonstrated in [7] that it can be efficiently computed using our state-of-the-art branch-and-cut solver Trajan [45] which solves an integer linear programming problem and makes the solving of medium to large sized instances feasible in practice. The main contribution of Trajan is its application of novel clique constraints which will be explained in detail during the course of this chapter.

3.2.1 The Robinson-Foulds metrics

The RF metric introduced in [85] can be defined by selecting an appropriate δ in (3.1). Specifically, let X be a clade in T_1 and Y a clade in T_2. We set $\delta(X,-) = \delta(-,Y) = 1$ as a cost of leaving clades unmatched and define

$$\delta(X,Y) = \begin{cases} 0, & X = Y, \\ c, & X \neq Y \end{cases},$$

where $c \geq 2$ is a constant. Note that δ is defined in such way so that the cost of matching different clades is at least as expensive as leaving them unmatched. It is proven in [85] that such function is indeed a distance function, while [27] shows that the RF distance can be computed in linear time with regards to the number of nodes in two trees. Another favorable property of the RF metric is that the optimal matching it computes preserves the ancestral relationships, i.e., descendants of some matched nodes may only be matched with each other.

Despite its favorable properties, the RF metric suffers from several shortcomings. First of all, the distances computed using the RF metric are very discreet, meaning that solving many problem instances coming from the same class of problems will yield a small number of distinct values despite the inputs being highly diverse. We will present more details about this property in the Experiment section. Secondly, two trees which are topologically similar might have very large RF distance. An example illustrating this can be seen in Figure 3.1. The example is an adapted version of the similar example found in [7].

3.2.2 Tree-edit distance

Tree edit distance [97][105], a distance which minimizes the number of edit operations that transforms one tree into another, was first introduced on ordered and labeled trees. Its value may be computed in polynomial time. The same problem, when introduced to unordered trees, was proven to be NP-hard [106]. Let $T = (V,E)$ be a rooted, labeled, unordered tree with node labels over an alphabet Σ. Edit operations which can be done on a tree T are deletion, insertion and substitution.

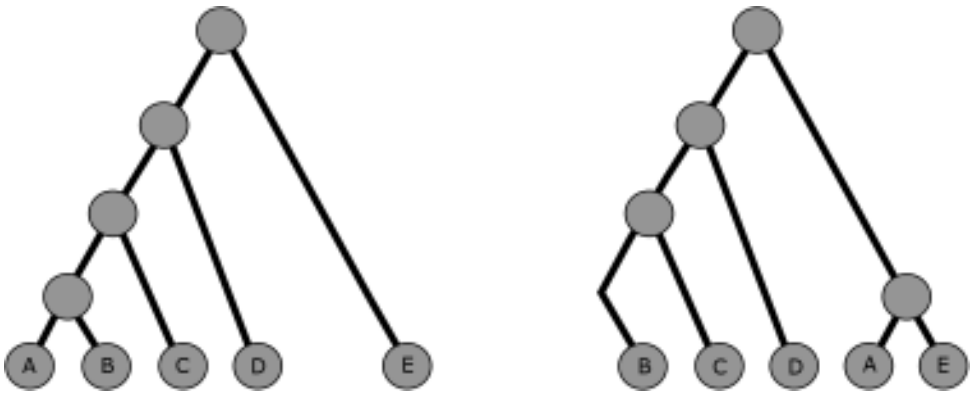


Figure 3.1: The RF distance between the two trees is maximal possible (8) despite their similarities.

Deletion operation deletes a non-root node u from T by rearranging its children and connecting them to its former parent. Insertion is a complementary operation to deletion which inserts a node v as a child of u and makes v a parent of some child of u. Finally, substitution (relabeling) operation changes the label of a node in T. Each edit operation is assigned a cost: $\gamma(a,-)$ for deleting a node with label a, $\gamma(-,a)$ for inserting a node with label a and $\gamma(a,b)$ for substituting a node with label a for a node with label b. Now, the tree edit distance may be defined in a manner similar to (3.1).

Again, let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be two rooted, labeled, unordered trees with node labels over an alphabet Σ and let ℓ be a mapping which assigns to nodes their respective labels. Formally, tree edit distance is defined as a minimum cost sequence of edit operations which transform T_1 into T_2 and is computed as follows.

$$d(T_1,T_2) = \min_{M'} \left\{ \sum_{(u,v)\in M'} \gamma(\ell(u),\ell(v)) + \sum_{\substack{u\in V_1,\\ \text{unmatched}}} \delta(\ell(u),-) + \sum_{\substack{v\in V_2,\\ \text{unmatched}}} \delta(-,\ell(v)) \right\} \tag{3.2}$$

$M' \subseteq V_1 \times V_2$ is a matching which preserves the ancestral relations such that for any two elements $(u_1,v_1),(u_2,v_2) \in M$ it either holds that $u_1 = u_2 \ \wedge \ v_1 = v_2$ or u_1 is a descendant of u_2 if and only if v_1 is a descendant of v_2. Note the resemblance of the above expression to that of (3.1). Evidently, they are equivalent up to the simple substitution of cost functions.

An important final thing to mention about tree edit distance is that the general function d does not satisfy the triangle inequality [70].

3.2.3 Generalized Robinson-Foulds metrics

The main focus of [7] is the definition of a generalization of the RF metric function through the selection of a specific cost function δ. It also explores several topics which are of great interest for us. Two cost functions (dissimilarity measures) are being analyzed: one induced by the symmetric difference and by a class of Jaccard weight functions. The main idea behind these measures is to supplant the absolute penalization of matching clades done by the RF metric with one that penalizes matching according to the topological dissimilarity between them. The proof that the functions induced by these two cost functions are indeed metrics is also given.

Taking the assumptions from 3.1, a **symmetric difference** dissimilarity measure is defined by setting $\delta(X,-) = |X|$ and $\delta(-,Y) = |Y|$. Furthermore,

$$\delta(X,Y) = |X \triangle Y| = |X \cup Y| - |X \cap Y|.$$

Similarly, the **Jaccard weight of order** k, where $k > 0$ is a constant, can be defined with

$$\delta(X,Y) = 2 - 2\left(\frac{|X \cap Y|}{|X \cup Y|}\right)^k,$$

and $\delta(X,-) = \delta(-,Y) = 1$. Note that for $X = Y$, we have $\delta(X,Y) = 0$ and that the following holds for $X \neq Y$.

$$\lim_{k\to\infty} \delta(X,Y) = 2 - 2\lim_{k\to\infty}\left(\frac{|X \cap Y|}{|X \cup Y|}\right)^k \overset{|X\cap Y|\leq|X\cup Y|}{=} 2$$

We can conclude that the distance function induced by the Jaccard dissimilarity measure converges towards the RF metric. Note that, in general, if we were to compute an optimal non-restricted matching using any of the aforementioned dissimilarity measures, unlike with the RF case, those matchings would not necessarily honor the ancestry relations. Thus, the notion of arboreal matching that respects ancestry relations is introduced which we will cover in detail in the sections to come.

3.2.4 GENO solver

Trajan uses GENO [68], a general-purpose non-linear solver, to solve its linear programming problems. It combines a quasi-Newton optimization method with an augmented Lagrangian approach in order to tackle constrained optimization problems. A framework is provided which we use in order to generate a solver tailored to the arboreal matching problem (explained in detail in the following section). In a general case, GENO doesn't require that neither the constraint set nor the objective function are differentiable, but it transforms such problems into differentiable ones.

To solve smooth, constrained problems, GENO takes the augmented Lagrangian approach [57][81]. Let $x \in \mathbb{R}^n$ be a variable and $f : \mathbb{R}^n \to \mathbb{R}$ a differentiable objective function, $h : \mathbb{R}^n \to \mathbb{R}^m$ and $g : \mathbb{R}^n \to \mathbb{R}^p$ differentiable constraint functions which evaluate the variable on a per component basis. Then the general problem which GENO solves may be written as

$$\begin{aligned} \min_x \quad & f(x) \\ \text{s.t.} \quad & h(x) = 0, \\ & g(x) \leq 0. \end{aligned}$$

The augmented Lagrangian problem is then expressed as the following function.

$$L_\rho(x, \lambda, \mu) = f(x) + \frac{\rho}{2} \left| h(x) + \frac{\lambda}{\rho} \right|^2 + \frac{\rho}{2} \left| \max \left\{ 0,\ g(x) + \frac{\mu}{\rho} \right\} \right|^2,$$

where $\lambda \in \mathbb{R}^m$, $\mu \in \mathbb{R}^p_+$ are Lagrangian multipliers and $\rho > 0$ is a constant. An algorithm which minimizes L_ρ iteratively solves unconstrained optimization problems for variables x, λ and ρ. Note that the global minimum is guaranteed only in the case where L_ρ is convex. In each step, if the maximal component-wise violation of the constraints decreases by too small a factor, ρ is doubled to benefit the convergence rate. Augmented Lagrangian algorithm is presented in Algorithm 8.

Algorithm 8 This procedure minimizes L_ρ using the Augmented Lagrangian algorithm. Its inputs are initial approximations of variables x, λ and μ while its outputs are approximations of their respective optimal values.

1: **procedure** AUGMENTEDLAGRANGIAN($x \in \mathbb{R}^n$, $\lambda \in \mathbb{R}^m$, $\mu \in \mathbb{R}^p_+$)
2: $x^0 = 0$, $\lambda^0 = 0$, $\mu^0 = 0$, $\rho = 1$
3: **while** no convergence **do**
4: $x^{k+1} = \arg\min_x L_\rho(x, \lambda^k, \mu^k)$
5: $\lambda^{k+1} = \lambda^k + \rho h(x^{k+1})$
6: $\mu^{k+1} = \max\left\{0,\ \mu^k + \rho g(x^{k+1})\right\}$
7: **end while**
8: **return** x^*, λ^*, μ^*
9: **end procedure**

In the line 4 of the augmented Lagrangian algorithm, an unconstrained smooth optimization problem is being solved (if the problem isn't smooth, it is approximated as such using [75]).

To perform its smooth and unconstrained (barring the variables) optimization step, GENO uses a quasi-Newton limited-memory algorithm for bound-constrained optimization presented in [18]. Let l and u be constant vectors such that we minimize some function $f : \mathbb{R}^n \to \mathbb{R}$ subject

to $l \leq x \leq u$ (point-wise). Furthermore, let the gradient g of f be available. A second order derivative (Hessian matrix), requirement of quasi-Newton methods, is approximated using linear storage ($\mathcal{O}(n)$).

In the k-th iteration of the algorithm, we are given x_k, the current approximation of the minimum, the current function value f^k, a gradient g^k and a Hessian approximation B^k. Using that information, the following quadratic model is constructed.

$$m^k(x) = f(x^k) + (g^k)^T(x - x^k) + \frac{1}{2}(x - x^k)^T B^k (x - x^k)$$

A gradient projection method is used to find a set of active bounds and, afterwards, m^k is minimized. A point-wise linear path

$$x(t) = P(x^k - t g_k, l, u),$$

which projects the steepest descent direction onto the feasible region is considered. Function P is defined, point-wise, as

$$P(x, l, u)_i = \max(l_i, \min(x_i, u_i)).$$

The first local minimizer x^c of the point-wise quadratic function $q^k(t) = m^k(x(t))$ is then computed. The variables x_i^c which attain the values l_i or u_i are considered active and are a part of the active set $\mathcal{A}(x^c)$. The following problem is solved using either direct or dual iterative methods explained in [18].

$$\begin{aligned} &\min \left\{ m^k(x) : x_i = x_i^c,\ \forall i \in \mathcal{A}(x^c) \right\} \\ &\text{s.t.} \quad l_i \leq x_i \leq u_i, \quad \forall i \notin \mathcal{A}(x^c) \end{aligned}$$

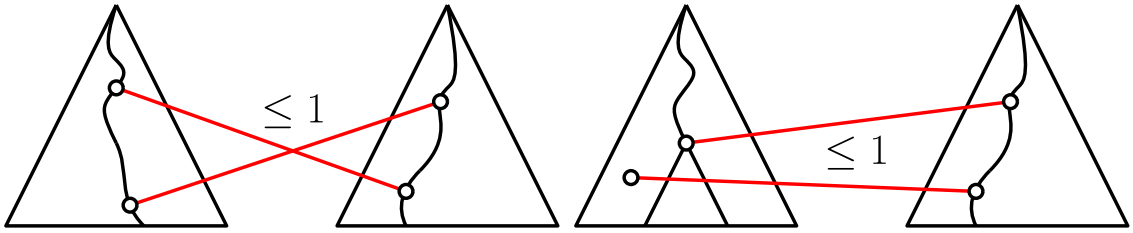
Sometimes, when the computed point does not follow the descent direction, the best feasible point on the path between the computed point and x^c is selected. Let $\bar{x}^{k+1}$ be the solution of the above defined minimization problem. The next iteration x^{k+1} is computed using a line search along $\bar{x}^{k+1} - x^k$.

3.3 Arboreal matching

We have already noticed that tree edit distance minimizes a similar function to that presented in Equation 3.1. If we were to replace γ with δ by substituting the vertices in M' with the subtrees induced by those vertices in T_1 and T_2, respectively, we would have concluded that the two minimization problems are equivalent barring the additional conditions imposed on M'. Following the ideas introduced in [7], we define arboreal matching which imposes constraints on validity of the matchings equivalent to the ones imposed by tree edit distance.

3.3.1 Pairwise conflicts

A matching that minimizes 3.1 doesn't necessarily have to respect the tree structure. In other words, depending on our choice of a dissimilarity measure, there can exist a matching minimal with regards to 3.1 with pairs of elements which do not respect the ancestral relations. An example can be found in the Figure 3.2.



(a) crossing conflict **(b)** independent set conflict

Figure 3.2: Pairwise ancestry violations of a matching which minimizes 3.1.

We consider two pairs of matched nodes to be in a **crossing conflict** if an ancestor of a node is matched to a descendant of its pair. Let M be a matching between trees T_1 and T_2 and $(x_1, y_1), (x_2, y_2) \in M$. We say that those nodes are in a crossing conflict if $x_1 < x_2$ and $y_2 < y_1$ or if $x_1 > x_2$ and $y_2 > y_1$.

Similarly, we say that two pairs of matched nodes are in an **independent set conflict** if the pairs of the nodes in a relation are not in a relation themselves. In other words, if for pairs $(x_1, y_1), (x_2, y_2) \in M$ it holds that $x_1 < x_2 \veebar y_1 < y_2$, then they are in an independent set conflict. Many metrics, such as ones presented in [11] and [12] do not forbid this type of conflict, even though most pairwise conflicts are of its type. We will argument this claim in the next section and confirm it experimentally in the Section 3.5.

Now that we have defined two types of pairwise conflicts, we can move on to the definition of an arboreal matching.

Definition 2. Let M be a matching which minimizes (3.1) such that no pairs of its edges are in either crossing or independent set conflict, i.e. $\forall (x_1, y_1), (x_2, y_2) \in M$ it holds

$$(x_1 < x_2 \Leftrightarrow y_1 < y_2) \vee (x_1 > x_2 \Leftrightarrow y_1 > y_2).$$

Then we say that M is an **arboreal matching**

The problem of finding an optimal arboreal matching is stated to be NP-hard by [7]. For that reason, we have decided to take on a different approach to solving this problem - formulating it as an integer linear programming problem.

3.3.2 Naive ILP formulation

In this section we will present an initial effort towards the formulation of an efficient ILP solution to the problem of finding an optimal arboreal matching between the trees. Let T_1 and T_2 be trees with their respective subtrees $X_i, i = 1,\ldots,n$ and $Y_j, j = 1,\ldots,m$. We start by defining binary variables x_{ij} which reflect the elements of a matching M as follows.

$$x_{ij} = \begin{cases} 1, & (X_i, Y_j) \in M \\ 0, & \text{otherwise} \end{cases}$$

A variable x_{ij} is set to one if subtrees X_i and Y_j are matched and is otherwise set to zero. In order to preserve the matching properties, we introduce the following set of matching constraints.

$$\sum_{i=1}^{n} x_{ij} \leq 1 \qquad j = 1,\ldots,m \tag{3.3}$$

$$\sum_{j=1}^{m} x_{ij} \leq 1 \qquad i = 1,\ldots,n \tag{3.4}$$

Inequalities (3.3) ensure that no subtree in T_2 is matched more than once, while inequalities (3.4) ensure the same thing for T_1. Finding a minimal arboreal matching can be formulated as a maximization problem. First, we take the minimization function from (3.1) and introduce the variables x_{ij} into it.

$$\sum_i \sum_j \delta(X_i, Y_i) x_{ij} + \sum_i \delta(X_i, -) \left(1 - \sum_j x_{ij}\right) + \sum_j \delta(-, Y_j) \left(1 - \sum_i x_{ij}\right) \tag{3.5}$$

$$= \sum_i \sum_j \left[\delta(X_i, Y_i) - \delta(X_i, -) - \delta(-, Y_j)\right] x_{ij} + \sum_i \delta(X_i, -) + \sum_j \delta(-, Y_j) \tag{3.6}$$

Note that expressions $1 - \sum_j x_{ij}$ and $1 - \sum_i x_{ij}$ in (3.5) are equal to one if X_i and Y_j are not matched. Last two summands in (3.6) are, effectively, constant expressions exclusively depending on the definition of δ and the topology of the trees. In an optimization problem they can be disregarded as such. Let $w_{ij} = \delta(X_i, -) + \delta(-, Y_j) - \delta(X_i, Y_i)$ and C a constant. Then we can write (3.6) as

$$-\sum_i \sum_j w_{ij} x_{ij} + C. \tag{3.7}$$

After performing an easy computation we see that the generalized RF metric with Jaccard dissimilarity measure of order k has its corresponding

$$w_{ij} = 2\left(\frac{|X \cap Y|}{|X \cup Y|}\right)^k,$$

while for the symmetric difference induced generalized RF it has the form

$$w_{ij} = 2|X \cap Y|.$$

Let $\mathcal{I}$ be a set of all possible pairwise conflicts between T_1 and T_2. We can formulate (3.1) as a (naive) maximization ILP problem.

$$\max \sum_{i=1}^{n}\sum_{j=1}^{m} w_{ij}x_{ij} \tag{3.8}$$

$$\text{s.t.} \sum_{j=1}^{m} x_{ij} \leq 1 \qquad \forall i = 1,\ldots,n, \tag{3.9}$$

$$\sum_{i=1}^{n} x_{ij} \leq 1 \qquad \forall j = 1,\ldots,m, \tag{3.10}$$

$$x_{ij} + x_{kl} \leq 1 \qquad \forall\{(i,j),(k,l)\} \in \mathcal{I}, \tag{3.11}$$

$$x_{ij} \in \{0,1\}, \tag{3.12}$$

Unfortunately, as will be confirmed in the experiments section, the above described formulation is, at best, suitable for computing an arboreal matching of small-to-medium sized trees. Despite the problem being classified as NP-hard and any algorithm which finds solution to an ILP problem having an exponential running time, we will try to offer a more practical formulation than the naive one and a solver which solves it. The additional complexity of the problem stems from a large amount of constraints in the set $\mathcal{I}$. In order to illustrate it, the following theorem gives the number of pairwise conflicts for two complete binary trees.

Theorem 3.2 Let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be two complete binary trees such that any root-to-leaf path in T_1 has k_1 nodes and k_2 in T_2. Then the total number of pairwise conflicts can be expressed as

$$\begin{aligned} &2^{k_2}(2^{k_2} - k_2 - 1)((k_1-2)2^{k_1}+2) \;+\; 2^{k_1}(2^{k_1} - k_1 - 1)((k_2-2)2^{k_2}+2) \\ &\quad -((k_1-2)2^{k_1}+2)((k_2-2)2^{k_2}+2). \end{aligned} \tag{3.13}$$

Proof. The nodes of a complete binary tree with k long root-to-leaf paths can be separated into k levels based on how far away are they from the root node. Then, the i-th level has 2^{i-1} nodes, while the entire tree has $2^k - 1$ nodes. A subtree induced by a node at the i-th level has $2^{k-i+1} - 1$ nodes, while its complement has $2^k - 2^{k-i+1}$ nodes. We will iterate through k_1 and k_2 levels of

each tree.

For each i-th level in T_1 and j-th level in T_2 we select one of their nodes $x \in V_1$ and $y \in V_2$. This can be done in $2^{i-1} \cdot 2^{j-1}$ different ways. Let X be a subtree induced by x in T_1, Y a subtree induced by y in T_2 and $\bar{X}, \bar{Y}$ their complements in their respective trees. Then, a pair (x,y) is in conflict with all pairs from a set

$$S_{xy} = \{(u,v) : (u \in X \wedge v \in \bar{Y}) \ \vee \ (u \in \bar{X} \wedge v \in Y)\}.$$

Since we have explicit formulas for sizes of $X, Y, \bar{X}, \bar{Y}$, one may simply determine $|S_{xy}|$. Then, by summing over all i-s and j-s, we get the following expression.

$$\sum_{i=1}^{k_1} \sum_{j=1}^{k_2} 2^{i-1} 2^{j-1} \left[(2^{k_1-i+1} - 2)(2^{k_2} - 2^{k_2-j+1}) + 2^{k_2-j+1} - 2)(2^{k_2} - 2^{k_1-i+1}) \right]$$

With the assistance of some basic algebra, the above expression can be transformed into

$$2^{k_2}(2^{k_2} - k_2 - 1)((k_1 - 2)2^{k_1} + 2) \ + \ 2^{k_1}(2^{k_1} - k_1 - 1)((k_2 - 2)2^{k_2} + 2). \tag{3.14}$$

Due to some pairs being counted twice, the number of pairwise conflicts given in (3.14) is overestimated. For each pair of internal nodes, we have double counted each of the vertices in its nodes subtrees. Thus, in order to acquire the real amount of conflicting pairs, we subtract the following expression from (3.14).

$$\left(\sum_{i=2}^{k_1-1} 2^{i-1}(2^{k_1-i+1} - 1) \right) \left(\sum_{j=2}^{k_2-1} 2^{j-1}(2^{k_2-j+1} - 1) \right)$$

The above expression can be transformed into

$$((k_1 - 2)2^{k_1} + 2)((k_2 - 2)2^{k_2} + 2).$$

This leads us to the expression in the statement of the theorem and concludes the proof. □

To illustrate the claim of Theorem 3.2, for $k_1 = k_2 = 5$ we would have around 150 thousand constraints, while for $k_1 = k_2 = 10$ the number of conflicts would be almost 17 billlion. If each constraint could be represented only by 8 integers (two integers for each node in a sparse format), it would take almost 128 GB of RAM just to store the constraints. In practice, using state-of-the-art ILP solvers such as CPLEX or GUROBI, memory requirements are significantly higher. This has motivated us to come up with a novel ILP formulation and implement a specialized solver - Trajan.

Before we continue, let us show that the number of crossing edge conflicts is significantly smaller than the total number of pairwise conflicts. In the $\mathcal{O}$-notation, one might express the number of pairwise conflicts given in Theorem 3.2 with

$$\mathcal{O}(2^{k_1+k_2}(k_2 2^{k_1} + k_1 2^{k_2})).$$

In order to single out crossing from the conflicts of both types, it is enough to observe the restrictions of the sets S_{xy} from the proof of Theorem 3.2

$$S'_{xy} = \{(u,v) : (u \in X \wedge v \in [r_2, y]) \ \vee \ (u \in [r_1, x] \wedge v \in Y)\},$$

as the only edges in a crossing conflict with edge (x,y) are ones that start in either of the subtrees and end on the root-to-z, $z = x \vee y$ path of the other tree. Following the same procedure as before we arrive at the $\mathcal{O}$-approximation of the number of crossing conflicts:

$$\mathcal{O}(k_1 k_2 2^{k_1+k_2}).$$

Since the number of crossing conflicts is significantly smaller than the number of independent set conflicts, metrics such as [11] and [12] may return a matching which, potentially, violates a lot of constraints.

3.3.3 Branch and cut algorithm

Since the main computational bottleneck of the naive ILP formulation (3.6) are the constraints in $\mathcal{I}$, we have decided to implement our variant of branch-and-cut algorithm which solves a series of relaxed LP problems. We obtain each relaxation by changing the integrallity constraint (3.12) into $x \in [0,1]$. The initial iteration of our branch-and-cut approach includes all matching constraints (3.9) and (3.10), but omits any constraints coming from $\mathcal{I}$. After computing the initial relaxed solution, we resolve it until no conflicting edges exist. Then, we select a single variable using different heuristical approaches and generate (branch into) two distinct subproblems by fixing the said variable to either zero or one. That way, we have generated two subproblems which are then solved. This iterative procedure continues until all branches of this tree-like computation scheme have reached an integral solution. One with the maximal objective function value is then considered optimal. An example can be seen in Figure 3.3. Various techniques which allow us to avoid computation on some branches are described in the implementation subsection. They all exploit the fact that the solution of an ILP relaxation overestimates the optimal ILP solution (feasible region of an LP is a subset of that of the corresponding ILP).

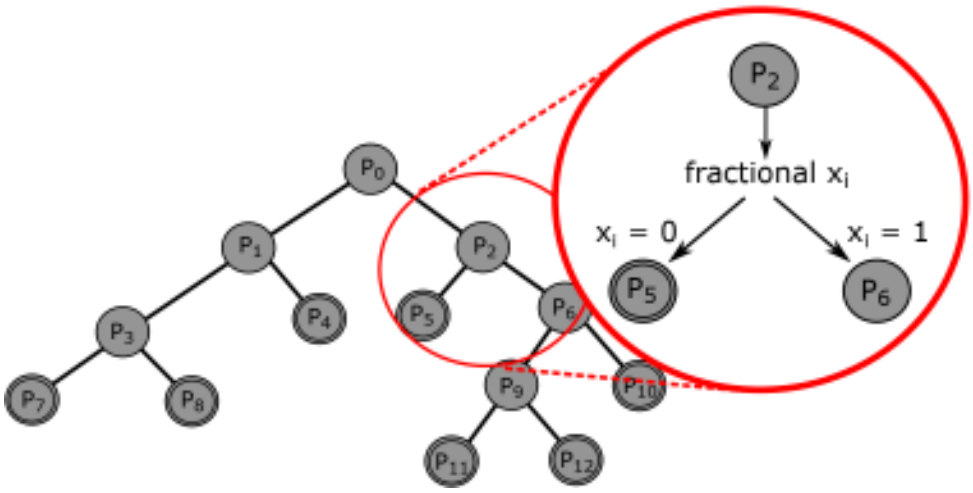


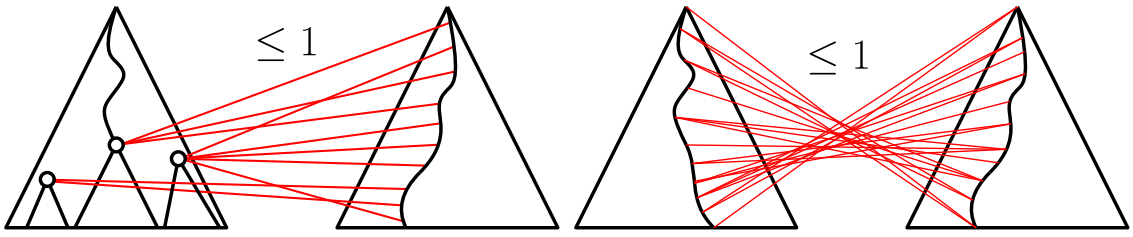
Figure 3.3: An example of the branch-and-cut computation tree which solves subproblems in the BFS order. Each of the points P_i represents a relaxed subproblem. An integral solution is reached in each leaf.

3.3.4 Clique violations

An important computational part of each node in the branch-and-cut tree is the violation detection. In this subsection we show that it is possible to detect maximal sets (cliques) of edges in a crossing and independent set violation, instead of dealing with pairwise conflicts. Stronger bounds introduced this way enable us to contract the polyhedral search space of the relaxation, bringing it closer to the feasible arboreal matching much faster than with the naive approach.

Let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be rooted trees with roots r_1 and r_2, sets of leaves $\mathcal{L}_1$ and $\mathcal{L}_2$, and parent mappings π_1 and π_2, respectively. For any two vertices $p, q \in V_i$, we denote a path from p to q by $[p,q]$. Furthermore, let x^* be a fractional solution to the LP relaxation in any of the branch-and-cut tree nodes. A separation problem pertains to finding a hyperplane which separates x^* from the polytope containing all feasible solutions (arboreal matchings). We will give two procedures which separate the relaxation based the on crossing and independent set violations. An example of their outcomes can be found in Figure 3.4.

A maximal set of (pairwise) crossing edges between two root-to-leaf paths $[r_1, \ell_1]$, $[r_2, \ell_2]$ can be obtained using the following iteration. In the first step, we fix an edge between r_1 and ℓ_2 and push the edge between them into the set $\mathcal{Q}_c$. According to which of the fractional solution components is larger, we either fix r_1 and move to $\pi(\ell_2)$ or fix ℓ_2 and move to the descendant of r_1 along the path $[r_1, \ell_1]$. The edge between the new node and the fixed one is pushed into $\mathcal{Q}_c$. This procedure continues until we have reached the edge between ℓ_1 and r_2. In a symmetric case, we can also start from the edge (ℓ_1, r_2), move downwards along the path $[r_2, \ell_2]$ and upwards along the path $[r_1, \ell_1]$. Since every two edges in $\mathcal{Q}_c$ are incompatible in the crossing sense, the



(a) crossing clique violation **(b)** independent set clique violation

Figure 3.4: Clique ancestry violations.

following sum must not exceed 1.

$$\sum_{(i,j)\in\mathcal{Q}_c} x_{ij} \leq 1 \tag{3.15}$$

Using a dynamic programming approach, we have efficiently identified a crossing edge clique which is most violated, ie., one for which the sum (3.15) is the largest. Let $D[u,v]$ denote a dynamic table which values represent the maximum clique weight (with regards to the sum (3.15)) between subpaths $[r_1,u]$ and $[v,\ell_2]$. Its values are assigned while moving along the root-to-leaf paths such that

$$D[u,v] = x^*_{uv} + \max\left\{D[\pi(u),v], D[u,v']\right\}, \tag{3.16}$$

where $\pi(v') = v$ and $D[r_1,\ell_2] = x^*_{r_1\ell_2}$. The maximum x^*-weight clique can then be computed by backtracking from an entry in D corresponding to (ℓ_1,r_2). An illustration of the procedure is shown in Figure (3.5).

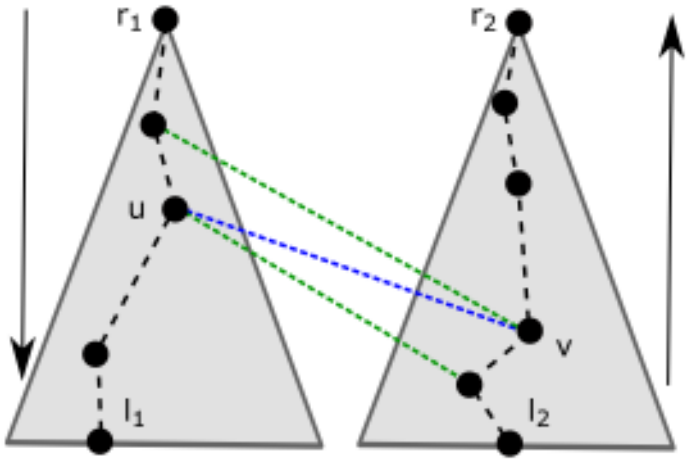


Figure 3.5: An illustration of the step in the dynamic program in which we assign $D[u,v]$ (blue) by selecting a maximum of the green edges.

The time complexity of this procedure is obviously $\mathcal{O}(|\mathcal{P}_1||\mathcal{P}_2|)$, where $\mathcal{P}_1 = [r_1,\ell_1]$ and

$\mathcal{P}_2 = [r_2, \ell_2]$. In order to compute all maximal pathwise violations, we have to run this procedure a total of $|\mathcal{L}_1||\mathcal{L}_2|$ times. Since many of the nodes are shared between different paths in a tree, we propose the following generalization of the dynamic programming scheme obtained by considering all children nodes of v during the procedure. This algorithm, formalized in the following theorem, is depicted in Figure 3.6.

Theorem 3.3 Given a fractional solution x^* we can determine whether a crossing edge clique inequality (3.15) is violated in time $\mathcal{O}(|V_1||V_2|)$.

Proof. For $u \in V_1$ and $v \in V_2$, let $D[u,v]$ denote the weight of a maximum x^*-weight clique between $[r_1, u]$ and $[v, \ell_i]$, for any leaf ℓ_i in the subtree rooted at v. Then

$$D[u,v] = x^*_{uv} + \max\left\{ D[\pi(u), v], \max_{v':\pi(v')=v} \{D[u,v']\} \right\}$$

and

$$D[r_1, \ell_j] = x^*_{r_1 \ell_j}, \forall \text{ leaves } \ell_j \in T_2$$

□

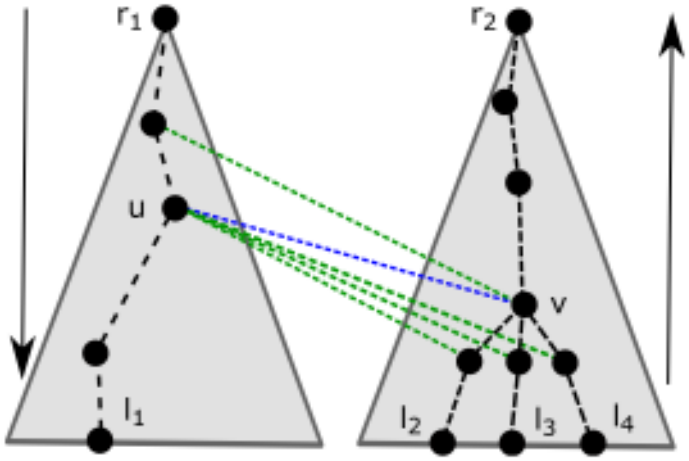


Figure 3.6: An illustration of the step in the dynamic program in which we assign $D[u,v]$ (blue) by selecting a maximum of the green edges. This time we consider all children nodes of v.

As we did for the crossing edges violation, we will now present a dynamic programming procedure which lifts multiple pairwise independent set conflicts into a maximal violation set (clique). Such set consist of edges that are all incident to nodes on a common root-to-leaf path in one tree, and are incident to nodes in the second tree that all lie on distinct root-to-leaf paths, i.e., are independent. Again, edges in such clique $\mathcal{Q}_i$ must satisfy

$$\sum_{(i,j)\in\mathcal{Q}_i} x_{ij} \leq 1 \tag{3.17}$$

For a root-to-leaf path $[r_1, \ell_j]$ in T_1 we start by assigning weights x'_j to nodes in T_2 so that

$$x'_j(v) = \sum_{u \in [r_1, \ell_j]} x^*_{uv}, \quad \forall v \in V_2.$$

Note that this can be done in $\mathcal{O}(|V_1||V_2|)$.

Now we define a dynamic table $D^1_j[v]$ which will keep track of maximum x'_j-weight cliques between $[r_1, \ell_j]$ and an independent set in the subtree of T_2 rooted in v. We will traverse the table starting from the nodes in $\mathcal{L}_2$, gradually moving upwards towards r_2. In each visited node v we will take a maximum of its x'_j weight and a cumulative x'_j weight of its direct descendants.

$$D^1_j[v] = \max \left\{ x'_j(v), \sum_{v' : \pi(v') = v} D^1_j[v'] \right\}$$

The procedure is illustrated in Figure 3.7. The maximum x^*-weight of a clique is then given in $D^1_j[r_2]$ while its elements can be determined by a backtracking algorithm. Analogously, we can define a dynamic table $D^2_i[u]$ between $[r_2, \ell_i]$ and an independent set in the subtree of T_1 rooted at u. Finally, the maximum weight semi-independent clique constraint can be computed as

$$\max \left\{ \max_{\ell_j \in \mathcal{L}_1} D^1_j[r_2], \max_{\ell_i \in \mathcal{L}_2} D^2_i[r_1] \right\}.$$

Formally, the separation of the independent set clique constraints with respect to T_1 and T_2 is given by the following theorem.

Theorem 3.4 Given a fractional solution x^* we can determine whether an independent set clique inequality (3.17) is violated in time $\mathcal{O}(|V_1||V_2|)$.

□

3.4 Implementation

Trajan has been implemented in the C++ programming language. It is comprised of multiple solvers, but the focus of this paper will be on its branch-and-cut part. In this section we will present its most important features. Inputs to Trajan branch-and-cut solver are trees encoded in the newick format and a dissimilarity measure of choice. It returns an optimal matching and its score, which is consistent with the objective function (3.6).

3.4.1 Basic framework

Trajan's branch-and-cut procedure has been implemented as a framework which provides several basic methods used to construct an arboreal matching ILP solver. Its backbone is an

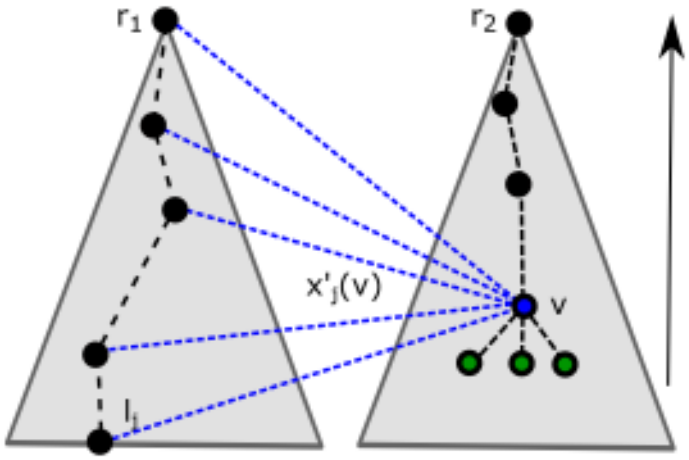


Figure 3.7: An illustration of the step in the dynamic program in which we assign $D_j^1[v]$ (blue) by selecting either a maximum of the sum of x^*-weights on blue edges and table entries for green nodes (children of v).

arbitrary data structure *Open* which stores branch-and-cut tree nodes in order in which they are to be evaluated. The basic functionality is presented in Algorithm 9.

Algorithm 9 This procedure computes an optimal arboreal matching between trees T_1 and T_2 for a given vector of weights w.

```
1:  procedure SOLVE(T1, T2, w)
2:      generate solution vector x and set it to 0
3:      generate matching constraints from T1 and T2 and store them in matrix A
4:      initialize structure Open with the initial problem (w,A)
5:      while Open not empty do
6:          P = Evaluate(Open)
7:          xP = SolveSubproblems(P)
8:          if xP is integral then
9:              set best solution to xP if it has higher objective function value
10:         else
11:             GenerateSubproblems(Open,xP)
12:         end if
13:     end while
14:     return best solution and objective function value
15: end procedure
```

Depending on the selection of the structure *Open*, method Evaluate selects one (or several) subproblems which will be solved by the method SolveSubproblems. A more detailed explanation of the subproblem selection will be given in the section about branching and rounding. For each subproblem, method SolveSubproblems invokes the SolveSubproblem method presented in Algoritm 10. It iteratively solves the given subproblem by resolving the LP relaxation and adding violated constraints. If the relaxation objective value is lower than the currently best

known one, the whole branch will be cut and omitted from further computation. This procedure is called **pruning**. Any found integral solution will be compared to the best integral solution (upper bound) found so far and the best one will be kept. The nodes with fractional values in the solution will use different rounding schemes presented further in the section about branching and rounding in order to generate new problems and add them to *Open*.

Algorithm 10 This procedure solves a subproblem $P = (w, A)$.

```
1:  procedure SolveSubproblem(P)
2:      let x be a previous solution with some fixed variables
3:      while True do
4:          x = solve P with GENO using x as the initial solution
5:          if clique violations exist in x then
6:              add constraints to A
7:          else if w^T x less than objective value of best solution then
8:              return NIL
9:          end if
10:     end while
11:     return x
12: end procedure
```

3.4.2 Branching and rounding schemes

Function Evaluate in Algorithm 9 selects one or more subproblems from data structure *Open* and hands them over to the solver. We have implemented three main branching strategies which have been used in experiments - depth first, best first and hybrid strategies.

Depth first strategy uses stack as a data structure *Open*. When a variable is fixed, it evaluates both generated subproblems. A subproblem with a lower objective value is pushed to the stack before one with the higher objective value. As a result, this procedure traverses a single root-to-leaf path in the branch-and-cut tree always taking branches with a higher objective value. The advantage of this strategy is that it acquires the first integral solution (upper bound) fairly quickly so it can start pruning branches. Best first strategy uses an array as a data structure *Open*. It solves all nodes in *Open* and branches on the one with the highest objective value. Even thought it finds an integral solution slower than the depth first strategy, it is often closer to the optimal and allows better pruning. An example of the depth first and best first approach can be seen in Figure 3.8. By combining the aforementioned strategies we get a hybrid strategy which evaluates *Open* data structure in a depth first order until first integral solution is computed. Afterwards, it evaluates the rest of the branches using best first approach.

New subproblems are generated by the function GenerateSubproblems in Algorithm 9 by selecting and fixing a variable to both one and zero. The decision which variable should be fixed

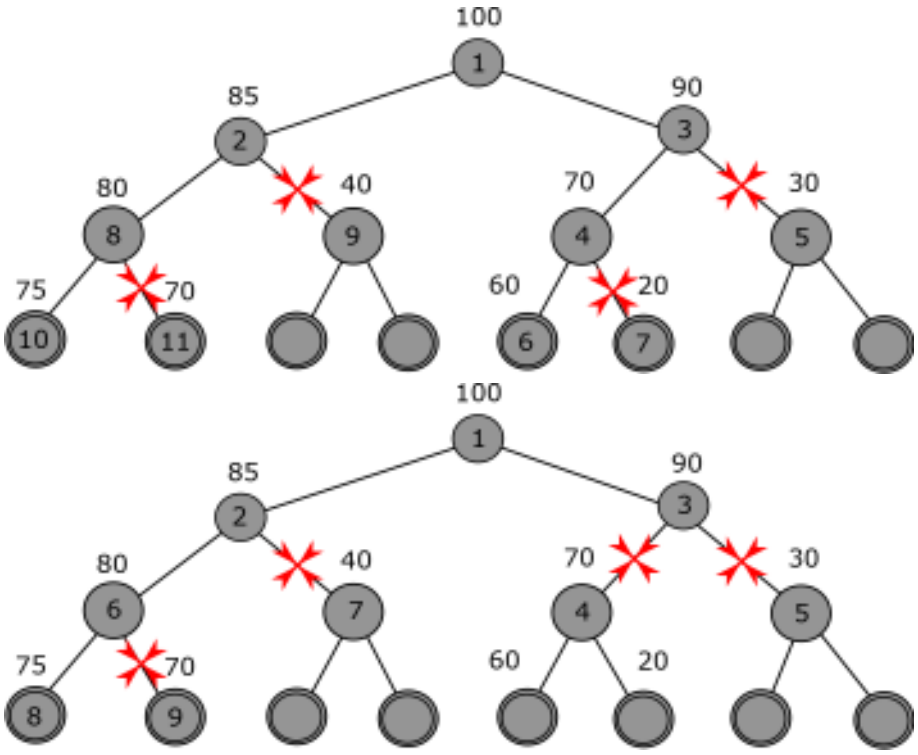


Figure 3.8: A comparison between depth first (top) and best first (bottom) branching strategy on a branch-and-cut tree. The objective function values are written above, while the order of computation is written inside the node. Pruned branches are marked with a red cross. Depth first approach obtains its first integral solution after 6 computations and it computes 11 subproblems in total, while the best first approach finds first integral solution in the 8-th out of 9 subproblems.

can be done in several ways. We can select any x_{ij}, which hasn't yet been fixed, according to any of the following criteria:

1. "least fractional" variable - $\arg\max_{i,j} |\frac{1}{2} - x_{ij}|$,
2. "most fractional" variable - $\arg\min_{i,j} |\frac{1}{2} - x_{ij}|$,
3. maximal weight - $\arg\max_{i,j} w_{ij}$,
4. maximal objective function contribution - $\arg\max_{i,j} w_{ij} x_{ij}$,
5. x_{ij} is in conflict with the least amount of variables.

By selecting any combination of the branching and rounding strategies we get different branch-and-cut solvers. It is possible to select multiple strategies and run them in parallel. The advantage of this is that these solvers share information about the best current solution. This way, a solver which has computed the best integral solution can share it with others in order to prune more branches. A special form of parallelism has been implemented for the best first

branching scheme. Trajan may use an arbitrary number of threads to compute solutions to all subproblems in *Open* so the best solution among them is obtained faster.

3.4.3 Greedy strategy

A greedy algorithm is used to compute an approximate lower bound in each of the LP relaxations Trajan is solving. It iterates through all variables Trajan has not yet fixed and attempts to obtain a feasible solution by fixing as many variables as possible to 1, without violating any of the clique constraints, starting from those variables with the highest weight (w_{ij}). This approach is very fast, but the results it returns are often very far from optimal. Consider the example given in Figure 3.9. Greedy strategy would include the blue edge between x_2 and y_n in the solution which is in conflict with all green edges while the optimal solution includes all green edges between nodes x_i and y_{i-1} for $i = 2,\ldots,n$. Thus, the gap between the greedy and optimal solution can be made arbitrarily large by selecting $n \geq 4$.

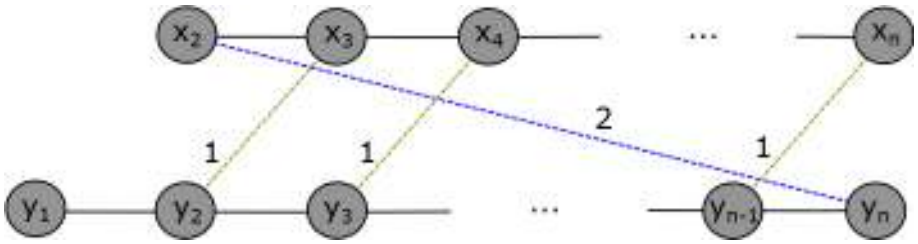


Figure 3.9: The edges between nodes of the paths $x_2,\ldots,x_n$ and $y_1,\ldots,y_n$ are colored green and blue. The weight of the blue edge is 2, while the weight of green edges is 1. The weight of the rest of the edges (not displayed) is 0.

Hence, greedy strategy is used to provide an initial (integral) solution as well as to approximate the lower bound in each iteration. A pseudocode of the greedy procedure is presented in Algorithm 11.

If the feasible solution generated by the greedy strategy is better than the currently best integral solution obtained by Trajan, it is used instead. This heuristic approach is particularly useful for pruning branches where a lot of variables are fixed to zero and before the first integral solution has been computed.

3.5 Experiments

In this section computational results will be given and discussed. We have generated datasets which consist of 10000 trees with 50 leaves as well as 1000 trees with 75 and 100 leaves from Yule and Uniform distributions which create biologically plausible random trees, where all phylogenetic trees are equally likely under the former model, and the latter one assumes a constant

Algorithm 11 Greedy algorithm used to approximate an integral solution of the LP relaxation. Let x be the sequence of variables x_{ij} and w of weights w_{ij}.

```
1:  procedure GREEDY(x, w)
2:      v = variables from x sorted by their respective weights in w
3:      for i = 0, ..., |V| − 1 do
4:          if v[i] is fixed then
5:              continue
6:          else if v[i] not in conflict with v[0, ..., i − 1] then
7:              set the component of x corresponding to v[i] to 1
8:          else
9:              set the component of x corresponding to v[i] to 0
10:         end if
11:     end for
12:     return x
13: end procedure
```

speciation rate [9]. We have also obtained 1000 real-world green algae phylogenetic trees from [69] and a dataset with flowering plants [92]. Trees will be compared using the metrics induced by the symmetric difference and Jaccard weight dissimilarity measures.

This section is comprised of four main parts. In the first subsection we will argue about metrics induced by Jaccard weight being the generalization of RF metrics. The second subsection will deal with the importance of two aforementioned constraint sets. Then, we are going to give general results regarding the distance distributions and running times for all of the datasets. Finally, we will give a comparison between ours and the naive ILP formulation. The results may also be found in [13].

All tests were conducted on a server computer with two 2.30 GHz Intel® Xeon® E5-2697 v4 processors with 18 cores / 36 threads each, 320GB @ 2,40GHz DDR4 memory operating on Scientific Linux 7.5 (Nitrogen). C++ compiler used to compile Trajan was GCC 4.8.5 20150623 and all running times were captured using the GNU time command.

3.5.1 Convergence to the RF metrics

We have already mentioned that the metrics induced by Jaccard weight converge towards the original RF metrics when their orders increase. In this subsection we will experimentally support that claim, i.e. that the metrics induced by Jaccard weight are a generalization of the RF metrics. To that end we have analyzed instances from 100 leaf Yule and green algae datasets which we tested subsequently with increasing values of Jaccard order k until convergence was reached. The convergence point was confirmed by the tool presented in [10].

In Figure 3.10 we present the distances for green algae dataset. For each integer k we have made a box-and-whiskers plot depicting the normalized distance distribution quantiles while the dashed horizontal black line at y = 1 represents the RF distance. Distances of the metrics induced by Jaccard weight indeed converge towards the ones obtained by the RF metrics. If we were to run unconstrained matching instead of arboreal matching, it would not be clear as to how many clique constraints do the computed matchings violate. Figure 3.11 shows that with increasing values of k the number of conflicts decreases. Note that zero violations must occur once the non-arboreal distance converged to the RF distance. The same figure also shows the difference in distance distributions of the metrics induced by Jaccard weight with regards to arboreal matching and unconstrained matching. We can see that as Jaccard converges towards RF, unconstrained matching becomes a good approximation of arboreal matching.

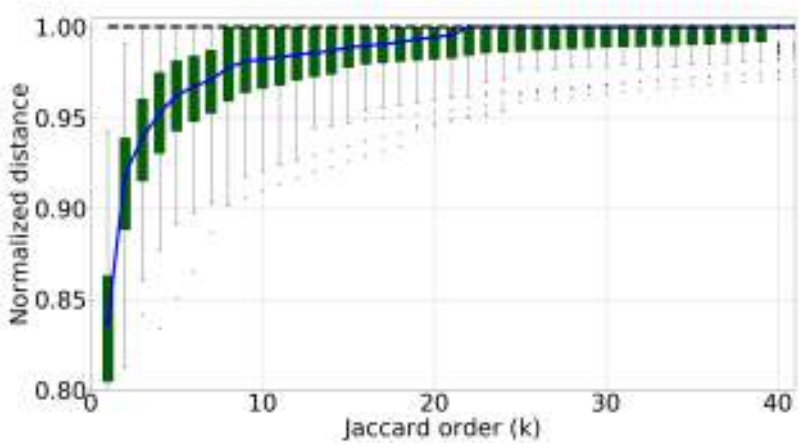


Figure 3.10: Normalized Jaccard weight distances of the green algae dataset for increasing k.

We have run the same experiments on the random instances from both distributions. Figures 3.12 and 3.13 show the results obtained for the 100 leaf instances from Yule distribution which follow the same pattern as the green algae instances with the convergence being reached for a far smaller k.

3.5.2 Impact of the constraint sets

In the last subsection we have argued that for a sufficiently large k unconstrained matching does not violate the ancestry in the context of arboreal matching. Since the RF metrics can be computed in polynomial time and computing its generalization is NP-hard, it is important to argue the benefits of computing the arboreal matching over the RF metrics.

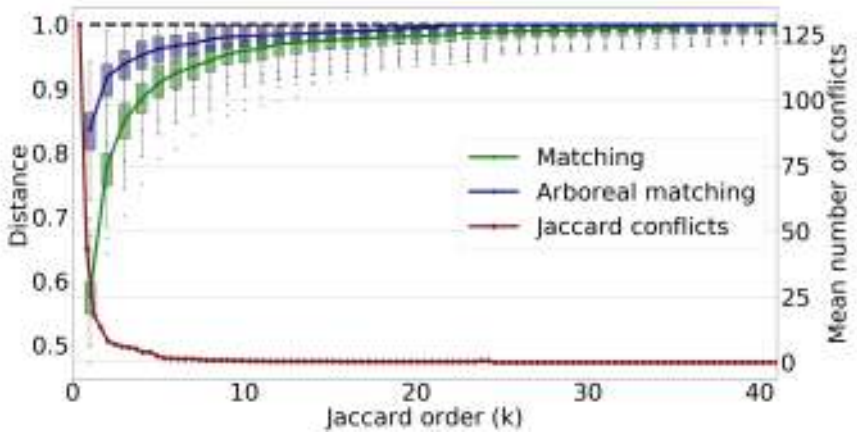


Figure 3.11: A comparison of unconstrained and arboreal matching distance distributions with a number of produced conflicts for the green algae dataset.

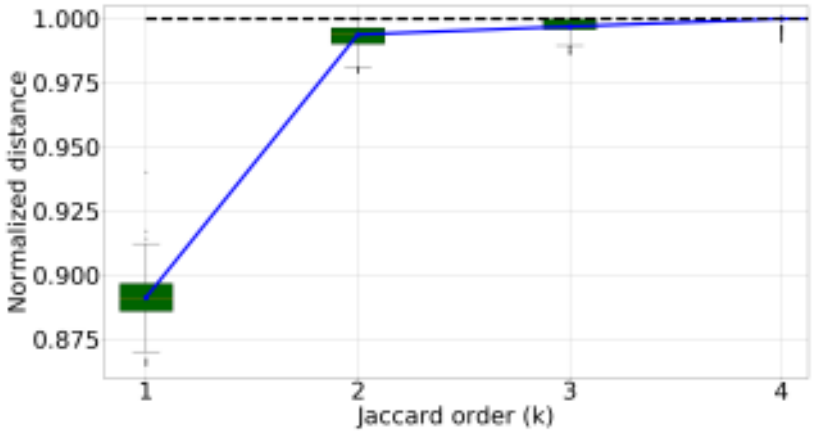


Figure 3.12: Normalized Jaccard weight distances of the 100 leaf Yule dataset for increasing k.

Figure 3.14 represents the histogram of distances computed using Jaccard weight of order 1 from 50 leaf uniform dataset while the RF distances for the same dataset fall within only 3 different values – 92, 94 and 96. Most of the instances (slightly less than 80%) produce an RF distance of 96. It is clear that the generalized RF metrics provide a far greater "resolution", i.e., a wider distance distribution. This means that the generalization of the RF metrics could be more suitable for the procedures which rely on detecting slight differences between the trees. The rest of the random datasets yielded comparable results where the distance distribution of the Jaccard weight induced metrics was wide in comparison to the very discrete RF distances. The RF distances of green algae instances are slightly more diverse as can be seen in Figure 3.15. But, akin to the previous results, the generalized RF metrics show a wider distribution which is presented in Figure 3.16 (blue).

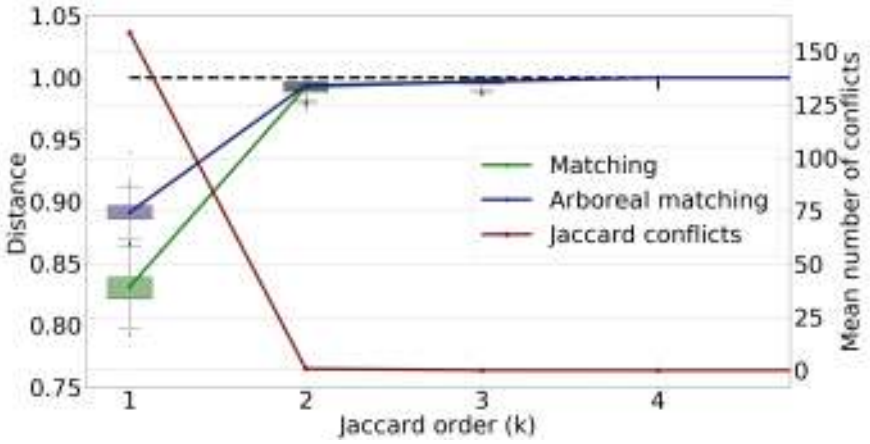


Figure 3.13: A comparison of unconstrained and arboreal matching distance distributions with a number of produced conflicts for the 100 leaf Yule dataset.

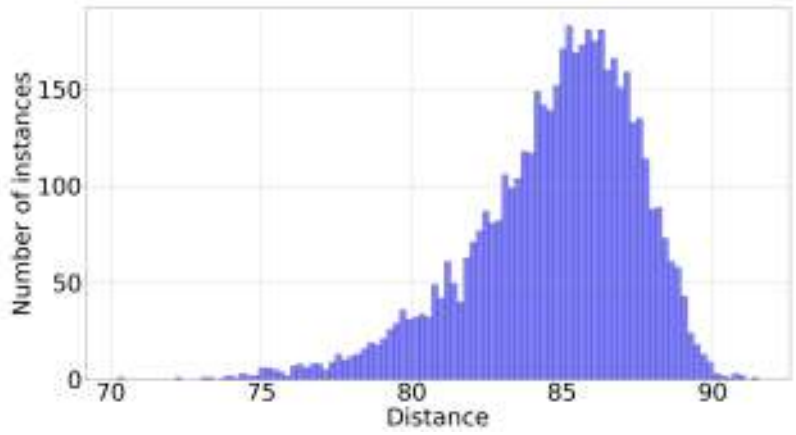


Figure 3.14: Distribution of the distances of the 50 leaf instances from uniform distribution with Jaccard weighted arboreal matching.

On the other hand, as hinted in Figure 3.11 and Figure 3.13, there exist significant differences between arboreal and unconstrained matching for Jaccard weight of order 1. Unconstrained matching produces a lot of conflicts of which around 95% are independent set violations. In Figure 3.16 we show the demeaned distance distributions of arboreal and unconstrained matching for green algae instances. We have run a similar experiment on the green algae instances using unconstrained matching with metrics induced by symmetric difference. The resulting distribution is given in Figure 3.17. The summary of the distribution of conflicts for this and all following experiments in this section is given in Table 3.1.

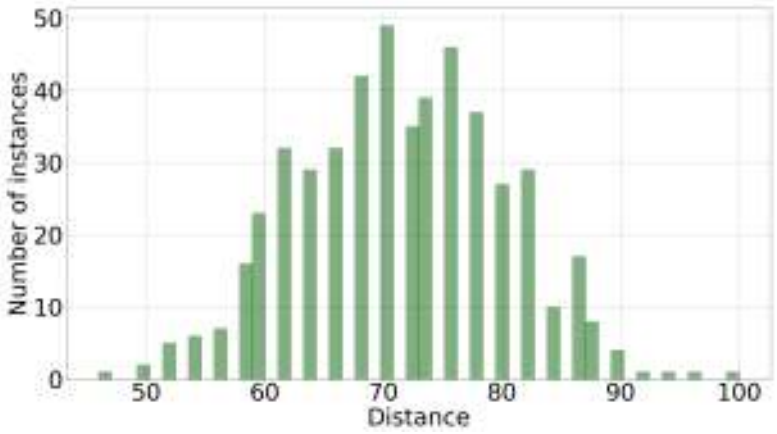


Figure 3.15: Distribution of the RF distances of the green algae instances.

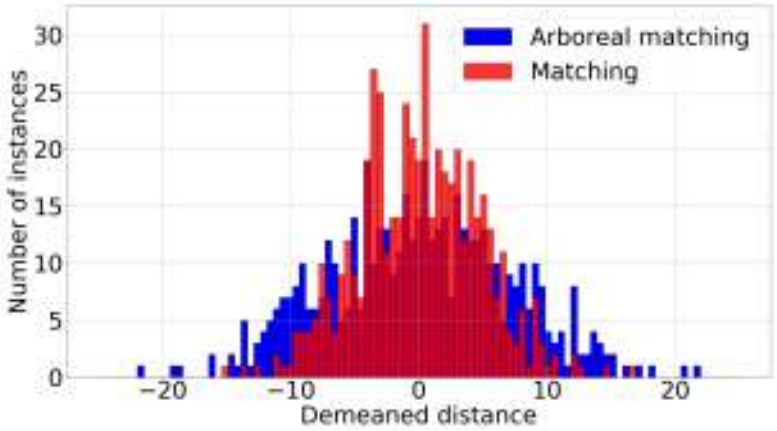


Figure 3.16: Demeaned distribution of distances (arboreal matching and unconstrained matching) induced by Jaccard weight from green algae dataset.

It has been shown in [11] that a polynomial time algorithm exists for computing matching which does not violate crossing edge constraints when using symmetric difference dissimilarity measure (MC distance). More precisely, in [11] it was proven that distance doesn't change in the presence of crossing constraints. Therefore, on the green algae dataset, we have computed matchings which only satisfy crossing edge constraints and counted the amount of produced independent set conflicts (see Table 3.1). It is worth noting that not a single instance ended up having no conflicts. Also, the distances we computed agreed with the values obtained by unconstrained matching. Following the previous experiment, we have also tested how does the green algae dataset perform with arboreal matching and Jaccard induced metrics of order 1

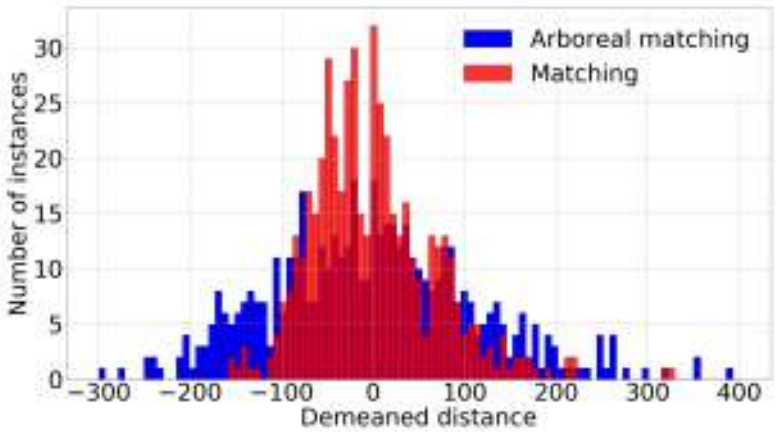


Figure 3.17: Demeaned distribution of the distances (arboreal matching and unconstrained matching) induced by the symmetric difference from the green algae dataset.

against the same metrics without explicitly enforcing independent set constraints. The distance distributions are shown in Figure 3.18, while the number of produced conflicts is located in Table 3.1. Like before, no matching was conflict-less, which indicates the importance of enforcing the independent set constraints in computation of the arboreal matching.

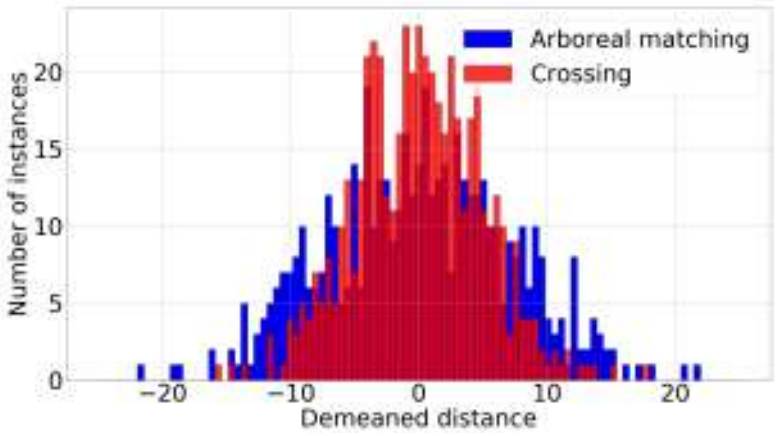


Figure 3.18: Demeaned distribution of the distances (arboreal matching and crossing) induced by Jaccard weight from the green algae dataset.

3.5.3 Distribution details and running times

In Figures 3.20 and 3.14 we present the distance distributions for 10000 random trees from uniform distribution using arboreal matching with symmetric difference and Jaccard weight,

Table 3.1
Distribution of the conflicts for unconstrained matching and matching with crossing constraints.

		Mean	St. dev.	Q1	Q2	Q3
Matching	Symmetric	216.32	60.86	176.75	205.00	252.00
	Jaccard	141.69	48.91	103.00	142.00	174.25
Matching with crossing constraints	Symmetric	202.40	60.14	163.00	190.50	238.25
	Jaccard	91.05	38.31	63.00	85.00	113.25

Table 3.2
Summary for the random 50 leaf datasets.

		Distance		% data within n-th st. dev.			Running time (s)		
		Mean	St. dev.	1	2	3	Min	Mean	Max
Uniform 50	Symmetric	589.37	72.87	40.06	95.66	99.02	3.85	50.46	565.53
	Jaccard	84.75	2.77	39.15	95.52	98.88	0.17	3.98	92.54
Yule 50	Symmetric	425.40	28.46	38.06	95.27	99.46	2.10	22.59	189.23
	Jaccard	83.92	1.72	38.33	96.58	99.48	0.09	0.85	21.14

respectively. For both dissimilarity measures around 99 percent of the distances fall within 3 standard deviations around the mean, while around 95% and 40% of the distances are within 2 and 1 standard deviations, respectively. This may infer that both symmetric difference and Jaccard weight induce comparable resolution. Figure 3.19 gives a plot of running times (Trajan) for each of the dissimilarity measures with the values sorted by symmetric difference running time. It is clear that the empirical running times are in favor of Jaccard weight by quite a margin. Summary for the dataset is given in Table 3.2.

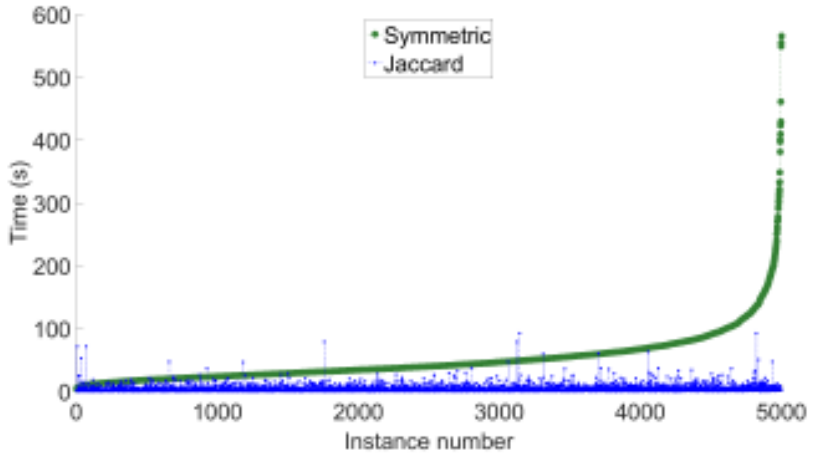


Figure 3.19: Running times of 50 leaf instances from uniform distribution with symmetric difference and Jaccard weight.

Trees with 50 leaves from Yule distribution yield similar results and the main results are summarized in Table 3.2. We can see that the "resolution" is similar regardless of the dissimilarity

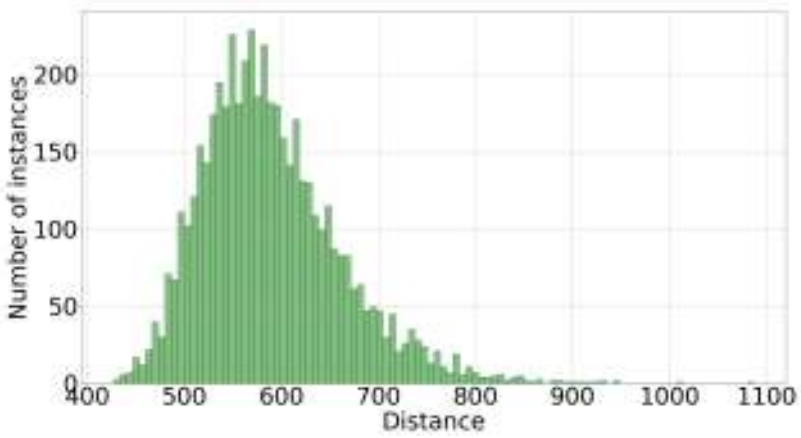


Figure 3.20: Distribution of the distances from 50 leaf uniform distribution using the symmetric difference.

measure used while the running times are again lower for the Jaccard weight. We have presented the results of the remaining random datasets, namely the 1000 uniform and Yule instances with 75 and 100 leaves in Table 3.3.

Figures 3.16 and 3.17 (blue) depict the distance distributions for the green algae dataset. The distance induced by symmetric difference has a mean of 580.05 and a standard deviation 113.80 while the Jaccard distances have a mean 52.91 and a standard deviation 7.21. Both distributions have more than 99% of their values within 3 standard deviations while Jaccard has 96.6% within 2 and 66% within 1. On the other hand, symmetric difference places 94.99% of the distances within 2 standard deviations and 69.34% within one. Similarly to the random trees, the computation time for the green algae instances largely favored Jaccard weight which mean time was 4.77 seconds as opposed to 459.05 seconds required for the symmetric difference. The extremes were relatively close to mean for Jaccard with a minimum of 0,65 seconds and a maximum of 24.40 seconds. Unlike the former, the symmetric difference saw a minimum of 14.64 seconds, which was even quite above the mean for Jaccard, and a maximum of 4210.82 seconds.

The common trend for all data is that Jaccard weight and symmetric difference offer comparable resolutions while the running time of the symmetric difference is larger. To summarize, we have tested the properties of the metrics induced by symmetric difference and Jaccard weight which did honor ancestry relations. Both have been tested on simulated and real world datasets for the differences in “resolution”, i.e., how well can the metrics discern between various trees, relations to other metrics such as RF and MC, and running time. We have concluded that the “resolution” of both metrics is similar while the running time favors Jaccard weight by a large margin.

Table 3.3
Main results for the 75 and 100 leaf datasets.

		Uniform75		Yule75		Uniform100		Yule100	
		Sym	Jaccard	Sym	Jaccard	Sym	Jaccard	Sym	Jaccard
Distance	Mean	1138.46	132.32	753.88	129.34	1830.27	179.18	1124.18	174.52
	St dev	141.42	3.30	43.76	1.89	243.41	3.53	60.56	2.00
% data	1	45.50	40.60	41.00	35.60	39.40	37.80	37.88	41.80
within	2	94.20	94.20	95.40	96.20	95.80	94.00	95.79	94.80
n-th st. dev.	3	98.80	98.40	99.20	99.00	99.20	99.80	99.40	99.60
Running	Min	43.09	0.30	16.86	0.20	221.39	0.57	82.61	0.22
time (s)	Mean	379.50	23.35	163.31	1.66	2514.42	103.05	1000.10	4.27
	Max	5676.33	1990.04	3266.62	22.07	27071.3	5947.51	35059	75.38

3.5.4 Trajan vs naive ILP

In this section we argument the claim that the naive ILP formulation is not practical and that Trajan may be used to efficiently solve problem instances which the naive ILP formulation is unable to tackle. Also, we use the bad performance of the naive ILP as an demonstration of the hardness of the problem of finding the optimal arboreal matching. The benchmark has been conducted using the same solver, CPLEX version 12.4, for which we have implemented both the naive ILP and Trajan's formulation. We haven't used Trajan's standalone implementation for the purpose of comparison. The dissimilarity measure used was induced by Jaccard weight of order 1. All-against-all comparison has been conducted on the 100 phylogenetic trees from the green algae dataset [69] and the flowering plants dataset [92]. All of the 4950 green algae problem instances have been solved using Trajan's formulation in slightly more than an hour, while the implementation of the naive formulation took as much as 9415 seconds for a single problem instance. The results are presented in Figure 3.21. We have observed similar running times on the flowering plants dataset. While Trajan solved 4950 larger problem instances in 17 seconds on average, the naive formulation terminated due to exceeding the memory limit of 80 GB after 100 instances and roughly 49 hours of computation time.

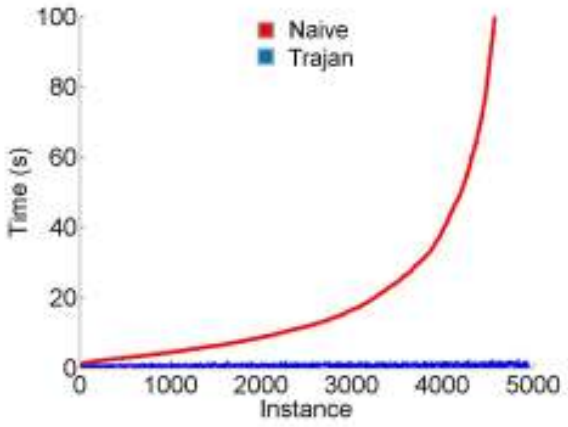


Figure 3.21: A comparison of the running times for a CPLEX based implementation of the naive ILP formulation and Trajan's formulation.

The results of our benchmarks suggest that Trajan's formulation is both practical and superior to the naive one.

3.6 DAG generalization

It is possible to generalize the problem of finding an optimal arboreal matching to directed acyclic graphs (DAGs). Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. We can generalize our definition of ancestry partial ordering to DAGs. Then, two edges $(i, j) \in E_1$ and $(k, l) \in E_2$ are considered compatible if $i < k \Leftrightarrow j < l$. By redefining the set $\mathcal{I}$ of pairwise conflicts in the naive ILP formulation, a problem of maximum weight arboreal matching is defined for DAGs.

Generalized clique constraints (crossing and independent set as seen in Figure 3.4) are still valid for DAGs, but the separation algorithms need to be re-defined. Let π be a mapping which maps a node u in a DAG to a set of its parents. Then, we can generalize the crossing clique constraint dynamic table to

$$D[u,v] = x^*_{uv} + \max\left\{ \max_{u' \in \pi(u)} D[u',v], \max_{v':\pi(v')=v} \{D[u,v']\} \right\}$$

and

$$D[r_1, \ell_j] = x^*_{r_1 \ell_j}, \quad \forall \text{ roots } r_i \in G_1, \ \forall \text{ leaves } \ell_j \in G_2.$$

The running time of this procedure is $\mathcal{O}(|E_1||E_2|)$ and, as such, it is still polynomial. Note that this scheme is valid for a fixed path in one of the DAG-s against the entire second DAG. In order to determine the maximal violation, we must run the procedure for all paths in the first DAG. Unfortunately, determining the most violated independent set clique, as stated in [45],

is NP-hard. The construction and implementation of an algorithm which solves this problem defines one of the aims of our future research.

CHAPTER 4

Fortuna

On a high level, fortuna quantifies abundances of the equivalence classes of short read samples and identifies novel splicing events using a refined annotation. Prior to explaining the method, we will present the current state in the literature and explain the principles of two additional data structures which are essential parts of fortuna. We will start the method section by giving our definition of the equivalence classes of the reads and relating it to the existing work. Fortuna's workflow, which the rest of the method section will closely follow, consists of three steps: index building, alignment and postprocessing. Given an annotation and a genomic sequence, fortuna builds an alignment index used in the alignment step. Such an index is constructed by supplementing (refining) the annotation in a novel way which allows for the detection of alternative splicing events. It consists of a set of transcript fragments which represent regions of the genome of a particular interest. After the construction of an index, fortuna uses kallisto [14] to obtain the alignment information for the reads. In the postprocessing step, the alignment information is processed in order to quantify the abundances and identify novel splicing events using a well defined set of rules. In order to process novel splice sites, we propose transcript refinement with the assistance of a genomic aligner. This procedure allows us to use the information retrieved by tools such as [30] to refine our equivalence classes to, potentially, capture additional novel splicing events. Fortuna has been extensively tested and the results of the tests are presented in the results section. Three types of datasets have been processed in our experiments - simulated datasets with moderately sized samples, datasets with the data coming from autism patients with huge sample sizes and datasets with samples coming from a subspecies of a fly.

4.1 Additional definitions and data structures

In this section we are going to explain two data structures which are used by fortuna: red-black trees and suffix tries. They are used to represent the annotation and the alignment index, respectively, in the memory. Before we proceed with explaining them, we give some basic definitions. For the sake of simplicity, let the entire genome be represented as a continuous

string of nucleic acids obtained by concatenating the chromosomes. Let T be a transcriptome annotation such that any transcript $t \in T$ is given by a sequence of exons sorted by their genomic coordinates. Due to the fact that exons coming from different transcripts may overlap, we alter the definition in such way that each t is given by a sequence of disjoint bins called subexons which we define as minimal exonic regions bounded by splice sites or transcription start and end sites. Like it was the case with exons, subexons are sorted by their respective genomic coordinates. We illustrate the procedure with the following example. Let $t_1, t_2 \in T$ be transcripts containing two exons each, as depicted on Figure 4.1. First exons of both transcripts define three subexons s_1, s_2, s_3. Last two exons in both transcripts are identical and, thus, they define a single subexon s_4.

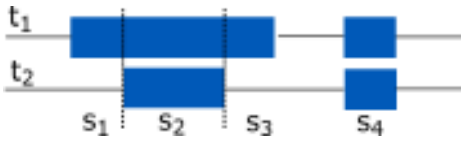


Figure 4.1: Subexon generation based on known splice sites.

4.1.1 Red-black tree

Binary search trees are ordered tree data structures such that the values in each of their elements are greater than all values in their left child subtree and smaller than the values in their right child subtree [23]. To search for an element with a value v in such tree can be achieved by a guided DFS search. Knowing the properties of a binary search tree, for a given node, it would be easy to deduce which subtree may contain v and perform a DFS search only in that subtree. In an average case scenario, to search for an element in a tree with n nodes it takes $\mathcal{O}(\lg n)$ time. Inserting or deleting an element from such data structure takes, on average, the same amount of time. Unfortunately, there exist cases in which the running times of all operations are $\mathcal{O}(n)$. One such case is a tree in which we insert values $1, \ldots, n$ in the ascending order. Its structure would be linear, making the insertion and search operations run in $\mathcal{O}(n)$ time. In order to alleviate these shortcomings, a notion of balancing the search tree has been introduced.

One such balanced tree variant is the red-black tree [6]. It is stated in [23] that, in addition to the constraints set on an ordinary binary search tree, red-black trees have several additional requirements. All of their nodes are colored either red or black. All nodes which have at most 1 child are introduced the empty children nodes which comprise the entirety of the tree's set of leaves. The root and the newly generated leaves are colored black. Children of the each red node must be colored black. Lastly, any path from a node to a leaf contained in its induced subtree must contain the same amount of black nodes. By following these requirements, the resulting

tree structure is balanced in a way that every root-leaf path contains, roughly, the same amount of nodes, thus making the basic tree operations run in $\mathcal{O}(\lg n)$ time in every case.

One of important applications of a red-black tree is an implementation of a hash table which uses it to resolve its conflicts. Due to the self-balancing property of a red-black tree, hash table lookups (searches) and insertions are resolved in logarithmic time. Another essential application of a red-black tree will be discussed further in the chapter.

4.1.2 Suffix trie

In order to minimize the amount of redundancy during its runtime, fortuna maintains a data structure which allows it to efficiently keep track of the strings it is processing. We will use the relation "$\subseteq_c$" to denote contiguous subsequences, i.e., if $(a) \subseteq_c (b)$, we say that (a) is contiguously contained in (b). Let S be a data structure and (s) a string. We would like S to have an insert operation such that (s) is inserted into it only if there exists no string $(t) \in S$ such that $(s) \subseteq_c (t)$. If (s) is inserted into S, we want all previously contained strings (t) such that $(t) \subseteq_c (s)$ to be removed. These criteria can be met efficiently using a data structure called the suffix trie [16].

Our suffix trie implementation uses a tree data structure where each node represents an element of the sequence. Additionally, each node contains a hash table which maps elements of the sequence (keys) to children nodes with the same values. Sequences are stored such that all of their suffixes are potentially inserted into the data structure. Because of the hash tables, checking whether a sequence exists in a trie or its insertion is an operation which running time is dependent on the length of the sequence itself as well as the number of currently stored sequences. The procedure is given in Algorithm 12. Note that newly instanced suffix trie contains an empty root node which does not have a value.

Procedure *Exists* checks whether there exists a path in S which contiguously contains the entire (t) as its sequence of keys. We start from the root node and check whether its hash table contains the key for the first element $t_1 \in (t)$. If it does, we jump to that element and continue searching for t_2. This procedure continues until an element of (t) is not found or the entire (t) has been iterated through. Suppose that hash tables in each of the nodes contain at most k keys and that each key lookup is done in $\mathcal{O}(\log k)$ (using a red-black tree), then the running time of this procedure is $\mathcal{O}(|(t)| \cdot \log k)$. In our case, the set of possible keys will be finite, thus there exists a constant K such that $k \leq K$.

Now we will analyze the procedure *Insert*. It starts by selecting the empty root node and adding it to the array L. For each element of (t), iterating in the ascending order, we check

Algorithm 12 Procedures which check whether a sequence $(t) = t_1, \ldots, t_n$ exists in a suffix trie S and which insert (t) into S.

```
1:  procedure EXISTS(S, (t))
2:      n = root node in S
3:      for i = 0, ..., length((t)) − 1 do
4:          if t_i ∈ hash keys of n then
5:              n = hash table value of t_i in n
6:          else
7:              return False
8:          end if
9:      end for
10:     return True
11: end procedure
12:
13: procedure INSERT(S, (t))
14:     L = [root node of S]
15:     for i = 0, ..., length((t)) − 1 do
16:         for j = 0, ..., length(L) − 1 do
17:             if t_i ∈ hash keys in L[j] then
18:                 n = hash table value of t_i in L[j]
19:             else
20:                 n = new node with value t_i
21:                 insert n with key t_i into hash table in n
22:             end if
23:             if i > 0 then
24:                 L[j] = n
25:             else
26:                 append n to L
27:             end if
28:         end for
29:     end for
30: end procedure
```

whether it exists as a child of all nodes currently in L. If it doesn't exist, we add it to the hashtables of nodes in L. Then we update each non-root element of L and set them to their children with the value of the current element. Finally, we append t_i to L. By doing so, we are making sure that all suffixes of (t) are inserted into S as well. Time complexity of *Insert* procedure in big-$\mathcal{O}$ notation is not difficult to obtain. A sequence (t) has $\mathcal{O}(|(t)|)$ suffixes, thus we are iterating over at most $\mathcal{O}(|(t)|^2)$ elements. Each iteration consists of several constant time operations and hash table lookup and insertion which both run in $\mathcal{O}(\log k)$ time. Thus, the final running time is $\mathcal{O}(|(t)|^2 \cdot \log k)$.

4.2 Literature review

During the past 20 years, RNA sequencing techniques have vastly improved in terms of speed and cost to the point where having multiple samples sequenced and ready for analysis in a course of a single day is a widespread phenomenon. Along the traditional genomic reference assisted alignment methods [32][66][67], new transcriptome reference pseudoalignment methods [14][78][94] started to appear. Their main feature is significantly lower running time. Despite the running time discrepancies, the most popular genome reference aligner STAR outperformed the most popular pseudoaligner kallisto in terms of accuracy as reported by [31]. Another shortcoming of pseudoaligners, as pointed out by [99], is their inability to capture some of the relevant mapping information such as the exact mapping position. This leads us to the conclusion that both types of software have their merits and that it would be beneficial to somehow use them so that they complement eachother. In order to understand the complementarity at hand, we will give a small introduction to both tools in the next two subsections. Finally, we will briefly discuss the impacts of the downstream analysis of quantification.

4.2.1 STAR

STAR is a software which implements an algorithm designed to align large RNA sequencing datasets using an uncompressed suffix array approach [73] and a reference genome. It uses a seeding strategy which allows it to, to some extent, align reads correctly despite the presence of sequencing errors and genomic variations.

Given two strings s_1, s_2, suffix arrays are data structures which enable us to query whether s_1 is a subsequence of s_2 in $\mathcal{O}(|s_1| + \lg |s_2|)$ time. They are well suited for usage cases where s_2 is constant and queries are expected to be numerous, such as in the alignment problem. A suffix array is, essentially, a list of all suffixes of s_2 which holds information about the longest common prefixes of its subsequent elements. Queries can then be performed as a slightly modified binary search and are, therefore, faster than ones performed upon commonly used suffix trees [5]. The downside of a suffix array is, as reported by [73], the time which it takes for one to be constructed

- 3 to 5 times more on average than for suffix trees. For that reason, STAR constructs its suffix array once per reference genome and stores it on the disk for future use. Compared to most other aligners which use reference genome, STAR is quite fast due to a trade-off that has been made by not compressing the suffix array. The cost is increased memory usage. Additionally, STAR can perform a lot of independent computation in parallel using multiple processor cores. An example of STAR's capabilities may be found in [30].

A STAR alignment run can be divided into two phases - seed search phase and clustering, stitching and scoring phase. Let r be a read sequence, i its location and G reference genome sequence. During the first phase, for each read r, STAR sequentially searches for the maximal mappable prefix (MMP) of r which is its longest substring that matches one or more substrings of G. Once MMP has been found, this procedure is repeated for the unmapped parts of the read. Using this approach, arbitrarily spliced reads are handled in a single pass. MMPs found in this phase are used in the following phase.

The second phase of the algorithm consists of clustering of the seeds according to their distance from a set of selected anchor seeds. Anchor seeds are determined by the number of the genomic locations they have been mapped to. Generally, an anchor seed has fewer of them which could point to a greater confidence in the accuracy of their origins. All seeds that are within a user defined window around the anchor are stitched together into structures similar to transcripts. Most of the time, a single read is aligned inside a single window, but exceptions might occur where STAR considers multiple windows in order to compute an alignment. A scoring scheme which penalizes various unaligned parts of the read is used to determine the best (primary) alignment.

4.2.2 Kallisto

In contrast to STAR, kallisto searches for the compatibility between the reads and the sequences of a reference transcriptome. For a given read, it does not provide an exact mapping location, but assigns it to a set of transcripts based on how compatible their nucleotide sequences are. For that reason, kallisto is considered to be one of the pseudoalignment methods. The aforementioned sets of transcripts define equivalence classes, so every read is assigned to a class in a unique way making its pseudoalignment procedure well defined. In addition to the pseudoalignments, kallisto provides the transcript counts as one of its outputs. Its workflow consists of three steps - indexing, pseudoalignment and quantification. We will briefly discuss each of them.

Index plays a similar role for kallisto as a suffix array does for STAR. Applying the approach as seen in [22][59], using transcriptomic reference instead of the reads, kallisto's index

generation consists of building a colored De Bruijn graph. Each color, along the path of a graph, represents a single transcript. Let k be an integer constant greater than zero. Nodes represent k-mers of the reference and are possibly colored in multiple colors depending on their association with the transcripts. Multi-sets of the vertices on the same path induce k-compatibility classes to which the reads will be mapped. Once such graph has been constructed, contiguous stretches (contigs) which are colored the same have hashes of their k-mers associated to them for the ease of searching.

Given an index constructed in the previous step, alignment is a relatively simple procedure for kallisto. It pseudoaligns reads by shredding a read into k-mers and associating them with their k-compatibility classes. In the end, for a read, an intersection of all k-compatibility classes is computed which constitutes an equivalence class over the set of reads. Note that k-mers of the read belonging to the same contig carry the same information on the compatibility classes. Using the distances to the junctions going out of the contig, kallisto determines whether certain k-mers even need their hashes queried. In most cases, kallisto does a hash lookup for only two k-mers which significantly accelerates the procedure.

Once the reads have been assigned to equivalence classes, kallisto quantifies transcript abundances using a likelihood function which is iteratively optimized using the EM algorithm which has been used in the field for over half a century [20]. Since fortuna generates its own and different counts using only pseudoalignment information from kallisto, the application of this step will not be necessary.

4.2.3 Whippet

Another pseudoaligner which will be important to us as a competitor is whippet [96] due to its speed. It utilizes a highly heuristic approach with an idea similar to that of kallisto where a key difference is the fact that whippet is able to identify, but not classify, some of the novel alternative splicing events.

Whippet builds a model of the annotation into contiguous splice graphs (CSG) which are directed graphs whose vertices represent non-overlapping exonic sequences while the edges represent splice junctions between exonic regions or their adjacency. A path in such graph would represent an isoform. CSGs are supplemented by additional edges representing theoretically possible isoforms whose abundances are determined, in addition to the annotated ones, using the EM algorithm. Another limitation to the whippet's alternative splicing event discovery is its inability of detection of the novel splice junctions which are not derived from the annotation, ie., which fall within disjoint exonic and intronic regions.

Whippet builds its index by storing k-mers flanking each splice junction in its CSGs. As was the case with kallisto, whippet hashes its k-mers for $k \leq 32$ using a hashing function $h(x) = \sum_i g(x_i) \cdot 2^i$, where x, such that $|x| = k$, is a k-mer and g maps i-th nucleotide x_i of x to an integer such that A is mapped to 0, C to 1, G to 2 and T to 3. Each node in CSG contains its position and length. After the indexing, whippet computes alignments by merging all CSGs and doing a seeding step. It chooses valid seeds in each read based on the FASTQ quality scores which map to a small number of positions in the index (less than 5 by default) and, if the seeding fails, it considers the reverse complement of the read. After the seeds have been mapped, they're contiguously extended along the CSG. The alignment is considered successful if it contains less than a predefined amount of mismatches (25% of the read length is the default).

4.2.4 Downstream analysis

As pointed out by [93], one of the main challenges of RNA sequencing data analysis is to determine a set of units, like genes or transcripts, which change their expression level (abundances) when the conditions in which the organism is change (e.g. disease). Analyzing the differences between samples may provide valuable insight since most human genes which consist of multiple exons can be alternatively spliced [35][76] and it is known that deviations from the regular splicing process can have a significant impact on the organism [42].

Differential transcript usage (DTU) problem pertains to determining the changes in the abundances of isoforms linked to alternative splicing. In [93] authors classify methods that deal with it into three groups. First class consists of the assembly-based methods like the pipeline [33][83][34]. They reconstruct transcripts that best explain the sampled reads which abundances are then quantified. Second class [43] quantifies the abundance of reads that either support or do not support a predetermined type of an alternative splicing event. Third class of methods determines transcript expression by differential exon usage as a proxy [4][40]. They divide the genome into counting bins whose abundances (supported by the reads) are analyzed. This class is of importance to us because fortuna is able provide abundances of a particular kind of disjoint bins defined in the following sections, motivated by the alternative splicing. Furthermore, fortuna can provide bin abundances compatible with [4]. Apart from the already mentioned counting bins used by [4], there are a few more of note. For example, [65] defines counting bins as combinations of isoforms, [86] as exon paths traced by pairs of reads, and [14] as sets of isoforms.

Alternative mRNA splicing gives important insight into splicing induced by mutations related to autism [38] or tissue specific splicing as found in [60] which will be the focus of our experiments. An example not covered by our experiments is cancer immunotherapy [48]. It is a rather recent method which utilizes the ability of T cells to reject tumor cells by binding to their

antigenic peptides. The most prominent drawback of this method lies in our lack of knowledge of tumor-specific antigens for most cancer types [79]. Alternative splicing information, derived from the next generation sequencing transcriptomic data, has been proven to be crucial in the identification of tumor-specific mRNA processing events which increase the amount of suitable targets for immunotherapy. Mutations in splicing regulators, which may lead to abnormal alternative splicing, were detected in numerous types of cancer, such as myelodysplastic syndrome, chronic lymphocytic leukemia, breast cancer, pancreatic ductal adenocarcinoma, uveal melanoma and adenocarcinoma. In [48], it is stressed that the accurate identification of tumor-specific mRNA splicing events is of vital importance for the efficacy of these immunotherapies. This highlights the importance of selecting the right tool which can process short reads in order to discover splice junctions which are not a part of the annotation. Having this in mind, we proceed to explaining the methodology used by fortuna.

4.3 Method

In this section, we are going to give a theoretical background for the index building and post processing steps. For the sake of simplicity, each sequence of genomic features mentioned in this section will be considered to be sorted by the genomic coordinates of its elements in an ascending manner.

4.3.1 Equivalence classes of reads

Fortuna counts the number of reads falling within the equivalence classes implied by the mapping signatures of the reads. Given an annotation T and a read r of length l, a mapping signature for r is a sequence of subexons which the read spans. This definition is consistent with the ones given in [19][62]. That paves the way for a definition of equivalence between the reads which will be instrumental for defining equivalence classes.

Definition 3. Let r_1, r_2 be reads with their respective mapping signatures s_1, s_2. We say that r_1 and r_2 are equivalent if their mapping signatures are equal, i.e. $s_1 = s_2$.

Consequently, equivalent reads belong to the same equivalence class. An example can be seen in Figure 4.2. Such definition of a mapping signature preserves the information about the structure of a read allowing for the enumeration of the specific features supported by the alignment. These include junctions [96] and different alternative splicing events. Reconstruction of less granular equivalence classes proposed by [14][53] is also possible.

Kallisto considers two reads equivalent if they are compatible with the same set of transcripts, i.e., to determine an equivalence class a read r with the signature s belongs to, one may compute $\bigcap_{e \in s} t(e)$, where $t(e)$ denotes the set of transcripts subexon e is contained in. Following the

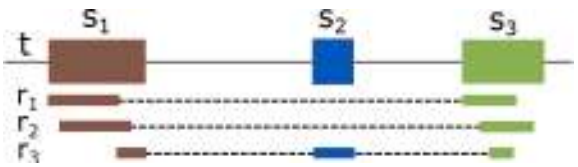


Figure 4.2: Reads r_1 and r_2 belong to the same equivalence class induced by subexons s_1 and s_3 despite their different mapping locations. On the other hand, read r_3 has a different mapping signature which includes s_2.

aforementioned procedure, it is possible to infer which kallisto's equivalence class a signature belongs to without additional read processing. Thus, given an annotation and signature counts, reconstruction of kallisto's counts is straightforward. On the other hand, a backward conversion cannot be achieved without the additional processing of all alignments since two different reads belonging to the same kallisto's equivalence class may have different mapping signatures.

Yanagi [53] is a tool which divides the transcriptome into disjoint segments used by pseudoalignment tools in order to produce segment counts instead of the transcript counts. The authors argue that using segments as equivalence classes provides an advantage over transcript based equivalence classes. Yanagi's equivalence classes are formed by merging consecutive exonic sequences in which no alternative splicing happens. The main difference between Yanagi's and kallisto's equivalence classes is that kallisto allows for the existence of exonic gaps within their equivalence classes, while Yanagi's classes are strictly connected, i.e. do not skip exonic regions. For a parameter L (which controls the degree of, so called, L-disjointness) fixed to the read length, the conversion from fortuna's equivalence classes can be done by merging consecutive subexons coming from the same set of transcripts. As it was the case with kallisto, backward conversion would require additional read processing.

As a result, both yanagi and kallisto may not offer enough insight into the features supported by the alignments without potentially costly postprocessing steps akin to the ones regularly used along with the exact genomic aligners such as [30][66].

4.3.2 Extended annotation

A common drawback of pseudoalignment tools such as kallisto is their inability to capture reads supporting events which are not annotated by some transcriptome annotation T. Here we define an extended transcriptome T' that extends T by a well-defined set of novel alternative splicing events. Let T_g be the set of transcripts annotated for a given gene g. We extend T_g by one of the sets of transcripts T_g^i, $i = 1,2,3$, where T_g^1 is defined as an empty set.

Definition 4. T_g^2 contains all transcripts t' that can be generated from a transcript $t \in T_g$ by

skipping exons in t and by modifying the boundaries of the remaining exons consistently with donor and acceptor sites observed in other transcripts in T_g. Any remaining exon in t' is a sequence of subexons $(s_1, s_2, \ldots, s_m)$ defining a continuous region along the genome such that

- there exists $i \in \{1, \ldots, m\}$ with $s_i \in t$,
- for any two subexons s_i, s_{i+1} there exists a transcript $t_1 \in T_g$ with $s_i, s_{i+1} \in t_1$,
- there exist transcripts $t_1, t_2 \in T_g$ such that s_1 defines a splice acceptor in t_1, and s_m defines a splice donor in t_2.

Definition 5. T_g^3 contains all transcripts $t = (s_1, s_2, \ldots, s_m)$, such that

- for any two subexons s_i, s_{i+1} that together define a continuous region along the genome, there exists a transcript $t_1 \in T$ with $s_i, s_{i+1} \in t_1$ or there exist transcripts t_2, t_3 such that s_i defines a donor site in t_2 and s_{i+1} defines an acceptor site in t_3,
- for any two subexons s_i, s_{i+1} that enclose a non-empty sequence gap, there exist transcripts $t_1, t_2 \in T$ such that s_i defines a splice donor in t_1, and s_{i+1} defines a splice acceptor in t_2,
- s_1 and s_m are annotated transcription start and end sites, respectively.

It is trivial to verify that $T_g^1 \subset T_g^2 \subseteq T_g^3$. The set T' can now be defined as the union of all $T_g \cup T_g^i$ for a given i, i.e.

$$T' = \bigcup_{g \text{ gene}} T_g \cup T_g^i.$$

Three sets used to extend T represent the degrees of freedom at which new transcripts are being constructed and reflect fortuna's running options. Any transcriptome extended by T_g^1 remains the same, thus this option corresponds to conventional pseudoalignment. Selecting the extension by T_g^2 results in the exon recombination strictly confined to the boundaries of the original transcripts and their exonic regions. Finally, T_g^3 removes previous restrictions and allows for recombination within the genes unhindered by existing transcript starts and ends. A partial refinement example can be seen on Figure 4.3.

4.3.3 Transcript fragments

In order to determine signature counts, reads must go through the alignment process. Fortuna does so using kallisto provided with an index generated using the information from the extended transcriptome T' and read length l. Such index should be able to capture common alternative splicing events, while retaining the information about the origin of the reads, i.e., the equivalence classes they are contained in. To that end fortuna encapsulates several signatures into more generic subexon sequences in such a way that the alignment step is computationally

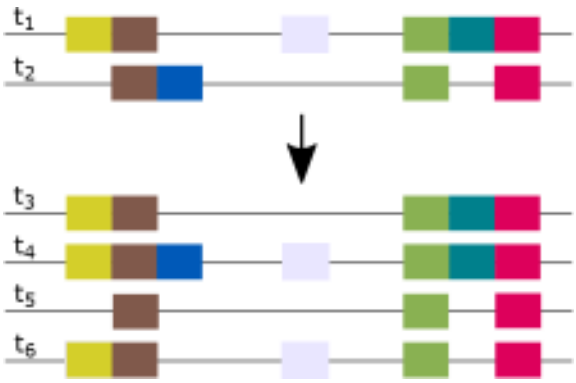


Figure 4.3: Assume that $t_1, t_2 \in T$ and we extend T by T^2. Transcript t_3 is formed by skipping an exon in t_1, t_4 by extending the donor site in t_1 with the subexon located in t_2, t_5 by shortening the donor site in t_2 and t_6 from t_1 by using the splice site found in t_2.

more efficient. Before we delve deeper into the matter, let us give a few technical definitions.

Let $(s) = (s_1, \ldots, s_n)$ be a sequence of subexons. We call subexons s_1 and s_n **boundary subexons** of (s), while subexons $s_i, i \in \{2, \ldots, n-1\}$ are called **internal subexons** of (s). We denote the set of internal subexons of (s) as $I((s))$.

We define mappings $start_g, start_t, start_e, start_i, start_s$ which return the starting genomic coordinate of their argument, respectively - gene, transcript, exon, intron and subexon. Similarly, we define mappings $end_g, end_t, end_e, end_i$ and end_s which return ending genomic coordinates of their argument. Further in the text, as a slight abuse of the notation, we will drop the indices of the previously defined mappings whenever there exists no ambiguity.

Assuming only transcripts from T' can be expressed, we propose the following theorem which will be used as a backbone for the construction of our index.

Theorem 4.1 Let l be the sample read length and assume that all transcripts in T' are expressed. For a given gene g, let S_g be the sequence of all its subexons induced by T'. A subsequence $(s) \subseteq S_g$ has a **non-zero count** if and only if it satisfies the following criteria.

(f_1) There must exist a transcript $t \in T'$ such that $(s) \subseteq_c t$.

(f_2) A read of length l can be sampled from (s):

$$\sum_{s \in (s)} |s| \geq l$$

(f_3) If $|(s)| \geq 3$, a read of length l must be able to cover all subexon junctions implied by (s):

$$\sum_{s \in I((s))} |s| \leq l - 2,$$

i.e., there exists an interval of length l within (s) which contains all junctions present in (s).

Proof. Let $(s) \subseteq S_g$ satisfy (f_1)-(f_3). According to (f_2), intervals of length l may be sampled from (s). If $|(s)| \leq 2$ its junctions can be trivially covered. Otherwise, as implied by (f_3), there exists an interval which covers all junctions in (s). Finally, due to (f_1), (s) is contiguous in some t, thus reads of length l can coincide with the intervals.

Now, lets assume that $(s) \subseteq S_g$ has a non-zero count, i.e. there is a read r of length l which has a signature corresponding to (s). Trivially, r must be sampled from the transcriptome, thus satisfy (f_1). In order for mapping to be possible, (s) must satisfy (f_2). Finally, since r covers all subexon junctions present in (s), it satisfies (f_3). □

Note that a subsequence (s) with a non-zero count may be a signature only for the reads mentioned in (f_3) which span all of its junctions. For technical purposes, let B_g be a sequence of nucleotides for a given gene g. Substring of B_g ranging from the coordinate i to the coordinate j we will denote as $B_g[i, j]$ and a sequence of nucleotides corresponding to a subexon s as $B_g[s] := B_g[\text{start}(s), \text{end}(s)]$.

Definition 6. Let $(s) = (s_1, \ldots, s_n)$ be a sequence with a non-zero count in some gene g, $K \geq 0$ and "$\cdot$" a string concatenation operator. For $n > 1$ we define mapping

$$\begin{aligned} \text{seq}((\text{s}), \text{K}) \quad &:= \quad B_g[\max(\text{start}(s_1), \text{end}(s_1) - K), \text{end}(s_1)] \\ &\cdot \quad B_g[s_2] \cdot \ldots \cdot B_g[s_{n-1}] \\ &\cdot \quad B_g[\text{start}(s_n), \min(\text{start}(s_n) + K, \text{end}(s_n))] \end{aligned}$$

and $\text{seq}((s), K) := B_g[s_1]$ for $n = 1$. If $n > 1$ it maps the subexonic sequence to its correspoding sequence of nucleotides such that the first and last subexons are trimmed to at most K bases from their, respectively, start and end. All (s) such that $n = 1$ are mapped to their exact nucleotide sequence.

A naive approach to constructing an index would be generating the following set:

$$F_{\text{naive}} = \left\{ \text{seq}\left((s),\ l - 1 - \sum_{s \in I((s))} |s| \right) : (s) \text{ has non-zero count} \right\},$$

which includes all non-zero count sequences such that all reads mapping to them are equivalent. Set F_{naive} could have many of its elements having long common subsequences, increasing the computational time required for the alignment step due to the large number of similar mapping

targets as can be seen in the example in Figure 4.4. An approach to the removal of the poten-

Figure 4.4: Assuming read length $l = 100$ and subexons have their lengths indicated on the figure, with the subexon lengths as stated in the figure, a naive approach would generate a total of 9 mapping targets with a high degree of overlap.

tially high amount of redundancy in F_{naive} would be enforcing a kind of a maximality principle upon the non-zero count sequences. Namely, if $(s) \subseteq_c (s')$ in F_{naive}, then (s) could be safely removed if the trimming procedure was abandoned by setting $K = 0$. This would address the problem of having too many mapping targets, but would introduce some new issues. There could exist multiple sequences of subexons which overlap significantly as depicted in Figure 4.5. As can be seen in the aforementioned example, by applying such approach, we lose the 1

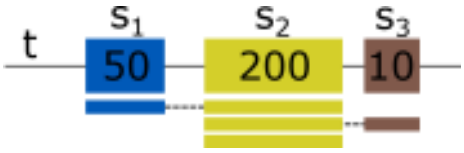


Figure 4.5: Assuming read length $l = 100$ and subexons have their lengths indicated in the figure, any read mapping exclusively to s_2 would map to 3 different mapping targets.

to 1 correspondence between the equivalence classes and mapping targets. Our novel solution to this problem includes the aforementioned maximality principle in conjuction with a specific trimming strategy. It trims long boundary subexons according to the worst case mapping which contains only the first or the last junction, but as a trade-off we have to retain single subexons which are large enough. The advantage of this approach is that each valid subexon is retained only once. It is formalized by the following definition.

Definition 7. Let $(s) = (s_1, \ldots, s_n)$ have a non-zero count and if it holds that

(f_4) either $|(s)| = 1$ or there doesn't exist a non-zero count sequence (s') such that $(s) \subseteq_c (s')$,

then we define **(transcript) fragment** f as a string of nucleotides $seq((s),\ l-1)$. For a fragment f, we will denote its corresponding sequence of subexons as $s(f)$.

It is trivial to conclude that, by using transcript fragments, one would alleviate shortcomings as introduced in Figures 4.4 and 4.5. Transcript fragments are trimmed nucleotide sequences corresponding to non-zero count subexon sequences which are either single, large enough subexons or maximal in the terms of contiguous subsequence relation. It is easy to verify that the

maximum length of a transcript fragment is $3l-4$ bases and that no subexon in a non-trivial fragment contributes to its sequence with more than $l-1$ bases.

Note that a read mapped to a transcript fragment consisting of more than two subexons might not span it entirely. Fortunately, determining an equivalence class of a read with such alignment can be done efficiently using its starting position and a simple linear search. A vast majority of the transcript fragments generated from the human transcriptome are less than 8 subexons long, making the linear search run in near atomic time. An example of the transcript fragment generation from a single transcript can be found in Figure 4.6.

Figure 4.6: Transcript fragments generation. A total of 8 transcript fragments have been generated out of which 4 represent single subexons which are large enough with regards to the read length.

Finally, for all genes, let F be a **set of all transcript fragments** over T' with regards to l which is going to be used for the creation of an index. Following theorems deal with the properties of F.

Theorem 4.2 Let l be read length and T' an extended annotation. The following claims hold for a set of all transcript fragments F over T' with regards to l.

(1) F only contains the sequences which can be derived from T', namely $\forall f \in F,\ \exists t \in T'$ such that $s(f) \subseteq_c t$.

(2) Any read of length l sampled from $t \in T'$ can be sampled from some $f \in F$, thus F is complete.

Proof. Statement (1) follows directly from (f_1). Let read r of length l cover junctions in $s(f) = (s_1,\ldots,s_n) \subseteq_c t \in T'$. Thus, f trivially satisfies (f_1)-(f_3). If f satisfies (f_4), we are done. Otherwise, there must exist $f' \in F$ such that $s(f) \subseteq_c s(f')$. Then, r can be sampled from f'. □

Theorem 4.2 proves that F, as we have defined it, retains all information from T', while not adding any additional sequences which might compromise the outcome of the alignment step.

4.3.4 Alternative splicing events

Identification of the novel alternative splicing events is the final step of fortuna's workflow which uses the alignment information provided by kallisto. Mapping information of each read

will be represented in form of segment sequences which will generalize the term of mapping signatures with regards to introns in order to capture intron retention. Alignments to introns are obtained in the postprocessing step using a genomic aligner. We distinguish two different types of segments: *intronic segments* and *exonic segments*. An intronic segment is defined as a maximal contiguous subsequence of an intronic region not intersecting with any annotated exon. We consider exonic segment to be a synonym to a subexon. Mappings *start* and *end* can be intuitively defined for segments and segment sequences.

The extended annotation T' uses known information from T in order to supplement it, but sometimes even that is not enough to capture certain types of events. For example, an unannotated transcript might exist which defines a splice site such that its flanks do not coincide with any annotated subexon boundaries or fall within an intron. For that reason, we propose using alignement information obtained by an exact genomic aligner such as [30] to further extend the annotation.

Definition 8. (**Segment refine**) Let each alignment done by a genomic aligner be defined as a sequence of genomic coordinates $((a_j, b_j))$. For each $j = 1, \ldots, n$ we define segment refinement as follows. Let (s) be a sequence of segments such that for each $s \in (s)$ holds $(start(s), end(s)) \cap (a_j, b_j) \neq \emptyset$.

- If $j = 1$, let b_j be contained within $s \in (s)$. We partition s into segments spanning intervals $(start(s), b_j)$ and $(b_j + 1, end(s))$.
- If $j = n$, let a_j be contained within $s \in (s)$. We partition s into segments spanning intervals $(start(s), a_j - 1)$ and $(a_j, end(s))$.
- If $1 \leq j \leq n$, let a_j be contained within $s_1 \in (s)$ and b_j within $s_2 \in (s)$. We partition s into segments spanning intervals $(start(s), a_j - 1)$, (a_j, b_j) and $(b_j + 1, end(s))$.

Note that some of the aforementioned intervals may be empty. In that case, we skip creating them.

Let f be a sequence of segments sorted by their genomic coordinates. We call f a **segment sequence**. For each f we define maximal consecutive subsequences $f_1, \ldots, f_n$ of f such that there exist no sequence gaps between their respective elements. Therefore, for each f_k it holds that

$$\text{start}(f_{k_i}) - 1 = \text{end}(f_{k_{i-1}}), \quad \forall i = 2, \ldots, |f_k|.$$

Note that $f_1, \ldots, f_n$ partition f. We say that a segment sequence is **intronic** if all of its elements are intronic segments. If a segment contains at least one exonic segment, we call it **non-intronic**.

Definition 9. (**Novel features**) Let f be a segment sequence with its partition $f_1, \ldots, f_n$. We classify the following two features as an **intron retention**.

(ir_1) If f_k is intronic and $n = |f_k| = 1$.

(ir_2) Let f_{k_i} be an intronic segment which is flanked by exonic segments. A novel feature, identified by $\text{start}(f_{k_i}), \text{end}(f_{k_i})$ exists if $1 < i < |f_k| \vee k = i = 1 \vee k = n, i = |f_k|$.

Let f_k and f_{k+1} be non-intronic, t an annotated transcript and an intron flanked by $\text{end}(f_k)$ and $\text{start}(f_{k+1})$ not contained in any annotated transcript. We classify the following features flanked by $\text{end}(f_k), \text{start}(f_{k+1})$ with regards to t.

(es) If there exist exonic regions e_1, e_2, e_3 in t such that $e_1 \cap f_k, e_3 \cap f_{k+1} \neq \emptyset$ and $e_2 \cap f_k, e_2 \cap f_{k+1} = \emptyset$, the feature is an **exon skipping**.

(ad) If there exist exonic regions e_1, e_2 in t such that $f_k \cap e_1 \neq \emptyset$ and $\text{start}(f_{k+1}) = \text{start}(e_2)$, the feature is an **alternative donor site**.

(aa) If there exist exonic regions e_1, e_2 in t such that $f_{k+1} \cap e_2 \neq \emptyset$ and $\text{end}(f_k) = \text{end}(e_1)$, the feature is an **alternative acceptor site**.

(ap) If there exist exonic regions e_1, e_2 in t such that $e_1 \cap f_k, e_2 \cap f_{k+1} \neq \emptyset$, $\text{end}(f_k) \neq \text{end}(e_1)$ and $\text{start}(f_{k+1}) \neq \text{start}(e_2)$, the feature is an **alternative splice sites pair**.

(ie) If there exists an exonic region e in t such that $e \cap f_k, e \cap f_{k+1} \neq \emptyset$, the feature is an **intron in exon**.

Any novel feature that satisfies none of the aforementioned criteria we classify as **unknown**.

Let t_1, t_2, t_3 be transcripts in T with subexons $s_1, \ldots, s_8$ as depicted in Figure 4.7. If there exists a read spanning the junction between s_1 and s_7, it would be regarded as a novel exon skipping (ES). Similarly, s_3 and s_5 span a novel alternative donor site (AD), s_4 and s_6 a novel acceptor site (AA), s_5 and s_8 a novel alternative pair (AP), s_1 and s_3 a novel intron in exon (IE) while the subexons s_6 and s_7 including the intron between them constitute a novel intron retention (IR).

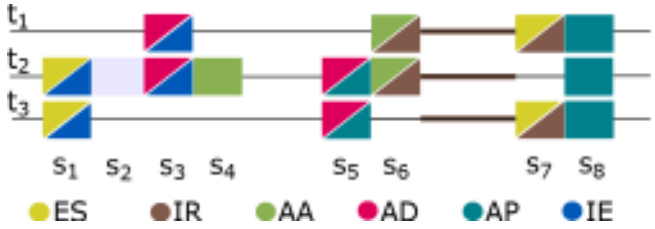


Figure 4.7: Alternative events classification. Subexons and introns have been colored according to which novel alternative splicing event they belong to. The events are exon skipping (ES), intron retention (IR), alternative acceptors (AA), alternative donors (AD), alternative pairs (AP) and intron-in-exon (IE).

4.4 Implementation

As was the case with Trajan, fortuna has been implemented in C++ programming language. As was previously mentioned, it runs in three steps: index generation, alignment and postprocessing (refinement). In order to construct an index, fortuna generates a set of transcript fragments using a GTF file with the annotation, a FASTA file with chromosomial sequences and a few runtime options such as the selection of a set used to extend the annoataion.

A chromosomial FASTA file is parsed and its chromosome identifiers (names) mapped to their corresponding nucleotide sequences using a hash table. Such hash table usually has between a dozen and a hundred entries so the collisions are easily resolved with a string hashing function provided in the standard C++ library. GTF file is parsed into a data structure that is a little bit more complex. All subexons found in the GTF annotation file have their relevant information encapsulated - their genomic coordinate starts and ends, lists of genes and transcripts they belong to. Afterwards, all subexons belonging to the same chromosome are arranged as the nodes of a red-black tree using their starting coordinates as keys. Note that, due to the disjointness property of the subexons, such an arrangement is feasible. Furthermore, hashtables are being generated which map gene and transcript names to the lists of their respective subexons. This enables fortuna to quickly retrieve the information related to any subexon or to fetch the entire sequences.

According to the selection of the annotation refinement set (T^1, T^2 or T^3) fortuna traverses a directed graph G using a depth-first search algorithm in order to generate transcript fragments which will constitute an index. In the case where the annotation is refined using T^1 or T^2, a directed graph G is constructed for every transcript such that its every node is connected to its direct successor, genomic coordinate-wise. If the refinement is done using T^3, graphs G are constructed for every gene in a manner identical to the ones constructed using two other sets. An example can be seen in Figure 4.8. In order to generate transcript fragments, each path of the graphs G is traversed. Transcript fragments which satisfy (f_1)-(f_4) are recorded as the series of corresponding subexons. A suffix trie data structure, using subexons identifiers as its nodes, is used to ensure the maximality and uniqueness of every generated fragment. The formal procedure is given in Algorithm 13.

Index generation step concludes with writing down the nucleotide sequences of every transcript fragment in form of a FASTA file and finalizing the index generation by running kallisto. Then, reads (FASTQ format) are passed to kallisto along with an index so that the alignment step can be performed. Each read aligned by kallisto, which can be done using multiple threads, is passed to fortuna for further processing. With an alignment information of a read, fortuna can detect which equivalence class it belongs to or if it supports any of the aforementioned alternative

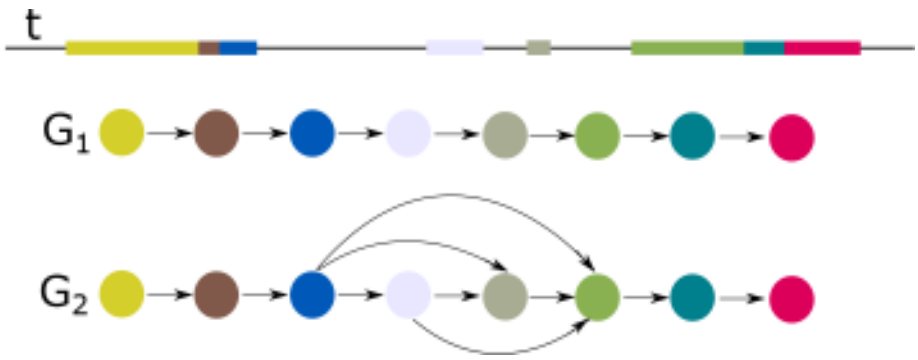


Figure 4.8: Graph G_1 is derived from the transcript t using set T^1 to extend the annotation. In order to construct G_2, transcript t was first refined into additional transcripts using T^2.

splicing events. Furthermore, fortuna can asses the quality of the alignment by checking the number of base pair mismatches between the read sequence and the sequence it was mapped to. Reads which weren't aligned or were poorly aligned by kallisto can be outputted into a FASTQ file which can be used for realignment in the final step of this method.

First part of fortuna's alignment processing is determining the equivalence class of the read. It is done by subexon enumeration in a fragment with a simple linear search from the alignment starting coordinate until the end of the read. Each junction the read spans is then tested whether it supports an alternative splicing event. The procedure, formalized in Algorithm 14, starts by grouping subexons the read spans into segment sequences as previously defined. Between each of the subsequences there lies an intron. If the intron is novel, we classify which event it belongs to. We consider an intron to be novel if it does not exist in the original annotation, that is if its flanks are not consecutive in any annotated transcript. Note that we do not check for intron retention events yet due to our transcript fragments not containing any intronic parts. For any annotated transcript t which contains at least one subexon from the subsequences flanking the intron, we check whether it supports a novel event by our definition. If the event is successfully classified, the contribution of the read is added to a hash table. Once the junction has been identified as novel, if we encounter it later in some other read, using the fast hash table search, we don't have to classify it again. In that case, it is enough to increase the count of the already existing entry in the hash table.

Another difficulty which arises during the alignment step is the coordinate conversion between kallisto's transcriptomic coordinates and the genomic coordinates used in the annotation. Since kallisto is only aware of the index constructed from a set of transcript fragments, its alignment coordinates are reported relative to them. We devised Algorithm 15 in order to convert transcriptomic coordinate x into the genomic coordinate originating either from the transcript or transcript fragment on forward or reverse strand. Transcripts or transcript fragments may

Algorithm 13 Transcript fragment generation. Assume that S was generated as an empty suffix trie and that graphs G are generated as already explained in the text. Let l be the sample read length. Procedure DFS is initialized with $V =$ first of G.

```
1:  procedure DFS(G, l, S, V)
2:      s = last of V
3:      if fragment [s] doesn't exist in S then
4:          add [s] to S
5:      end if
6:      if |I(V)| ≥ l − 1 then
7:          return
8:      end if
9:      for successor t of s in G do
10:         DFS(G, l, S, [t])
11:         DFS(G, l, S, append t to V)
12:     end for
13:     if |I(V)| ≤ l − 2 or |V| = 2 then
14:         add V to S
15:     end if
16: end procedure
```

be trimmed or not. Coordinates on the reverse strand are counted from the end of the feature and are first converted into the coordinates on the forward strand. Afterwards, forward strand coordinate x is adjusted for trimming of the left-most subexon. Finally, starting from it, we are counting (adjusted) x bases along the, possibly non-contiguous, subexons and returning the relevant coordinate.

To reiterate, all reads which have remained unmapped or which mapping quality has been insufficient can be written down in a separate FASTQ file so they can be remapped using a genomic aligner. In our experiments, we have used STAR. Once the genomic aligner maps previously unmapped reads, fortuna processes its outputs. This step is similar to the processing of kallisto's alignments with the only difference being the procedure which further refines the segments (both introns and subexons) as stated in Definition 8. An algorithm which refines the segments is given in Algorithm 16. Prior to its execution, intronic segments are being created and added to red-black trees representing the annotation. Due to that, intron retention events may only be identified and quantified during this step.

There are two main outputs fortuna returns - a count file and an alternative splicing file. Count file lists the abundances of the equivalence classes it has encountered, while the alternative splicing file contains all the information pertaining to the novel alternative events it has found. Count files may be converted to different formats, possibly more suitable for the downstream analysis, using extra tools bundled with fortuna. Fortuna may also output alignments in BAM format.

Algorithm 14 Alternative splicing event identification. Inputs A and B are consecutive segment sequences of some read r.

procedure ALTERNATIVESPLICING(A, B, $EventTable$)
    l = last of A
    r = first of B
    if (l,r) in $EventTable$ **then**
        increase count of (l,r) in $EventTable$ by 1
        return
    end if
    T = intersection of sets of transcripts of l and r
    for $t \in T$ **do**
        for $i = 1,\ldots,|t|-1$ **do**
            if $t[i] = l$ and $t[i+1] = r$ **then**
                return
            end if
        end for
    end for
    T_1 = union of all transcripts of subexons in A
    T_2 = union of all transcripts of subexons in B
    $T = T_1 \cap T_2$
    if $T = \emptyset$ **then**
        return
    end if
    if all subexons of A or B are intronic **then**
        add (l,r) to $EventTable$ as unknown event
        return
    end if
    for transcript t in T **do**
        a = left flanking subexon of region intersecting A in t
        b = right flanking subexon of region intersecting B in t
        if $a < 1$ or $b < 1$ **then**
            continue
        else if $b < a$ and $a - b > 1$ **then**
            add (l,r) to EventTable as novel intron in exon event w.r.t. t
        else if $a \neq l$ and $b = r$ and $b - a = 1$ **then**
            add (l,r) to EventTable as novel alternative donor event w.r.t. t
        else if $a = l$ and $b \neq r$ and $b - a = 1$ **then**
            add (l,r) to EventTable as novel alternative acceptor event w.r.t. t
        else if $a \neq l$ and $b \neq r$ and $b - a = 1$ **then**
            add (l,r) to EventTable as novel alternative pair event w.r.t. t
        else if $b - a > 1$ and $start(a+1) - end(l) > 1$ and $start(r) - end(b-1) > 1$ **then**
            add (l,r) to EventTable as novel exon skipping event w.r.t. t
        end if
    end for
end procedure

Algorithm 15 Conversion of transcriptomic into genomic coordinates. Let T be a transcript fragment with its list of subexons, x a transcriptomic coordinate in T, l read length, *rev* indicating if T is on a reverse strand and *trim* indicating if T should be trimmed. Procedure returns -1 if the coordinates cannot be converted.

procedure TRANSCRIPTOMIC2GENOMIC(T, x, l, *rev*, *trim*)
  $trim = trim$ and $|T| > 1$
  $tl = trim$ and $|\text{first of } T| \geq l$
  $tr = trim$ and $|\text{last of } T| \geq l$
  if rev **then**
    $c = 0$
    for $i = 0, \ldots, |T| - 1$ **do**
      $c = c + |T[i]|$
    end for
    $x = c - (tl + tr + 1) \cdot (l - 1) - x + 1$
  end if
  if tl **then**
    $x = x + |\text{first of } T| - l + 1$
  end if
  for $i = 0, \ldots, |T| - 1$ **do**
    if $x \leq |T[i]|$ **then**
      return $\max(-1, \text{start}(T[i]) + x - 1)$
    else
      $x = x - |T[i]|$
    end if
  end for
  return -1
end procedure

Algorithm 16 Segment refinement procedure. Let T represent the red-black tree used to store the annotation for the chromosome the read was sampled from, v contain lengths of the alternating exonic and intronic parts of the read (parsed CIGAR string) and s be the alignment starting position. New nodes that are created as subsets of already existing nodes inherit their properties such as their affiliation to transcripts and genes.

procedure REFINE(T, v, s)
  for $i = 0, \ldots, |v| - 1$ **do**
    if i is even **then**
      $n =$ subexon in T containing coordinate $s + 1$
      $m =$ subexon in T containing coordinate $s + v[i]$
      if start$(n) = s + 1$ and end$(m) = s + v[i]$ **then**
        continue
      end if
      if $n = m$ **then**
        if $i = 0$ and $s + v[i] <$ end(n) **then**
          remove n from T
          add new node with coordinates $(\text{start}(n), s + v[i])$ to T
          add new node with coordinates $(s + v[i] + 1, \text{end}(n))$ to T
        else if $i = |v| - 1$ and start$(n) < s + 1$ **then**
          remove n from T
          add new node with coordinates $(\text{start}(n), s)$ to T
          add new node with coordinates $(s + 1, \text{end}(n))$ to T
        else if $0 < i < |v| - 1$ **then**
          remove n from T
          if start$(n) < s + 1$ **then**
            add new node with coordinates $(\text{start}(n), s)$ to T
          end if
          add new node with coordinates $(s + 1, s + v[i])$ to T
          if $s + v[i] <$ end(n) **then**
            add new node with coordinates $(s + v[i] + 1, \text{end}(n))$ to T
          end if
        end if
      else
        if $i > 0$ and start$(n) < s + 1$ **then**
          remove n from T
          add new node with coordinates $(\text{start}(n), s)$ to T
          add new node with coordinates $(s + 1, \text{end}(n))$ to T
          $m =$ subexon in T containing coordinate start(m)
        end if
        if $i < |v| - 1$ and $s + v[i] + 1 <$ end(m) **then**
          remove m from T
          add new node with coordinates $(\text{start}(m), s + v[i])$ to T
          add new node with coordinates $(s + v[i] + 1, \text{end}(m))$ to T
        end if
      end if
    end if
    $s = s + V[i]$
  end for
end procedure

4.5 Experiments

All of the experiments presented in this section have been conducted on a server computer which we have already described in the previous chapter. The entire testing environment remained the same.

4.5.1 Simulated data

In order to simulate transcript extrapolation from the incomplete annotation, we have used Flux Simulator [44] to simulate reads from a complete annotation and proceeded to remove transcripts in a way which produced novel alternative splicing events consistent with our definitions. Six 80 million read datasets with read lengths 75, 100 and 125, with and without simulated sequencing errors, were simulated. In order to detect possible events, we have used ASTALAVISTA [47] to identify and enumerate transcripts in which they occur. ASTALAVISTA gave us flanking exonic coordinates of several alternative event types, namely exon skipping, alternative donors and acceptors. According to that data, we have determined which transcripts are to be removed from the annotation. Once the events have been identified and parts of the annotation hidden, we have checked the correctness of the mapping of the reads which were sampled from the removed transcripts.

Let T_e be a set of transcripts which contain exonic regions e_1, e_2, e_3 as described in the (es) part of the novel features definition and T_f transcripts containing f_k and f_{k+1}. By our definition, any read splicing over an intron between e_1 and e_3 would not indicate a novel feature unless all of the transcripts from T_f were removed. Thus, in order to artificially introduce a novel exon skipping event we remove all of the transcripts in T_f from the annotation. As a result, any read supporting the junction between any of the regions e_1 and e_3 also supports a novel exon skipping event.

Consistently with the alternative donor (ad) part of the definition of the novel features, we define sets of transcripts T_e, containing exonic regions e_1, e_2, and T_f containing segment sequences f_k and f_{k+1}. Just like in the case of exon skipping, we can artificially induce a novel event signature by removing all of the transcripts in T_f from the annotation. Following the procedure, it is clear that for every exonic region e_1 it holds that $end(e_1) \neq end(f_k)$. Thus, in order to capture aforementioned alternative donor event (without using additional segment refinement) fortuna has to construct T' using either T_g^2 or T_g^3 and there must exist a transcript having an exonic region ending in the same genomic coordinate as f_k while not containing the exact intron between f_k and f_{k+1}.

Novel alternative acceptor events can be introduced under the similar conditions as novel al-

ternative donor events following the (aa) part of the novel splicing event definition by removing all transcripts present in T_f using T' constructed in the same way and having a transcript containing an exonic region starting with $start(f_{k+1})$ and such that it doesn't contain an intron between f_k and f_{k+1}. Sets T_f for both novel alternative donor and acceptor events can be constructed in a way which either the novel donor site appears within known exonic regions (shortening of the splice site) or continues onto an intron (prolonging the splice site). We will distinguish those two cases in our experiments and report the results separately.

The aim of this experiment is to count the reads which mappings support the aforementioned alternative splicing events. Since the ground truth (output of FLUX simulator) is known, we can easily compare any mappings to it. A mapping of a single read is considered valid if it spans the junction which supports the event according to the ground truth and if it is mapped to it by the aligner. If the read is multimapped, we consider it valid if one of its mappings spans the junction of focus. This means that a read is considered valid if one or more of its alignments span the novel junction without the need for a said alignment to exactly match the ground truth. Using these criteria, we have compared fortuna to whippet, kallisto and STAR. STAR was running in a regular and two-pass mode in which it refines its index according to the structure of the mapped reads. Thus, at the cost of longer running time, STAR gains more accuracy. Since kallisto is unable to map any reads supporting novel splicing events it is omitted from the results which are presented in the Table 4.1 and was used only as a sanity check. Along the aforementioned criteria of validity of the mapping, we have included the number of reads which primary alignments exactly match the ground truth.

The results concerning datasets without errors suggest that STAR, in both modes, maps more reads over the observed splice junctions than fortuna or whippet. But, according to the ground truth, it may also produce a lot of incomplete alignments by partially aligning some of the reads. Partially aligned reads are aligned without a small-to-medium amount of base pairs on either ends of the read in a procedure called soft clipping. Fortuna does not soft clip reads by design. Whippet soft clips between approximately 3.1% and 7.8% of the reads where larger numbers are attained for longer reads and datasets with errors. STAR run in both single-pass and two-pass mode soft clips between 0.6% and 0.8% of the reads coming from datasets without errors, while it soft clips between 11.9% and 18.6% reads coming from datasets with errors, increasing with read length. While processing samples with errors, fortuna seems to perform better than other tools which makes it especially useful for samples with high degree of errors due to plethora of technological or biological reasons. A case which we would like to point out are mutations observed in autism patients which are the focus of the next chapter.

Despite being the only of the considered tools which is able to classify novel alternative splicing events in one of its outputs, fortuna has overall lower running times as well. They may

be found in Table 4.2. We did another running time analysis in the next section using real data.

4.5.2 Autism data

We have acquired 36 autism patient blood-derived lymhpoblast cell line samples previously used by [38] which consist of approximately 300 million 151 base pair long reads. Authors of [38] have implemented a deep residual neural network [55] which, using only gene sequences, is able to predict locations of splice sites. In order to predict splice donors and acceptors, SpliceAI takes a novel approach which checks a window of ten thousand base pairs around each candidate, as opposed to small windows used in previous research [104]. As predicted by SpliceAI, several samples exhibit novel alternative splicing in certain genes. We have run fortuna, whippet and STAR to record the number of reads which support verified events and have compared them to eachother and to the results presented in [38]. Later, we have randomly subsampled the reads from one sample (sample 29) with 10% increments in the sample size in order to capture and compare running times between competing tools.

Samples 29, 12, 36, 26, 20, 4, 30 and 28 have been reported to contain alternative exon skipping events which possibly happen due to a mutation of a single base pair at the splice acceptor site. The alternative event found in sample 28 is annotated in the GRCh38 annotation used in the experiment and has, hence, been omitted from the analysis. Using the alignment data from the outputs of the aforementioned tools, we have created sets of reads aligned over the novel exon skipping junctions. A read was considered to be aligned over the observed junction if at least one of its alignments spanned it. In samples 12 and 26, SpliceAI reported a single read supporting the novel exon skipping event which has been corroborated by other tools. In each of the remaining samples, fortuna has found the most reads supporting novel junctions. Venn diagrams corresponding to samples 4 and 20 are depicted in Figure 4.9. We can see that all tools agree on a large percentage of the reads. It is interesting to point out that STAR doesn't make any alignments which can't be confirmed, at least, by fortuna. Reads mapped by fortuna which have exhibited a large number of mismatches according to the reference ($> 5\%$) have undergone the realignment process. Also note that a vast majority of the reads aligned by fortuna and not by other two tools have been aligned without any mismatches. Other tools have either left them unmapped (as is the most frequent case with whippet) or have soft clipped the novel part of the read. An important thing to mention is that SpliceAI has, compared to other tools, underreported the number of reads supporting the junctions. Complete data can be found in Table 4.3.

SpliceAI has found evidence of novel alternative donor and acceptor events in samples 11, 27, 25, 34, 9, 1, 31, 7 and 15. As was the case with exon skipping samples, we omit sample 34 due to the reported junction not being novel in the new annotation. Once again, we have compared the same three tools. In none of the samples has whippet mapped any reads over the

Table 4.1

Number of correctly mapped reads supporting alternative splicing events for samples with 75, 100, 125 base pair long reads simulated with and without errors. Table headers contain a total number of such reads according to the ground truth while each field contains both number of valid reads and number of exact matches amongst the primary alignments, in that particular order. Exon skipping events are labeled *es*, prolonged donor sites are labeled ad_1 and acceptors aa_1, shortened sites have index 2.

75bp	*es* (242886)	aa_1 (13418)	ad_1 (17424)	aa_2 (24698)	ad_2 (13592)
fortuna	239684, 231299	13359, 12896	17293, 17145	24539, 24380	13356, 13175
star	241166, 222350	13336, 12128	17201, 16299	24489, 22640	13427, 12541
star2	241872, 237262	13347, 13008	17250, 17198	24509, 24174	13462, 13366
whippet	234251, 229184	12547, 11310	17117, 16901	24243, 23672	13264, 12801

75bp errors	*es* (243287)	aa_1 (13341)	ad_1 (16970)	aa_2 (24592)	ad_2 (13626)
fortuna	236501, 218775	13064, 12102	16573, 15884	24024, 23225	13157, 12603
star	230215, 187548	12633, 10099	15950, 13359	23242, 18892	12820, 10475
star2	233373, 203351	12795, 10974	16176, 14293	23457, 20432	12975, 11347
whippet	217566, 206239	11586, 10044	15462, 14766	22648, 21145	12540, 11508

100bp	*es* (348943)	aa_1 (18895)	ad_1 (25398)	aa_2 (35833)	ad_2 (19607)
fortuna	337991, 325506	18653, 18014	24962, 24526	35381, 35020	19094, 18789
star	345131, 324534	18650, 17393	24951, 24071	35342, 33412	19226, 18318
star2	345709, 339205	18650, 18278	24955, 24859	35346, 34823	19248, 19127
whippet	334016, 325231	17712, 15963	24828, 23924	35032, 34148	18970, 18367

100bp errors	*es* (348611)	aa_1 (19028)	ad_1 (25533)	aa_2 (35644)	ad_2 (19600)
fortuna	335821, 308517	18627, 17138	24929, 23338	34957, 33595	18986, 18107
star	326517, 259836	17855, 14086	23030, 19355	33329, 26698	18251, 14707
star2	329843, 274260	17994, 14963	23765, 20173	33535, 28058	18381, 15426
whippet	308886, 279673	16642, 13879	23982, 20628	32653, 29245	17899, 15842

125bp	*es* (451135)	aa_1 (24527)	ad_1 (32930)	aa_2 (44970)	ad_2 (25009)
fortuna	426718, 411471	23637, 22828	31887, 31368	43293, 42718	23969, 23616
star	441191, 421269	23951, 23003	31925, 31096	43780, 41987	24257, 23642
star2	441691, 435071	23951, 23665	31929, 31819	43797, 43160	24285, 24104
whippet	425943, 411885	22731, 20505	31774, 30764	43403, 42082	23913, 22949

125bp errors	*es* (451370)	aa_1 (24637)	ad_1 (32766)	aa_2 (44603)	ad_2 (24928)
fortuna	424948, 388665	23618, 21518	31565, 29553	42683, 40844	23779, 22478
star	418024, 324441	21202, 17818	30028, 23800	41076, 31883	22915, 18132
star2	421036, 338300	22771, 18469	30224, 24530	41326, 33019	23050, 18635
whippet	395424, 341640	22905, 17010	29159, 25323	40420, 34688	22529, 18942

Table 4.2
Running times of alternative splicing event experiments for samples with 75, 100, 125 base pair long reads simulated with and without errors. Exon skipping events are labeled *es*, prolonged donor sites are labeled ad_1 and acceptors aa_1, shortened sites have index 2.

75bp	*es*	aa_1	ad_1	aa_2	ad_2
fortuna	12m17.904s	12m29.752s	12m13.111s	12m33.732s	12m30.750s
kallisto	17m48.151s	20m51.704s	20m54.900s	21m3.827s	21m25.023s
star	21m55.824s	21m58.139s	23m6.271s	22m55.509s	22m14.256s
star2	41m26.362s	41m2.229s	41m40.723s	41m11.213s	42m29.149s
whippet	29m0.562s	28m58.884s	29m15.039s	28m54.749s	29m13.531s

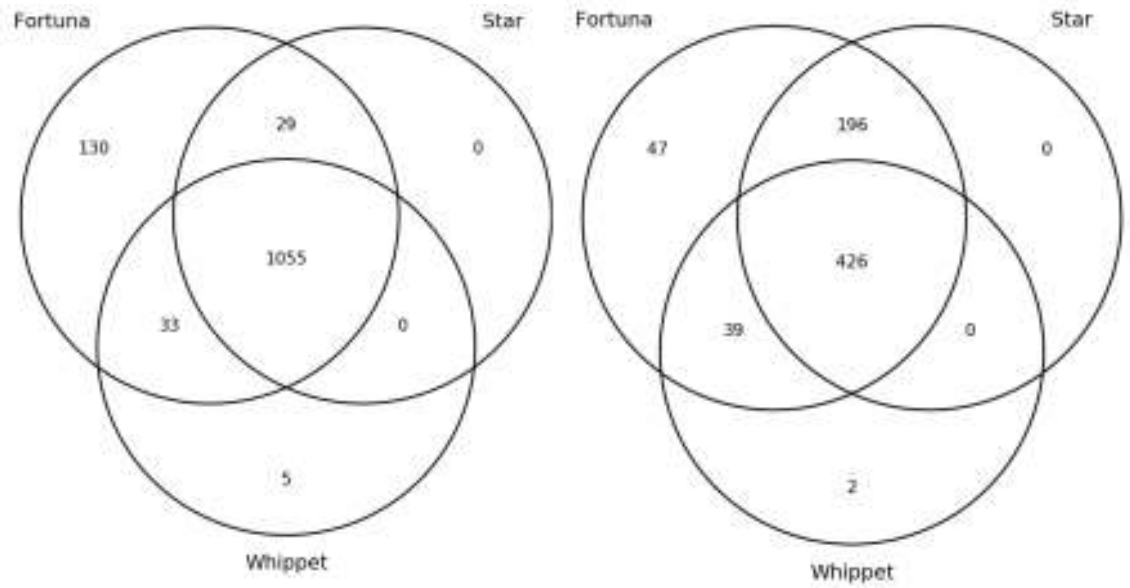
75bp errors	*es*	aa_1	ad_1	aa_2	ad_2
fortuna	27m53.544s	21m10.048s	21m40.163s	20m55.247s	20m57.929s
kallisto	37m38.476s	35m34.215s	34m34.273s	34m6.870s	35m1.809s
star	42m2.961s	36m20.777s	31m38.968s	32m27.137s	31m39.605s
star2	77m0.043s	59m36.670s	57m18.302s	61m23.681s	60m1.966s
whippet	44m32.004s	32m53.693s	32m30.783s	32m21.383s	32m10.026s

100bp	*es*	aa_1	ad_1	aa_2	ad_2
fortuna	23m23.167s	20m20.174s	23m41.568s	17m59.024s	17m30.254s
kallisto	24m24.915s	23m37.856s	32m24.385s	23m30.950s	23m24.518s
star	29m44.741s	25m54.309s	33m12.314s	28m23.682s	26m55.305s
star2	55m36.642s	47m48.429s	49m20.641s	48m11.642s	48m43.638s
whippet	33m42.932s	32m58.751s	34m12.169s	35m24.516s	37m47.166s

100bp erors	*es*	aa_1	ad_1	aa_2	ad_2
fortuna	28m37.210s	29m50.084s	29m48.754s	33m15.476s	29m33.163s
kallisto	31m44.960s	38m33.365s	38m28.012s	37m20.552s	38m43.292s
star	38m10.621s	38m7.352s	38m0.674s	37m53.737s	36m58.683s
star2	70m54.751s	70m56.477s	72m17.699s	70m17.344s	68m19.124s
whippet	39m27.184s	39m19.787s	39m48.310s	38m42.819s	40m32.371s

125bp	*es*	aa_1	ad_1	aa_2	ad_2
fortuna	25m33.415s	23m4.344s	22m49.780s	22m7.908s	28m16.577s
kallisto	19m52.587s	25m14.996s	25m31.716s	24m57.598s	25m49.526s
star	28m56.754s	29m8.586s	32m17.154s	29m30.665s	37m36.706s
star2	57m48.283s	57m3.265s	58m19.000s	55m14.296s	56m7.874s
whippet	38m13.330s	38m10.261s	38m48.814s	39m15.133s	38m33.468s

125bp errors	*es*	aa_1	ad_1	aa_2	ad_2
fortuna	40m8.152s	41m27.401s	43m46.530s	42m48.737s	41m48.747s
kallisto	36m10.403s	45m39.963s	44m34.863s	44m2.556s	44m46.588s
star	44m40.586s	45m28.693s	44m48.631s	43m44.401s	44m7.706s
star2	80m45.134s	83m27.190s	81m55.356s	83m59.184s	82m26.148s
whippet	47m22.823s	47m17.675s	46m59.106s	55m55.368s	47m56.444s



(a) sample 4 **(b)** sample 20

Figure 4.9: Venn diagrams representing number of reads supporting novel exon skipping events in autism patient samples as found by fortuna, STAR and whippet.

novel junctions, while STAR and fortuna (after realignment) agree on all of their findings. The results can be found in Table 4.4.

In order to produce counts and identify novel splicing events, alignments done by STAR have to be processed by counting tools such as SplAdder [62], which is the most common one. Thus, the rate at which we can process samples is much lower if using STAR than fortuna or whippet. In fact, since STAR's implementation is quite efficient and considering the fact it can be run using multiple threads, the main bottleneck of such analysis would be SplAdder. This is even more evident if taking into account SplAdder's requirement for a sorted and indexed

Table 4.3
Read sets of aligned novel exon skipping junctions by fortuna (F), STAR (S) and whippet (W) and various sets used to depict Venn diagrams. Sets $S \setminus (F \cup W)$ and $(W \cap S) \setminus F$ have been omitted due to being empty in all samples. Number of reads aligned by SpliceAI is denoted as AI.

#	AI	F	S	W	$F \setminus (S \cup W)$	$W \setminus (S \cup F)$	$(F \cap W) \setminus S$	$(F \cap S) \setminus W$	$F \cap W \cap S$
29	45	107	99	96	8	0	3	0	96
12	1	2	2	2	0	0	0	0	2
36	483	763	619	620	134	26	35	10	584
26	1	1	1	1	0	0	0	0	1
20	536	708	622	467	47	2	196	39	426
4	914	1247	1084	1093	130	5	29	33	1055
30	120	971	812	890	72	8	17	87	795

Table 4.4
Read sets of novel donor and acceptor junctions, aligned by fortuna and STAR. Whippet has been omitted due to not mapping any reads supporting those junctions.

Sample ID	11	27	25	9	1	31	7	15
SpliceAI	15	9	6	12	4	12	67	3
fortuna and STAR	14	18	9	19	4	10	110	2

STAR output which can be done by samtools [36]. On the figure 4.10 we can see that STAR and SplAdder are much slower than fortuna and whippet. Fortuna's advantage over whippet becomes even more apparent if it's run using multiple threads (4 in our experiment) since the current implementation of whippet doesn't support multi-threading.

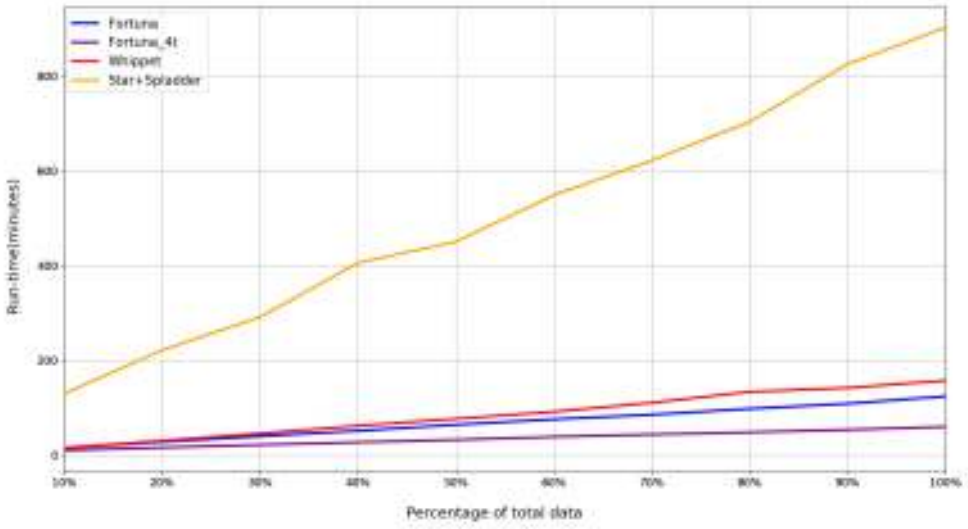


Figure 4.10: Alignment and counting time for a subsampled FASTQ file containing approximately 300 million 151 base pair long reads collected from autism patient sample.

To summarize, we have been testing how well does fortuna compare against STAR and whippet on the real-world autism patient samples. Despite using pseudoaligner as its backbone, fortuna has been superior to other tools both in terms of running time and the quality of the alignment in situations where mutations are present.

4.5.3 *Drosophila* data

Fortuna was tested on on sequencing data generated from different *Drosophila* (fruit fly) pupal tissues, including dissected brain, indirect flight muscle (IFM) and whole leg. It is an interesting species for genetic research due to its low cost maintenance, short life cycle, large

number of offspring and the ease of editing of its genes [60]. Additionally, alternative splicing in *Drosophila* is as complex as that of the mammals [98]. That makes it suitable for testing fortuna as well. Using the aforementioned samples, fortuna identified thousands of novel splicing events across hundreds of genes. Only about 10% of these events were identified with the assistance of the genomic aligner (novel splice acceptor or donor sites). The rest of the novel events were novel junctions between annotated exons. The most common novel events were exon skippings (over 80%) followed by alternative donor and acceptor events, as can be seen in Figure 4.11. About 2% of novel events were supported by 150 or more reads, which will be the focus of further analysis. Notably, over 60% of the events are supported by no more than 5 reads for which we consider to be the noise in the data.

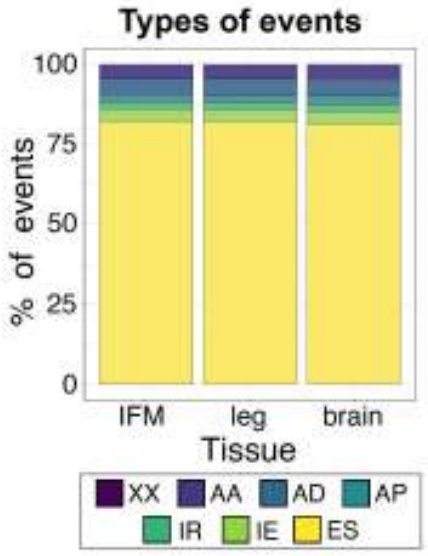


Figure 4.11: A ratio of detected novel alternative splicing events in *Drosophila* samples. The events are exon skipping (ES), intron retention (IR), alternative acceptors (AA), alternative donors (AD), alternative pairs (AP), intron-in-exon (IE) and unknown (XX).

Of all novel events supported by 150 or more reads, only 7.5% were common to all three tissues and 13%, 15% and 48% are specific to IFM, leg and brain, respectively. Genes in which these events occur are diverse and reflect tissue-specific functionalization. Events that are not tissue-specific are related to common processes such as cytoskeleton organization and behavior. Brain-specific events are related to synapse organization, neuron recognition and ion transport. IFM and leg events are related to muscle-specific processes such as actomyosin structure organization, oxidation-reduction process, flight and mesoderm development. We have found many novel events in genes that undergo tissue-specific alternative splicing. Some of those are essential sarcomere proteins such as Mhc, bt, Unc-89, up, Tm1, wupA, Strn-Mlck and sls in IFM or leg samples which all exhibit over 10 novel events. Similar case was identified with the

brain genes kcc, slo, Atpα, CaMKII, brp, Cadps, stj and Rdl. Novel events in genes Bru1 and bt have been experimentally confirmed.

This experiment adds further weight to our claim that fortuna may be useful in detecting biologically relevant alternative splicing events, especially ones that are not represented in existing transcriptome annotations.

CHAPTER 5

Conclusion

We have presented two software tools - Trajan and fortuna - which implement novel algorithms for solving problems in the field of bioinformatics. Their properties have been described in detail and performance measured by the experiments undertaken on both real-world and simulated data.

Trajan is an efficient software tool used to quantify the distance between two phylogenetic trees by computing a maximal cost matching between them which respects the ancestral relations between their nodes. A term arboreal matching is adopted for a class of valid matchings to separate Trajan from the algorithms which produce solutions that do not honor ancestral relationships between the nodes of the trees. The importance of this algorithm is reflected in numerous applications it may have, beyond the phylogenetic trees, for the comparison of hierarchical structured which represent tumor subclones formed during tumor evolution, protein-protein interaction networks reconstructred by hierarchical clustering methods, co-evolution and even string matching. The underlying mechanism which powers Trajan is a natural evolution of the widely used Robinson-Foulds metric given in an integer linear programming formulation. First, a naive approach is presented after which we define an improved formulation with an aim to greatly increase the computational performance. The justification for the application of an exponential time branch-and-cut algorithm used to solve the integer linear program comes from the NP-hardness of the problem. In the experiments we have conducted, we have presented a few important properties of our metrics and compared the performance of our problem formulation to the naive one.

The second software tool presented in this thesis is fortuna. Its main purpose is to classify and quantify the abundances of different alternative splicing events by analyzing the data collected by the next-generation RNA-sequencing techniques. Fortuna improves upon the existing tools by being able to efficiently identify even the events which have not been annotated. That feature makes it extremely useful in the fields where genetic mutations play crucial roles in our understanding of biological processes, such as cancer research. There are three main steps fortuna takes when processing a biological sample. In the first step, known genomic features

are enriched by a novel set of possible events which might transpire in the sample forming an extended annotation. This new annotation is used to build an alignment index. After the sample has been aligned to the index by kallisto, a post processing step begins in which the bulk of identification and classification is done. At the cost of some running time, the post processing step may gain in accuracy with the help of a genomic aligner used to re-compute possibly ambiguous alignments. The experiments show that fortuna is indeed superior in terms of precision and recall on samples simulated with errors. Its all-in-one approach and the possibility of the exploitation of multiple processor cores makes it faster and easier to use than the competition as seen in the experiments with the real world data. Apart from the speed advantage, fortuna also identifies the highest amount of novel events, thus corroborating our claims.

The fast and ever changing field of bioinformatics motivates us to continue our research by opening several avenues. Both the adoption and the development of new algorithms and hardware can, potentially, increase the efficiency of our software. An implementation of tighter clique constraints would increase the efficiency of the branch-and-cut presently implemented in Trajan. Also, a complete generalization of the arboreal matching to directed acyclic graphs and the formulation of efficient separation of the independent set constraints is a direction we would like to take. Considering fortuna, it is possible that machine learning techniques could be used on fortuna's output in order to draw conclusions on the state of a sample, e.g., to give a diagnosis on certain types of illness.

Bibliography

[1] B. Alberts, A. Johnson, J. Lewis, D. Morgan, M. Raff, K. Roberts, and P. Walter. *Molecular Biology of the Cell*. Garland Science, New York, USA, 2020.

[2] B. L. Allen and M. Steel. Subtree transfer operations and their induced metrics on evolutionary trees. *Annals Combinatorics*, 5:1–15, 2001.

[3] A. Alpert, L. S. Moore, T. Dubovik, and S. S. Shen-Orr. Alignment of single-cell trajectories to compare cellular expression dynamics. *Nature Methods*, 15(4):267–270, 2018.

[4] S. Anders, A. Reyes, and W. Huber. Detecting differential usage of exons from rna-seq data. *Genome Research*, 22(10):2008–2017, 2012.

[5] A. Apostolico. The myriad virtues of sub- word trees. *NATO AS1 Series F: Computer and System Sciences*, 12:85–96, 1985.

[6] R. Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972.

[7] S. Böcker, S. Canzar, and G. W. Klau. The generalized robinson-foulds metric. *Algorithms in Bioinformatics. WABI 2013. Lecture Notes in Computer Science*, 8126:156–169, 2013.

[8] D. Bertsimas and J. N. Tsitsiklis. *Introduction to Linear Optimization*. Athena Scientific, Nashua, New Hampshire, USA, 1997.

[9] M. G. B. Blum, O. Francois, and S. Janson. The mean, variance and limiting distribution of two statistics sensitive to phylogenetic tree balance. *Annals of Applied Probability*, 16(4):2195–2214, 2006.

[10] A. Boc, A. B. Diallo, and V. Makarenkov. T-rex: a web server for inferring, validating and visualizing phylogenetic trees and networks. *Nucleic Acids Research*, 40(1):573–579, 2012.

[11] D. Bogdanowicz and K. Giaro. On a matching distance between rooted phylogenetic trees. *International Journal of Applied Mathematics and Computer Science*, 23(3):1–16, 2013.

[12] D. Bogdanowicz and K. Giaro. Comparing phylogenetic trees by matching nodes using the transfer distance between partitions. *Journal of Computational Biology*, 24(5):422–435, 2017.

[13] L. Borozan, D. Matijevic, and S. Canzar. Properties of the generalized robinson-foulds metric. *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 330–335, 2019.

[14] N. L. Bray, H. Pimentel, P. Melsted, and L. Pachter. Near-optimal probabilistic rna-seq quantification. *Annals of Human Genetics*, 34:525–527, 2016.

[15] D. Brett, H. Pospisil, J. Valcárcel, J. Reich, and P. Bork. Mechanisms of alternative pre-messenger rna splicing. *Nature Genetics*, 30(1):29–30, 2002.

[16] R. D. L. Briandais. File searching using variable length keys. pages 295–298, 1959.

[17] D. Bryant and M. Steel. Computing the distribution of a tree metric. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 6(3):420–426, 2009.

[18] R. H. Byrd, P. Lu, J. Nocedal, and C. Zhu. A limited memory algorithm for bound constrained optimization. *SIAM Journal of Scientific Computing*, 16(5):1190–1208, 1995.

[19] S. Canzar, S. Andreotti, D. Weese, K. Reinert, and G. W. Klau. Cidane: Comprehensive isoform discovery and abundance estimation. *Genome Biology*, 17(1):1–18, 2016.

[20] R. M. Ceppelini. The estimation of gene frequencies in a random-mating population. *Annals of Human Genetics*, 20(2):97–115, 1955.

[21] Z. Chang, G. Li, and J. L. et al. Bridger: a new framework for de novo transcriptome assembly using rna-seq data. *Genome Biology*, 16(30), 2015.

[22] P. Compeau, P. A. Pevzner, and G. Tesler. How to apply de bruijn graphs to genome assembly. *Nature Biotechnology*, 29(11):987–991, 2011.

[23] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to ALgorithms*. MIT press, Cambridge, Massachusetts, USA, 1990.

[24] D. E. Critchlow, D. K. Pearl, C. Qian, and D. Faith. The triples distance for rooted bifurcating phylogenetic trees. *Systematic Biology*, 45(3):323–334, 1996.

[25] G. B. Dantzig. Linear programming. *Applied Mathematics*, 15:18–21, 1951.

[26] C. Darwin. *On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life*. London: John Murray, London, United Kingdom, 1859.

[27] W. H. E. Day. Optimal algorithms for comparing trees with labeled leaves. *Journal of Classification*, 2:7–28, 1985.

[28] M. O. Dayhoff and R. S. Ledley. Comprotein: a computer program to aid primary protein structure determination. *Proceedings of the December 4-6, 1962, Fall Joint Computer Conference. New York, NY: ACM*, pages 262–274, 1962.

[29] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[30] A. Dobin and T. R. Gingeras. Mapping rna-seq reads with star. *Current Protocols in Bioinformatics*, 51(1):11.14.1–11.14.9, 1985.

[31] Y. Du, Q. Huang, C. Arisdakessian, and L. X. Garmire. Evaluation of star and kallisto on single cell rna-seq data alignment. *G3: Genes, Genomes, Genetics*, 10(5):1775–1783, 2020.

[32] A. D. et al. Star: ultrafast universal rna-seq aligner. *Bioinformatics*, 29(1):15–21, 2013.

[33] C. T. et al. Transcript assembly and quantification by rna-seq reveals unannotated transcripts and isoform switching during cell differentiation. *Nature Biotechnology*, 28(5):511–515, 2010.

[34] C. T. et al. Differential gene and transcript expression analysis of rna-seq experiments with tophat and cufflinks. *Nature Protocols*, 7(3):562–578, 2012.

[35] E. W. et al. Alternative isoform regulation in human tissue transcriptomes. *Nature*, 456(7221):470–476, 2008.

[36] H. L. et al. The sequence alignment/map (sam) format and samtools. *Bioinformatics*, 25:2078–2079, 2009.

[37] J. D. et al. A gene ontology inferred from molecular networks. *Nature Biotechnology*, 31(1):38–45, 2012.

[38] K. J. et al. Predicting splicing from primary sequence with deep learning. *Cell*, 176(3):535–548, 2019.

[39] L. D. et al. Asgal: aligning rna-seq data to a splicing graph to detect novel alternative splicing events. *BMC Bioinformatics*, 19(1):444, 2018.

[40] M. E. R. et al. Limma powers differential expression analyses for rna-sequencing and microarray studies. *Nucleic Acids Research*, 43(7):47, 2105.

[41] M. M. et al. Genome sequencing in microfabricated high-density picolitre reactors. *Nature*, 437:376–380, 2005.

[42] O. K. et al. Function of alternative splicing. *Gene*, 514(1):1–30, 2013.

[43] S. S. et al. rmats: robust and flexible detection of differential alternative splicing from replicate rna-seq data. *Proceedings of the National Academy of Sciences of the United States of America*, 111(51):5593–5601, 2014.

[44] T. G. et al. Modelling and simulating generic rna-seq experiments with the flux simulator. *Nucleic Acids Research*, 40(20):10073–10083, 2012.

[45] V. H. D. et al. Dynamic pseudo-time warping of complex single-cell trajectories. *bioRxiv*, 2019.

[46] W. M. Fitch and E. Margoliash. Construction of phylogenetic trees. *Science*, 155(3760):279–284, 1967.

[47] S. Foissac and M. Sammeth. Astalavista: dynamic and flexible analysis of alternative splicing events in custom gene datasets. *Nucleic Acids Research*, 35:297–299, 2007.

[48] L. Frankiw, D. Baltimore, and G. Li. Alternative mrna splicing in cancer immunotherapy. *Nature Reviews Immunology*, 19:675–687, 2019.

[49] J. Gauthier, A. T. Vincent, S. J. Charette, and N. Derome. A brief history of bioinformatics. *Briefings in Bioinformatics*, 20(6):1981–1996, 2019.

[50] W. Gilbert. Why genes in pieces? *Nature*, 271(5645):501, 1978.

[51] C. S. Goh, A. A. Bogan, M. Joachimiak, D. Walther, and F. E. Cohen. Coevolution of proteins with their interaction partners. *Journal of Molecular Biology*, 299(2):283–293, 2000.

[52] M. Grabherr, B. Haas, and M. Y. et al. Full-length transcriptome assembly from rna-seq data without a reference genome. *Nature Biotechnology*, 29(7):644–652, 2011.

[53] M. K. Gunady, S. M. Mount, and C. B. Corrada. Yanagi: Fast and interpretable segment-based alternative splicing and gene expression analysis. *BMC Bioinformatics*, 20(1):421, 2019.

[54] K. Hainke, J. Rahnenführer, and R. Fried. Cumulative disease progression models for cross-sectional data: A review and comparison. *Biometrical Journal*, 54(5):617–640, 2012.

[55] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[56] A. D. Hershey and M. Chase. Independent functions of viral protein and nucleic acid in growth of bacteriophage. *Journal of General Physiology*, pages 39–56, 1952.

[57] M. R. Hestenes. Multiplier and gradient methods. *Journal of Optimization Theory and Applications*, 4(5):303–320, 1969.

[58] M. Hölzer and M. Marz. De novo transcriptome assembly: A comprehensive cross-species comparison of short-read rna-seq assemblers. *Gigascience*, 8(5), 2019.

[59] Z. Iqbal, M. Caccamo, I. Turner, P. Filcek, and G. McVean. De novo assembly and genotyping of variants using colored de bruijn graphs. *Nature Genetics*, 44(2):226–232, 2012.

[60] B. H. Jennings. Drosophila – a versatile model in biology & medicine. *Materials Today*, 14(5):190–195, 2011.

[61] L. Kachiyan. A polynomial algorithm in linear programming. *Soviet Mathematics Doklady*, 20:191–194, 1979.

[62] A. Kahles, C. S. Ong, Y. Zhong, and G. Ratsch. Spladder: identification, quantification and testing of alternative splicing events from rna-seq data. *Bioinformatics*, 32(12):1840–1847, 2016.

[63] M. Kao, T. Lam, W. Sung, and H. Ting. An even faster and more unifying algorithm for comparing trees via unbalanced bipartite matchings. *Journal of Algorithms*, 40(2):212 – 233, 2001.

[64] R. M. Karp. *Reducibility among Combinatorial Problems. In: Miller R.E., Thatcher J.W., Bohlinger J.D. (eds) Complexity of Computer Computations*. Springer, Boston, Massachusetts, USA, 1972.

[65] Y. Katz, E. T. Wang, E. M. Airoldi, and C. B. Burge. Analysis and design of rna sequencing experiments for identifying isoform regulation. *Nature Methods*, 7(12):1009–1015, 2010.

[66] D. Kim, J. M. Paggi, and C. P. et al. Graph-based genome alignment and genotyping with hisat2 and hisat-genotype. *Nature Biotechnology*, 37:907–915, 2019.

[67] D. Kim, G. Pertea, C. Trapnell, R. K. H. Pimentel, and S. L. Salzberg. Tophat2: accurate alignment of transcriptomes in the presence of insertions, deletions and gene fusions. *Genome Biology*, 14(4):R36, 2013.

[68] S. Laue, M. Mitterreiter, and J. Giesen. Geno–generic optimization for classical machine learning. pages 2190–2201, 2019.

[69] L. A. Lewis, P. O. Lewis, and K. Pryer. Unearthing the molecular phylodiversity of desert soil green algae (chlorophyta). *Systematic Biology*, 54(6):936–947, 2005.

[70] Y. Li and Z. Chenguang. A metric normalization of tree edit distance. *Frontiers of Computer Science in China*, 5:119–125, 2011.

[71] Y. Lin, V. Rajan, and B. M. E. Moret. A metric for phylogenetic trees based on matching. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 9(4):1014–1022, 2012.

[72] D. Maier. The complexity of some problems on subsequences and supersequences. *Journal of ACM*, 25(2):322–336, 1978.

[73] U. Manber and G. Meyers. Suffix arrays—a new method for online string searches. *SIAM J. Comput.*, 22:935–948, 1993.

[74] A. M. Maxam and W. Gilbert. A new method for sequencing dna. *Proceedings of the National Academy of Sciences of the United States of America*, 74:560–564, 1977.

[75] Y. Nesterov. Smooth minimization of non-smooth functions. *Mathematical Programming*, 103(1):127–152, 2005.

[76] Q. Pan, O. Shai, L. J. Lee, B. J. Frey, and B. J. Blencowe. Deep surveying of alternative splicing complexity in the human transcriptome by high-throughput sequencing. *Nature Genetics*, 40(12):1413–1415, 2008.

[77] C. H. Papadimitriou. The euclidean traveling salesman problem is np-complete. *Theoretical Computer Science*, 4:237–244, 1977.

[78] R. Patro, G. Duggal, and M. L. et al. Salmon provides fast and bias-aware quantification of transcript expression. *Nature Methods*, 14:417–419, 2017.

[79] R. D. Paucek, D. Baltimore, and G. Li. The cellular immunotherapy revolution: arming the immune system for precision therapy. *Trends in Immunology*, 40:292–309, 2019.

[80] Y. Peng, H. Leung, and S. Y. S. et al. Idba-tran: a more robust de novo de bruijn graph assembler for transcriptomes with uneven expression levels. *Bioinformatics*, 29:326–334, 2013.

[81] M. J. D. Powell. Algorithms for nonlinear constraints that use lagrangian functions. *Mathematical Programming*, 14(1):224–248, 1969.

[82] A. Pozo, F. Pazos, and A. Valencia. Defining functional distances over gene ontology. *BMC Bioinformatics*, 9(50), 2008.

[83] A. Roberts, C. Trapnell, J. Donaghey, J. L. Rinn, and L. Pachter. Improving rna-seq expression estimates by correcting for fragment bias. *Genome Biology*, 12(3):22, 2011.

[84] G. Robertson, J. Schein, and R. C. et al. De novo assembly and analysis of rna-seq data. *Nature Methods*, 7(11):909–912, 2010.

[85] D. F. Robinson and L. R. Foulds. Comparison of phylogenetic trees. *Mathematical Biosciences*, 53(1–2):131–147, 1981.

[86] D. Rossell, C. S. O. Attolini, M. Kroiss, and A. Stoecker. Quantifying alternative splicing from paired-end rna-sequencing data. *The Annals of Applied Statistics*, 8(1):309, 2014.

[87] F. Sanger, S. Nicklen, and A. R. Coulson. Dna sequencing with chain-terminating inhibitors. *Proceedings of the National Academy of Sciences of the United States of America*, 74:5463–5467, 1977.

[88] F. Sanger and E. O. P. Thompson. The amino-acid sequence in the glycyl chain of insulin. i. the identification of lower peptides from partial hydrolysates. *Journal of Biochemistry*, 53:353–366, 1953.

[89] F. Sanger and E. O. P. Thompson. The amino-acid sequence in the glycyl chain of insulin. ii. the investigation of peptides from enzymic hydrolysates. *Journal of Biochemistry*, 53:366–374, 1953.

[90] M. Schulz, D. Zerbino, and M. V. et al. Oases: robust de novo rna-seq assembly across the dynamic range of expression levels. *Bioinformatics*, 28:1086–1092, 2012.

[91] E. P. Solomon, L. R. Berg, and D. W. Martin. *Biology*. Thomson Brooks/Cole, Boston, USA, 2020.

[92] D. E. Soltis, M. A. Gitzendanner, and P. S. Soltis. A 567-taxon data set for angiosperms: The challenges posed by bayesian analyses of large data sets. *International Journal of Plant Sciences*, 168(2):137–157, 2007.

[93] C. Soneson, K. L. Matthes, M. Nowicka, C. W. Law, and M. D. Robinson. Isoform prefiltering improves performance of count-based methods for analysis of differential transcript usage. *Genome Biology*, 17(12), 2016.

[94] A. Srivastava, N. G. H. Sarkar, and R. Patro. Rapmap: a rapid, sensitive and accurate tool for mapping rna-seq reads to transcriptomes. *Bioinformatics*, 32(12):i192–i200, 2016.

[95] M. Steel and D. Penny. Origins of life: Common ancestry put to the test. *Nature*, 465(7295):168–169, 2010.

[96] T. Sterne-Weiler, R. J. Weatheritt, K. C. H. H. A. J. Best, and B. J. Blencowe. Efficient and accurate quantitative profiling of alternative splicing patterns of any complexity on a laptop. *Molecular Cell*, 72:187–200, 2018.

[97] K. Tai. The tree-to-tree correction problem. *Journal of the ACM*, 26(3):422–433, 1979.

[98] J. P. Venables, J. Tazi, and F. Juge. Regulated functional alternative splicing in drosophila. *Nucleic Acids Research*, 40(1):1–10, 2012.

[99] H. Vuong, T. Truong, T. Tran, and S. Pham. A revisit of rsem generative model and its em algorithm for quantifying transcript abundances. *bioRxiv*, page doi: http://dx.doi.org/10.1101/503672, 2019.

[100] R. Wagner and M. Fischer. The string-to-string correction problem. *Journal of ACM*, 21:168–178, 1974.

[101] J. D. Watson and F. H. C. Crick. Molecular structure of nucleic acids: a structure for deoxyribose nucleic acid. *Nature*, 171:737–738, 1953.

[102] J. D. Watson and F. H. C. Crick. The origin of the genetic code. *Journal of Molecular Biology*, 39:367–379, 1968.

[103] Y. Xie, G. Wu, and J. T. et al. Soapdenovo-trans: de novo transcriptome assembly with short rna-seq reads. *Bioinformatics*, 30:1660–1666, 2014.

[104] G. Yeo and C. B. Burge. Maximum entropy modeling of short sequence motifs with applications to rna splicing signals. *Journal of Computational Biology*, 11:377–394, 2004.

[105] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal of Scientific Computing*, 18(6):1245–1262, 1989.

[106] K. Zhang, R. Statman, and D. Shasha. On the editing distance between unordered labeled trees. *Information Processing Letters*, 42:133–139, 1992.

Curriculum vitae

Luka Borozan was born in Osijek, Croatia in 1991. After graduating from the high school in Osijek in 2010, he enrolled in the undergraduate Mathematics program at the Department of Mathematics, University of Osijek where he obtained his bachelor's degree in 2013. During the same year, he enrolled in a masters Mathematics and Computer Science program which he finished and obtained his master's degree in 2015. Since the late 2015, he is a PhD student at the Department of Mathematics of the Faculty of Science, University of Zagreb and a teaching assistant at the Department of Mathematics, University of Osijek.

He was a member of the project "Parameter estimation problem in some two-parameter monotonic mathematical models" financed by the University of Osijek from 25.9.2013. to 24.9.2014., lead by Darija Marković. Currently he is a member of the bilateral project "Metode optimizacije u biomedicini" under the leadership of Slobodan Jelić and Dušan Jakovetić.

Luka Borozan is a coauthor of the papers "On parameter estimation by nonlinear least squares in some special two-parameter exponential type models" which was presented at 15th international conference of operational research KOI 2014, "The stationarity of per capita electricity consumption in Croatia allowing for structural break(s)" published in the proceedings of the 13th international conference on operational research SOR 2015, "Integer Linear Programming methods for Group Steiner Tree and related problems" presented at the international conference ECCO XXXI - CO 2018, "Dynamic pseudo-time warping of complex single-cell trajectories" presented at the international conference RECOMB2019, and "Properties of the generalized Robinson-Foulds metric" presented at the international conference MIPRO2019 and published in its proceedings.