Univerzita Komenského v Bratislave

Fakulta matematiky, fyziky a informatiky

Efektívne vyhľadávanie záznamov v databáze dbNSFP

bakalárska práca

2015

Adam Mikušovský

Univerzita Komenského v Bratislave

Fakulta matematiky, fyziky a informatiky

Efektívne vyhľadávanie záznamov v databáze dbNSFP

bakalárska práca

Študijný program: Informatika
Študijný odbor: 2508 Informatka
Školiace pracovisko: Katedra Informatiky
Školiteľ: Mgr. Jaroslav Budiš

Bratislava, 2015
Adam Mikušovský

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta:	Adam Mikušovský
Študijný program:	informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor:	9.2.1. informatika
Typ záverečnej práce:	bakalárska
Jazyk záverečnej práce:	slovenský
Sekundárny jazyk:	anglický

Názov: Efektívne vyhľadávanie záznamov v databáze dbNSFP
Effective querying in the dbNSFP database.

Cieľ: Databáza dbNSFP zhromažďuje údaje o genetických variantoch v textových súboroch typu TSV. Veľký objem dát komplikuje uloženie dát v SQL databáze, navyše povaha dát a dotazov umožňuje efektívnejší výber údajov.
Úlohou študenta je navrhnút, implementovať a porovnať viacero dátových štruktúr na získavanie dát z tejto databázy. Vybrané riešenie pripojí na webovú službu Variant Annotation Service.

Vedúci:	Mgr. Jaroslav Budiš
Katedra:	FMFI.KI - Katedra informatiky
Vedúci katedry:	doc. RNDr. Daniel Olejár, PhD.
Dátum zadania:	13.10.2014

Dátum schválenia: 28.10.2014 doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Poďakovanie: Na úvod by som sa chcel najmä poďakovať svojmu školiteľovi Mgr. Jaroslavovi Budišovi za veľmi dobrú spoluprácu.

Abstrakt

Databáza dbNSFP zhromažďuje údaje o genetických variantoch v textových súboroch typu TSV. Veľký objem dát komplikuje vyhľadanie potrebných záznamov. Navyše údaje majú špecifické usporiadanie vo forme intervalov, vďaka čomu je možné použiť efektívnejší spôsob získania požadovaných údajov. Cieľom bakalárskej práce bolo implementovať a porovnať viacero dátových štruktúr na získavanie dát s tejto databázy, ako aj porovnanie z existujúcou SQL a NoSQL databázou. Výsledkom práce je nosný zdroj atribútov genetických variantov pre webovú službu Variant Annotation Service.

Kľúčové slová: Genetický variant, Databáza dbNSFP, Efektívne dátové štruktúry

Abstract

Database dbNSFP aggregates attributes of genetic variants in TSV text files. Large amount of data complicates import of data to the SQL databases. Furthermore, the data and queries have interval character, allowing us to use more effective way of collecting required data.
The aim of the thesis was to implement and compare multiple data structures for extracting data from the database, as well as a comparison with an SQL and NoSQL database. The most feasible structure would be used as a main attribute provider of attributes of genetic variants for web service Variant Annotation Service.

Keywords: Genetic variant, Database dbNSFP, Effective data structures

Obsah

Úvod **1**

1 Prehľad **2**
1.1 Biologický úvod 2
1.2 dbNSFP 4
1.2.1 Formát databázy dbNSFP 5
1.3 Projekt Variant Annotation Service 6

2 Implementácia **11**
2.1 Rozhranie 11
2.1.1 getAttributes 12
2.1.2 getInterval 12
2.2 Objekty 12
2.2.1 Query 12
2.2.2 Attribute 12
2.2.3 Attributes 13
2.2.4 Interval attributes iterator 13
2.2.5 Index 14
2.3 Návrhové vzory 15
2.3.1 Strategy 15

3 Dátové štruktúry **18**
3.1 Binárne vyhľadávanie 18

3.1.1 getAttributes 18
3.1.2 getInterval 19
3.2 Avl vyhľadávacie stromy 19
3.2.1 getInterval 21
3.3 Vyhľadávanie za pomoci intervalov 21
3.4 Skip list 23
3.4.1 getAttributes 24
3.4.2 getInterval 24
3.5 Metóda konštantného okna 24
3.5.1 Hľadanie optimálneho rozdelenia 25
3.5.2 getAttribute 26
3.5.3 getInterval 27
3.6 Sql riešenie 28
3.7 MongoDB 29
3.8 Časová a pamäťová zložitosť 29

4 Porovnanie **30**
4.1 Časová efektívnosť 30
4.2 Pamäťové nároky 33
4.3 Vyhodnotenie 34

Záver **35**

Zoznam obrázkov

1.1 Špirála DNA s prepojením AT/CG, (zdroj: wikipedia.org) . . 3
1.2 Distribúcia intervalov podľa dĺžky na chromozóme 1. 8
1.3 Rozdelenie dĺžok intervalov na všetkých chromozómoch okrem Y. 9
1.4 Rozdelenie dĺžok intervalov na chromozóme Y. 9
1.5 Distribúcia veľkosti skokov medzi intervalmi na všetkých chromozómoch. 10

2.1 GetAttributesInterface UML diagram 11
2.2 Index UML diagram 13
2.3 Attribute UML diagram 13
2.4 Index UML diagram 14
2.5 Strategy UML diagram 17

3.1 Ukážka binárneho vyhľadávania 19
3.2 Reprezentácia vrcholov pre načítanie a vyhľadávanie. 20
3.3 Inorder traverzovanie. 21
3.4 Utriedené pole obsahujúce polia indexov 23
3.5 Ukážka skiplistu 24
3.6 Metóda optimálneho okna. 25
3.7 Pamäťová závislosť od veľkosti okna 27

Úvod

Cieľom tejto bakalárskej práce je navrhnúť, implementovať a porovnať viacero dátových štruktúr, ktoré umožňujú efektívne vyhľadávanie v databáze genetických variantov dbNSFP [19]. Podstatnou vlastnosťou riešenia je čo najnižšia časová zložitosť, kvôli potenciálne veľkému množstvu dotazov v krátkom časovom horizonte.
V praxi nie sú akceptovateľné dlhodobé výpadky webovej služby, najlepšie riešenie bude slúžiť ako hlavný dátový zdroj pre webovú službu Variant Annotation Service. Preto je snaha znížiť časové nároky na opätovné vytvorenie dátovej štruktúry a pomocných súborov v prípade novej verzie databázy. Kľúčom k úspechu bude predpočítanie si niektorých údajov, ktoré sa pri načítaní dátovej štruktúry využívajú, čo podstatne skráti čas výpadku.
Predstavíme si niektoré biologické pojmy ako chromozóm, alela a mutácia. Opíšeme význam význam parametrov obsiahnutých dát v dbNSFP a zobrazíme štatistické distribúcie o nich.
V druhej kapitole si zadefinujeme spoločné objekty, ktoré budeme využívať vo všetkých implementáciách stratégií. Vytvoríme si interface, s ktorým sa bude najlepšie riešenie napájať na webovú službu a tiež navrhneme objektový model pre jednoduchšiu implementáciu a testovanie.
Po tejto časti bude nasledovať prehľad implementovaných dátových štruktúr, ako aj riešenie za pomoci databázových systémov PostgreSQL a MongoDB. Na záver si zadefinujeme spôsob testovania, implementujeme ho a uvedieme výsledky testov časovej efektivity a pamäťovej náročnosti riešení.

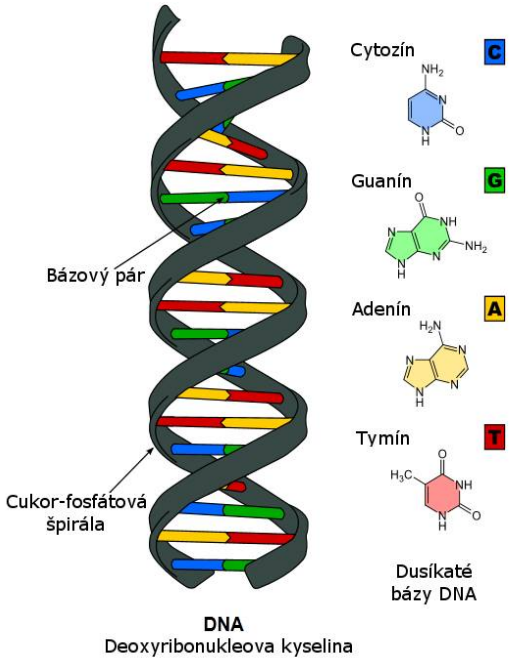
Kapitola 1

Prehľad

1.1 Biologický úvod

Deoxyribonukleová kyselina známa tiež ako DNA (z anglického slova deoxyribonucleic acid), patrí spolu s kyselinou ribonoukleovou RNA medzi nukleové kyseliny, nachádzajúce sa v jadrách buniek. DNA je nositeľom genetickej informácie bunky, riadi jej rast, dedenie, regeneráciu. Jej základnou zložkou je nukleodit, ktorý pozostáva s troch zložiek: kyseliny fosforečnej, molekuly deoxiribózy a dusíkatej bázy. Dusíkatú bázu tvoria štyri zlúčeniny: Adenín (skratka A), Guanín (G), Cytozín (C) a Tymín (T). Tieto bázy spájajú obe vlákna pomocou vodíkových väzieb do tvaru známej špirály, kde platí Watsonovo-Crickovo pravidlo, ktoré hovorí o tom, že sa párujú vždy purínová s pyrimidínovou bázou. V našom prípade sa teda adenín páruje s tymínom a guanín s cytozínom (obr 1.1).

Niektoré oblasti dusíkatých báz podmieňujú vlastnosti organizmu. Takéto oblasti nazývame gény. Gény obsiahnuté v bunkách sú zabudované v chromozómoch očíslovaných od 1 po 22 a X,Y. Každý chromozóm sa skladá z jednej makromolekuly DNA, súčasne chromozóm obsahuje len určité gény, ktoré sú uložené lineárne za sebou. Pozície, na ktorých sa gény nachádzajú, sú označované ako lokusy(relatívne súvislé úseky). Pozície na chromozóme



Obr. 1.1: Špirála DNA s prepojením AT/CG, (zdroj: wikipedia.org)

označujeme číselne a maximálna hodnota nepresiahne veľkosť 247 199 719.

Každý gén môže mať viacero foriem (postupností dusíkatých báz), tieto formy nazývame alely. Ak existuje viacej alel na jednej pozícii hovoríme o genetickom polymorfizme (variant génu). Nato aby sa odlišnosť v géne považovala za alelu musí ale platiť, že výskyt danej odlišnosti prevyšuje 1% v bežnej populácii. Vznik novej alely je daný mutáciou génu, ku ktorej dochádza spontánne alebo indukované (vyvolaná určitými vonkajšími faktormi). Ďalšia existencia mutácie je spočiatku závislá od náhody a veľkosti populácie, čím je populácia menšia, tým sa v nej vyskytuje menej mutácii a naopak.

Poznámka: V ďalšom texte budeme používať pojem alela na označenie jednonukleónového polymorfizmu(bodovej mutácie) SNP a budeme ich ozna-

čovať A,C,G,T.

Vznik mutácie vedie v niektorých prípadoch k zlepšeniu vlastností, čo sa využíva napríklad pri šľachtení rastlín. V takom prípade hovoríme o pozitívnej mutácii. Ale čo sa týka živočíchov, väčšina mutácii je neutrálnych a z tých, ktoré majú vplyv na vlastnosti, je viac mutácií škodlivých.
Podrobnejšie informácie z biológie a genetiky môžete nájsť v knihe Genetika [32].

1.2 dbNSFP

Úlohou databázy NSFP (database for nonsynonymous Single-nucleotide polymorphism functional predictions), je uchovávať informácie o variantoch na ľudskom genóme.
V súčasnej dobe nie je možné z časových a finančných dôvodov otestovať následky každého variantu experimentálne, využívajú sa na to algoritmy, za pomoci ktorých je možné určiť z určitou pravdepodobnosťou zmeny prejavu zapríčinenú mutáciou. Vytvorenie dbNSFP spočíva vo vyhodnotení všetkých možných mutácii, jednonukleónových polymorfizmov (SNP), v ľudskom genóme (celkom 91 569 327 pre verziu 2.7). Vyhodnocovanie je realizované pomocou rôoznych algoritmov, ktoré môžeme všetky rozdeliť do štyroch kategórií:

Funkcionálna predpoveď určuje, ako zmena nukleotidu na géne ovplyvňuje funkciu proteínu v géne, niektoré metódy z tejto kategórie dokážu na základe mutácie určiť, aký patologický dôsledok z nej vyplýva. Jedná sa o tieto algoritmy: Polyphen-2 v2.2.2 [1], SIFT [23], LRT [21], MutationTaster [26], MutationAssessor[15], FATHMM[13], CADD[34] a VEST[16].

Skóre zachovania určuje, ktoré sekvencie majú funkčný význam. Jednou z metód je GERP++, ktorý identifikuje funkčné sekvencie na základe

porovnania genómu odlišných živočíšnych druhov a porovnaním zistí, ktoré z nich boli zachované počas evolúcie, čo uľahčí nájdenie sekvencií, ktoré majú funkčný význam. Do tejto kategórie ešte patria výsledky nasledovných prístupov: phyloP46 [29], phastCons [30], GERP++ [6].

Iné zdroje popisov variantov poskytujú doplňujúce informácie. Ide napríklad o rozdelenie proteínových sekvencií do kategórií, o popísanie génov a mechanizmov medzi nimi, ktoré vedú k srdcovým, pľúcnym a krvným poruchám. Zdrojmi týchto a ďalších informácii sú: Interpro [2], 1000 Genomes project [31], ANNOVAR [35], ESP [24], dbSNP [28] a clinvar [17].

Iné zdroje popisov génov poskytujú doplňujúce informácie pre cele gény. Zdrojmi týchto a ďalších informácii sú: (HGNC [25], Uniprot [5], IntAct, GWAS catalog [36], BioGRID [33], ...);

Nad databázou zväčša vyhľadáva podľa trojice chromozómu, pozícií na chromozóme a allely. Tieto tri atribúty sú pre nás najvýznamnejšie, lebo iba na základe týchto hodnôt sa presne určí ľubovoľný záznam v databáze. [18].

1.2.1 Formát databázy dbNSFP

TSV (Tab separated values) je jednoduchý formát pre ukladanie údajov formou pripomínajúcou tabuľku, kde na oddeľovanie hodnôt sa používa znak tabulátora a na oddelenie záznamov sa používa symbol nového riadku, každý záznam je teda tvorený jedným riadkom v súbore [20].
Vďaka svojej jednoduchosti je široko používaný, napríklad sa využíva pri prenose údajov medzi databázovými a tabuľkovými programami.

Databáza dbNSFP je dostupná v TSV formáte, kde na prvom riadku sú uvedené názvy atribútov, pričom záleží na ich poradí, lebo na základe neho sú v nasledujúcich riadkoch uvedené hodnoty pre jednotlivé atribúty záznamov. Jeden záznam je tvorený informáciami o jednej alele pre pozíciu

na chromozóme. Ak hodnota niektorých atribútov v zázname nie je ešte známa, nachádza sa na jej pozícií symbol "."(bez úvodzoviek), podobne ak niektorý atríbút obsahuje viacero hodnôt sú oddelené ";". Ak by sme teda vedeli tieto údaje naformátovať podľa symbolu tabulátora, dostali by sme podobné zobrazenie ako v tabuľkových programoch.

Záznamy k variantom sú v databáze dbNSFP rozdelené na viacero súborov podľa čísla chromozómu. V ďalšom texte budeme pod dbNSFP chápať tieto súbory typu TSV.

Na jednej pozícii na géne sa v dbNSFP zvyčajne nachádzajú tri alebo dve obmeny. To pre nás znamená zmenu písmena v zázname a teda viacero riadkov pre jednu pozíciu.
Pozície na chromozóme vytvárajú intervaly rôznej dĺžky(lokusy). Tieto intervaly sú pomerne rovnomerne rozdistribuované po celom chromozóme, distribúcia je znázornená na obrázku 1.2. Taktiež pre tieto intervaly platí, že na niektorých ich pozíciách sa nenachádzajú v databáze dbNSFP záznamy. Nazvime si tieto prázdne miesta skokmi, ktoré môžu nadobúdať veľkosť jednej alebo dvoch pozícií. Nevylučuje sa ani existencia väčších skokov, ale takéto sa vyskytujú zriedkavejšie. Takisto vymedzenie presnej hranice, kedy budeme hovoriť o súčasti intervalu(lokusu) a kedy už o inom intervale, je podstatné pre niektoré riešenia. Viac sa ale dozvieme pri implementáciách.

Rozdelenie veľkosti intervalov a skokov si môžete všimnúť na obrázkoch 1.3, 1.4 a 1.5. Pre skrátenie sme použili iba jeden graf pre všetky chromozómy, pretože majú približne rovnaké rozdelenie a jeden samostatný pre chromozóm Y, ktorý má odlišné vlastnosti spôsobené menšou dĺžkou, obsahom menšieho počtu génov a najmä pomalšou evolúciou.

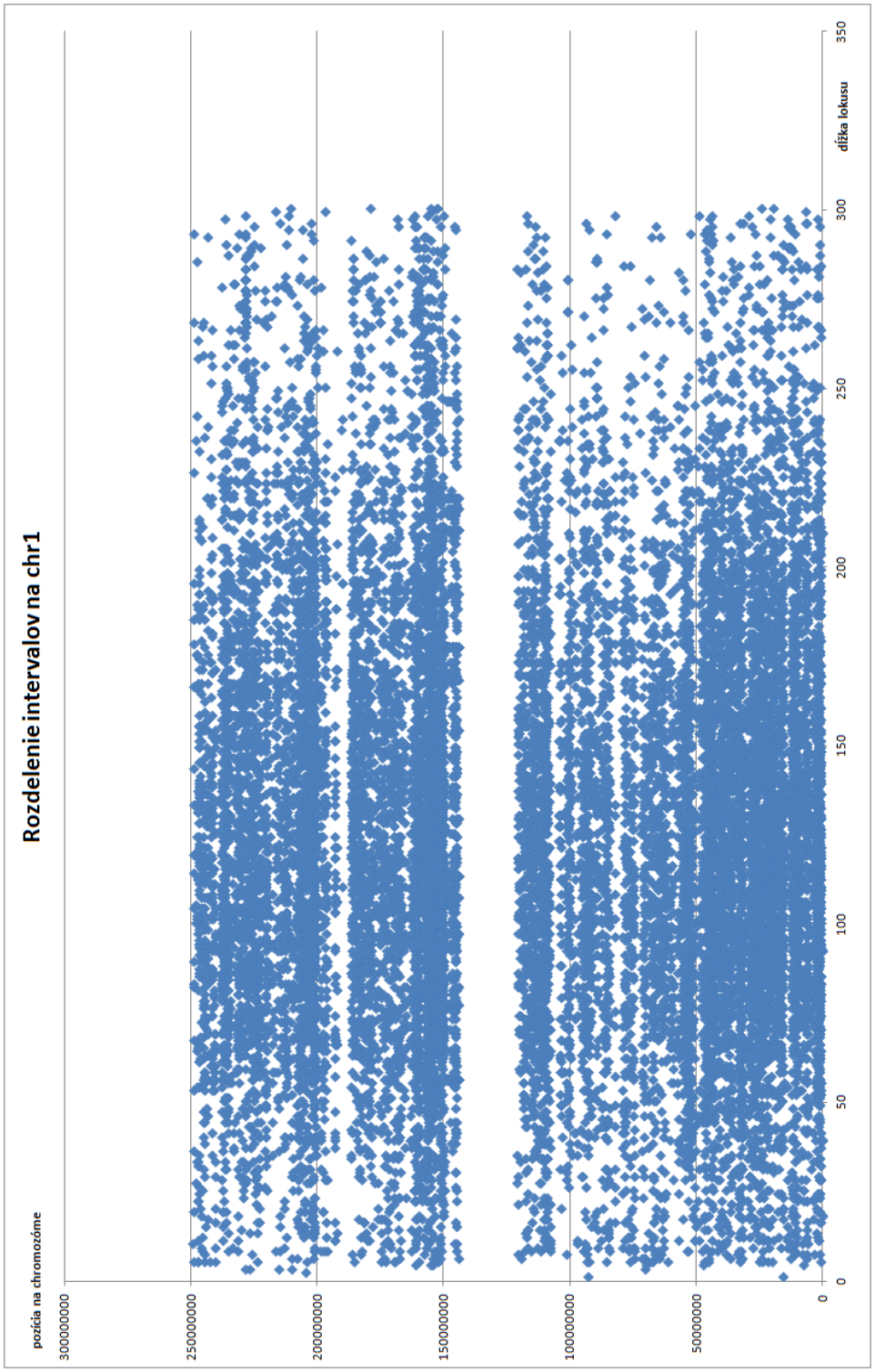
1.3 Projekt Variant Annotation Service

Variant Annotation Service (VAS) zhromažďuje informácie o genetických variantoch z viacerých zdrojov. Hlavným poskytovateľom atribútov je databáza

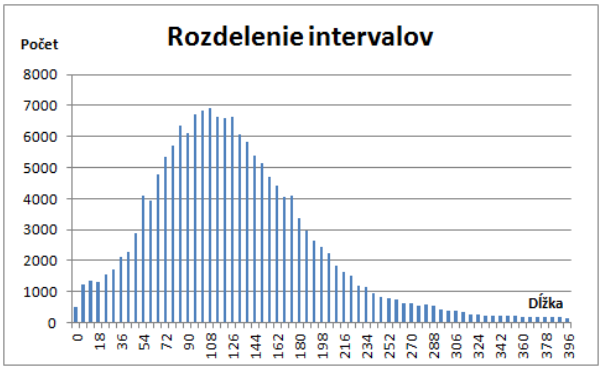
dbNSFP. Po vyplnení formulára obsahujúceho tri hodnoty potrebné na vykonanie výberu služba vráti atribúty v XML alebo HTML podobe. Službu navyše využíva aplikácia Variant Annotatioin Analyser na anotáciu variantov.

Pôvodné riešenie pre výber z dbNSFP bolo napísané v jazyku Python, ktorého nevýhodou bolo obtiažnejšie napájanie k VAS, ktorý je implementovaný v Jave. Pôvodné riešenie nevyužívalo optimálnu dátovú štruktúru čo s kombináciou jazyka Python viedlo k neefektívnemu vyhľadávaniu a celkovej pomalosti. Našou snahou bolo preto urýchliť vyhľadávanie a zjednodušiť napájanie na webovú službu.

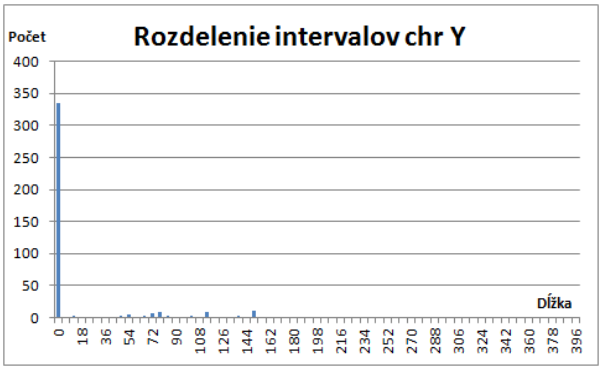
Projekt Variant Annotation Service bude zdrojom genetických informácií, ktoré môže získavať z viacerých zdrojov. Jedným z nich bude aj naša implementácia. Kvôli niektorým pozitívnym vlastnostiam napájanie zabezpečujeme pomocou javového frameworku Spring.



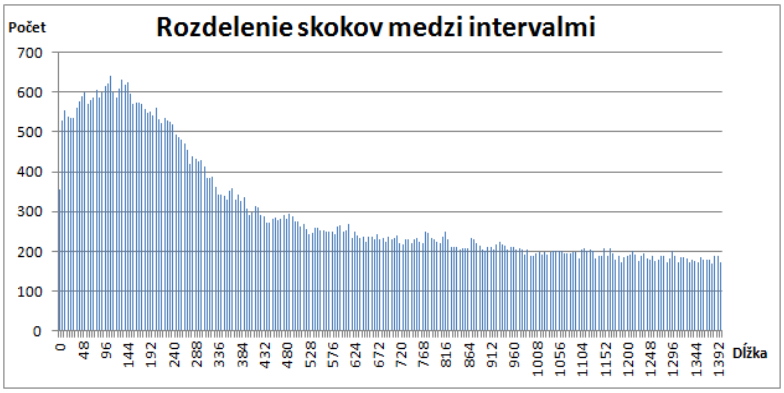
Obr. 1.2: Distribúcia intervalov podľa dĺžky na chromozóme 1.



Obr. 1.3: Rozdelenie dĺžok intervalov na všetkých chromozómoch okrem Y.



Obr. 1.4: Rozdelenie dĺžok intervalov na chromozóme Y.



Obr. 1.5: Distribúcia veľkosti skokov medzi intervalmi na všetkých chromozómoch.

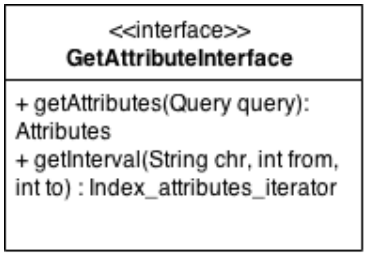
Kapitola 2

Implementácia

Pre realizáciu implementácie sme si vytvorili objektový model, ktorý je znázornený na obrázku 2.5.

2.1 Rozhranie

Aby sme zabezpečili kompatibilitu všetkých stratégií so službou Variant Annotation Service, určili sme si spoločný interface (2.1) obsahujúci dve metódy `getAttributes` a `getInterval`.



Obr. 2.1: GetAttributesInterface UML diagram

2.1.1 getAttributes

```
Attributes getAttributes(Attributes attributes, Query query)
```

Metóda slúži na získanie záznamu podľa hodnôt, ktoré sú uložené v objekte `Query`. V prípade ak chromozóm obsahuje hľadanú pozíciu a alelu, naplní sa `attriubtes` záznamom z dbNSFP.

2.1.2 getInterval

```
Interval_attributes_iterator getInterval(String chr,
long from, long to)
```

Pomocou tejto metódy získame hodnoty všetkých záznamov nachádzajúcich sa na intervale určeného hodnotami `from` a `to`.

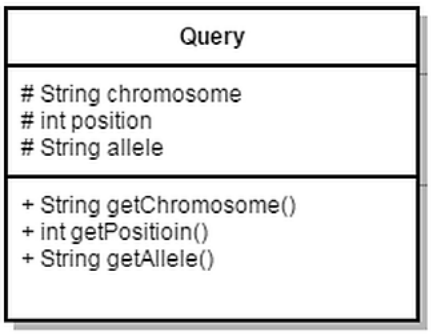
2.2 Objekty

2.2.1 Query

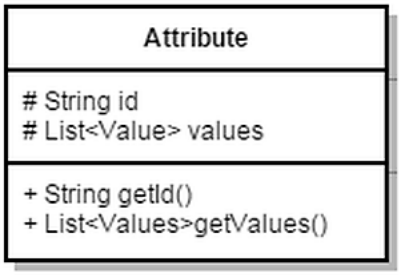
Objekt Query predstavuje vstupnú požiadavku zadanú používateľom pre metódu `getAttributes`. Obsahuje tri základné hodnoty, podľa ktorých sa vyberie záznam. Ide o identifikáciu Chromozómu, pozície na ňom a Allely(2.2). K hodnotám sa pristupuje cez get metódy.

2.2.2 Attribute

Predstavuje hodnoty, ktoré sa nachádzajú na pozícii atribútu v TSV súbore. Obsahuje identifikačný reťazec atribútu a zoznam hodnôt(2.3).



Obr. 2.2: Index UML diagram



Obr. 2.3: Attribute UML diagram

2.2.3 Attributes

Pozostáva zo zoznamu objektov Attribute pre záznam v databáze. Hodnoty sa pridávajú cez metódu `boolean addAttribute(Attribute attribute)`, kde návratová hodnota je **true**, ak záznam pre atribút doposiaľ nebol vložený, inak **false**.

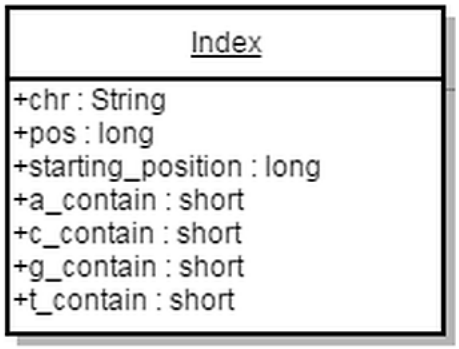
2.2.4 Interval attributes iterator

Slúži ako návratová hodnota metódy getInterval, ktorá umožňuje prechádzanie cez záznamy všetkých variantov nachádzajúcich sa na zadanom intervale. Výhodou tohto riešenia je, že v pamäti sa v jednom momente uchovávajú

hodnoty atribútov len pre jednu alelu. Presnejšie Interval attributes iterator implementuje interface `Iterator<Attributes>`, z čoho vyplýva podpora pre metódy hasNext() a next() slúžiace na iterovanie cez indexy, respektíve ich hodnoty na intervale. Konkrétne v konštruktore dostane zoznam Indexov a metódou next() sa iteruje cez alely, ktoré obsahujú nejaký záznam. Je podstatné, aby zoznam Indexov bol z utriedeného intervalu a obsahoval všetky indexy na ňom. Lebo z dôvodu časovej optimalizácie sa vytvára len jeden RandomAccesReader, ktorý sa posúva vždy o časť, ktorá sa prečítala.

2.2.5 Index

Nadmerná veľkosť databázy nám neumožňuje uchovávať v pamäti celú databázu v textovej podobe, zvolili sme preto riešenie, pri ktorom sa v pamäti indexujú iba pozície v textových súboroch, na ktorých začínajú záznamy pre jednotlivé alely.



Obr. 2.4: Index UML diagram

Pozície sú uložené v objektoch triedy `Index`, ktorá je znázornená na UML diagrame (2.4). V nej si udržiavame hodnotu chromozómu a pozíciu na nej. Atribút `starting_position` obsahuje počiatočnú pozíciu v TSV súbore. V premenných `a_contain`, `c_contain`, `g_contain` a `t_contain` je uložená číselná hodnota počtu bajtov, ktoré obsahuje záznam pre jednotlivé alely. Tieto

údaje a RandomAccessFile nám postačujú na získanie záznamov z dbNSFP.

Čítanie z TSV súborov potom funguje nasledovne: posunieme sa na počiatočný bajt v TSV súbore na základe hodnoty
`starting_position + var`, kde var určuje obsah alel, ktoré sú pred alelou, na ktorú sa pýtame. Tento krok uskutočníme metódou RandomAccesFile.seek(long). Po nastavení sa na správnu pozíciu v súbore potrebujeme prečítať vhodný počet bajtov, ktorý sa nachádza v `*_contain`(znak "*"reprezentuje hľadanú alelu). Po zavolaní metódy `RandomAccesFile.read(byte[] array)` sa do poľa array, o veľkosti `*_contain`, uložia hodnoty začínajúce na pozícii `starting_position + var` a končiace `starting_position + var + *_contain`. Po načítaní nám teraz už len stačí previesť pole bajtov do podoby stringu. Takýmto spôsobom budeme uskutočňovať čítanie z TSV súborov.
Vytvorenie indexov je pomerne časovo náročné, preto si ich po vytvorení uložíme do súborov, ktoré sa využijú pri opätovnom načítaní.

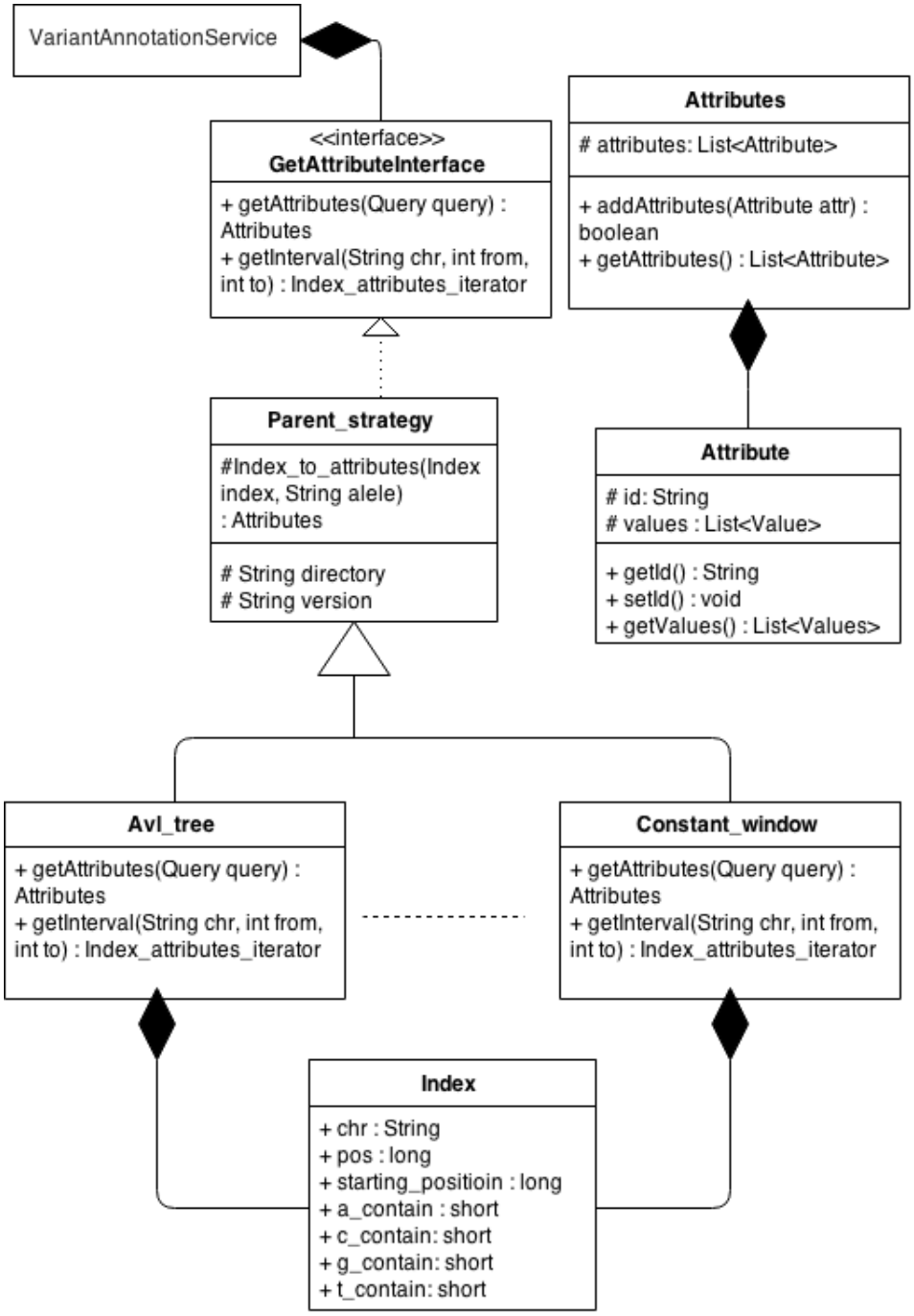
2.3 Návrhové vzory

Pred začatím akejkoľvek implementácie je odporúčané vytvoriť si určitú hierarchiu tried, vďaka čomu si uľahčíme ďalšiu prácu, pridávanie funkcionality a testovanie.

2.3.1 Strategy

Návrhový vzor Strategy sa používa, ak potrebujeme vyberať medzi rozdielnymi algoritmami pred spustením funkcie alebo celého programu. Rovnaký prístup zvolíme aj my, kde každá stratégia výberu záznamu z dbNSFP bude implementovať metódy getAttributes a getInterval. Tiež si môžeme uvedomiť, že veľa metód bude spoločných pre všetky stratégie. Aby sme sa vyhli písaniu rovnakých metód v každej stratégii, je výhodnejšie použiť rodičovskú triedu, v ktorej si zadefinujeme funkcie a premenné, s ktorými stratégie

pracujú. Od rodičovskej triedy potom všetky stratégie budú dediť jej vlastnosti(2.5). [9]



Obr. 2.5: Strategy UML diagram

Kapitola 3

Dátové štruktúry

Máme teda indexy, ktorých jeden chromozóm má v priemere dva milióny a súčasne vieme o nich, že sa vyskytujú v blokoch, ktoré sú rovnomerne rozdistribuované po chromozóme. Požiadavky na riešenie sú v rýchlosti nájdenia indexu v dátovej štruktúre a nájdenia indexov nachádzajúcich sa na určitom intervale. Riešenie musí byť rýchle a musí rozumne využívať pamäťové zdroje.

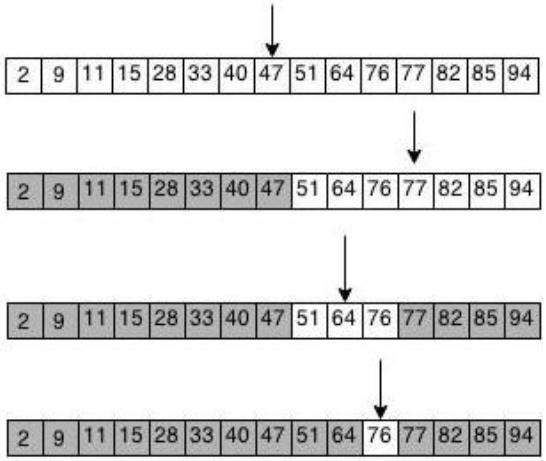
3.1 Binárne vyhľadávanie

Vzhľadom na povahu dát nachádzajúcich sa v databáze a podmienky výberu, môžeme sa na tento problém pozrieť ako na vyhľadávanie v utriedenom poli. Všetko čo potrebujeme zabezpečiť je zistiť pri načítaní, aký je počet indexov, ktoré boli vytvorené pre jednotlivé chromozómy. Následne alokujeme pole požadovanej veľkosti, do ktorého uložíme indexy podľa vzrastajúcej pozície, ktorej zodpovedá aj zoradenie záznamov v databáze NSFP.

3.1.1 getAttributes

Metódou getAttributes sa vyhľadá index v utriedenom poli, ktoré je určené číslom chromozómu. Vyhľadanie sa uskutoční pomocou binárneho vyhľada-

nia (3.1). V prípade, ak nájdeme index v poli, potrebujeme ešte vyhodnotiť, či allela, na ktorú sa odkazujeme, obsahuje záznam. Následne naplníme pole `Attributes` hodnotami z dbNSFP.



Obr. 3.1: Ukážka binárneho vyhľadávania

3.1.2 getInterval

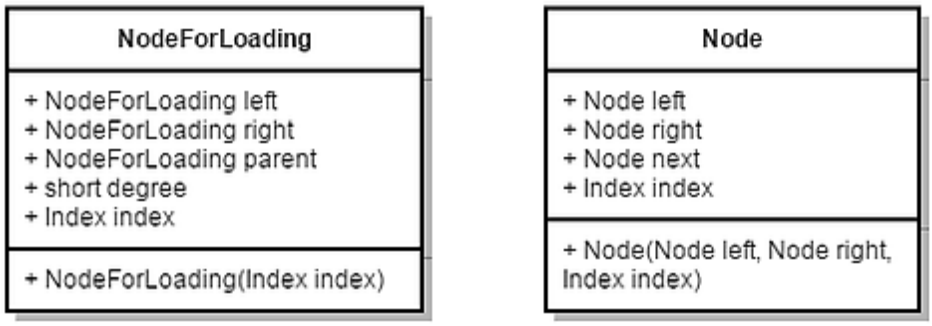
Zo vstupných hodnôt `from` a `to` nájdeme počiatočnú a koncovú pozíciu v utriedenom poli indexov, označme si ich ako `j, k`. Po zistení týchto hodnôt vložíme indexy nachádzajúce sa na intervale `<j; k>` do zoznamu, ktorý sa podsunie ako parameter konštruktora `Interval_attributes_iterator`. Vytvorený iterátor sa vráti ako návratová hodnota.

3.2 Avl vyhľadávacie stromy

Avl vyhľadávacie stromy [14] sú binárne samo vyťažovacie stromy, z čoho vyplýva, že toto riešenie nepotrebuje predpoklad utriedenia vstupných hodnôt.

Vytvorenie avl stromu má časovú zložitosť *O(nln(n))*. V našom prípade samovyvažovanie stromu sa uskutočňuje len pri vkladaní prvkov do stromovej štruktúry, kde vyváženosť znamená, že rozdiel hĺbok medzi ľavým a pravým synom nie je väčšia ako jedna. Overenie hĺbok sa testuje zdola nahor po ceste, ktorou sme išli pri vložení posledného vrcholu. Vyvažovanie stromu sa zabezpečí vhodnými rotáciami vrcholov, ktoré nie sú vyvážené. Na overenie a uskutočnenie rotácie je potrebné poznať, respektíve pamätať si synov, rodiča a vzdialenosť k listom(hĺbku). Tieto hodnoty sú potrebné iba pri vytváraní Avl stromu. Navyše po jeho načítaní už žiadané vrcholy nebudú pridávané, je ich teda vhodné po vytvorení stromu kvôli úspore pamäti odstrániť. Súčasne nám chýba parameter, ktorý by nám uľahčil jednoduchšie traverzovanie cez indexy.

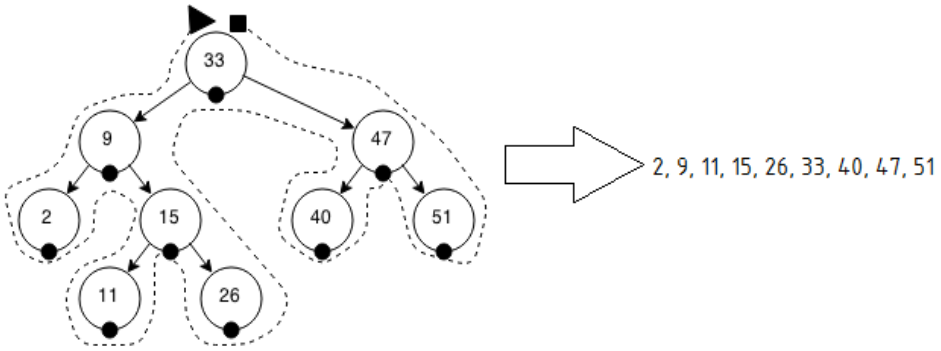
Z týchto dôvodov sme si vytvorili vrchol pre načítavanie a vrchol, ktorý slúži



Obr. 3.2: Reprezentácia vrcholov pre načítanie a vyhľadávanie.

na vyhľadávanie(3.2). Po vytvorení AVL stromu sa prekopírujú hodnoty do nového Binárneho stromu, s rovnakou architektúrou, uzly ale neobsahujú atribúty pre nadradený uzol a hĺbku.

Po prekopírovaní a zmazaní pôvodného stromu potrebujeme zabezpečiť ako zjednodušiť vyhľadávaie susedných pozícií. Zvolili sme riešenie pomocou inorder traverzovania stromov 3.3, kde si každý vrchol zapamätá svojho nasledovníka vzhľadom na pozíciu na chromozóme.



Obr. 3.3: Inorder traverzovanie.

3.2.1 getInterval

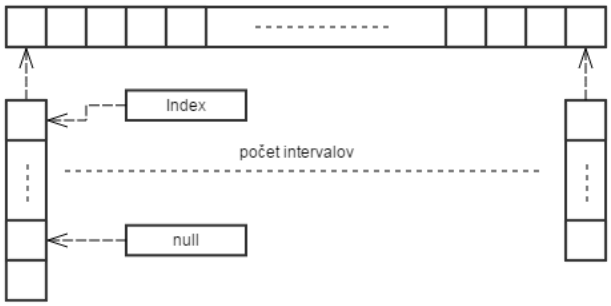
Pri hľadaní intervalov sa najskôr hľadá začiatok intervalu a to tak, že pokiaľ sa nenájde vrchol s rovnakou hodnotou ako je hodnota `from`, tak postupne prejdeme až do listu. Z tohoto listu sa musíme posunúť o jeden vrchol doprava, kde je vrchol s prvou hodnotou v intervale. Takáto situácia môže vzniknúť v prípade, ak by sme sa pýtali na všetky indexy na intervale <50,60>, ale vrcholy majú hodnoty {48,49,55,56,57,59,60}.
Po nájdení prvého prvku v intervale postupujeme pomocou parametru next cez všetky vrcholy, ktoré sa nachádzajú v hľadanom intervale a ukladáme si ich do zoznamu, pokiaľ nenarazíme na prvý vrchol, ktorý nie je z hľadaného intervalu `<from; to>`, alebo parameter next neobsahuje ďalšiu pozíciu.

3.3 Vyhľadávanie za pomoci intervalov

Predchádzajúce dve dátové štruktúry využívajú binárne vyhľadávanie. Skúsme sa teraz zamyslieť, či sa nedá nejakým spôsobom zmenšiť počet prvkov potrebných pre porovnanie. Jednou z možností by bolo pokúsiť sa čo najrýchlejšie približovať k hľadanej pozícii. Toto môžeme spraviť tak, že si určíme, koľkým pozíciám na chromozóme prislúcha jedna pozícia v poli. To vieme určiť vzťahom $k = maxPos/countIndex$ (maxPos je najväčšia pozícia na

chromozóme a countIndex určuje počet indexov na chromozóme). A približné určenie bunky v poli potom bude vyplývať zo vzťahu $k * (pos - firstPos)$ (pos je hodnota hľadanej pozície a firstPos je prvá pozícia na chromozóme, ktorá obsahuje alelu). Takýmto spôsobom vieme veľmi rýchlo skočiť do okolia hľadaného indexu. Toto platí ešte pre niekoľko iterácií, ale neskôr už nekonvergujeme tak rýchlo, dokonca sa môže stať, že budeme potrebovať až h krokov, kde h je počet pozícií medzi lokusmi na nájdenie správnej pozície. Tiež tento postup sa nedá použiť pre Avl stromy. Takže zdanlivo dobrý spôsob nie je vhodný pre naše dáta.
Skúsme teraz popremýšľať, či by sa nedali využiť lokusy, ktoré nám bránili v predošlej optimalizácii.

Nebudeme si pamätať indexy v poli alebo vo vrchole, ale celistvý interval indexov (lokus). Vyhľadávanie konkrétnej pozície potom bude vyzerať tak, že namiesto podmienky väčší, menší, sa budeme pýtať: menší ako najmenšia hodnota v lokuse, alebo väčší ako najväčšia hodnota v lokuse, ak obidve podmienky budú vyhodnotené záporne vieme, že hľadaná pozícia sa nachádza v intervale na súčasnom vrchole/bunke poľa. Teraz nám stačí pozrieť sa do poľa v čase *O(1)* na pozíciu, na ktorej by sa mal nachádzať hľadaný variant. Toto vieme spraviť, keďže poznáme pozíciu na chromozóme prvého prvku v poli, označme si ju ako `S` a po overení pozície `pos-S` v poli určíme, či sa na pos nachádza nejaký záznam. Ukážka takto vytvorenej dátovej štruktúry pre binárne vyhľadávanie je na obrázku 3.4. Obdobným spôsobom sme implemetovali riešenie za použitia samovyvažovacích stromov. Tieto implementácie sme nazvali Binárne vyhľadávanie s intervalmi a Avl strom s intervalmi.

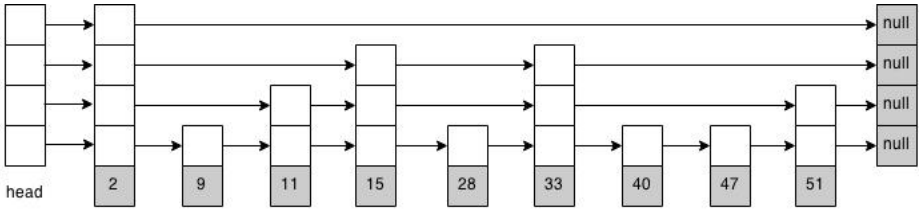


Obr. 3.4: Utriedené pole obsahujúce polia indexov

3.4 Skip list

Je dátová štruktúra umožňujúca vyhľadávanie v utriedenej postupnosti, v časovej zložitosti *O(log(n))*. Zabezpečí sa to tak, že sa vytvorí viacúrovňový spájaný zoznam (3.5), ktorý umožňuje preskakovať (skip) niektoré hodnoty, a tým urýchliť nájdenie hľadaného prvku. Vytvorenie skiplistu [12] pre vopred neznáme hodnoty sa vykonáva tak, že s určitou pravdepodobnosťou sa vytvárajú jedno, dvoj, troj, ... úrovňové uzly, kde pravdepodobnosť klesá z narastajúcou výškou uzla. Vloženie prvku teda začína vygenerovaním uzla náhodnej výšky, ktorý sa potom vloží na pozíciu tak, aby platilo, že spodná úroveň spájaného zoznamu je utriedená. Nájdenie takejto pozície je opísané v getAttribute.

Keďže dopredu poznáme všetky hodnoty, vytvoríme spájaný zoznam pre všetky prvky. V ďalšej iterácii vytvoríme opätovne spájaný zoznam, ale budeme doň vkladať každý druhý prvok z predošlého zoznamu. Súčasne si vytvoríme linku z nižšej úrovne pre každý prvok v zozname. Takýmto spôsobom budeme postupovať až kým nám nezostane len jeden prvok.



Obr. 3.5: Ukážka skiplistu

3.4.1 getAttributes

Vyhľadávanie začína v (Head) prvku, ktorý je prvý v horizontálnom aj vertikálnom zozname. Pri hľadaní prvku sa kontroluje náš ďalší smer dvomi podmienkami. Ak hľadaná hodnota je väčšia ako ďalšia hodnota v zozname (horizontálny smer), choď na ňu a proces opakuj. V opačnom prípade choď na nižšiu úroveň. Tento proces opakuj pokiaľ sa nepríde na koniec zoznamu alebo sa nenájde hľadaná hodnota.

3.4.2 getInterval

Podobným spôsobom ako v metóde getAttributes sa nájde začiatočný index. Po jeho nájdení potrebujeme preiterovať cez hodnoty v spájanom zozname, na najnižšej úrovni, pokiaľ neprekročíme hodnotu konca intervalu alebo nezájdeme na koniec zoznamu.

3.5 Metóda konštantného okna

Doteraz sme spomínali prístupy, ktoré zabezpečili nájdenie prvku v logaritmickom čase. Myšlienka tejto stratégie pozostáva z nájdenia riešenia s konštantnou časovou zložitosťou vyhľadávania.

Na chromozóm sa budeme teraz pozerať ako na pole začínajúce na najmenšej pozícii, na ktorej sa nachádza nejaký záznam s. Pole si rozdelíme na

okná konštantnej dĺžky w, ktoré budú uložené v poli H. Na pozícii $H[i]$ sa teda nachádza pole G, ktoré obsahuje indexy s pozíciami na chromozóme $<i \cdot w+s;(i+1) \cdot w+s)$. Výhodou takejto reprezentácie chromozómu je možnosť vynechania polí G, v ktorých nie sú uložené žiadne indexy, a tým dovoľuje zmenšiť počet prvkov poľa H. Takýmto spôsobom vieme značne znížiť pamäťové nároky. Podstatnou úlohu zostáva nájsť optimálnu veľkosť okna w.

Obr. 3.6: Metóda optimálneho okna.

3.5.1 Hľadanie optimálneho rozdelenia

Pri implementácii je jedným z najväčších problémov získanie najoptimálnejšej veľkosti okna. Môžeme postupne skúšať všetky hodnoty veľkosti, na ktorú budeme deliť chromozóm, počet možností na otestovanie je ale príliš veľký. Môžeme ale využiť, že pozície, ktoré potrebujeme otestovať, sa nachádzajú na intervaloch oddelených skokmi. Dĺžky lokusov sú pomerne z malého rozsahu, kde mediálna hodnota sa pohybuje okolo čísla 129, označme si ju *mil*. Síce skoky majú hodnoty z väčšieho rozsahu, ukážeme, že tie nám až tak nevadia,

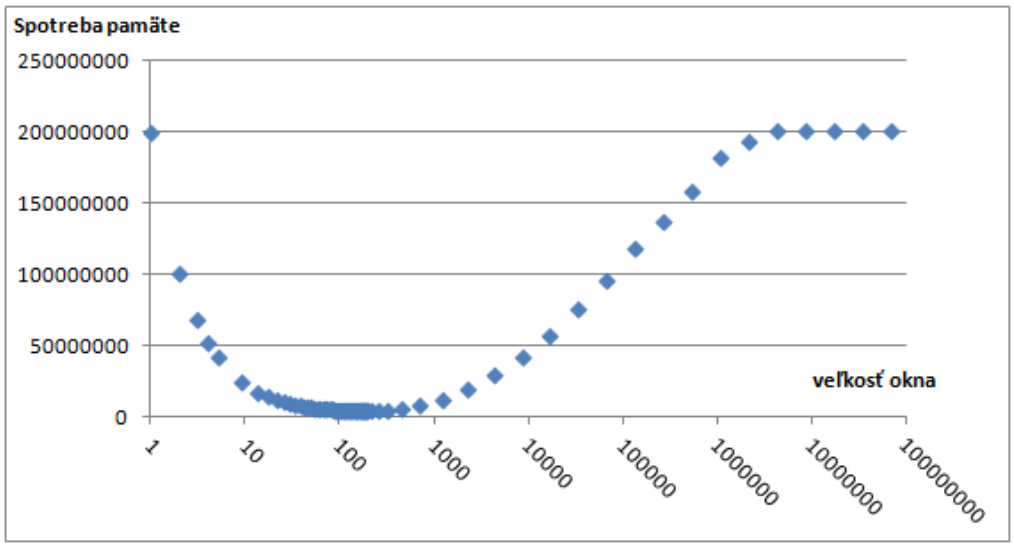
podobne medián týchto hodnôt označme *mjl*. Uvažujme, že optimálne rozdelenie sa nachádza v intervalu$<mil; mjl>$. Ak budeme deliť chromozóm podľa hodnôt nachádzajúce sa v tomto intervale budeme pozorovať dve udalosti
1. Niektoré polia G budú obsahovať priveľa null hodnôt.
2. Niektoré zase budú musieť byť rozdelené na viacero polí G, na ktoré budeme musieť uchovávať referencie v poli H, ktoré sa tým pádom zväčší.

Z vlastnosti mediánu (veľkosti intervalov) vieme, že počet lokusov, ktoré ovplyvňujú využitie pamäte pre prípad 1. je rovnaký počtu lokusov pre prípad 2. . Rovnakú úvahu môžeme použiť aj pre veľkosti medzier medzi lokusmi. Tiež si môžeme uvedomiť, že viac pamäte potrebujeme použiť vtedy, ak budeme používať väčšie okno, lebo sa spotrebuje násobne viac pamäte na uchovanie null hodnôt v poliach G, ako keď ich budeme deliť na menšie polia, kedy je potrebné uchovávať v menšom množstve null hodnoty v poli H. Tým chceme povedať, že optimálna veľkosť okna je teda bližšie k hodnote Medián(il). Môžeme si to všimnúť aj na grafe závislosti (3.7). A teda na nájdenie optimálnej veľkosti postačuje prejsť hodnotami nachádzajúcimi sa medzi mediánom dĺžky lokusov a mediánom dĺžky skokov.

Vďaka spravnej veľksti okna a vlastnosti distribúcie variantov na chromozóme sa pamäťové nároky a časová zložitosť vytvorenia dátovej štruktúry zredukuje z O(g) (g - veľkosť genomu) na hodnotu O($\lceil g/w \rceil$ + n) (n - počet indexov)(3.8).

3.5.2 getAttribute

Nájdenie prvku uskutočnené v konštantnom čase. Hľadanú pozíciu budeme označovať *P*. Najprv sa pozrieme do poľa *H* na pozíciu $\lfloor (P\text{-}s)/w \rfloor$. Ak na tejto hodnote je null hodnota, proces ukončíme. Inak sa pozrieme do poľa *G* uloženého na tejto hodnote. V poli *G* vyberieme hodnotu *(P-s)%W*, kde sa nachádza hľadaný index alebo null hodnota, ak sa záznam v databáze nenachádza.



Obr. 3.7: Pamäťová závislosť od veľkosti okna

3.5.3 getInterval

Označme si hodnoty `from` a `to` zadané používateľom ako F a T. Pozrieme na pozíciu $\lfloor (F\text{-}s)/W \rfloor$ v poli H v prípade, ak sa na tejto pozícii nachádza pole G, pozrieme sa na pozíciu $(F\text{-}s)\%W$ a ak by miesto referencie na G bol na pozícii $\lfloor (F\text{-}s)/W \rfloor$ `null`, skúsime sa pozrieť na pozíciu $\lfloor (F\text{-}s)/W)+1 \rfloor$, a ďalej pokiaľ nenájdeme pozíciu j v H, na ktorej sa nachádza pole G, kde pozícia je menšia $\lfloor (T\text{-}s)/W \rfloor$.
Ak už takéto pole G nájdeme, preiterujeme cez všetky prvky i a pozbierame Indexy nachádzajúce sa v postupnosti, pre ktoré platí, $(T\text{-}s) <= (F\text{-}s)*j+i$, čiže prechádzame ako aj cez prvky v poli G tak aj cez prvky v poli H.
V prípade ak sa pole G nachádza na pozícii $H\lfloor (F\text{-}s)/W \rfloor$, a na hodnote $G[(F\text{-}s)\%W]$je `null` hodnota, tak preiterujeme cez všetky nasledujúce prvky poľa G, pokiaľ nenájdeme prvý nenulový alebo nezájdeme na koniec poľa. Ak by sme zašli až na koniec poľa a nenašli žiaden prvok, postupujeme rovnako ako v prípade, že pole G na pozícii $H \lfloor (F\text{-}s)/W \rfloor$ nebolo.

Vylepšením tohto prístupu by bol interval tree, ktorý by nám teraz už v čase log(n) našiel počiatok intervalu a v čase O(1) ostatné prvky. Toto riešenie je síce časovo efektívnejšie ako to, pre ktoré sme sa nakoniec rozhodli. Jeho nevýhodou je ale pamäťová náročnosť.

3.6 Sql riešenie

Ďalším zo skúmaných riešení je kopírovanie celej TSV tabuľky do relačnej databázy. Zvolili sme databázový systém PostgreSQL [22], kde sme vytvorili tabuľku, podľa pomenovacích štandardov [27], ktorá obsahovala atribúty ako TSV súbory, a spustili sme načítavanie. Problémom tohto riešenia bola viac ako dvojnásobná spotreba miesta na disku, ale najmä čas, ktorý bol potrebný na naplnenie tabuľky, ktorý sa pohyboval okolo 24 hodín. Toto riešenie nespĺňalo tretiu normálnu formu, keďže u niektorých atribútov platí, že obsahujú viac hodnôt oddelených bodkočiarkou. Pre riešenie, ktoré spĺňalo 3. normálnu formu bol čas naplnenia predĺžený na 7-8 dní. Takéto časy boli neprijateľné.

Preto sme sa rozhodli pamätať si iba indexy, pre ktoré sme si vytvorili SQL tabuľku. Na nej sme vytvorili SQL index pre chromozóm a pozíciu na ňom. Prepojenie s Java projektom je realizované pomocou JDBC konektoru a indexy sa získavajú jednoduchým výberom a mapovaním na objekt Index.

```
SELECT * FROM nsfp WHERE chr='c' AND pos=p
SELECT * FROM nsfp WHERE chr='c' AND pos=>from AND pos<=to
```

3.7 MongoDB

Jednou z možností, ktorú sme zvažovali, bolo použitie Mongo databázy [4]. Ide o nerelačnú databázu vhodnú pre ukladanie vopred neznámych objektov. Umožňuje vytvorenie indexov pre atribúty záznamov, kde sa dá nastaviť priorita, podľa ktorej sa indexy používajú. Nevýhodou tohto riešenia je veľká pamäťová náročnosť. Pri našej implementácii sme ukladali do MongoDB len indexy, kde pre uloženie prvého chromozómu bolo predpokladaných 1,5 GB RAM pamäte. Táto hodnota prevyšovala hodnoty testovacieho hardvéru a preto z tohto dôvodu sa nám ani nepodarilo vložiť celý 1. chromozóm do databázy. A rovnako z tohto dôvodu sme nemohli uskutočniť testy na časovú náročnosť tohto riešenia.

3.8 Časová a pamäťová zložitosť

Označenie:
n - počet indexov
m - počet intervalov
g - veľkosť genómu

Stratégia	Vytvorenie	Vyhľadávanie	Pamäťová zložitosť
Binárne vyhľadávanie	O(n)	O(log(n))	O(n)
Bin. vyh. s intervalmi	O(n+m)	O(log(m))	O(n+m)
Avl strom	O(nlog(n))	O(log(n))	O(n)
Avl strom s intervalmi	O(n+mlog(m))	O(log(m))	O(n+m)
Skip list	O(nlog(n))	O(log(n))	O(nlog(n))
Konštantné okno	O(g)	O(1)	O(g)

Tabuľka 3.1: Časová a pamäťová náročnosť.

Kapitola 4

Porovnanie

Požiadavky, ktoré sme si kládli na začiatku celého projektu boli zamerané najmä na časovú efektívnosť a pamäťové nároky. Vytvorili sme sadu testov, ktorými sme testovali tieto kritériá.

4.1 Časová efektívnosť

Pre objektívne posúdenie stratégií sme test opakovali desaťkrát a za ohodnotenie stratégie sme považovali medián výsledného času týchto testov. Skúmali sme aj okrajové prípady stratégií. Vytvorili sme dva testovacie súbory, jeden obsahujúci dopyty na pozície, ktoré obsahujú záznamy, a druhý, ktorý ich neobsahuje. Takto sme testovali obidve metódy getAttributes aj getInterval. Rovnako sme testovali aj prípad, že k službe pristupuje viacero používateľov naraz, čo bolo simulované viacerými vláknami. V takom prípade sa výsledné časy testov pomerovo približne zhodovali s jedným používateľom. Boli zhruba tretinové vzhľadom na tri vlákna, na ktorých sme testy uskutočnili. Tretinové časy boli viditeľné, ak sa používali metódy bez načítania hodnôt z disku. Ako sa výsledné časy ovplyvnili si môžeme všimnúť na teste metódy getAttributes, kde sme testovali aj prípad čítania zo súborov TSV a aj bez neho (4.1,4.2).

Stratégia	Jeden používateľ[ms]	Viacerí používatelia[ms]
Binárne vyhľadávanie	2891,5	1501,0
Bin. vyh. s intervalmi	2844,0	1505,5
Avl strom	2711,5	1455,5
Avl strom s intervalmi	2724,0	1431,0
Skip list	2853,5	1491,0
Sql s indexmi	16847,5	10002,5
Konštantné okno	2807,5	1511,0

Tabuľka 4.1: getAttributes 10 000 dopytov na pozície, ktoré obsahujú záznamy s čítaním z disku.

Stratégia	Jeden používateľ[ms]	Viacerí používatelia[ms]
Binárne vyhľadávanie	43,12	15,83
Bin. vyh. s intervalmi	16,22	5,21
Avl strom	35,05	13,77
Avl strom s intervalmi	12,56	5,17
Skip list	56,94	23,82
Sql s indexmi	1019,65	974,95
Konštantné okno	5,91	1,69

Tabuľka 4.2: getAttributes 10 000 dopytov na pozície, ktoré obsahujú záznamy bez čítania z disku.

U metódy getAttributes sme sa zamerali aj na porovnanie časovej náročnosti pre vyhľadanie prvkov, ktoré sa v dbNSFP nenachádzali. V tomto teste sa najviac odzrkadlil logaritmický faktor, ktorý bol u väčšiny riešení, okdrem metódy konštantného okna(4.3).

Stratégia	Jeden používateľ[ms]	Viacerí používatelia[ms]
Binárne vyhľadávanie	31,03	12,42
Bin. vyh. s intervalmi	9,10	3,38
Avl strom	25,11	9,04
Avl strom s intervalmi	12,05	4,21
Skip list	41,30	15,25
Sql s indexmi	9860,00	9700,00
Konštantné okno	4,49	1,86

Tabuľka 4.3: getAttributes 10 000 dopytov na pozície, ktoré neobsahujú záznamy bez čítania z disku.

Pri testovaní metódy getInterval u jednotlivých stratégií sme test vykonali bez čítania z disku, aby sme minimalizovali čas potrebný na načítanie záznamov (4.4, 4.5).

Stratégia	Jeden používateľ[ms]	Viacerí používatelia[ms]
Binárne vyhľadávanie	1406	955,5
Bin. vyh. s intervalmi	1380,5	920,5
Avl strom	1385,5	930,5
Avl strom s intervalmi	1367	938
Skip list	1432	955,5
Sql s indexami	170850	87460
Konštantné okno	1395,5	923

Tabuľka 4.4: getInterval, intervaly, ktoré obsahujú súvisle úseky s indexami, 10 000 dotazov.

Stratégia	Jeden používateľ[ms]	Viacerí používatelia[ms]
Binárne vyhľadávanie	1263,5	882,5
Bin. vyh. s intervalmi	1229	870,5
Avl strom	1243,5	880,5
Avl strom s intervalmi	1231,5	895,5
Skip list	1281,5	901
Sql s indexmi	187806	88530
Konštantné okno	1268,5	893

Tabuľka 4.5: getInterval, intervaly, ktoré neobsahujú žiadne indexy, 10 000 dotazov.

4.2 Pamäťové nároky

Fyzická pamäť je dostupná pre všetky procesy. Jej správa je riadená operačným systémom pomocou virtuálnej pamäte, ktorá mapuje virtuálne adresy na fyzickú pamäť. Pri vytvorení Objektu sa naalokuje miesto na heape, na ktoré si program uchováva referenciu. Ak už nemáme referenciu na niektoré naalokované miesto, tak garbage collector môže objekt zmazať, a miesto, ktoré mu patrilo, v prípade potreby ponúknuť pre vytvorenie iného Objektu. Ak chceme vedieť aktuálny stav heapu, vytvoríme si takzvaný heap dump, čo predstavuje obraz aktuálneho stavu java virtuálnej pamäte spusteného procesu so všetkými referenciami, objektami, ako aj informáciou o počte jednotlivých objektov a využitím pamäte. Vytvorenie memory dumpu vykonáme pomocou ManagementFactory, MBeanServer a HotSpotDiagnosticMXBean a uložíme si ho do súboru. Pre vizualizáciu údajov obsiahnutých vo vygenerovanom súbore využijeme voľne dostupnú aplikáciu MemoryAanalyser [8], a pomocou nej si môžeme všimnúť, že počet objektov typu Index je vo všetkých stratégiách rovnaký. Líši sa v objektoch, ktoré sú použité v spomínaných dátových štruktúrach. Čiže pre vyhodnotenie využitia pamäte budeme porovnávať obsah celého obrazu pamäte po načítaní všetkých chromozómov. Výsledky, ktoré sme takýmto spôsobom namerali sú v tabuľke 4.6.

Stratégia	1. chr[kB]
Binárne vyhľadávanie	283 081
Binárne vyhľadávanie s intervalmi	280 089
Avl strom	280 583
Avl strom s intervalmi	381 275
Skip list	464 888
Konštantné okno	314 753
Sql_with_indexes	2 713
Mongo_db	1 500 000

Tabuľka 4.6: Využitie pamäte

4.3 Vyhodnotenie

K najväčším časovým rozdielom prichádza pri vyhľadaní prvku ktorý sa v databáze nenachádza, môžeme si to všimnúť aj na výsledkoch testovania. Ide hlavne o implementácie dátových štruktúr, ktoré potrebujú skontrolovať log(n) možnosti na overenie existencie záznamu. Pre takýto prípad má najlepšie vlastnosti metóda konštantného okna. Aj keď pre ostatné testy metóda konštantného okna nezískala najlepšie hodnotenie, ale rozdiel medzi najlepšou implementáciou pre daný test bol zanedbateľný. Z toho dôvodu sme sa rozhodli použiť ako zdroj pre Variant Annotation Service práve stratégiu Konštantného okna.

Záver

Cieľom tejto bakalárskej práce bolo poskytnúť efektívne riešenie výberov z databázy dbNSFP, ktorá je nosným zdrojom atribútov pre webovú službu Variant Annotation Service a desktopovú aplikáciu Variant Annotation Analyser. Oba nástroje budú prístupné pre akademické účely a očakáva sa využitie na globálnej úrovni.

V práci sme opísali dáta obsiahnuté v tejto databáze a tiež sme uviedli ich vlastnosti. Tieš sme vytvorili objektový model implementácie a zadefinovali sme si interface, cez ktorý sa výsledná stratégia priamo napojila na webovú službu VAS.

Popísali sme podrobne implementačné detaily stratégií, ktoré boli testované. Poukázali sme na veľké pamäťové nároky pre riešenie s MongoDB a časovo náročné načítanie údajov do PostgreSQL databázy. V poslednej kapitole sme určili spôsob testovania časovej efektivity a pamäťových nárokov stratégií. Z výsledkov testov sme určili najvhodnejšiu dátovú štruktúru, ktorou sa stala metóda konštantného okna.

Napriek optimalizovanému vyhľadaniu prvku v dátovej štruktúre, výsledný čas potrebný na výber záznamu z dbNSFP je z veľkej miery závislý na rýchlosti čítania z disku. Pri ďalšej optimalizácii by sme mohli urýchliť čítanie z disku udržiavaním si v pamäti niektorých alebo všetkých atribútov, vďaka čomu by sa zredukoval celkový čas výberu záznamu z databázy.

Literatúra

[1] Ivan Adzhubei, Daniel M Jordan, and Shamil R Sunyaev. Predicting functional effect of human missense mutations using polyphen-2. *Current protocols in human genetics*, pages 7–20, 2013.

[2] Rolf Apweiler, Terri K Attwood, Amos Bairoch, Alex Bateman, Ewan Birney, Margaret Biswas, Philipp Bucher, Lorenzo Cerutti, Florence Corpet, Michael D. R. Croning, et al. The interpro database, an integrated documentation resource for protein families, domains and functional sites. *Nucleic acids research*, 29(1):37–40, 2001.

[3] Phil Blunsom. Hidden markov models. *Lecture notes, August*, 15:18–19, 2004.

[4] Kristina Chodorow. *MongoDB: the definitive guide*. Ö'Reilly Media, Inc.", 2013.

[5] UniProt Consortium et al. The universal protein resource (uniprot). *Nucleic acids research*, 36(suppl 1):D190–D195, 2008.

[6] Eugene V Davydov, David L Goode, Marina Sirota, Gregory M Cooper, Arend Sidow, and Serafim Batzoglou. Identifying a high fraction of the human genome to be under selective constraint using gerp++. *PLoS computational biology*, 6(12):e1001025, 2010.

[7] Wlodzimierz Dobosiewicz. Optimal binary search trees. *International journal of computer mathematics*, 19(2):135–151, 1986.

[8] Eclipse. Memory analyzer (mat), 2015. [Citované 2015-5-26] Dostupné z `https://eclipse.org/mat/`.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[10] et al. Garber, Manuel. Identifying novel constrained elements by exploiting biased substitution patterns. *Bioinformatics*, 25.12.2009.

[11] Lars Grunske and Dipl Inf André van Hoorn. Evaluation von java profiler werkzeugen. 2013.

[12] Eric N. Hanson and Theodore Johnson. The interval skip list: A data structure for finding all intervals that overlap a point. In *In Proc. of the 2nd Workshop on Algorithms and Data Structures*, pages 153–164. Springer, 1992.

[13] David N. Cooper Ian N. M. Day Hashem A. Shihab, Julian Gough and Tom R. Gaunt. Predicting the functional, molecular, and phenotypic consequences of amino acid substitutions using hidden markov models. *Genome Res*, 15:1034–1050, 2005.

[14] Robert W Irving and Lorna Love. The suffix binary search tree and suffix avl tree. *Journal of Discrete Algorithms*, 1(5):387–408, 2003.

[15] jantipin. mutationassessor, 2015. [Citované 2015-9-3] Dostupné z `http://mutationassessor.org/`.

[16] Johns Hopkins University. Karchin lab johns hopkins university vest, 2015. [Citované 2015-9-3] Dostupné z `http://karchinlab.org/apps/appVest.html`.

[17] Riley GR Jang W Rubinstein WS Church DM Maglott DR Landrum MJ, Lee JM. Clinvar: public archive of relationships among sequence variation and human phenotype. *Nucleic Acids Res*, (42), 2014.

[18] Xiaoming Liu, Xueqiu Jian, and Eric Boerwinkle. dbnsfp: a lightweight database of human nonsynonymous snps and their functional predictions. *Human mutation*, 32(8):894–899, 2011.

[19] Xiaoming Liu, Xueqiu Jian, and Eric Boerwinkle. dbnsfp v2. 0: A database of human non-synonymous snvs and their functional predictions and annotations. *Human mutation*, 34(9):E2393–E2402, 2013.

[20] Matti. Work with tab-delimited tsv format, 2015. [Citované 2015-3-3] Dostupné z `https://support.google.com/docs/answer/63377/`.

[21] Mikhail V Matz and Rasmus Nielsen. A likelihood ratio test for species membership based on dna sequence data. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 360(1462):1969–1974, 2005.

[22] Bruce Momjian. *PostgreSQL: introduction and concepts*, volume 192. Addison-Wesley New York, 2001.

[23] Pauline C Ng and Steven Henikoff. Sift: Predicting amino acid changes that affect protein function. *Nucleic acids research*, 31(13):3812–3814, 2003.

[24] NHLBI. Nhlbi grand opportunity exome sequencing project (esp), 2015. [Citované 2015-9-3] Dostupné z `https://esp.gs.washington.edu/drupal/l`.

[25] Sue Povey, Ruth Lovering, Elspeth Bruford, Mathew Wright, Michael Lush, and Hester Wain. The hugo gene nomenclature committee (hgnc). *Human genetics*, 109(6):678–680, 2001.

[26] Jana Marie Schwarz, Christian Rödelsperger, Markus Schuelke, and Dominik Seelow. Mutationtaster evaluates disease-causing potential of sequence alterations. *Nature methods*, 7(8):575–576, 2010.

[27] Sehrope Sarkuni. How i write sql, part 1: Naming conventions, 2014. [Citované 2014-12-3] Dostupné z `https://launchbylunch.com/posts/2014/Feb/16/sql-naming-conventions/`.

[28] Kholodov M Baker J Phan L Smigielski EM Sirotkin K Sherry ST, Ward MH. dbsnp: the ncbi database of genetic variation. *Nucleic Acids Res*, 1(29):308–11, 2001.

[29] Katherine S. Pollard Siepel, Adam and David Haussler. New methods for detecting lineage-specific selection. *Research in Computational Molecular Biology. Springer Berlin Heidelberg*, 2006.

[30] Pedersen JS Hinrichs AS Hou MM Rosenbloom K Clawson H Spieth J Hillier LW Richards S et al. Siepel A, Bejerano G. Evolutionarily conserved elements in vertebrate, insect, worm, and yeast genomes. *Human mutation*, 2013.

[31] Nayanah Siva. 1000 genomes project. *Nature biotechnology*, 26(3):256–256, 2008.

[32] D SNUSTAD. a michael j simmons. genetika. překlad jiřina relichová. brno: Masarykova univerzita, 2009, xxi, 871 s. Technical report, ISBN 978-802-1048-522.

[33] Chris Stark, Bobby-Joe Breitkreutz, Teresa Reguly, Lorrie Boucher, Ashton Breitkreutz, and Mike Tyers. Biogrid: a general repository for interaction datasets. *Nucleic acids research*, 34(suppl 1):D535–D539, 2006.

[34] University of Washington. Combined annotation dependent depletion (cadd), 2015. [Citované 2015-9-3] Dostupné z `http://cadd.gs.washington.edu/`.

[35] Kai Wang, Mingyao Li, and Hakon Hakonarson. Annovar: functional annotation of genetic variants from high-throughput sequencing data. *Nucleic acids research*, 38(16):e164–e164, 2010.

[36] Danielle Welter, Jacqueline MacArthur, Joannella Morales, Tony Burdett, Peggy Hall, Heather Junkins, Alan Klemm, Paul Flicek, Teri Manolio, Lucia Hindorff, et al. The nhgri gwas catalog, a curated resource of snp-trait associations. *Nucleic acids research*, 42(D1):D1001–D1006, 2014.