Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Vizualizácia evolučných histórií

Bakalárska práca

2016
Dávid Simeunovič

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Vizualizácia evolučných histórií

Bakalárska práca

Študijný program:	Informatika
Študijný odbor:	2508 Informatika
Školiace pracovisko:	Katedra informatiky
Školiteľ:	doc. Mgr. Bronislava Brejová, PhD.

Bratislava, 2016
Dávid Simeunovič

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

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

Názov: Vizualizácia evolučných histórií
Visualization of evolutionary histories

Cieľ: Počas evolúcie dochádza v DNA k lokálnym mutáciám, ktoré menia jeden alebo niekoľko susedných nukleotidov, ale aj k väčším zmenám, ktoré menia poradie alebo počet výskytov dlhších oblastí. Cieľom práce je implementovať systém na vizualizáciu evolučnej histórie jednej alebo viacerých DNA sekvencií s dôrazom na tieto väčšie zmeny. Samotná história je daná na vstupe a cieľom je zobraziť ju tak, aby sa dali prehľadne sledovať jednotlivé mutácie a tiež vzťahy rôznych častí sekvencie.

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

Dátum zadania: 28.10.2014

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

.....
študent

.....
vedúci práce

Poďakovanie:

Abstrakt

Táto práca sa venuje zobrazovaniu evolučných histórií. Výsledkom tejto práce je program *EHDraw*, ktorý umožňuje vytvárať vizualizácie, meniť nastavenia použité pre tvorbu týchto vizualizácií a automaticky zredukovať počet génov, ktoré sa nachádzajú vo vizualizácii, bez straty pre nás podstatnej informácie - aké mutácie sa odohrali v evolučnej histórii. Redukciu počtu génov transformujeme na Problém množinového pokrytia, jeho aproximáciu získame pomocou greedy algoritmu, a jeho optimum pomocou Celočíselného lineárneho programovania. Rozdiely týchto dvoch prístupov porovnávame testami.

Kľúčové slová: vizualizácia, evolučná história, výber génov, Problém množinového pokrytia, greedy algoritmus, Celočíselné lineárne programovanie

Abstract

The subject of this work is visualisation of evolution histories. Result of this work is program *EHDraw*, which allows creating visualisations. changing settings used for creation of those visualisations and automatically reduce number of genes contained in visualisation, without loss of relevant information for us - which mutations occurred in evolution history. Reduction of the number of genes is transformed onto Set Cover Problem, his approximation is obtained by greedy algorithm and his optimum by Integer Linear Programming The differences of these two approaches are compared by tests.

Keywords: visualisation, evolution history, gene selection, Set Cover Problem, greedy algorithm, Integer Linear Programming

Obsah

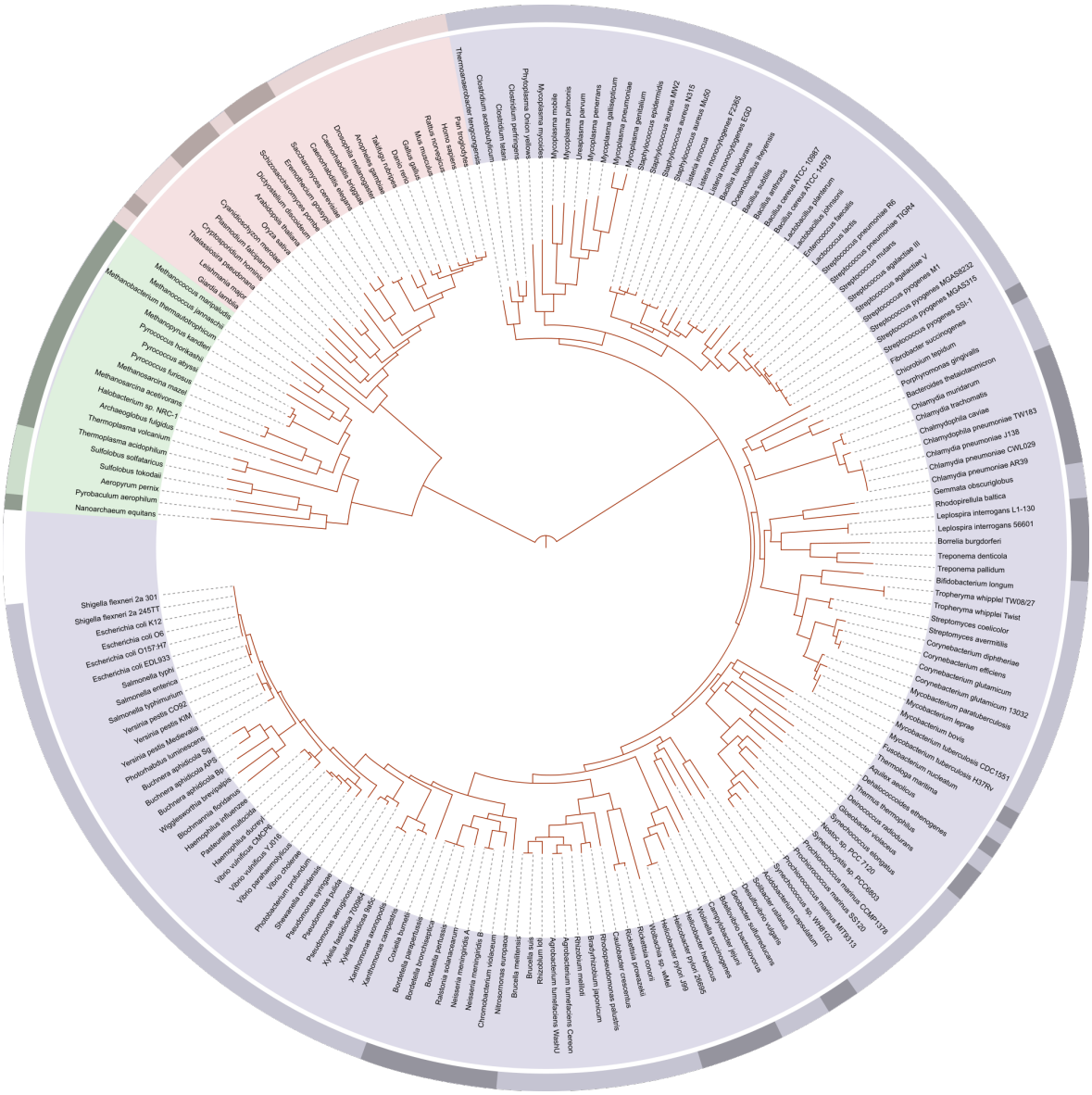
Zoznam obrázkov

Úvod

Už v antike sa niektorí antickí filozofi zaoberali myšlienkou, že základná charakteristika jednotlivých biologických druhov sa časom mení. A práve v gréčtine môžeme nájsť pôvod slova *Fylogenéza*, kde fylé = kmeň a genesis = zrodenie/pôvod. Fylogenéza sa zaoberá vývojom druhu organizmov. Väčšinou sa odohráva počas veľmi dlhého časového úseku, a preto ju nie sme schopní priamo pozorovať a nezostáva nám iná možnosť ako vytvoriť rekonštrukciu na základe poznatkov o evolúcii. Neskôr v tomto vednom poli urobil významný pokrok Charles Darwin, keď v roku 1859 publikoval svoju knihu *Pôvod druhov*. Okrem toho, že z evolúcie vytvoril široko uznávanú teóriu, predstavil aj myšlienku že akékoľvek dva veľmi rozdielne druhy zdieľajú spoločného predka a vizuálne ju znázornil vo forme stromového grafu, takzvaného *stromu života*. Tak položil základy Evolučnej teórie, ktorá skúma evolučné procesy, ktoré na zemi vytvorili rôznorodosť života z počiatočnej živej formy. Ďalšie poznatky v oblasti genetiky, súvisejúce s DNA a RNA viedli k tomu, že na evolúciu sa dnes pozeráme hlavne prostredníctvom génov. Sekvenovanie DNA umožnilo vzťahy medzi jednotlivými organizmami odsledovať na základe zmien, ktoré prebehli v ich DNA sekvencii. To nám ponúka množstvo presných dát využiteľných pri rekonštrukcii fylogenetického procesu. Aj na našej fakulte vzniklo niekoľko prác, ktoré sa venujú rekonštrukcii DNA sekvencie pokiaľ poznáme jej súčasný vzhľad, prípadne sa snažia zrekonštruovať fylogenetický strom pokiaľ poznáme DNA sekvencie súčasných druhov

[9, 6, 5, 12].

Strom stále patrí medzi najpopulárnejšie spôsoby ako zobraziť evolučné vzťahy medzi druhmi alebo inými objektami. Najčastejšie sa stretávame s fylogenetickým stromom, ktorý zobrazuje vývoj druhov z posledného spoločného predka, ako napríklad vidíme na obrázku 1. V takomto zobrazení nevidno samotné zmeny DNA, informácie, ktoré nám poskytnú, ako napríklad vzdialenosť dvoch objektov na základe rozdielnosti ich DNA sekvencie, však bývajú použité na zostavenie stromu. Cieľom tejto práce je zostrojiť program, ktorý zobrazí jednoduchú postupnosť sekvencií DNA s dôrazom na zmeny, ktoré sa udiali s génmi v týchto sekvenciách. Výsledok by mal predstavovať malú vetvu fylogenetického stromu, v ktorom prepojenie objektov zobrazí reálne zmeny, ktoré sa odohrali na ich DNA sekvencii. Inšpiráciou pre túto prácu sú vyššie spomenuté práce pochádzajúce z našej fakulty, náš program má byť schopný vizuali-



Obr. 1: Fylogenetický strom zobrazujúci evolučné vzťahy veľkého počtu druhov. Zdroj: wikipedia.org

zovať výsledky, ktoré produkujú a poslúžiť okrem iného ako rýchla optická kontrola správnosti.

Prvá kapitola poskytne úvod do problematiky, predstavíme si základné biologické a bioinformatické pojmy, potrebné pre našu prácu. Druhá kapitola sa bude venovať implementácii nášho programu *EHDraw*,popíšeme ako vyzerá jeho vstup a výstup, aké možnosti interakcie poskytuje používateľovi a ktoré nastavenia v ňom vieme meniť. Okrem základnej funkcionality súvisiacej so zobrazovaním evolučnej histórie, sme sa zamerali na problém, ako zredukovať množstvo zobrazených génov. Naším cieľom je odstrániť niektoré gény tak, aby sa výsledný obrázok stal prehľadnejším, a aby v ňom zostali zachované informácie o udalostiach, ktoré sa nachádzajú v evolučnej histórii.

Tretia kapitola popisuje, ako problém výberu takýchto génov pretransformujeme na *Problém množinového pokrytia*, ako aj dve metódy ktorými budeme *Problém množinového pokrytia* riešiť. Konkrétne sa jedná o greedy algoritmus, ktorý nájde aproximačné riešenie, a Celočíselné lineárne programovanie, ktorým hľadáme optimálne riešenie. Štvrtá kapitola nadväzuje na tretiu, porovnávame v nej, aké výsledky dostaneme v závislosti od toho, akú metódu výberu génov zvolíme, a v ktorých prípadoch dochádza k najväčšej redukcii množstva zobrazených génov.

Kapitola 1

Úvod do problematiky

V tejto kapitole si vysvetlíme základne pojmy z biológie a bioinformatiky potrebné pre túto bakalársku prácu, a popíšeme podobné už existujúce programy.

1.1 DNA, gén a genóm

Deoxyribonukleová kyselina (DNA) je nositeľom genetickej informácie bunky. Má štruktúru dvojzávitnice, skladajúcej sa z dvoch komplementárnych vlákien. Vlákno je tvorené nukleotidmi, ktoré obsahujú jednu zo štyroch báz adenín, guanín, tymín a cytozín. DNA zvykneme zapisovať ako postupnosť týchto báz, kde každú bázu kódujeme jej počiatočným písmenom A,G,T,C. Vzdialenosť dvoch DNA sekvencií vyjadruje, ako veľmi sú rozdielne.

Gén je súvislý úsek DNA, ktorý kóduje tvorbu proteínu. Gén je základnou jednotkou dedičnosti.

Genóm je súbor DNA molekúl v bunke. Jednotlivé molekuly DNA sa väčšinou nachádzajú v štruktúrach nazývaných chromozómy. Napríklad v ľudskej bunke je 46 chromozómov, každý obsahujúci jednu molekulu dna [16].

1.2 Evolučná história

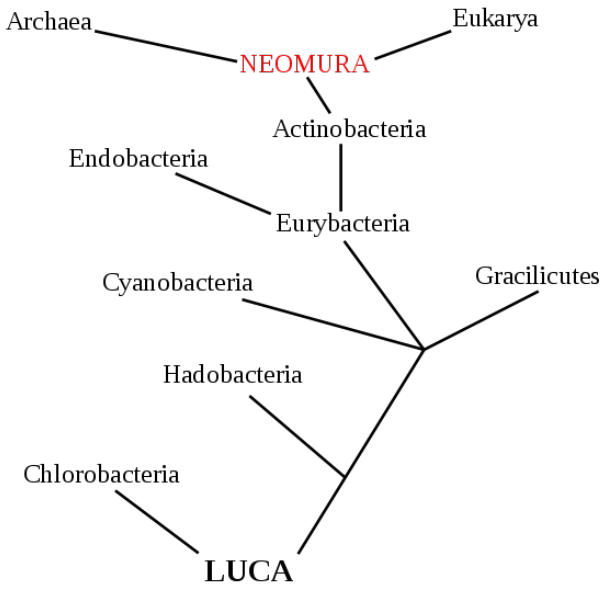
Evolučná história je postupnosť udalostí, ktoré sa odohrali na nejakej DNA sekvencii. V tejto práci budem uvažovať iba udalosti ktoré menia poradie alebo počet génov na chromozóme. Okrem nich sa ale počas evolúcie môžu vyskytnúť aj udalosti, ktoré menia bázy DNA sekvencie, takéto udalosti môžu ovplyvniť funkciu génu, pre našu prácu však nie sú podstatné
Budeme uvažovať hlavne tieto udalosti:

- *Duplikácia* - skopírovanie génu alebo skupiny génov na iné miesto v DNA.

- *Inzercia* - vloženie jedného alebo viacerých nových génov.
- *Delécia* - odstránenie jedného alebo viacerých génov.
- *Inverzia* - zmena poradia a orientácie génu alebo génov.
- *Transpozícia* - zmena poradia génu alebo génov.
- *Speciácia* - špeciálna udalosť, ktorá označuje vznik nového druhu. Vzniká nová vetva v evolučnej histórii.

V našej reprezentácii je evolučná história postupnosť krokov, pričom každý krok pozostáva zo známej sekvencie génov na jednom chromozóme určitého organizmu. Medzi jednotlivými krokmi došlo k jednej alebo viacerým udalostiam.

1.3 Fylogenetický strom



Obr. 1.1: Príklad zakoreneného fylogenetického stromu, Zdroj: wikipedia.org

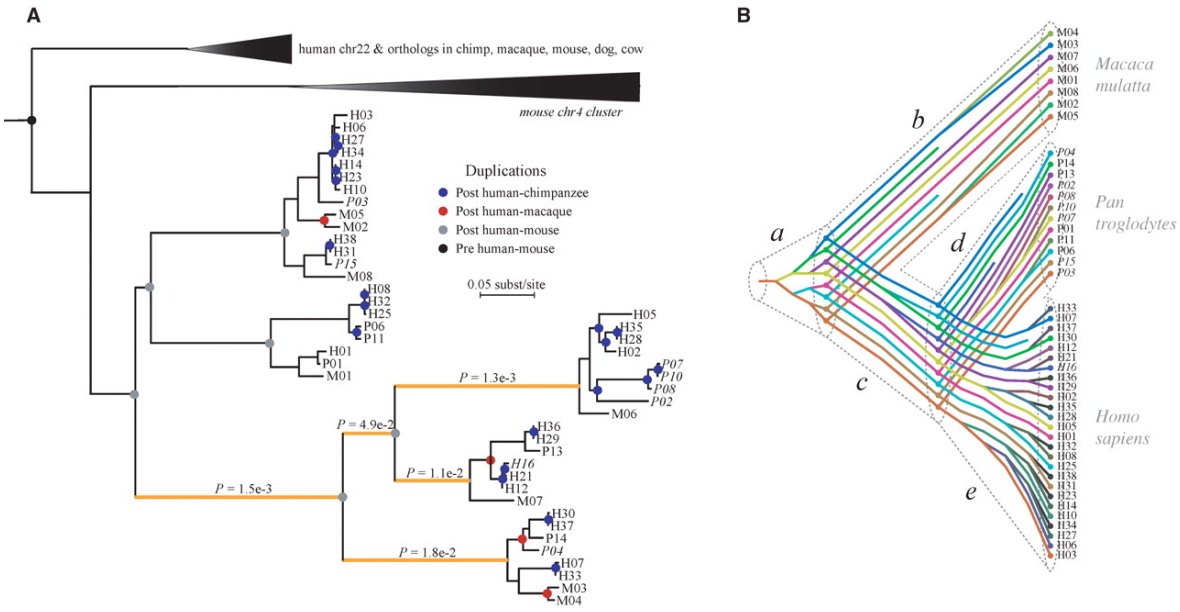
Fylogenetický strom reprezentuje evolučné vzťahy medzi sadou objektov. Pokiaľ si za objekty zvolíme biologické druhy, jedná sa o takzvaný *druhový strom*. Jednotlivé druhy sú pospájané hranami, ktoré reprezentujú evolučný vzťah. Druhy, ktoré sa nachádzajú na *listoch* stromu, sú buď existujúce druhy, z ktorých sa nevyvinuli nové druhy, alebo vyhynuté druhy bez potomkov. Vnútorné vrcholy predstavujú predchodcov, o ktorých sa predpokladá, že sa vyskytli počas evolúcie. Každý vnútorný vrchol zodpovedá speciácii, kde z jedného druhu vznikajú dva nové druhy. Pokiaľ je v strome

známy posledný spoločný predok všetkých listov, nazveme ho *koreň*, a takýto strom označíme ako *zakorenený*. V *zakorenenom* strome je zrejmá orientácia vnútorných hrán, ktorá určuje, ktorý druh sa vyvinul z ktorého. Na obrázku 1.1 vidíme príklad zakoreneného fylogenetického stromu. Prvok *LUCA* predstavuje posledného univerzálneho spoločného predchodcu. Na tomto strome vidíme že Eukaryoty a Archeóny sú od seba fylogeneticky menej vzdialené, ako od Baktérií [14].

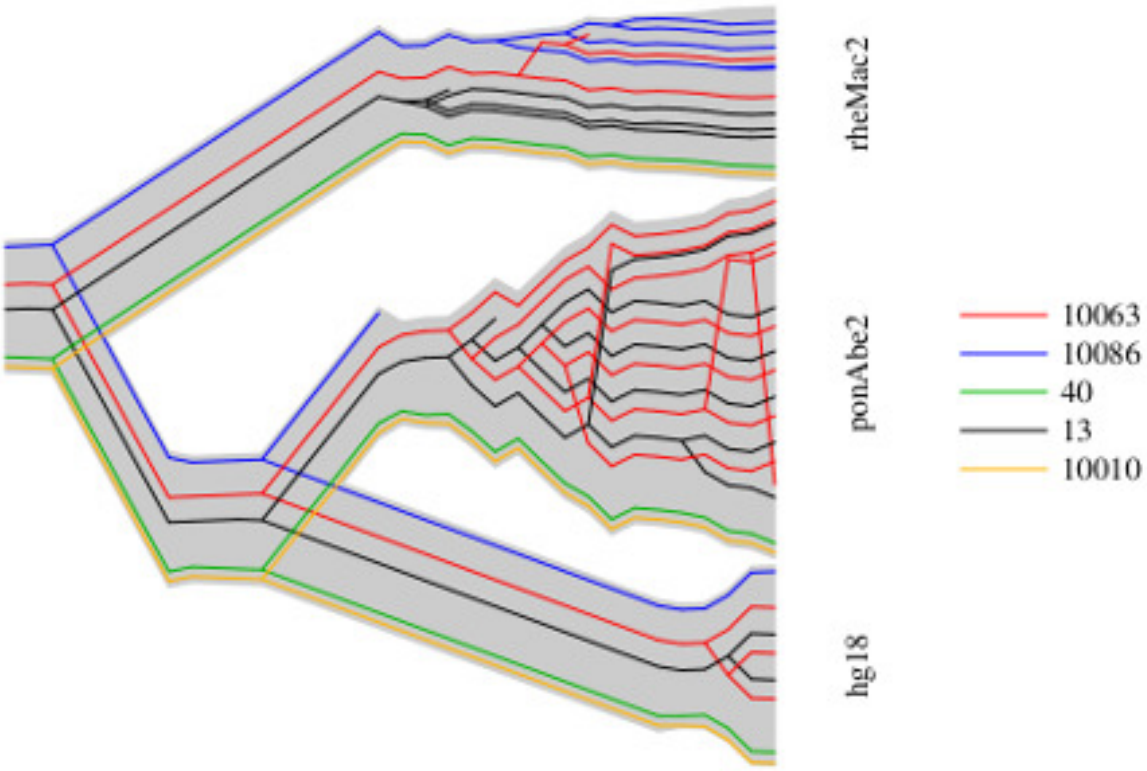
1.4 Vizualizácia evolučných histórií

Vizualizácia je spôsob prevodu dát do grafickej formy, ktorú vieme spracovať naším zrakom, najdominantnejším zmyslom, aký máme. To nám umožňuje okrem lepšieho pochopenia problému aj rýchlu analýzu a odhalenie existujúcich súvislostí a vzorov ktoré sa nachádzajú vo výsledku.

V oblasti vizualizácie evolučných histórií je najčastejšie zobrazovanie fylogenetických stromov. Na tento účel existuje množstvo programov ako napríklad *phylo.io*, *ETE toolkit* alebo *Archaeopteryx* [10, 7, 2]. Poskytujú vizualizáciu fylogenetických stromov v ktorých gény buď vôbec nevystupujú alebo sa nachádzajú iba pri listoch stromu. Rozdiely medzi jednotlivými vrcholmi, sa zvyknú zobrazovať vzdialenosťou týchto vrcholov, alebo číslom, ktoré predstavuje vzdialenosť DNA sekvencií týchto vrcholov. My však cheme, aby sa vo výslednom zobrazení nachádzali jednotlivé udalosti, ktoré sa odohrali počas evolučnej histórie, podobne ako na obrázku 1.2, alebo v článku [4]. Takéto zobrazenie, v ktorom sa nachádzajú viaceré gény naraz, vidíme na obrázku 1.3, zatiaľ však neexistuje používateľsky pohodlný spôsob, ako ho vytvoriť. Náš program by mal umožniť používateľovi, vytvárať podobné obrázky, s pomocou jednoducho ovládateľnej aplikácie.



Obr. 1.2: Zobrazenie evolučnej histórie, v ktorom sledujeme duplikácie pre skupinu génov. V časti **B** vidíme duplikačnú históriu . Zdroj: [3]



Obr. 1.3: Zobrazenie evolučnej histórie viacerých génov naraz,na pozadí sa nachádza druhový strom. Zdroj:[12]

Kapitola 2

Implementácia programu

V tejto kapitole sa budeme venovať niektorým významnejším črtám implementácie nášho programu, ktorý na vstupe dostane súbor popisujúci evolučnú históriu, umožní používateľovi zmeniť niektoré nastavenia a prípadne spustiť optimalizáciu a nakoniec zobrazí grafickú reprezentáciu vstupnej evolučnej histórie podľa toho, aké zmeny vykonal používateľ. Popíšeme v akom formáte má byť zapísaný vstup, ako bude vyzerať výstup, aké kroky vykonajú triedy nášho programu, akým spôsobom dokáže používateľ interagovať s programom, ako nastavenia programu.

2.1 Vstup

predok	e1	root	0	root	1 2 1 5 4 3 2	#	-1 -1 -1 -1 -1 -1 -1
predok	e2	e1	0.05	dup	1 2 1 2 5 4 3 2	#	0 1 2 1 3 4 5 6
clovek	e3	e2	0.12	sp	1 2 1 2 5 4 3 2	#	0 1 2 3 4 5 6 7
clovek	e4	e3	0.13	del	1 2 1 2 4 3 2	#	0 1 2 3 5 6 7
clovek	e5	e4	0.14	ins	1 2 1 6 7 2 4 3 2	#	0 1 2 -1 -1 3 4 5 6
clovek	e6	e5	0.2	inv	1 -1 -2 6 7 2 4 3 2	#	0 2 1 3 4 5 6 7 8
clovek	e7	e6	0.25	leaf	1 -1 -2 6 7 2 4 3 2	#	0 1 2 3 4 5 6 7 8
simpanz	e8	e2	0.12	sp	1 2 1 2 5 4 3 2	#	0 1 2 3 4 5 6 7
simpanz	e9	e8	0.2	leaf	1 2 1 2 5 4 3 2	#	0 1 2 3 4 5 6 7

Tabuľka 2.1: Ukážka vymysleného vstupu v súčasnom formáte.

Vstupom do nášho programu je evolučná história, ktorá popisuje poradie génov v jednotlivých krokoch, a aké sú vzťahy medzi génmi v nasledujúcich krokoch histórie. Vstupný súbor nášho programu bude tvoriť postupnosťou riadkov, podobná tej akú vidíme v tabuľke 2.1. Každý riadok predstavuje jeden *krok evolučnej histórie*. Prvý riadok je koreňom danej histórie, opisuje prvotný stav sekvencie a má priradenú

špeciálnu udalosť "root". Každý ďalší riadok opisuje niektorý z nasledujúcich krokov histórie. Riadok obsahuje zoznam génov nachádzajúcich sa v tomto kroku a spolu so svojím predchodcom nám umožňuje určiť aké evolučné udalosti viedli k súčasnému stavu. Predchodca sa v súbore musí nachádzať vždy skôr než nasledovník, aj preto je prvým riadkom koreň. Riadok obsahuje niekoľko reťazcov a čísel, oddelených medzerou alebo viacerými medzerami, ak je to potrebné pre lepšiu prehľadnosť.

Význam stĺpcov:

Prvý stĺpec je názov biologického druhu (súčasného alebo určitého predka), ktorého sa týka daný riadok.

Druhý stĺpec je id riadku.

Tretí stĺpec je id predchodcu, t.j. kroku histórie ktorý nastal bezprostredne pred aktuálnym krokom. Prvý riadok má špeciálne id predchodcu s hodnotou ”root’. Hoci každý riadok okrem koreňa má jednoznačného predka, naopak to neplatí. V príklade vyššie má riadok $e2$ dvoch potomkov, to znamená že v tomto kroku histórie nastala speciácia a vznikol nový druh, riadok $e7$ nemá žiadneho potomka, tento krok je ukončením jednej vetvy evolučnej histórie.

Štvrtý stĺpec je čas, v ktorom sa daná udalosť odohrala. Koreň sa nachádza v čase 0, a čas smerom k súčasnosti rastie.

Piaty stĺpec je skratka niektorej z udalostí, popísaných v sekcii **??**, pokiaľ sa v danom kroku evolučnej histórie odohrala iba jedna z týchto udalostí, alebo jedna zo trojice udalostí root/leaf/other. Root je udalosť slúžiaca na identifikáciu koreňa. Leaf slúži na určenie času, v ktorom sa daná vetva končí, medzi udalosťou označenou ako leaf a jej predchodcom nemuselo prísť k žiadnym zmenám. Other použijeme, ak rozdiely medzi týmto a predchádzajúcim krokom nie sme schopní popísať pomocou jednej udalosti. Znamená to, že takýto krok vznikol kombináciou viacerých udalostí ako napríklad dve duplikácie nasledujúce po sebe alebo translokácia s následnou deléciou.

Nasledujúce stĺpce obsahujú postupnosť génov v poradí, v akom sa nachádzajú v chromozóme. Identifikátorom každého génu je celé číslo, pričom znamienko určuje jeho orientáciu. To znamená že gén 2 je rovnaký ako gén -2, iba opačne orientovaný v rámci DNA.

Znak # slúži ako ukončenie zoznamu génov.

Zvyšné stĺpce pre každý gén určujú poradie predka génu v predchodcovi jeho riadku. Ak tento gén nemá predchodcu, obsahuje riadok hodnotu -1. Napríklad pre druhý

výskyt génu 2 v riadku e4 vieme, že poradie jeho predchodcu má index 3. Keďže predchodcom e4 je e3 vieme spojiť štvrtý (indexujeme od nuly) gén z riadku e3 s štvrtým génom (gén 2) riadku e4.

2.2 Návrh výstupu

Naším cieľom je zobraziť kompletnú informáciu o génoch, ktoré sa nachádzajú v evolučnej histórii. Potrebujeme preto nájsť spôsob ako túto informáciu pridať do fylogenetického stromu. Naše zobrazenie evolučnej histórie bude predstavovať les stromov, v ktorom jednotlivé stromy reprezentujú gény nachádzajúce sa v skúmaných druhoch. Každý gén bude zobrazený počas celej jeho existencie v evolučnej histórii. Os x slúži ako os času, naľavo sa nachádza čas 0, ktorý postupne rastie tak, aby sa do obrázku zmestil aj najneskorší krok evolučnej histórie.

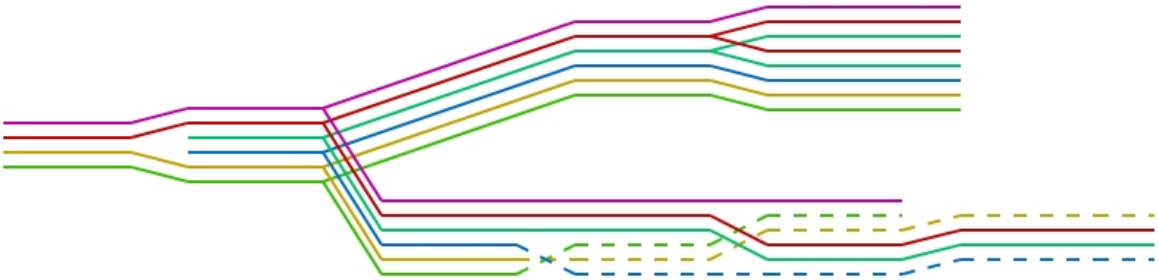
Gény sú znázornené farebnými čiarami, ktoré idú vodorovne, až kým sa nedostanú do bodu, v ktorom má nastať udalosť. Ak sa nasledujúci krok k evolučnej histórie nachádza v čase t_k, tak gény dojdú bez zmeny do bodu $t_k - d$, kde d reprezentuje čas potrebný pre zobrazenie zmien. Zmeny vedúce ku kroku k sa teda udejú počas časového rozmedzia $t_k - d$ až t_k. Pozícia génov v rámci osi y je v udalosti root daná ich reálnym poradím, pričom prvý gén sa nachádza najvyššie. Pre každý ďalší krok tiež platí, že gény daného kroku sú na osi y zoradené podľa poradia, v akom sa nachádzajú v tomto kroku evolučnej histórie.

Duplikáciu, teda skopírovanie jedného alebo viacerých génov, zobrazíme tak, že každý zo stromov reprezentujúci zduplikované gény rozvetvíme. Vetvenie začne v čase $t_k - d$.

Speciáciu by sme v klasickom druhovom fylogenetickom strome zobrazili ako rozvetvenie druhového stromu, počas speciácie teda rozvetvíme všetky gény - získame dve sady vetiev reprezentujúce naše gény v dvoch dcérskych organizmoch. Rozdiel medzi speciáciou a duplikáciou všetkých génov je zreteľný vo vzdialenosti na osi y, kedy pri speciácii rozoznávame dve sady vetiev, ktoré sú od seba dostatočne vzdialené, zatiaľ čo duplikácia by nechala všetky nové gény spolu. Ak sa jedna vetva speciácie nachádza v čase t_a a druhá v čase t_b, zobrazíme začiatok speciácie do času $t_x - d$ kde t_x je skorší z časov t_a, t_a.

Inzerciu génov znázorníme pridaním nového stromu pre každý vložený gén, začiatok pridaných stromov sa nachádza v čase t_k. Deléciu génu zobrazíme ako ukončenie vetvy stromu, ktorá reprezentovala inštanciu tohto génu, takúto vetvu ukočíme v čase $t_k - d$.

Transpozíciu zobrazíme ako kríženie vetiev transponovaných génov tak, aby sa po tomto krížení nachádzali vetvy v správnom poradí, kríženie sa začne v čase $t_k - d$ a skončí v čase t_k.



Obr. 2.1: Príklad zobrazenia histórie naším programom

Inverziu znázorníme podobne ako transpozíciu, avšak keďže pri inverzii okrem zmeny poradia génov dochádza aj k zmene ich orientácie, pridáme do zobrazenia údaj o orientácii génu. Gén so zmenenou orientáciou nebudeme zobrazovať plnou čiarou ale prerušovanou. To nám umožní zobraziť inverziu jedného génu, ktorá by inak nemusela byť viditeľná a inverziu dvoch génov, ktorú by sme si mohli pomýliť s transpozíciou. Zmena štýlu čiary nastáva už v čase $t_k - d$.

Root je začiatok pre všetky stromy génov, ktoré sa nachádzajú v počiatočnom predkovi. Leaf ukončí vetvy génov v čase t_k.

Obrázok 2.1 ilustruje všetky udalosti, ktoré sme práve opísali. Naľavo sa nachádza krok evolučnej histórie "root" obsahujúci štyri gény, nasleduje inzercia dvoch ďalších génov a po nej speciácia. Pri speciácii vzniknú dve vetvy génov ktoré sú od seba dostatočne vzdialené na to aby sme ich rozoznali. Navyše vrchná vetva speciácie sa začína v neskoršom čase, nachádza sa v nej duplikácia dvoch génov a ukončenie v liste. V spodnej vetve sa odohráva inverzia, po nej transpozícia a nakoniec delécia dvoch génov a ukončenie zvyšných v liste.

2.3 Triedy programu EHDraw

V stručnosti si predstavíme základne funkcie, ktoré plnia triedy v našom programe.

EHDraw je hlavnou triedou nášho programu. Načíta vstupné parametre, a na ich základe sa rozhodne či program prebehne neinteraktívne, iba na základe prvotného vstupu, alebo sa spustí interaktívna grafická aplikácia využívajúca JavaFX, ktorá je zapísaná v tejto triede.

EvolutionTree je kľúčovou triedou celého projektu, reprezentuje evolučnú históriu, obsahuje výpočty potrebné pre jej vykreslenie ako aj optimalizáciu. Obsahuje podtriedu *EvolutionNode*, ktorá popisuje jeden krok evolučnej histórie. Táto trieda dostane

vstupnú históriu a tú si uloží vo vhodných dátových štruktúrach. Pre každý krok evolučnej histórie si vytvorí jeden *EvolutionNode* v ktorom okrem referencie predchodcu, drží aj referencie na jeho nasledovníka alebo nasledovníkov, pokiaľ nejakých má. V triede *EvolutionTree* si zapamätáme *EvolutionNode* pre krok root, a ku všetkým zvyšným *EvolutionNode* sa z neho dostaneme rekurzívne.

Pri vykresľovaní zisťuje, akú šírku bude krok zaberať na obrázku. Pre list je táto šírka daná súčtom šírok jeho génov a medzier medzi nimi. Pre zvyšné kroky je daná buď ako ich vlastná šírka, to znamená výpočet rovnaký ako pri liste, alebo ako šírka kroku, ktorý po ňom nasleduje. Vyberáme väčšiu z týchto dvoch hodnôt. Pokiaľ v kroku nastáva vetvenie druhového stromu, vyberieme ako jeho šírku väčšiu z hodnôt jeho vlastnej šírky, alebo súčtu šírok krokov, ktoré z neho vychádzajú, a medzery, ktorú medzi nimi musíme nechať. Keď poznáme šírky, ktoré potrebujú jednotlivé kroky, môže vykresliť našu evolučnú históriu.

V tejto triede taktiež prebieha aj výber génov na zobrazenie, ktorý bližšie popíšeme v časti 3.1.1.

DrawFactory je abstraktná trieda slúžiaca pre vykresľovanie evolučnej histórie. V našom programe sa nachádzajú dve triedy, ktoré ju rozširujú. *FXDrawFactory* slúži na vykreslenie evolučnej histórie v našej JavaFX aplikácii. *SVGDrawFactory* slúži na vykreslenie evolučnej histórie vo formáte *svg* - škálovateľná vektorová grafika, využíva k tomu knižnicu *Apache Batik*. Históriu v tomto formáte vieme následne vyexportovať do súboru. Export je funkcionalita ktorú ponúka iba *SVGDrawFactory*. *EvolutionTree* dostane inštanciu jednej z týchto dvoch tried a pomocou nej vykreslí evolučnú históriu.

GeneMeta reprezentuje nastavenia jedného génu, spôsob ako ich exportovať do XML a tiež možnosť načítať takéto nastavenia z XML súboru.

Settings V tejto triede sa nachádzajú globálne nastavenia ktoré využívame pre vykresľovanie. Sú tu uložené aj *GeneMeta* pre niektoré gény. pokiaľ pre gén neexistuje *GeneMeta*, alebo neobsahujú hodnotu pre požadované nastavenie, využije sa všeobecná hodnota tohto nastavenia, ktorá sa nachádza v triede *Settings*. Trieda *Settings* taktiež obsahuje možnosť vyexportovať nastavenia, vrátane vrátane všetkých *GeneMeta*, do XML súboru, ako aj možnosť nastavenia z XML súboru načítať.

2.4 Ovládanie programu

Náš program ponúka dve možnosti ako ho ovládať. Prvou z nich je grafická aplikácia ktorá používateľovi umožňuje interagovať s nastaveniami a spúšťať funkcie ktoré program obsahuje. Druhou možnosťou je vložiť programu všetky požiadavky, ktoré má

Obr. 2.2: Vzhľad grafickej aplikácie

splniť v príkazovom riadku už pri jeho spustení, tento spôsob nevyžaduje žiadnu ďalšiu interakciu a je ideálny pri potrebe spracovať väčšie množstvo vstupov. Teraz sa pozrieme na to aké možnosti ovládania tieto dve alternatívy ponúkajú.

2.4.1 Grafická aplikácia

Na obrázku 2.2 vidíme základné okno našej aplikácie. V strede sa nachádza plocha na ktorú sa vykresľuje evolučná história. Na ľavo sa nachádza sada tlačidiel a na pravo ovládacie prvky pre zmenu nastavení. Pri spustení má aplikácia rozmery 1200x800 a plocha, na ktorú sa vykresľuje rozmery 800x800.

Tlačidlá na ľavej strane slúžia k základnému ovládaniu nášho programu, popíšeme čo ktoré tlačidlo po stlačení vykoná, a aké to má využitie.

Open otvorí nové okno ktoré nám umožní zvoliť súbor, ktorý chceme otvoriť, tento súbor by mal byť vo formáte popísanom v sekcii 2.1. Zvolená evolučná história sa načíta a vykreslí.

Save Img otvorí nové okno, ktoré nám umožní zvoliť si do akého súboru sa uloží náš súčasný obrázok, ten sa ukladá vo formáte SVG.

Import Settings v novom okne nám umožní vybrať si súbor vo formáte XML obsahujúci nastavenia, ktoré má načítať.

Export Settings umožňuje vybrať súbor do ktorého sa uložia naše nastavenia. Nastavenia sa ukladajú vo formáte XML.

Optimize zostrojí problém množinového pokrytia tak ako je to popísané v kapitole 3 a následne nájde jeho približné riešenie pomocou greedy algoritmu. Podla toho akú hodnotu má pole *optimization* sa toto riešenie prenesie do zobrazenia.

Export ILP podobne ako tlačidlo *Optimize* zostrojí problém množinového pokrytia, a ten uloží zapísaný ako celočíselný lineárny program, do nami zvoleného súboru.

Import ILP v novom oknem nám umožní zvoliť súbor, ktorý sa otvorí, tento súbor obsahuje riešenia celočíselného lineárneho programu. Náš program je nastavený tak aby vedel spracovať riešenia ktoré produkuje *CPLEX*, to akým spôsobom sa načítané riešenie prenesie do zobrazenia evolučnej histórie, závisí od toho, aká hodnota sa nachádza v poli *optimization*.

Redraw po stlačení tohto tlačidla dôjde k prekresleniu obrázku.

Ovládacie prvky na pravej strane Na pravej strane sa nachádzajú dve sady ovládacích prvkov. Prvá sada mení nastavenia ktoré sa nachádzajú priamo v triede *Settings* a sú zdielané všetkými génmi a celým prostredím. Druhá sada mení nastavenia špecificky pre zvolený gén, je to teda spôsob akým meníme nastavenia triedy *GeneMeta* pre konkrétny gén.

Globálne

Time diff je číslo vyjadrujúce koľko času má zabrať zobrazenie udalosti na X-ovej osi, ak má krok evolučnej histórie k známy čas t_k tak jeho udalosť sa začne odohrávať už v čase $t_k - \mathit{Time\ diff}$, tento čas však nesmie byť menší ako čas v ktorom sa odohral predchodca kroku k. Time diff reprezentuje rovnakú premennú ako znak d v sekcii 2.2.

Width určuje dĺžku osi x, teda šírku obrázku, náš program každú evolučnú históriu naškáluje tak, aby ležala na celej tejto osi. Zadáva sa celé číslo.

Height určuje dĺžku osi y, teda výšku obrázku, náš program na šírku neškáluje, to aká časť šírky je pokrytá závisí od šírok génov a ich medzier. Zadáva sa celé číslo.

Node Gap určuje vzdialenosť medzi dvoma sadami vetiev, ktoré vznikli počas speciáce. Zadáva sa celé číslo.

Line Gap určuje medzeru medzi dvoma susednými génmi, ktoré sa nachádzajú v jednom kroku evolučnej histórie.Zadáva sa celé číslo.

Line Size šírka čiary jedného génu, platí pre všetky gény, pokiaľ nemajú svoje vlastné špecifické nastavenie.Zadáva sa celé číslo.

Saturation a *Brightness* nastavenie sýtosti a jasu farieb. Farby génom priraďujeme z modelu HSB - hue,saturation,brightness - slovensky odtieň,sýtosť,jas. Odtieň v tomto modeli je daný v stupňoch hodnotou $0° - 360°$. To nám umožňuje rovnomerne rozdeliť odtiene medzi n génov. Sýtosť a jas sú potom pre všetky gény rovnaké, určujeme ich reálnym číslo z rozmedzia $< 0, 1 >$.

Optimization určuje akým spôsob sa prejaví optimalizácia na výslednom zobrazení evolučnej histórie. Pri hodnote 2 sa génom, ktoré sa nenachádzajú v riešení nastaví atribút $\mathit{higlighted} = \mathit{false}$, vykreslia sa farbou ktorá je daná ako *Non highlighted*, takéto nastavenie vidíme na obrázku [?] v okienku číslo 2. Pri hodnote 1 sa nezvole-

ním génom nastaví atribút $trasnparent = true$, zobrazia sa priesvitne, rovnako ako v okienku číslo 3. Pri hodnote 0 sa nezvolením génom nastaví atribút $draw = false$, a teda sa nezobrazia vôbec, rovnako ako štvrtom okienku.

Non highlighted umožňuje výber farby pre gény, ktoré nie sú zvýraznené. To sú tie pre ktoré $highlighted = false$ a zároveň $draw = true$ a $transparent = false$. Tieto hodnoty génom manuálne nastavíme alebo ich získajú pokiaľ sa nenachádzajú v riešení optimalizácie a $optimization = 2$.

Redraw je prepínač, pokiaľ je zapnutý, obrázok sa automaticky prekreslí po tom, čo zmeníme niektoré z jeho nastavení. V opačnom prípade obrázok prekreslíme stlačením ľavého tlačidla *Redraw*.

Špecifické pre gén sa aplikujú na vybraný gén ktorý zvolíme zo zoznamu. V tomto zozname sa nachádzajú všetky gény z histórie, bez ohľadu na to, či pre ne existuje záznam *GeneMeta* uložení v nastaveniach *Settings*. Okrem toho sa v liste nachádzajú dve špeciálne hodnoty. *None* vyjadruje že nemáme zvolený žiaden gén a nebudeme vedieť meniť žiadne nastavenia špecifické pre gén. *Default* slúži k nastaveniu hodnôt, ktoré využívajú všetky gény bez vlastných *GeneMeta*. Pre *Default* nevieme nastavovať *Name* ani *Colour*. Pokiaľ si z listu zvolíme gén ktorý nemá vlastný záznam *GeneMeta*, tento záznam sa vytvorí až keď tomuto génu nastavíme niektorú z hodnôt.Do polí *Draw,Transparent a Highlighted* sa prekopíruje hodnota nastavená v *Default*, zvyšné polia môžu zostať prázdne. Pre prázdne polia sa aj naďalej využívajú hodnoty z *Default*.

Draw boolean prepínač, hodnota určuje či sa gén vykreslí.

Transparent boolean prepínač, pokiaľ sa má gén vykresliť ($draw = true$), tak hodnota *Transparent* určí, či bude priesvitný $transparent = true$, kedy gén nevidíme, ale zaberá miesto. Alebo bude nepriesvitný, kedy farbu získa v závislosti od hodnoty *Highlighted*.

Highlighted pokiaľ sa má gén nepriesvitne vykresliť $draw = true, transparent = false$ highlighted rozhodujem o tom, či bude zvýraznený. Ak je zvýraznený zobrazí sa s farbou ktorú má nastavenú v *Colour*, v prípade že nemá nastavenú vlastnú farbou použije predpočítanú farbu. Ak nieje zvýraznený použije farbu nastavenú v *Non Highlighted*.

Line Width umožňuje individuálne nastavenie šírky génu. Vkladáme celé číslo.

Name je textové pole, v ktorom môžme génu nastaviť jeho meno.

Colour ponúka výber farby špeciálne pre daný gén.

Delete gene meta odstráni záznam *GeneMeta* pre práve zvolený gén.Nefunguje pre *Default*.

Delete all meta odstráni všetky existujúce záznamy *GeneMeta* okrem záznamu *Default*.

2.4.2 Argumenty príkazového riadku

Nie vždy pre nás musí byť výhodnejšie, ovládať náš program cez grafickú aplikáciu. Ako alternatívu ku grafickej aplikácii ponúka náš program možnost, vložiť mu základné parametre potrebné pre jeho beh už pri jeho spustení. Program potom sám vykoná zvolené úkony, bez potreby akejkoľvek ďalšej interakcie zo strany uživateľa. Tento spôsob ovládania programu je výhodný napríklad vtedy, keď cheme automatizovať spracovanie väčšieho počtu vstupov. Predstavíme si aké argumenty vieme vložiťna príkazový riadok, a aký vplyv budú mať na priebeh nášho programu. Každý argument ktorý týtmto spôsobom programu vložíme má na začiatku pomlčku. Poradie argumentov nie je dôležité.

-*nogui* argument určujúci že náš program nespustí grafickú aplikáciu, ale vystačí si len s údajmi zo vstupu.

-*input:filename.history* argument, ktorým ukážeme na súbor "filename.history" obsahujúci evolučnú históriu. Evolučná história ktorá sa nachádza v tomto súboré sa načíta do programu.

-*drawsvg* nahraná evolučná história sa vykreslí vo formáte svg do súboru output.svg.

-*svg_ output:filename.svg* miesto súboru output.svg sa obrázok uloží do súboru "filename.svg".

-*opt:x* prebehne optimalizáciu počtu zobrazených génov, číslo x má rovnakú funkciu ako nastavenie *Optimization* z grafickej aplikácie.

-*exportgreedy* do súboru output.greedy uloží greedy riešenie problému množinového pokrytia pre nahranú evolučnú históriu.

-*greedy_ output:filename.svg* miesto do súboru output.greedy sa greedy riešenie uloží do súboru "filename.svg".

-*exportilp* zadanie celočíselného lineárneho programu - 3.3.2, ktorý je ekvivalentný k problému množinového pokrytia pre nahranú evolučnú históriu, sa uloží do súboru output.lp

ilp_ output:filename.lp namiesto output.lp sa pre uloženie zadanie celočíselného lineárneho programu využije súbor "filename.lp".

-*load_ lp:input.sol* optimalizuje zobrazené gény na základe riešenia zo súboru ”input.sol". Program je nastavený tak aby akceptoval riešenie ktoré prodkuje CPLEX.

-*load_ settings:filename.xml* do triedy *Settings* načítana nastavenia zo súboru “filename.xml”.

Kapitola 3

Problém množinového pokrytia a výber génov

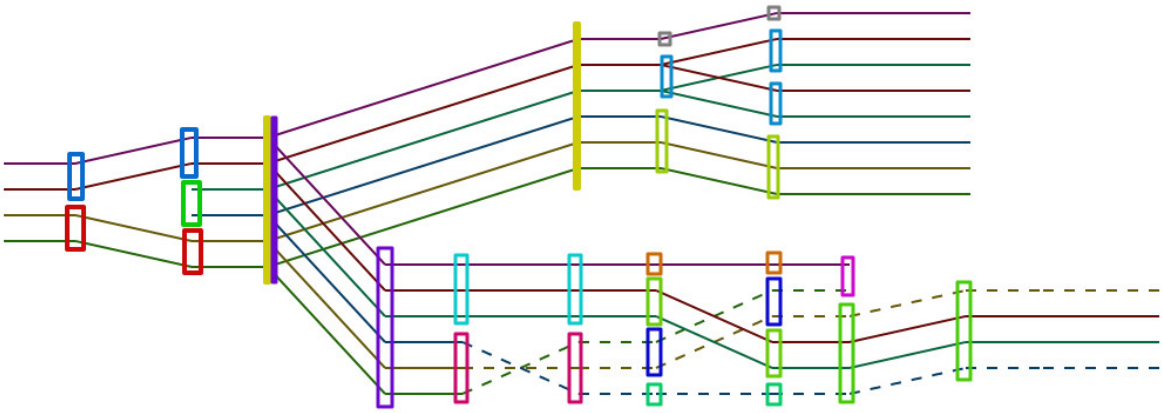
V predchádzajúcej kapitole sme predstavili základné prvky nášho programu ktorý, dostane na vstupe súbor, popisujúci evolučnú históriu a na výstupe danú históriu vykreslí. Problém nastane, pokiaľ sa v histórii nachádza príliš veľa génov. Výsledný vygenerovaný obrázok sa stáva neprehľadným, a získanie informácie z neho obtiažne. Potrebujeme teda vybrať iba niektoré gény na zobrazenie tak, aby na obrázku zostali zachované podstatné informácie. V tejto kapitole si predstavíme spôsob, akým môžme automaticky vyberať, ktoré gény zobrazíme. V našej metóde využijeme *Problému množinového pokrytia* pričom implementujeme dva algoritmy, ktoré riešia daný problém.

3.1 Výber génov

Najpodstatnejšou informáciou pri analýze evolučnej histórie je pre nás to, aké udalosti sa v nej odohrali. Budeme sa teda snažiť nájsť podmnožinu všetkých génov tak, aby všetky udalosti ostali na obrázku zachované. Zvyšné gény následne z obrázku odstránime, čo môže viesť k strate informácie, ktorú považujeme za menej podstatnú, ako napríklad, koľko a ktoré gény sa nachádzajú v danej histórii, ako aj koľko a ktoré gény sú ovplyvnené jednotlivými udalosťami.

3.1.1 Blok

Aby sme formálne zadefinovali problém výberu génov, zavedieme pojem bloku. Tento pojem sa vzťahuje na jednu konkrétnu dvojicu krok a jeho predchodca. *Blok* predstavuje postupnosť génov, ktoré sa pred aj po zmenách, medzi krokom a jeho predchodcom, nachádzali vedľa seba v rovnakom poradí, a jednotlivé gény nemenili svoju orientáciu. Jedná sa teda o súvislý úsek DNA, ktorý počas kroku nebol prerušený. Ak sa pri delécii alebo inzercii odobralo alebo pridalo niekoľko génov, ktoré tvoria súvislý



Obr. 3.1: Ukážka blokov.

úsek, tvoria jeden blok. Gény naľavo a na pravo od týchto zmazaných resp. vložených génov budú tvoriť ďalšie dva bloky. Pri duplikácii gény tvoria blok, ak sa nachádzali pri sebe pred duplikáciou a rovnako aj po nej vo všetkých zduplikovaných inštanciách. Pri inverzii tvoria jeden blok gény, ktorým sa zmení orientácia, t.j. Napr blok génov (4,5,-6) bude po inverzii vyzerať ako (6,-5,-4). Ukážka evolučnej histórie so zobrazenými blokmi je na obrázku 3.1, v hornej vetve vidíme blok duplikácie, kedy z jedného bloku vznikli dva totožné bloky. Pre každú vetvu speciácie sa hľadajú bloky medzi predkom a novým druhom zvlášť.

Pre každú dvojicu krok k a jeho predchodca k_p, bez ohľadu na to, koľko evolučných udaostí sa medzi nimi odohralo, nájdeme kroky nasledovním spôsobom. Každý gén g_i z kroku k_p vložíme do nového bloku $b_g i$, každému génu v bloku k, ktorý má predka v bloku k_p priradíme rovnaký blok, ako má jeho predok. Ak má gén $-g_i$ z bloku k predka g_i v bloku k_p, a g_i má priradený blok $b_g i$, tak génu $-g_i$ priradíme blok $-b_g i$. Génom v k ktoré nemajú predka priradíme nové bloky. Každú dvojicu blokov $b1$ a za ním nasledujúci $b2$ z k_p spojíme do jedného bloku ak: Blok $b1$ ani $b2$ a ani blok $-b1$ alebo $-b2$ sa nenachádza v kroku k, teda došlo k delécii a bloky môžme spojiť. Ak sa v kroku k nachádza blok $b1$, vždy za ním nasleduje $b2$, a naopak bloku $b2$ vždy predchádza blok $b1$, tak isto, po bloku $-b2$ musí vždy nasledovať $-b1$, a bloku $-b1$ vždy predchádza blok $-b2$, to nám zaručí že bloky (a teda gény v nich) sa počas udalosti vždy nachádzali vedľa seb.

Ked bloky $b1$ a $b2$ spojíme do bloku $b1$ v k_p, spoja sa do bloku $b1$ aj v k a bloky $-b2$ a $-b1$ sa spoja do bloku $-b1$. Týmto spôsobom prechádzame bloky v kroku k_p, pokiaľ dochádza k spojeniam blokov. Na strane k potom pospájame susedné bloky, ktoré vznikli inzerciou - to sú tie bez predka na strane k_p.

Pokrytie blokov

Blok považujeme za pokrytý pokiaľ sa na obrázku vyskytuje aspoň jeden gén patriaci do daného bloku. Pokrytie všetkých blokov jednej dvojice krok a predchodca kroku nám zaručí zobrazenie všetkých udalostí, ktoré sa medzi týmito krokmi vyskytli. Musíme preto nájsť také gény, ktoré pokryjú všetky bloky v kompletnej evolučnej histórii, a tým zaistiť zobrazenie všetkých udalostí vo výslednom zobrazení. Ako cieľ si zvolíme, aby bola vybraná množina génov čo najmenšia.

3.2 Problém množinového pokrytia

Ako ukážeme nižšie, problém výberu génov úzko súvisí s dobre známym problémom množinového pokrytia.

Definícia Máme dané univerzum U, ktoré obsahuje n prvkov, a systém jeho podmnožín $S = \{P_i : P_i \subseteq U\}$, ktorý pokrýva celé univerzum, t.j. $\cup_{P_i \epsilon S} P_i = U$. Cieľom je vybrať čo najmenšiu množinu podmnožín $C \subseteq S$, ktorá tiež pokryje celé univerzum $\cup_{P_i \epsilon C} P_i = U$. Problém množinového pokrytia (anglicky Set Cover Problem), ďalej len *PMP*. patrí medzi NP-úplne problémy.[8]

Príklad Pre univerzum $U = \{1, 2, 3, 4, 5, 6\}$
a systém jeho podmnožín $S = \{\{1, 2, 3\}, \{2, 3\}, \{3, 4\}, \{3, 4, 6\}, \{5\}\}$
je riešením množina podmnožín $C = \{\{1, 2, 3\}, \{3, 4, 6\}, \{5\}\}$ veľkosti 3.

3.2.1 Výber génov pomocou Problému Množinového Pokrytia

Výber génov, ktoré pokryjú všetky bloky v celej evolučnej histórii, vieme formulovať ako Problém množinového pokrytia. Univerzum predstavuje všetky bloky, ktoré sa nachádzajú v našej histórii. Každý gén predstavuje jednu podmnožinu systému S, v ktorej sa nachádzajú tie bloky, cez ktoré gén prechádza. Riešením je taká množina génov, ktorých zjednotenie pokrýva všetky prvky univerza, v našom prípade všetky bloky nachádzajúce sa v evolučnej histórii.

3.3 Riešenie Problému množinového pokrytia

Keďže *PMP* patrí medzi NP-ťažké problémy, znamená to, že zatiaľ neexistuje, a možno nikdy ani nebude nájdený algoritmus, ktorý by dokázal nájsť riešenie v polynomiálnom čase. Potrebujeme sa teda rozhodnúť, či je pre nás výhodnejšie hľadať najlepšie riešenie *PMP*, čo môže byť časovo náročné, alebo sa uspokojíme s približným riešením, ktoré sme schopní nájsť aproximačným algoritmom v polynomiálnom čase, a ktoré môže taktiež

predstavovať dostatočné odstránenie prebytočných génov z obrázku. Predstavíme si jeden spôsob ktorým budeme hľadať optimálne riešenie a jeden spôsob na nájdenie približného riešenia. V nasledujúcej kapitole porovnáme výsledky, ktoré produkujú.

3.3.1 Greedy algoritmus

Greedy algoritmus patrí medzi najlepšie polynomiálne aproximačné algoritmy pre riešenie *PMP*.[11] Greedy algoritmus v každom kroku pridá do riešenia takú podmnožinu, ktorá obsahuje najviac zatiaľ nepokrytých prvkov univerza. Riešenie teda hľadáme nasledovným spôsobom:

Všetky podmnožiny zoradíme na základe toho, koľko prvkov obsahujú. Do riešenia vyberieme najväčšiu podmnožinu a prvky, ktoré sa v nej nachádzajú odstránime z univerza aj zo zvyšných podmnožín. Zvyšné podmnožiny opäť zoradíme podľa veľkosti, a postup opakujeme až pokým nie je univerzum prázdne.

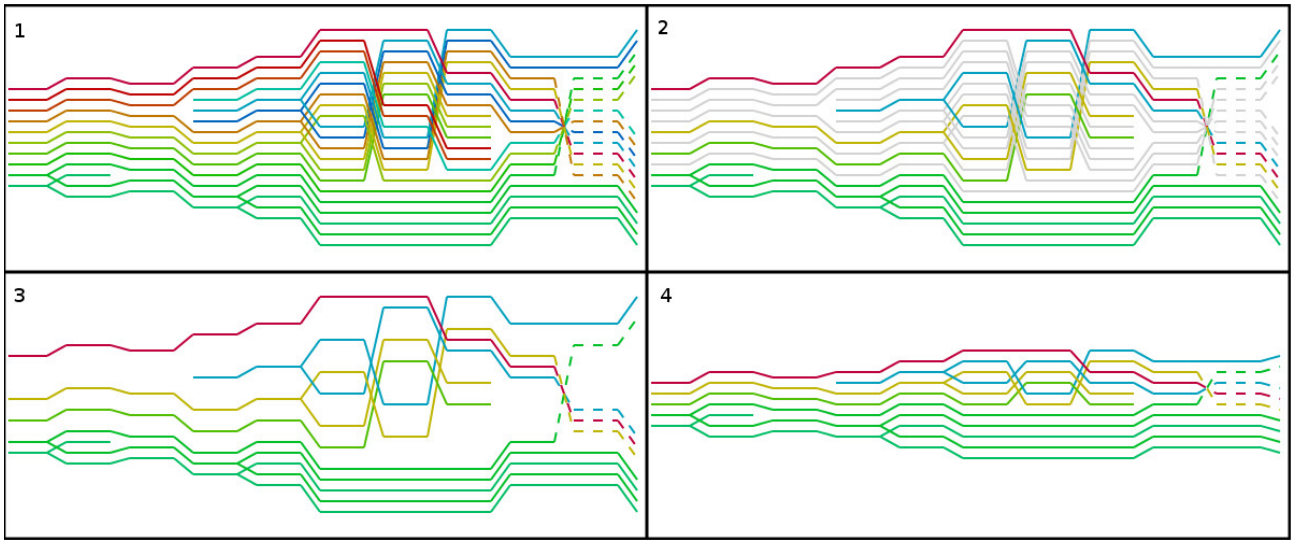
Tesná analýza podľa Slavíka ukazuje, že aproximačný koeficient takéhoto riešenia je $\ln m - \ln \ln m \ + \Theta(1)$ [11] kde $m = |U|$.

3.3.2 Binárne celočíselné lineárne programovanie

Lineárne programovanie je optimalizačná úloha, pri ktorej je cieľom nájsť minimum alebo maximum lineárnej funkcie f s n premennými, ktoré spĺňa dané obmedzenia vo forme lineárnych rovníc a nerovníc. V prípade Binárneho celočíselného lineárneho programovania nadobúdajú premenné hodnotu 0 alebo 1, a všetky atribúty obmedzujúcich rovníc a nerovníc sú celočíselné. Binárne celočíselné lineárne programovanie (angl: binary integer linear programming),ďalej BCLP patrí medzi NP-úplné problémy [8]. Mnohé iné problémy, ako napríklad Problém obchodného cestujúceho, Problém vrcholového pokrytia a *PMP*, môžu byť formulované ako Celočíselné linárne programovanie. Navyše pre Celočíselné lineárne programovanie existuje množstvo programov, špecializovaných na hľadanie riešenia v čo najlepšom čase [13]

Prevedenie výberu génov na BCLP

Všetky gény nachádzajúce sa v našej evolučnej histórii očíslujeme číslami $1, 2, .., n$. Premenná x_i bude nadobúdať hodnotu 0 alebo 1 v závislosti od toho či sa gén čísl i nachádza v riešení. Lineárna funkcia, ktorú chceme minimalizovať, bude v tvare $\min \ x_1 + x_2 + x_3 + .. + x_n$. Lineárne obmedzenia vytvoríme tak, že pre každý blok B_a sa pozrieme na všetky gény, ktoré daný blok pokrývajú $x_{ai} : x_{ai} \in B_a$, a pridáme podmienku, že súčet premenných reprezentujúcich takéto gény musí byť aspoň 1, t.j. $\sum_{x_i \in B_a} x_i \geq 1$. Čiže v riešení sa musí nachádzať aspoň jeden gén pokrývajúci daný blok. Vo výslednom *BCLP* sa teda bude nachádzať jedna lineárna funkcia, ktorú chceme mi-



Obr. 3.2: Priebeh výberu génov

nimalizovať a k lineárnych obmedzení, kde k je počet blokov v celej histórii. Program obsahuje n premenných kde n je počet génov.

Riešenie BCLP Cieľom tejto práce nie je nájsť najlepší spôsob, alebo zostrojiť najlepší program pre riešenie *BCLP* či *PMP*. Oba problémy sú len prostriedkom ako dosiahnuť optimalizáciu množstva zobrazených génov a tým zvýšiť prehľadnosť. V prípade greedy algoritmu sa implementácia nachádza priamo v našom programe, čo nám umožnuje v krátkom čas dospieť aspoň k čiastkovej optimalizácii. V prípade BCLP náš program nevie nájsť riešenie, ponúka však možnosť vyexportovať sformulovaný *BCLP*. Pre samotné riešenie *BCLP* je vhodné použiť niektorý z existujúcich nástrojov [13], a riešenie nahrať do nášho programu. Pre účely tejto práce bol využívaný IBM ILOG CPLEX Optimization Studio - *CPLEX*[1].

3.3.3 Výsledok optimalizácie

Priebeh výberu génov si môžeme ilustrovať na obrázku 3.2. Na začiatku dostane náš program evolučnú históriu s kompletnou informáciou, ako vidíme v časti 1). Následne zostrojíme všetky bloky a nájdeme riešenie pre daný *PMP*. V časti 2) sú gény, ktoré sa nachádzajú v riešení vyznačené farebne, zvyšné gény sú šedé. Prebytočné gény z obrázku odstránime. V časti 3) môžme pozorovať, ako sa ich odstránením obrázok preriedi, napriek tomu ostávajú všetky udalosti zachované. V časti 4) prebytočné gény už nezaberajú žiadne miesto. Všetky udalosti zostali zachované, nie sme však už schopní určiť ich pôvodný rozsah, ani počet génov, ktoré sa pôvodne nachádzali na obrázku. Podľa cieľa vizualizácie môže byť vhodné využiť finálny obrázok, alebo aj niektorý z medzi stupňov.

Kapitola 4

Porovnanie optimalizačných metód

V predchádzajúcej kapitole sme popísali, čo je to problém množinového pokrytia, a ako ho vieme využiť pre výber takej sady génov, ktorá nám zobrazí všetky udalosti, ktoré sa odohrali počas evolučnej histórie. Náš program obsahuje greedy algoritmus, ktorý nájde približné riešenie daného problému. Inou možnosťou je, previesť problém množinového pokrytia na celočíselný lineárny program, ktorého vyriešenia nájde optimálne riešenie. Pre greedy algoritmus poznáme jeho aproximačný koeficient $\ln m - \ln \ln m + \Theta(1)$ [11] kde $m = |U|$. Všetky problémy množinového pokrytia, ktoré riešime majú spoločnú črtu, a to že vznikli ako prepis evolučnej histórie. Toto špecifikum môže mať vplyv na to, ako blízko k optimálnemu riešeniu sa priblíži greedy algoritmus. V tejto kapitole sa budeme venovať tomu, aké sú reálne rozdiely vo výsledkoch medzi aproximáciou a optimálnym riešením. Popíšeme spôsob akým testy prebiehajú, pozrieme sa na vyprodukované dáta, a porovnáme výsledky pre greedy algoritmus a celočíselné lineárne programovanie.

4.1 Priebeh testovania

Priebeh testovania rozdelíme na tri časti. Prvou časťou je vytvorenie vstupných dát, na tento účel slúži pomocný program ktorý vytvorí požadované evolučné histórie. Druhým krokom je spracovanie týchto histórii, to znamená vyriešenie problému množinového pokrytia pre každú z nich pomocou greedy algoritmu ako aj pomocou celočíselného lineárneho programovania. Záverečná časť spočíva v zozbieraní získaných dát a ich analýze.

4.1.1 Generovanie evolučných histórií

Pre tento účel sme zostrojili jednoduchý program umožňujúci generovanie evolučných histórií. Program dostane na vstupe súbor s údajmi ako v tabuľke 4.1.1. Node_numb udáva koľko udalostí sa bude vo vygenerovanej histórii nachádzať a gene_numb určuje

node_numb	10	
gene_numb	10	
len_rate	0.75	
duplication	1	
inversion	1	
deletion	1	3
insertion	1	5
transposition	1	

Tabuľka 4.1: Ukážka vstupu pre generátor histórií, súbor settings.generator

počet počiatočných génov. Každý krok evolučnej histórie bude obsahovať jednu udalosť, tú umiestnime na náhodné miesto do sekvencie génov, jej dĺžku, teda to koľko génov sa v tejto udalosti nachádza vypočítame na základe hodnoty len_rate. V udalosti sa určite nachádza aspoň jeden gén, každý ďalší pridáme s pravdepodobnosťou len_rate, jedná sa teda o geometrickú postupnosť. To, aká udalosť sa v danom kroku odohrá, určíme náhodne. Prvá hodnota za každou udalosťou z tabuľky 4.1.1, vyjadruje pomer, s akým sa daná udalosť vyberie. V našom prípade sú udalosti v pomere 1:1:1:1:1 mali by byť teda zastúpené rovnomerne. Pokiaľ niektorú z udalostí v histórii nechceme, nastavíme jej hodnotu 0. Pre inzerciu a deléciu vieme navyše nastaviť maximálny počet génov, ktoré ovplyvnia, keďže sú to udalosti ktoré menia počet génov v kroku (Duplikácia taktiež mení počet génov, ale vytvára iba kópie, nepridáva žiadne nové gény). Pri transpozícii gény presunieme na náhodné miesto. Okrem vstupného súboru, môžme generátoru zadať aj počet, koľko histórií má pre dané parametre vytvoriť.

Spracovanie vygenerovaných histórií Pre všetky vygenerované histórie následne spustíme hlavný program, a vyexportujeme riešenie problému množinového pokrytia pomocou greedy algoritmu. Ďalej vyexportujeme formuláciu celočíselného lineárne programu, a jeho riešenie pomocou CPLEXu. Pre každú históriu si zapamätáme, aké množstvo blokov sme pokrývali a koľko génov sa na to mohlo vuyžiť. Aký počet z týchto génov bol zvolený do greedy riešenia a aký do optimálneho riešenia CPLEXom. Posledným údajom budú časy, čas potrebný pre načítanie evolučnej histórie, čas ktorý potreboval greedy algoritmus pre nájdenie riešenia a čas pri hľadaní optima. Pre všetky histórie, ktoré generátor vytvoril z rovnakých nastavení, dáme tieto údaje dokopy. Získame priemerné hodnoty pre dané nastavenia generátora. To nám umožní sledovať vplyv nastavení generátora na výsledky testovania.

udalosti	gény	p	id	univerzum	subsety	greedy	greedy čas	ilp	ilp čas
10	10	0.8	1	32	12	9	0.9400	9	0.9500
10	10	0.8	2	42	20	15	0.9700	15	0.9400
10	10	0.8	3	40	20	14	0.9600	13	1.0000
10	10	0.8	4	30	10	6	0.9400	6	0.9600
10	10	0.8	5	31	11	7	0.9300	7	0.9600
10	10	0.8	6	38	20	16	0.9800	16	0.9500
10	10	0.8	7	38	16	12	0.9400	12	0.9400
10	10	0.8	8	34	16	14	0.9300	13	0.9700
10	10	0.8	9	34	10	10	0.9200	10	0.9100
10	10	0.8	10	37	17	15	0.9000	15	0.9500
10	10	0.8	avg	35.60	15.20	11.80	0.9410	11.60	0.9530

Tabuľka 4.2: Ukážka získaných dát zo súboru file.data, pre jedny nastavenia generátora file.generator

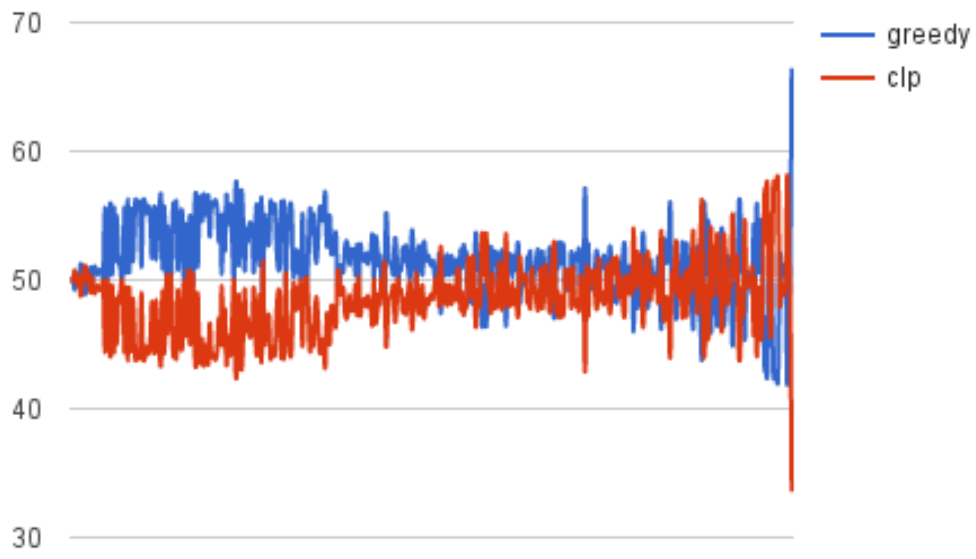
4.1.2 Ako zopakovať testovanie ?

Ak má čitateľ záujem zopakovať testovanie, tu je návod ako na to. K tejto práci, v priečinku *Testovanie* prikladám všetky súbory potrebné pre zopakovanie testovania. Konkrétne sú to tri java programy: EHDraw.jar je hlavný program popísaný v tejto práci, spracováva histórie, hľadá greedy riešenie a exportuje zadanie celočíselného lineárneho programu. Generator.jar je popísaný v sekcii 4.1.1, na vstup dostáva súbor podobný tomu v tabuľke 4.1.1 a údaj koľko histórií z neho má vytvoriť. Pre zadaný vstup $filename.generator$ a počet n vytvorí histórie $filename\#1.history$,$filename\#2.history$ .. $filename\#n.history$. TestParser.jar je slúži k spracovaniu údajov pre všetky histórie ktoré vznikli z rovnakých nastavení generátora. Zoradí ich do CVS tabuľky - tabuľka v ktorej sú hodnoty oddelené medzeramy alebo iným znakom, v našom prípade čiarkami, každá história bude v jednom riadku, plus sa tu nachádza riadok *avg* s priemernými hodnotami, tak ako to môžme vidieť v tabuľke 4.1.2. Okrem týchto troch programov prikladám aj program ToGenerator.jar, vygeneruje súbory *.generator, ktoré boli použité v našom testovaní. Automatizáciu testovania zabezpečuje Linuxový príkaz *make*. Vzťahy medzi súbormi, a spôsob akým *make* vytvára nové súbory je popísaný v súbore *Makefile*. Na začiatku musíme existovať súbor $filename.generator$. Prvé zavolanie príkazu *make* pre každý takýto súbor spustí generátor histórií. Údaj o tom, koľko histórií sa vygeneruje pre jedny nastavenia sa nachádza v *Makefile*. Druhé zavolanie príkazu *make* pre každú históriu $filename\#i.history$ vytvorenú v predchádzajúcom kroku, nájde greedy riešenie, optimálne riešenie za pomoci CPLEXu, a všetky výsledné údaje podstatné pre testovanie uloží do súboru $filename\#i.grepped$. Tretie

zavolanie príkazu *make* pre každý súbor $filename.generator$, z ktorého sme generovali histórie $filename\#1.history$,$filename\#2.history$ .. $filename\#n.history$, vytvorí *filename.data*. V tomto súbore sa nachádza tabuľka, podobná tabuľke 4.1.2, vytvorená programom *TestParser.jar*. Prvý stĺpec je počet krokov ktoré sa odohrali, nasleduje počet génov v kroku "root", pravdepodobnosť s akou sme gén pridali, id súboru s históriou, veľkosť univerza, počet podmnožín ktorými pokrývame (t.j. počet všetkých génov ktoré sa nachádzajú v histórii),počet prvkov v riešení a čas pre greedy a lineárne programovanie. Následné spracovanie dát prebieha manuálne, na základe informácií ktoré poznáme o vstupe pre generátor.

4.1.3 Výsledky testovania

Dáta, ktoré analyzujeme, sme získali postupom uvedením v v sekcii 4.1.2. Jedná sa o celkovo 475 rôznych vstupných súborov $*.data$, v tomto súbore je každý riadok jedna evolučná história, my budeme pracovať s riadkami "avg", v ktorých sa nachádzajú priemerné hodnoty pre tieto histórie. Pokúsim sa uviesť niekoľko výsledkov, ktoré sú podstatné pre túto prácu.



Obr. 4.1: Pomer času greedy a clp, pri hľadaní riešenia

V kapitole **??**, sa spomína, že greedy algoritmus nájde aproximáciu NP-ťažkého problému množinového pokrytia v polynomiálnom čase. Preskúmame, či greedy algo-

poradie	udalosti	gény	p	univerzum	subsety	greedy čas	ilp
1	10	10	0.9	22.3	10	0.75	0.75
2	10	10	0.7	24.6	10	0.759	0.757
.							
80	100	10	0.9	225.4	10	0.949	0.979
81	10	50	0.7	38.5	55.9	1.063	0.87
.							
407	200	100	0.97	3060.3	1440.4	11.234	11.443
408	100	500	0.95	298.4	500	12.136	10.834
.							
472	200	500	0.97	5609.5	5709.75	64.88	90.0225
473	200	500	0.97	594.67	500	74.6933	81.7333
474	200	500	0.97	596.9	500	86.937	82.496
475	200	50	0.97	593.1	50	129.09	65.178

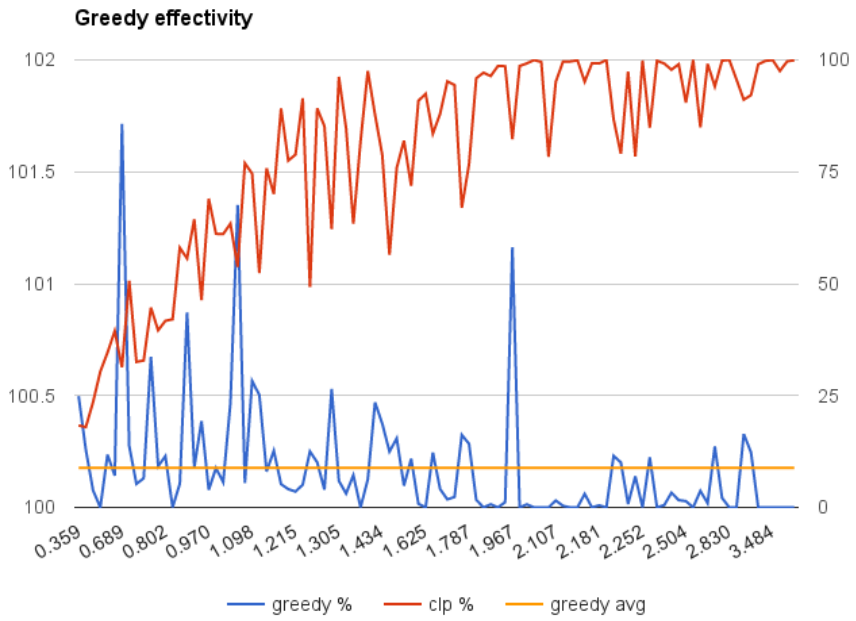
Tabuľka 4.3: Vybraných 10 zo 475 riadkov "avg", zoradené sú podla súčtu času greedy a ilp.

ritmus predstavuje časovú úsporu, a ak áno ako významnú. Na obrázku 4.1 vidíme pomer času potrebného pre riešenie greedy, vzhľadom k času potrebnému pre riešenie pomocou Celočíselného lineárneho programu. Záznamy sú na osi x zoradené vzostupne podľa toho, koľko času potreboval greedy algoritmus spolu s Celočíselným lineárnym programovaním (CLP) pre nájdenie riešenia. Pri krátkych vstupoch nájdeme riešenie rýchlejšie s *CLP*, než s greedy algoritmom, zhruba v polovici záznamov sa tento pomer začína meniť. Pri najkrajnejších záznamoch, ktoré vidíme aj v tabuľke 4.1.2, dostávame pre greedy a *CLP* veľmi nesúrodé časy. Pokiaľ preskúmame časy, pre jednotlivé histórie v pozadí týchto záznamov, zistíme, že väčšina času pri riešení *CLP* bola využitá programom EHDraw na načítanie histórie a vytvorenie celočíselného lineárneho programu. *CPLEXu* pre takéto histórie, ktoré sú aj po optimalizácii príliš rozsiahle na zobrazenie naším programom, stačil na nájdenie riešenia iba zlomok z celkového času.

Druhou vlastnosťou greedy algoritmu ktorá nás zaujíma je, vzdialenosť jeho riešení od optima.

Graf na obrázku 4.2 je zostrojený zo záznamov $all*.data$, v tých sa nachádzajú dáta z histórií, v ktorých boli povolené všetky udalosti. Na osi x sa nachádza pomer medzi medzi množstvom blokov, ktoré musíme pokryť, a počtom génov, s ktorými tieto bloky pokrývame. Modrá čiara *greedy %* vyjadruje percentuálne, aký počet génov sa nachádza v greedy riešení vzhľadom na optimum, odčítavame ju z hodnôt na ľavej strane. Žltá čiara *greedy avg* je priemerná hodnota tohto údaju, *greedy avg* = 100.177437%, odčítavame ju z hodnôt na ľavej strane. Oranžová čiara *clp %* udáva, aké percento z

génov sa nachádza v optimálnom riešení, odčítavame ju z hodnôt na pravej strane. S pomocou *clp %* vidíme, že pokiaľ máme príliš veľa blokov, vzhľadom k počtu génov, využijeme pri riešení problému množinového pokrytia väčšinu génov. V takom prípade prestáva byť optimalizácia efektívna, a aproximácia sa približuje k optimu, lebo obe riešenia potrebujú vybrať skoro všetky gény. Naopak, pokiaľ si optimum postačí s malou časťou génov, narastá priestor pre odchýlku aproximácie od optima. Z grafu ďalej vidíme, že aproximácia je v zaznamenanom maxime 101.74% a priemerne 100.18%. Táto odchýlka aproximácie od ptima sa v praxi, t.j. pri evolučných históriách, ktoré sú zobraziteľné naším programom, prejaví rozdielom ktorý je zanedbateľný.



Obr. 4.2: Graf zobrazujúci aké percento génov sa nachádza v optimálnom riešení, v závislosti od priemerného počtu unikátnych blokov na jeden gén. (pre 10 blokov a 100 génov pripadá 0.1 bloku na gén) V grafe sa nachádza aj údaj o tom, ako ďaleko od tohto optima je aproximácia.

Záver

Cieľom tejto práce bolo implementovať systém na vizualizáciu evolučných histórií, s dôrazom na zmeny, ktoré sa v nich odohrali. Výsledkom je program EHDraw ktorý umožňuje jednoduchú manipuláciu za pomoci grafickej aplikácie. Určený je najmä pre vedeckých pracovníkov, ktorý sa zaoberajú problémami z oblasti biológie a bioinformatiky, konkrétne rekonštrukciou DNA sekvence alebo celého fylogenetického stromu. Ponúka im jednoduchú vizuálnu kontrolu ich práce, ako aj možnosť prezentovať svoje výsledky formou, ktorá dokáže osloviť širšie publikum. Za týmto účelom umožňuje náš program zmenu veľkého počtu nastavení, používateľ získava kontrolu nad vzhľadom vyprodukovanej vizualizácie. S dôrazom na zobrazenie všetkých zmien, riešime aj problém výberu génov. Jeho cieľom je nájsť takú sadu génov, ktoré môžme zo zobrazenia odstráni bez toho, aby sme stratili informáciu o udalostiach v evolučnej histórií. Odstránenie niektorých génov má uľahčiť nájdenie podstatnej informácie na obrázku. Takúto sadu génov nájdeme pomocou Problému množinového pokrytia, pre jeho vyriešenie môžme využiť buď greedy algoritmus alebo Celočíselné lineárne programovanie. Pri Celočíselnom lineárnom programovaní máme záruku nájdenia optima, zatiaľčo greedy hľadá aproximáciu. Praktickými testami sme overovali či greedy algoritmus predstavuje časovú úsporu, a o koľko horšie riešenie od optima vyhľadá. Prvotné predpoklady že greedy algoritmus nájde riešenie v kratšom čase sa nepotvrdili, v niektorých prípadoch bol greedy rýchlejší, ale pri optimalizácii histórií, ktoré sú dosť malé na to, aby sme ich zobrazovali, nebudú takéto rozdiely v čase podstatné. Najväčšie množstvo času,pri najväčších testovaných históriách oba algoritmy strávili v programe EHDraw, ktorému vygenerovanie lineárneho programu zo vstupu trvalo niekoľko krát dlhšie než jeho vyriešenie CPLEXom. Na druhú stranu, riešenia ktoré nachádzal greedy boli iba málo rozdielne od optím. V praxi to znamená jeden alebo dva gény naviac v zobrazení.

Táto práca nerieši všetky problémy súvisiace so zobrazovaním evolučných histórií,

Literatúra

[1] IBM ILOG CPLEX Optimizer. urlhttp://www-01.ibm.com/software/integration/optimization/cplex-optimizer/, Last 2010.

[2] Zmasek C.M. Archaeopteryx: Visualization, analysis, and editing of phylogenetic trees. `https://sites.google.com/site/cmzmasek/home/software/archaeopteryx`, 2016. [Online; accessed 15-May-2016].

[3] Richard A. Gibbs, Jeffrey Rogers, Michael G. Katze, et al. Evolutionary and biomedical insights from the rhesus macaque genome. *Science*, 316(5822):222–234, 2007.

[4] Pawel Górecki, Gordon J Burleigh, and Oliver Eulenstein. Maximum likelihood models and algorithms for gene tree evolution with duplications and losses. *BMC bioinformatics*, 12(1):1, 2011.

[5] Albert Herencsár. An improved algorithm for ancestral gene order reconstruction. Master's thesis, Comenius University in Bratislava, 2014. Supervised by Broňa Brejová.

[6] Ján Hozza. Rekonštrukcia duplikačných histórií pomocou pravdepodobnostného modelu. Bachelor thesis, Comenius University in Bratislava, 2014. Supervised by Tomáš Vinař.

[7] Jaime Huerta-Cepas, Joaquín Dopazo, and Toni Gabaldón. Ete: a python environment for tree exploration. *BMC Bioinformatics*, 11(1):1–7, 201.

[8] R. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[9] Jakub Kovac, Brona Brejova, and Tomas Vinar. A Practical Algorithm for Ancestral Rearrangement Reconstruction. In Teresa M. Przytycka and Marie-France Sagot, editors, *Algorithms in Bioinformatics, 11th International Workshop (WABI)*,

volume 6833 of *Lecture Notes in Computer Science*, pages 163–174, Saarbrücken, Germany, September 2011. Springer.

[10] Oscar Robinson, David Dylus, and Christophe Dessimoz. Phylo.io: Interactive viewing and comparison of large phylogenetic trees on the web. *Molecular Biology and Evolution*, page msw080, Apr 2016.

[11] Petr Slavík. A tight analysis of the greedy algorithm for set cover. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 435–441, New York, NY, USA, 1996. ACM.

[12] Tomas Vinar, Brona Brejova, Giltae Song, and Adam C. Siepel. Reconstructing Histories of Complex Gene Clusters on a Phylogeny. *Journal of Computational Biology*, 17(9):1267–1279, 2010. Early version appeared in RECOMB-CG 2009.

[13] Wikipedia. Linear programming — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Linear%20programming&oldid=713865251`, 2016. [Online; accessed 10-May-2016].

[14] Wikipedia. Phylogenetic tree — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Phylogenetic%20tree&oldid=718650559`, 2016. [Online; accessed 15-May-2016].

[15] David P. Williamson and David B. Shmoys. *The Design of Approximation Algorithms*. Cambridge University Press, New York, NY, USA, 1st edition, 2011.

[16] M.J. Zvelebil and J.O. Baum. *Understanding Bioinformatics*. Garland Science, 2008.