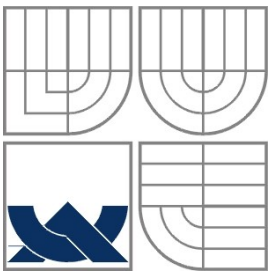
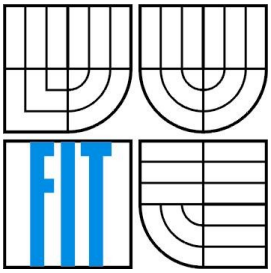


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

ANALÝZA DAT Z MIKROČIPŮ PRO ZJIŠŤOVÁNÍ GENOVÉ EXPRESE

DNA MICROARRAYS DATA ANALYSIS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR
Bc. et Bc. Tomáš Hebelka

VEDOUCÍ PRÁCE
SUPERVISOR
Ing. Ivana Rudolfová, Ph.D.

BRNO 2010

Abstrakt

Práce se zabývá analýzou dat z DNA mikročipů pomocí shlukové analýzy. Vysvětluje biologické pojmy – genová exprese a DNA mikročip. Následně obsahuje matematický a informatický popis metod shlukování a popisuje, jak tyto metody aplikovat na data z mikročipů. Dále práce obsahuje implementační detaily shlukovacích metod k-means, DBSCAN a představuje originální shlukovací algoritmus Strom++. Poté následuje popis implementace a ovládání programu. Na konec jsou zhodnoceny dosažené výsledky.

Abstract

This work concerns with data analysis of DNA microarrays by using cluster analysis. It explains biological terms – gene expression and DNA microarray. Next, it contains mathematical and informatical description of clustering methods and describes a way to apply these methods to microarrays data. Next, the work contains implementation's detail of clustering methods k-means, DBSCAN and introduces an original clustering algorithm Strom++. Then, description of implementation and application manual follow. Finally, accomplished results are evaluated.

Klíčová slova

genová exprese, DNA mikročip, shlukování, analýza dat z mikročipů, Java

Keywords

gene expression, DNA microarray, clustering, microarrays data analysis, Java

Citace

Tomáš Hebelka: Analýza dat z mikročipů pro zjišťování genové exprese, diplomová práce, Brno, FIT VUT v Brně, 2010

Analýza dat z mikročipů pro zjišťování genové exprese

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Ivany Rudolfové, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Tomáš Hebelka
26.5.2010

Poděkování

Rád bych poděkoval své školitelce Ing. Ivaně Rudolfové, Ph.D. za odborné konzultace, poskytování materiálů a vedení práce k cíli. Dále bych chtěl poděkovat svým rodičům za podporu během studia, které tato práce završuje.

© Bc. et Bc. Tomáš Hebelka, 2010
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod

Jak je již patrno z názvu, tato práce se zabývá analýzou dat z DNA mikročipů pro zjišťování genové exprese. Laikovi takový název zřejmě mnoho neřekne, proto se nejprve pokusíme blíže osvětlit termíny genová exprese a DNA mikročip. Dále čtenáře motivujeme k dalšímu studiu této problematiky tím, že se pokusíme nastínit možné využití, avšak spíše ve formě teoretických úvah, které může čtenář dále rozvíjet. Možných aplikací totiž jistě bude mnoho a ne všechny jsou dosud známy. Navíc tato práce se primárně nezabývá využitím v praxi, ale spíše matemtickou či informatickou analýzou dat, která teprve poslouží pro další využití. Konečně na konci tohoto úvodu stručně popíšeme obsah celé práce.

Pojmy genová exprese a DNA mikročip podrobněji popíšeme v následujících kapitolách, nyní se jen pokusme přiblížit čtenáře oboru, který se těmito pojmy zabývá. Informace, která kóduje vlastnosti a funkčnost všech pochodů člověka je uložena v genech, tedy v molekulách DNA v jádře buňky. Jistým mechanismem se tato informace z genů v každé buňce přepisuje do posloupnosti aminokyselin. Lineární řetízky těchto aminokyseln potom vytváří proteiny, které jsou např. iniciátory, účastníky, inhibitory oněch pochodů v člověku a stavebním materiálem buněk. I když je DNA v každé buňce stejná, pomocí jistých mechanismů každý typ buňky vytváří jinou sadu proteinů. Různým typem buňky rozumíme např. nervovou a svalovou, ale také např. zdravou a rakovinnou. Oné vlastnosti buňky, kolik kterého proteinu vyrábí na základě totožné DNA, říkáme genová exprese. V rámci genové exprese lze však zkoumat také ještě další věc. Každý protein je vyráběn na základě genu, tedy úseku DNA, četnost proteinu tak odpovídá četnosti přepisu z onoho genu. My tak budeme pod pojmem genová exprese rozumět, kolikrát (v relativní míře) se který gen přepsal do proteinů.

Genovou expresi zjišťujeme pomocí DNA čipů. Nezjišťuje se přímo relativní počet proteinů z genů, ale relativní počet meziproduktu – mRNA. To se děje přiložením obsahu buňky na DNA čip. Čip nám poté vrátí pro každý gen číslo, které udává, jak moc byl tento gen exprimován, tedy jak moc se přepisoval v dané buňce. Např. u člověka je oněch genů až 25 000, dostáváme tedy 25 000 hodnot pro jediné měření. Většinou se provádí dvě měření, jedno např. pro zdravou buňku a jedno pro rakovinnou, obě samozřejmě stejného typu v těle. Povaha čipů navíc umožňuje takové měření provést zaráz na jednom čipu tím, že se každá ze dvou buněk obarví jinou barvou. Porovnáním genové exprese, tedy oněch 25 000 čísel od zdravé a rakovinné buňky, pak zjišťujeme, co je v rakovinné buňce jiné, tedy které geny se v ní uplatňují více a které méně než ve zdravé buňce.

Víme-li, které geny se v rakovinné buňce uplatňují více než ve zdravé, můžeme se pokusit je vhodnou léčbou inhibovat, tedy omezit produkci proteinů dle jejich předlohy. Inhibovat lze celý děj na více místech, zabránit přepisu genu do proteinu, likvidovat nějaký meziprodukt nebo blokovat funkci samotného produktu – proteinu. Otázkou samozřejmě zůstává, jak takový inhibitor do nemocné rakovinné tkáně dostat. Asi by nebylo vhodné takový inhibitor rozset po celém těle, neboť by omezoval produkci onoho proteinu ve všech buňkách. Lze si však představit různé mechanismy transportu účinné léčící látky do nemocné tkáně, které jsou však možná ještě nyní ze světa sci-fi. Můžeme daný inhibitor uzavřít do nějak označeného molekulového obalu, který znemožní jeho funkci, a vpravit jej do těla. Potom pomocí nějakého scanování (např. magnetické rezonance) můžeme sledovat, zda již některé molekuly s inhibitorem doputovaly tělem do rakovinné tkáně a pokud ano, pak laserem obaly v daném místě rozbít a nechat tak působit účinnou látku – inhibitor. Lze si představit, také přímé mechanické vpravení do tkáně, nejde-li tato tkáň vyoperovat. Nejdůmyslnější metodou by asi bylo místo molekulového inhibitoru použít nějaký inhibitor na bázi vlnění, tedy inhibitovat daný protein přímo pomocí vlnění. Samozřejmě lze vymyslet spoustu dalších metod, avšak většina z nich je v dnešní době zatím nerealizovatelných.

Data získané na základě genové exprese nemusíme používat pouze pro léčbu rakovinných onemocnění. Můžeme také zjišťovat genovou expresi pro jednotlivé (zdravé) typy buněk. To nám poslouží k analýze metabolických a signálních drah v buňce, zjišťujeme tím tedy, jak organismus funguje. Nevýhodou zjišťování genové exprese je, že sice víme, které geny se jak mnoho exprimují, ale nevíme proč. Data tak můžou sloužit také jako zdroj pro další hypotézy o genech nebo potvrzení či

vyvrácení takových hypotéz. Dále zkoumáme genovou expresi jedné tkáně v závislosti na měnících se vnějších podmínkách a čase, z čehož můžeme také usoudit, jak lze onu expresi ovlivňovat.

Tím jsme vnesli trochu světla do biologických pojmů, avšak stále není vidět význam informatiky v této oblasti. Zásadní problém tkví v tom, že z každého měření máme (pro člověka) 25 000 hodnot, měření se s časem opakují a obsahují samozřejmě náhodné ale i systematický chyby. Prvním úkolem tedy zůstává určování takových chyb na základě dřívějších zkušeností nebo statistické analýzy.

Zkoumat každy gen samostatně by však bylo určitě chybné a zavádějící. Geny se navzájem ovlivňují v expresi (i když naším úkolem nyní není zjistit proč nebo jak), takže zvýšení produkce proteinů z jednoho genu sebou nese zvýšení produkce proteinů z druhého genu a tak podobně. Zejména patrné to může být při změně vnějších podmínek buňky či organismu, že se genová exprese určité skupiny genů mění podobným způsobem. Z toho lze usuzovat, že produkce proteinů z těchto genů je povzbuzována nebo utlumována stejnými činidli (inhibitory, enzymy atd.). To nám může jednak pomoci v hledání mechanismů, které ovlivňují expresi genů, pokud navíc analyzuje pořadí nukleotidů těchto genů (viz další kapitola). Druhak nám to přináší výhody ale nevýhody, že jsme schopni jedním inhibitorem, resp. enzymem utlumit, resp. povzbudit exprimace celé skupiny genů. Dalším, podstatnějším, úkolem informatika tak zůstává určit, které geny se, co se týče exprese chovají podobně a které naopak spíše ne. Uveďme příklady. Oproti zdravé buňce se v rakovinné například jistá sada genů exprimovala více, další sada mnohem více a jiná naopak méně. Podobný rozdíl může být např. mezi nervovou a svalovou (zdravou) buňkou. Stejně tak existují sady genů, které reagují podobně (tedy vyšší či nižší expresí) na změny vnějšího prostředí. Úkolem informatika je určit tyto sady, které nazveme shluky. Od toho metody pro zjišťování těchto shluků nazýváme shlukovacími metodami. Shlukovacích metod je více, všechny jsou založeny na nějakém matematickém principu. Algoritmy, které tyto metody provedou, jsou implementovány na počítačích a mohou být časově náročné, proto je dobré věnovat se vymýšlení nových efektivnějších algoritmů a jejich optimalizaci.

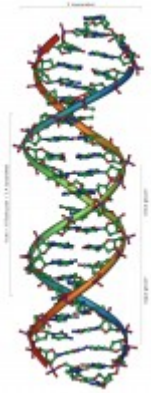
Samotným měřením genové exprese pomocí čipů se v této práci zabývat nebudeme. Data z měření budeme získávat z databází genové exprese, které jsou volně přístupné na internetu. Přinese to výhody, že máme k dispozici nepřeberné množství dat, které lze analyzovat. Data pocházejí z různých typů buněk různých organismů, my se budeme zabývat buňkami člověka. Naopak nevýhody tkví v tom, že takto nejsme přímo v kontaktu s žádnou biologickou institucí, takže výsledky naší analýzy budou čistě informatické a pro využití v praxi by ještě bylo vyžadováno jejich přezkoumání odborníkem z oblasti biologie.

Popišme nyní stručně obsah celé práce. V prvních kapitolách se zaměříme na detailnější popis genové exprese a DNA mikročipů, v rámci něhož popíšeme i způsob získávání dat z internetových databází, Potom již přikročíme k samotným informatickým tématům. Popíšeme různé metody shlukování včetně nutných matematických a informatickým pojmů. Poté se na chvíli vrátíme zpět k biologii a popíšeme možnosti předzpracování dat z mikročipů a jejich následné shlukování. Dodejme, že právě uvedená témata byla obsahem semestrální práce se stejným názvem předcházející této diplomové práci. Pro tuto práci byl však obsah mírně přepracován.

V dalších kapitole popíšeme konkrétní shlukovací algoritmy, které budeme implementovat, včetně definice jejich vstupních a výstupních dat. V rámci těchto algoritmů představíme také nový vlastní algoritmus. Po teoretické přípravě popíšeme samotnou implementaci algoritmů, ale i celé aplikace s grafických rozhraním spouštějící tyto algoritmy. Implementace bude provedena v progamovacím jazyce Java a její popis doplněn UML diagramy. V následné kapitole popíšeme ovládání výsledné aplikace. Této aplikace poté využijeme k analýze náročnosti a vhodnosti implementovaných algoritmů ke shlukování dat z mikročipů.

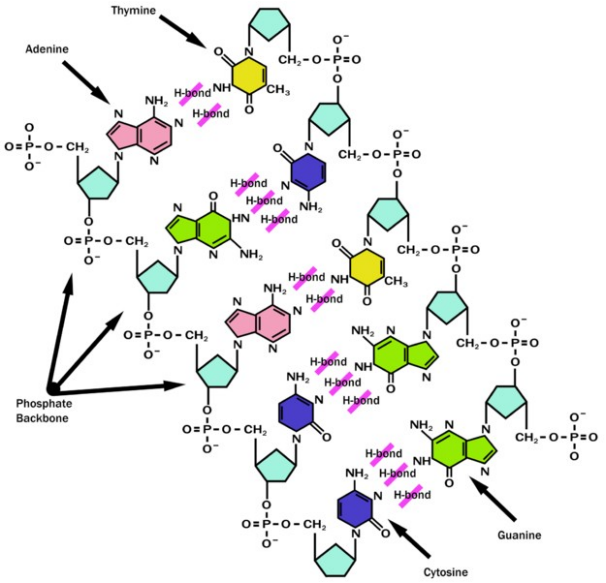
2 Genová exprese

K detailnějšímu vysvětlení a pochopení pojmu genové exprese si musíme neprve popsat celý mechanismus ukládání genové informace člověka, ale v podstatě též jakéhokoliv jiného živého organismu, a uplatňování této informace ve vlastnostech toho člověka ale i v fungování celého lidského těla. Nyní tak popíšeme mechanismus, který je obsahem tzv centálního dogmatu molekulární biologie. Následující text o centrálním dogmatu obsahuje informace získáné především ze 3. kapitoly a apendixu článku [9].



Ilustrace 1: DNA (zdroj [10])

Veškerá genová informace člověka je uložena v chromozomech, z nichž každý obsahuje právě jednu molekulu DNA. Chromozomy se nacházejí v jádru každé buňky člověka, jejich počet je v každé buňce stejný, u člověka 46 (výjimkou jsou pohlavní buňky, kde je počet poloviční). Obsah jednotlivých chromozomů, tedy ona molekula DNA, je také v každé buňce stejný. Molekula DNA má tvar dvoušroubovice, tedy dvou lineárních řetězců, které se spirálovitě otáčí kolem stejné osy a jsou navzájem spojené. Vzhled molekuly vidíme na obrázku 1. Samotná délka těchto řetězců je několik centimetrů, avšak díky složitému stočení a složení molekuly se vleze do onoho jádra o velikosti několik desítek mikrometrů. Samotný lineární řetězec se skládá z tzv. nukleotidů, molekul pospojovaných za sebou, které jsou čtyřech typů. Nukleotid se skládá z molekuly báze, která určuje typ nukleotidu, molekuly cukru deoxyribózy a fosfátového zbytku. Báze (i příslušné nukleotidy) jsou tedy čtyř typů, označovaných písmeny: A pro Adenin, C pro Cytosin, G pro Guanin a T pro Thymin. Pořadí těchto typů nukleotidů v DNA pak plně určuje veškerou genovou informaci (člověka). Schéma navazujících nukledotidů lze vidět na obrázku 2. U člověka se těchto bází nachází v jednom řetězci za sebou desetitisíce až miliony, v závislosti na chromozomu, v němž se nalézají. Celkový počet bází přes všechny řetězce DNA v jádře buňky je přibližně 6 miliard, druhá polovina z nich, jak uvidíme dále, je však již plně určena první polovinou. Pořadí bází druhého lineárního řetězce oné molekuly DNA je totiž plně dáno pořadím bazí prvního řetězce. Molekule Adeninu v prvním řetězci vždy odpovídá molekula Thyminu ve druhém, Cytosinu v prvním Guanin ve druhém a též opačně. Toho je dosaženo vazbou každé (i-té) báze prvního řetězce k i-té bázi druhého řetězce pomocí vodíkových můstků. Říkáme, že druhé vlákno DNA je komplementární k prvnímu. Genová informace člověka je tedy zakódována do oné sekvence 3 (6/2) miliard písmen A,C,G,T, přičemž se soudí, že celkem 97 % této délky žádnou genovou informaci nenese. Tyto sekvence jsou tak v DNA nadbytečné, nebo dosud nevíme, jakou mají funkci.



Ilustrace 2: Sekvence nukleotidů (zdroj [7], heslo DNA)

Vlastní geny jsou tedy obsaženy ve zbývajících 3 % bází DNA. Soudíme, že člověk má 20 000 až 25 000 genů. Jeden gen je tak dán několika blízkými lineárními posloupnostmi bází, přičemž jednotlivé geny mají různou celkovou délku a mohou se dokonce navzájem překrývat. V podstatě si tak gen lze představit jako určitý úsek DNA, z něhož ještě vystříháme menší úseky a vzniklé okraje

těchto střihů zase pospojujeme. Každý gen se tak musí nacházet uvnitř jednoho chromozomu. Lidé nemají obecně mezi sebou všechny geny stejné, jinak bychom totiž vypadali všichni stejně, měli bychom stejné vlastnosti i schopnosti a byli bychom tedy všichni sobě klony. Jeden gen člověka může mít více variant – tzv. alel, které se navzájem liší některými bázemi (např. i jen jedinou bazí). Ve skutečnosti má člověk každý gen ve své DNA dvakrát, má tedy dvě alely od každého genu, které nejsou nutně stejné. V nepohlavní buňce ke každému chromozomu totiž existuje párový chromozom, který obsahuje stejné geny. Výjimkou jsou ony pohlavní buňky, kde párové chromozomy nejsou, proto je v nich jen po 23 chromozomech. Další výjimkou je chromozom X a Y u mužů, které ani v nepohlavních buňkách nemají párové chromozomy, i když část genů (ne nutně alel) obou chromozomů je totožná.

Každému genu v DNA odpovídají nějaký protein, který se v buňce syntetizuje. Nyní popíšeme, jak probíhá přepis genu do proteinu. Dvě komplementární vlákna DNA se od sebe v místě daného genu odtrhnou. K jednomu z vláken se pak dle komplementarity syntetizuje druhé vlákno molekuly RNA. RNA má stejnou strukturu jako DNA, pouze obsahuje místo deoxyribózy jiný cukr – ribózu a místo báze Thyminu obsahuje Uracil, jeho komplementarita k Adeninu však zůstává nezměněna. Navazování vláken DNA mezi sebou dle komplementarity pomocí vodíkových můstků, resp. DNA a RNA, říkáme DNA-DNA, resp.DNA-RNA hybridizace. Po přepsání celého úseku DNA obsahující daný gen do oné RNA se RNA odtrhne, nyní se nazývá preRNA. Protože však gen není jen lineární část DNA, ale soubor lineárních částí, je třeba preRNA upravit. preRNA se skládá ze střídajících se exonů a intronů. Exony jsou její části, které odpovídají danému genu, tedy onem linárním částem DNA, zatímco introny jsou jen výplň mezi těmito částmi. Jistými mechanismy jsou tedy introny z preRNA vystřiženy, vzniklé sousední konce spojeny a vzniká tak mRNA, která již odpovídá genu. mRNA vycestovává z jádra buňky do cytoplazmy a váže se k ribozómům. V ribozómech probíhá přepis mRNA do proteinů. Každá trojice bazí mRNA, nazývaná kodon, se přepíše do jedné aminokyseliny, sousední kodon se přepíše na další aminokyselinu, která se připojí za první aminokyselinu atd. Řetízek aminokyselin tak vytváří protein. Protože pro kodon existuje 4^3= 64 možností, zatímco aminokyselin je (u člověka) jen 20, více různých kodonů odpovídá jedné aminokyselině. Také existují kodony, které znamenají začátek a konec proteinu, tzv start a stop kodón. Samotné proteiny se pak stáčí a skládají do dalších struktur, to však již není obsahem centrální dogmatu molekulární biologie a ani této práce.

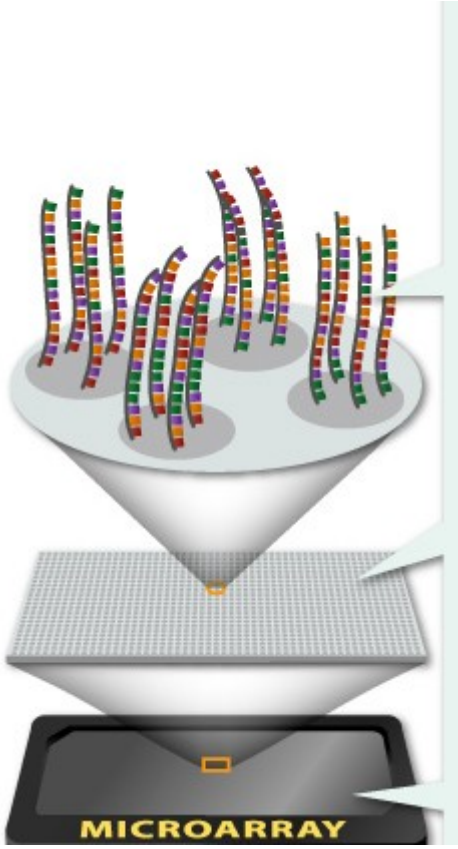
Popsali jsme tak mechanismus, jak se na základě DNA předlohy genu vytváří mRNA a následně protein. V různých typech buněk jsou přepisovány do proteinů jen některé geny, o takových genech říkáme, že byly exprimovány, nebo též že proběhla exprese genů. V bližším přiblížení se zajímáme o to, kolikrát se který gen přepsal v dané buňce (za daný čas) na protein, vyjadřujeme tedy genovou expresi číslem. Spíše než absolutně vyjadřujeme genovou expresi relativně k expresi jednoho genu nebo nějaké jiné hodnotě. Genovou expresi lze zjišťovat několika způsoby. Můžeme po nějakých časových intervalech zjišťovat relativní počet molekul mRNA v cytoplazmě buňky. Ty odpovídají exprimovaného genu, jejich celkový počet pak hodnotě exprese. Dále můžeme zjišťovat relativní počet samotných proteinů. Proteiny buňky se většinou projevují nějakou funkcí, můžeme proto také sledovat tyto funkce a jejich výsledky a z toho určovat hodnotu genové exprese. My se budeme v práci zabývat hodnotou exprese zjišťovanou pomocí mRNA, i když s drobnými změnami bude možné uvedené postupy použít i pro expresi zjišťovanou pomocí samotných proteinů.

Exprese daného genu většinou není nezávislá na expresi ostatních genů. Geny tak tvoří skupiny, které se (co se týče exprese) chovají podobně při změně vnějších podmínek buňky a plynutím času. Domníváme se, že geny v takových skupiných mají nějaký společný rys v pořadí svých nukleotidů. Jistá část genu by tak ovlivňovala to, zda se gen bude přepisovat do proteinu či nikoliv. Dalším problémem však samozřejmě zůstává zjistit, jaký je celý mechanismus zvýšení nebo snížení produkce jistého proteinu z genu.

3 Mikročipy

Po přiblížení pojmu genové exprese můžeme přikročit k metodám, jimiž ji zjišťujeme. Metod existuje více, my se však zaměříme na určování exprese pomocí DNA mikročipů. Popíšeme nyní jeho vzhled a funkci. Po objasnění funkce se podíváme na některé internetové databáze, která uchovávají výstupní data těchto mikročipů.

3.1 Vzhled a funkce mikročipů

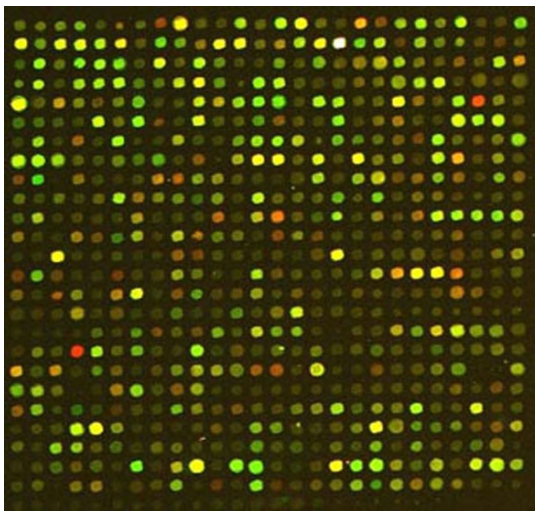


Ilustrace 3: Mikročip (zdroj [11])

Následující text vychází z 15. kapitoly knihy [1] a 5. části článku [4]. DNA mikročipy využívají pro zjišťování genové exprese mRNA, která se nachází v cytoplazmě buňky. Každá molekula mRNA je, jak víme „komplementární" kopií nějakého genu. DNA mikročipy jsou řádově centimetry široké i dlouhé destičky, na nichž je virtuální mřižka. V jednom bodě mřížky jsou k povrchu destičky „přilepeny" miliony konců navzájem totožných krátkých molekul DNA o délce desítek až stovek nukleotidů. Tyto molekuly DNA jsou však pouze jednovláknové, neboť se k nim následně bude vázat jednovláknová mRNA. Tato DNA je kopií krátkého úseku DNA z jednoho genu organismu. Celkem je takových bodů na destičce kolem 20 000 (pro člověka) a každý obsahuje krátký úsek DNA nějakého genu, přičemž v každém bodě je úsek kopií jiného genu. Dá se tak říci, že každý bod mřížky odpovídá nějakému genu člověka, proto jich je také právě kolem 20 000. Základní schéma mikročipu lze vidět na obrázku 3.

Máme-li již takto připravený mikročip, extrahujeme molekuly mRNA z dané buňky a označkujeme každou fluorescenční barvou. Metodami extrakce a značkování se zde zabývat nebudeme. Užitečné však může být značkovat tak, aby každá molekula mRNA byla označkována stejným množstvím barvy. Následně je roztok s označkovanými molekulami přeliž přes destičku mikročipu. Molekuly mRNA se poté začnou dle komplementarity pomocí vodíkových můstků navazovat na DNA v bodech mřížky. Molekula mRNA se tak naváže právě na tu molekulu DNA, které je kopií genu (jeho kouskem), z kterého byla mRNA přepsána. Protože v každém bodě mikročipů jsou miliony stejných úseků DNA, čím více stejných molekul mRNA buňka obsahuje, tím více se jich naváže na tento bod mikročipu.

Nyní přejíždíme laserem vlnové délky odpovídající barvě značkování (trochu odlišná od přesné vlnové délky barvy) jednotlivé body mřížky a snímáme intenzitu odraženého světla.



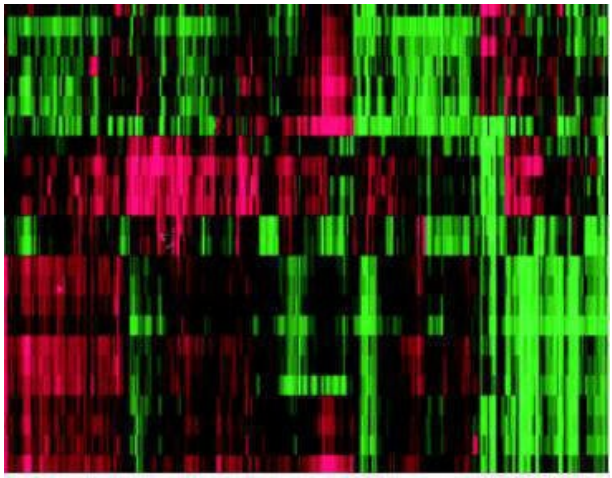
Ilustrace 4: Nasnímaný mikročip (zdroj [12])

Intenzita je tím vyšší, čím více je v daném bodě navázaných molekul mRNA, neboť každá je označkováná fluorescenční barvou. Naopak nulová bude v místě, kde se žádná mRNA nenavázala. Z hodnot intenzity pro každý bod, tedy každého genu, tak vyplývá hodnota genová exprese. Je-li daný bod černý, daný gen se vůbec neexprimoval, čím více je barevný, tím větší exprese genu probíhá. Mikročip tak po sejmutí všech intenzit vrátí na výstup oněch 20 000 čísel reprezentujících expresi každého genu. Onen obrázek různých intenzit barev lze vidět na obrázku 4.

Údaje z mikročipu nemůžeme brát se 100 % jistotou, naznačíme některé problémy. Tento odstavec vychází na rozdíl od ostatního v této sekci z 16. kapitoly knihy [1]. Různé geny mohou mít stejné ony krátké úseky DNA v bodech mřížky. To platí zvláště pro geny, jejichž úseky v DNA se překrývají. Pokud navíc neznáme polohu exonů daného genu v rámci celé DNA, mohou být krátké úseky DNA chybné (obsahující intronové části celé DNA). Uvedené problémy musíme řešit výběrem vhodných krátkých úseků DNA, je-li to možné. Dále mRNA se může navázat i na ne zcela komplementární DNA. Molekuly mRNA nemusí při svém náhodném putování v roztoku na destičce nemusí vůbec dorazit k bodu s komplementární DNA. Tyto problémy by mělo být možné řešit opakováním experimentů a jejich statistickým zpracováním – můžeme např. vzít vážený průměr hodnot exprese z navazujících experimentů. Intenzita odraženého světla od různých bodů je různá v důsledku různě odrážejícího pozadí (destičky mikročipu). To lze vyřešit měřením intenzity odrazu pozadí v blízkém okolí bodu mřížky a vztažením intenzity bodu k této intenzitě pozadí. A konečně intenzita odrazu světla od různých molekul mRNA může být různá v důsledku různé délky a obsahu oněch molekul a tím absorbování různého obsahu barviva. Tento problém můžeme vyřešit zvolením lehce odlišné metody měření genové exprese, kterou nyní popíšeme.

Máme-li vzorky buněk ze dvou tkání (např. jeden ze zdravé a druhý z rakoviné), můžeme jejich genovou expresi měřit zaráz na jednom mikročipu. mRNA z buňky první tkáně označkujeme zeleným barvivem, z druhé tkáně červeným. Poté „přelijeme“ roztoky s oběma extráty mRNA přes mikročip. mRNA obou buněk si tak nyní konkurují ve vázání se na DNA v bodech mikročipu. Domníváme se, že ona konkurence probíhá na základě množství molekul mRNA buněk, tedy např. je-li molekul mRNA daného genu více červených než zelených, bude příslušný bod mřížky červenější. Mikročip pak scanujeme lasery s dvěma vlnovými délkami a určíme intenzity pro obě barvy. Pro každý bod mřížky tak dostáváme dvě hodnoty exprese, přičemž pro další zpracování používáme typicky jejich podíl. Celková suma intenzit od zelených molekul a červených může vycházet různě, je proto vhodné obě intenzity normovat tak, aby jejich celková suma byla stejná. Pokud je pak podíl takových intenzit větší jak 1, dostáváme, že se daný gen více exprimuje v první tkáni, pokud menší jak 1, tak ve druhé.

Výsledné intenzity zobrazujeme v podobě 2D grafu, ve kterém je více měření (za různých podmínek nebo časů) dohromady. Svislá osa indexuje jednotlivé geny, vodorovná podmínky a čas měření. Samotný graf je složen z barevných pixelů, jejichž barva a intenzita je určena hodnotami exprese genu – jeho svislé souřadnice za podmínek jeho vodorovné souřadnice. Dostáváme tak pixely černé pro nulovou expresi z obou tkání, zelené pro expresi z první tkáně, červené ze druhé a žluté pro přibližně stejné exprese z obou tkání. Většina pixelů však bude obsahovat obě barvy – zelenou i červenou v různém zastoupení a budou tedy různě barevné. Samozřejmě pro další zpracování dat nepoužíváme takový barevný graf, nýbrž na místo pixelů dáváme číselné hodnoty genové exprese. Takový 2D graf můžeme vidět na obrázku 5. Výsledky



Ilustrace 5: 2D graf genové exprese (zdroj [13])

měření jsou ukládány v databázích, kterým se budeme věnovat nyní.

3.2 Databáze genové exprese

Jak jsme již řekli, z každého jednotlivého měření mikročipem dostáváme záznam o desítkách tisíc číselných hodnot – velikostí genové exprese pro jednotlivé geny. V rámci takového záznamu musí být každý gen označen jednoznačným identifikátorem a rovněž podmínky experimentu musejí být specifikovány. Mezi ty patří např. složení vnějšího prostředí, případné působení záření a čas. Dále u záznamu musí být uveden organismus a typ tkáně, z kterého je vzorek, včetně případného popisu vlastností organismu a musí být definován význam hodnot exprese – například, že jde o poměr expresí zdravé a rakovinné buňky. Samozřejmě záznam může obsahovat také další údaje o nestandardních metodách provedení měření a podobně. Měření takového vzorku probíhají s narůstajícím časem znovu, přičemž se mohou měnit vnější podmínky. Enormní množství dat z měření je proto nutné skladovat v databázích.

Z tohoto důvodu vzniklo několik internetových databází genové exprese. Zde mohou registrovaní uživatelé vkládat výsledky svých experimentů s mikročipy, neregistrovaní uživatelé mohou stahovat veřejně přístupná data, záznamů jsou zde tisíce. Databáze genové exprese bývají součástí větších bioinformatických databází, jmenujme nejznámější:

- ArrayExpress jako součást kolekce databází institutu EBI
  http://www.ebi.ac.uk/microarray-as/ae (3.1)
- GEO jako součást kolekce databází centra NCBI
  http://www.ncbi.nlm.nih.gov/geo (3.2)
- CIBEX jakožto součást databáze DDBJ
  http://cibex.nig.ac.jp (3.3)
- Stanford Microarray Database při Stanfordské univerzitě
  http://smd.stanford.edu (3.4)

Do databází jsou ukládána data jednotlivých experimentů, přičemž každý experiment musí být popsán pomocí MIAME formátu (Minimum Information About a Microarray Experiment) – minimálními informacemi o experimentu s mikročipem, jehož popis je definován konsorciem MGED na stránkách http://www.mged.org/Workgroups/MIAME/miame.html. Nepředepisuje formát dat, pouze říká, co všechno by měl záznam o experimentu obsahovat. Formát dat je však doporučen, jmenuje se MAGE-TAB a je specifikován na http://www.mged.org/mage-tab/. Samotná hrubá data z experimentu, tedy číselné hodnoty genové exprese bývají ve formátu:

- CEL popsaném na:
  www.stat.lsa.umich.edu/~kshedden/Courses/Stat545/Notes/AffxFileFormats/cel.html (3.5)
- GPR popsaném na:
  www.moleculardevices.com/pages/software/gn_genepix_file_formats.html (3.6)

Soubor formátu CEL je binární, je podobný souborům typu obrázek. Obsahuje hlavičku a poté za sebou hodnoty exprese pro každý bod mikročipu. Naproti tomu soubor formátu GPR je textový. Obsahuje hlavičku a poté ve sloupcích hodnoty expresí a názvy genů, je tedy podobný souborům typu tabulka (pro MS Excel, OO Calc). Pro lepší zorientování v databázích si ukážeme příklad downloadu dat z databáze SMD (Stanford Microarray Database).

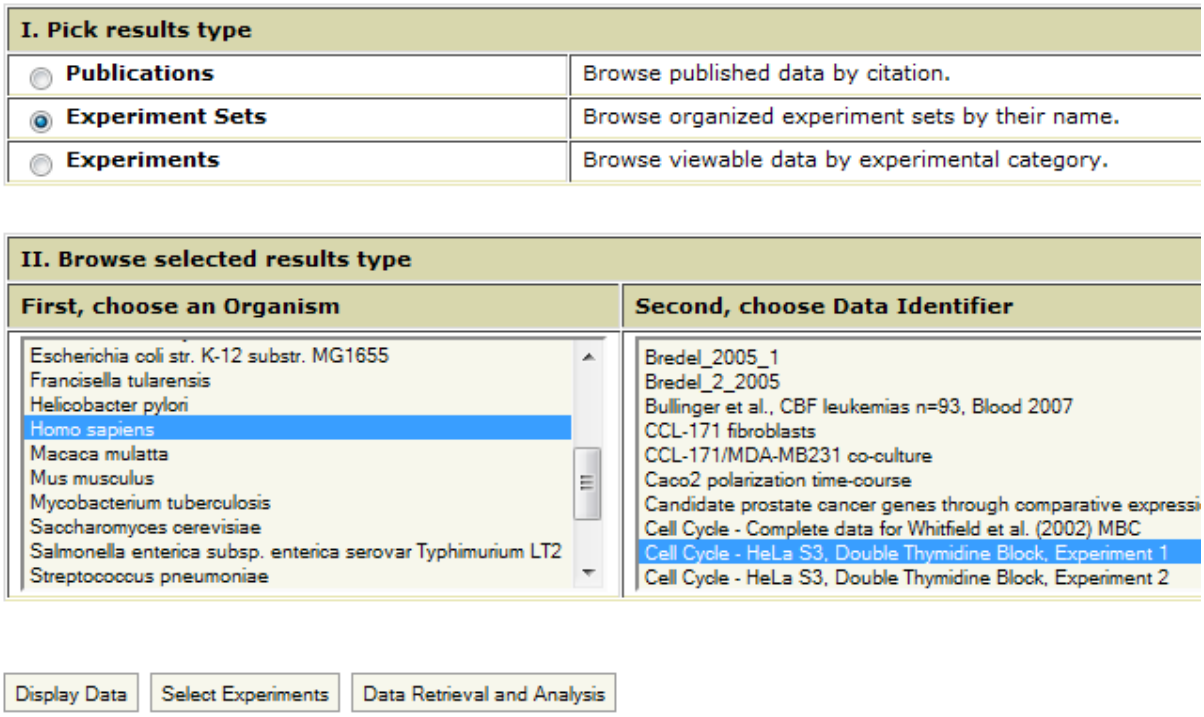
3.2.1 SMD

V SMD budeme vyhledávat hrubá data z experimentů na DNA mikročipech. Po zobrazení hlavní stránky SMD klikneme na Search v horní liště stránky. Zde si můžeme vybrat, zda chceme vyhledávat nezávisle dle více klíčů (Experimentátora, Kategorie a organismus), což je poskytováno

Advanced Search (Search By Expriments), nebo se spokojíme s Basic Search (Search By Datasets). Ten hledá podle totožných klíčů, avšak ne nezávisle, zvolíme proto tento jednodušší Basic Search. V kolonce I. Pick results type vybereme Experiment Sets, neboť chceme data z celé sady experimentů. Jedná se vlastně o tentýž experiment provedený za měnících se podmínek nebo s přibývajícím časem. V dalších kolonkách vybereme např.:

- First, choose an Organism: Homo sapiens
- Second, choose Data Identifier: Cell Cycle – HeLa S3, Double Thymidine Block, Exp. 1

Tento výběr také vidíme na obrázku 6.



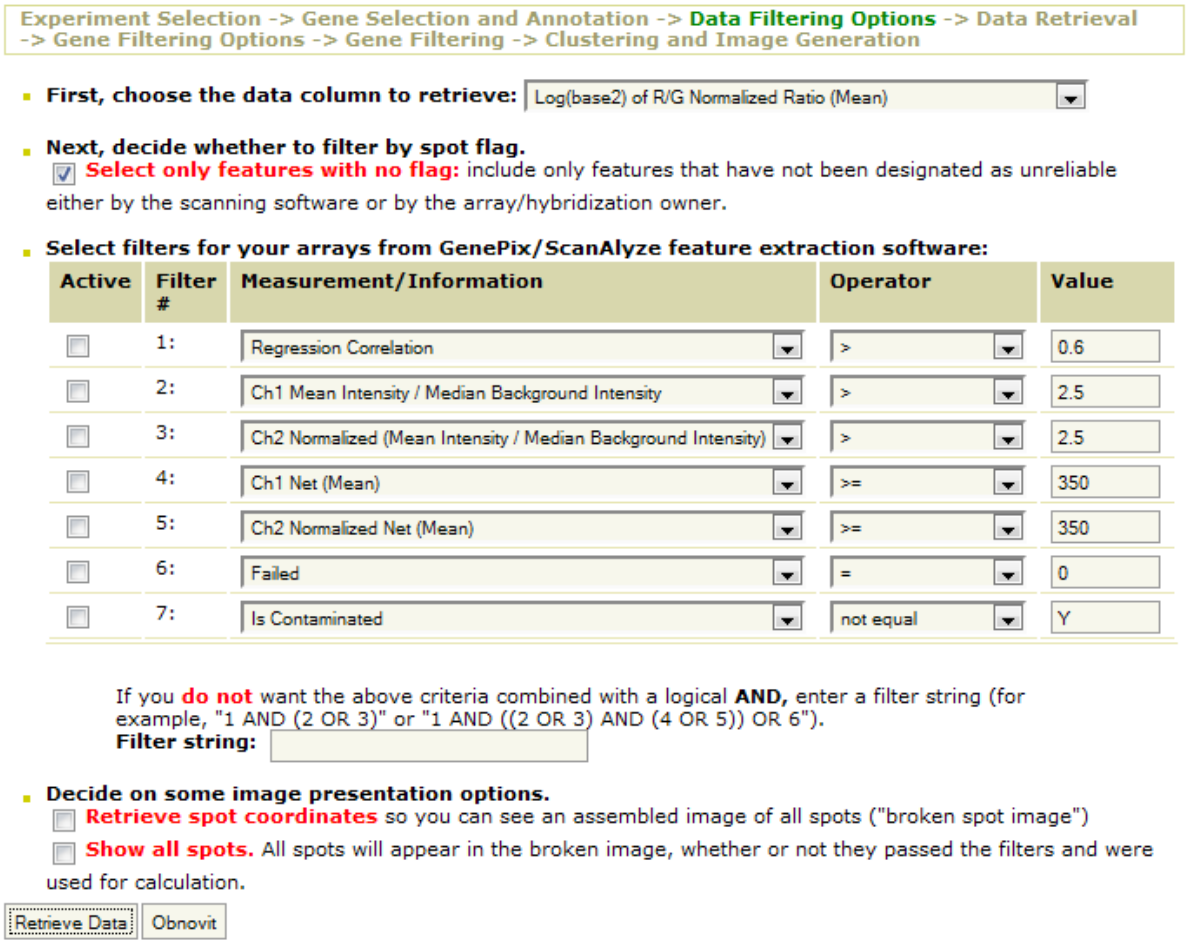
Ilustrace 6: SMD Basic Search

Po kliknutí na Display Data se nám zobrazí seznam všech experimentů se zadanými klíči. Pomocí ikonek ve sloupci Options můžeme zjistit podrobnosti o experimentu, stáhnout hrubá data, ale také vidět (analyzované) výsledky v grafech.

Pro stáhnutí dat je lépe kliknout po vybrání kolonek (viz obrázek 6) na Data Retrieval and Analysis. Následně je uživatel žádán o zadání genů, které chce zahrnout do výsledku, o to, jak mají být data a jednotlivé geny popsány a případně o redukci opakujících se záznamů. Ponecháme standardní nastavení, čímž jsou vybrány všechny geny a záznamy odpovídající jednomu genu sloučeny poměrným systémem do jednoho záznamu. Klikneme na Proceed to Data Filtering. Dále je uživatel žádán o zadání předzpracování hodnot exprese a případných filtrů. Pro předzpracování genových expresí ponecháme standardní volbu Log(base2) of R/G Normalized Ratio (Mean), což odpovídá předzpracování popsaném v části 5.2, konkrétně pomocí vzorce 5.3 na straně 23, tedy logaritmu podílu normalizovaných hodnot expresí pro rakovinnou a zdravou buňku. Co se týče filtrů, ponecháme pouze volbu Select only features with no flag, ostatní necháme nezatržené. Zadaný výběr můžeme vidět též na obrázku 7. Po kliknutí na Retrieve Data jsou (vybraná a předzpracovaná) data sloučeny z databáze do souboru, který je nabídnut ke stažení. Stažení souboru provedeme kliknutím na volbu Download PreClustering File.

Kromě stažení souboru máme také možnost nechat vybraná data analyzovat softwarem přímo

na stránkách SMD, čímž dostaneme ony barevné grafy genové exprese popsané v části 3.1 nebo dokonce shluky navzájem podobně se chovajících genů, o kterých bylo pojednáno na konci 2.kapitoly.



Ilustrace 7: SMD: Nastavení filtrování dat

My se však spokojíme se staženým souborem, který je formátu pcl, jehož popis lze nalézt na http://smd.stanford.edu/help/formats.shtml. Jedná se o tabulku, kde v prvních třech sloupcích jsou názvy, popisy a váhy genů a v dalších předzpracované hodnoty genových expresí pro ony navazující experimenty. Podrobněji je formát popsán ještě v části 6.1 na straně 26, kde se zabýváme načtením vstupních dat z (pcl) souboru do (našeho) programu.

Nyní na krátkou chvíli opustíme biologická témata a budeme se věnovat tématům informatickým, a sice shlukování.

4 Shlukování dat

Začněme vysvětlením pojmu shlukování. Představme si, že máme n p-dimenzionálních dat, kde p je libovolné přirozené číslo. Například můžeme mít n bodů v 3D prostoru, pro jednoduchost tak budeme pro data používat termín body. Chceme tyto body roztřídit do shluků, tedy navzájem diskrétních množin bodů, přičemž body uvnitř každého shluku by měly být mezi sebou navzájem co nejpodobnější a dva body z různých shluků by naopak měly být co nejméně podobné. Onu podobnost posuzujeme podle vzdálenostní funkce definované pro každou dvojici bodů, čím menší je tato vzdálenost bodů, tím jsou si podobnější. V našem příkladě v 3D prostoru tak může být vzdálenostní funkcí standardní Euklidova vzdálenost bodů a shluky pak budou tvořeny body, které jsou si navzájem blízké, takové shluky mají tedy ve svém vnitřku velkou hustotu bodů. Naopak oblasti mezi hranicemi jednotlivých shluků budou mít hustotu bodů řídkou. Celý proces roztřídění bodů do shluků nazýváme shlukováním. Zdrojem pro následující text byla zejména 7. kapitoly knihy [3].

4.1 Vzdálenostní funkce

Pro tvar a velikost shluků je podstatná volba vzdálenostní funkce d(**x**,**y**) mezi dvěma body **x**, **y** v p-dimenzionálním prostoru. Tu můžeme volit libovolně do té míry, že musí splňovat vlastnosti metriky, které jsou (**x,y,z** jsou libovolné):

- $d(\mathbf{x}, \mathbf{y}) \geq 0$
- $d(\mathbf{x}, \mathbf{x}) = 0$
- $d(\mathbf{x}, \mathbf{y}) = d(\mathbf{y}, \mathbf{x})$ (symetrie)
- $d(\mathbf{x}, \mathbf{z}) \leq d(\mathbf{x}, \mathbf{y}) + d(\mathbf{y}, \mathbf{z})$ (trojúhelníková nerovnost)

Body však mohou mít své souřadnice různých typů – intervalové, binární, nominální nebo ordinální. Ukažme si proto nyní příklady vzdálenostních funkcí pro jednotlivé typy bodů a nakonec pro body smíšených typů, tedy jejichž souřadnice se skládají z více různých typů.

4.1.1 Intervalové souřadnice

Pod pojmem intervalová souřadnice rozumíme, že je to reálné číslo z nějakého intervalu. Body p-dimenzioního prostoru označíme $\mathbf{x_1}, ..., \mathbf{x_n}$, souřadnice bodu $\mathbf{x_i}$ označíme $\mathbf{x_i^f}$ pro $f \in \{1, ..., p\}$. Obecně bývá dobré před počítáním vzdálenosti bodů souřadnice normalizovat, tedy přiřadit bodu $\mathbf{x_i}$ nové souřadnice $\mathbf{z_i^f}$ definované jako:

$$\mathbf{z_i^f} = \frac{\mathbf{x_i^f} - \mathbf{m^f}}{\mathbf{s^f}} \tag{4.1}$$

, kde:

$$\mathbf{m^f} = \frac{\sum_i \mathbf{x_i^f}}{p} \tag{4.2}$$

je aritmetický průměr a:

$$\mathbf{s^f} = \frac{\sum_i |\mathbf{x_i^f} - \mathbf{m^f}|}{p} \tag{4.3}$$

Záměrně nepoužíváme standardní statistickou odchylku, neboť ta by nám snižovala rozpoznatelnost vychýlených hodnot, tedy bodů, které leží daleko od všech větších shluků. Totiž tyto vychýlené hodnoty by příliš zvýšily standardní statistickou odchylku a tím by jejich $\mathbf{z_i^f}$ bylo menší.

Nové souřadnice vektorů **x,y** budeme však pro jednoduchost nadále značit $\mathbf{x^f}, \mathbf{y^f}$ podobně jako staré. Potom definujeme vzdálenostní funkci jako:

$$d(\mathbf{x}, \mathbf{y}) = \sqrt[q]{\sum_f |x^f - y^f|^q} \tag{4.4}$$

, kde q je libovolné přirozené číslo. Pro Euklidovu vzdálenost je tak q = 2, pro Manhattanskou q = 1.

4.1.2 Binární souřadnice

Binární souřadnice nabývají pouze hodnot 1 a 0. Pro porovnání dvou bodů **x, y** sestavíme jejich kontigenční tabulku:

		Bod x		
		1	0	Suma
Bod **y**	1	q	r	q + r
	0	s	t	s + t
	Suma	q + s	r + t	q + r + s + t

Tabulka 1: Kontingenční tabulka

V této tabulce (1) hodnota q určuje v kolika souřadnicích mají **x** i **y** shodně hodnoty 1, analogicky pro ostatní buňky tabulky.

Pokud mají obě hodnoty souřadnic (1 i 0) stejnou váhu, nazýváme souřadnice symetrickými a vzdálenostní funkci definujeme jako:

$$d(\mathbf{x}, \mathbf{y}) = \frac{r+s}{q+r+s+t} = \frac{r+s}{p} \tag{4.5}$$

Pokud je naopak hodnota souřadnice 0 nevýznamná (např. indikace, že 2 různí lidé nemají vzácnou nemoc by nemělo mít velký vliv na jejich vzdálenost), definujeme vzdálenostní funkci jako:

$$d(\mathbf{x}, \mathbf{y}) = \frac{r+s}{q+r+s} \tag{4.6}$$

4.1.3 Nominální souřadnice

Nominální souřadnicí rozumíme, že může nabývat konečného počtu vzájemně nesrovnatelných hodnot. Zacházíme s nimi analogicky jako s binárními souřadnicemi a pro dva body **x**, **y** definujeme vzdálenostní funkci jako:

$$d(\mathbf{x}, \mathbf{y}) = \frac{r}{p} \tag{4.7}$$

, kde r je celkový počet souřadnic, v nichž se **x** a **y** navzájem liší.

4.1.4 Ordinální souřadnice

Ordinální souřadnicí rozumíme, že může nabývat konečného počtu vzájemně porovnatelných hodnot. Je proto vhodné s nimi zacházet analogicky jako s intervalovými souřadnicemi.

Předpokládejme, že ony ordinální hodnoty souřadnic jsou celá čísla $\mathbf{x_i^f} \in \{1, \ldots, M_f\}$. Definujeme proto nové souřadnice $\mathbf{z_i^f}$ jako:

$$\mathbf{z_i^f} = \frac{\mathbf{x_i^f} - 1}{M^f - 1} \tag{4.8}$$

, čímž dostaneme $\mathbf{z_i^f} \in < 0, 1 >$. S těmito novými souřadnicemi pak zacházíme jako s intervalovými.

4.1.5 Smíšené souřadnice

Jsou-li souřadnice bodů **x**, **y** různých typů, definujeme jejich vzdálenostní funkci jako:

$$d(\mathbf{x}, \mathbf{y}) = \frac{\sum_f \delta^f(x, y) d^f(x, y)}{\sum_f \delta^f(x, y)} \tag{4.9}$$

, kde $\delta_f(\mathbf{x}, \mathbf{y}) = 0$, pokud $\mathbf{x^f}$ nebo $\mathbf{y^f}$schází, nebo pokud je f-tá souřadnice asymetrická binární a $\mathbf{x^f} = \mathbf{y^f} = 0$. Jinak $\delta_f(\mathbf{x}, \mathbf{y}) = 1$. $d^f(\mathbf{x}, \mathbf{y})$ je vzdálenostní funkce f-té souřadnice, definovaná dle typu této souřadnice, tedy:

- pro binární nebo nominální $\mathbf{x^f}$ je $d^f(\mathbf{x}, \mathbf{y}) = 0$, pokud $\mathbf{x^f} = \mathbf{y^f}$, jinak $d^f(\mathbf{x}, \mathbf{y}) = 1$ (4.10)
- pro intervalovou $\mathbf{x^f}$ je $d^f(\mathbf{x}, \mathbf{y}) = \frac{|\mathbf{x^f} - \mathbf{y^f}|}{\max_i \mathbf{z_i^f} - \min_j \mathbf{z_j^f}}$, kde $\mathbf{z_i^f}$, $\mathbf{z_j^f}$ probíhají všechny shlukované body prostoru (4.11)
- pro ordinální $\mathbf{x^f}$ spočítíme $\mathbf{z^f} = \frac{\mathbf{x^f} - 1}{M^f - 1}$ a s $\mathbf{z^f}$ zacházíme jako s intervalovou (4.12)

4.1.6 Vzdálenost shluků

Máme-li již definovánu vzdálenost mezi každými dvěma body, je vhodné ještě definovat vzdálenost mezi samotnými shluky. To nám také slouží pro informaci, jak jsou si jednotlivé shluky navzájem podobné, tedy blízké. Předpokládejme, že ve shluku $s \in \{1, \ldots, k\}$ jsou body $\mathbf{x_{s,i}}$ pro $i \in \{1, \ldots, n_s\}$, tedy platí $\sum_s n_s = n$. Pak pro definici vzdálenosti d(s,t) shluků s, t používáme jednu z následujících metrik:

- $d(s, t) = \min_{i,j} d(\mathbf{x_{s,i}}, \mathbf{x_{t,j}})$ (4.13)
- $d(s, t) = \max_{i,j} d(\mathbf{x_{s,i}}, \mathbf{x_{t,j}})$ (4.14)
- $d(s, t) = \frac{1}{n_s n_t} \sum_{i,j} d(\mathbf{x_{s,i}}, \mathbf{x_{t,j}})$ (4.15)
- $d(s, t) = d(\frac{1}{n_s} \sum_i \mathbf{x_{s,i}}, \frac{1}{n_t} \sum_j \mathbf{x_{s,j}})$ (4.16)

První, resp. druhá, resp. třetí metrika je založena na minimální, resp. maximální, resp. průměrné vzájemné vzdálenosti bodů shluků, čtvrtá na vzdálenosti centroidů. Nejpřirozenější definicí je patrně první a poslední, ostatní nebudeme dále používat.

4.2 Shlukovací metody

Pro shlukování dat existuje více metod. Tyto metody můžeme rozdělit do kategorií, které si postupně představíme. Ve všech následujících sekcích předpokládáme, že již máme definovanou vzdálenostní funkci. Data budeme nazývat objekty, budou p-dimenzionální a bude jich celkem n. Říkáme, že objekty ve shluku tento shluk reprezentují. Používáme stejné značení objektů ve shlucích. jako v části 4.1.6. Kromě 7. kapitoly knihy [3] vychází následující text také z 16. kapitoly knihy [1].

4.2.1 Rozdělovací metody (Partitioning methods)

U těchto metod musíme předem zadat celkový počet shluků k. Metoda nejprve přiřadí každý objekt do právě jednoho z k shluků a toto přiřazení iterativně mění tak, aby si objekty uvnitř shluku byly co nejvíce podobné a naopak objekty z různých shluků mezi sebou co nejvíc vzdálené.

4.2.1.1 k-means

Každý shluk s je zde (mimo svých objektů) reprezentován aritmetickým průměrem souřadnic objektů, v něm obsažených, který nazveme centroidem $\mathbf{m_s}$, tedy:

$$\mathbf{m_s} = \frac{1}{n_s} \sum_i \mathbf{x_{s,i}} \tag{4.17}$$

To platí jen jen pro intervalové a ordinální souřadnice, pro binární a nominální je shluk reprezentován nejčastější hodnotou souřadnic objektů ve shluku – modusem (nebudeme v popisu metody uvažovat). Tuto metodu popíšeme následujícím algoritmem, který následně vysvětlíme.

Vstupem je číslo k – počet shluků a samotné objekty.

(1) inicializace: vyber náhodně k objektů jako počáteční reprezentanty k shluků

(2) opakuj

(3) přiřaď každý objekt do shluku, jehož centroid je ze všech k centroidů shluků objektu nejbližší (dle dané vzdálenostní funkce)

(4) aktualizuj centroidy shluků (dle nově přidaných objektů)

(5) dokud se přiřazení objektů do shluků mění nebo chyba je menší než zadaná hodnota

Nejprve tedy vytvoříme k shluků, z nichž každý bude obsahovat právě jeden náhodně vybraný objekt, přičemž tyto objekty mezi sebou budou různé. Pokud je tato metoda volána z jiné metody, nemusí být počáteční inicializace náhodná, avšak volající metodou zadána. Centroid takového shluku tak bude roven obsaženému objektu. Následně každý objekt přiřadíme do shluku, který je mu nejbližší (tedy má nejbližší centroid). Toto přiřazení objektů do shluků změní centorid, který musíme přepočítat. Touto změnou se ovšem může stát, že pro některé objekty se stane bližší centroid jiného shluku, než do kterého patří, proto se celý cyklus opakuje. Může se stát, že se přiřazení objektů do shluků po každém cyklu neustále mění, proto definujeme kvadratickou chybu jako:

$$E = \sum_s \sum_i (\mathbf{m_s} - \mathbf{x_{s,i}})^2 \tag{4.18}$$

Kritériem pro pokračování v cyklu algoritmu tak může být pokles chyby pod zadanou hodnotu.

Nevýhodou této metody je, že musíme ono k předem zadat, i když ho předem neznáme. Rovněž pro různé (náhodné) inicializace můžeme dostat různé výsledky. Tyto výsledky však můžeme porovnávat pomocí kvadratické chyby. Řešením tak může být vyzkoušet více počátečních inicializací nebo více různých k od menších k větším a vybrat řešení s nejmenší chybou. Často se stává, že pro k větší než jistá optimální hodnota k_{opt} se již kvadratická chyba nelepší a vhodné bývá pak zvolit $k = k_{opt}$

$k = k_{opt}$. Algoritmus lze rovněž obohatit o identifikaci objektů, které jsou vzdáleny od všech shluků více než zadaná mez. Tyto objekty pak vytvoří základ dalších shluků a jejich celkový počet tak přesáhne k. Tím můžeme dosáhnout, že do jisté míry zvolí hodnotu k sám algoritmus.

4.2.1.2 k-medoid

V této metodě je každý shluk s reprezentován jedním svým objektem – medoidem o_s, ne tedy centroidem. Opět popíšeme metodu algoritmem:

(1) vyber náhodně k objektů jako počáteční medoidy k shluků
(2) s = 1
(3) opakuj
(4) přiřaď každý objekt do shluku s nejbližším medoidem
(5) náhodně vyber objekt o_{random}, který není medoid
(6) vypočti cenu S za prohození o_{random} za o_s
(7) pokud je S < 0, změň o_s na o_{random}
(8) s = (s+1) pro s < k, s = 1 pro s = k
(9) dokud není chyba E dost malá

Objevuje se zde termín cena, ten znamená o kolik se změní chyba E při změně medoidu reprezentující shluk s. E je definovaná jako:

$$E = \sum_{s} \sum_{i} (\mathbf{o_s} - \mathbf{x_{s,i}})^2 \tag{4.19}$$

Algoritmus nebudeme více popisovat, neboť je značně analogický s k-means v sekci 4.2.1.1.

4.2.1.3 CLARA (Clustering LARge Applications)

Je-li počet objektů příliš velký, je časová náročnost algoritmu k-medoid příliš velká. Ze všech objektů proto (náhodně) vybereme reprezentativní vzorek v a algoritmus spustíme pouze na tomto vzorku. Dostaneme tak k medoidů z vzorku v. Ostatní objekty nyní přiřadíme do shluků reprezentovaných těmito medoidy a zjistíme chybu. Je-li příliš velká, k těmto k medoidům do počtu přidáme další náhodné objekty, tak aby vytvořili další reprezentativní vzorek w. Na tomto w spustíme opět k-medoid, avšak již s onou danou počáteční volbou shluků. Takto celý cyklus opakujeme, dokud není chyba dostatečně malá. Celý algoritmus se nazývá CLARA.

4.2.2 Hierarchické metody

Hierarchické metody jsou dvojího druhu – aglomerační a dělící.

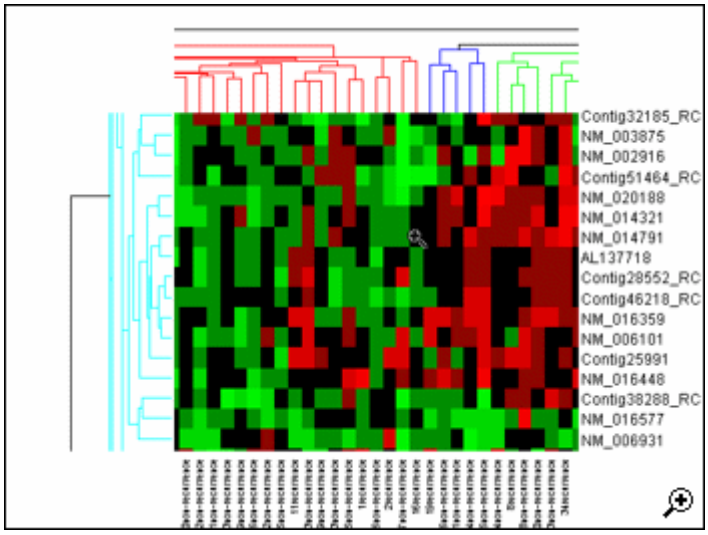
Aglomerační vytvoří z n objektů n shluků, každý shluk je tedy tvořen právě jedním objektem a objekty shluků jsou navzájem různé. Poté vždy spojí dva nejbližší shluky, vzdálenost shluků posuzuje dle některé vzdálenostní funkce z 4.1.6. Tím dostáváme čím dál větší shluky, jejichž celkový počet naopak klesá. Nevýhodou je, že spojíme-li dva shluky, je tato změna nevratná, tedy objekty z takto spojených shluků budou ve stejném shluku i při spojování dalších shluků. Můžeme však provést spojení této metody například s k-means, kdy po vytvoření jistého počtu shluků pomocí aglomerační hierarchické metody spustíme na těchto shlucích (jejichž počet, tedy k, známe) k-means s danou počáteční inicializací shluků (nikoliv tedy náhodnou). Tím dosáhneme toho, že objekty, které byly již při prvních iteracích hierarchické metody sloučeny do jednoho shluku se mohou opět rozdělit do různých shluků.

Dělící metody postupují opačně. Vytvoří jeden velký shluk tvořený všemi objekty a ten postupně dělí do více shluků. Máme-li takto k shluků (na počátku k = 1), metoda vybere dle jistého kritéria shluk, který rozdělí na dva a tím vzniká k+1 shluků. Rozdělení shluku na dva shluky lze provést některou jinou shlukovací metodou. Také kritérií pro výběr, který shluk se bude rozdělovat je více, uveďme některé z nich:

- shluk má ze všech nejvíce objektů, je-li takových více, tak náhodný z nich
- shluk při svém rozdělení nejvíce sníží celkovou chybu – výpočetně náročné
- vzdálenost dvou shluků vytvořených z vybraného shluku je maximální (ze všech vzdáleností dvojic shluků vytvořených ze shluků) – výpočetně náročné
- shluk, jehož minimální nebo průměrná vzdálenost dvou vlastních objektů je maximální (ze všech shluků)

Nevýhody dělící metody jsou totožné jako nevýhody aglomerační, rozdělení shluku je nevratné, lze však rovněž dělící metodu zkombinovat např. s k-means a odstranit tak tuto nevýhodu.

Samotný výsledek hierarchického shlukování znázorňujeme rovněž pomocí dendrogramu. Ten obsahuje historii toho, jak byly shluky rozdělovány nebo spojovány a lze z něho tedy vyčíst shluky vzniklé po libovolném počtu iterací algoritmu. Rovněž délka čáry dendrogramu mezi jednotlivými uzly, které reprezentují shluky, odpovídá vzdálenosti těchto shluků. Můžeme tak z dendrogramu vyčíst, kdy již mají shluky příliš malou nebo velkou vzdálenost mezi sebou a získat tak optimální počet iterací pro hierarchickou metodu. Příklad dendrogramu pro mikročip je na obrázku . Dendrogram nalevo odpovídá hierarchickému shlukování genů, nahoře vzorků.



Ilustrace 8: Dendrogram mikročipu (zdroj [14])

Pro implementaci těchto metod stačí uvedené popisy. Uveďme však některé pokročilejší implementace metod.

4.2.2.1 BIRCH (Balanced Iterative Reducing and Clustering using Hierarchies)

Jedná se o dělící hierarchickou metodu, která udržuje shluky v listech vyvážené stromové struktury. Vnitřní uzly potom odpovídají původním shlukům, z kterých dělením vznikly jeho potomci. Každý uzel obsahuje agregační informace o bodech v celém svém podstromu, a sice počet bodů, sumu jejich souřadnic (každou zvlášť) a sumu kvadrátů souřadnic (všech dohromady). Maximální počet potomků jednoho uzlu je pevně daný. Nyní se podívejme jak pracuje samotný algoritmus.

Objekty jsou po jednom přidávány do počátečního jednoho shluku. Přesáhne-li jeho poloměr (minimální koule, do které se shluk vleze) pevně zadanou mez, je shluk rozdělen na dva. Tyto dva shluky vytvoří dva listy, které obsahují seznam svých objektů, a jejich předkem se stane počáteční shluk. Další objekty jsou přidány do takového listu, kde způsobí nejmenší nárůst poloměru. Přesáhne-li opět poloměr listu zadanou mez, je list rozdělen na dva, případně je list sloučen se svým sourozencem a poté rozdělen na dva. To může vést k nárustu potomků předka, který pak může být také rozdělen. Hodnota poloměru je postupně algoritmem volena od menší k větší tak, aby se seznam uzlů s agregačními hodnotami vlezl do hlavní paměti počítače.

4.2.2.2 CURE (Clustering Using REprezentatives)

Jedná se o aglomerační hierarchickou metodu. Podstatou algoritmu je reprezentace shluku nikoliv jen pomocí jediného centroidu nebo medoidu ale pomocí pevně daného počtu r objektů. To přináší výhody, že shluky nemusejí mít nutně kulovitý konvexní tvar jako u předchozích metod. Je-li však počet objektů ve shluku menší než r, jsou reprezentanty všechny. Vzdálenost objektu od shluku je pak posuzována dle minimální vzdálenosti k některému z r reprezentantů shluku. Vzdálenost shluků mezi sebou je posuzována jen podle jejich reprezentantů, avšak opět je užito postupů z sekce 4.1.6 . Nyní popíšeme samotný algoritmus a poté jeho obměnu pro větší databáze.

Na začátku každý objekt vytvoří samostatný shluk. Poté jsou aglomerativně spojovány vždy shluky s nejmenší vzdáleností. Při spojování dvou větších shluků do jednoho je potřeba z 2r reprezentantů vybrat zase jenom r. Spočítá se tedy celkový centroid spojeného shluku a z 2r reprezentatnů se vybere r, které jsou tomuto centroidu nejblíže. Postup je zastaven je-li dosaženo požadovaného počtu shluků nebo kvadratická chyba je již dostatečně malá.

Algoritmus lze obměnit pro použití v rozsáhlých databázích. Z celé databáze objektů je vybrán reprezentativní vzorek. Ten je pak ještě rozdělen na části. Na každé části je poté spuštěn algoritmus k vytvoření počátečních shluků. Je-li počet shluků v každé části již dostatečně malý, je algoritmus dále spuštěn s těmito shluky jako počátečními. Tím se vytvoří shluky z celého vzorku. Ostatní objekty jsou poté přidány do shluků na základě minimální vzdálenosti ke shluku.

4.2.3 Metody založené na hustotě (Density-based methods)

Podstatou těchto metod je vytvářet takové shluky, že pokud vezmeme dva objekty z téhož shluku, lze se z jednoho dostat do druhého přes oblasti shluku jisté minimální hustoty. Uvedeme dvě konkrétní metody.

4.2.3.1 DBSCAN

K popisu metody si musíme nejprve definovat několik použitých pojmů. Předpokládáme, že ε, m jsou pevně daná čísla.

- ε-okolí objektu: objekty v okolí objektu, jejichž vzdálenost od objektu není větší než ε
- objekt jádra: objekt, v jehož ε-okolí je alespoň m objektů (včetně sebe sama)
- objekt r nazveme „přímo hustotně-dosažitelný“ od objektu q, pokud q je objekt jádra a r leží v jeho ε-okolí
- objekt r je „hustotně-dosažitelný“ od objektu q, existuje-li řetěz objektů q, q1, …,qx, p, v němž jsou všechny q,q1,...qx objekty jádra a každý objekt z řetězu objektů leží v ε-okolí svých sousedů (z řetězu), tedy q1 je přímo hustotně-dosažitelný z q, q2 z q1 atd.
- objekty r,t jsou „hustotně-spojené“, existuje-li objekt jádra q tak, že r i t jsou hustotně-dosažitelné z q

Po zadání ε, m algoritmus nejprve určí, které objekty jsou objekty jádra. Ke každému objektu jádra přitom uloží odkaz na jeho sousedy v ε-okolí. Pro zjišťování objektů jádra je dobré mít objekty uložené v struktuře s prostorovým indexem, aby bylo snadné hledat sousední objekty. Vhodné je tak mít např. seřazeny jednotlivé souřadnice objektů nebo použít hashování sítě buněk, v nichž leží objekty nebo použít R-strom. Každý objekt jádra je nyní základem svého shluku. Odkazy na sousedy vytváří orientovaný graf, pomocí prohledávání grafu do hloubky nebo šířky pak vytvoříme jednotlivé shluky.

4.2.3.2 DENCLUE

U této metody počítáme hustotu všech bodů (ve kterých nemusí být žádný objekt) prostoru databáze. Hustota je počítána jako součet funkcí vlivů od jednotlivých objektů **x**, což je příspěvek hustoty od jednoho objektu k celkové hustotě. Označme tuto funkci f^x, tedy příspěvek k hustotě v bodě **y** od objektu **x** je $f^x(y)$. Celková hustota $f(y)$ v bodě **y** je potom:

$$f(y) = \sum_i f^{x_i}(y) \tag{4.20}$$

kde x_i probíhá všechny objekty databáze. Samotná funkce f^x bývá nejčastěji definována jako:

$$f^x(y) = f(x,y) = \begin{cases} 0 & \text{pokud } d(x,y) > \sigma \\ 1 & \text{pokud } d(x,y) \leq \sigma \end{cases} \tag{4.21}$$

kde σ je pevně dané, nebo jako:

$$f^x(y) = f(x,y) = e^{-\frac{d(x,y)^2}{2\sigma^2}} \tag{4.22}$$

Po určení funkce f jsou numericky spočítány její lokální maxima. Ty jsou nalezeny „šplháním“ z náhodně zvolených startovacích bodů ve směru gradientu funkce. Tyto maxima pak tvoří základ každého shluku. Shluk je pomocí numerických metod zvětšován tím, že se najde okolí maxima takové, že v celém okolí neklesne hustota pod jistou mez. Příslušnost objektu do shluku je potom posuzována dle jeho příslušnosti do těchto okolí.

4.2.4 Metody založené na modelech využívající neuronové sítě

Tyto metody předpokládají, že každý shluk odpovídá nějakému (matematickému) modelu. Snaží se tak najít takové rozdělení do shluků, které nejvíce odpovídá předpokládanému modelu. Z široké škály takových metod uvedeme pouze metody založené na přístupu neuronových sítí.

4.2.4.1 SOM

V tomto algoritmu, podobně jako v k-means, je třeba nejprve vybrat počet shluků k a jejich počáteční inicializaci. To přináší stejné nevýhody, jaké jsou popsány v sekci 4.2.1.1 o k-means. Každý shluk je zde reprezentován jedním neuronem, což je uměle dotvořený objekt v prostoru databáze, a objekt přísluší do shluku toho neuronu, jemuž je dle zadané vzdálenostní funkce nejblíže.

Na počátku je vytvořeno k neuronů, které v prostoru objektů uspořádáme do mřížky. Každý neuron je „imaginárně“ propojen se všemi ostatními, nejedná se tedy o klasickou neuronovou síť s vrstvami. Volba rozměru mřížky i její tvar je rovněž volena uživatelem, pro k = 12 tak můžeme mít např. mřížku 3x4 nebo 2x6. Poté spustíme epochy iterací. V každé epoše je provedeno právě n iterací, v nichž vystupuje vždy právě jeden objekt (pro každou iteraci v epoše jiný) a neurony. V každé iteraci jsou neurony přiblíženy o nějakou hodnotu směrem k vkládanému objektu, přičemž v počátečních iteracích (v první epoše) jsou přibližovány všechny neurony, s přibývajícími iteracemi (tedy i epochami) jen ty nejbližší neurony, případně jen jeden nejbližší neuron. Rovněž hodnota o kterou jsou neurony posunuty je (relativně) největší pro nejbližší neuron od objektu a postupně se zmenšuje pro vzdálenější neurony. Uvedený popis tak můžeme zapsat vztahem:

$$S_m(i+1) = S_m(i) + \tau(i) h_{m,near\_j}(i)(\mathbf{x_j} - S_m(i)) \tag{4.23}$$

, kde $S_m(i)$ je poloha m-tého neuronu v i-té iteraci (iterace číslujeme souhrně nezávisle na epochách), $\mathbf{x_j}$ poloha objektu, s kterým se v iteraci pracuje. Z tohoto vztahu je vidět, že každý neuron je skutečně ze své původní pozice $S_m(i)$ posunut k objektu $\mathbf{x_j}$, tedy ve směru vektoru $(\mathbf{x_j} - S_m(i))$ $\tau(i)$

$(\mathbf{x_j} - S_m(i))$. $\tau(i)$ je zde konstanta, která se s rostoucím i zmenšuje, jde tedy o ono zmenšování vlivu objektů při vyšších iteracích. $\tau(i)$ většinou měníme skokově mezi konstantními hodnotami, tyto hodnoty jsou typicky 0.01 a 0.001. Konečně $h_{m,near\_j}(i)$ je hodnota závislá pozici m-tého neuronu $S_m(i)$ a pozici neuronu, který je nejblíže k $\mathbf{x_j}$, tedy pozici S_{near_j} . $h_{m,near\_j}(i)$ je rovno 1 pro neuron s pozicí S_{near_j}, pro vzdálenější neurony klesá k 0. Typicky používáme funkci:

$$h_{m,near\_j}(i) = \begin{cases} 0 & \text{pokud } d(S_m(i), S_{near\_j}) > 3\sigma(i) \\ e^{-\frac{d(S_m(i),S_{near\_j})^2}{2\sigma(i)^2}} & \text{jinak} \end{cases} \tag{4.24}$$

, kde se $\sigma(i)$ s rostoucím i zmenšuje. $h_{m,near\_j}(i)$ vystihuje ono zmenšování vlivu objektu $\mathbf{x_j}$ na vzdálenější neurony a rovněž s vyšším počtem iterací klesá ještě strměji od pozice S_{near_j} .

Výhodou algoritmu SOM je, že neurony, které jsme zvolili jako bližší v inicializaci, budou mít též shluky bližší, podobnější. Z pohledu na výsledné shluky tak můžeme určit, které počáteční neurony byly zbytečné, neboť příslušející shluky jsou příliš podobné a je tedy možno je navzájem sloučit. Míru podobnosti objektů v jednom shluku měříme opět dle kvadratické chyby (spočítané jen s objekty daného shluku). Dle ní můžeme naopak poznat, zda by nebylo dobré daný shluk rozdělit, zvolit tedy více počátečních neuronů v dané oblasti.

4.2.4.2 SOTA

SOTA je obměnou algoritmu SOM, který se snaží poprat s jeho nevýhodami. Jedná se o skloubení hierachických metod shlukování s algorimem SOM, svým konceptem tak připomíná metodu BIRCH (viz 4.2.2.1). Shluky jsou v SOTA také reprezentovány neuronem. Neurony jsou zde uloženy v uzlech binárního stromu. To nám udává hierarchii neuronů, neznamenáto, že by v prostoru databáze vytvářely struktur stromu. Objekty zde také přisouvají neurony směrem k sobě, avšak pouze ten nejbližší a případně jeho sourozence a přímého předka v stromu.

Na začátku je vytvořen neuron – kořen stromu a umístěn do centroidu celé databáze objektů. K němu jsou vytvořeny dva potomci – neurony, které jsou rovněž umístěny do tohoto centroidu. Dále však budeme algoritmus popisovat pro jakkoli veliký strom neuronů. Následně je spuštěna etapa iterací s každým objektem databáze (podobně jako v SOM). V každé iterace hledáme opět nejbližší neuron, avšak pouze mezi neurony v listech stromu. Při tom pro novou pozici $S_{near\_j}(i+1)$ tohoto nejbližšího neuronu bude platit:

$$S_{near\_j}(i+1) = S_{near\_j}(i) + \tau_{near\_j}(i)(\mathbf{x_j} - S_{near\_j}(i)) \tag{4.25}$$

$\tau_{near\_j}(i)$ opět klesá s rostoucími etapami, pro počáteční etapy bývá 0.01. Pokud je sourozenec tohoto neuronu také list, pak se sourozenec i jeho předek posunou stejným způsobem, avšak s menšími τ, např. $\tau_{sibling}(i) = 0.001$ a $\tau_{parent}(i) = 0.0005$. Není-li sourozenec list, žádný další neuron se neposouvá. Stane-li se, že je oba sourozenci jsou nejbližší, je z nich vybrán vždy jen ten samý jeden, např. levý sourozenec.

Po proběhnutí epochy je pro každý neuron m v listě spočítána průměrná absolutní chyba R_m

$$R_m = \frac{1}{||S_m||} \sum_{j \in S_m} d(\mathbf{x_j}, S_m) \tag{4.26}$$

kde $||S_m||$ značí počet prvků shluku m-tého neuronu. Neuronům, jejichž R_m přesáhne zadanou mez, jsou vytvořeny potomci s pozicí inicializovanou na pozici předka a je spuštěna další etapa.

Algoritmus vede k vytvoření hierarchie shluků – každé patro stromu reprezentuje jeden výsledek shlukování. Navíc uživatel nemusí zadat předem počet shluků a výslednou hierarchii shluků může posoudit podle dendrogramu, jak je uvedeno v sekci 4.2.2 o hierarchických metodách.

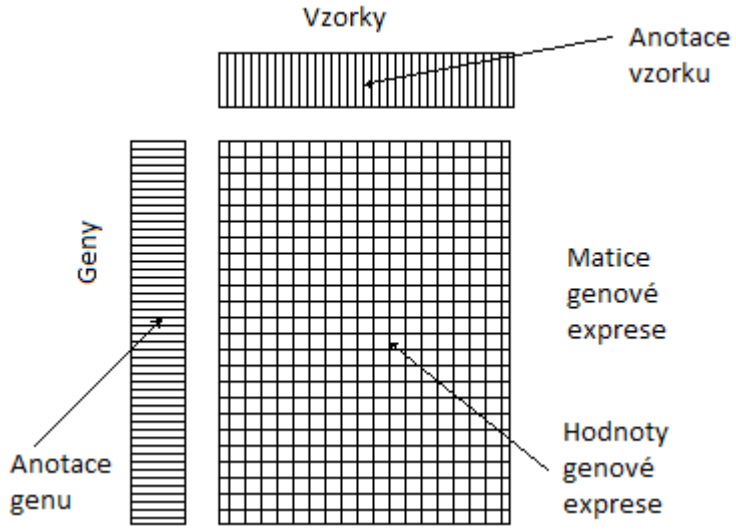
Tímto končí výčet algoritmů pro shlukování. Nyní se pokusíme uvedenou informatickou teorii skloubit s předchozí biologií, budeme vytvářet shluky výstupních dat mikročipů.

5 Shlukování dat z mikročipů

Jak již nadpis naznačuje, stále se budeme věnovat shlukování dat, avšak jako vstupní objekty nám budou sloužit výstupní data z mikročipů, o nichž pojednáme hned v 1. části této kapitoly. Následující text vychází zejména z 16. kapitoly knihy [1] a z 5. části článku [4].

5.1 Výstupní data z mikročipů

Výstupní data z mikročipů určují hodnoty expresí pro jednotlivé geny v jednom nebo dvou typech buněk, přičemž výstupem může být jedna hodnota exprese pro každý gen, měříme-li pouze na jednom typu buňky, nebo dvě hodnoty, měříme-li zaráz dva typy buněk – např. rakovinnou a zdravou obarvené dvěma různými barvivy. V rámci jednoho experimentu provádíme více měření se stejným typem buněk s odstupem času za případně se měnícího prostředí (např. aplikací léčivé látky v okolí buňky). Vstup každého jednotlivého měření pak nazýváme vzorek, dostáváme tak jednu či dvě hodnoty expresí pro každý gen a vzorek. Vzorek není charakterizován jen hodnotami expresí genů, nýbrž také specifikací podmínek v okolí buňky, příp. tkáně, jejíž je buňka součástí, času a údajů o organismu, z něhož je tkáň s buňkou, např. věk, pohlaví nebo aktuální nemoci. Základní schéma dat z mikročipů lze vidět na obrázku 9, kde řádky reprezentují různé geny a sloupce různé vzorky.



Ilustrace 9: Matice genové exprese (zdroj [4])

Samotným vstupem pro další analýzu tak bude matice (dvojic) hodnot expresí pro každý gen a vzorek. Nebude nás zajímat chování systému (buňky) v jednom konkrétním okamžiku jako spíše dynamika tohoto chování. Úkolem tak bude zjistit, exprese kterých genů se chovají podobně s měnícími se podmínkami nebo přibývajícím časem. Nebudou nás přitom zajímat absolutní hodnoty expresí, ale spíše poměry expresí jednotlivých vzorků, tedy např. se zajímáme o to, zda se u dvou genů shodně zvýšila nebo snížila dvojnásobně exprese, ačkoliv hodnoty absolutních hodnot expresí těchto dvou genů v jednom vzorku byly různé. Tyto podobné geny tedy tvoří shluky, které identifikujeme pomocí shlukovacích metod. Je pravděpodobné, že takové geny jsou součástí stejných metabolických nebo signálních cest v daném organismu. Dá se tak očekávat, že budou shodně reagovat na podání léčiva nebo indikovat nějakou nemoc. Naměřené hodnoty expresí však mohou být lehce deformovány různým poločasem rozpadu molekul mRNA od různých genů. Ve vzorku tak může přetrvávat mRNA z genů, které se již neexprimují. Nyní však musíme před samotným shlukováním dat provést jejich úpravu, jejíž kroky si nyní popíšeme.

5.2 Úprava dat z mikročipů

V dalším budeme o dvou hodnotách expresí stejného genu mluvit jako o červené a zelené. Označení vychází z použitého barviva při měření, červené tak např. odpovídá rakovinná buňka a zelené zdravá. Nyní se budeme mimo jiné snažit odstranit chyby popsané v části 3.1 o funkci mikročipů. Po uvedení postupu (transformace), jak se zbavit daného typu chyby, vždy předpokládáme, že do následné transformace již vstupují transformovaná data bez předchozích chyb, není-li uvedeno

jinak.

Naměřené hodnoty intenzit odrazů od bodů mikročipu jsou zatíženy chybou – odrazivostí pozadí každého bodu (destičky) nezávislé na množství barviva v bodě. Od intenzity každého bodu tak odečteme intenzitu odraženého světla z okolí bodu (kterou tak musíme také změřit), nebo průměrnou hodnotu intenzit z bodů, jejichž exprese je velmi nízká. Tím se stanou některé intenzity zápornými nebo nulovými, pro potřebu dalšího zpracování všechny (upravené) intenzity menší než pevně zadaná kladná mez změníme na tuto mez.

Dalším typem chyby je náhodná statistická chyba ε hodnoty exprese s normálním rozložením, nulovou střední hodnotou a standardní odchylkou σ_ε. Tu můžeme odstranit průměrováním hodnot expresí vzorků za stejných podmínek, ty však většinou nemáme k dispozici a tuto chybu prostě ignorujeme. S rostoucí expresí rovněž roste chyba, jedná se o multiplikativní chybu e^η s nulovostí střední hodnoty η a její standardní odchylkou σ_η. Uvedené chyby tak způsobí, že změřená hodnota exprese X se bude od skutečné χ lišit vztahem:

$$X = \chi e^\eta + \varepsilon \tag{5.1}$$

Pro změřenou intenzitu I, která je lineární funkcí X, dostáváme:

$$I = \beta_1 + \beta_2 \chi e^\eta + \varepsilon \tag{5.2}$$

Hodnoty β_1 se zbavujeme odečtením pozadí, ε ignorujeme a zbývající konstanty se odstraní následnými transformacemi.

Jak jsme již naznačili v úvodu kapitoly, nezajímají nás absolutní hodnoty expresí, nýbrž poměry expresí pro různé podmínky nebo čas. Hodnotu exprese genu tak normujeme vydělením ethalonem exprese genu, což může být průměr hodnot expresí genu přes všechna měření našeho experimentu. To vede k odstranění dvou nedostatků, jednak různé schopnosti vázání barviva mRNA od daného genu, druhak podobné hodnoty exprese pro stejný typ buňky a stejné podmínky v různých experimentech, kdy celkový obsah mRNA v buňce může být různý. Taková normalizace vede rovněž k odstranění chyby $\beta_2 e^\eta$, neboť tyto chyby se vykrátí.

Máme-li dvě hodnoty exprese pro každý gen a vzorek, postupujeme jinak. Nedělíme exprese ethalonem, ale nazvájem mezi sebou, většinou červenou děleno zelenou. Tím mlčky předpokládáme stejnou hodnotu ethalonu pro gen a vzorek pro obě barviva. Jsou-li intenzity od všech bodů mikročipu pro jedno barvivo systematicky menší než pro druhé, hodnota ethalonu se bude lišit, avšak tento problém vyřeší následné transformace. Jinak předpoklad podobnosti ethalonů není tak špatný pro stejný gen a vzorek, ethalony však mohou být jiné proto, že máme dva typy buněk. Tím docíleme toho, že geny, jejichž exprese se (za daný čas v daných vzorcích) zvýšila dvakrát nemusí být přiřazeny do stejného shluku, neměly-li stejný počáteční poměr intenzit. To může být výhoda i nevýhoda.

Jelikož nás zajímají spíše poměry expresí genů (pro různé vzorky), je nesmyslné porovnávat absolutní hodnoty expresí, například srovnání hodnot ½ a 1 by vycházelo jiné než hodnot 1 a 2. Na tom nic nemění to, že máme expresi normovanou k ethalonu nebo počítáme s poměrem červené a zelené exprese. Budeme proto již normalizovaná data (poměry červené a zelené exprese) logaritmovat. Tím dosáhneme rovnoměrnějšího rozložení poměrů expresí kolem 0. Pro nestejné intenzity odrazivosti jednotlivých barviv můžeme dostat exprese rovnoměrně rozložené kolem jiné hodnoty než nuly, korekci dosáhneme jejím odečtením. Ukazuje se (centrální limitní větou a hodnotami měření z více experimentů), že logaritmus poměrů expresí má normální rozložení dle četnosti, alespoň v oblastech kolem 0. Ve vzdálenějších oblastech tomu tak není.

Většinou pro analýzu genové exprese používáme právě ony logaritmy poměrů rozdílu intenzit a pozadí, celkově tedy hodnoty y:

$$y = \log_2 \frac{I_{red} - I_{background_{red}}}{I_{green} - I_{background_{green}}} - c \tag{5.3}$$

, kde c je hodnota korekce, aby průměrná hodnota y vycházela 0. Může se však stát, že rozložení hodnot y v závislosti na $\log_2(I_{red} - I_{background_{red}})(I_{green} - I_{background_{green}})$, tedy logaritmované střední hodnotě intenzity (střední ve smyslu pro červenou a zelenou) stále není rovnoměrné kolem 0, ale vykazuje zakřivení. Oblasti zakřivení jsou proto rozděleny na menší oblasti a v nich je spočítána lineární regrese závislosti y na log. střední hodnotě intenzity. Hodnota y je pro další výpočet následně upravena tak, že je rovna vzdálenosti původní hodnoty y od regresní přímky. Nyní přikročme k definici vzdálenostní funkce takto transformovaných dat.

5.3 Vzdálenost genů nebo vzorků

Transformované hodnoty expresí jsou intervalového typu, proto použijeme vzdálenostní funkci definovanou v části 4.1.1 o intervalových souřadnicích. Z možných hodnot q používáme většinou q = 2, tedy Euklidovskou metriku. Máme však dvě možnosti, které věci shlukovat, a sice geny nebo vzorky. Předpokládejme proto, že transformovaná hodnota exprese pro i-tý gen a A-tý vzorek je $X_{i,A}$. Pro shlukování genů budou objekty prostoru databáze tvořeny body X_i, přičemž souřadnicemi těchto bodů bude $X_{i,A}$ pro A probíhající všechny vzorky. Pro shlukování vzorků budou objekty tvořeny body X_A se souřadnicemi $X_{i,A}$ pro i probíhající všechny geny. Souřadnicí vzorků X_A však mohou být též další údaje jako je věk, pohlaví organismu a podmínky okolního prostředí, ty nemusejí být intervalového typu, označíme je $Y_{j,A}$. Pro vzdálenost bodů X_i, resp. X_A, tak platí:

$$d(X_i, X_j) = \sqrt{\sum_A (X_{i,A} - X_{j,A})^2} \tag{5.4}$$

$$d(X_A, X_B) = \sqrt{\sum_i (X_{i,A} - X_{i,B})^2 + \sum_j (Y_{j,A} - Y_{j,B})^2} \tag{5.5}$$

Hodnoty $X_{i,A}$ samozřejmě můžeme před počítáním vzdálenosti normalizovat standardní odchylkou tak, jak je uvedeno v části 4.1.1.

Chceme-li zjišťovat korelaci mezi expresemi dvou vzorků nebo dvou genů, je výhodné použít jinou metriku, a sice Pearsonovu nebo Mahalanobisovu. Zajímá nás tak, exprese kterých genů narůstají nebo klesají shodným způsobem (co se týče poměru expresí), což odstraní uvedené nevýhody logaritmu poměru červené a zelené exprese. Pearsonova vzdálenost mezi vzorky je definována jako:

$$d(X_A, X_B) = \frac{1}{n-1} \sum_i \left(\frac{X_{i,A} - \bar{X}_A}{s_A} \right) \left(\frac{X_{i,B} - \bar{X}_B}{s_B} \right) \tag{5.6}$$

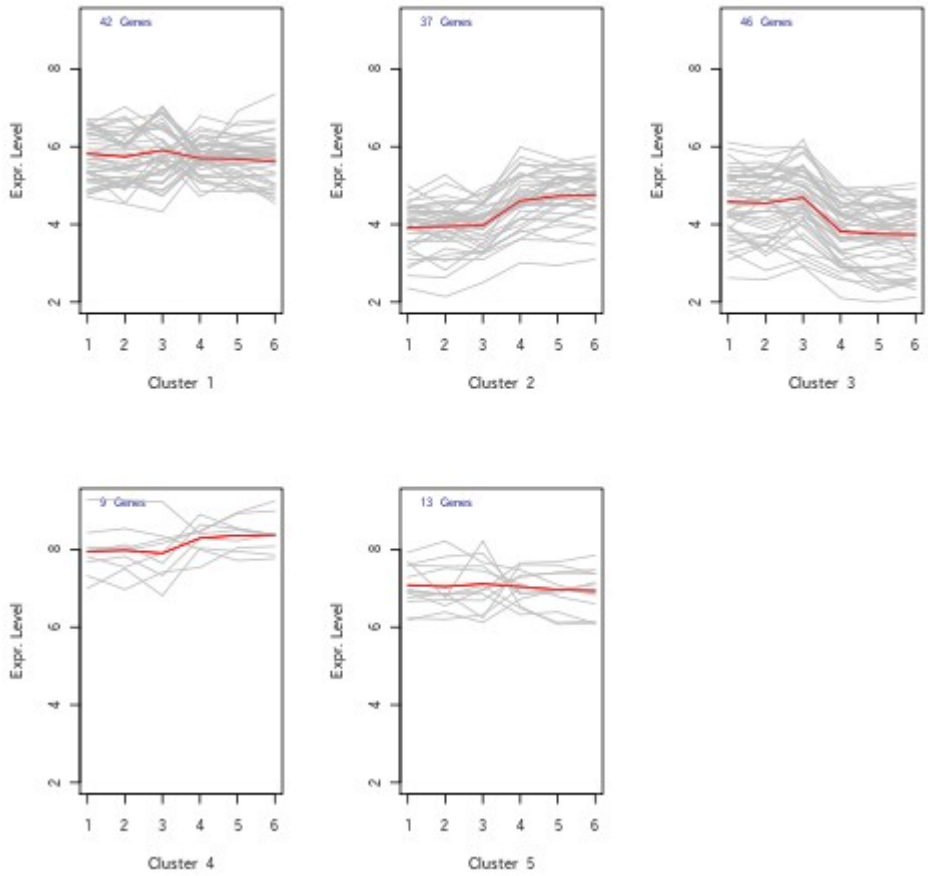
kde s_A je standardní odchylka hodnot $X_{i,A}$. V závěrečná části této kapitoly zopakujme, jak můžeme nabytých znalostí využít pro shlukování dat z mikročipů.

5.4 Shlukování dat z mikročipů

Zde shrneme celý postup shlukování dat z mikročipů. Nejprve jsou hrubá výstupní data z mikročipů upravena pomocí transformacích uvedených v části 5.2. Následně je zvoleno, zda budeme shlukovat geny nebo vzorky a k tomu zvolena vzdálenostní funkce, např. některá z uvedených v části 5.3. Poté nastupuje samotné shlukování upravených dat. Ke shlukování používáme nejčastěji algoritmy k-means, hierarchické, SOM a SOTA, ale obecně je možné použít jakékoliv.

Výsledkem shlukování jsou shluky genů, které se chovají podobně za měnících se podmínek okolního prostředí buňky, nebo shluky vzorků charakterizované podmínkami a vlastnostmi

organismu, které mají podobné genové exprese. Je též možno provést shlukování zaráz podle genů i vzorků v metodě známé jako biclustering (více viz konec části 16.3 knihy [1]). Takto jsou utvořeny shluky genů a shluky vzorků, jejich průnik pak vytvoří ony shluky vykazující podobné chování genů z důvodu podobných podnětů z okolí. Výsledek shlukové analýzy pomocí metody SOTA vidíme na obrázku . Ke každému shluku je zobrazen graf, kde vodorovná osa označuje čas a svislá (transformovanou) hodnotu genové exprese. Tenké čáry označují jednotlivé geny shluku, tlustá čára průměr hodnot expresí všech genů shluku. Čím je suma kvadrátu vzdálenosti tenkých čar od tlusté čáry menší, tím je výsledek shlukové analýzy lepší.



Ilustrace 10: Shluky vzniklé algoritmem SOTA (zdroj [15])

Kapitolu nyní završíme shrnutím využití shlukové analýzy. Shluková analýza hodnot genové exprese tak může posloužit k lepší orientaci v metabolických a signálních drahách organismu, víme, které geny spolu souvisí. Rovněž víme, které geny podobně reagují na změnu okolního prostředí a které druhy okolního prostředí vyvolají podobné genové exprese, což může posloužit k tvorbě účinných léků proti rakovině. Také lze na základě vlastností organismu určit nebo potvrdit náchylnost k jistým nemocem pomocí shlukování vzorků.

Následující kapitola se také zabývá zejména shlukováním dat z mikročipů, je však již značně technickou přípravou k implementaci a proto je od této kapitoly oddělena.

6 Zpracování dat z mikročipů programem

Jak jsme v předešlých kapitolách uvedli, data z mikročipů budeme zpracovávat pomocí metod shlukování. Před vlastní implementací však ještě provedeme teoretickou přípravu, samotná implementace bude popsána v kapitole následující. Nejprve definujeme vstupy programu, poté konkrétní shlukovací metody, které budeme implementovat a nakonec výstupy programu, nad kterými bude možné provádět analýzu. Protože se na odstavce v celé této kapitole budeme v dalších kapitolách značně odkazovat, budeme odstavce číslovat.

6.1 Vstup programu

(1) Vstupem našeho programu budou data z mikročipů uložená v internetové databázi SMD popsané v části 3.2 na straně 9. Budeme shlukovat geny (tedy nikoliv vzorky), podstatné pro náš program však bude, aby bylo možné jej snadno rozšířit i pro shlukování vzorků. Jako konkrétní formát souboru s geny a jejich expresemi použijeme soubor typu pcl (Pre-CLustering file) generovaný databází SMD, jak je popsáno v části 3.2.1 na straně 9. Opět budeme požadovat, aby bylo možné program snadno rozšířit i pro zpracování jiných typů souborů.

(2) Pro názornost zopakujme formát souboru pcl a nastiňme některé aspekty jeho zpracování programem. Přesnou specifikaci lze najít na http://smd.stanford.edu/help/formats.shtml. Každý řádek souboru je rozdělen do sloupců oddělených tabulátory. Každý řádek odpovídá právě jednomu genu. Může se stát, že více různých řádků odpovídá stejnému genu, tuto možnost však nebudeme v našem programu uvažovat, neboť takové řádky budou sloučeny do jediného ještě před stažením z databáze SMD pomocí standardních voleb zadávaných při výběru obsahu souboru pcl. První sloupec řádku obsahuje identifikátor genu. Podstatné je, že musí být v rámci souboru jednoznačný a program ho bude dále k identifikaci genu používat. Druhý sloupec obsahuje popis genu, který bude program ignorovat. Třetí sloupec obsahuje váhu, s jakou gen vstupuje do dalšího zpracování. Tato hodnota je v drtivé většině souborů na všech řádcích rovna 1, budeme proto v programu uvažovat, že tato hodnota je vždy 1. Program však musí umožnit snadné rozšíření i pro menší váhy genů. Konečně další sloupce již obsahují reálné hodnoty genové exprese.

(3) U čtvrtého a dalšího sloupce odpovídá každý sloupec jednomu měření. Celkový počet měření tak odpovídá dimenzi bodu, reprezentující gen v algoritmu shlukování. Souřadnicemi jsou pak hodnoty těchto sloupců. Pokud pro některý gen nebylo některé měření provedeno, je v daném řádku a sloupci prázdný řetězec, počet tabulátorů oddělující sloupce však zůstává stejný (tedy například pokud pro 5. sloupec chybí naměřená hodnota, jsou mezi 3. a 5. sloupcem 2 tabulátory za sebou). Pro aplikaci shlukovacích algoritmů je nutné, aby každý gen měl stejnou (plnou) dimenzi, chybějící hodnoty je tedy třeba doplnit. Pro doplnění bude zvolena následující metoda. K danému genu A s chybějící hodnotou je nalezen nejbližší gen B v rámci zadané metriky (přičemž se ignorují souřadnice, které u genu A chybí), kterému žádná souřadnice nechybí (od počátku nebo mu již byly prázdné souřadnice doplněny). Chybějící hodnoty genu A se pak položí rovny příslušným hodnotám genu B. Je třeba však pamatovat, že to přinese následující nevýhodu. Je-li počet přítomných souřadnic genu A nízký (např. 1), může se stát, že bude nalezen gen B s téměř stejnými souřadnicemi, jako má gen A (na přítomných souřadnicích). Po doplnění chybějících hodnot tak v souboru genů budou dva geny s téměř stejnými hodnotami exprese, což nemusí odpovídat skutečnosti.

(4) Samotné hodnoty genových expresí v jednotlivých řádcích mohou být různého významu v obecném pcl souboru. V databázi SMD je nutné po výběru dat ke stažení pomocí Basic search (viz obrázek 6 na str. 10), avšak ještě před stažením pcl souboru v sekci Data filtering options (viz obrázek 7 na str. 11), tedy možnosti filtrování dat, zadat význam oněch sloupců s genovými expresemi. Využijeme zde toho, že databáze data sama předzpracuje pomocí metod uvedených v části 5.2. Konkrétně zatrhneme volbu Log(base2) of R/G Normalized Ratio (Mean), což znamená, že data budou předpočítána dle poměru v rovnici (5.3) na straně 23.

(5) Další volbou, kterou musíme provést, je volba vzdálenostní funkce mezi geny. Použijeme

klasickou euklidovskou vzdálenost definovanou rovnicí (5.4) na straně 24. Implementace programu však musí umožnit jednoduché rozšíření pro použití jiných vzdálenostních funkcí.

(6) Nyní se ještě vraťme k formátu pcl souboru. První řádek souboru obsahuje hlavičku, tedy záhlaví sloupců. Tento řádek budeme v programu ignorovat, neboť je pouze informativní, tedy určený pro uživatele. Druhý řádek obsahuje váhy jednotlivých měření, ty jsou opět v drtivé většině rovny 1 a program to bude také vždy implicitně předpokládat (opět s možností snadného rozšíření programu). Třetí a další řádky již obsahují údaje pro jednotlivé geny. Prázdné řádky by měl program ignorovat. Rovněž by měl vypisovat, které řádky mají špatný formát, aby tyto mohl uživatel opravit.

6.2 Použité algoritmy

Z předešlé sekce víme, že vstupní data již budeme mít jednak předzpracována internetovou databází, jednak v nich budou doplněny chybějící hodnoty genových expresí. Biologický problém tak nyní přeneseme do čistě informatické roviny. Každý řádek souboru pcl (mimo dvou počátečních) bude představovat jeden gen, tedy jeden bod n-dimenzionálního euklidovského prostoru, n hodnot expresí v tomto řádku bude hodnotami souřadnic tohoho bodu.

Z oněch zadaných bodů budeme nyní vytvářet shluky pomocí různých shlukovacích algoritmů. Největším požadavkem u všech algoritmů bude patrně vysoká rychlost a nízká paměťová náročnost, což je třeba zdůraznit zejména proto, že například u člověka shlukujeme cca 25 000 bodů. Dodejme, že dimenze samotných bodů již nehraje velkou roli, neboť se projeví jen při počítání vzdálenostní funkce, která není časově ani paměťově náročná, i když je počítána mnohokrát (pro různé dvojice bodů), samozřejmě myšleno relativně k časové a paměťové náročnosti celých algoritmů. Nyní uvedeme jednotlivé shlukovací algoritmy, které budeme využívat, včetně požadavků kladených na jejich implementaci. Program však musí umožnit jednoduché přidání dalších shlukovacích algoritmů do implementace kvalitní volbou použitých rozhraní.

6.2.1 k-means

(1) Prvním implementovaným algoritmem by měl být zástupce třídy rozdělovacích algoritmů – k-means (viz část 4.2.1.1 na straně 15). Přestože tento algoritmus má nevýhodu v tom, že předem musíme zadat počet shluků k, nespornou výhodou je, že je velice rychlý. Onu nevýhodu je tak možné odstranit spočítáním shluků pro mnoho různých hodnot k. Další výhodou je, že algoritmus můžeme použít jako součást hierarchických algoritmů (o tom viz další sekce).

(2) Speciálně ještě vyjádřeme časovou složitost algoritmu. V každé iteraci se určuje nejbližší centroid pro každý bod, to je celkově řádu O(kn), kde n počet bodů. Následně se přepočítávají centroidy pro každý shluk, což je zřejmě řádu O(n). Celkově je tak iterace řádu O(kn), je však třeba připomenout, že počet operací pro jeden bod je velice nízký. Počet iterací však není předem znám (závisí na zadané maximální požadované chybě), celkovou složitost algoritmu tak neznáme. Můžeme ji však omezit zadáním maximálního počtu provedených iterací p na O(pkn).

(3) Jmenujme vstupní parametry, které by měla implementace připouštět. Kromě zadání počtu shluků k by mělo být možné zadat požadavek na maximální celkovou kvadratickou chybu a maximální počet iterací algoritmu, nepodaří-li se požadavek maximální chyby splnit. Program by měl také ukončit iterace, pokud se již během poslední iterace nezměnilo rozdělení bodů do shluků nebo nesnížila celková kvadratická chyba. Program by měl být schopen náhodně generovat všechny počáteční centroidy k shluků (před první iterací algoritmu), rovněž by však mělo být možné místo toho předat počáteční volbu centroidů všech k shluků parametrem. Náhodností se zde míní, že každá k-tice počátečních centroidů by měla být vybrána se stejnou pravděpodobností. Poznamenejme, že díky předzpracování uvedeném v odstavci (3) části 6.1 může mít více bodů (téměř) totožné souřadnice. Následkem toho se může stát, že některé z k shluků budou po proběhnutých iteracích úplně prázdné.

(4) Výstupem algoritmu by měl být seznam všech k nalezených shluků. Každý záznam o shluku by měl obsahovat seznam svých bodů, centroid a kvadratickou chybu.

6.2.2 Vlastní hierarchický dělící shlukovací algoritmus (Strom++)

(1) Dalším implementovaných algoritmem by měl být dosud neprezentovaný, originální hierarchický dělící shlukovací algoritmus, který budeme dále nazývat Strom++. Před uvedením výhod, nevýhod a požadavků na algoritmus jej nejprve definujeme. Algoritmus bude mít několik modifikací, které budou přesahovat také do následující sekce.

(2) Vedoucí myšlenkou tvorby nového algoritmu bylo odstranit nutnost zadání výsledného počtu shluků k jako u algoritmu k-means, avšak současně se pokusit zachovat časovou nenáročnost s ohledem na velký počet shlukovaných bodů. Dalším motivem byla možnost dalšího využití v algoritmech založených na hustotě, o nichž bude pojednáno v následující sekci. Proto byl vybrán obecný přístup využívaný v hierarchických dělících algoritmech, kde jsou shluky organizovány do stromu, podobně jako v algoritmu SOTA (viz sekce 4.2.4.2 na str. 20).

(3) Pomocnou strukturou tohoto algoritmu je tedy strom. Listy odpovídají shlukům nejnižší kategorie. Obsahují seznam bodů n-dimenzionálního prostoru, jimiž je shluk tvořen, mají spočítán centroid a poměr kvadratické chyby ku počtu svých prvků (dále jen relativní chyba). Podstatné je, že relativní chyba každého listového shluku nesmí překročit maximální relativní chybu zadanou uživatelem. Listový shluk tedy může obecně obsahovat libovolně vysoký počet bodů prostoru. Nelistové uzly odpovídají shlukům vyšší kategorie. Obsahují seznam svých potomků, centroid a relativní chybu. Podstatné je, že má vždy maximálně 3 potomky. Záměrně není u nelistového uzlu zvolen žádný maximální limit pro relativní chybu, neboť by uživatel musel tento limit zadávat pro každé patro stromu.

(4) Nyní si popíšeme, jak je strom tvořen, tedy jak jsou do něj přidávaný další body prostoru. Na počátku obsahuje strom jediný list, který je zároveň kořenem stromu. Poté jsou do stromu postupně přidávány další a další body prostoru (z nichž chceme počítat shluky). Jakmile relativní chyba listu překročí zadanou mez, list je pomocí k-means rozdělen na dva listové shluky, zde tedy k = 2. Algoritmus k-means nám správně přepočítá i centroidy i kvadratickou chybu těchto dvou shluků. Následně je těmto dvěma shlukům vytvořen (nelistový) předek, který se tak stává novým kořenem stromu. Z hodnot centroidů a relativní chyby svých potomků je mu následně spočítán vlastní centroid a vlastní relativní chyba.

(5) S ohledem na řečené si nyní odvoďme, jak lze jednoduše spočítat centroid a kvadratickou chybu předka se dvěma potomky, známe-li centroidy a kvadratické chyby obou jeho potomků. Nechť $\mathbf{x_i}$ jsou prvky prvního pomotka, x_i^j jejich j. souřadnice, a jejich počet, $\mathbf{y_i}$ prvky druhého potomka, y_i^j jejich j. souřadnice, b jejich počet. Dále označme $\bar{\mathbf{x}}$ centroid prvního potomka, $\bar{x}^j$ jeho j. souřadnici, $\bar{\mathbf{y}}$ centroid druhého potomka, $\bar{y}^j$ jeho j. souřadnici a $\bar{\mathbf{z}}$ centroid předka, $\bar{z}^j$ jeho j. souřadnici. Konečně nechť kvadratická chyba prvního potomka, resp. druhé potomka, resp. předka bude E_x , resp. E_y , resp. E_z . Platí tedy:

$$\bar{x}^j = \frac{1}{a} \sum_i x_i^j \tag{6.1}$$

$$\bar{y}^j = \frac{1}{b} \sum_i y_i^j \tag{6.2}$$

$$E_x = \sum_{i,j} \left(x_i^j - \bar{x}^j \right)^2 \tag{6.3}$$

$$E_y = \sum_{i,j} \left(y_i^j - \bar{y}^j \right)^2 \tag{6.4}$$

Pro centroid předka zřejmě dostáváme:

$$\bar{\mathbf{z}} = \frac{1}{a+b} \left(a\bar{\mathbf{x}} + b\bar{\mathbf{y}} \right) \tag{6.5}$$

Nyní odvodíme výraz pro E_z, zřejmě:

$$E_z = \sum_{i,j} \left(x_i^j - \bar{z}^j \right)^2 + \sum_{i,j} \left(y_i^j - \bar{z}^j \right)^2 \tag{6.6}$$

Upravujme nyní první ze sčítanců, u druhého by byla úprava analogická.

$$\begin{aligned} \sum_{i,j} \left(x_i^j - \bar{z}^j \right)^2 &= \sum_{i,j} \left(x_i^j - \bar{x}^j + (\bar{x}^j - \bar{z}^j) \right)^2 = \\ = \sum_{i,j} \left(x_i^j - \bar{x}^j \right)^2 &+ \sum_{i,j} \left(\bar{x}^j - \bar{z}^j \right)^2 + 2 \sum_{i,j} \left(x_i^j - \bar{x}^j \right) \left(\bar{x}^j - \bar{z}^j \right) = \\ &= E_x + a \sum_j \left(\bar{x}^j - \bar{z}^j \right)^2 \end{aligned} \tag{6.7}$$

, neboť:

$$\begin{aligned} 2 \sum_{i,j} \left(x_i^j - \bar{x}^j \right) \left(\bar{x}^j - \bar{z}^j \right) &= 2 \sum_j \left((\bar{x}^j - \bar{z}^j) \sum_i (x_i^j - \bar{x}^j) \right) = \\ = 2 \sum_j \left((\bar{x}^j - \bar{z}^j) \left(\sum_i (x_i^j) - a\bar{x}^j \right) \right) &= 0 \end{aligned} \tag{6.8}$$

Celkově tak pro E_z dostáváme:

$$E_z = E_x + E_y + a \sum_j (\bar{x}^j - \bar{z}^j)^2 + b \sum_j (\bar{y}^j - \bar{z}^j)^2 \tag{6.9}$$

Tento vztah lze snadno zobecnit na větší počet potomků předka. Speciálně pro dva potomky však můžeme poslední dva sčítance předchozího vztahu ještě zjednodušit. Neboť platí:

$$\bar{\mathbf{x}} - \bar{\mathbf{z}} = \frac{1}{a+b} (a\bar{\mathbf{x}} + b\bar{\mathbf{x}} - a\bar{\mathbf{x}} - b\bar{\mathbf{y}}) = \frac{b}{a+b} (\bar{\mathbf{x}} - \bar{\mathbf{y}}) \tag{6.10}$$

$$\bar{\mathbf{y}} - \bar{\mathbf{z}} = \frac{1}{a+b} (a\bar{\mathbf{y}} + b\bar{\mathbf{y}} - a\bar{\mathbf{x}} - b\bar{\mathbf{y}}) = \frac{a}{a+b} (\bar{\mathbf{y}} - \bar{\mathbf{x}}) \tag{6.11}$$

, lze sčítance upravit na:

$$a\sum_j(\bar{x}^j-\bar{z}^j)^2+b\sum_j(\bar{y}^j-\bar{z}^j)^2=$$

$$=a\sum_j\left(\frac{b}{a+b}\right)^2\left(\bar{x}^j-\bar{y}^j\right)^2+b\sum_j\left(\frac{a}{a+b}\right)^2\left(\bar{y}^j-\bar{x}^j\right)^2= \tag{6.12}$$

$$\frac{ab^2}{(a+b)^2}+\frac{ba^2}{(a+b)^2}\sum_j(\bar{x}^j-\bar{y}^j)^2=\frac{ab}{a+b}\sum_j(\bar{x}^j-\bar{y}^j)^2$$

Pro dva potomky pak tedy dostáváme:

$$E_z=E_x+E_y+\frac{ab}{a+b}\sum_j(\bar{x}^j-\bar{y}^j)^2 \tag{6.13}$$

(6) Dále si přiblížíme, jak jsou do stromu, který již obsahuje více než jeden list přidávány další body prostoru. Bod je nejprve předán kořenu stromu. Ten, jakožto nelistový uzel, hledá mezi svými potomky uzel, jehož centroid je nejblíž přidávanému bodu. Následně je bod předán tomuto uzlu. Není-li uzel listem, provádí i on a jeho potomci totéž co kořenový uzel, dokud není bod předán listu, který reprezentuje shluk. V tomto listu je uzel přidán do seznamu, aktualizován centroid i relativní chyba listu.

(7) Překročí-li relativní chyba povolenou mez, je (starý) list pomocí k-means s k = 2 rozdělen na dva (nové) listy. Ze seznamu potomků předka (starého) listu je potom odebrán starý list a přidány dva nové. Můžeme též uvažovat mírnou modifikaci algoritmu při rozdělování starého listu na dva. Nechť je počet potomků předka (starého) listu p (tedy $p \in \{1,2,3\}$, neboť jsme povolili maximálně tři potomky nelistovému uzlu). Soubor všech bodů potomků tohoto předka potom necháme rozdělit na p+1 nových shluků pomocí k-means s k = p+1. Připomeňme, že vzhledem ke struktuře stromu, byly všichni potomci předka listy. Těchto p (starých) listů potom odebereme ze seznamu potomků předka a přidáme místo nich oněch p+1 (nových) listů.

(8) U nelistového uzlu se opět zajímáme o to, jestli po změně jeho potomků nepřerostl jejich počet číslo tři. Pokud ano, má tedy čtyři potomky, je potřeba tento uzel rozdělit na dva. Požadujeme aby v obou (nových) uzlech bylo po dvou potomcích. Existují tedy tři kombinace rozdělení. Pro každou kombinaci spočítáme součet kvadratických chyb dvou nových uzlů a vybereme kombinaci, která má tento součet nejmenší. Starý uzel opět nyní odebereme ze seznamu potomků předka (starého) uzlu a přidáme dva nové. Je-li to nutné, pokračujeme v dělení uzlů směrem nahoru až ke kořeni. Pokud je třeba rozdělit i kořen, je vytvořen nový kořen a ony dva rozdělené uzly vzniklé z původního kořene jsou mu přidány jako potomci. Celý proces je tak analogický přidávání prvku do B+ stromu. Dodejme ještě, že po přidání bodu do stromu jsou aktualizovány centroidy i relativní chyby všech uzlů, přes které byl bod předán, než „došel“ z kořene do listu.

(9) Předpokládejme nyní, že jsme již všechny zadané body prostoru přidali do stromu a strom je tedy úplně zkonstruován. Za výsledek shlukování pak můžeme považovat všechny listové shluky, ale také shluky reprezentované uzly v jednom konkrétním patře stromu. Ke každému takovému (nelistovému) uzlu je potom třeba dopočítat všechny body, které patří do listů, které jsou (přímými i nepřímými) potomky tohoto uzlu. Tento výsledek shlukování však můžeme dále upravit. Jednou možností je použít shlukovací algoritmy založené na hustotě, to je popsáno v další sekci. Zde popíšeme úpravu pomocí k-means.

(10) Nejprve vybereme patro. Centroidy uzlů v tomto patře se stanou iniciálními centroidy pro k-means, kde k bude rovno počtu uzlů v patře. K tomu musíme udat požadovanou maximální kvadratickou chybu, při jejímž podkročení se k-means zastaví. Bude-li tato požadovaná chyba stejná nebo větší než jaký je součet kvadratických chyb uzlů patra, k-means skončí ihned po první iteraci,

pouze potenciálně přeuspořádá body k bližším centroidům a příšlušně se poté změní také poloha centroidů. Toto tvrzení (že k-means skončí hned po první iteraci) je třeba dokázat.

(11) Nechť je tedy vstupem k-means k centroidů a k nim příslušné body tak, že celková kvadratická chyba je menší než E. Nyní stačí dokázat, že po proběhnutí jedné iterace k-means se chyba zmenší nebo zůstane stejná, což nutně znamená, že bude stále menší než E. V první fázi iterace k-means pouze (potenciálně) přemísťuje body k nejbližším centroidům, centroidy však zůstavají nehybné, celková chyba se tak může jedině zmenšit (nebo zůstat stejná). Po této fázi nastává přepočítání centroidů a tím i přepočítání chyby. Uvažme tedy jeden takový shluk s body $\mathbf{x_i}$ o souřadnicích x_i^j a libovolný bod $\mathbf{w}$ o souřadnicích w^j. Určíme nyní, jaké musí mít w souřadnice, aby byl výraz (funkce proměnných w^j) $\sum_{i,j}\left(x_i^j - w^j\right)^2$ minimální. Hledáme tedy známou metodou z diferenciálního počtu minimumem funkce o více proměnných w^j. Nejprve hledáme stacionární body funkce. Zřejmě se jedná o spojitou funkci, která minima nabývá jen pro konečné hodnoty souřadnic. Položíme parciální derivace dle w^j rovny 0:

$$0 = \frac{\partial}{\partial w^j}\sum_{i,k}\left(x_i^k - w^k\right)^2 = \sum_i \frac{\partial}{\partial w^j}\left(x_i^j - w^j\right)^2 = \sum_i -2(x_i^j - w^j)$$

$$0 = \sum_i \left(x_i^j\right) - aw^j \tag{6.14}$$

, kde a je počet bodů shluku. Z toho dostáváme pro w^j:

$$w^j = \frac{1}{a}\sum_i x_i^j \tag{6.15}$$

, což je definice centroidu. Pokud bychom dosadili tento w do Hessovy matice druhých derivací, pravděpodobně bychom zjistili, že je matice pozitivně definitní a skutečně se jedná o minimum. My se však spokojíme s úvahou, že minimum pro konečné w musí existovat a tudíž bude rovno tomuto jedinému nalezenému stacionárnímu bodu. Původní chyba shluku tedy byla se starým centroidem obecně různým od (skutečného) nového centroidu větší nebo rovna nové přepočítané chybě. To znamená, že přepočítání centroidu může chybu opět jen zmenšit (nebo nechat stejnou). Protože toto platí pro každý shluk, může se celková kvadratická chyba po jedné iteraci k-means jen zmenšit (nebo zůstat stejná).

(12) Pokud bychom tedy chtěli významně změnit polohu centroidů, museli bychom zadat požadovanou chybu menší než součet kvadratických chyb uzlů patra. Otázkou ovšem zůstává, může-li se chyba ještě zmenšovat.

(13) Věnujme se nyní ještě chvíli časové složitosti algoritmu. Vzhledem k analogii přidávání prvků do stromu s B+ stromem lze předpokládat, že strom bude vyvážený, aspoň co se týče počtu uzlů (listových i nelistových). Je-li počet uzlů u, přidání dalšího prvku do stromu bude řádu O(ln u), neboť operace na jednodem uzlu na cestě od kořene dolů k listu budou mít s ohledem na řečené konstantní složitost, stejně jako případně rozdělování uzlů. Opomněli jsme však fakt, že při přidávání bodu do listu se může spustit k-means, jehož složitost je řádu O(t), kde t je počet bodů v listu. Protože jsme počet bodů v listu nijak neomezili, mohou v něm být obsaženy (v nejhorším případě) téměř všechny body stromu. Složitost přidání bodu v tomto špatném případě by tak byla řádu O(n ln u), kde n je celkový počet bodů ve stromu. Tuto nevýhodu lze odstranit nastavením maximálního počtu bodů, které smí list obsahovat (například v řádu stovek), čímž se zřejmě dostaneme na složitost O(ln n). Tato mez má také výhodu v tom, že hustěji „osídlené“ oblasti prostoru, budou rozděleny do shluků s menší relativní chybou, jedná se tak o výhodnou modifikaci původního algoritmu. Přidání všech bodů by pak mělo složitost O(n ln n). Nutno však poznamenat, že pokud algoritmus použijeme ve spojení s algoritmem založeným na hustotě popsaným v následující sekci, nemá tato mez počtu bodů v listu smysl (viz další sekce). Případné následující k-means provedené na konci algoritmu na konkrétní

patro může časovou složitost ještě zvětšit. Počet shluků v patře totiž může být řádu O(n) a k-means by tak proběhl v řádku $O(pn^2)$, kde p je počet proběhlých iterací.

(14) Nyní přistupme k požadavkům implementace programu realizující algoritmus Strom++. Uživatel by měl mít možnost zadat onu maximální mez relativní chyby listu. Po přidání všech bodů by program měl informovat o počtu pater stromu a nabídnout uživateli možnost vypsat všechny shluky daného patra včetně seznamu svých prvků, centroidu a kvadratické chyby. Dále by mělo být možné spustit k-means na shluky zadaného patra s volbou maximálního počtu iterací a případně volbou nižší kvadratické chyby než dosažené v daném patře stromu. V rozšíření by mělo být možné zadat též mez pro maximální počet bodů v listovém uzlu.

6.2.3 DBSCAN

(1) Posledním z rodiny algoritmů, který by měl program implementovat, je shlukovací algoritmus založený na hustotě – DBSCAN uvedený v sekci 4.2.3.1 na straně 18. Body, které se nachází v ε-okolí daného bodu nazveme jeho sousedy. Poznamenejme, že ε je zadané uživatelem.

(2) Prvním problémem, kterým musíme před samotným spuštěním algoritmu řešit, je nalezení sousedů každého bodu. To by bylo možné, pokud bychom měli všechny body uloženy v nějaké speciální struktuře umožňující hledat nejbližší sousedy. Nabízí se například rozdělení prostorů do m-dimenzionálních krychlí, přičemž jednotlivé krychle by mohly být rozděleny na další, obsahují-li větší počet bodů. Další možností by bylo využití struktury analogické B+ stromu, avšak modifikované pro více dimenzí nebo použití již známého R stromu. Všechny tyto přístupy by byly patrně přímočaře realizovatelné, vyžadovaly by však mnoho práce, jež by patrně překračovala rozsah této práce, která se těmito tématy ústředně nezabývá. Pro hledání sousedů proto zvolíme jinou, možná o něco méně účinnou metodu, využívající výsledků předchozí sekce.

(3) Předpokládejme, že máme z bodů, u nichž hledáme shluky, již spočítán strom, popsaný v předchozí sekci. Předpokládáme tedy, že každý list má limitovánu relativní chybu (kvadratickou chybu dělenou počtem prvků), počet prvků v listu limitován není (na rozdíl od jedné z uvedených modifikací). Předpokládejme nyní idealizovaný případ, že listové shluky jsou kulového tvaru a hustota bodů v této kouli je všude stejná, rozložení bodů je tedy rovnoměrné. Pokusme se nyní určit, jak souvisí poloměr takového shluku s jeho relativní chybou. K tomu bude zapotřebí několik matematických vzorců.

(4) Zřejmě pro objem m-dimenzionální koule $V_m(R)$ o poloměru R platí:

$$V_m(R) = C_m R^m \tag{6.16}$$

, kde C_m je konstanta (její konkrétní hodnota nás nezajímá, konkrétně ji lze najít například na http://en.wikipedia.org/wiki/Hypersphere). Tato koule je složena z přiléhajících (m-1)-dimenzionálních sfér o povrchu $S_{m-1}(r)$ tloušťky dr o poloměrech postupně 0 až R. Platí tedy:

$$V_m(R) = \int_0^R S_{m-1}(r)dr \tag{6.17}$$

Derivací této rovnice dle R dostáváme vztah pro velikost povrchu (m-1)-dimenzionální sféry:

$$S_{m-1}(R) = \frac{dV_m(R)}{dR} = mC_m R^{m-1} \tag{6.18}$$

(5) Předpokládejme nyní, že hustota bodů v oné kouli je ρ, přičemž:

$$\rho = \frac{a}{V_m} \tag{6.19}$$

, kde a je počet bodů v kulovém shluku. Neboť body jsou v kouli rovnoměrně rozloženy, je zřejmě

centroid ve středu koule. V diferenciálním rozmezí vzdáleností $(r, r+dr)$, které odpovídá (m-1)-dimenzionální slupce (sféře) o poloměru r a tloušťce dr se nachází $p(r)$ bodů, kde zřejmě:

$$p(r) = \rho S_{m-1}(r)dr \tag{6.20}$$

S ohledem na diferenciální rozmezí můžeme předpokládat, že všechny tyto body jsou od středu (centroidu) vzdáleny r, jejich celkový příspěvek k kvadratické chybě je tedy $p(r)r^2$. Pro celkovou kvadratickou chybu E koule je třeba tyto příspěvky sečíst pro všechny slupky o poloměrech 0 až R, protože se jedná o slupky o diferenciální tloušťce, sčítání se mění na integrování, tedy:

$$\begin{aligned} E &= \int_0^R p(r)r^2 = \int_0^R \rho m C_m r^{m-1} r^2 dr = \rho m C_m \int_0^R r^{m+1} dr = \\ &= \rho C_m \frac{m}{m+2} R^{m+2} = \frac{a C_m}{C_m R^m} \frac{m}{m+2} R^{m+2} = a \frac{m}{m+2} R^2 \end{aligned} \tag{6.21}$$

Získali jsme tak závislost mezi relativní chybou E/a a poloměrem R:

$$\frac{E}{a} = \frac{m}{m+2} R^2 \tag{6.22}$$

(6) Přejeme-li si nyní, aby všechny body koule o poloměru ε měli své sousedy (v ε-okolí každého bodu) v jednom listu, musí jeho poloměr být $R = 2\varepsilon$. Proto relativní chyba shluku musí být:

$$\frac{E}{a} = \frac{m}{m+2} R^2 = 4 \frac{m}{m+2} \varepsilon^2 \tag{6.23}$$

Tím jsme získali závislost mezi relativní chybou shluku a hodnotou ε. Samozřejmě musíme mít na paměti, že jsme si idealizovali předpoklady a tento vztah neplatí obecně. Navíc body, které se nachází ve vzdálenosti (ε, 2ε) od středu koule, své ε-okolí již nebudou mít uvnitř shluku.

(7) V předchozích odstavcích jsme tak naznačili, že sousedé bodu by se mohly nacházet ve stejném shluku jako tento bod. Je však také možné, že se sousedé nachází ještě v několika sousedních shlucích. Pro každý list i uzel tedy spočítáme rádius, což bude největší vzdálenost, které nabývá nějaký bod tohoto listu či uzlu, od centroidu. Předpokládejme tedy, že máme dán listový shluk A s radiem r_A a centroidem c_A a chceme k bodům tohoto listu najít všechny sousedy v jejich ε-okolích. S ohledem na nastavenou relativní chybu nejdříve zkusíme všechny dvojice bodů v tomto listu, zda jsou navzájem sousedy. Poté začneme k listu A hledat sousední listy. Toto hledání rozdělíme pro snadnější popis do etap (které odpovídají stoupání stromem nahoru), které se samy ještě budou dělit do iterací (které dopovídají klesání stromem dolů).

(8) V první etapě vystoupíme z listu A o úroveň výš k jeho předkovi U. Poté v iteracích této etapy postupně zjišťujeme u všech ostatních potomků předka U, zda by ε-okolí bodů z listu A mohlo mít s těmito potomky průnik. Označíme-li takové potomka P, jeho radius r_P a centroid c_P , pak k takovému průniku zřejmě může dojít jen pokud platí nerovnost:

$$|c_P - c_A| - r_P - r_A - \varepsilon < 0 \tag{6.24}$$

, kde jsme $|c_P - c_A|$ označili vzdálenost centroidů c_P a c_A . Pokud je tato podmínka splněna, opět vyzkoušíme všechny páry bodů, kdy první pochází z listu A a druhý z listu P. Pro urychlení výpočtu, můžeme ještě pro každý bod B z listu A testovat lokálně tuto podmínku:

$$|c_P - c_B| - r_P - \varepsilon < 0 \tag{6.25}$$

, která opět určuje, zda samotný bod B může mít ve svém ε-okolí body z listu P. Není-li první nebo druhá podmínka splněna, netřeba příslušné kombinace bodů zkoušet pro sousedství.

(9) V dalších etapách postupujeme postupně ke vzdálenějším předkům F listu A, ve druhé tedy k přímému předkovi U atd. Opět v iteracích etap zkoušíme u všech ostatních potomků předka F

(tedy u jiných potomků, než ze kterého jsme směrem od listu A přišli) platnost podmínky (6.24). Pokud u takového potomka P tato podmínka platí, ověřujeme platnost podmínky (6.24) opět u jeho potomků v dalších iteracích, a tak dále rekurzivně, dokud se nedostaneme do listů. Pro listy L, které splňují podmínku (6.24) (kde samozřejmě místo symbolu P je symbol L) opět ověřujeme ony páry bodů, kdy první pochází z listu A a druhý z listu L, na sousedství. Opět využíváme urychlení testováním lokální podmínky.

(10) Touto metodou zjistíme všechny sousedy každého bodu z listu L. Metodu tedy spustíme na každém listu celého stromu. Jelikož by se nám každá potenciální dvojice sousedů testovala dvakrát, začneme výpočet od nejlevějšího listu stromu a postupně se přesouváme k pravějším listům, přičemž k danému listu testujeme na sousedství jen listy napravo od něj. V rámci tohoto testování nezjišťujeme jen sousedství, ale také, které body jsou jádrem. Připomeňme, že bod je jádrem, pokud se v jeho ε-okolí nalézá aspoň m sousedů, kde m je zadané uživatelem, pro všechny body stejné.

(11) Předpokládejme nyní, že máme již zkonstruován strom, nalezené sousedy každého bodu a jádra. Dá se říct, že tak máme dán orientovaný graf, kde body tvoří vrcholy a vztah sousedství hrany. Hrany jsou orientované a míří vždy od jádra k jeho sousedu, případně obousměrně pokud jsou oba tito sousedé jádra. Nyní musíme z tohoto orientovaného grafu „extrahovat“ shluky.

(12) Záležitost extrakce shluků z grafu by byla jednoduchá, pokud bychom měli všechny relace sousedství uložené v paměti počítače. Potom by stačilo nechat prohledávát graf do hloubky, všechny vrcholy, do kterých bychom se byli schopni dostat z jednoho (výchozího) bodu, by patřili do téhož shluku a byly by při prohledávání do hloubky označeny nějakým id výchozího bodu. S ohledem na velký počet bodů však nemůžeme mít relaci sousedství v paměti uloženou, shluky je tedy třeba počítat přímo při hledání sousedů.

(13) Jeví se tak výhodné každý shluk reprezentovat jedním stromem (v žádném případě se nejedná o strom algoritmu Strom++ určený k vyhledávání sousedů). Pro jezdnoznačnost bude aktuálním identifikátorem takového stromu jeho kořen. Každý bod stromu (shluku) bude mít ukazatel na svého předka v tomto stromu a počet pater, které se nacházejí v tomto stromu pod ním, rozhodně však nebude mít ukazatele na své potomky. Pokud nalezneme novou dvojici jader, které jsou navzájem sousedy, bude třeba jejich stromy sjednotit. Nalezneme proto kořeny obou stromů a pokud nejsou stejné, připojíme kořen o menším počtu pater jako potomka druhého kořene. Aktualizujeme počet pater u nového kořene a nastavíme všechny prošlé uzly stromu jako potomky nového kořene. Pokud je jen jeden z dvojice nově nalezených sousedů jádro, připojíme nejádrový bod do stromu k jádrovému, jen pokud ještě do žádného stromu (shluku) nepatří. Dva nejádrové sousedy samozřejmě neslučujeme do jednoho stromu. Poznamenejme, že určení, který prvek je jádro a který ne, musíme provést ještě před samotnou konstrukcí těchto stromů metodou hledání sousedů, kdy u každého bodu zaznamenáváme pouze počet sousedů. Po sestavení tohoto stromu je identifikátor shluku každého bodu dán kořenem jeho stromu.

(14) Věnujme nyní pozornost časové složitosti algoritmu. Konstrukce stromu algoritmem Strom++ může trvat až $O(n^2)$, neboť list do kterého přidáváme další bod může obsahovat téměř všechny ostatní body a relativní chyba může být stále překračována a tím volán k-means. Při zjišťování sousedů se také může stát, že nastavené ε bude tak vysoké, že všechny body budou navzájem sousedy, tedy bude zapotřebí opět $O(n^2)$ operací. Samotné procházení stromu při hledání sousedů se zřejme „vleze“ do času $O(nu)$, kde u je počet uzlů stromu (díky vyhodnocování lokálních podmínek), v případě vyváženosti stromu tedy nepřekročí $O(n \ln n)$. Nyní je třeba se ještě vyjádřit k časové složitosti slučování stromů, reprezentující shluky. Popsaný mechanismus je obsahem struktury dat pro rozklady množin popsané na http://en.wikipedia.org/wiki/Disjoint-set_data_structure. Na téže stránce lze také nalézt složitost všech operací konaných s těmito stromy. Je třeba provést t operací sloučení s n body. Pak složitost bude $O(t\alpha(n))$, kde $\alpha(n)$ je inverzní Ackermannova funkce, jejíž definici lze nalézt na http://en.wikipedia.org/wiki/Ackermann_function. Tato funkce roste tak pomalu, že ji v našem případě můžeme považovat za konstantu. t může být v nejhorším případě, kdy by byly všechny body navzájem sousedící jádra řádu $O(n^2)$, celkově tak bude složitost operací se stromy nejhůře řádu $O(n^2)$ $O(n^2)$

$O(n^2)$. To znamená, že celý tento algoritmus bude v nejhorším též řádu $O(n^2)$.

(15) Přistupme nyní k požadavkům na uživatelské vstupy a výstupy algoritmu. Uživatel by měl zadávat pouze hodnoty ε a m algoritmu DBSCAN. Maximální relativní chyba uzlů při počítání stromu uvedeným hierarchickým shlukovacím algoritmem bude spočítána programem dle rovnice (6.23). Výstupem by měl být seznam shluků (včetně výpisu jejich prvků), centroid ani kvadratická chyba nikoliv.

Shrňme a doplňme nyní ještě obecné požadavky na výstup programu.

6.3 Výstup programu

(1) Nejprve uveďme, jaké výstupy by měl program generovat po načtení dat, avšak ještě před spuštěním samotných shlukovacích algoritmů. Kromě celkkového počtu načtených bodů (genů) by měl být uživatel informován o centroidu všech bodů, celkové kvadratické chybě a maximální vzdálenosti nějakého bodu od centroidu. Tyto údaje mu potom mohou posloužit pro lepší nastavení parametrů shlukovacích algoritmů.

(2) Dále by program měl při načítání dat uživatele upozornit, u kterých bodů (genů) bylo nutné doplnit chybějící údaje. Během spuštění algoritmů by bylo dobré zobrazovat, jak velká část algoritmu již proběhla. U k-means tak průběžně zobrazovat po jistém počtu iterací, jaké již bylo dosaženo chyby a kolik iterací již proběhlo. Při tvorbě stromu pro hierarchické shlukování kolik bodů již bylo načteno. Konečně u shlukovacího algoritmu založeného na hustotě zobrazit ke kolika listům již byly nalezeni všichni sousedé (myšleno ke všem bodům v listu).

(3) Výstupem všech algoritmů bude samozřejmě seznam shluků, každý s počtem a seznamem svých prvků. U k-means a hierarchického algoritmu navíc bude u každého shluku uveden centroid a kvadratická či relativní chyba. U hierarchického algoritmu bude též umožněno vypsání struktury stromu, tedy vypsání předka a potomků každého uzlu. Nemělo by chybět ani opětovné vypsání uživatelských vstupních parametrů. Ty byly popsány v předchozích sekcích.

Tímto uzavřeme teoretickou část této práce. V další kapitole se budeme věnovat implementaci uvedených algoritmů.

7 Implementace programu

Program je imlementován v programovacím jazyce Java, standardní edici verze 6 (Java SE 6). Nepoužívá žádné přídavné knihovny. Tento jazyk byl vybrán z následujících důvodů. Implementace v jazyce Java má nespornou výhodu v tom, že po přeložení jejích zdrojových kódů do tzv. bytecode, lze tento bytecode spustit na většině známých operačních systémů, které zahranují platformy jako Windows, Linux či Mac OS X. Konkrétně lze bytecode sputit na systémech, pro které je k dispozici Java Virtual Machine (JVM), jejich seznam lze nalézt na http://java.com/en/download/manual.jsp (o JVM více v následující kapitole). Další výhodou je velké množství balíčků obsažených ve standardní edici, které umožňují tvorbu grafického uživatelského rozhraní a práci s vlákny. Rovněž umožňuje používání tzv. generických typů (generics), což jsou speciální typy, jejichž namapování na standardní typy se provádí až v době běhu programu. Připomeňme, že Java je objektový jazyk, veškerý kód se zde musí nacházet uvnitř tříd. Vlastní implementace probíhala ve volně stažitelném (open-source) integrovaném vývojovém prostředí Netbeans, verze 6.8 (viz http://netbeans.org).

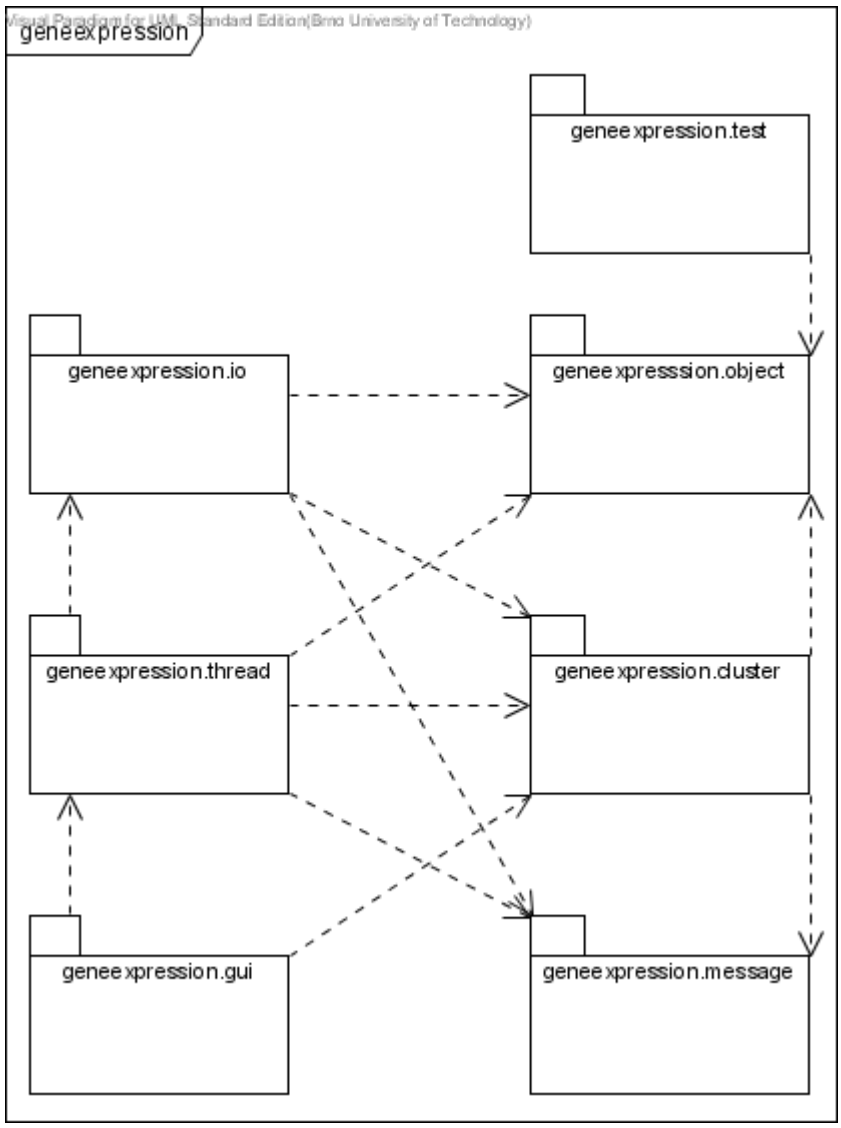
Celá implementace je přehledně rozdělena do javovských balíčků (package), které teprve obsahují třídy. Pro lepší názornost uvedeme UML diagramy tříd každého balíčku. V programu je značně používáno generických typů tříd, to je v UML zohledněno nestandardním způsobem, který je však snadno pochopitelný pro čtenáře. Standardní notace by diagramy příliš zesložitila a tím znepřehlednila. Rovněž privátní atributy tříd, které mají definované set i get metody bez omezení budeme označovat jako veřejné a výpis příslušných set/get metod vynechávat. UML diagramy byly vytvořeny v prostředí Visual Paradigm for UML, Standard edition, verze 7.1. Toto prostředí je komerčním produktem firmy Visual Paradigm (viz http://www.visual-paradigm.com/) a studentům školy je zdarma poskytnuta licence k jeho používání.

Program obsahuje implementaci algoritmů popsaných v předešlé kapitole. Kromě toho obsahuje též testovací třídy pro generování vhodných vstupních dat a grafické uživatelské rozhraní pro snadné ovládání programu (ovládání přes příkazový řádek není umožněno).

Celá implementace je uzavřena v balíčku geneexpression, který přímo neobsahuje žádnou třídu. Balíček je však dále rozdělen do 7 balíčků, které již třídy obsahují. Žádné další balíčky nejsou přítomny. Zjednodušený UML diagram balíčků je na obrázku 11. Balíček object obsahuje definice tříd pro základní objekty jako je gen nebo mikročip a příslušná rozhraní. Balíček test slouží pro generování vhodných vstupních dat, na kterých lze algoritmy otestovat. Balíček cluster obsahuje třídy reprezentující shluky vzniklé různými shlukovacími algoritmy, shlukované body a výsledky těchto algoritmů. Mimo to obsahuje implementaci samotných algoritmů. Balíček io poskytuje třídy s metodami pro načtení a zápis mikročipů z a do souboru a pro zápis výsledných shluků do souboru. Balíček thread obsahuje třídy reprezentující vlákna, v kterých běží shlukovací algoritmy a obslušnou třídu, která tato vlákna spouští a řídí. Balíček message definuje rozhraní, která umožňují komunikaci běžícího algoritmu s uživatelem. Konečně balíček gui obsahuje třídy užité pro grafické uživatelské rozhraní.

V následujících sekcích popíšeme každý balíček podrobněji. Upozorněme, že v definicích mnohých tříd se objeví generický typ E. Nebude-li řečeno jinak, bude tento generický typ určovat třídu implementující rozhraní SpaceElement<E> (viz sekce 7.1.1 na straně 37). Rovněž nebudeme v textu podrobně vypisovat výčet parametrů a návratových hodnot u metod, neboť toto je uvedeno v UML diagramech. Poznamenejme, že v textu se rovněž často vyskytuje pojem relativní chyba. Tím je míněna celková kvadratická chyba (nějaké množiny prvků) dělená počtem prvků (této množiny). Pod jménem bodu budeme rozumět název genu, jehož souřadnice genové exprese tomuto bodu odpovídají.

Dodejme, že následující popis rozhodně nemůže sloužit jako ucelená dokumentace ke všem implementovaným třídám a jejich metodám. Takovou dokumentaci lze najít přímo ve zdrojových kódech či vygenerovaném javadocu, což je HTML dokumentace generovaná z komentářů ve zdrojových kódech.



Ilustrace 11: Diagram balíčků programu

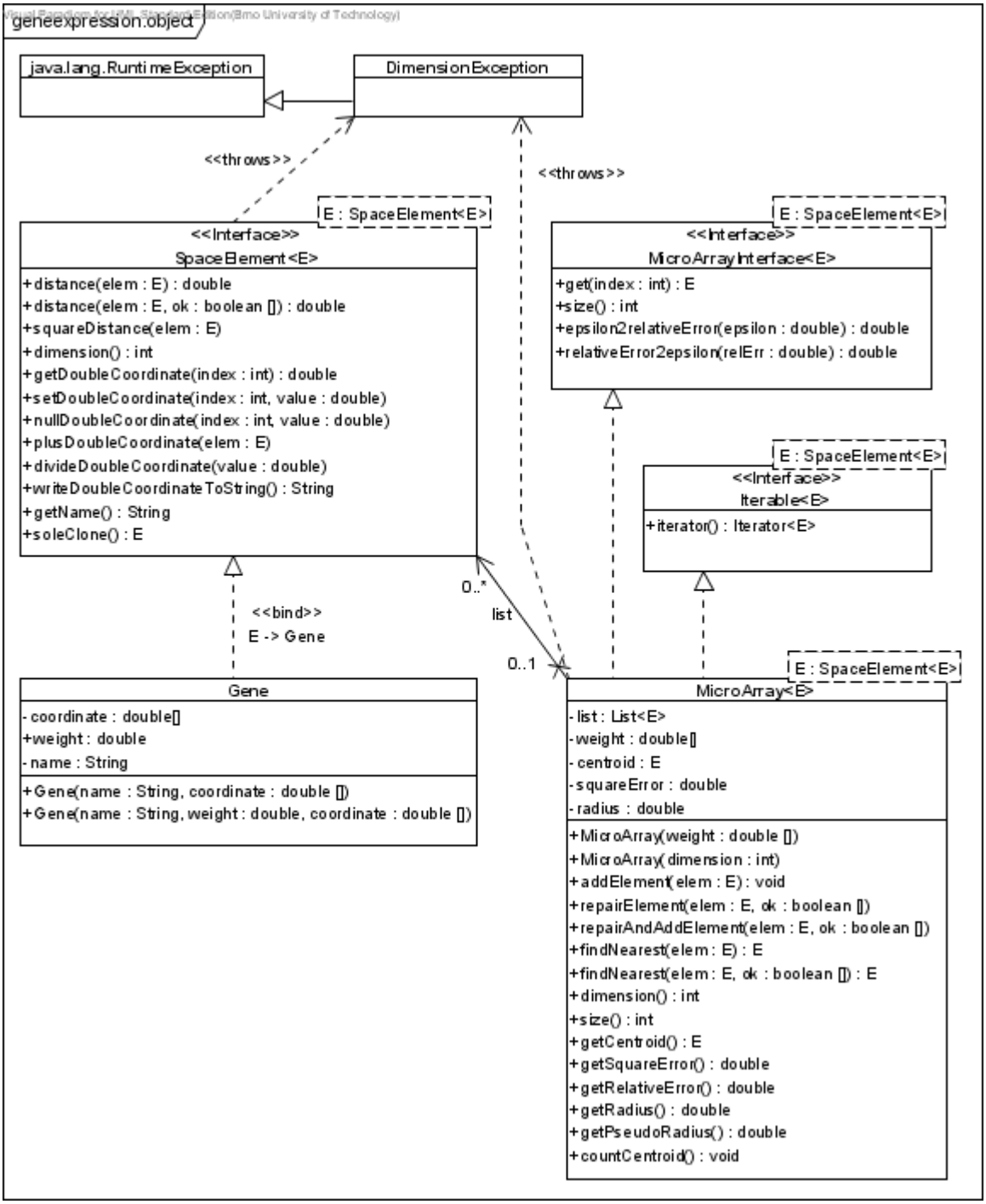
7.1 balíček object

Balíček obsahuje rozhraní SpaceElement reprezentující bod prostoru a MicroArrayInterface reprezentující skupinu bodů. Tato rozhraní jsou zde implementovány třídami Gene reprezentující jeden gen (popis s hodnotami genové exprese) a MikroArray reprezentující mikročip. Tyto třídy však nejsou ve shlukovacích algoritmech používány přímo, nýbrž jsou používany přes ona rozhraní. To umožňuje shlukovat jakékoliv body reprezentované třídami implementující rozhraní SpaceElement, tedy např. shlukovat vzorky. Následuje podrobnější popis tříd a rozhraní. UML diagram pro snadnější orientaci nalezneme na obrázku 12.

7.1.1 rozhraní SpaceElement<E>, třída Gene

Rozhraní SpaceElement<E> reprezentuje bod prostoru. Generický typ E definuje konkrétní typ bodu (určený třídou). To je potřeba, neboť jsou zde definovány operace, které lze provádět pouze mezi stejným typem bodů (např. vzdálenost). Rozhraní definuje metody distance, resp. squareDistance počítající Euklidovskou vzdálenost k jinému bodu, resp. její kvadrát, metodu dimension vracející dimenzi bodu. Dále jsou přítomny metody se sufixem DoubleCoordinate

manipulující s reálnými souřadnicemi bodu. Pomocí metod writeDoubleCoordinateToString a getName je možné vypsat do Stringu (javovský řetězec znaků) hodnoty reálných souřadnic a název bodu. Konečně metoda soleClone slouží pro klonování bodu, ovšem do nového bodu mají být překopírovány pouze hodnoty souřadnic.



Ilustrace 12: Balíček object

Konkrétní implementaci SpaceElement<E> poskytuje třída Gene, přičemž zde generický typ E = Gene. Implementuje všechny metody definované v SpaceElement, navíc poskytuje set/get metody pro nastavení jména a váhy. Třída má 3 atributy: name (jméno), weight (váha), coordinate (hodnoty reálných souřadnic). Konstruktury Gene vyžadují zadání těchto hodnot parametrem. Pouze váha může být implicitně považována za rovnu 1.

7.1.2 rozhraní MicroArrayInterface<E>, třída MicroArray<E>

Rozhraní MicroArrayInterface<E> reprezentuje množinu bodů prostoru. Poskytuje metody size určující jejich počet a get, která získá konkrétní bod dle jeho indexu v množině. Kromě toho poskytuje metody epsilon2relativeError a relativeError2epsilon pro přepočítání poloměru ε a relativní chyby E/a dle vzorce (6.23) na straně 33.

Rozhraní MicroArrayInterface<E> je plně implementováno třídou MicroArray<E>. Obsahuje atributy list (seznam bodů), weight (váhy jednotlivých souřadnic bodů). Dále obsahuje atributy charakteruzjící statistiky souboru: centroid, squareError (celková kvadratická chyba), radius (max. vzdálenost nějakého bodu od centroidu). Konstruktor třídy vyžaduje pouze zadání dimenze. Obsaženy jsou dále metody addElement pro přidávání dalšího bodu do množiny, repairElement pro doplnění chybějících souřadnic jak je pojednáno v odstavci (3) části 6.1 na straně 26, findNearest pro nalezení nejblížšího prvku v množině k danému prvku. Tato metoda může mít parametr boolean[ ] ok, pokud je i-tý prvek v poli false, i-tá souřadnice hledaného a daného prvku budou považovány za rovné (bez výpočtu). To je potřeba pro metodu repairElement. Dále jsou přítomny get metody pro vypsání dimenze, velikost, centroidu, kvadratické a relativní chyby, radiu a pseudoradiu R vypočteného z relativní chyby dle vzorce 6.22 na straně 33 (tedy R = ½ ε). Konečně metodou countCentroid jsou statistiky o množině přepočítány (po addElement se totiž neaktualizují).

7.1.3 výjimka DimensionException

Tato výjimka bývá vyvolána v některých metodách tříd a rozhraní tohoto balíčku a balíčku io, je-li zadán jako parametr bod s jinou dimenzí (jiným počtem reálných souřadnic), než jakou metoda předpokládá.

7.2 balíček test

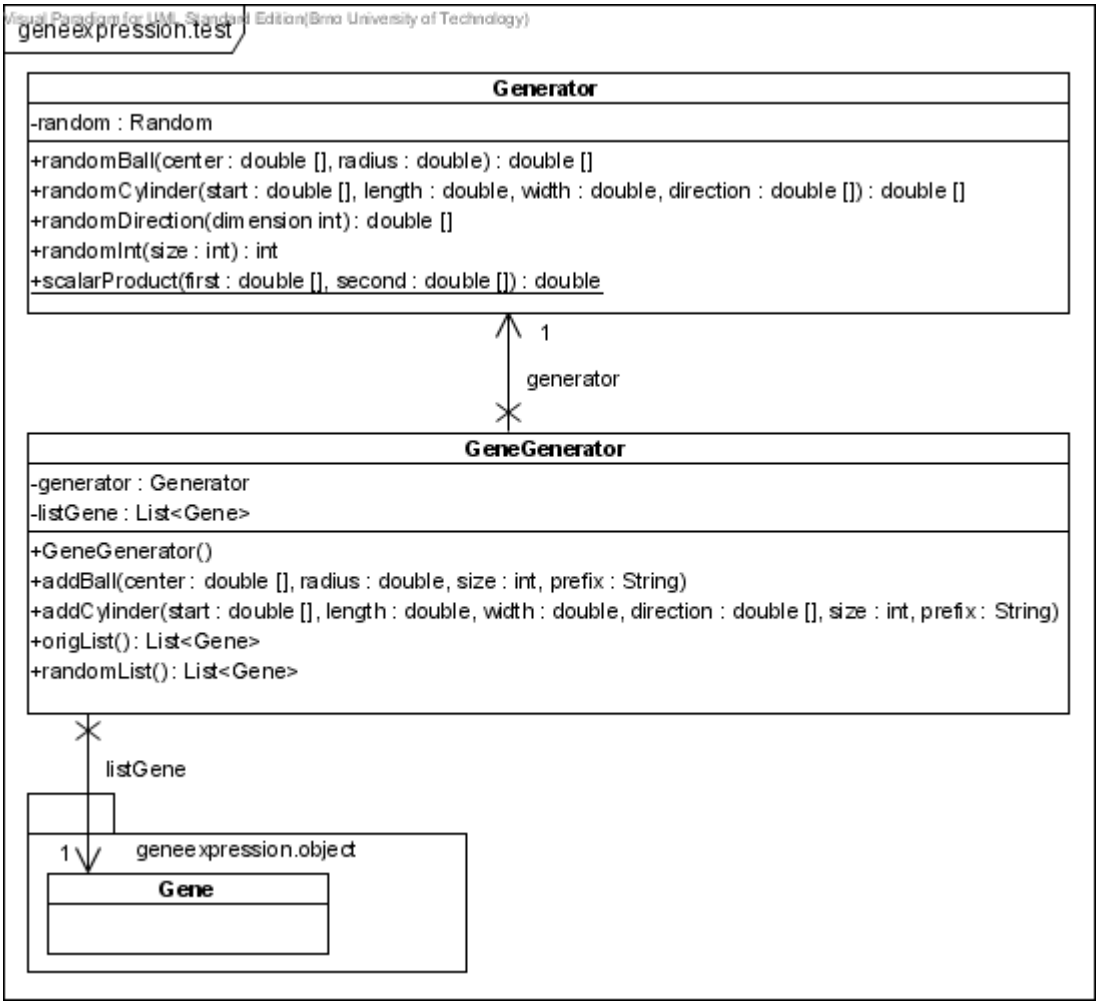
Tento balíček je používán pro generování množin bodů pomocí třídy Generator nebo množiny genů (objektů třídy Gene) pomocí třídy GeneGenerator, na kterých lze testovat a porovnávat shlukovací algoritmy. Umí generovat množinu bodů rovnoměrně rozložených v zadané kouli nebo válci, včetně jejich sjednocení. UML diagram je zobrazen na obrázku 13.

7.2.1 třída Generator

Třída slouží pro generování náhodných souřadnic bodu uvnitř zadané koule nebo válce, či náhodných souřadnic směru. Pro generování náhodných reálných a celých čísel využívá třída svůj jediný atribut random. Konstruktor je bezparametrický.

Metodou randomBall vygenerujeme náhodné souřadnice bodu uvnitř (hyper)koule se zadaným středem (o libovolné kladné dimenzi) a poloměrem. V rámci generování bodu je využito (hyper)sférických souřadnic (viz http://en.wikipedia.org/wiki/N-sphere), čímž je zajištěno, že žádný bod nemá vyšší pravděpodobnost vygenerování než ostatní. Konkrétně je pomocí random vybrána náhodná hodnota vzdálenosti od středu a sférických úhlů. Metoda randomDirection slouží pro vygenerování náhodného směru (tedy vektoru o jednotkové velikosti) zadané dimenze. Metodu využívá týž princip jako randomBall, pouze nevolí hodnotu vzdálenosti od středu [0,0,...,0], ale nastaví ji na 1.

Metodou randomCylinder vygenerujeme náhodné souřadnice bodu uvnitř (hyper) válce o zadaném středu základny, výšce, směru výšky (vektoru) a poloměru. Metoda nejdříve náhodně vybere rovinu kolmou na osu válce, protínající válec. Tím vytne rovina ve válci (hyper)kruh, v němž pomocí randomBall vygenerujeme bod. Metoda scalarProduct slouží pro počítání skalárního součinu.



Ilustrace 13: Balíček test

7.2.2 třída GeneGenerator

Tato třída slouží ke generování celé množiny bodů, reprezentující geny, náhodně (rovnoměrně) rozesetých uvnitř (hyper)koule nebo (hyper)válce. Má atribut generator pro generování jednotlivých bodů a listGene, kde udržuje vznikající množinu. Konstruktor je bezparametrický.

Metodou addBall přidáme do množiny zadaný počet genů (bodů) se zadaným prefixem jména (další část jména tvoří pořadí vygenerování), uvnitř zadané koule. Podobně metoda addCylinder přidá zadaný počet genů do množiny uvnitř zadaného válce. Konečně metodou randomList můžeme náhodně zamíchat pořadí genů v listGene.

7.3 balíček geneexpression.cluster

V tomto balíčku jsou implementovány všechny shlukovací algoritmy uvedené v 6. kapitole, jejichž spouštění zajišťují metody třídy Clustering<E>. Body uvnitř shluků jsou reprezentovány rozhraním ClusterElement<E>, které dědí z rozhraní SpaceElement<E>. Toto rozhraní je implementováno třídou ClusterPoint<E> a její specializací ClusterPointDensity<E> určenou pro shluky vzniklé pomocí algoritmu DBSCAN založeném na hustotě (viz část 4.2.3.1 na str. 18) .

Obecný typ reprezentující shluk je abstraktní třída Cluster<E>. Z této třídy dědí ClusterDensity<E> pro shluky počítané pomocí DBSCAN a abstraktní třída ClusterWithCentroid<E>,

pro obecný shluk, u kterého, jak již název napovídá, je počítán též centroid. Z ClusterWithCentroid<E> poté dědí třídy ClusterKMeans<E> pro shluk vzniklý pomocí k-means (viz 4.2.1.1 na str. 15) a abstraktní ClusterDivisive<E>. Tato abstraktní třída zastřešuje uzly stromu (což jsou také shluky), který je výsledkem algoritmu Strom++ popsaného v sekci 6.2.2 od strany 28. Z ClusterDivisive<E> potom dědí ClusterDivisiveNode<E> a ClusterDivisiveLeaf<E> pro vnitřní uzel a list stromu.

Výsledek každého shlukovacího algoritmu je reprezentován abstraktní třídou ClusterResult<E>. Z ní dědí ClusterDivisiveResult<E> pro výsledek algoritmu Strom++ a abstraktní ClusterResultWithCollection<E>. Tato abstraktní třída je předkem tříd ClusterDensityResult<E> a ClusterKMeansResult<E> pro výsledky algoritmu DBSCAN (před nímž mohl proběhnout algoritmus Strom++) a k-means. Z ClusterKMeansResult<E> ještě dědí ClusterDivisiveKMeansResult<E> pro výsledek algoritmu Strom++ následovaného k-means.

Dodejme, že ve zdrojových kódech jsou ještě další třídy pro aglomerativní shlukování a přímý DBSCAN, který ukládá seznam sousedů. V průběhu testování se však ukázalo, že tyto algoritmy jsou natolik náročné na paměť, že jsou prakticky nepoužitelné.

Z důvodu velkého počtu tříd v tomto balíčku nejsou všechny zobrazeny v jednom UML diagramu. Celkový diagram tříd balíčku je tak nahrazen diagramem tříd pro každou podsekci této sekce. Všechny třídy balíčku (bez vnitřního popisu) jsou však zobrazeny v UML diagramu třídy Clustering<E> (viz obrázek 18 na straně 49).

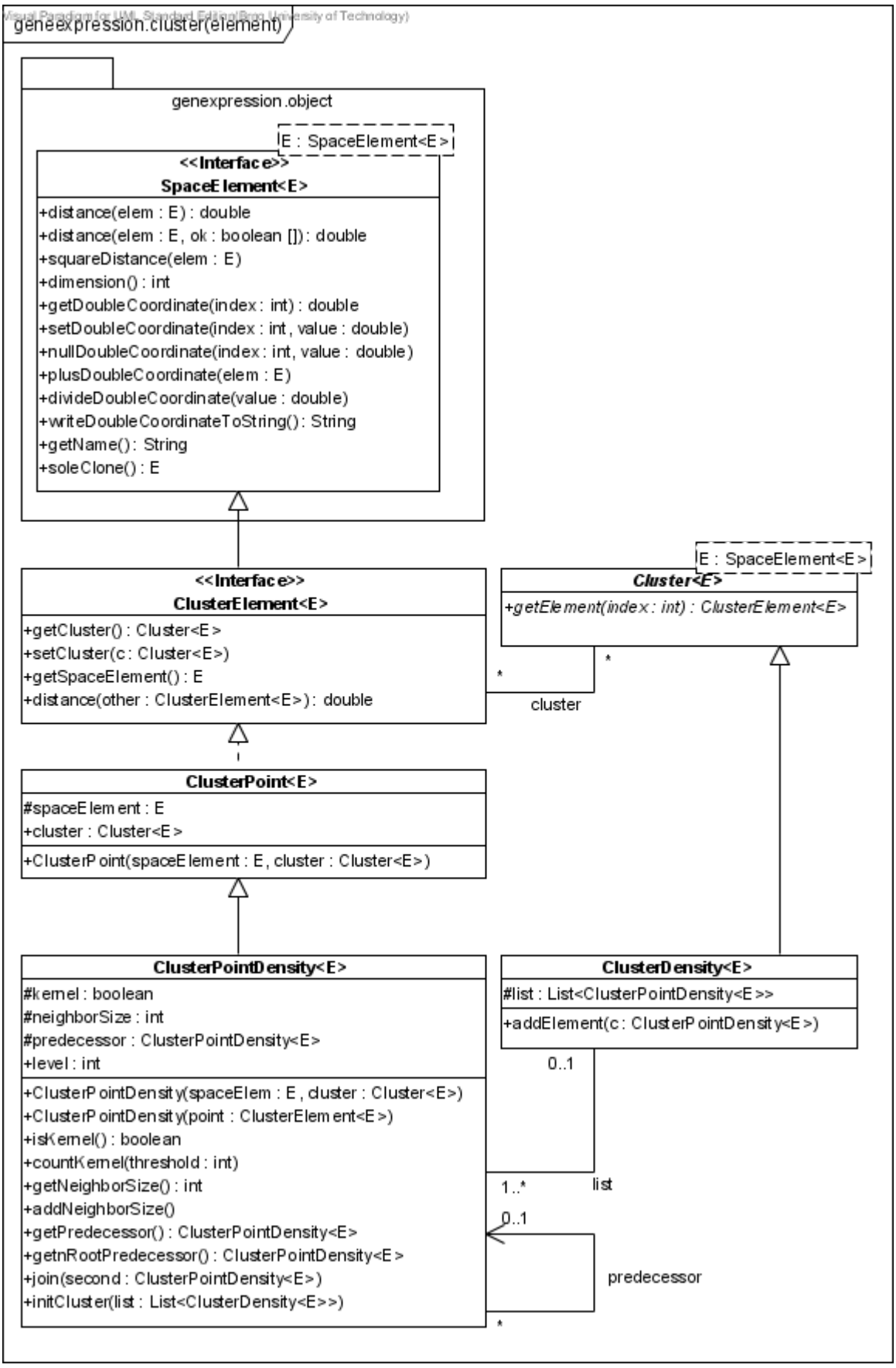
7.3.1 rozhraní ClusterElement<E> a jeho implementace

UML diagram rozhraní ClusterElement<E> a jeho implementujících tříd v kontextu balíčku je na obrázku 14. Toto rozhraní definuje obecný bod prostoru nacházející se uvnitř nějakého shluku, je tedy jistou obálkou bodu. Dědí z rozhraní SpaceElement<E> (viz část 7.1.1 na str. 37) a v podstatě přidává jen odkaz na shluk, ve kterém se bod nachází. Z tohoto důvodu se zde nachází metody getCluster a setCluster. Metodou getSpaceElement lze získat odkaz přímo na bod prostoru (bez této obálky). Je dodefinována metoda distance pro vzdálenost mezi ClusterElementy.

Třída ClusterPoint<E> plně implementuje rozhraní ClusterElement<E>. Obsahuje atributy spaceElement, což je bod samotný (bez této obálky) a cluster, do něhož bod patří. Konstruktor požaduje zadání těchto dvou atributů, cluster může být null.

Třída ClusterPointDensity<E> dědí z ClusterPoint<E>. Jedná se o bod shluku vzniklého shlukovacím algoritmem DBSCAN založeným na hustotě. Tento algoritmus využívá atributů a metod těchto bodů již při konstrukci samotných shluků. Atribut kernel určuje, zda se jedná o jádro, neighborSize počet bodů v ε-okolí bodu. K těmto atributům jsou definovány get metody isKernel a getNeighborSize. Metoda countKernel nastaví správně atribut kernel, podle toho, zda počet sousedů přesahuje nebo je roven zadané mezi. Metod addNeighborSize zvedne počet sousedů o 1.

Jak jsme uvedli v odstavci (13) části 6.2.3 na straně 34, pro snadné sjednocování podshluků konečných shluků algoritmu DBSCAN, jsou tyto podshluky organizovány do stromové struktury. Každý bod v podshluku má definován odkaz na bod ležící ve směru kořene stromu. Pomocí řetězce těchto odkazů se lze z každého bodu podshluku dostat do kořene stromu (reprezentující tento podshluk). Tímto odkazem je právě atribut predecessor (pro kořen má hodnotu null). Atribut level pak určuje délku nejdelší větvě vycházející z kořene směrem dolů (pro nekořenové body nemá smysl). Metoda getPredecessor vrací onen predecessor, metoda getRootPredecessor projde řetězcem odkazů až ke kořeni stromu podshluku a tento kořen (bod v něm) vrací. Přitom změní všechny nepřímé potomky kořene, kterými prošla, na přímé. Metoda join sjednotí strom (podshluk) příslušný tomuto bodu se stromem (podshlukem) příslušným zadanému bodu, pokud je tento i zadaný bod jádro. Pokud je jádro právě jeden z těchto dvou bodů, nastane sjednocení, jen pokud nejádrový bod ještě není v žádném podshluku. Pro dva nejádrové body sjednocení nenastává.

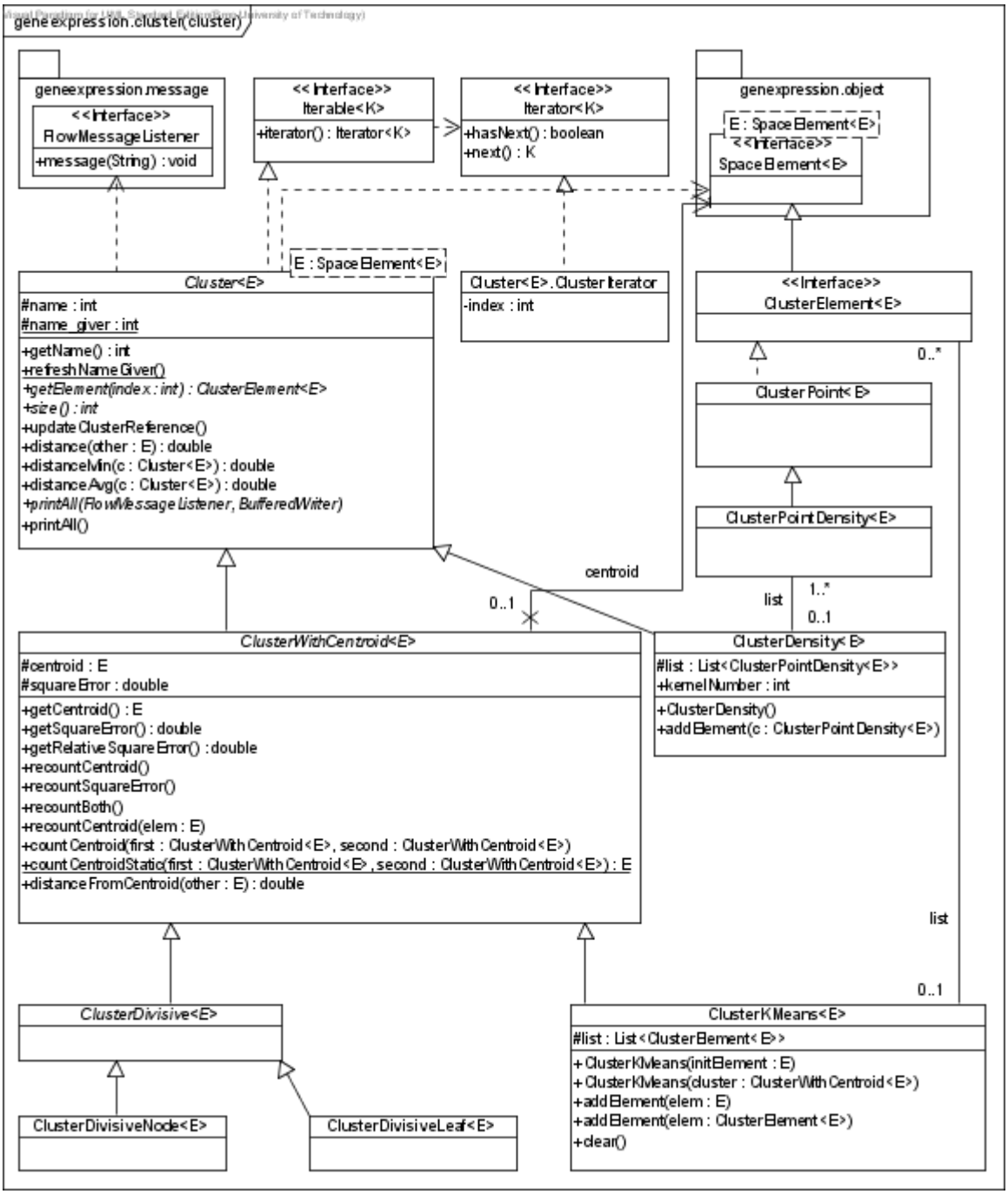


Ilustrace 14: Rozhraní ClusterElement<E> a jeho implementace v balíčku cluster

Konečně metoda initCluster vytváří z vzniklých stromů (podshluků) konečné shluky. Metoda v podstatě přidává body do existujících shluků, případně vytváří shluky nové. K tomuto bodu najde tedy metoda kořen a zjistí, zda již tento kořen patří do nějakého shluku. Pokud ne, je vytvořen nový shluk, který je přidán do parametrem zadaného seznamu shluků. Do nalezeného nebo nového shluku jsou poté přidány všechny body na cestě od tohoto bodu ke kořeni (pokud tam již nejsou).

7.3.2 třída Cluster<E> a její potomci

UML diagram třídy Cluster<E> a jejích potomků v kontextu balíčku je na obrázku 15.



Ilustrace 15: Třída Cluster<E> a její potomci v balíčku cluster

Abstraktní třída Cluster<E> reprezentuje obecný shluk. Atributem name je jméno (celé číslo) a statický name_giver, který udržuje poslední (nejvyšší) přidělené jméno. Jména jsou přiřazována shlukům automaticky vzestupně a popořadě dle hodnoty name_giver. Metodou refreshNameGiver lze name_giver nastavit na 1. Tato jména program nikde nepoužívá (kromě výpisů), slouží pouze pro snažší orientaci uživatele ve výpise shluků. Abstraktní metoda size vrátí počet bodů ve shluku, abstraktní getElement vrátí bod s daným indexem. Metoda updateClusterReference aktualizuje odkazy (atribut cluster rozhraní ClusterElement<E>) v bodech shluku na tento shluk. Tyto odkazy totiž nejsou některými shlukovacími algoritmy udržovány aktuální (pokud je program nepotřebuje), ve výsledcích metod třídy Clustering<E> však tyto odkazy nastavené správně jsou. Metoda distance slouží pro nalezení (minimální) vzdálenosti (bodů) shluku od zadaného bodu. Metody distanceMin a distanceMax potom k nalezení minimální nebo průměrné vzdálenosti shluků dle vzorců 4.13 a 4.15 na straně 14. Konečně metody printAll s parametry slouží k výpisu informací o shluku a jmen jeho bodů, parametry a způsob jejich použití je identický s parametry metod printAll popsaných v sekci o třídě ClusterResult<E> (7.3.4 na str. 47).

Abstraktní třída ClusterWithCentroid<E> dědí z třídy Cluster<E>. Jak již název napovídá, jedná se o specializaci shluku, který ke svému výpočtu využívá svůj centroid. Obsahuje tedy atribut centroid a squareError, což je celková kvadratická chyba shluku definována jako součet čtverců vzdáleností bodů shluku od jeho centroidu (viz vzorec 4.19 na straně 16). Tyto atributy mají příslušné get metody. Metoda getRelativeSquareError vrátí relativní chybu shluku, tedy kvadratickou dělenou počtem bodů. Dále jsou přítomny metody recountCentroid, resp. recountSquareError, resp. recountBoth (bez parametrů) pro přepočítání centroidu, resp. kvadratické chyby, resp. obojího ze souřadnic bodů obsažených ve shluku (nejsou-li tyto udržovány stále aktuální). Tyto metody mají složitost O(n), kde n je počet bodů ve shluku. Metody recountCentroid, resp. recountSquareError, resp. recountBoth s parametrem přepočítají centroid, resp. kvadratickou chybu, resp. obojí po přidání daného prvku dle vzorců 6.5 a 6.13 na stránkách 29 a 30. Metody mají konstatní složitost. Metoda countCentroid, resp. countSquareError, resp. countBoth spočítá centroid, resp. kvadratickou chybu, resp. obojí tohoto (nového) shluku právě vzniklého sjednocením dvou zadaných shluků s centroidy. Výpočet opět probíhá dle vzorců 6.5 a 6.13 a má konstatní složitost. První dvě z posledních třech uvedených metod mají také své statické „kopie“ se sufixem Static, které počítají totéž, avšak výsledek neuloží do atributů instance, ale vrátí ho. Z důvodu úspory místa v UML diagramu je z každé trojice (dvojice) metod uvedena jen jedna. Konečně poslední metodou je distanceFromCentroid která vrátí vzdálenost daného bodu od centroidu.

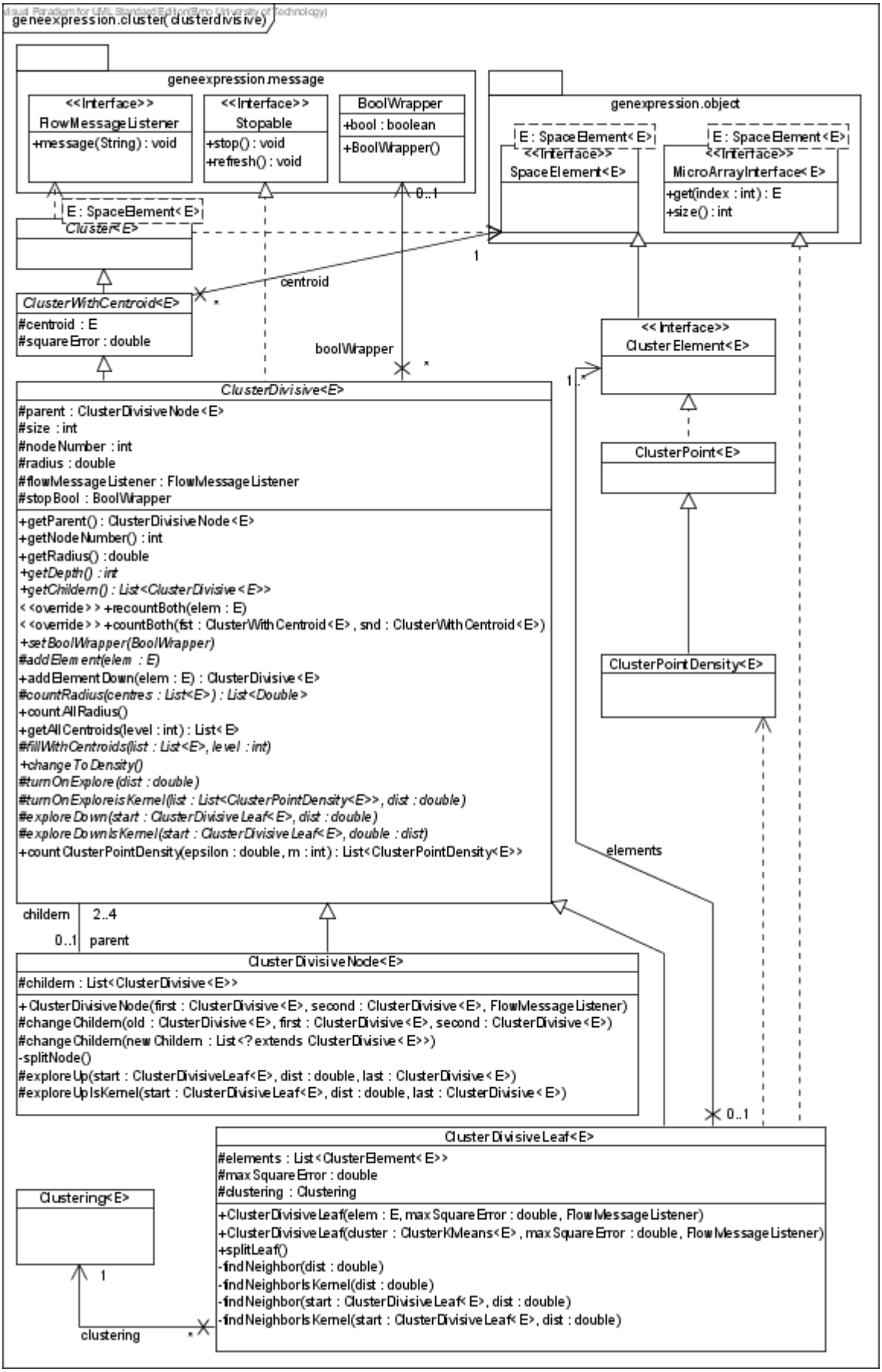
Třída ClusterDensity<E> plně implementuje abstraktní metody třídy Cluster<E>, která je jejím (přímým) předkem. Tato třída reprezentuje shluk vzniklý algoritmem DBSCAN. Navíc má atributy list se seznamem svých bodů (třídy ClusterPointDensity) a kernelNumber udávající počet jader v tomto shluku. Konstruktor je bezparametrický, navíc je přítomna metoda addElement umožňující přidat bod třídy ClusterPointDensity do shluku.

Třída ClusterKMeans<E> dědí z třídy ClusterWithCentroid<E> a plně doimplementovává její abstraktní metody. Tato třída reprezentuje shluk vzniklý pomocí k-means. Navíc obsahuje atribut list se seznamem svých bodů. Konstruktor vyžaduje zadání centroidu, který je pomocí metody soleClone parametru naklonován, není však do seznamu bodů přidán. Místo centroidu konstruktor též akceptuje zadání jiné ClusterWithCentroid<E> instance, jejíž centroid je naklonován. Přítomný jsou opět metody addElement pro přidání bodu do shluku (bez přepočítání centroidu a kvadratické chyby) a metoda clear pro vymazání všech bodů ze shluku.

Z třídy ClusterWithCentroid<E> také dědí abstraktní třída ClusterDivisive<E>. Tato třída včetně svých potomků je popsána v následující sekci. Poznamenejme, že z ClusterWithCentroid<E> také dědí třída ClusterAgglomerative<E>, která však není používána.

7.3.3 třída ClusterDivisive<E> a její potomci

UML diagram třídy ClusterDivisive<E> a jejích potomků v kontextu balíčku je na obrázku 16.



Ilustrace 16: Třída ClusterDivisive<E> a její potomce v balíčku cluster

Abstraktní třída ClusterDivisive<E> dědí z třídy ClusterWithCentroid<E> a představuje obecný uzel ve stromu počítáném shlukovacím algoritmem Strom++. Jejími dalšími atributy jsou tak parent – odkaz na předka (pro kořen null), size – počet bodů v podstromu, nodeNumber – počet uzlů v podstromu, depth – hloubka podstromu, radius – vzdálenost nejvzdálenějšího bodu v podstromu od centroidu. Pro tyto atributy jsou přítomné get metody. Další atributy jsou flowMessageListener – objekt pro zasílání hlášení o průběhu vykonávání metod (viz část 7.6.1 na straně 55) a stopBool, který svou proměnnou bool určuje, zda je požadávno zastavení aktuálně probíhající metody v kterémkoliv uzlu stromu. Třída tedy imlementuje rozhraní Stopable, jejíž metoda stop nastavuje stopBool.bool na true. Třída předefinovává metody ze svého předka se sufixem Both tak, aby správně aktualizovali i svou velikost. Pro účely zobrazení stromu v GUI je přítomna též metoda getChildern která vrátí seznam přímých potomků uzlu.

Následující metody se používají v metodě clusterDivisive třídy Clustering<E>., která imlementuje Strom++. Pomocí addElementDown lze do stromu přidat další bod, metoda vrátí odkaz na kořen aktualizovaného stromu. Samotné přidání prvku do stromu řeší metoda addElement, její průběh odpovídá popisu přidávání bodu do stromu v odstavcích (4) až (8) části 6.2.2 na stranách 28 až 30. Metodou countAllRadius je pak přepočítán radius každého uzlu ve stromu. Tato metoda využívá metodou countRadius, která nese v parametru seznam centroidů a vrací k němu seznam radiů pro příslušný podstrom. Každý uzel přidá do seznamu centroidů svůj centroid a zavolá metodu na všechny své potomky. Z vracených seznamů radiů poté vybírá maxima a předává je předkovi.

Metodu fillWithCentroids používá metoda clusterDivisiveKMeans třídy Clustering<E>, která implementuje k-means využívající předešlého výsledku Strom++ popsaný v odstavci (10) části 6.2.2 na straně 30, pro získání všech centroidů daného patra stromu.

Další metody slouží pro potřeby metody clusterDivisiveDensity třídy Clustering<E>, která implementuje DBSCAN využívající předešlého výsledku Strom++, popsaný v části 6.2.3 od strany 32. V listech jsou body udržovány v seznamu instancí obecně jakékoliv třídy dědící z rozhraní ClusterElement<E>. Metodou changeToDensity jsou v těchto seznamech body nahrazeny instancemi ClusterPointDensity<E>. Metoda turnOnExploreIsKernel a turnOnExplore spouští hledání jader mezi body ve všech listech podstromu a nastavování podshluků (algoritmu DBSCAN) reprezentovaných pomocí stromu (s uzly typu ClusterPointDensity<E>), jak je popsáno v odstavci (13) části 6.2.3 na straně 34. Hledání sousedních listů, popsané v odstavcích (7) až (10) v části 6.2.3 na stranách 33 až 34, imlementují metody exploreDown, exploreDownIsKernel, exploreUp a exploreUpIsKernel, z nichž poslední dvě jsou pouze v dědící třídě ClusterDivisiveNode<E>. Vzhledem k reprezentaci podshluků (DBSCANu) pomocí stromu je operace hledání a nastavování sousedů každému bodu (v listu) nahrazena dvěmi operacemi – hledání jader a nastavením počtu sousedů metodami s příponou IsKernel a spojováním podshluků (stromů) do shluků pomocí algoritmu uvedeného v odstavci (13) části 6.2.3 na straně 34. exploreUp(IsKernel) provádí jednu etapu hledání sousedních listů a exploreDown(IsKernel) potom jednotlivé iterace etapy popsané v odstavcích (8) a (9) části 6.2.3 na straně 33. Parametrem start je určen list, k němuž hledáme sousední listy, parametrem last uzel, ve kterém probíhala poslední provedená etapa. Konečně metodou countClusterPointDensity dostaneme seznam všech bodů ze všech listů podstromu, u kterých byly již určeny jádra a příslušné podshluky.

Z třídy ClusterDivisive<E> dědí třída ClusterDivisiveNode<E> reprezentující vnitřní uzel stromu. Má proto atribut childern se seznamem potomků – uzlů. Konstruktor vyžaduje zadání dvou potomků, jímž se stane tento předkem. Pro dělení uzlu na dva, jak je popsáno v odstavci (8) části 6.2.2 na straně 30, je implementována metoda splitNode. Metody changeChildern slouží pro nastavení nových potomků. Parametr old určuje potomka který bude smazán a first a second nové potomky místo něj. Pole newChildern určuje všechny nové potomky, všechny staré potomky metoda smaže. Metody exploreUp a exploreUpIsKernel jsou popsány v přechozím odstavci.

Druhým dědicem třídy ClusterDivisive<E> je ClusterDivisiveLeaf<E> reprezentující list stromu. Má atribut elements se seznamem bodů patřících do tohoto shluku. Atribut maxSquareError určuje maximální relativní chybu, při které má dojít k rozdělení listu (resp. „přeshlukování“

sourozenců). Atribut maxElement určuje maximální počet bodů v listu, 0 pro neomezený. Atribut clustering slouží ke spouštění k-means pro rozdělování listu. Konstruktory vyžadují zadání maximální relativní chyby listu, maximálního počtu bodů v listu a flowMessageListener pro výpis hlášení o průběhu metod. Dále konstruktor vyžaduje zadání prvního bodu nebo celého shluku bodů – výsledku algoritmu k-means. Je přítomna metoda splitLeaf, která vezme body ze všech j listů (tohoto a jeho sourozenců) a spustí na ně k-means pro k = j+1, jak je popsáno v odstavci (7) části 6.2.2 na straně 30.

Další metody slouží pro algoritmus DBSCAN. Metody findNeighborIsKernel provádí hledání jader v listu nebo mezi listem a daným jiným listem. Konečně metody findNeighbor provádí sjednocování podshluků, do kterých patří body, pomocí volání metody join třídy ClusterPointDensity<E> na nalezené sousedy. Tyto metody jsou spouštěny metodami exploreDown a exploreDownIsKernel třídy předka ClusterDivisive<E>.

7.3.4 třída ClusterResult<E> a její potomci

UML diagram třídy ClusterResult<E> a jejích potomků v kontextu balíčku je na obrázku 17. Abstraktní třída ClusterResult<E> reprezentuje obecný výsledek shlukovacího algoritmu, který obsahuje mimo jiné seznam nalezených shluků. Definuje metody pro zápis tohoto výsledku např. do souboru nebo Stringu. Třída implementuje rozhraní Stopable (viz část 7.6.2 na str. 55), což znamená, že vykonávání některých jejích metod (bude uvedeno kterých) lze zastavit metodou stop. To je výhodné, pokud zápis trvá příliš dlouho a uživatel jej chce přerušit. Pro účel zastavení obsahuje třída atribut stopBool, jehož nastavení na true indikuje, že je požadováno zastavení prováděné metody. Dalšími atributy jsou minClusterSize a maxClusterSize určující minimální a maximální velikost shluku v nalezeném souboru shluků.

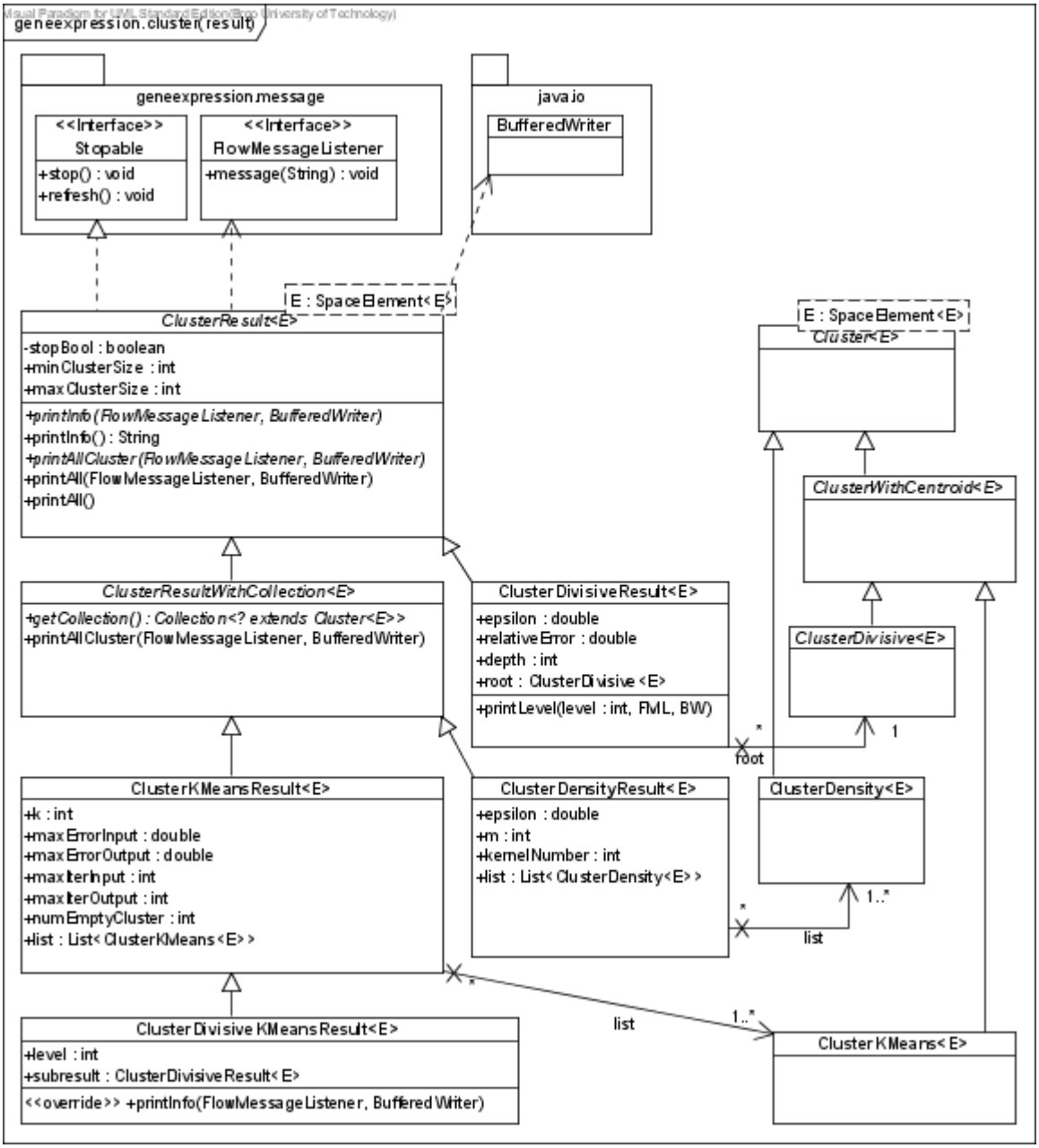
Abstraktní metoda printInfo s parametry vypíše základní informace o výsledku (hlavičku – např. počet shluků) do zadaného BufferedWriteru, což je standardní javovský objekt pro zápis řádků. Tento BufferedWriter může být napojen na výstupní soubor, ale také na String nebo textovou komponentu GUI. Případné informace o průběhu zápisu jsou zapisovány do zadaného FlowMessageListeneru jeho metodou message (viz část 7.6.1 na straně 55). Metoda printInfo bez parametrů zapíše totéž do nového Stringu, který pak vrací. Případná hlášení FlowMessageListeneru jsou zde ignorovány. Žádnou z metod printInfo() nelze zastavit pomocí metody stop.

Abstraktní metoda printAllCluster zapíše seznam všech shluků, včetně jejich hlavičky (obsahující např. počet prvků) a jmen bodů obsažených ve shluku do zadaného BufferedWriteru. Případné informace o průběhu jsou opět zapisovány do zadaného FlowMessageListeneru, vykonávání metody lze zastavit pomocí stop.

Konečně metoda printAll s parametry zapíše hlavičku celého výsledku a za ní seznam všech shluků, volá tak metody printInfo a printAllCluster. Metoda printAll bez parametrů slouží spíše pro programátora, neboť vypisuje vše (hlavičky, shluky i informace o průběhu) na standardní výstup. Obě metody lze zastavit voláním stop.

Z třídy ClusterResult<E> dědí abstraktní třída ClusterResultWithCollection<E>. Ta definuje abstraktní metodu getCollection, která má vracet kolekci všech shluků ve výsledku. Pomocí této metody třída implementuje zděděnou metodu printAllCluster. Nyní přejděme již k neabstraktním potomkům třídy ClusterResult<E>. Všechny tyto neabstraktní třídy samozřejmě implementují všechny metody třídy ClusterResult<E>.

Třída ClusterDivisiveResult<E> dědí přímo z ClusterResult<E> a reprezentuje výsledek shlukovacího algoritmu Strom++ (viz část 6.2.2 od strany 28). Obsahuje vstupní parametr algoritmu relativeError, čímž je maximální relativní chyba (listového) shluku. Dále obsahuje atribut epsilon, což je ε přepočítáné z relativní chyby relativeError dle vzorce 6.23 na straně 33. Atribut depth ukládá hloubku vzniklého stromu a root odkaz na jeho kořen. Přes tento kořen je poté možné dostat se k dalším uzlům ve stromu.



Ilustrace 17: Třída ClusterResult<E> a její potomci v balíčku cluster

Třída ClusterDensityResult<E> dědí z ClusterResultWithCollection<E> a reprezentuje výsledek algoritmu DBSCAN. Obsahuje tedy seznam výsledných shluků v atributu list. V atributu kernelNumber je uložen celkový počet jader ve všech shlucích. Atributy epsilon a m jsou vstupní parametry algoritmu, tedy poloměr ε-okolí a minimální počet sousedů v tomto ε-okolí.

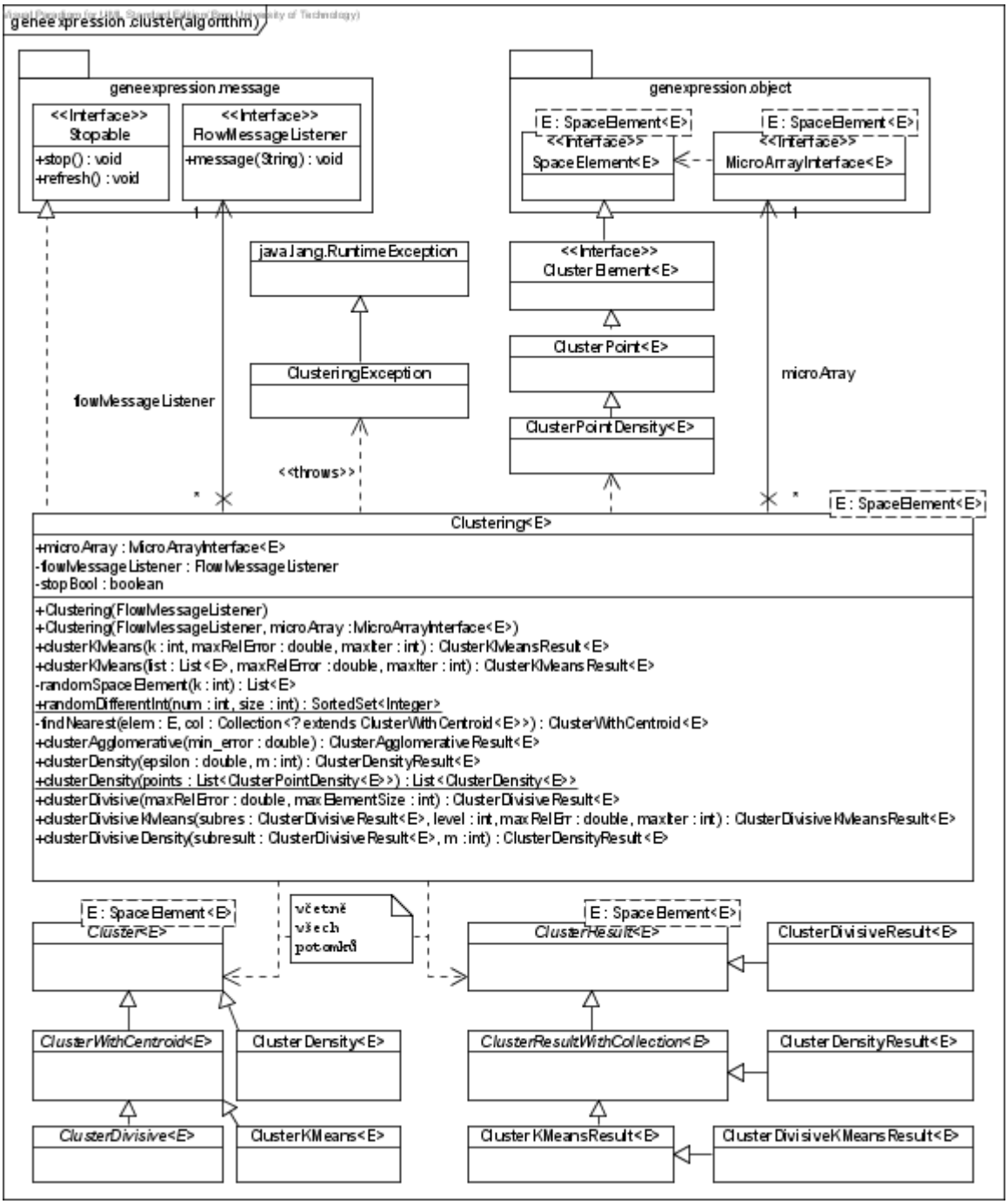
Třída ClusterKMeansResult<E> dědí z ClusterResultWithCollection<E> a reprezentuje výsledek algoritmu k-means. Obsahuje tedy seznam výsledných shluků v atributu list. Atributy k, maxErrorInput, maxIterInput uvádí vstupní parametry algoritmu, tedy počet shluků, maximální požadovaná relativní chyba (lze-li jí dosáhnout) a maximální počet iterací. Atributy maxErrorOutput a maxIterOutput potom uvádí dosaženou relativní chybu a provedený počet iterací.

Specializací třídy ClusterKMeansResult<E> je třída ClusterDivisiveKMeansResult<E>, která reprezentuje výsledek algoritmu k-means, kterému jako inicializační hodnoty centroidů posloužily centroidy uzlů daného patra stromu z algoritmu Strom++. Obsahuje tedy další atributy level určující

ono patro (0 pro nejvyšší) a subresult, což je odkaz na onen výsledek algoritmu Strom++. Z tohoto důvodu předefinovává metodu printInfo oproti předku.

7.3.5 třída Clustering<E>

UML diagram třídy Clustering<E> v kontextu balíčku je na obrázku 18.



Ilustrace 18: Třída Clustering<E> v balíčku cluster

Třída Clustering<E> implementuje s pomocí potomků třídy Cluster<E> všechny shlukovací algoritmy, které program umí. Třída implementuje rozhraní Stopable (viz část 7.6.2 na str. 55), což umožňuje pomocí implementované metody stop zastavit aktuálně prováděný algoritmus. Požadavek

zastavení je metodám indikován nastavením atributu stopBool na true. Metodou refresh lze požadavek zastavení zrušit (je potřeba volat, kdykoliv chceme po zastavení nějaký další algoritmus spustit). Třída obsahuje též atribut flowMessageListener, jejíž metodou message podává hlášení o průběhu vykonávání aktuálně běžícího algoritmu (viz část 7.6.1 na straně 55). Nejdůležitějším atributem je microArray, což je množina bodů, které mají být shlukovány. Konstruktory vyžadují zadání flowMessageListeneru, odkaz na microArray lze doplnit set/get metodami až před spuštěním nějakého shlukovacího algoritmu.

Metodou clusterKMeans spouštíme algoritmus k-means se zadaným k, požadavanou maximální chybou a maximální počtem iterací. V rámci této metody je metodou randomSpaceElement vygenerováno k náhodných centroidů z microArray pomocí vygenerování k náhodných hodnot indexů v rámci microArray statickou metodou randomDifferenInt. Kromě této metody clusterKMeans je ještě k dispozici její přetížení, kde se místo parametru k přímo zadává seznam inicializačních centroidů. Obě metody používají k nalezení nejbližšího centroidu ze seznamu centroidů metodu findNearest.

Věnujme ještě pozornost statické metodě randomDifferentInt. Ta má generovat k náhodných, navzájem různých celých čísel v rozsahu 0 (včetně) až size (mimo). Náhodností se myslí, že každá k-tice celých čísel má stejnou pravděpodobnost vygenerování. K tomu by bylo potřeba nejprve všechny čísla od 0 po size uložit do B+ stromu a poté náhodně vybírat směr cesty z kořene k listům a v těch náhodně vybrat prvek. Tento prvek pak v B+ stromu smazat a opakovat celý cyklus znovu od kořene. Implementace takového algoritmu však ve standardní Javě SE 6 chybí a vzhledem k náročnosti implementace a tomu, že k bývá malé a size velké, bylo přistoupeno k náhradnímu řešení, které negeneruje každou k-tici se stejnou pravděpodobností. Každé z k čísel je generováno takto. Nejprve je generováno náhodné číslo v rozsahu 0 až size. Pokud je toto číslo již obsaženo ve výsledném (seřazeném) seznamu čísel (dále jen podmínka), je maximálně 4-krát generováno další číslo v témže rozsahu a kontrolováno na stejnou podmínku. Pokud se ani poté nepodaří podmínku splnit, jsou postupně popořadě brána čísla větší než poslední vygenerované číslo, příp. menší.

Metoda clusterAgglomerative slouží ke spuštění aglomerativního hierarchického shlukovacího algoritmu, vzhledem k jejímu nároku na paměť a čas však bylo od jejího použití upuštěno. Stejně tak (instanční) metoda clusterDensity slouží k přímému spuštění algoritmu DBSCAN, avšak vzhledem k její časové náročnosti není používána. Její statické přetížení však používáno je, viz níže.

Poslední ze seznamu metod jsou metody využívající shlukovacího algoritmu Strom++. Metoda clusterDivisive přímo implementuje tento algoritmus, jako parametr má maxRelError jakožto maximální relativní chybu listů stromu a maxElementSize jakožto maximální počet bodů v listu (0 pro neomezený). Většina implementace metody je však skryta ve volaných metodách třídy ClusterDivisive<E> a jejích potomků. Metoda clusterDivisiveKMeans spouští algoritmus k-means, kde jako inicializační polohy centroidů jsou zvoleny centroidy uzlů (shluků) daného patra stromu z algoritmu Strom++. Metoda má tedy za vstup subres – výsledek metody clusterDivisive, level – patro stromu a maximální požadovanou relativní chybu a maximální počet iterací. Poslední metodou je clusterDivisiveDensity. Ta spouští algoritmus DBSCAN, avšak ne přímo na vstupních bodech, ale na stromu z algoritmu Strom++, jak je popsano v části 6.2.3 na straně 32. Jako parametry má tedy výsledek algoritmu Strom++ a požadovaný počet sousedů v ε-okolí. ε je převzato z proměnné epsilon výsledku ClusterDivisiveResult<E> (to vzniklo přepočítáním maxRelError dle vzorce 6.23 na straně 33). V rámci metody je vytvořen seznam bodů, tedy instancí třídy ClusterPointDensity<E>. Z něho jsou poté vytvářeny shluky pomocí statické metody clusterDensity způsobem popsaným v odstavci (13) části 6.2.3 na straně 34.

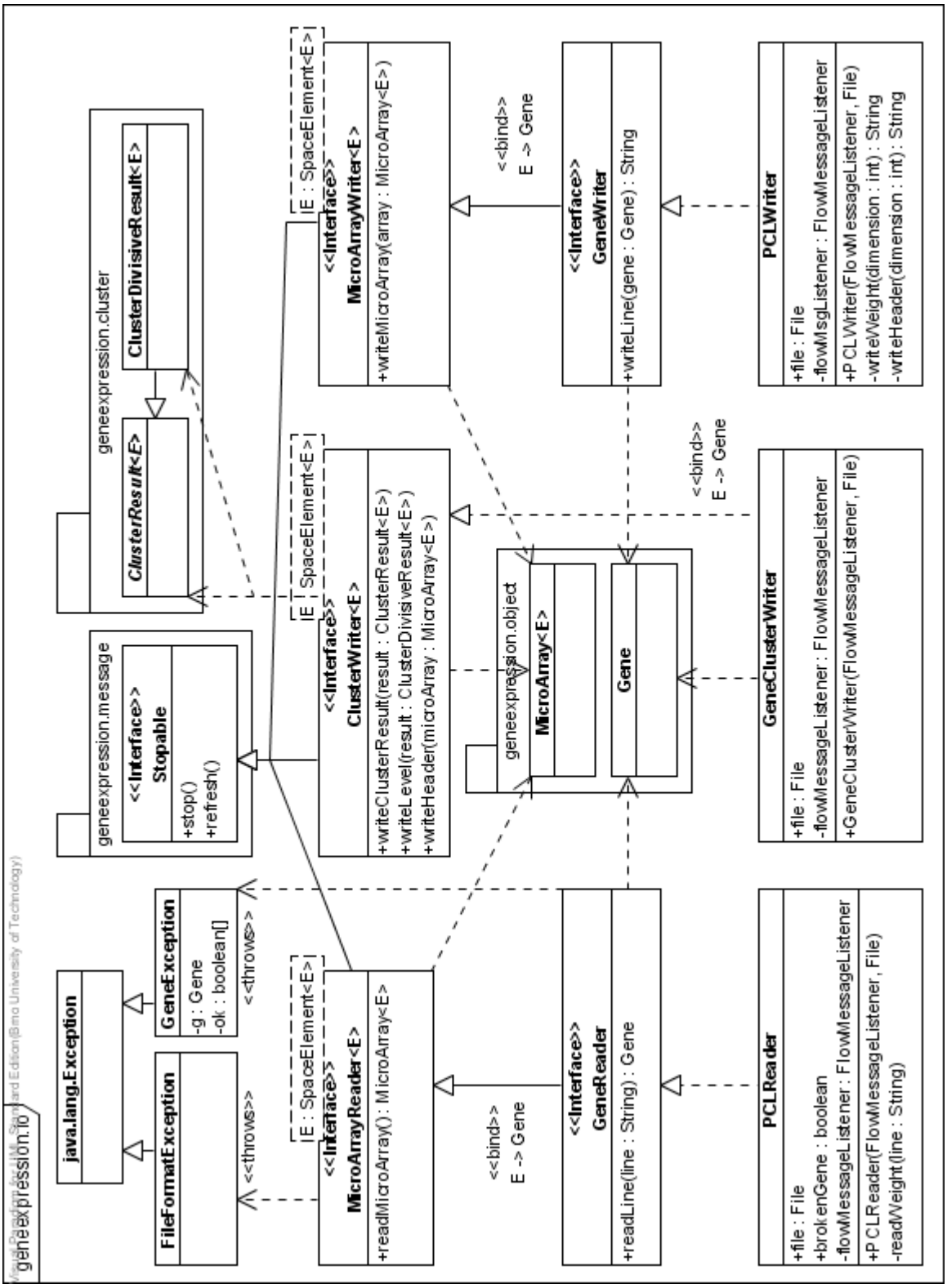
7.3.6 výjimka ClusteringException

Tato výjimka může být vyvolána při zadání nesprávných parametrů do některých metod třídy Clustering. Jmenujme konkrétní – při k v clusterKMeans větším než je počet bodů v microArray nebo

pokud je microArray prázdné v ostatních metodách spouštěcí shlukovací algoritmy.

7.4 balíček geneexpression.io

UML diagram balíčku je na obrázku 19.



Ilustrace 19: Balíček io

Tento balíček slouží k načtení bodů či genů ze souboru, jejich uložení do souboru a k zápisu shluků do souboru. Třída provádějící načtení bodů musí implementovat rozhraní MicroArrayReader<E>, zápis bodů MicroArrayWriter<E> a konečně zápis shluků ClusterWriter<E>.

7.4.1 rozhraní MicroArrayReader<E> a jeho potomci, výjimky GeneException a FileFormatException

Rozhraní MicroArrayReader<E> slouží k načtení bodů ze souboru. Metodou readMicroArray jsou body načteny do nově vytvořené instance třídy MicroArray<E>. Při chybném formátu dat metoda vyvolá výjimku FileFormatException. Rozhraní dědí z rozhraní Stopable.

Z rozhraní MicroArrayReader<E> dědí rozhraní GeneReader, přičemž zde tedy E = Gene, které má navíc metodu readLine. Ta vytvoří novou instanci třídy Gene dle zadaného Stringu. Při chybějících údajích expresí vyvolá metoda výjimku GeneException, která obsahuje (polo)načtený gen v atributu g a pole booleanů, přičemž j. prvek tohoto pole je nastaven na true, je-li j. souřadnice genu g načtena, jinak na false.

Konkrétní implementaci rozhraní GeneReader poskytuje třída PCLReader, který načítá geny z pcl souboru (popis formátu viz část 3.2.1 na straně 9 nebo odstavce (2) až (4) části 6.1 na straně 26). Její atributy jsou file (soubor s daty) a brokenGene, který udává, zda načítání některého genu vyvolalo GeneException. Po vyvolání této výjimky je „neúplný“ gen uložen do speciálního seznamu a je pokračováno v načítání. Po dokončení čtení jsou pomocí metody repairAndAddElement třídy MicroArray<E> do výsledného microArray přidány též tyto „neúplné“ geny. Atribut flowMessageListener slouží pro výpis průběhu načítání, stopBool pro zastavení načítání. Třída tedy takto implementuje rozhraní Stopable. Konstruktor vyžaduje zadání flowMessageListeneru a vstupního souboru.

7.4.2 rozhraní MicroArrayWriter<E> a jeho potomci

Rozhraní MicroArrayWriter<E> slouží k zápisu bodů do souboru. Metodou writeMicroArray jsou body zapsány ze zadané instance třídy MicroArray<E> do souboru. Rozhraní dědí z rozhraní Stopable.

Z rozhraní MicroArrayWriter<E> dědí rozhraní GeneWriter, přičemž zde E = Gene, které má navíc metodu writeLine. Ta zapíše instanci třídy Gene do vraceného Stringu.

Konkrétní implementaci rozhraní GeneWriter poskytuje třída PCLWriter, který zapisuje geny do pcl souboru (popis formátu viz část 3.2.1 na straně 9 nebo odstavce (2) až (4) části 6.1 na straně 26). Dodejme, že v pcl souboru je sloupec s popisem genů ponechán prázdný, neboť není programem nikde uchováván. Rovněž první řádek neobsahuje popisy sloupců s hodnotami genových expresí. Atributy jsou file (soubor pro zápis), flowMessageListener sloužící pro výpis průběhu zápisu a stopBool pro zastavení zápisu. Třída tedy takto implementuje rozhraní Stopable. Konstruktor vyžaduje zadání flowMessageListeneru a výstupního souboru. Mimo to je přítomna metoda writeHeader pro zápis prvních dvou řádků (hlavičky) pcl souboru do Stringu.

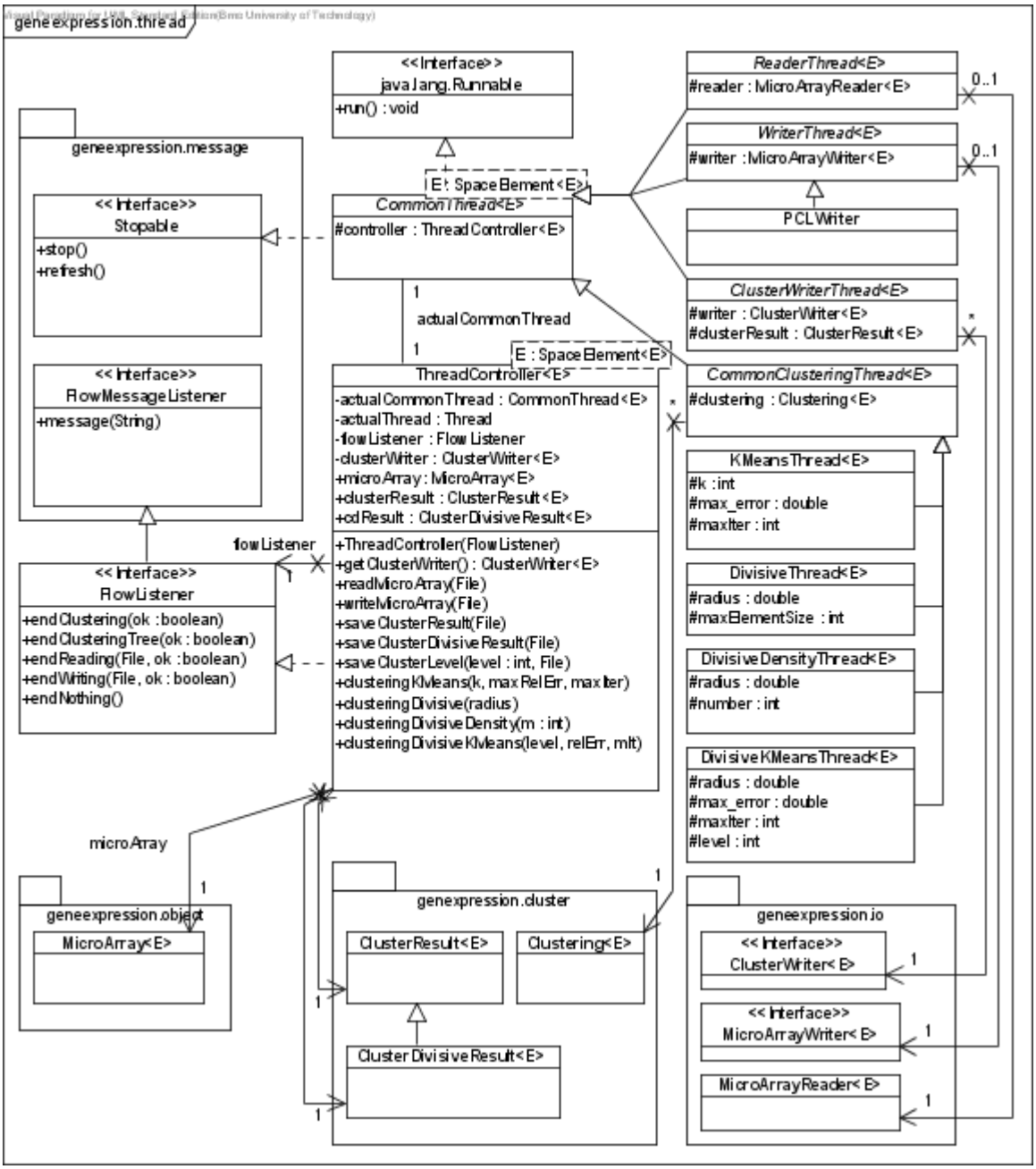
7.4.3 rozhraní ClusterWriter<E> a jeho potomci

Rozhraní ClusterWriter<E> slouží pro zápis shluků do souboru. Metodou writeClusterResult zapíše výsledek shlukovacího algoritmu (instanci potomka třídy ClusterResult<E>) do souboru. Mimo to definuje metodu writeLevel pro zápis jediného patra stromu – výsledku algoritmu Strom++ a metodu writeHeader pro výpis obecných charakteristik množiny bodů (určených parametrem microArray), které byly shlukovány. Více o zapisovaných údají viz část 8.7 na straně 62.

Jedinou implementující třídou je GeneClusterWriter, přičemž zde E = Gene. Obsahuje tytéž atributy jako PCLWriter (viz předchozí sekce). Konstruktor tyto atributy vyžaduje.

7.5 balíček geneexpression.thread

UML diagram balíčku je na obrázku 20.



Ilustrace 20: Balíček thread

V tomto balíčku jsou obsaženy třídy reprezentující vlákna v nichž běží shlukovací algoritmy nebo načítání či zápis dat. Tyto třídy dědí z abstraktní třídy CommonThread<E>. Rovněž je přítomna obslužná třída ThreadController<E>, která vlákna spouští, případně též žádá o jejich zastavení a udržuje jejich výsledky. Pro tuto obslužnou třídu je definováno rozhraní FlowListener, kterým jsou předávány informace o dokončení nějaké akce.

7.5.1 třída CommonThread<E> a její potomci

Abstraktní třída CommonThread<E> reprezentuje obecné vlákno, v němž běží shlukovací

algoritmus nebo načítání ze či zápis do souboru. Tato třída implementuje rozhraní Stopable, metodou stop je tak možné vlákno zastavit. Dále metoda implementuje javovské standardní rozhraní java.lang.Runnable, jejíž metoda run je hlavní (main) metoda vlákna. Třída obsahuje atribut controller s odkazem na ThreadController<E>, který toto vlákno spustil. Instance potomků této třídy informují spouštějící ThreadController<E> o dokončení prováděné akce prostřednictvím metod rozhraní FlowListeneru, který je ThreadControllerem implementován. Rovněž tato vlákna ThreadControlleru „injektují“ výsledky své činnosti. Zprávy výjimek vyvolancýh v rámci vykonávání metody run jsou předávány pomocí metody message ThreadControlleru (ten totiž implementuje rozhraní FlowMessageListener).

Z CommonThread<E> dědí abstraktní třída CommonClusteringThread<E> reprezentující vlákno, v němž běží nějaký shlukovací algoritmus. Vlákno obsahuje další atribut clustering s odkazem na instanci třídy Clustering<E>, která provádí shlukovací algoritmy. Tuto instanci si vytváří třída sama.

Z třídy CommonClusteringThread<E> dědí 4 třídy: KmeansThread<E>, DivisiveThread<E>, DivisiveDensityThread<E> a DivisiveKMeansThread<E> pro algoritmy k-means, Strom++, DBSCAN ze spočítaného výsledku Stromu++ a k-means ze zadaného patra spočítaného výsledku Stromu++.

Z třídy CommonThread<E> ještě dědí 3 další abstraktní třídy: ReaderThread<E>, WriterThread<E> a ClusterWriterThread<E>. ReaderThread<E> slouží k načtení bodů ze souboru, k tomu má atribut MicroArrayReader<E>. WriterThread<E> slouží k zápisu bodů do souboru, k tomu využívá svůj atribut MicroArrayWriter<E>. Konečně ClusterWriterThread<E> slouží k zápisu výsledku shlukování (ClusterResult<E>) uloženém v atributu clusterResult pomocí metod atributu writer třídy ClusterWriter<E>. Z těchto 3 abstraktních tříd pak dědí po řadě třídy PCLReaderThread, PCLWriterThread a GeneClusterWriterThread, které čtou, zapisují geny a zapisují shluky genů.

7.5.2 rozhraní FlowListener

Toto rozhraní dědí z rozhraní FlowMessageListener balíčku message. Přidává metody endClustering, endClusteringTree, endReading, endWriting informující o dokončení shlukování (obecného nebo algoritmu Strom++), čtení a zápisu. Parametr ok udává úspěch operace. Metoda endNothing informuje o nezdařeném spuštění algoritmů, čtení či zápisu. Rozhraní je implementováno třídou ThreadController, čehož využívají vlákna třídy CommonThread<E> ke komunikaci s touto. Rozhraní je implementováno také třídou MainFrame v balíčku gui ke komunikaci s ThreadControllerem.

7.5.3 třída ThreadController<E>

Tato třída spouští a případně zastavuje vlákna provádějící shlukovací algoritmy, zápis nebo čtení. Rovněž udržuje výsledky shlukovacích algoritmů. Implementuje rozhraní FlowListener (viz předchozí sekce) tak, že zprávy předává dalšímu FlowListeneru kterého má v atributu flowListener. Tím je v programu konkrétně instance třídy MainFrame balíčku gui. Přes rozhraní FlowListener též implementuje rozhraní FlowMessageListener, čímž reaguje na zprávy o průběhu prováděných operací zasílané instancemi tříd balíčku cluster a io. Tyto zprávy přeposílá svým flowListenerem dále. Konstruktor zadání tohoto atributu vyžaduje.

Dále má třída atribut actualCommonThread nesoucí aktuálně prováděné vlákno jakožto instanci potomka třídy CommonThread<E>. Atribut actualThread nese aktuálně prováděné vlákno jakožto standardní javovský object třídy java.lang.Thread, v němž běží metoda run instance actualCommonThread.

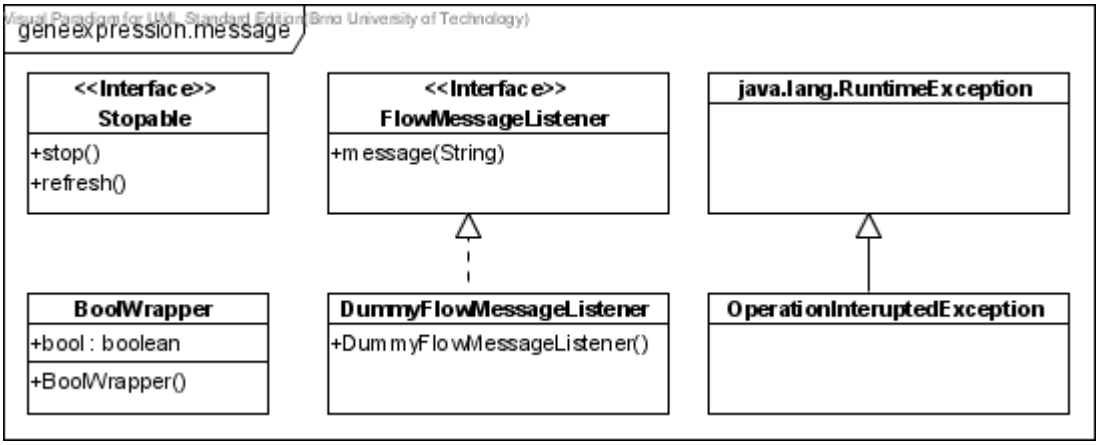
Dále jsou přítomny atributy microArray nesoucí aktuálně načtený mikročip (množinu bodů), clusterResult nesoucí poslední výsledek shlukování (kromě algoritmu Strom++) a cdResult nesoucí

poslední výsledek shlukovacího algoritmu Strom++. Z důvodů častého zapisování shluků do souboru obsahuje třída též atribut clusterWriter, který potom využívají vlákna rozhraní ClusterWriterThread.

Metody s prefixem clustering spouští jednotlivé shlukovací algoritmy. Ostatní metody slouží pro zápis do nebo čtení ze souboru.

7.6 balíček geneexpression.message

UML diagram balíčku lze vidět na obrázku 21.



Ilustrace 21: Balíček message

Tento balíček obsahuje rozhraní a třídy, jejichž instance slouží pro předávání zpráv mezi uživatelem a programem. Jejich funkce jsou různorodé, podívejme se proto přímo na popis jednotlivých tříd.

7.6.1 rozhraní FlowMessageListener

Toto rozhraní definuje jedinou metodu message. Obsahuje-li třída objekt tohoto rozhraní, mohou její metody volat metodu message s parametrem typu String informujícím o průběhu operace. Toho je značně využíváno u metod tříd balíčků cluster a io, které tak informují instanci ThreadControlleru, který implementuje toto rozhraní a který tyto metody (nepřímo) spustil.

Pro testovací účely byla udělána ještě třída DummyFlowMessageListener implementující toto rozhraní, která zprávy vypisuje přímo na standardní výstup. Má pouze bezparametrický konstruktor.

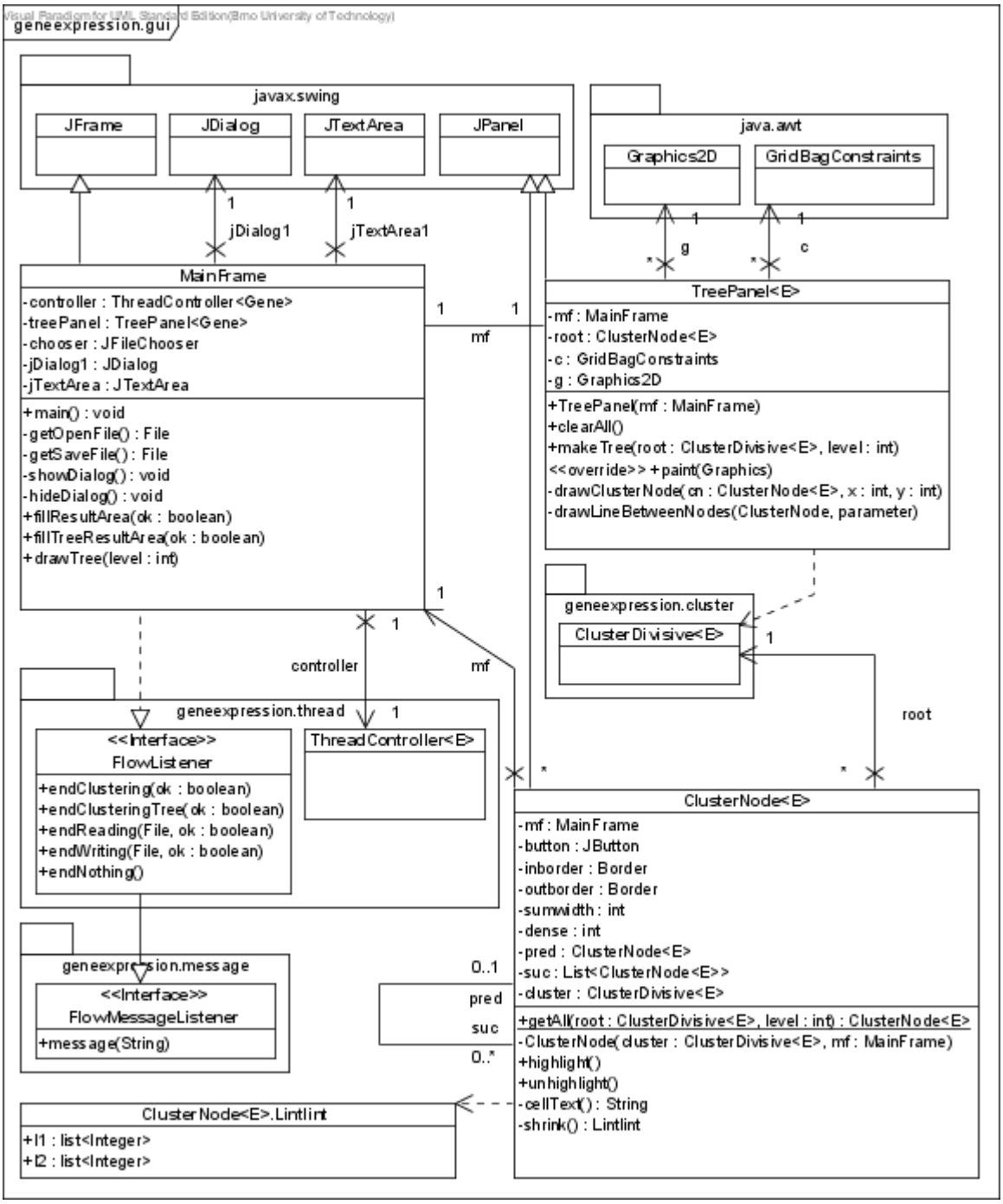
7.6.2 rozhraní Stopable a třída BoolWrapper

Třídy implementující toto rozhraní mají metody, jejichž vykonávání je možno násilně ukončit. K tomuto účelu rozhraní definuje metodu stop, která požádá o ukončení aktuálně prováděné metody. To je prakticky implementováno logickým atributem tříd, který se v metodě stop na staví na true. V metodách je pak tento atributy periodicky testován na true a při jeho splnění se metoda sama ukončí. Metoda refresh potom slouží pro návrat do normálního stavu po zastavení metody. Metody mohou při indikaci požadavku ukončení též „hodit“ výjimku OperationInteruptedException definovanou touto třídou s popisem místa zastavení.

Třída BoolWrapper byla definována jako pomocná pro ony atributy, které jsou periodicky testovány. Jedná se v podstatě o objektovou obálku jediné logické proměnné bool. Konstruktor nastaví hodnotu bool na false.

7.7 balíček geneexpression.gui

UML diagram balíčku lze vidět na obrázku 22.



Ilustrace 22: Balíček gui

Tento balíček definuje třídy pro realizaci grafického uživatelského rozhraní programu. Hlavní okno aplikace je reprezentováno třídou MainFrame. Ta obsahuje panel třídy TreePanel<E> pro zobrazení stromu – výsledku algoritmu Strom++. Ten se skládá z jednotlivých uzlů reprezentovaných třídou ClusterNode<E>. Dodejme, že značná část kódu tříd ClusterNode<E> a TreePanel<E> byla převzata z mé bakalářské práce (viz [8]), která se mimo jiné zabývala zobrazením stromu na obrazovce. Konkrétně se v této bakalářské práci jednalo o třídy bpa.gui.ButtonEquation

a bpa.gui.TreePanel. Celkové procento kódu těchto dvou tříd v poměru k celé implementaci programu je však mizivé.

7.7.1 třída ClusterNode<E>

Tato třída reprezentuje jeden zobrazený uzel stromu, který vznikl shlukovacím algoritmem Strom++. Dědí ze standardní javovské třídy java.swing.JPanel, jedná se tedy o grafický panel. Má atributy mf – okno, ve kterém je uzel zobrazen, button – tlačítko, které je zobrazeno jakožto uzel, inborder a outborder – jeho vnitřní a vnější hranice. Atribut sumwidth pak určuje šířku zobrazeného podstromu uzlu a dense posun doprava či doleva oproti předku. Atribut pred určuje předka uzlu (třídy ClusterNode<E>) a suc seznam potomků (také třídy ClusterNode<E>). Atribut cluster dělá ono napojení na uzel stromu algoritmu Strom++.

Konstruktor třídy vyžadující cluster (uzel Stromu++) a okno (MainFrame), ve kterém je uzel zobrazen je pouze privátní. Celý strom instancí je pak vytvářen statickou metodou getAll, která má za parametr kořen stromu (algoritmu Strom++) a počet pater, která chceme zobrazit (0 pro kořen). Metody highlight a unhighlight slouží k o- a od-značení uzlu myší. Metoda cellText vrací html kód, který určuje, které informace o uzlu a v jakém formátu jsou zobrazeny.

Nechť nejpravější, nejlevější, nejvyšší a nejnižší pixel zobrazeného podstromu tohoto uzlu vytínají obdélník. Metoda shrink pak vrací vzdálenosti každého patra podstromu (v pořadí od kořene podstromu směrem dolů) od levého a pravého okraje obdélníku. Konkrétně je návratová hodnota třídy ClusterNode<E>.Lintlint, obsahuje dva seznamy čísel – jeden pro levý okraj, druhý pro pravý. Pomocí objektů Lintlint metoda také nastaví správně hodnoty sumwidth a dense všem uzlům podstromu. Poznamenejme, že tato metoda je volána rekurzivně pro každý uzel stromu. Kód metody byl převzat převzána z kódu metody shrink třídy bpa.Equation v mé bakalářské práci ([8]) a její podrobnější popis lze nalézt v části 3.3 bakalářské práce ([8]).

7.7.2 třída TreePanel<E>

Tato třída reprezentuje panel, ve kterém je zobrazen strom – výsledek algoritmu Strom++. Dědí proto ze standardní javovské třídy java.swing.JPanel. Má atributy mf – okno, ve kterém je tento panel zobrazen, root – vrcholový uzel zobrazeného stromu, g – objekt pro vykreslování spojnic mezi uzly a c určující pozice uzlů v panelu. Konstruktor vyžaduje zadání instance MainFrame.

Metodou clearAll lze celý panel vyprázdnit. Metoda makeTree naopak od zadaného kořene stromu (výsledku Stromu++) zobrazí zadaný počet pater stromu. Při tom vytvoří strom instancí ClusterNode<E> a zobrazí je pomocí série rekurzivních volání drawClusterNode. Metoda rovněž předefinovává metodu paint svého předka, aby zobrazoval též spojnice uzlů. Ty jsou vykreslovány metodou drawLineBetweenNodes.

7.7.3 třída MainFrame

Tato třída repreztuje hlavní okno na obrazovce, kterým program komunikuje s uživatelem. Dědí proto ze standardní javovské třídy javax.swing.JFrame. Tato třída obsahuje metodu main, kterým se celá aplikace spouští. Konstruktor je bezparametrický. Třída obsahuje velké množství atributů a metod realizující GUI, které nebudeme uvádět. Popis ovládání tohoto hlavního okna je uveden v následující kapitole. Přesto však jmenujme některé význačné metody a atributy.

Atribut controller nese instanci ThreadControlleru, který spouští jednotlivá vlákna. Atribut treePanel potom nese instanci TreePanelu, který zobrazuje strom. Pomocí dialogu chooser je vybírán soubor k načtení nebo uložení a pomocí dialogu jDialog1 je informován uživatel o průběhu požadovaných operací. jDialog1 obsahuje textové pole jTextArea, kam jsou tyto informace vypisovány.

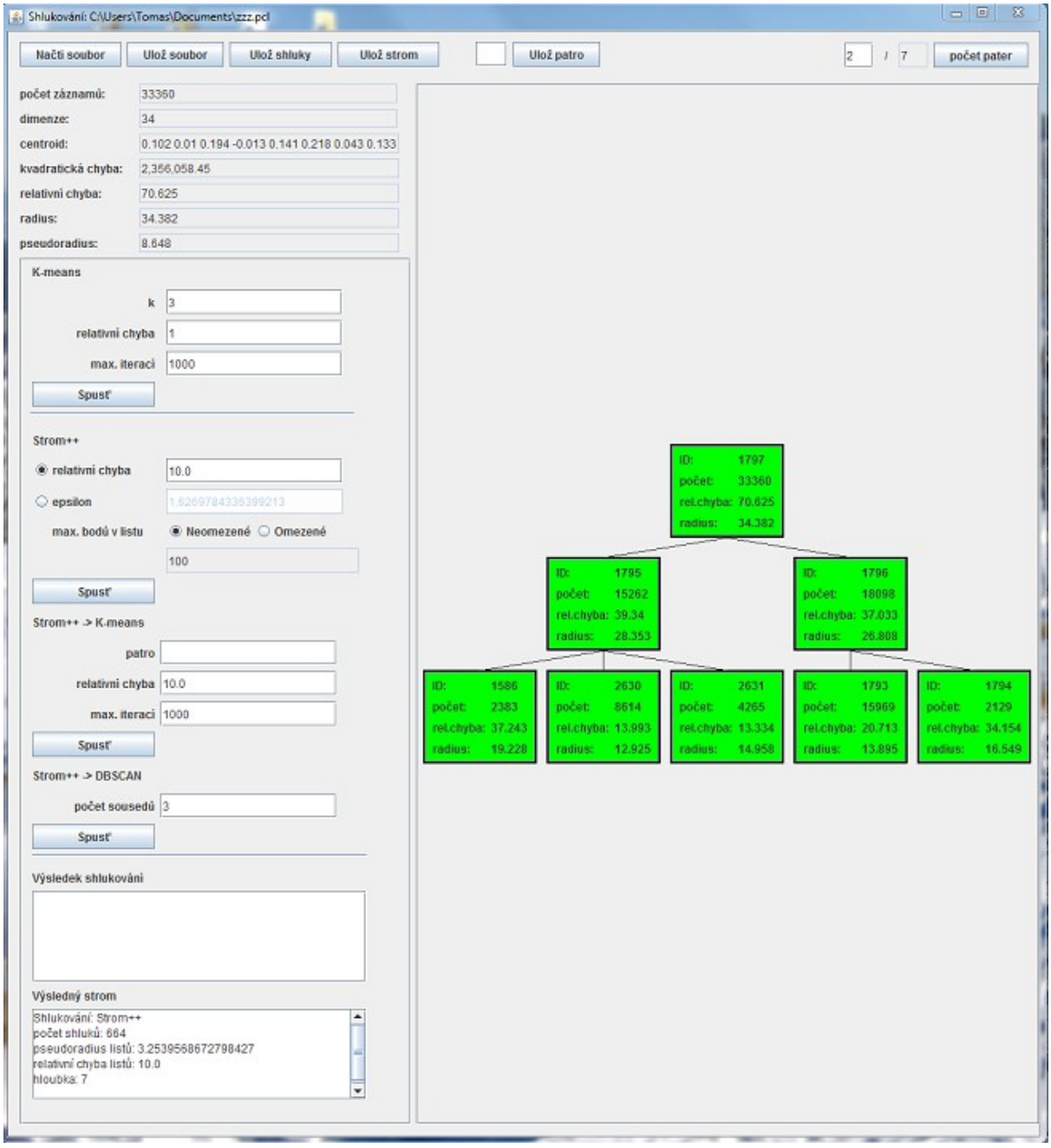
Metody getOpenFile a getSaveFile slouží k výběru souboru k načtení nebo uložení dat. Metoda showDialog, resp. hideDialog potom zobrazí jDialog1, resp. ho skryje. Při zobrazení jsou zablokována tlačítka hlavního okna. Třída dále implementuje rozhraní FlowListener balíčku thread. Reakce na dokončení operací jsou uživateli oznamovány v jDialog1. Rovněž jsou v tělech těchto metod volány následující metody. Metoda fillResultArea v endClustering vyplní textové informace o výsledku shlukování, fillTreeResultArea v endClusteringTree o výsledku shlukování Strom++. Konečně metoda drawTree volaná v endClusteringTree, ale i samostatně, zobrazí daný počet pater stromu v treePanelu.

V následující kapitole se zaměříme na praktické použití programu.

8 Ovládání aplikace

Aplikace je implementováno v jazyce Java SE verze 6. Spouští se pomocí jar balíčku geneexpression.jar. Ke spuštění je třeba mít instalovanou tzv. Java Virtual Machine (JVM) minimálně verze 6 pro používaný operační systém. Jejich seznam společně s možností stažení najdeme na http://java.com/en/download/manual.jsp. JVM je zde označována jako JRE 6. Instalace žádných dalších programů ani knihoven není potřebná. Pro vyzkoušení je k balíčku přiložen pcl soubor, který obsahuje zdrojová data genových expresí z mikročipu.

V následujících sekcích popíšeme jednotlivé možnosti programu. Dodejme, že vlastní shlukovací algoritmy a operace načítání a zápisu dat běží v samostatných vláknech, takže neblokují grafické rozhraní programu. To umožňuje mimo jiné vlákna pomocí tohoto grafického rozhraní zastavit nebo volně přesouvat hlavní okno a dialogy aplikace během běhu algoritmu. Ukázka hlavního okna aplikace je na obrázku 23 (ve zmenšené formě).



Ilustrace 23: Hlavní okno aplikace

8.1 Načtení genových expresí ze souboru

Aplikace umí načítat data z pcl souboru (popis formátu viz část 3.2.1 na straně 9 nebo odstavce (2) až (4) části 6.1 na straně 26). Ten obsahuje na každém řádku hodnoty genových expresí pro jeden gen, přičemž počítáme s tím, že tyto exprese již byly předpočítány dle vzorce 5.3 na straně 23, tj. bereme logaritmy podílu normovaných expresí.

Načtení se provádí kliknutím na tlačítko „Načti soubor“ v horní liště hlavního okna vlevo a výběrem souboru ve vyskočeném dialogu. Průběh operace je zobrazen v následně zobrazeném dialogu. Pokud má soubor špatný formát, je vypsán řádek, kde načítání souboru selhalo a načítání se zastaví. Jsou-li v některém řádku chybějící hodnoty genových expresí, jsou čísla těchto řádků vypsány. Následně jsou chybějící hodnoty doplňovány dle nejbližších genů jak je popsáno v odstavci (3) části 6.1 na straně 26. O počtu již opravených řádků je uživatel opět informován.

Operaci načítání lze přerušit kliknutím na tlačítko Zrušit. Načtené hodnoty expresí budou z programu vymazány. Po informování o načtení souboru se tlačítkem OK vrátíme zpět do hlavního okna, které má všechna tlačítka během načítání zablokována.

Po úspěšném načtení se název souboru s cestou objeví v titulním pruhu hlavního okna. Dále se v panelu vlevo nahoře objeví základní charakteristiky souboru dat (bodů danými genovými expresemi načtených genů), jako je počet záznamů, dimenze bodů, centroid, kvadratická chyba (suma čtverců vzdáleností všech bodů od centroidu), relativní chyba (kvadratická chyba dělená počtem záznamů), radius (maximální vzdálenost bodu od centroidu) a pseudoradius, vypočítaný dle vzorce 6.22 na straně 33, kde E je kvadratická chyba, a počet záznamů a R pseudoradius. Doplňme, že ε figurující v následujícím vzorci 6.23 je dvojnásobkem pseudoradiu. Charakteristiky jsou (pro účely zobrazení) ořezány na 3 desetinná místa.

8.2 Uložení genových expresí do souboru

Aplikace umožňuje načtená data s genovými expresemi uložit do souboru v pcl formátu (popis formátu viz část 3.2.1 na straně 9 nebo odstavce (2) až (4) části 6.1 na straně 26). Sloupec s popisem genu (description) je ponechán pro každý řádek souboru prázdný. Rovněž hlavičky sloupců s genovými expresemi v prvním řádku jsou ponechány prázdné. Každý řádek obsahuje všechny hodnoty genových expresí, neboť chybějící hodnoty byly doplněny již při načítání.

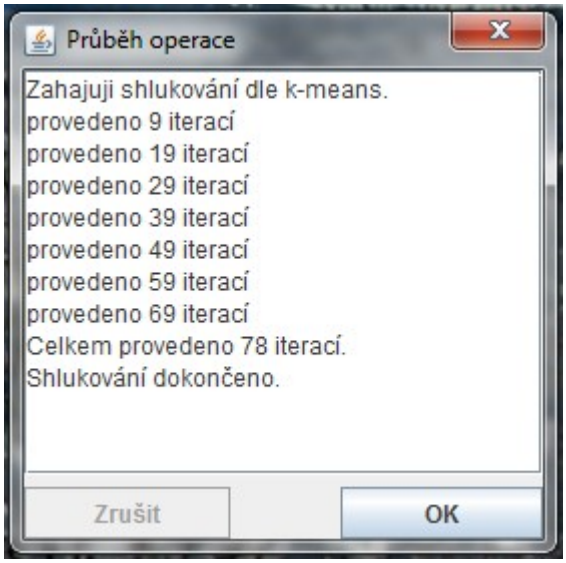
Zápis se provádí kliknutím na tlačítko „Ulož soubor“ v horní liště hlavního okna vlevo a výběrem souboru ve vyskočeném dialogu. Při přepisu jiného souboru je uživatel požádán o potvrzení. Průběh operace je zobrazen v následně zobrazeném dialogu. Jsou zde zobrazena i případná selhání zápisu.

Operaci zápisu lze přerušit kliknutím na tlačítko Zrušit. V souboru budou ponechány již zapsané (celé) řádky a bude uzavřen. Po informování o zápisu celého souboru se tlačítkem OK vrátíme zpět do hlavního okna, které má všechna tlačítka během zápisu zablokována.

8.3 Spuštění k-means

Algoritmus k-means může být spuštěn teprve po načtení dat ze souboru. Je vyžadováno zadání parametrů k – počtu shluků, požadované maximální relativní chyby a maximálního počtu iterací v kolonce vlevo s hlavičkou K-means. Spuštění se provádí pomocí tlačítka Spusť v této kolonce. Následně je zkontrolována korektnost vstupních parametrů a spuštěn algoritmus. Ve vyskočeném dialogu je uživatel informován o aktuálním počtu provedených iterací. Ukázko takového dialogu je na obrázku 24. Algoritmus lze přerušit tlačítkem Zrušit. Výsledky shlukování jsou poté anulovány.

Po doběhnutí algoritmu a stisknutí OK je vlevo dole v kolonce Výsledek shlukování zobrazen popis výsledku, tedy popis metody, vstupních parametrů, dosažené relativní chyby, počtu proběhnutých iterací., velikost minimálního a maximálního shluku. Dosažená relativní chyba může být horší (větší) než požadovaná, pokud je dosaženo maximálního počtu iterací nebo se relativní chyba s dalšími iteracemi již (téměř) nemění. Spočtené shluky je možné uložit tlačítkem „Ulož shluky“ (viz níže). Dodejme, že kolonka Výsledný strom se spuštěním tohoto algoritmu nepřepisuje a výsledek algoritmu Strom++ (pokud nějaký je) je stále možno uložit tlačítkem „Ulož strom“.



Ilustrace 24: Dialog průběhu operace

8.4 Spuštění algoritmu Strom++

Algoritmus Strom++ popsaný v sekci 6.2.2 od strany 28 může být spuštěn teprve po načtení dat ze souboru. Je vyžadováno zadání parametrů v kolonce Strom++, a sice (maximální) relativní chyby (listů) nebo hodnoty epsilon, pro jejíž převod na relativní chybu platí vzorec 6.23 na straně 33, Při zadání jedné hodnoty (zadání obou ani program nedovolí) je automaticky druhá dopočítána a zobrazena. Dále je možno zadat maximální počet bodů v listech nebo povolit tento neomezený. Spuštění se provádí tlačítkem Spusť v příslušné kolonce. Následně je zkontrolována korektnost parametrů a spuštěn algoritmus. Ve vyskočeném dialogu je uživatel informován o počtu aktuálně přidaných bodů do stromu. Algoritmus lze přerušit tlačíkem Zrušit. Výsledky shlukování jsou poté anulovány.

Po doběhnutí algoritmu a stisknutí OK je vlevo dole v kolonce Výsledný strom zobrazen popis výsledku, tedy popis metody, vstupních parametrů, počet uzlů stromu, jeho hloubka a velikost maximálního a minimálního listového shluku. Spočtené shluky je možné uložit tlačítkem „Ulož strom“ (viz níže). Rovněž je možno uložit pouze zadané patro stromu pomocí tlačítka „Ulož patro“. Dodejme, že kolonka Výsledek shlukování se spuštěním tohoto algoritmu nepřepisuje a výsledek jiného než tohoto algoritmu je stále možno uložit tlačíkem „Ulož shluky“.

Kromě uvedeného výpisu je po doběhnutí algoritmu zobrazen výsledný strom v pravém panelu hlavního okna. Pro urychlení zobrazení jsou zobrazeny pouze 3 patra stromu. Počet zobrazených pater stromu lze měnit zadáním počtu a stiknutím tlačítka „počet pater“ v horní liště v pravém rohu (0 pro patro s kořenem). Zde je také zobrazen celkový počet pater stromu (za lomítkem). Ve stromu je u každého uzlu uvedeno ID (jméno shluku), počet bodů v podstromu, relativní chyba a radius.

8.5 Spuštění algoritmu k-means na patrech stromu

Tento algoritmus může být spuštěn teprve po proběhnutí algoritmu Strom++. Spustí k-means, kde za inicializační centroidy vezme centroidy shluků daného patra stromu. Parametry algoritmu jsou zadávány do kolonky Strom++ ->K-means, a sice patro stromu (0 pro kořen, zvětšuje se směrem dolů), maximální počet iterací a maximální požadovnou relativní chybu. Ta je po doběhnutí algoritmu Strom++ nastavena stejná, jakou měl tento algoritmus. Spuštění se provádí opět tlačítkem Spusť v příslušné kolonce. Následně je zkontrolována korektnost parametrů a spuštěn algoritmus. Ve vyskočeném dialogu je uživatel informován o aktuálním počtu provedených iterací. Algoritmus lze přerušit tlačítkem Zrušit. Výsledky shlukování jsou poté anulovány.

Po doběhnutí algoritmu a stisknutí OK je vlevo dole v kolonce Výsledek shlukování zobrazen popis výsledku, tedy popis metody, vstupních parametrů (včetně pořadí patra a počtu shluků), dosažené relativní chyby, počtu proběhnutých iterací a maximální a minimální velikost shluku. Spočtené shluky je možné uložit tlačítkem „Ulož shluky“.

8.6 Spuštění algoritmu DBSCAN

Algoritmus DBSCAN může být spuštěn teprve po proběhnutí algoritmu Strom++. Algoritmus pracuje, jak je popsáno v sekci 6.2.3 od strany 32. Parametry algoritmu jsou zadávány do kolonky Strom ->DBSCAN. Požadovaný poloměr ε-okolí byl již zadán v algoritmu Strom++ parametrem epsilon (popř. přepočítán z maximální relativní chyby listů), stačí tedy zadat (minimální) počet sousedů v ε-okolí. Spuštění se provádí opět tlačítkem Spusť v příslušné kolonce. Následně je zkontrolována korektnost parametrů a spuštěn algoritmus. Ve vyskočeném dialogu jsou vypisovány jména listů, ve kterých již byly nalezeny jádra a ke kterým již byly nalezeny všechny sousední listy. Algoritmus je časově náročný, pro mikročipy o desítkách tisíc bodů může běžet v řádu minut. Výpis průběhu algoritmus bohužel ještě více zpomaluje, řádově o desítky procent. Algoritmus lze přerušit tlačítkem Zrušit. Výsledky shlukování jsou poté anulovány.

Po doběhnutí algoritmu a stisknutí OK je vlevo dole v kolonce Výsledek shlukování zobrazen popis výsledku, tedy popis metody, vstupních parametrů (včetně epsilon), celkový počet shluků, jader a maximální, minimální i průměrná velikost shluku. Spočtené shluky je možné uložit tlačítkem „Ulož shluky“.

8.7 Uložení shluků do souboru

Spočtené shluky je možné uložit do souboru pomocí tlačítek „Ulož strom“, „Ulož shluky“ nebo „Ulož patro“ vlevo na horní liště.

Tlačítko „Ulož strom“ ukládá spočítaný strom (výsledek algoritmu Strom++). Do souboru jsou nejprve vypsány základní statistiky souboru dat zobrazené po načtení souboru vlevo nahoře. Poté je vypsána hlavička shodná s popisem výsledku v kolonce Výsledný strom. Poté následuje seznam jednotlivých uzlů oddělený prázdnými řádky. U každého shluku je vypsáno jméno, typ uzlu (vnitřní/list), počet potomků (pro vnitřní uzel), velikost (počet bodů v podstromu), centroid, relativní chyba, radius, jméno předka a seznam potomků oddělený mezerami (pro vnitřní uzel). Za tím následuje seznam jmen bodů příslušných tomuto shluku (uzlu) oddělený mezerami.

Tlačítko „Ulož patro“ ukládá tytéž informace jako tlačítko „Ulož strom“, pouze jsou ze spočítaného stromu vypsány jen shluky zadaného patra (0 pro patro s kořenem, 1 pro 1. patro pod kořenem atd.). Rovněž je vypsána informace, o které patro se jedná.

Tlačítko „Ulož shluky“ naproti tomu ukládá výsledky ostatních shlukovacích algoritmů. Do souboru jsou nejprve vypsány základní statistiky souboru dat zobrazené po načtení souboru vlevo nahoře. Poté je vypsána hlavička shodná s popisem výsledku v kolonce Výsledek shlukování. Poté následuje seznam jednotlivých shluků oddělný prázdnými řádku. U každého shluku je vypsáno jméno, velikost (počet bodů ve shluku), pro k-means navíc centroid a relativní chyba, pro DBSCAN počet jader. Za tím následuje mezerami oddělený seznam jmen bodů příslušných tomuto shluku.

9 Výsledná analýza dat a algoritmů

V této kapitole se pokusíme určit, na čem zejména závisí časová i paměťová náročnost jednotlivých shlukovacích algoritmů. Poté se zaměříme na porovnání jednotlivých shlukovacích algoritmů, co se týče kvality výsledných shluků. Připomeňme však, že toto kvalitativní srovnání bude mít spíše jen informatický charakter, neposoudíme kvalitu výsledných shluků z biologického hlediska genů s podobnou genovou expresí. To by bylo nutné přenechat nějakému odborníku z oblasti genového inženýrství.

9.1 Časová a paměťová náročnost algoritmů

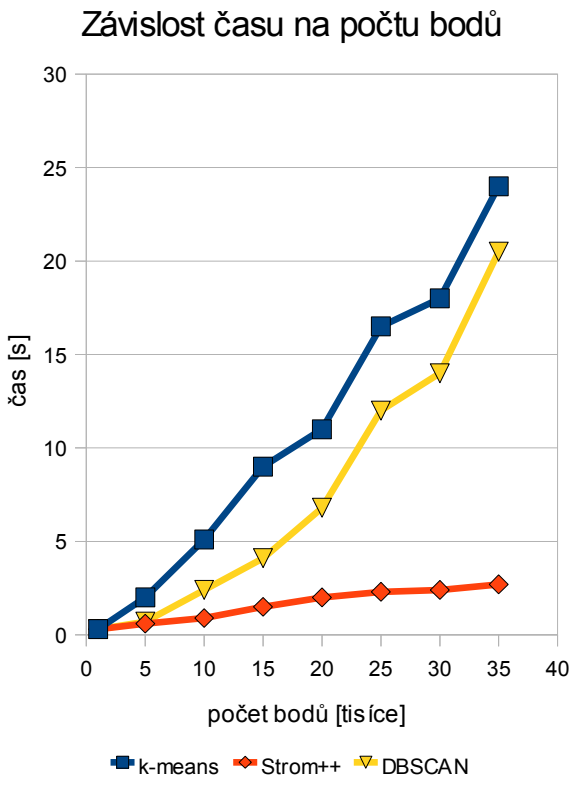
Časovou a paměťovou náročnost budeme zkoumat na testovacím vzorku dat, nepůjde tedy o data z nějaké databáze genové exprese. Rozlišitelnost časových nároků „nastavíme“ na 1 sekundu, paměťových nároků na 100 MB vzhledem k velikosti pamětí dnešních stolních počítačů. Tím míníme, že parametry, které ovlivní časovou náročnost o méně než 1 s nebo paměťovou o méně než 100 MB, budeme považovat za neovlivňující náročnost algoritmu. Algoritmy budeme testovat na stolním počítači s 2-jádrovým procesorem AMD frekvence 2.5 Ghz a operační pamětí 2 GB.

Postupně vyzkoušíme závislost nároků na dimenzi bodů, jejich počtu a parametrech jednotlivých algoritmů. Testovací data vygenerujeme pomocí třídy GeneGenerator (viz 7.2.2 na str. 40). Spokojíme se s 20 000 body o dimenzi 25 rozloženými rovnoměrně uvnitř (hyper)koule o poloměru 25 (interních jednotek programu), nebude-li řečeno jinak. Tomuto zadání budeme říkat ethalon.

9.1.1 dimenze bodů

Pro otestování závislosti nároků na dimenzi jsme vyzkoušeli shlukovat oněch 20 000 bodů dimenze 3, 5, 10 a dále po 5 až do 50. Hustotu rozložení bodů jsme požadovali zachovat konstatní vzhledem k ethalonu, proto byl příslušně měněn poloměr (hyper)koule. k pro k-means jsme zvolili 100, relativní chyby pro k-means i Strom++ 1.0, min. počet sousedů pro jádra u DBSCANu 3. Max. počet iterací k-means jsme nastavili na 1000 (tolik iterací k-means stejně v našem testování nikdy nedosáhl).

Doba běhu všech algoritmů se zdá být na dimenzi nezávislá, pro Strom++ zhruba 2 sekundy, pro k-means zhruba 15 sekund a pro DBSCAN zhruba 9 sekund. Rovněž paměťová náročnost nikdy nedosáhla 100 MB. Spokojíme se tak se závěrem, že náročnost algoritmů není výrazně ovlivňována dimenzí bodů.



Ilustrace 25: Závislost času na počtu bodů

9.1.2 počet bodů

Zde jsme vyzkoušeli shlukovat postupně 1000, 5000, 10000 a dále po 5000 až do 40000 bodů. Poloměr (hyper)koule byl opět měněn tak, aby hustota bodů odpovídala ethalonu. Opět jsme zvolili k pro k-means 100, relativní chyby pro k-means i Strom++ 1.0, min. počet sousedů

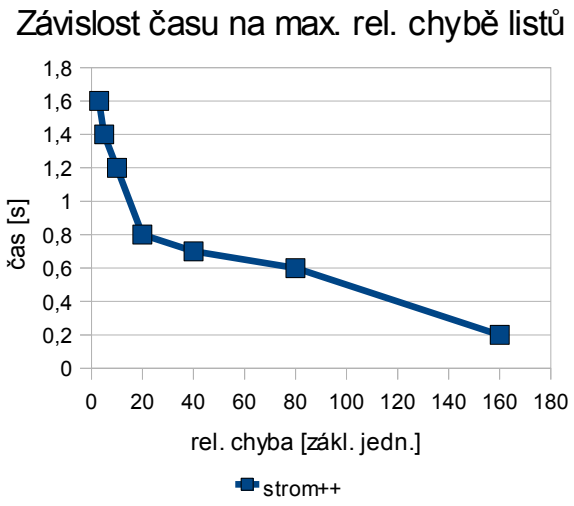
pro jádra u DBSCANu 3. Max. počet iterací k-means jsme nastavili rovněž na 1000.

Patrně v souladu s předpokladem se ukázalo, že nároky algoritmu závisí na počtu bodů výrazně. Paměťové nároky však opět nepřesáhli 100 MB a nebudeme proto o nich dále pojednávat. Graf závislosti časových nároků (v sekundách) na počtu bodů (v tisících) vidíme na obrázku 25. Z něj je patrné, že tendence je pro všechny algoritmy rostoucí, avšak Strom++ roste s výrazně menší strmostí. To dává dobré předpoklady pro jeho další využití.

9.1.3 parametry k-means

Zkusili jsme spouštět k-means na našem ethalonu postupně pro různé k a relativní chyby. k jsme nastavovali postupně od 5 po 5 do 100 a poté po 100 do 300. Relativní chyba byla zadávána v rozmezí 20 až 60, vždy přibližně o 1.0 větší, než jaké byl k-means schopen pro dané k dosáhnout. Tím jsme omezili počet iterací na konci, které již velmi málo ovlivňují celkovou relativní chybu.

Paměťové nároky opět nepřekročili hranici 100 MB. Pro různé hodnoty k a relativní chybu nastavenou uvedeným způsobem jsme vždy dostávali čas zhruba 4 sekundy. Velikost k tak neměla na časovou složitost výrazný vliv. Z důvodu malých časů (pro různá k) i při nízce nastavené hodnotě relativní chyby (zhruba 1.0 nad minimem, kterého lze dosáhnout) jsme odpustili od testování závislosti času na relativní chybě, neboť by nám jeho výsledky nijak nepomohly.



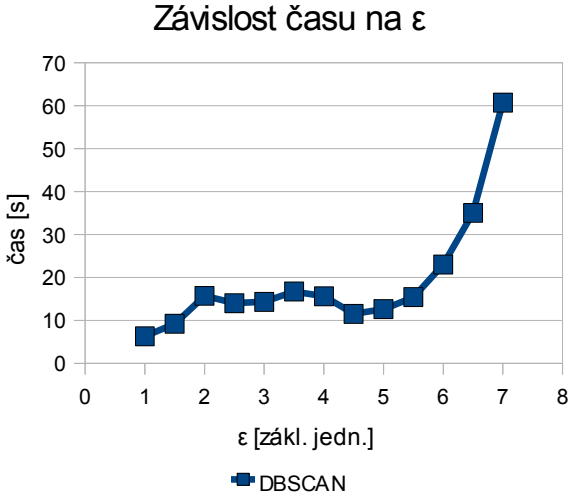
Ilustrace 26: Závislost času na rel. chybě

9.1.4 parametry Stromu++

Algoritmus Strom++ jsme zkoušeli testovat na našem ethalonu pro různé hodnoty maximální relativní chyby listů, konkrétně 3 (kdy počet listů dosáhl 10 000, menší chybu jsme tedy nezkoušeli), 5, 10, 20, 40, 80 a 160 (kdy strom obsahoval kromě listů jen kořen). V souladu s předpokladem potřebný čas klesal. Graf této závislosti je na obrázku 26. Paměťová náročnost algoritmu byla minimální (nikdy nepřesáhla 100 MB).

9.1.5 parametry DBSCANu

Algoritmus DBSCAN jsme zkusili spustit na výsledném stromu algoritmu Strom++ (jak je popsáno v části 6.2.3 od str. 32). Hodnotu ε jsme volili postupně 1.0 (kdy počet listů stromu dostáhl přibližně 10 000) a dále po 0.5 až do 7 (kdy byl celý strom redukován v jediný list – kořen), m jsme ponechali rovné 3. Z minima, kdy je stromová struktura pro hledání sousedních listů výhodná, však čas s přibývajícím ε rychle vzroste, poté zůstává téměř konstantní. V okamžiku podstatné redukce stromu na malý počet listů však čas opět „raketově“ stoupá až k hodnotě kolem jedné minuty, kde se zastaví. Graf této



Ilustrace 27: Závislost času na ε

závislosti je na obrázku 27.

Změna hodnoty m (požadovaného minimálního počtu sousedů v ε-okolí jádra) neměla na časovou náročnost vliv. Zkoušeli jsme různá m v rozsahu od 1 do 200.

Paměťová náročnost algoritmu mírně přesáhla oněch 100 MB, avšak nezávisle na parametrech. Nyní přejděme ke shodnocení kvality výsledných shluků jednotlivých algoritmů.

9.2 Kvalita výsledných shluků

V této části se pokusíme porovnat jednotlivé algoritmy, co se týče počtu shluků, jejich minimální a maximální velikosti a popíšeme některé výhody a nevýhody algoritmů. Připomínáme, že zhodnocení kvality shluků z biologického hlediska neprovádíme. Testování proběhlo na 5 různých souborech formátu pcl, stažených metodou popsané v části 3.2.1 na straně 9 z databáze SMD. Počet záznamů (genů) v souborech se pohyboval mezi 20 000 a 40 000, jejich dimenze v rozmezí 10 až 40.

9.2.1 k-means

Tento algoritmus má nevýhodu v nutnosti zadání počtu budoucích shluků. Navíc nevytváří žádnou hierarchii shluků, která bývá požadována. Ani rychlost v porovnání s algoritmem Strom++ není lepší. Naopak zadání relativní chyby a max. počtu iterací nepředstavuje výrazné omezení, stačí zadat nízkou hodnotu chyby (o řád nižší než relativní chyba celého souboru) a algoritmus sám dosáhne nejmenší možné chyby (pro dané k). Počet iterací stačí naopak zvolit dostatečně vysoký (např. 1000).

Poměr počtu prvků v největším a nejmenším shluku roste přibližně s hodnotou k. Pro nízká k (do 5) se tak shluky ve velikosti příliš neliší, což nemusí odpovídat skutečnosti. I pro vyšší k (od 50) však nejmenší shluky obsahují desítky prvků, což znemožňuje určit odlehlé prvky. Naopak velké shluky velikostí řádu tisíců stále zůstávají nerozdělené. k-means tak pro analýzu genové exprese z nalezených shluků není příliš dobrý.

9.2.2 Strom++

Hlavní výhodou algoritmu je, že nemusíme zadávat výsledný počet shluků. Body jsou do shluků přiřazovány ve snaze minimalizovat relativní chybu shluku. Další výhodou je, že shluky vytváří stromovou hierarchii a výsledky jednotlivých pater stromu lze ještě dále zpřesnit pomocí k-means (bez nutnosti zadání k). Algoritmus je proti k-means i DBSCANu řádově rychlejší při zachování nízkých nároků na paměť, jak bylo ukázáno v předchozí části kapitoly (viz graf 25 na str. 63). Blízkost genů je tak odstupňována dle hierarchie stromu. Naopak nevýhodou je, že vytvářené shluky jsou kulového tvaru.

Při zadání maximální relativní chyby listů o řád nižší než je relativní chyba souboru algoritmus vytvoří velký počet shluků (tisíce). To je sice výhodné pro následující algoritmus DBSCAN, nikoliv však pro následný k-means. Pro ten je naopak vhodné volit hodnoty relativní chyby stejného řádu jako je relativní chyba celého souboru. V této oblasti počet shluků (uzlů ve stromu) rychle narůstá (s klesající relativní chybou). Mezi listovými shluky opět zůstávají dále nerozdělené shluky s velkým počtem prvků (tisíce), tuto neduhu však lze odstranit nastavením maximálního počtu prvků v listu. Shluky o minimálním počtu prvků (až o jediném) jsou přítomny. Otázkou ovšem zůstává, jestli nejsou tyto malé shluky spíš skrytým nedostatkem algoritmu a nemají žádný biologický význam (nejsou tvořeny odlehlými prvky).

9.2.3 DBSCAN

Hlavní motivací implementace tohoto algoritmu byla, že umí hledat též shluky, které nemají

kulový tvar. Toto je však vykoupeno velkou časovou náročností algoritmu pro velké hodnoty poloměru ε-okolí jádra. Navíc není jasná interpretace hodnoty m – počtu sousedů jádra (bodů v jeho ε-okolí) ve smyslu genové exprese, ukazuje se však, že na počet výsledných shluků nemá hodnota m velký vliv (pro m od 1 do 20). Počet shluků bude s přibývajícím m větší řádově o procenta (pro m=20 oproti m=1), shluky o velkém počtu prvků zůstanou stále nerozdělené.

Protože vstup DBSCANu je „napojen“ na výstup Stromu++, budeme hodnotu ε uvažovat přepočítanou z relativní chyby dle vzorce (6.23) na straně 33. Pro relativní chybu o řád nižší, než je relativní chyba celého souboru, dostaneme vysoký počet shluků (10 000 a více), z nichž většina obsahuje jediný prvek. Mezi nimi však najdeme též shluky o desítkách prvků, z toho lze usuzovat, že geny odpovídající těmto prvkům, jsou si velmi podobné. Díky této nízké relativní chybě (tedy také nízkému ε), je ve stromu algoritmu Strom++ značný počet listů o malém počtu prvků a sousedé jader se nacházejí v témže nebo několika sousedních listech, což urychluje hledání jader a jejich sousedů a tím i shluků založených na hustotě (algoritmu DBSCAN) .

Naopak pro relativní chybu řádově stejnou, jako je relativní chyba celého souboru, dostáváme jeden obrovský shluk s 10 000 a více prvky a k tomu spoustu malých shluků (stovky až tisíce), které obsahují většinou jen 1 prvek. Tolik malých shluků stěží identifikuje odlehlé prvky a celkově tak pro shlukovou analýzu nemají tyto výsledky velký význam.

Popsali jsme tak výhody a nevýhody implementovaných algoritmů a závislosti času (jejich běhu) na jejich parametrech. Nyní přistoupíme k závěru této práce, kde zhodnotíme dosažené výsledky a ukážeme možný směr dalšího rozvoje práce či programu.

10 Závěr

Na začátku této práce jsme vysvětlili pojmy pojmy genové exprese a DNA mikročipů. Uvedli jsme jaké těžkosti přinášejí naměřená data a jak se s tím můžeme vyrovnat jejich transformací. Představili jsme některé shlukovací algoritmy a uvedli jsme jak je aplikovat na data z mikročipů a k čemu jejich shlukování může sloužit. Po této teoretické přípravě nastudované z různých knih jsme se přesunuli k vlastní tvůrčí činnosti.

Kromě známých algoritmů k-means a DBSCAN se podařilo vytvořit a implementovat nový hierarchický algoritmus Strom++ s nízkými nároky na čas a paměť počítače, jehož výsledků lze navíc použít v předchozích dvou uvedených algoritmech. Rovněž bylo pojednáno o časové složitosti algoritmů a byly detailněji popsány metody a úskalí implementace. V předešlé kapitole jsme tyto algoritmy porovnali a shrnuli jejich hlavního výhody a nevýhody. Pro běžného uživatele bylo navíc vytvořeno grafické rozhraní, kterým může snadno algoritmy spouštět a následně pak analyzovat jejich výsledky z výpisů do souborů.

Výsledkem implementovaných algoritmů jsou tedy shluky genů, které jsou si podobné hodnotami genové exprese za měnících se podmínek s přibývajícím časem. Nevýhodou je, že tyto výsledky již nebyly konzultovány s žádným odborníkem z oblasti biologie, např. genovým inženýrem. Teprve další experimenty nebo návazná analýza výsledků dle databáze již provedených experimentů by mohly interpretovat nalezené shluky, tedy přiřadit oné podobnosti genů v rovině hodnot genových expresí nějaký konkrétní význam.

Další vývoj programu by bylo možné ubírat rozšířením možností pro větvení stromu a změnou podmínek pro dělení listů. Také k-means použitým i v samotném algoritmu Strom++ by mohlo být vhodné nahradit algoritmem SOM. Konečně by mohlo být vhodné porovnat výsledky algoritmů Strom++ a SOTA a dle tohoto porovnání algoritmus Strom++ ještě více vylepšit.

11 Literatura

[1] Zvelebil M., Baum J. O.: Understanding Bioinformatics, ISBN: 0-8153-4024-9, 2008

[2] Guojun Gan, Chaoqun Ma, Jianhong Wu: Data Clustering, Theory, Algorithms and Applications, ISBN: 978-0-898716-23-8, 2007

[3] Han J., Kamber M.: Data Mining, Concepts and Techniques, ISBN: 1-55860-489-8, 2001

[4] Alvis Brazma, Helen Parkinson, Thomas Schlitt, Mohammadreza Shojatalab: A quick introduction to elements of biology – cells, molecules, genes, functional genomics, microarrays, dostupné on-line na <http://www.ebi.ac.uk/microarray/biology_intro.html> (květen 2010)

[5] ArrayExpress, internetová databáze, <http://www.ebi.ac.uk/microarray-as/ae/> (květen 2010)

[6] Stanford Microarray Database, internetová databáze, <http://smd.stanford.edu/> (květen 2010)

[7] Wikipedia, internetová encyklopedie, <http://wikipedia.org> (květen 2010)

[8] Hebelka Tomáš: Webový generátor tabel pro normované BPA, bakalářská práce, Masarykova univerzita v Brně, 2008, dostupné on-line na <http://is.muni.cz/th/139863/fi_b/> (květen 2010)

[9] Cohen Jacques: Bionformatics – An introduction for computer scientists, Brandeis University, 2003

[10] Hidden 'junk' gene separater human brains from chimpanzees, internetový článek, <http://notexactlyrocketscience.wordpress.com/2006/09/16/hidden-%E2%80%98junk%E2%80%99-gene-separates-human-brains-from-chimpanzees/> (květen 2010)

[11] Profiling technique: Microarray analysis, internetový článek, <http://learn.genetics.utah.edu/content/labs/microarray/analysis/> (květen 2010)

[12] WormBook: Germline genomics, internetový článek <http://www.wormbook.org/chapters/www_germlinegenomics/germlinegenomics.html> (květen 2010)

[13] Logical Implication of Tumor Cell Evolution, internetový článek, <http://www.curecancerproject.org/beta/page.php?=science&p=logical_implications_evolution> (květen 2010)

[14] Mathworks: Working with Clustergram Function, internetový manuál, <http://www.mathworks.com/products/demos/shipping/bioinfo/clustergramdemo.html?product=BI> (květen 2010)

[15] R Graphics: SOTA, internetový manuál, <http://bm2.genes.nig.ac.jp/RGM2/R_current/library/clValid/man/sota.html> (květen 2010)

12 Přílohy

K práci je přiloženo CD s následujícím obsahem. V adresáři text je obsažena tato práce, v adresáři program pak spustitelný program ve formě jar balíčku, ke kterému jsou v adresáři pcl přiloženy ukázky pcl souborů z databáze SMD. V adresáři uml je zdrojový kód UML diagramů, které lze otevřít v prostředí Visual Paradigm for UML verze 7.1 nebo vyšší. Konečně adresář kod obsahuje zdrojové kódy implementace s dokumentací. Adresář kod lze otevřít přímo v prostředí Netbeans verze 6.8 nebo vyšší.