BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

OPTIMIZATION OF G-QUADRUPLEX IDENTIFICATION ALGORITHM

OPTIMALIZACE ALGORITMU PRO DETEKCI G-KVADRUPLEXŮ

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR DOMINIKA LABUDOVÁ
AUTOR PRÁCE

SUPERVISOR Ing. JIŘÍ HON
VEDOUCÍ PRÁCE

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2018/2019

Zadání bakalářské práce

Studentka: **Labudová Dominika**
Program: Informační technologie
Název: **Optimalizace algoritmu pro detekci G-kvadruplexů**
Optimization of G-Quadruplex Identification Algorithm
Kategorie: Bioinformatika

Zadání:

1. Seznamte se s problematikou detekce G-kvadruplexů v DNA sekvencích a zhodnoťte existující řešení.
2. Zaměřte se na nástroj *pqsfinder* a navrhněte vhodné optimalizace, případně rozšíření jeho funkcionality.
3. Navrhněte webové rozhraní pro nástroj *pqsfinder*.
4. Po dohodě s vedoucím práce implementujte navržené změny jako novou verzi nástroje *pqsfinder* a vytvořte navržené webové rozhraní.
5. Zhodnoťte přínos všech implementovaných změn a navrhněte možné pokračování projektu.

Literatura:

- Hon, J., Martínek, T., Zendulka, J., & Lexa, M. (2017). pqsfinder: an exhaustive and imperfection-tolerant search tool for potential quadruplex-forming sequences in R. Bioinformatics, 33(21), 3373-3379. https://doi.org/10.1093/bioinformatics/btx413

Pro udělení zápočtu za první semestr je požadováno:

- Splnění bodů 1 a 3 zadání.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Hon Jiří, Ing.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 15. května 2019
Datum schválení: 30. října 2018

Abstract

Identification of G-quadruplexes in DNA has been the interest of many studies in recent years. As a result, many identification tools were created. This bachelor's thesis focuses on analysis and optimization of a tool called pqsfinder along with creating a web interface for this tool. The optimization provided significant increase in the algorithm's speed and ability to process long DNA sequences. The web interface was designed and implemented using modern and effective technologies with emphasis on the user friendliness of the web application.

Abstrakt

Detekcia G-kvadruplexov v DNA bola v posledných rokoch cieľom viacerých štúdií. Dôsledkom bol vznik viacerých identifikačných nástrojov. Táto bakalárska práca sa zameriava na analýzu a optimalizáciu nástroja pqsfinder ako aj vytvorenie užívateľského rozhrania pre tento nástroj. Optimalizácia poskytla významný nárast v rýchlosti algoritmu a taktiež jeho schopnosti spracovávať dlhé DNA sekvencie. Užívateľské rozhranie bolo navrhnuté a implementované za použitia moderných a efektívnych technológií s dôrazom na užívateľskú prívetivosť webovej aplikácie.

Keywords

DNA, G-quadruplex, analysis, optimization, web application, GUI, React

Kľúčové slová

DNA, G-kvadruplex, analýza, optimalizácia, webová aplikácia, GUI, React

Reference

LABUDOVÁ, Dominika. *Optimization of G-Quadruplex Identification Algorithm.* Brno, 2019. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Jiří Hon

Rozšírený abstrakt

Rozumieť štruktúre a funkcii DNA je veľmi dôležitá časť porozumenia života na Zemi ako ho poznáme. DNA je nosič genetickej informácie a je zodpovedná za rast, regeneráciu a replikáciu buniek vo všetkých živých organizmoch. Všeobecne akceptovanou kanonickou štruktúrou DNA je síce dvojitá závitnica, no DNA sa dokáže formovať aj v iné, sekundárne štruktúry. Táto práca sa zameriava na štruktúru nazývanú G-kvadruplex (G4), ktorá sa tvorí v tých častiach DNA, ktoré sú bohaté na guaníny.

Výskum G4 v posledných rokoch výrazne napredoval, no ich funkcia a vznik v živých organizmoch stále nie sú plne pochopené. Je dokázané, že bunky si vyvinuli mechanizmus, ktorým sú schopné vytvorené G-kvadruplexy rozmotať, čo môže byť náznakom ich negatívneho vplyvu na bunku. Existujú však aj štúdie dokazujúce potenciálne využitie G4 v liečení nádorových chorôb. Na lepšie pochopenie vzniku a funkcie G4 je potrebné byť schopný identifikovať ich pozíciu v DNA sekvencii.

Identifikácia G4 bola v minulosti záujmom viacerých štúdií, ktorých výsledkom sú rôzne nástroje na detekciu G4 v DNA sekvenciách. Všetky tieto nástroje sa zameriavajú na identifikáciu intramolekulárnych G4, t.j. G4 tvorené z jedného vlákna DNA. Keďže táto téma stále nie je dôkladne preskúmaná, nové poznatky o G4 sú neustále objavované a podľa nich rástla aj kvalita jednotlivých nástrojov. Táto práca sa zameriava na najnovší identifikačný nástroj pqsfinder, ktorý vykazuje najvyššiu úspešnosť v detekcii a taktiež najväčší potenciál prispôsobiť sa novým poznatkom o G4.

DNA môže byť reprezentovaná ako postupnosť dusíkatých bází. G4 sa skladá z dvoch základných častí ktoré sa navzájom striedajú – G-run, ktorý sa vyskytne 4 krát a slučka, ktorá sa vyskytne nanajvýš 3 krát. Z toho vyplýva, že G4 v sekvencii je vlastne špecifická postupnosť znakov, ktorá môže byť identifikovaná regulárnym výrazom. Algoritmus pqsfinder je vysvetlený dopodrobna, keďže jeho pochopenie je potrebné pre pochopenie implementovanej optimalizácie. Tento algoritmus identifikuje G4 používaním upraveného regulárneho výrazu, ktorý je schopný identifikovať aj nedokonalosti v G-runoch. Spôsob, akým pqsfinder prehľadáva stavový priestor všetkých možností pre nájdenie potenciálneho G4 je veľmi vyčerpávajúci a časovo náročný. Táto práca poskytuje optimalizáciu uvedeného algoritmu. Keďže je pqsfinder dostupný iba ako R balíček, pre bežného užívateľa bez výraznejších technologických zručností nie je moc prívetivý. Preto sa táto práca ďalej zameriava aj na návrh a implementáciu webového rozhrania, ktoré bude slúžiť ako grafické rozhranie pre používanie nástroja pqsfinder.

Pred tým, ako môže byť algoritmus optimalizovaný, je potrebné uskutočniť jeho analýzu. Podrobným skúmaním algoritmu bolo zistené, že má problémy so spracovávaním sekvencií s veľkou hustotou guanínu. Problém nastáva pretože sa sa na každej pozícii v sekvencii vyskytuje veľké množstvo potenciálnych G4, čo spôsobuje priveľa generovaných možností, ktoré musia byť spracované pri rozlišovaní prekrývajúcich sa G4. Pri počte 40 guanínov za sebou program dokonca prestal pracovať. Toto správanie veľmi negatívne ovplyvňovalo použiteľnosť pqsfinderu. Za použitia profilovacích nástrojov bol potvrdený pôvod zasekávania algoritmu.

Práca navrhuje riešenie tohto problému pridaním dodatočných podmienok, ktoré musí G4 splňovať po každom nájdenom G-rune. Algoritmus najprv vypočíta potenciálne najvyššie skóre, aké by mohol G4 v aktuálnom stave dosiahnuť. Toto skóre je potom porovnané s dvomi hodnotami. Prvá hodnota predstavuje minimálne požadované skóre definované užívateľom. Druhá hodnota predstavuje najväčšie dosiahnuté skóre zo všetkých doposiaľ nájdených G4 prekrývajúcich prvú pozíciu prvého G-runu práve skúmaného G4. Ďalšia optimalizácia spočívala vo vytvorení 2 vektorov, ktoré sa na začiatku programu

naplnili výsledkami výpočtov, ktoré algoritmus počas behu využíva. Keďže boli výpočty urobené dopredu, algoritmus k nim ďalej už len pristupuje. Tento spôsob zaručuje, že každý výpočet bude prevedený iba jeden krát. Po implementácii navrhnutých zlepšení bola rýchlosť algoritmu zvýšená až 1500-krát na sekvencii s nie 100% hustotou guanínov. Zrýchlenie optimalizovaného algoritmu sa oproti pôvodnému zvyšuje s pribúdajúcou hustotou guanínov.

Webové rozhranie je navrhnuté a implementované pomocou klient-server architektúry. Celá aplikácia beží na cloude Stratus.FI s použitím Apache servera. Pre komunikáciu medzi webovou aplikáciou a balíčkom pqsfinder bola navrhnutá jednoduchá REST API. Komunikácia prebieha pomocou HTTP dotazov a je zabezpečená HTTPS a certifikátom Let's Encrypt™. Aplikácia je implementovaná v JavaScripte za použitia knižnice React.

Optimalizácia bola 2.5.2019 publikovaná ako pqsfinder verzia 2.0.0. Funkčné webové rozhranie je dostupné na adrese https://pqsfinder.fi.muni.cz/.

Optimization of G-Quadruplex Identification Algorithm

Declaration

I declare that I have prepared this Bachelors thesis independently, under the supervision of Ing. Jiří Hon. I listed all of the literary sources and publications that I have used.

....
Dominika Labudová
May 13, 2019

Acknowledgements

I would like to thank my supervisor Ing. Jiří Hon for his time and numerous advice. His insight was an invaluable part of creating this thesis. I would also like to thank Ing. Matej Lexa, Ph.D. for his help in deploying the web application.

Contents

1 Introduction **2**

2 DNA **3**
- 2.1 Structure of DNA 3
- 2.2 G-quadruplex 5

3 Existing tools for G-quadruplex identification **10**
- 3.1 QGRS Mapper 10
- 3.2 ImGQfinder 11
- 3.3 TetraplexFinder/QuadBase2 12
- 3.4 G4Hunter 12
- 3.5 pqsfinder 13

4 Optimization of pqsfinder algorithm **18**
- 4.1 Analysis of the current state 18
- 4.2 Suggested improvements 20
- 4.3 Implementation 22

5 Design of pqsfinder web interface **26**
- 5.1 Architecture 26
- 5.2 Server 26
- 5.3 Web application 27

6 Implementation of pqsfinder web interface **30**
- 6.1 Used technologies 30
- 6.2 Server 31
- 6.3 Web application 32

7 Conclusions **40**

Bibliography **41**

A Contents of Attached CD **44**

Chapter 1

Introduction

Understanding the structure and functions of DNA is an important part of understanding all life on Earth as we know it. DNA stores genetic information of a cell and is responsible for growth, replication and regeneration of cells in all living organisms. Although the *double-helix* is commonly accepted as a canonical structure, DNA can form other alternative secondary structures. The focus of this thesis will be on a structure called a *G-quadruplex* (G4), which forms in DNA sequences with high density of guanines and has been at the foreground of scientific research in the last years.

The functions and creation of G4s in living organisms are not yet fully understood. However, coming from the fact that cells have evolved a mechanism to unwind formed G4, their formation might be potentially damaging to the cell. There are also some positive aspects that were discovered regarding G4s. For instance, they are believed to have therapeutic potential in cancer treatment. To better understand their formation and function, it is important to have the ability to detect their exact positions in a DNA sequence.

G4 identification has been the interest of many studies. As a result, some identification tools were created. As more information about G4s was discovered, the tools grew in quality and identification reliability. This thesis provides an overview of the tools and their abilities, but further focuses only on the *pqsfinder* tool. This tool was chosen as the most advanced and with the biggest potential to adapt to new discoveries. The main deficiency of *pqsfinder* is in its speed. Another important shortcoming is that the user needs to have slightly advanced technological knowledge to use it, as it is available as an R package and does not have a graphical user interface.

The goal of this thesis is to identify the source of *pqsfinder's* lack of speed and suggest and implement an optimization. Other aspect that this thesis focuses on is a web application for *pqsfinder*. To implement the web application, it will be necessary to design an architecture of the whole interface.

The text of this thesis is structured into several logical units. Chapter 2 explains the structure and function of DNA and provides an overview of G-quadruplex formation and relevance in biology. Chapter 3 serves as an overview of existing tools used for G-quadruplex identification and it thoroughly explains the *pqsfinder* algorithm. Next, chapter 4 focuses on the whole optimization process. It provides an analysis of the algorithm as well as suggested improvements and their implementation. Chapter 5 defines a proposed web interface architecture and functionality along with the design of *pqsfinder* web application. Last, chapter 6 clarifies the application development process, used technologies and application structure.

Chapter 2

DNA

DNA (Deoxyribonucleic acid) is a molecule that serves as a carrier for genetic information of a cell. It is necessary for growth, duplication and regeneration of cells in all living organisms [7]. This chapter is an overview of the structure and function of DNA and its relevant secondary structures.

2.1 Structure of DNA

Even though the existence of DNA – called „nuclein“ at the time – was detected as early as the 1860s, its function as transporter of genetic information was not discovered until much later. The first evidence that DNA plays a role in inheritance of genetic information came in 1944, when a team of scientists published experiments that are now considered as definitive proof that DNA is the hereditary material [2]. These findings had a profound impact on discovery of the *double helix* (illustrated in Figure 2.3) structure of DNA by a number of scientists in the 1950s [20].

A DNA molecule consists of two polynucleotide chains, that can be called DNA strands or DNA chains. The strands form a right-handed double-helix – B-DNA. Each of these strands is made of four types of nucleotide sub-units as seen in Figure 2.1. Every nucleotide contains a sugar group, a phosphate group and a nitrogen base, that can be either *adenine (A), thymine (T), guanine (G) or cytosine (C)*[7].

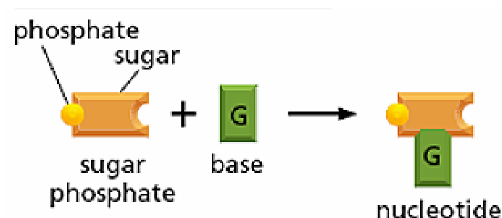


Figure 2.1: Structure of a nucleotide. Taken from article [6].

The strands are held together by *hydrogen bonding* between nitrogen bases as shown in Figure 2.2. Bonding of the bases is not random, but abides the *Watson-Crick* DNA base pairing model, where a purine always binds with one particular type of pyrimidine [20]. This means that the adenine of one strand always pairs with a thymine of the other strand and the cytosine always pairs with a guanine. This characteristic is referred to as *complementarity of nitrogen bases*. The A-T pairs are formed by two hydrogen bonds,

whereas the G-C pairs are formed by three. The genetic code of a cell is determined by the order of these bases. The complete human genome contains about 3 billion bases.[6]

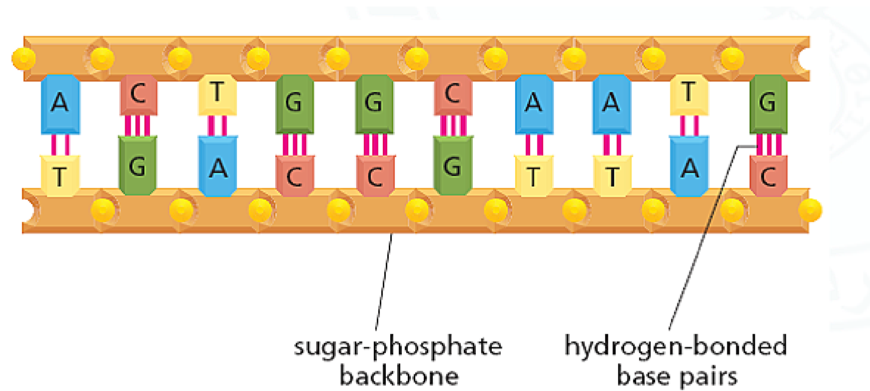


Figure 2.2: Hydrogen bonding between nitrogen bases. Taken from article [6].

Each polynucleotide strand has two different ends that can be distinguished based on whether they end with phosphate group or -OH (hydroxyl) group. These ends are respectively called the 5'end and the 3'end. DNA strands are *antiparallel*, meaning that the 3'end of one strand is paired with the 5'end of the other strand and are known as the '+' and '-' strands. The '+' strand is the one which runs from 5' to 3'. [6]

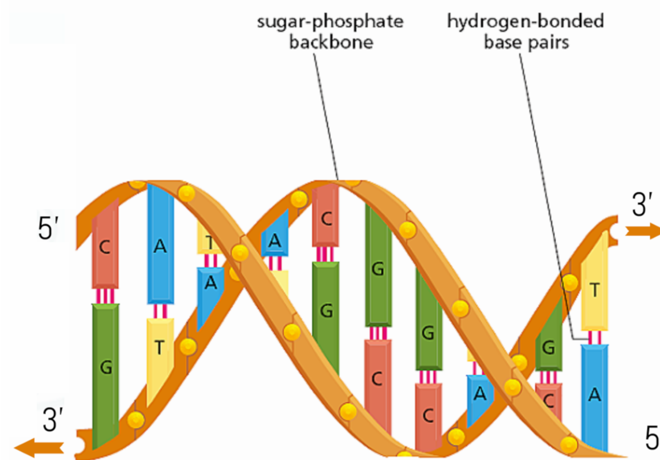


Figure 2.3: The double helix. Taken from article [6].

DNA replication

To start a DNA replication process, a part of the DNA has to be targeted by an *initiator protein*[1]. Once the initiator marked origin of the replication, it mobilizes other proteins that form a *pre-replication complex*[2]. Consequence of the forming of this protein complex is that the hydrogen bonds holding together the double-helix break and the helix opens.

[1] https://en.wikipedia.org/wiki/Initiator_protein

[2] https://en.wikipedia.org/wiki/Pre-replication_complex

The breaking of the bonds can be contributed to a *helicase*[3] enzyme. Each of the strands serves as a template for new complementary strand. Creation of the complementary strand is handled by enzyme *DNA polymerase*, which attaches nucleotides to the new strand to complement the bases on the template strand. Products of the replication process are two new identical DNA helixes.[6]

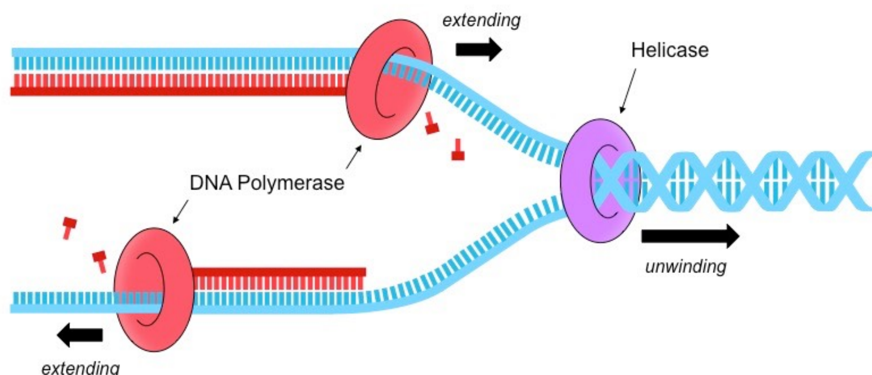


Figure 2.4: The replication of DNA. Taken from article [8].

Secondary structures

DNA is dynamic, meaning it is capable of forming alternative secondary structures, called non-B-DNA, such as triplexes, cruciforms, Z-DNA or G-quadruplexes. These structures are of growing interest in chemical and biological studies because of their potential ability to play regulatory roles in biological processes such as translation, transcription or replication of cells, some of which are already known [10]. Even before the discovery of the double-helix, it was known that guanine-rich DNA strands have the ability to self-associate. Structures that form by self-associating guanines are called G-quadruplexes. The next section covers the structure and functional roles of G-quadruplexes.

2.2 G-quadruplex

"If G-quadruplexes form so readily in vitro, Nature will have found a way of using them in vivo." – Aaron Klug, Nobel Prize Winner in Chemistry (1982). This sentence best expressed the significance of discovery of G4s, which where identified in 1962. These quadruple-helix DNA structures had only been demonstrated in the human genome in 2013 [4].

Structure

DNA sequences that are rich on guanine can lead to creation of *G-quartets*, or *G-tetrads*. G-quartet is a structure formed by four guanine bases linked cyclically through non-Watson-Crick base pairs called Hoogsten hydrogen bonds, depicted in Figure 2.5. *Hoogsten base pairing*[4] contains a purine and two pyrimidines and has different geometry between A-C

[3] https://en.wikipedia.org/wiki/Helicase
[4] https://en.wikipedia.org/wiki/Hoogsteen_base_pair

and G-C bonds. It applies a C6 amino group and the N7 position of the purine base, which bind the Watson-Crick (N3–C4) face of the pyrimidine base.

At least two or more of G-quartets stacked on top of each other form a non-canonical helical structure called *G-quadruplex* (G4) shown in Figure 2.5. G4 is additionally stabilized by the existence of a cation located in the centre between each pair of G-quartets.[16]

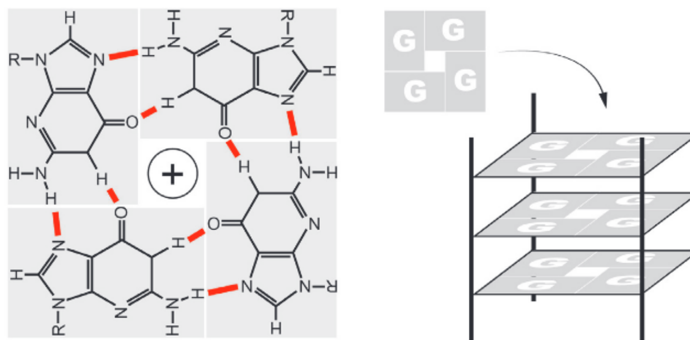


Figure 2.5: Left: four guanines looped into a stabilized guanine quartet by Hoogsten hydrogen bonds. Right: a formation of G-quadruplex by the stacking of G-quartets. Taken from article [16].

Classification

G4s can have numerous topologies and can be categorized based on the following criteria [5]:

1. Direction of strands or parts of a strand:
    - *parallel* – all of the strands run in the the same 5'–3' polarity, so the linking loops always run top-to-bottom,
    - *antiparallel* – two of the strands run in one direction and the other two in the other direction,
    - *hybrid* – three of the strands have the same orientation, the fourth has a different one.
2. Number of strands:
    - *intramolecular* – all G-quartets occur on the same strand. See Figure 2.6,
    - *intermolecular* – guanines forming G-quartets come from two or four different strands (DNA or RNA). See Figure 2.7.

The type of quadruplex that forms in *in vitro* experiments depends on many factors, such as strand length, sequence, type of stabilizing cation and composition of the dissolvent.

Functional roles

The mechanism of creation and purpose of G4s in living organisms is not yet fully identified. However, based on the fact that cells have evolved a technique for unwinding formed quadruplexes, their formation might be damaging to the cell. Nevertheless, due to their *in*

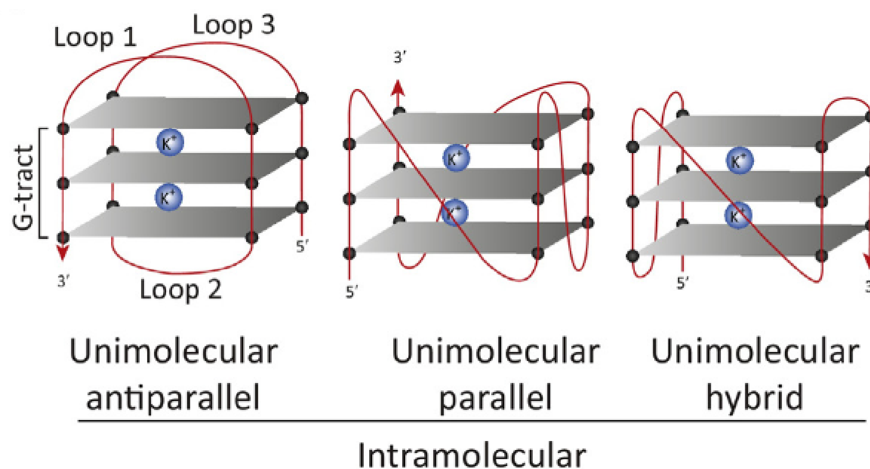


Figure 2.6: Intramolecular topologies. Taken from article [13].

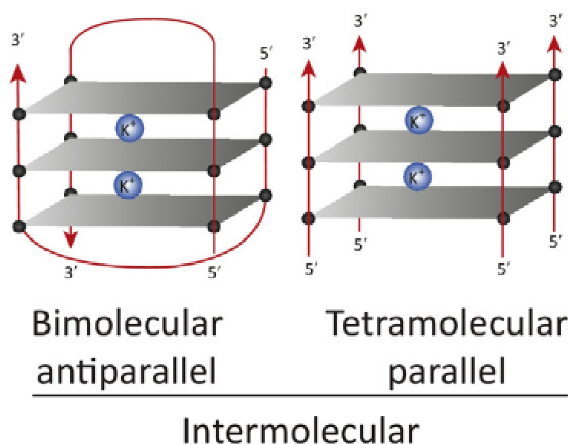


Figure 2.7: Intermolecular topologies. Taken from article [13].

vivo occurrence in promoter regions of some genes and at telomeric regions in humans, they are believed to have therapeutic potential in cancer treatment. G4s are predicted to play a role in regulation of gene expression. Quadruplexes formed in healthy cells can be unwound by helicase. This is not the case in cancerous cells with mutated helicase, where the G4s can not be unwound. This potentially leads to replication of damaged and cancerous cells. The quadruplex creation and dissolution also relates to activity of telomerase enzyme and the length of telomeres[5]. [15]

Possible defects

As confirmed by various experiments [19, 14], G4s are able to form with imperfections. So far, two types of imperfect G4s that appeared to be stable under physiological conditions have been identified: G4s with bulges and G4s with mismatches. Both are visualized in Figure 2.8.

[5] https://en.wikipedia.org/wiki/Telomere

A *bulge* is two or more projecting nucleotides that separate two quanines from neighbouring G-tetrads in one column.

A *mismatch* is defined as a nucleotide other than G that substitutes a guanine. Mismatches may participate in stacking.

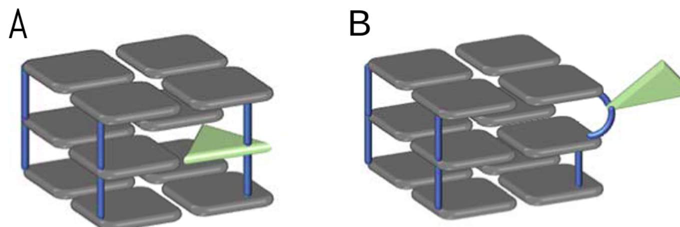


Figure 2.8: G4 defects: A) Mismatch B) Bulge. Taken from article [18].

Detection in DNA

DNA can be represented as a sequence of nitrogen bases, meaning that an intramolecular G4 itself can be represented as a string. Every G4 contains two types of elements: G-runs and loops. The representation is showed in Figure 2.9 along with marked G-runs and loops.

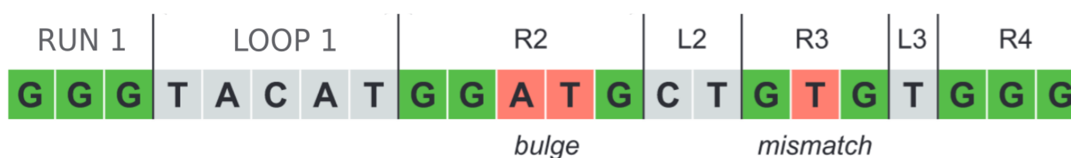


Figure 2.9: Representation of a G4 by nitrogen bases.

A *G-run* is a sequence of stacked nitrogen bases that bond with other G-runs to create the G-tetrads. These runs can contain bulges and mismatches, which were explained earlier. *Loops* are excessive nitrogen bases that do not bond but connect individual runs. Each G4 consists of four runs and at most three loops.

Figure 2.10 represents what the folded G4 from sequence in Figure 2.9 could look like. This is a unimolecular antiparallel G4, but its parallelity depends on many other things and hence the G4 is not guaranteed to take this particular form. The Gs that form one G-tetrad are connected by a red line.

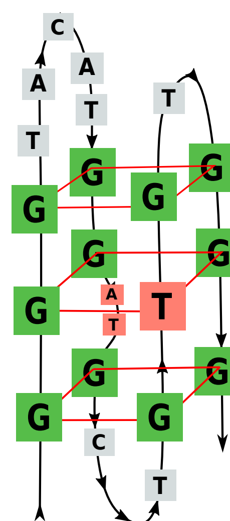


Figure 2.10: Sequence from Figure 2.9 folded into a G4.

Since intramolecular G4s can be represented as sequences, they are very easy to detect using regular expressions. The most

commonly used regular expression is G{3,6}.{1,8}G{3,6}.{1,8}G{3,6}.{1,8}G{3,6}, which depends on the G4 having at most six G-tetrads, the maximum loop length to be eight and the G-runs to be without imperfections.

Intramolecular G4s are currently also the most studied types of quadruplexes. This is mainly because of their potential involvement in the replication process that was explained earlier. Regarding the further application in this thesis, only intramolecular G4s will be further taken into consideration on account of their easy detection and the demand for their study.

Chapter 3

Existing tools for G-quadruplex identification

There are several existing tools for detecting potential quadruplex forming sequences (PQS). As the research on G-quadruplexes (G4s) grew, so did the quality of emerging tools. The main features that are important for a solid identification tool and that are going to be talked about in relation to tools discussed in this chapter are:

- *overlapping PQS detection* – a sequence can have multiple overlapping PQS that are mutually exclusive and it is highly useful to identify them and assign them a score based on their susceptibility to form a G4,
- *imperfection tolerance* – the existence of imperfect G4s has been evidenced in recent years. G-runs can include bulges or mismatches, meaning the standard regular expression G{3,6}.{1,8}G{3,6}.{1,8}G{3,6}.{1,8}G{3,6} does not capture all PQS and the search algorithm should be customized accordingly,
- *score assignment* – individual PQS having a score based on their stability can profoundly help in determining their potential to form stable G4s,
- *availability* – it is preferable that a user without advanced technological knowledge can easily use the tool and customize the searching algorithm, so its accessibility and additionally format of its output are quite important.

3.1 QGRS Mapper

QGRS Mapper [12] is one of the first PQS detection tools and as such does not tolerate any imperfections and is based on a simple folding rule defining four G-runs with rather short loops. It does however detect overlapping PQS and implements a scoring function as well as has a web interface, which is shown in Figure 3.1.

Customization of the search algorithm is limited to setting maximum length of G4, minimum size of G-run and a range for loop size. QGRS Mapper is the only tool that provides the user with an option to enter a regular expression representing a specific string that should be contained in one or more loops.

This tool has many ways in which the user can search for PQS. In addition to raw or FASTA format, it is possible to search and analyze a sequence by Gene name or symbol, Gene ID and Accession number or a Gene ID for NCBI Entrez Gene database[1].

QGRS Mapper provides several formats of data output. The user can pick a table view with information on each PQS found in the entered sequence either with overlaps or without them. One of the options to choose from is also a full sequence view with Another option is a graphics view with visual display of found PQS. The graphic module is user-friendly and interactive, but it needs a Java Plugin to run, which is not supported by most modern browsers[2].

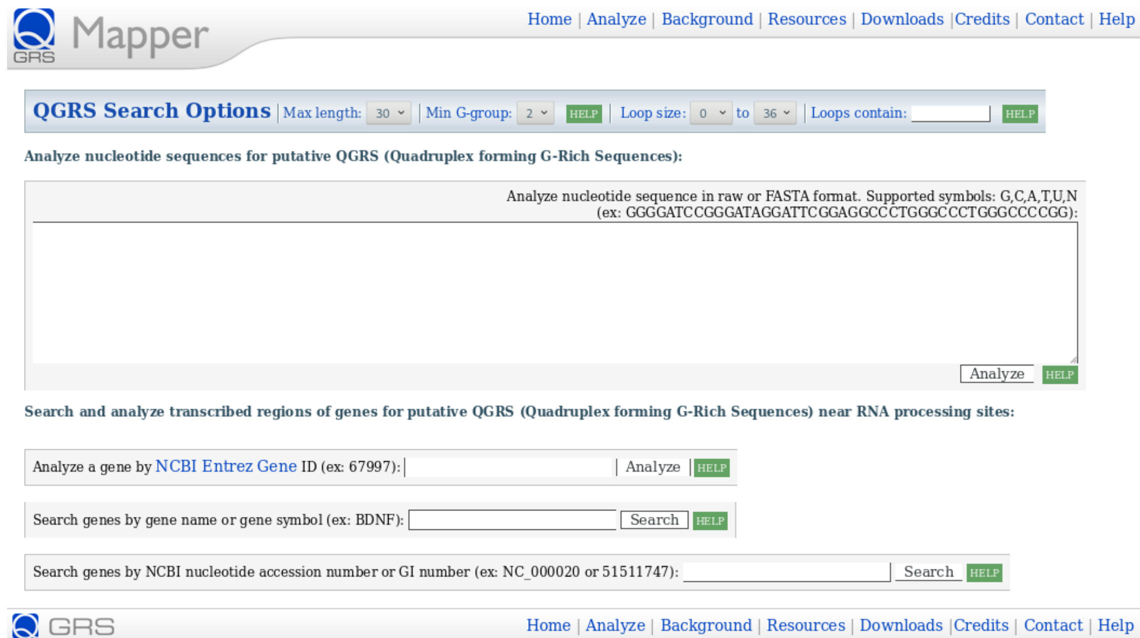


Figure 3.1: Main page of the QGRS Mapper web application.

3.2 ImGQfinder

ImGQfinder [18] is capable of detecting overlapping PQS but does not provide a scoring function, so it is not immediately clear which PQS has the highest potential to form a G4. Similarly to QGRS Mapper, ImGQfinder uses a folding rule to find PQS and has an easy to use web interface, albeit as shown in Figure 3.2, not very intricate.

This was the first tool that included imperfection tolerance in its algorithm, although rather limited, since it allows a maximum of one defect per G4.

Available input type is FASTA file and raw or FASTA format sequence. A user can customize the algorithm by selecting maximal loop length, number of G-tetrads and number of defects. One of the shortcomings is that the user can only select one option at a time, meaning the algorithm either searches for perfect G4s or G4s with exactly one defect. The same issue comes with the number of G-tetrads.

[1] https://www.ncbi.nlm.nih.gov/gene
[2] https://www.java.com/en/download/help/browser_activate_plugin.xml

ImGQfinder only provides basic output in the form of a simple table of found PQS with elementary information on each individual sequence.

Figure 3.2: Main page of the ImGQfinder web application.

3.3 TetraplexFinder/QuadBase2

QuadBase2 [9] is an updated version of QuadBase [21]. QuadBase2 is divided into three modules, one of which is TetraplexFinder, which is a module capable of PQS search in custom sequences. TetraplexFinder uses a regular expression to search for PQS in a sequence. It is able to detect overlaps and imperfections, but fails to provide a score for individual sequences. The web server can detect bulges of user provided length that is fixed between 0 and 7 and works only if the G-run length is set to three.

User can define custom G-run length, minimum and maximum loop size, bulge size and strands on which the search should be executed. The tool provides three pre-configured combinations of parameters and it is also possible to select 'greedy' or 'non-greedy' algorithm for search. Using the former variation of algorithm can maximize the loop length and the found sequences can contain multiple internal PQS, while the latter aims to shorten the loop length and the found sequences do not contain any internal PQS.

TetraplexFinder accepts a sequence in FASTA format as an input and displays results in two possible formats. The first format is a display of the whole sequence with marked found sequences differentiated by strand. The second format is a simple graph which shows position of found PQS in the sequence, also with strand differentiation.

3.4 G4Hunter

G4Hunter [3] allows imperfections as well as it provides a scoring function, but it merges neighbouring and overlapping PQS, resulting in the boundaries of individual PQS not being clear-cut. The detection of imperfect G4s is limited by not defining individual defect types.

G4Hunter detects PQS using a sliding window, so the resulting sequences are detected based on a scoring function that expresses PQS propensity. Each position in a sequence is given a score based on the type of nucleotide. A and T are given a score of 0. To account for G-richness, Gs are given score that is also based on their surroundings. One G has a score of 1, two continuous Gs a score of 2, three have a score of 3 and four or more consecutive Gs are awarded a 4. The same principle is applied to Cs, but with negative values from -1 to -4. The algorithm then takes a sliding window of 25 nucleotides and computes a mean score for each nucleid acid sequence of this length. Values where the absolute value of computed mean score rises above threshold are discarded. This is a very simple identification technique but was also proved to be one of the most precise out of so far discussed tools. The speed of G4Hunter is not lowered by its success rate in identifying PQS.

This tool is only available as an R-script and as such is not easily accessible to users without advanced technological knowledge.

3.5 pqsfinder

The last evaluated tool is pqsfinder [11], which was released later than all previously discussed tools as an R package[3]. It does not have a graphical user interface. pqsfinder allows imperfections as both bulges and mismatches, allows excessively long loops between individual G-runs and provides a scoring function based on the stability of PQS. Furthermore, it can detect all overlapping PQS and/or only the locally best and provide a density of all possible PQS on any position in a sequence. Additionally, it is highly customizable – a user can define their own scoring function, a regular expression for PQS search and many configuration options.

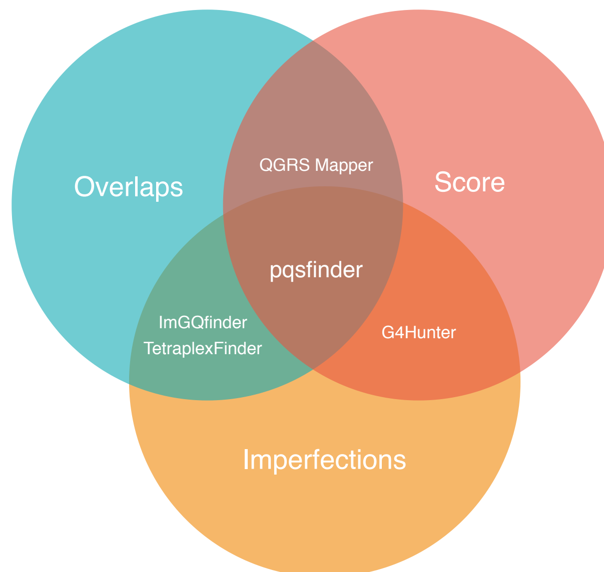


Figure 3.3: Summary of capabilities of the discussed tools.

As evidenced in Figure 3.3, pqsfinder is the most relevant search tool to this day. This, along with it having the biggest potential, is why I deemed it valuable to provide an opti-

[3] https://git.bioconductor.org/packages/pqsfinder

mization for this algorithms deficiency in speed and create a web based graphical interface so it can be used by a wider variety of users. For better understanding of the optimization explained in chapter 4, this algorithm is going to be explained in greater details.

Algorithm

The foundation of this algorithm stands on the fact that a G4 is formed by four consecutive G-runs which can be imperfect and are separated by loops of semi-arbitrary lengths. The whole algorithm can be split into three logical steps: identification of G-quartets, score assignment and overlap resolution.

Identification of G-quartets

To identify a G-run, pqsfinder uses a regular expression (regexp) *G{1,10}.{0,9}G{1,10}*, which along with requiring at least two guanines allows imperfect G-runs with both bulges and mismatches to be matched. The first G-run is found by applying this regexp to the sequence while limiting minimal and maximal length. This regexp is also used to match the remaining three G-runs, although with some added constraints. The first constraint is that each following G-run lies beyond the 3'-end of the previous one, which assures that the G-runs do not overlap. Next, the distance between two G-runs must fit in the range of minimal and maximal loop length while only one loop can have zero length. The last constraint is that with adding the new G-run, the PQS cannot exceed the user-defined maximal PQS length. A visual explanation of these constraints can be seen in Figure 3.4.

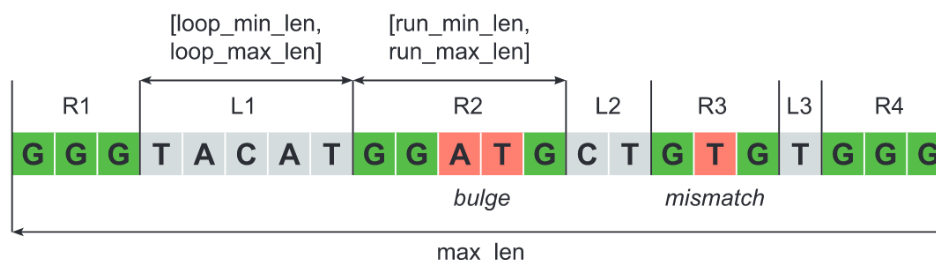


Figure 3.4: *PQS constraints.* Each PQS consists of G-runs (R1-4) and loops (L1-3), whose minimal and maximal lengths are constrained by overall PQS length as well as corresponding options shown in the figure. The algorithm takes all of these options as arguments and as such can be customized. Taken from article [11].

To identify all possible combinations, pqsfinder uses a backtracking approach. After matching and processing the whole PQS, the last G-run in the matched sequence is shortened by one nucleic acid base from the end. If the resulting G-run is still valid, the algorithm continues to scoring and overlap resolution. If the G-run is not valid, the algorithm backtracks to the preceding matched G-run and applies the same shortening process again. If this modified G-run is valid, the algorithm tries to match all the subsequent G-runs again. This continues until the algorithm reaches the first G-run. Once it finds its shortened state invalid, the whole process starts over from starting position of the first G-run shifted by one to the end. This backtracking procedure is beneficial in modeling the competition between overlapping PQS.

Score assignment

The scoring scheme of pqsfinder approximates the relationship between a G4 and its structural stability and can be split into two parts. Firstly, the quality of individual G-runs is assessed. A bonus score is awarded for each G-tetrad and a score penalty is given for every defect. The scoring is expressed by Equation 3.1,

$$S_r = (N_t - 1)B_t - N_m P_M - \sum_{i=1}^{N_b} P_b + F_b L_{bi}^{E_b} \tag{3.1}$$

where:

N_t = number of tetrads
B_t = G-tetrad stacking bonus
N_m = number of inner mismatches
P_m = mismatch penalization
N_b = number of bulges
P_b = bulge penalization
F_b = bulge length penalization factor
L_{bi} = length of the i-th bulge
E_b = bulge length exponent

The authors of pqsfinder decided to make two simplifying assumptions to neatly analyze defects. Firstly, at least one G-run in a PQS must be perfect (consisting of only guanines) and secondly, only one defect per G-run is allowed. Thanks to these premises, only lengths of G-runs and their G content are needed to detect bulges and mismatches.

In the second part of the scoring procedure, the destabilizing effect of loops is quantified by adding a penalty based on loop lengths. Complete scoring function is defined in Equation 3.2,

$$S = max(S_r - F_m L_m^{E_m}, 0) \tag{3.2}$$

where:

S_r = value from Equation 3.1
L_m = loop length mean
F_m, E_m = numerical parameters derived from experiments

Overlap resolution

The overlap resolution is designed as an iterative process that always prefers dominant PQS and is implemented to reduce memory usage as much as possible. First, the algorithm selects all PQS that share the highest obtained score. These selected PQS are then handled one at a time in the order of their rising starting position. If the currently processed PQS overlaps with the previous one, the current one is removed. If the current PQS is completely contained in the previous one, the previous is removed. Then, all lower-scoring PQS that overlap any of the selected PQS that are left are removed. All remaining PQS are reported and the also removed. The iteration starts again with selecting highest scoring PQS. These iterations continue until all PQS are processed.

Implementation

The pqsfinder package was implemented using languages R and C++. R is utilized to implement an interface for user interaction within the *Bioconductor*[4] framework using multiple R packages. Because of the main PQS search's computational intensity, C++ is used to implement the algorithm. To link the C++ code with R scripting language, the *Rcpp*[5] library was used.

Aside from the main information about individual PQS, *pqsfinder* provides two additional vectors – `density` vector and `maxScores` vector. The density vector presents the number of distinctive PQS overlapping each particular position in a sequence. The second vector gives a maximal achieved score of all overlapping PQS on each position.

As the general regular expression engine was found to be too inefficient for the needs of *pqsfinder*, an optimized matching function was implemented for the default G-run regular expression. If a user wants to use their own regular expression, the Boost regular expression library[6] is used.

Customization

pqsfinder was designed to be highly customizable. Users can adjust all relevant search and scoring parameters. Supported pqsfinder options are listed in Table 3.1 and can be divided into three logical groups:

- *Filter* options, that represent the main constraints used in PQS detection. They have great impact on sensitivity and speed of the algorithm as PQS that do not fit the constraints are immediately discarded and do not continue to the scoring part.
- *Scoring* options are all the constants from Equations 3.1 and 3.2. Default values are picked by experiments as the most reasonable values.
- *Advanced* options give the user full control over the algorithm. They provide a chance to set a custom G-run regular expression and a scoring function.

[4] `http://bioconductor.org/`
[5] `https://www.r-project.org/nosvn/pandoc/Rcpp.html`
[6] `https://github.com/boostorg/regex`

Group	Name	Description
Filters	*strand*	Strand symbol: +, − or * (both).
	overlapping	Enables overlapping PQS.
	max_len	Maximal PQS length.
	min_score	Minimal PQS score.
	run_min_len	Minimal G run length.
	run_max_len	Maximal G run length.
	loop_min_len	Minimal loop length.
	loop_max_len	Maximal loop length.
	max_bulges	Maximal number of bulges.
	max_mismatches	Maximal number of mismatches.
	max_defects	Maximal number of all defects.
Scoring	*tetrad_bonus*	G-tetrad stacking bonus B_t.
	mismatch_penalty	Inner mismatch penalization P_m.
	bulge_penalty	Bulge penalization P_b.
	bulge_len_factor	Bulge length penal. factor F_b.
	bulge_len_exponent	Bulge length penal. exponent E_b.
	loop_mean_factor	Loop mean penal. factor F_m.
	loop_mean_exponent	Loop mean penal. exponent E_m.
Advanced	*run_re*	G run regular expression.
	custom_scoring_fn	User-defined scoring function.
	use_default_scoring	Enables internal scoring system.
	verbose	Enables detailed text output.

Table 3.1: Overview of pqsfinder options

Chapter 4

Optimization of pqsfinder algorithm

As can be seen in Chapter 3, *pqsfinder* is currently the most advanced G-quadruplex (G4) identification tool. This chapter will provide an analysis of the algorithm based on its speed and efficiency, as well as suggested improvements and their implementation.

4.1 Analysis of the current state

Analysing an algorithm means to determine its computational complexity, that is the resources, storage and time it takes to execute it. The main deficiency of *pqsfinder* is in its speed, so this section focuses primarily on that. After assessing the algorithm's speed, the program will be profiled to find the most time consuming part of the algorithm. These findings should be sufficient to propose significant speed improvements.

Analysis

Some parts of the genome have very high G content density. These parts proved to be hard to process for *pqsfinder* due to its sensitivity to sequences with high G content. The sensitivity is demonstrated in Figure 4.1, where the exponential growth of execution time with increasing G content can be seen. The graph stops at 60% of G content in a sequence, because the execution time on sequences with higher density gets too high or even cause the program to crash. This proves that G rich sequences cause the program to significantly decrease its speed or even halt in case of full G sequences. This happens due to there being too many different potential G4s on each position in the searched sequence, which causes a large number of generated candidates that need to be processed in the overlap resolution as was explained on page 15.

When it comes to sequence length, a function representing the relation between sequence length and execution time is linear. This fact can be evidenced in Figure 4.2. Looking at Figures 4.1 and 4.2 it can be said that the main optimization focus should be on better PQS identification process in G rich parts of sequences.

After analysing the algorithm's time complexity, the next step is to find out what parts of the algorithm take up the most of execution time and figure out the cause and whether they can be optimized. This is performed by *dynamic program analysis*, or otherwise called *profiling*.

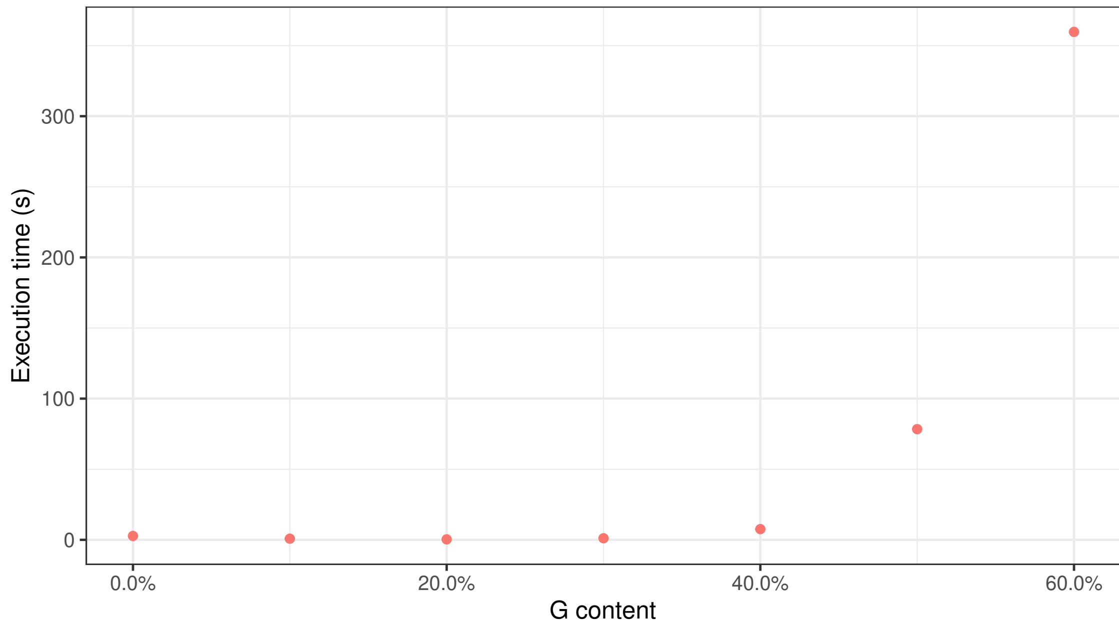


Figure 4.1: The relation between G content and execution time. Samples were taken on 5000-letter randomly generated sequences with gradually increased G content. For each sample, the algorithm was run 20 times and the execution times were averaged.

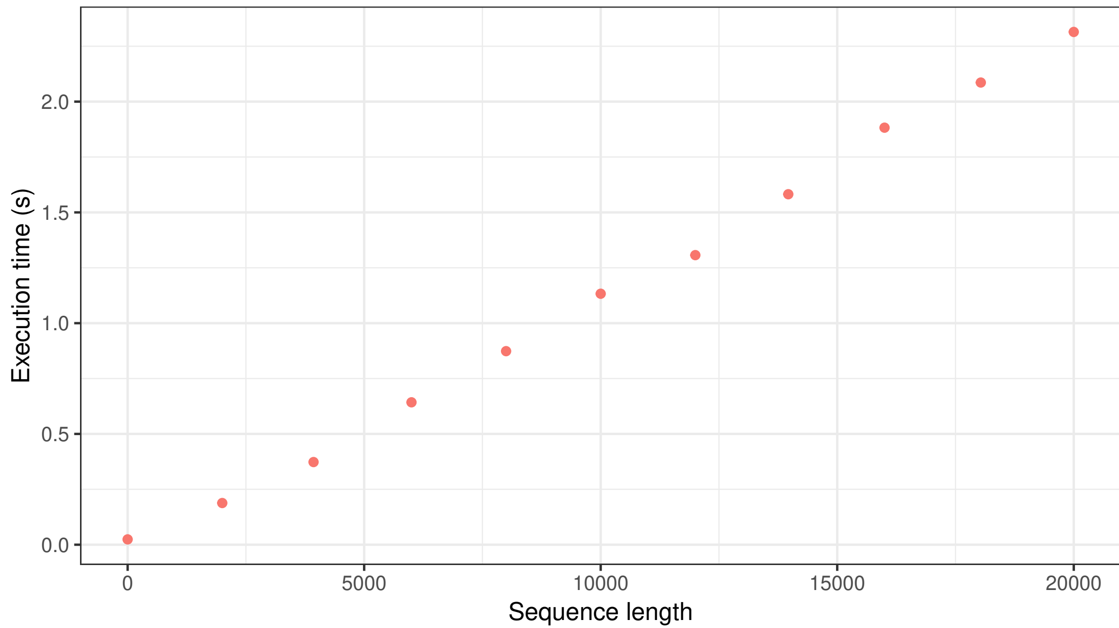


Figure 4.2: The relation between sequence length and execution time. Samples were taken on sequences with evenly distributed individual bases. For each sample, the algorithm was run 20 times and each time on a different randomly generated sequence. The execution times were averaged.

Profiling

The program was profiled using the *gperftools*[1] profiling tool. This tool uses sampling to obtain data. Sampling means that the profiler interrupts program execution at specific intervals and logs the state of program's call stack. As output, gperftools is able to draw a graph that represents how much time the program spent in individual procedures. This type of graph is called a *call graph*.

Each node in a call graph represents a procedure. The directed edges indicate caller to callee relations and the text node next to each one indicates the number of samples that were taken in the caller procedure on behalf of the callee. A node contains two important values, self weight and total weight. In a call graph generated by gperftools, the last two lines of a node contain these timing information. The third line represents the time it took to execute instructions directly contained in the procedure while the last line represents the time it took to execute instructions in called procedures along with the focused procedure.

Taking a look at Figure 4.3, it can be seen that 100% of the time is spent in procedure `find_all_runs`. This procedure is responsible for handling the whole PQS detection process. However, the self time of this procedure is only 8.2% with 73.1% of its total time being spent in procedure `score_pqs`. Over 30% of this time is spent in the procedure `score_run_defects`. Both of these procedures collectively implement the scoring function defined in Equations 3.1 and 3.2. The implementation of the scoring function in C++ uses function `pow` in its calculations, which, as can be seen in Figure 4.3, takes about 16% of the whole time it takes to score a PQS.

By assessing the generated call graph, it is certain that the parts of the algorithm that need to be evaluated for optimization are procedure `score_pqs` and the number of times procedure `find_all_runs` is recursively called.

4.2 Suggested improvements

The most significant optimization should be to *limit the number of recursive calls* to procedure `find_all_runs`.

As was previously explained in section 3.5, each run in the PQS is identified individually by applying regular expression with some additional constraints. However, between individual run searches, the only constraint the algorithm enforces is whether the loop between previous and current run is not too long. If the constraint does not fail, it always continues to search for the next run by recursively calling function `find_all_runs`. Because of no other constraints for continuing the search, the algorithm has to finish identification for every PQS. If the PQS has too low score, it is only found out after the identification of all four runs. The number of recursions could be reduced by applying conditions between individual run searches that check whether it is worth it to continue the search. By eliminating PQS before they make it to the scoring part and minimizing the available search scope, the execution time of the algorithm should rapidly decrease.

The main principle of suggested optimization is that after each run, the maximal potential score of a PQS is be computed. This value is then compared to the defined `min_score` and to maximal found score of all reported PQS so far overlapping the starting position of the first run in the current PQS. If the potential score is lower than `min_score` or the maximal found score, the search for other runs can be terminated. The maximal potential score

[1] https://gperftools.github.io/gperftools/cpuprofile.html

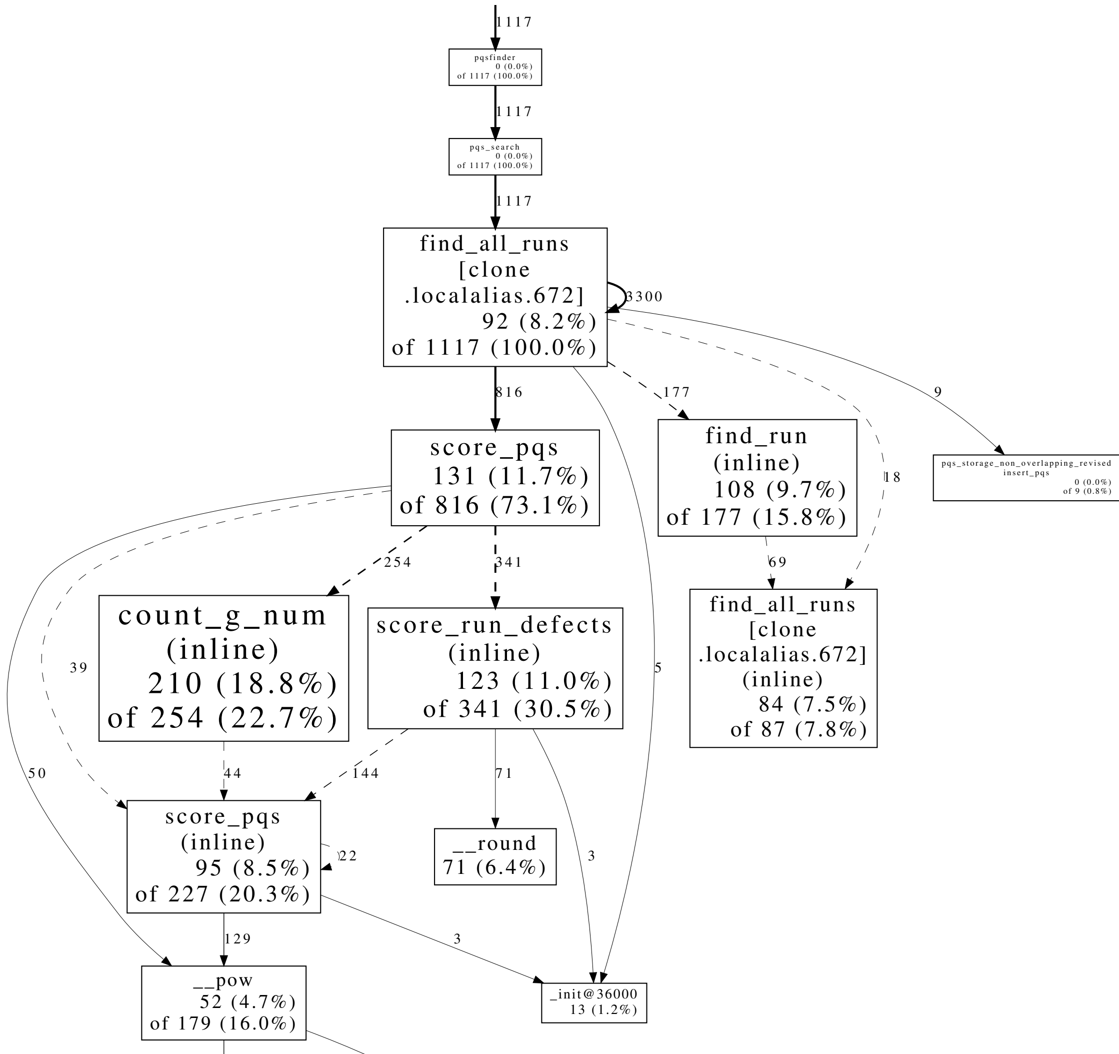


Figure 4.3: *gperftools* profiling run output. 1117 samples were taken during this run. The program was run on a 50000-letter sequence with default options. 155 PQS were identified on this sequence.

should be computed by the standard scoring function, while assuming that the remaining runs will not have any defects.

The second improvement that can be made is to optimize the time complexity of the scoring function. This can be done by removing the use of library function `pow`. This function is used while computing the values of loop and bulge penalties. The loop penalty is computed as sum of loop length divided by 3 to the power of `loop_mean_exponent`. Since the sum of loop lengths can only be in the range of 0 to `loop_max_len` times three, these values can be easily precomputed. This would mean that the values would only be computed once, not every time a score is calculated. The same principle can be used for precomputing values of bulge penalties. A bulge penalty is calculated as length of the bulge to the power of `bulge_len_exponent`. The bulge length can only be in the range of 0 to `run_max_len`.

4.3 Implementation

The main optimization occurred in file `pqsfinder.cpp`, in procedure `find_all_runs`. The principal of this change is to compare potential maximal score of PQS with defined minimal PQS score. To calculate the potential max score, these values are needed: tetrad count, loop lengths and defect count. Before starting the search, the tetrad count is set to MAX_INT and loop lengths sum and defect count to 0.

The next tetrad count is set as equal to run length in case of a mismatch and equal to G count in case of a perfect or bulged run. The tetrad count that is used in score calculations represents the minimal value from the next tetrad count and the minimum from previous tetrad counts. The loop lengths sum stays equal to zero in the first run. In the following runs, the loop length between the last found run and current run is added to the sum. The defect count is incremented by one if the length of the current run is not equal to its G count.

Using these values, the potential maximal score is calculated using the scoring function defined in Figure 3.2. The score is then compared to the defined minimal PQS score. If the score is lower than min score, the algorithm can continue to search for other run alternatives and will not continue to further recursion in this search path. The score is also compared to the maximal score reported so far overlapping the starting position of the first run in the current PQS. If both of these conditions succeed, the procedure continues either to scoring or to the recursive call of itself, in which case the computed values of tetrad count, loop lengths sum and defect count are passed as parameters.

Because this optimization is not compatible with searching for overlapping PQS, option called `deep` was added to *pqsfinder* parameters to turn the optimization on and off.

Another change in *pqsfinder* is precomputing the values of penalties. At the start of the program, `loop_max_len` and `run_max_len` are used to precompute the values of the bulge length penalty and the loop length penalty. These values are then stored in integer vectors. The algorithm then simply accesses the required value through the key representing either loop lengths sum or bulge size that it needs to compute a penalty for.

Optimization outcome

Looking at Figure 4.5, it can be clearly seen that the optimization provided a significant speed up of up to 1500 times on a sequence with only 60% G content. It is also evident that the speed up gets bigger with rising G content.

To demonstrate the speed increase, first 10 million nucleotides in chromosome 21 of the human genome were analyzed by the algorithm before and after optimization. The first code sample below shows the output of the `microbenchmark`[2] function run with *pqsfinder* before optimization. The second sample shows output of the same function run with the optimized algorithm.

```
Unit: seconds
                              expr      min       lq      mean    median
 pqsfinder(genome$chr21[1:1e+07]) 1541.676 1594.018  1644.125  1646.199
                                uq      max   neval
                          1702.385 1750.721      20
```

[2]https://cran.r-project.org/web/packages/microbenchmark/microbenchmark.pdf

```
Unit: seconds
                            expr      min       lq     mean   median
 pqsfinder(genome$chr21[1:1e+07]) 6.981571 7.505814 7.919199 7.795637
                              uq      max   neval
                        8.396269 8.935915     100
```

As can be seen, while the original algorithm took about 27 minutes to complete, the optimized version finished in 8 seconds.

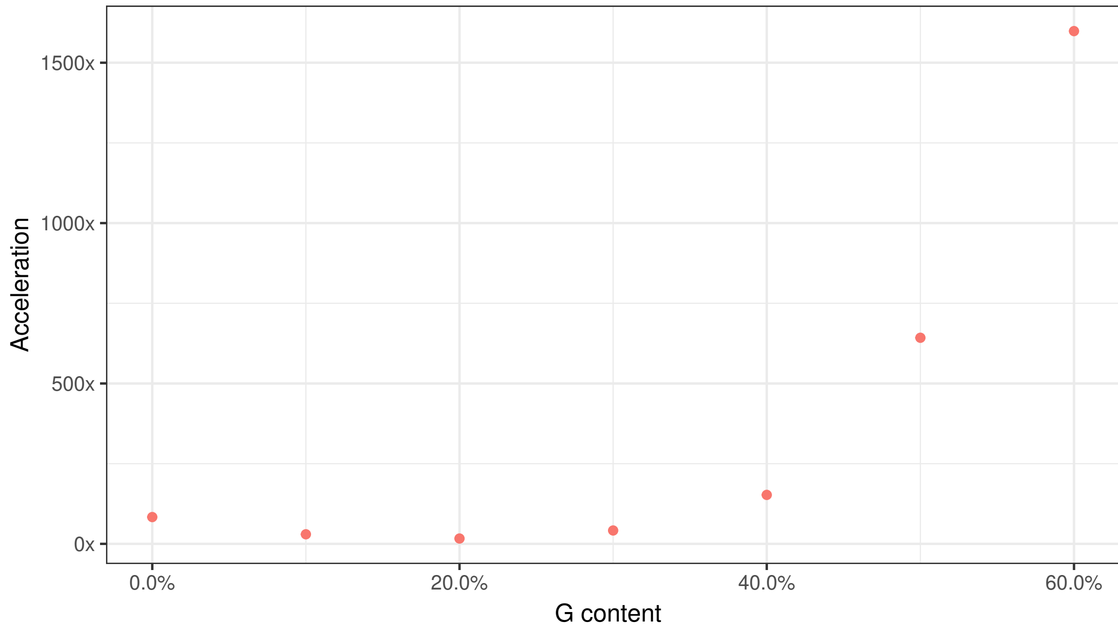


Figure 4.4: Visible acceleration of the algorithms speed in relation to G content. Samples were taken on 5000-letter randomly generated sequences with gradually increased G content. For each sample, the algorithm was run 50 times and the execution times were averaged.

Figure 4.5 shows how the optimization affected sequences with extremely high G content. The function representing relation of execution time to increasing of G content is no longer exponential. It reaches its high point a little after 75% and then rapidly decreases.

Figure 4.6 represents the *gperftools* profiling output after the optimization. The lack of procedure `score_pqs` and its callees indicates that the optimization reduced the number of times this function is called as well as decreased its execution time to be negligible. The number of recursive calls to `find_all_runs` is also notably smaller.

The provided optimization was released on 2nd May 2019 in *Bioconductor* version 3.9 as *pqsfinder* 2.0.0.[3]

[3] http://bioconductor.org/packages/release/bioc/html/pqsfinder.html

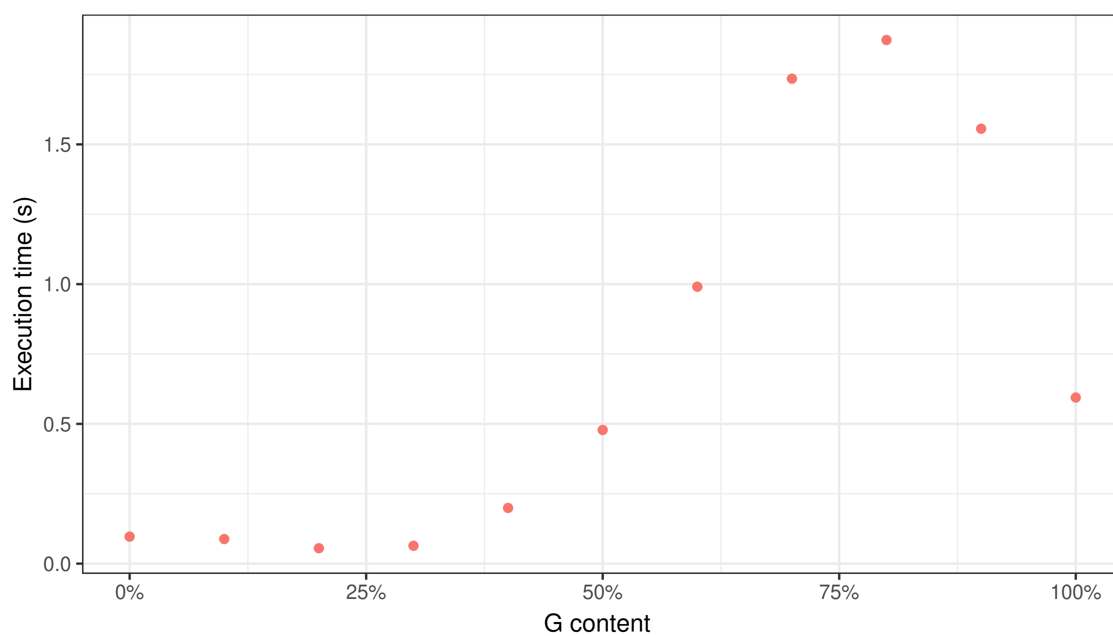


Figure 4.5: Execution time on sequences with rising G content after optimization. Samples were taken on 5000-letter randomly generated sequences with gradually increased G content. For each sample, the algorithm was run 100 times and the execution times were averaged.

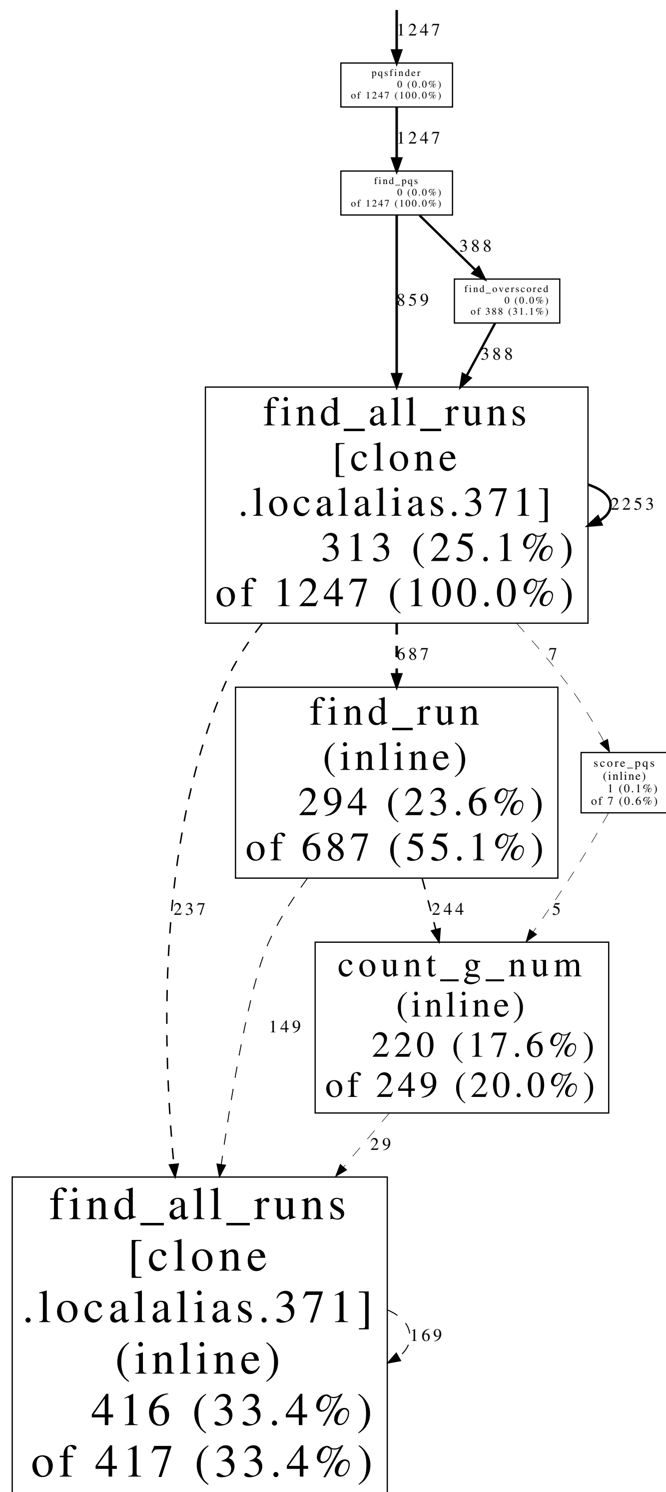


Figure 4.6: *gperftools* profiling run output after optimization. 1247 samples were taken during this run. The program was run on a 2800000-letter sequence with default options. 6506 PQS were identified on this sequence.

Chapter 5

Design of pqsfinder web interface

pqsfinder web application is designed for easy use of the *pqsfinder* package for users who do not have advanced technological knowledge. This chapter serves as an overview of its proposed architecture, functionality and design.

A processed user request to analyze a set of sequences will be referred to as a „job". A job can contain multiple search results if multiple sequences were inserted.

5.1 Architecture

pqsfinder application aims to be platform independent for its users. For that reason, a client-server architecture has been chosen. This allows for all the computations to be run on the server. The client only sends a request to the server, which carries out the request and responds with results. This project is designed so that it will not require a database, since it only needs to store simple files with search results. It will however require additional use of a technology that will allow the server to use the *pqsfinder* package.

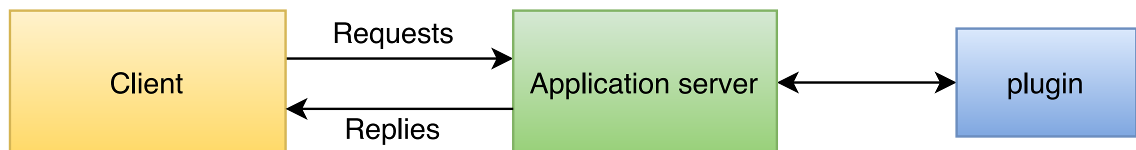


Figure 5.1: Client-server architecture with plugin.

5.2 Server

The server will be implemented as a simple REST API. All methods of the API need to be implemented to withstand incorrect input arguments, so that users can use them in their own programs, not just through this web application. It will need to be able to perform these actions for the client to work properly:

- Receive the input data and options, call the *pqsfinder* function and return the results.
- Return any previously computed results.
- Return simple data such as package version and default algorithm options.

Taking into consideration the previous requirements, the API will need to implement four main methods: return pqsfinder default options, return pqsfinder version, receive a job ID and return specific job, process sequence and return a job ID.

On account of the need to store computed results, it was decided that they will be stored as FASTA[1] formatted files into a single directory. The proposed process of storing the results is as follows:

1. For each new job request, a unique job ID is generated.
2. The results are formatted and saved into a file, with the file name consisting of the generated job ID.
3. Server returns the job ID to the client.
4. Files containing the results can be requested by client using the job ID.

5.3 Web application

The web application has two main functions. First, allow the user to insert a sequence and set search options. Second, display the results in a reasonable and easily understandable way. Some aspects of the design are inspired by existing tools described in chapter 3.

Before designing any other components, an application logo was designed. The logo can be seen in Figure 5.2. Apart from the application name, it contains an image of the most commonly known G4 structure. All presented design mockups were created using the Moqups[2] web application.

Figure 5.2: pqsfinder logo.

The common component for all pages is the *navigation bar*. This is the most important design element on a page, since it gives the user a sense of orientation and the apps capabilities. Users should not be confused with unexpected elements on common components, so this navigation bar is designed to be very simple. As can be seen in Figure 5.3, the navigation contains the application logo and name as is routine in all web applications. Next, all available pages are listed with focus on the currently opened one. Last, a search bar with placeholder *„Job id“* and a find button.

Figure 5.3: Design of navigation bar.

[1] https://en.wikipedia.org/wiki/FASTA_format
[2] https://moqups.com/

The application has three main functional components: a) Input of data, b) Table with information about search results, c) Graph representing found G-quadruplexes (G4s).

Analyze

The main page of the application is called *Analyze*. The label was chosen based on the fact that QGRS Mapper[3], one of the most known existing tools uses the same label in the same meaning, so most users are familiar with it. This page consist of two components – sequence input form and search options.

Figure 5.4: Layout of the sequence input form. 1 – text area for nucleotide sequence; 2 – buttons for input text manipulation; 3 – information about input restrictions.

User should be able to load data from his computer as well as to load some example sequence to quickly test how the application works. Figure 5.4 represents intended layout of the form for sequence input. Highlighted in blue are buttons for easy input manipulation. The paragraph highlighted in red should contain some brief information about *pqsfinder* along with listed restrictions for the input data.

Options are displayed in a separate component. Showing all possible search options could seem confusing for inexperienced users, so only four most commonly used were chosen to be shown implicitly – max length, min score, strand and max defects. The options component contains a button that reveals all the other possible options along with a button for setting the values to default ones.

Figure 5.5: Proposed design of the options component.

[3] http://bioinformatics.ramapo.edu/QGRS/analyze.php

Results

The most important part of pqsfinder is the page for presenting computed data. First thing the user should see is the job ID, so that he can access the results later. The user should also have a way to export either all results in a job into one file or each result individually. Possible export data formats are `gff`[4] and `csv`[5].

The table with results needs to contain all important G4 data – start, end, score, strand, number of tetrads, number of bulges, number of mismatches and the found sequence itself. The table needs to be sortable, with reasonable pagination.

While a table is the most used format to present this type of data, it was decided that pqsfinder should also contain a *graph* visually depicting individual positions of the potential quadruplex forming sequences (PQS) in the sequence.

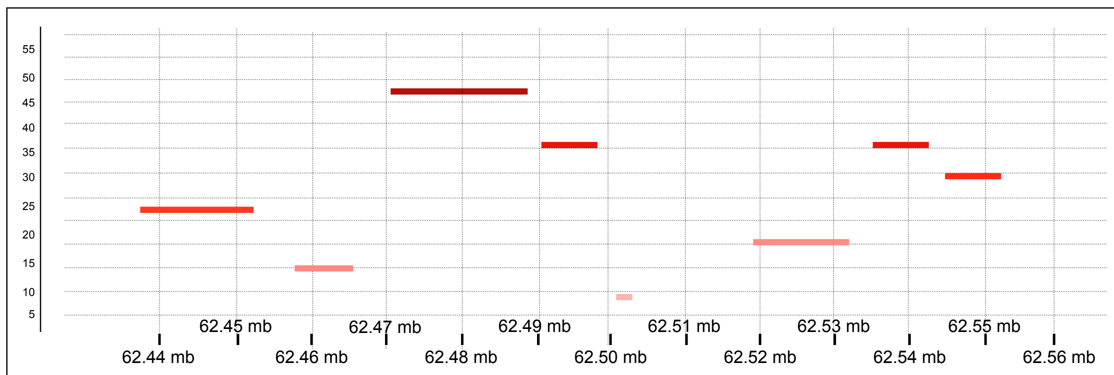


Figure 5.6: Mockup of the graph representing found PQS. Each rectangle represents individual PQS.

As can be seen in Figure 5.6, individual PQS color is dependant on the PQS score to better highlight the highest scoring PQS. The y-axis represents the score, while the x-axis the position in the square. The graph should also be scalable so that the user can have the option to focus on one part of the sequence. The user should have a chance to filter the PQS shown in graph based on the strand on which they were found.

[4]https://www.ensembl.org/info/website/upload/gff3.html
[5]https://en.wikipedia.org/wiki/Comma-separated_values

Chapter 6

Implementation of pqsfinder web interface

This chapter will provide a detailed overview of the application development process, used technologies and application structure. Greater focus is on the more challenging implementation parts. The complete and functioning web application can be found at https://pqsfinder.fi.muni.cz/.

6.1 Used technologies

Since the interface of the *pqsfinder* package is implemented in R, the REST API is also written in R language, using the *plumber*[17] library. This makes it simple to utilize all functions of *pqsfinder* and process information for the client. *React*[1] library is used to implement the web interface. A complex tool had to be chosen because an application as complicated as this one would be very difficult to implement in pure HTML and JavaScript. React was chosen for its declarative approach to programming user interfaces and its ability to rerender separate components without affecting the whole page.

plumber

Plumber is an R package for converting existing R code into REST API using only specialized comments. The comments can be prefixed by `#*` or `#'` to indicate they are specialized comments, although the former is recommended due to potential conflict with *roxygen*. An endpoint is generated by annotating a function with one of these HTTP methods: `@get`, `@post`, `@put`, `@delete`, `@head`. A single endpoint can support multiple methods. Plumber endpoints also support dynamic routes. The argument that will be dynamic is set between `< >` brackets and received as a function parameter.

React

React [1] is an open-source JavaScript library for creating user interfaces. It takes declarative views made out of encapsulated components. React detects the data change and updates and renders only those components where changes have been made. This makes the application more predictable and easier to debug.

[1] https://reactjs.org/

Each component in React has its own state. Passing values to other components can be only done through HTML properties. These passed values are immutable and can only be changed using a callback function.

6.2 Server

Apache[2] server was chosen for hosting the web application. The server is running on the Stratus.FI[3] cloud. The REST API is run by *plumber* commands on the server on localhost:8000. The client communicates with the server by HTTP requests through port 443. The communication with the server is secured by HTTPS[4] and a Let's Encrypt™[5] certificate. These requests are forwarded to the API which processes them and returns a result to the server which in turn sends a response to the client. The whole implemented architecture can be seen in Figure 6.1.

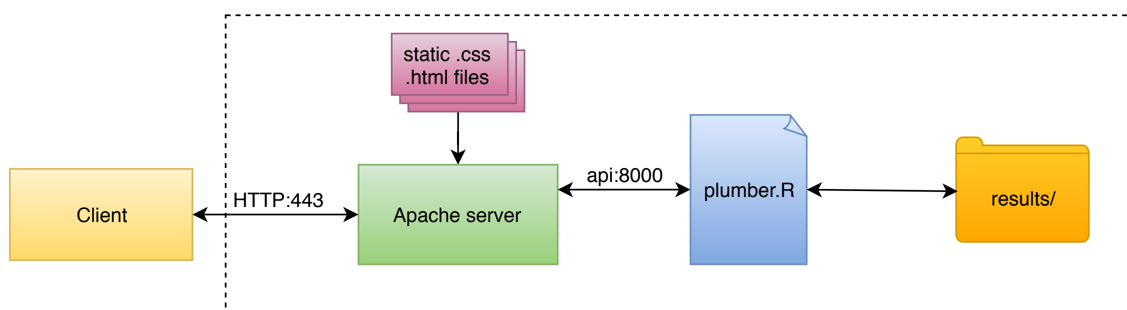


Figure 6.1: Architecture of pqsfinder web interface.

Implemented API endpoints and their functionality:

- `GET /formals` – returns the current default values of the *pqsfinder* function arguments using the `formals()` R function
- `GET /version` – returns currently used *pqsfinder* package version using the `packageVersion()` R function
- `GET /job/<id>` – returns file from the `results` folder identified by dynamic parameter `id`
- `POST /analyze` – accepts a list of sequences to be processed along with user defined algorithm options. Validates all received variables and generates a job ID. The function then iterates through all received sequences and calls the `pqsfinder()` function on each one using the received options. The results are then formatted and sequentially appended to one file. The file is stored in the `results` folder and it's name is set to the job ID. After all sequences are processed, the endpoint returns the generated job ID to the client.

[2] https://httpd.apache.org/
[3] https://www.fi.muni.cz/tech/unix/stratus.html.cs
[4] https://en.wikipedia.org/wiki/HTTPS
[5] https://letsencrypt.org/

6.3 Web application

Flux was chosen for the state management of the application. It enforces *unidirectional data flow*, shown in Figure 6.2, where the data flows in *only* one direction. Every time some data needs to be changed, the flow starts at the beginning.

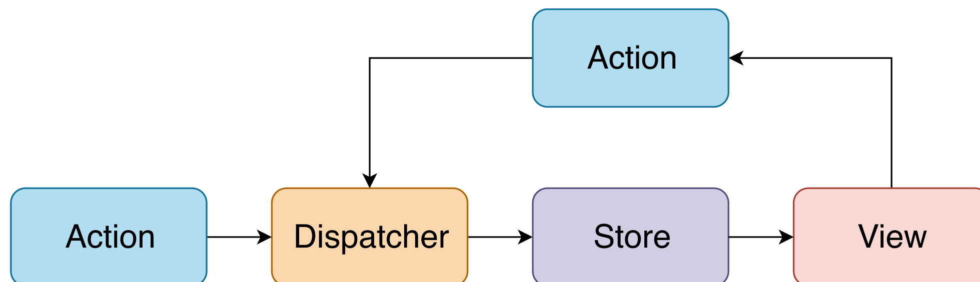


Figure 6.2: Basic Flux data flow. See http://facebook.github.io/flux/docs/overview.html

Actions are all the possible changes that can be made to the state of the application. Every time a data needs to be changed, an action is fired. There should be no other way in which to alter the state. An action consist of `type` and `payload`. *pqsfinder* uses three sets of actions that are respectively defined in individual files – `SubjectActions.js`, `ResultsActions.js` and `FindJobActions.js`.

Dispatcher receives the action and passes it to all of the registered stores. Stores receive all of the actions and decide on their own whether they will process them or not.

The stores handle the state of the entire application. There are many stores that each control their own part of the application state. A store handles received action based on its `type` and changes the state accordingly. Corresponding to actions, *pqsfinder* uses three different stores – `SubjectStore.js`, `ResultsStore.js` and `FindJobStore.js`. After handling the action, the store emits a change event that notifies the controller view.

The controller view asks the store for the new state, which it then passed to all the views under it. *The views* are simple components that output data to the user. When they receive user input, they just fire an action and the flow starts again.

Navigation

The page routing in this application is implemented using the `react-router-dom`[6] library. The router with available routes is defined in file `App.js`. All pages share a common instance of object `history` for easy location manipulation. Individual pages can be accessed either through the address bar or the application's navigation bar.

The navigation bar provides an option to enter a job ID of results to display. After clicking on the *Get job* button, contents of the input area are validated. If the validation succeeds, an action is fired that instructs the results store to fetch a job with entered ID. Component `NavMenu` is subscribed to change events from the results store and when it accepts a change event with information that the results were fetched, the application is rerouted to the `Results` component.

[6] https://www.npmjs.com/package/react-router-dom

pqsfinder Analyze Downloads Help Contact

job ID Get job

pqsfinder - imperfection tolerant G-quadruplex identification

Enter nucleotide sequence in FASTA format or choose a file which contains sequences in FASTA format. Maximal length of one sequence is 5000 nucleotides.

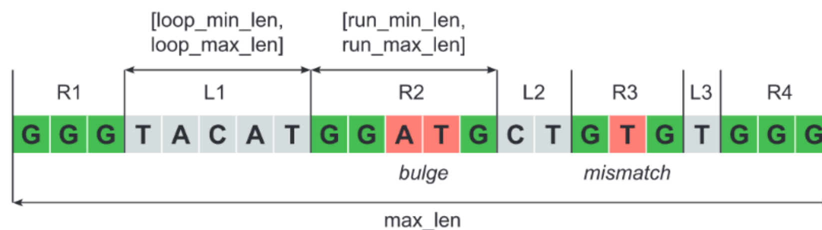
```
>HSGLTH1 Human theta 1-globin gene
CCACTGCACTCACCGCACCCGGCCAATTTTTGTGTTTTTAGTAGAGACTAAATACCA
TATAGTGAACACCTAAGACGGGGGGCCTTGGATCCAGGGCGATTCAGAGGGCCCCG
GTCGGAGCTGTCGGAGATTGAGCGCGCGCGGTCCCGGGATCTCCGACGAGGCCCT
GGACCCCCGGGCGGCGAAGCTGCGGCGCGGCGCCCCCTGGAGGCCGCGGGACCC
CTGGCCGGTCCGCGCAGGCGCAGCGGGGTCGCAGGGCGCGGCGGGTTCCAGCGC
GGGGATGGCGCTGTCCGCGGAGGACCGGGCGCTGGTGCGCGCCCTGTGGAAGAAG
CTGGGCAGCAACGTCGGCGTCTACACGACAGAGGCCCTGGAAAGGTGCGGCAGGC
TGGGCGCCCCCGCCCCCAGGGGCCCTCCCTCCCCAAGCCCCCCGGACGCGCCTCA
CCCACGTTCCTCTCGCAGGACCTTCCTGGCTTTCCCCGCCACGAAGACCTACTTCTC
```

Pqsfinder is able to detect G4s folded from imperfect G-runs containing bulges or mismatches or G4s having long loops. Pqsfinder assigns an integer score to each hit that was fitted on G4 sequencing data and corresponds to expected stability of the folded G4.

Clear input Example data Browse... No file selected.

Options

Option	Value
Max length	50
Min score	52
Strand	+ -
Max defects	3
Min loop length	0
Max loop length	30
Max bulges	3
Max mismatches	3
Min run length	2
Max run length	11



Default values Advanced options

Analyze

Cite: Hon J, Martinek T, Zendulka J, Lexa M (2017). pqsfinder: an exhaustive and imperfection-tolerant search tool for potential quadruplex-forming sequences in R., Bioinformatics, 33(21), 3373-3379. DOI: 10.1093/bioinformatics/btx413.

Version: 2.1.0

Figure 6.3: Main page of the web application.

Analyze

The main page *Analyze* is implemented in the file `Analyze.js` and its subcomponents. The final appearance of the page can be seen in Figure 6.3. Every change in the input text area or to one of the options is reported through actions to the subject store with the new value. After submitting the input to be analyzed, an action is fired that instructs the subject store to analyze the data that it has stored at that moment. While the data is processed and the results are send back from the server, the user sees a loader indicating that something is happening, so that he does not think that the application has crashed. First thing the algorithm does is to check the `strand` option to make sure that it contains at least one value. If none value is specified, it defaults the option to the '+' strand.

The algorithm then moves on to parsing the text input. It starts o loop to parse individual sequences that ends only after there is none data left to be processed. First, the sequence name is identified. Due to FASTA format of the input, it is required that the sequence name starts with character '>' and ends with '\n'. Data from the position of '\n' plus one, until the next '\n' or '>', is recognized as the sequence itself. All found sequences are gradually pushed into an array of objects, where each object represents the sequence and contains two name-value pairs: sequence description and the sequence. Individual sequences are then validated for allowed characters and maximal sequence length.

After all data is parsed and validated, a POST request to the server is send with options and sequences as the request body. After receiving an answer with the generated job ID, a change event is emitted to alert the `Analyze` component to reroute the application to `/results/<id>` which mounts the `Results` component.

Results

After the `Results` component is mounted, it instructs the results store to request a job based on the id in its location pathname. The final appearance of this page can be seen in Figure 6.4. The received file with results is then processed and saved to a data structure.

The file is processed in a loop that ends when the last sequence with found PQS is identified. Since the information about the PQS is in a FASTA format, they are easily parsed. The PQS description is identified by starting with character '>' and ending with '\n'. The information is then parsed and saved into a structure. The line after the description represents the PQS itself. All collected data is then saved into a structure containing key-value pairs, where the key is the sequence description and the value is an object consisting of the PQS and its features. After parsing the whole file, a change event is emitted to alert the `Results` component that the results are ready to be displayed.

The component gets the results from the store and first renders a general header containing the job ID and a button that the user can use to export results from all sequences. Next, it renders an individual header, table and graph for each set of results. The header contains information about the sequence length and a number of found quadruplexes. Additionally, the user has an option to export results only from that individual sequence. When the user clicks on the *export* button, an action is fired that instructs the results store to prepare a file for export. The store takes the data it has currently stored and formats them according to the user selected type. Then, using the `file-saver`[7] library, it offers the generated file for download to the client.

[7] https://www.npmjs.com/package/file-saver

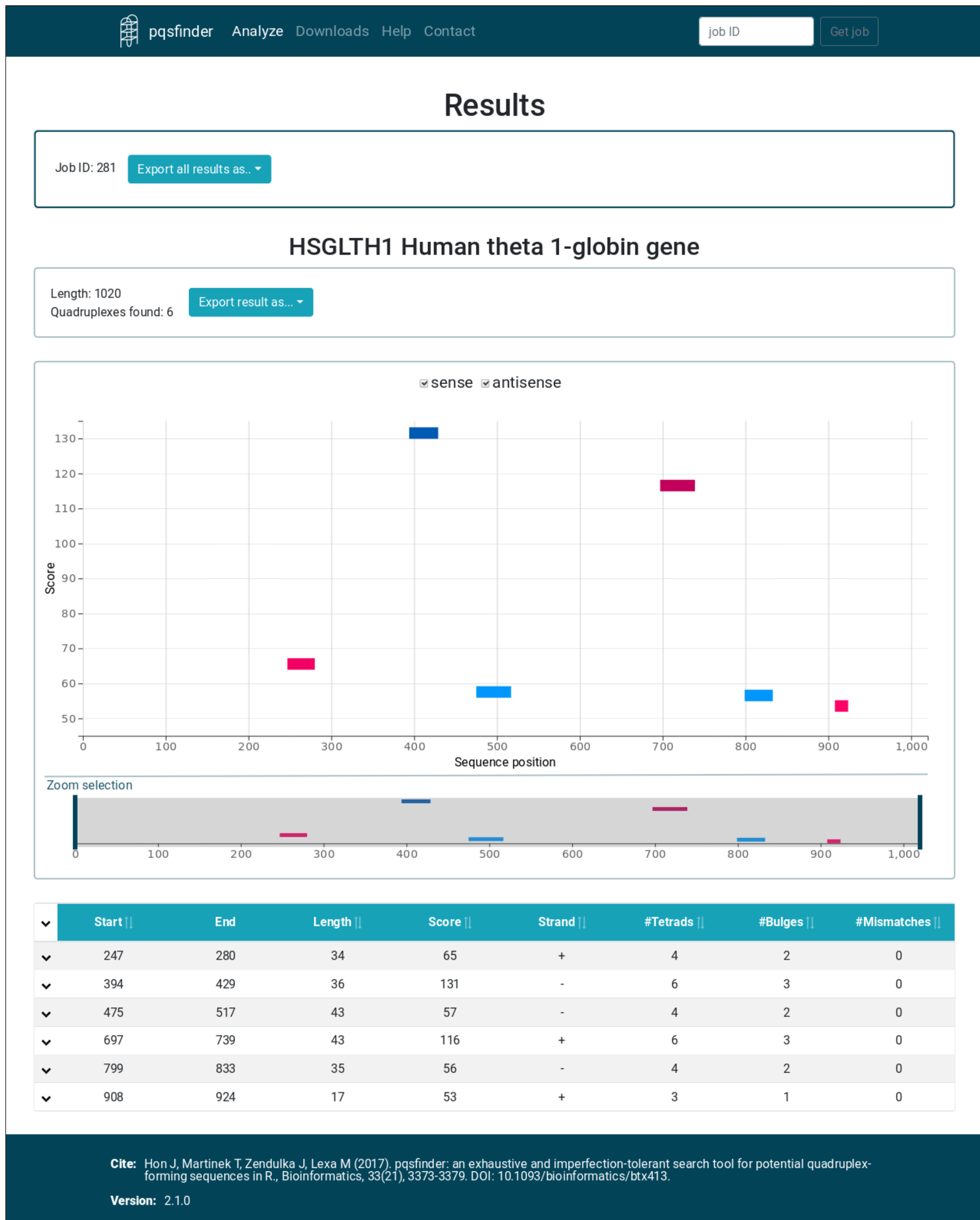


Figure 6.4: The page displaying results of the G4 identification.

The `Results.js` file also handles two important supplementary functionalities. First, it handles filtering values for the `Graph` component based on the type of strand. Second, it implements a function that colors the quadruplexes as can be seen in Figure 6.5. Light grey color is used for loops, green for G-runs, orange for mismatches and red for bulges. The positions of the G-runs are obtained from the server as additional information about individual G4s. This function is quite intricate when it comes to coloring the defects. The

function receives one run to process at a time. First, the G count of the run has to be found. If the run is on the antisense strand, G count will be equal to the count of Cs. If the G count along with the G-run length are both equal to number the of tetrads in the quadruplex, it is a perfect run and is whole colored with green. If the number of tetrads is greater than the G count by one, it is a mismatch. To determine which letter in the run should be colored, the position of a non-G character is found by a regular expression and that position is colored with orange, while all the other letters are green.

GGCTGGGAAGGGATGGTACGGGGGACAGGGTG

Figure 6.5: Colored quadruplex. Green represents G-runs, red bulges, orange mismatches and grey the loops.

Identifying the bulges is a little more complicated. The main variable needed for finding a whole bulge is the defect count, which is equal to the run length minus the number of tetrads. The defect is identified by finding the positions of the first and last non-G characters in the run. Everything between these positions is guaranteed to be a defect. This approach is however not adequate in two special cases. The G run can be made of just Gs and hence the bulge will also consist of Gs or there can be one or more Gs on either side of the defect that were not identified as part of the bulge. These cases were handled by special conditions.

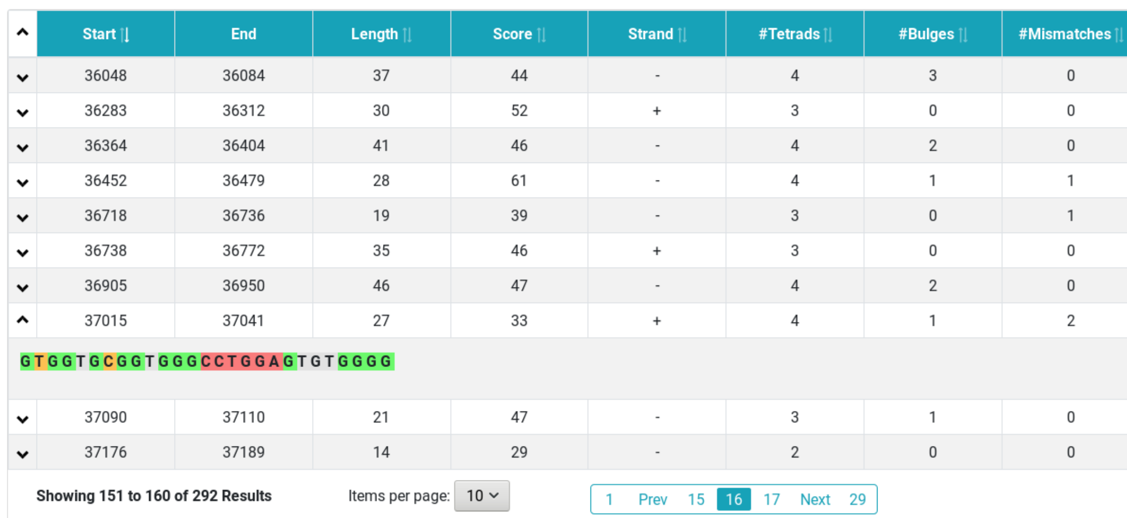
If the run consist of just Gs, the middle position of the run is computed to be half of the run length rounded down. This position is certainly part of a defect. If the defect count is larger than one, the defect part of the sequence is extended evenly to both sides by half of the defect count. Since the run length half is rounded down and the defect count half is rounded up, in case of even defect count, it favours adding more Gs to the right of the middle position.

If the identified defect contains characters other than G and the length of the identified defect does not equal the defect count, the defect contains Gs on one or both of its edges that were not identified. By subtracting the length of the identified defect from the defect count, the number of undetected Gs is found. As much Gs as possible and needed are added to the end of the defect. Then, if there are still some Gs left that to need to be attached, they are added to the start of the defect. The final identified defect represents the whole bulge and is colored red. Everything else is colored green.

Table

The *table* is created using external library `react-bootstrap-table-next`[8] and is implemented in the `ResultsTable.js` file. Detail of the table with active pagination can be seen in Figure 6.6. The library is sufficient in fast and easy creation of a table but its sorting and pagination functions were not satisfactory for the required `Table` component, so they had to be implemented separately. Changing the data to be displayed is handled by the `Table` component, however the pagination that can be seen at the bottom of Figure 6.6 is implemented separately in the `Pagination.js` file. The user can choose how many quadruplexes should be displayed at once. The available values range from 10 to 50, by a step of 10. As can be seen in Figure 6.6, the table displays all available information about

[8] https://www.npmjs.com/package/react-bootstrap-table-next



Start	End	Length	Score	Strand	#Tetrads	#Bulges	#Mismatches
36048	36084	37	44	-	4	3	0
36283	36312	30	52	+	3	0	0
36364	36404	41	46	-	4	2	0
36452	36479	28	61	-	4	1	1
36718	36736	19	39	-	3	0	1
36738	36772	35	46	+	3	0	0
36905	36950	46	47	-	4	2	0
37015	37041	27	33	+	4	1	2
GTGGTGCGGTGGGCCTGGAGTGTGGGG							
37090	37110	21	47	-	3	1	0
37176	37189	14	29	-	2	0	0

Figure 6.6: Detail of the table with active pagination.

individual G4s. All of these columns are sortable, with the exception of the second column `End`, because its values directly correspond to the order of the first `Start` column. Each row can be expanded, displaying the colored G4 itself. This feature is handled by the external library.

Graph

The `Graph` component is the most complex component in this application. The whole graph and all its parts are rendered as `svg`[9] elements. The majority of its functionality is implemented using the `D3.js`[10] library. The graph consists of two main parts, the focus window and its context.

The *y-axis* of the graph represents a G4 score. Its lower-bound is selected by finding the lowest score among found G4s and decreasing it by 5 for better visualization. The upper-bound is found using the same principle, only the highest score is increased by 5. The *x-axis* represents the nucleotide position in a sequence. Values of this axis vary depending on which part of the sequence is in focus. Although the d3 library implements a function to generate grid lines, its behaviour did not meet the visual requirements, so they had to be generated separately. The algorithm first selects all elements with specified class name that identifies the ticks on given axis. For each selected tick, a `line` element is appended to the focus element that starts at x[0,0] and ends at y[0, height], where *height* represents the height of the focus element, in case of the x-axis, or x[0, width] y[0,0] in case of the y-axis. Each line is then positioned by adding a `transform`[11] attribute with the values being equal to the relevant tick transform values.

The rectangles representing individual G4s are created using the `rect` element. As many rectangles as there are found G4s are generated and they are positioned based on their start position and score. To scale these values to fit into the graph, d3 function `scaleLinear()` is used. This function accepts a *domain* and a *range* values. It then takes the domain values

[9] https://www.w3schools.com/html/html5_svg.asp
[10] https://d3js.org/
[11] https://developer.mozilla.org/en-US/docs/Web/SVG/Attribute/transform

and maps them to output between the range values. In case of the x-axis, the domain is between the lowest nucleotide position to be displayed and the highest one. The range is always set to values from 0 to the width of the graph. The y-axis domain is set to the lower and upper bound values and its range is from the height of the graph to 0.

Before any G4s can be portrayed, it has to be decided which G4s to display based on the x-axis domain. On default, the domain is set from 0 to the sequence length. This can be changed by adjusting the context brush or zooming in on the graph, which will be explained in later paragraphs. Which data to display is determined by comparing individual G4s start and end positions to the domain values. If the starting position is greater than the upper-bound and the end position is smaller than the lower-bound, the G4 is not displayed at all. If the starting position is smaller than the lower-bound or the end position greater than the upper-bound, these positions are temporarily altered to the values of the lower and upper bound.

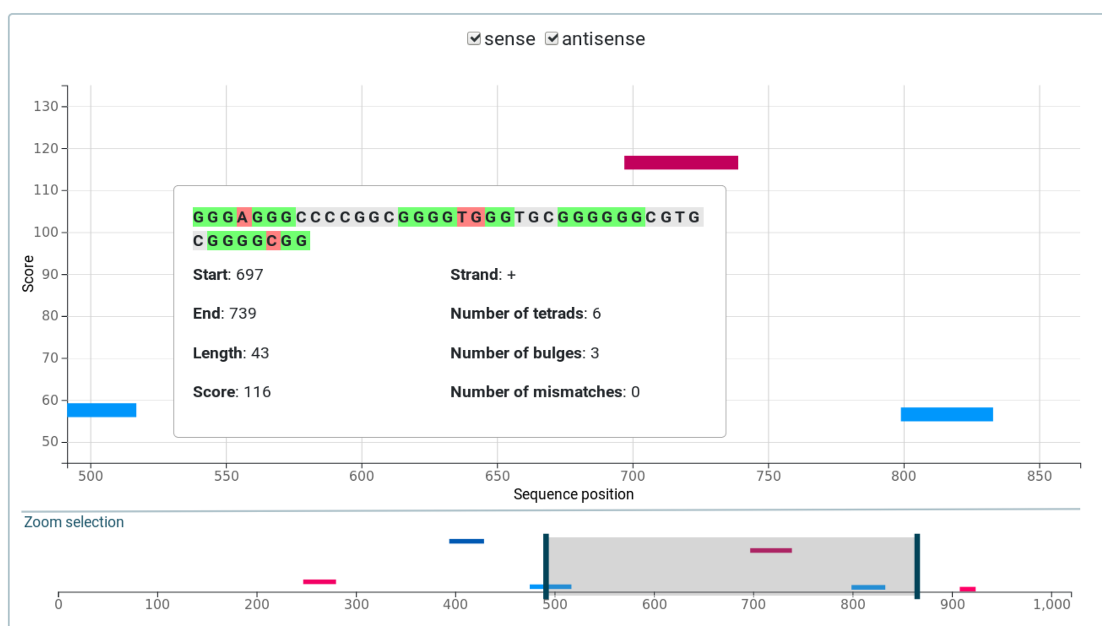


Figure 6.7: Detail of the graph component with visible tooltip and zoom.

The *context* is created similarly to the focus, only its y-axis range values are much smaller and its x-axis values are set and can not be changed. The brush covering the context is implemented using a d3 function `brushX()`, which handles the brush animation and fires a *brush end* event when the brush is moved. The event carries a `selection` attribute that is used to set the new x-axis domain. Once the new domain is set, the focus x-axis values are recomputed using these new values and new grid lines are created based on the new ticks that were generated. All `rect` elements are removed and the G4s to display are also reevaluated using the new x-axis domain. Once they are identified, new `rect` elements are created.

The whole *focus* area is covered by a rectangle that has a d3 function `zoom()` applied to it. This function takes a scale and translate extent and fires a *zoom* event when the user scrolls over it or zooms it by touch on touchscreens. The fired event carries a `transform` attribute that is used to rescale the x-axis domain. This attribute is computed automatically based on the extent of the zoom. Similarly to brush handling, once the new domain is set,

the focus x-axis and grid lines are recreated. The data to be displayed is recomputed and generated anew. Additionally, the brush element is instructed to position its handles according to the new domain.

The tooltip that appears when a user hovers over a rectangle is actually only a simple `div` element that is implemented in the `Detail.js` file. The element has an attribute `visible` that is set to false whenever none of the rectangles is hovered over. When the hover starts, an ID of that specific G4 is passed to the component and its visibility is set to true. The element is then positioned by adjusting its `left` and `top` style attributes. The attributes are set so that the tooltip is always displayed within the graph and adapt on every mouse move over the rectangle. When the mouse leaves the rectangle, the tooltip visibility is set back to false. While implementing the tooltip, a problem was encountered. If a scroll event was performed while the mouse was hovering over a rectangle, the event was passed directly to the root element and the whole page would scroll. This behaviour was eliminated by using a library called `react-scroll-locky`[12], that provides the possibility to disable scroll on the page. To enforce this restriction, the component had to be encapsulated in a `ScrollLocky` component that accepts an `enabled` attribute. This attribute is by default set to false, but changes to true when a hover event over rectangle is detected.

Server error

Server unavailability is caught every time a request is send. When the axios function catches an error, a change event is emitted that a server error occurred. This event is handled in the `App` component and affects only the Analyze and Tracks pages since those are the only pages that need a stable server connection. If there is no connection, the components on the page are replaced entirely by an error message that can be seen in Figure 6.8.

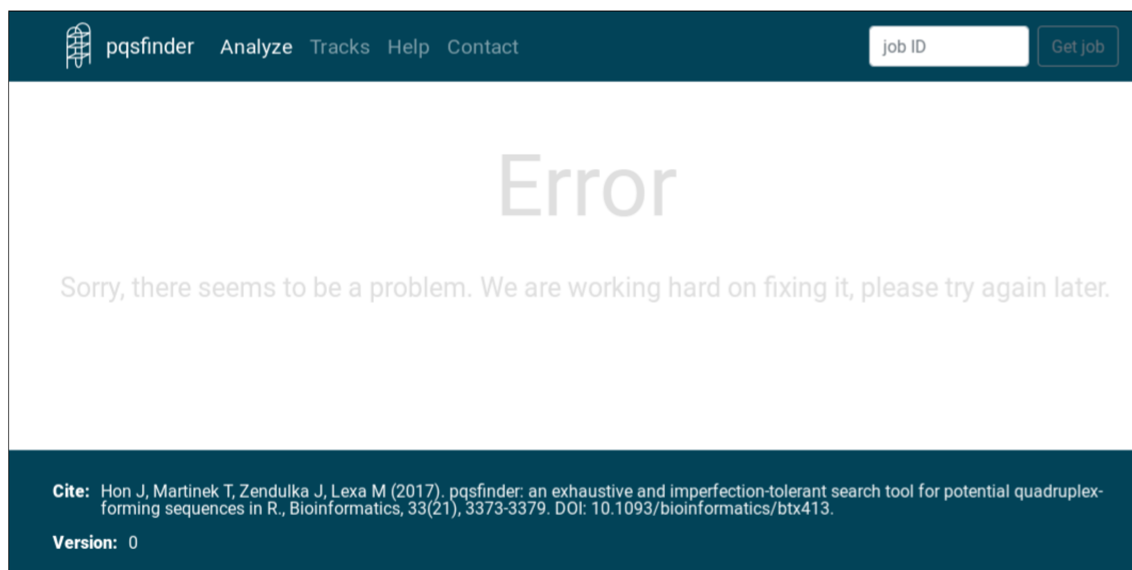


Figure 6.8: An error message displayed when the server is unavailable.

[12] https://www.npmjs.com/package/react-scroll-locky

Chapter 7

Conclusions

The aim of this thesis was to optimize a G-quadruplex (G4) identification tool *pqsfinder* and implement a web interface that uses *pqsfinder* and displays the search results. Both of these goals were accomplished and their outcomes have been released online.

For the algorithm to be optimized, it first had to be analyzed and profiled. The analysis showed a weakness in the algorithm's ability to process guanine rich sequences. In case of extremely high guanine density, the algorithm would even halt. Further examination of the algorithm provided sufficient information to suggest improvements. First optimization is based on adding a condition between individual G-run hits. Maximal potential score of current G4 is computed and then compared to two values. First value is the defined minimum required score. Second value represents the maximal found score of all reported G4s so far overlapping the starting position of the first run in the current potential quadruplex forming sequence. If the potential score is lower than the minimum score or the maximal found score, the search for other runs can be terminated. After optimization, the algorithm is up to 1500 times faster on not full G sequences. The speed up gets higher with rising G density in a sequence. To demonstrate the speed increase, first 10 million nucleotides in chromosome 21 of the human genome were analyzed by the algorithm before and after optimization. While the original algorithm took about 27 minutes to complete, the optimized version finished in 8 seconds. Overall, the optimization provided significant increase in speed and the algorithm's ability to process long sequences with high G density. The optimized algorithm was released on 2nd May 2019 as *pqsfinder* version 2.0.0[1].

The web interface was designed using a client-server architecture. To handle client side requests, a simple REST API was designed and implemented in R language. The web application is implemented as a single page application using JavaScript with the React[2] library. Design of individual components is partially inspired by existing G-quadruplex identification tools. The most important part of the web application is the graph representing found G4s. This component was implemented using the `D3.js`[3] library. The functioning web interface was deployed and can be found at https://pqsfinder.fi.muni.cz/.

[1] http://bioconductor.org/packages/release/bioc/html/pqsfinder.html
[2] https://reactjs.org/
[3] https://d3js.org/

Bibliography

[1] Abel, T.: *ReactJS: Become a Professional in Web App Development*. USA: CreateSpace Independent Publishing Platform. 2016. ISBN 1533118825, 9781533118820.

[2] Avery, O. T.: Studies on the chemical nature of the substance inducing transformation of pneumococcal types : induction of transformation by a desoxyribonucleic acid fraction isolated from pneumococcus type III. *Journal of Experimental Medicine*. vol. 79, no. 2. February 1944: pp. 137–158. doi:10.1084/jem.79.2.137.
Retrieved from: https://doi.org/10.1084/jem.79.2.137

[3] Bedrat, A.; Lacroix, L.; Mergny, J.-L.: Re-evaluation of G-quadruplex propensity with G4Hunter. *Nucleic Acids Research*. vol. 44, no. 4. January 2016: pp. 1746–1759. doi:10.1093/nar/gkw006.
Retrieved from: https://doi.org/10.1093/nar/gkw006

[4] Biffi, G.; Tannahill, D.; McCafferty, J.; et al.: Quantitative visualization of DNA G-quadruplex structures in human cells. *Nature Chemistry*. vol. 5, no. 3. January 2013: pp. 182–186. doi:10.1038/nchem.1548.
Retrieved from: https://doi.org/10.1038/nchem.1548

[5] Bochman, M. L.; Paeschke, K.; Zakian, V. A.: DNA secondary structures: stability and function of G-quadruplex structures. *Nature Reviews Genetics*. vol. 13, no. 11. October 2012: pp. 770–780. doi:10.1038/nrg3296.
Retrieved from: https://doi.org/10.1038/nrg3296

[6] Bruce Alberts, J. L., Alexander Johnson: *Molecular Biology of the Cell*. New York: Garland Science, 4th edition. 2002. ISBN 0-8153-3218-1.

[7] Charles Molnar, J. G.: Concepts of Biology – 1st Canadian Edition. https://opentextbc.ca/biology/. 2015. [Online; accessed 8-April-2019].

[8] Cornell, B.: DNA Replication. 2006. [Online; accessed 11-May-2019].
Retrieved from: http://ib.bioninja.com.au/standard-level/topic-2-molecular-biology/27-dna-replication-transcri/dna-replication.html

[9] Dhapola, P.; Chowdhury, S.: QuadBase2: web server for multiplexed guanine quadruplex mining and visualization. *Nucleic Acids Research*. vol. 44, no. W1. May 2016: pp. W277–W283. doi:10.1093/nar/gkw425.
Retrieved from: https://doi.org/10.1093/nar/gkw425

[10] Du, X.; Wojtowicz, D.; Bowers, A. A.; et al.: The genome-wide distribution of non-B DNA motifs is shaped by operon structure and suggests the transcriptional importance of non-B DNA structures in Escherichia coli. *Nucleic Acids Research*. vol. 41, no. 12. April 2013: pp. 5965–5977. doi:10.1093/nar/gkt308.
Retrieved from: https://doi.org/10.1093/nar/gkt308

[11] Hon, J.; Martínek, T.; Zendulka, J.; et al.: pqsfinder: an exhaustive and imperfection-tolerant search tool for potential quadruplex-forming sequences in R. *Bioinformatics*. vol. 33, no. 21. July 2017: pp. 3373–3379. doi:10.1093/bioinformatics/btx413.
Retrieved from: https://doi.org/10.1093/bioinformatics/btx413

[12] Kikin, O.; DAntonio, L.; Bagga, P. S.: QGRS Mapper: a web-based server for predicting G-quadruplexes in nucleotide sequences. *Nucleic Acids Research*. vol. 34, no. Web Server. July 2006: pp. W676–W682. doi:10.1093/nar/gkl253.
Retrieved from: https://doi.org/10.1093/nar/gkl253

[13] Kwok, C. K.; Merrick, C. J.: G-Quadruplexes: Prediction, Characterization, and Biological Application. *Trends in Biotechnology*. vol. 35, no. 10. October 2017: pp. 997–1013. doi:10.1016/j.tibtech.2017.06.012.
Retrieved from: https://doi.org/10.1016/j.tibtech.2017.06.012

[14] Mukundan, V. T.; Phan, A. T.: Bulges in G-Quadruplexes: Broadening the Definition of G-Quadruplex-Forming Sequences. *Journal of the American Chemical Society*. vol. 135, no. 13. March 2013: pp. 5017–5028. doi:10.1021/ja310251r.
Retrieved from: https://doi.org/10.1021/ja310251r

[15] Pavla Hubálková: G-kvadruplexy v oblasti lidských telomer a jejich terapeutický potenciál. *Chem. Listy*. vol. 109. 2015: pp. 918–922.
Retrieved from:
http://www.chemicke-listy.cz/docs/full/2015_12_918-922.pdf?fbclid=IwAR0AjXdobdAtz6YOax-_xsps9IP1vTEYyMiGXshQaJxqGBm25Yg3u9g_30o

[16] Rhodes, D.; Lipps, H. J.: G-quadruplexes and their regulatory roles in biology. *Nucleic Acids Research*. vol. 43, no. 18. September 2015: pp. 8627–8637. doi:10.1093/nar/gkv862.
Retrieved from: https://doi.org/10.1093/nar/gkv862

[17] Trestle Technology, LLC: *plumber: An API Generator for R*. 2018. r package version 0.4.6.
Retrieved from: https://CRAN.R-project.org/package=plumber

[18] Varizhuk, A.; Ischenko, D.; Smirnov, I.; et al.: An Improved Search Algorithm to Find G-Quadruplexes in Genome Sequences. January 2014. doi:10.1101/001990.
Retrieved from: https://doi.org/10.1101/001990

[19] Varizhuk, A.; Ischenko, D.; Tsvetkov, V.; et al.: The expanding repertoire of G4 DNA structures. *Biochimie*. vol. 135. April 2017: pp. 54–62. doi:10.1016/j.biochi.2017.01.003.
Retrieved from: https://doi.org/10.1016/j.biochi.2017.01.003

[20] Watson, J. D.; Crick, F. H. C.: Molecular Structure of Nucleic Acids: A Structure for Deoxyribose Nucleic Acid. *Nature*. vol. 171, no. 4356. April 1953: pp. 737–738. doi:10.1038/171737a0.
Retrieved from: https://doi.org/10.1038/171737a0

[21] Yadav, V. K.; Abraham, J. K.; Mani, P.; et al.: QuadBase: genome-wide database of G4 DNA occurrence and conservation in human, chimpanzee, mouse and rat promoters and 146 microbes. *Nucleic Acids Research*. vol. 36, no. Database. December 2007: pp. D381–D385. doi:10.1093/nar/gkm781.
Retrieved from: https://doi.org/10.1093/nar/gkm781

Appendix A

Contents of Attached CD

- `text/` – containing LaTeXsources and files used in this text
- `pqsfinder/` – containing the optimized pqsfinder R package source codes
- `pqsfinder-backend/` – containing the REST API and a script to start the server
- `pqsfinder-frontend/` – containing the web application source codes along with an npm script that compiles the application and starts the server