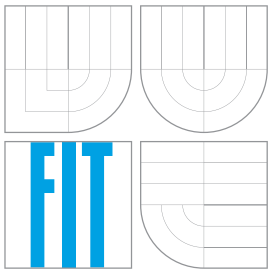
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



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

AKCELERACE ALGORITMŮ PRO SHLUKOVÁNÍ TUNELŮ V PROTEINECH

ACCELERATION OF ALGORITHMS FOR CLUSTERING OF TUNNELS IN PROTEINS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. MARTA ČUDOVÁ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. TOMÁŠ MARTÍNEK, Ph.D.

BRNO 2016

Zadání diplomové práce/18181/2015/xcudov00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačových systémů Akademický rok 2015/2016

Zadání diplomové práce

Řešitel: **Čudová Marta, Bc.**
Obor: Počítačové a vestavěné systémy
Téma: **Akcelerace algoritmů pro shlukování tunelů v proteinech**
Acceleration of Algorithms for Clustering of Tunnels in Proteins
Kategorie: Bioinformatika

Pokyny:

1. Seznamte se s existujícími nástroji pro výpočet tunelů v proteinech a s algoritmy, které využívají.
2. Seznamte se s existujícími algoritmy a přístupy pro shlukování rozsáhlých datových sad.
3. Identifikujte kritická místa algoritmů pro výpočet tunelů v proteinech.
4. Navrhněte vhodný způsob akcelerace kritických míst algoritmů pro výpočet tunelů a proveďte jeho implementaci.
5. Zhodnoťte dosažené výsledky a diskutujte možnosti dalšího pokračování projektu.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 až 3 zadání.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Martínek Tomáš, Ing., Ph.D.**, UPSY FIT VUT
Datum zadání: 1. listopadu 2015
Datum odevzdání: 25. května 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačových systémů a sítí
612 66 Brno, Božetěchova 2

L.S.

Kotásek

doc. Ing. Zdeněk Kotásek, CSc.
vedoucí ústavu

Abstrakt

Práce se zabývá problémem shlukování tunelů z dat získaných molekulární dynamikou proteinů. Tento proces je velmi výpočetně náročný a představuje výzvu pro vědecké komunity. Cílem je najít algoritmus s optimálním poměrem časové a prostorové složitosti. Práce začíná rešerší shlukovacích algoritmů. Rovněž se zabývá způsobem, jak pracovat s velkými datovými sadami, způsobem vizualizace a porovnání výsledků shlukování. Jádro práce představuje návrh řešení tohoto problému s využitím algoritmu *Twister Tries*. Rozebírá jeho implementační detaily a poskytuje výsledky testování z hlediska kvality výsledků a výpočetní náročnosti. Cílem práce bylo experimentálně ověřit, zda stochastickým algoritmem *Twister Tries* dosáhneme stejných výsledků jako s exaktním algoritmem (*average-linkage*). Tento předpoklad se nepovedlo jednoznačně potvrdit. Z poznatků při testování *hashovacích* funkcí vyplývá, že stejných výsledků jsme schopni dosáhnout i s funkcí, která pracuje na nízkém stupni dimenzionality, avšak v mnohem kratším výpočetním čase.

Abstract

This thesis deals with the clustering of tunnels in data obtained from the protein molecular dynamics simulation. This process is very computationaly intensive and it has been a challenge for scientific communities. The goal is to find such an algorithm with optimal time and space complexity ratio. The research of clustering algorithms, work with huge highdimensional datasets, visualisation and cluster-comparing methods are discussed. The thesis provides a proposal of the solution of this problem using the Twister Tries algorithm. The implementation details are analysed and the testing results of the solution quality and space complexity are provided. The goal of the thesis was to prove that we could achieve the same results with a stochastic algorithm – *Twister Tries*, as with an exact algorithm (*average-linkage*). This assumption was not confirmed confidently. Another finding of the hashing functions analysis shows that we could obtain the same results of hashing with a low dimensional hashing function but in much better computational time.

Klíčová slova

Protein, detekce tunelů, shlukovací algoritmy, OptiGrid, Gilpin, Twister Tries, LSH, Java, CAVER.

Keywords

Protein, tunnel detection, clustering algorithms, OptiGrid, Gilpin, Twister Tries, LSH, Java, CAVER.

Citace

ČUDOVÁ, Marta. *Akcelerace algoritmů pro shlukování tunelů v proteinech.* Brno, 2016. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Martínek Tomáš.

Akcelerace algoritmů pro shlukování tunelů v proteinech

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracovala samostatně pod vedením pana Ing. Tomáše Martínka, Ph.D. Uvedla jsem všechny literární prameny a publikace, ze kterých jsem čerpala.

.....
Marta Čudová
24. května 2016

Poděkování

Na tomto místě bych ráda vyjádřila velké poděkování vedoucímu mé práce, Ing. Tomášovi Martínkovi, Ph.D., a konzultantovi, Ing. Jiřímu Honovi, za cenné rady, doporučení a odborné směřování v průběhu jejího zpracování. Dále bych ráda poděkovala svému otci za podporu.

Obsah

1 Úvod **2**

2 Úvod do biologie proteinů **3**
2.1 Proteiny 3

3 Shlukovací algoritmy **8**
3.1 Mřížkové shlukovací algoritmy 8
3.2 Hierarchické shlukovací algoritmy 10
3.3 Metriky pro porovnání výsledků shlukování 18
3.4 Způsoby vizualizace vícedimenzionálních dat 19
3.5 Způsoby ukládání vícedimenzionálních dat 19

4 Kritika současného stavu a cíle práce **22**
4.1 Kritika současného stavu 22
4.2 Cíle práce 23

5 Návrh algoritmu **24**
5.1 Návrh hashovací funkce 24
5.2 Vstupy a výstupy 26
5.3 Návrh datové struktury 27
5.4 Návrh a popis tříd 28
5.5 Návrh procesu shlukování 30

6 Implementace **32**

7 Testování **36**
7.1 Metodika vyhodnocení výstupů 36
7.2 Výběr hashovací funkce a jejích parametrů 37
7.3 Výsledky shlukování 38

8 Závěr **44**

Literatura **46**

Přílohy **49**

A Třídní diagram **51**

B Obsah CD **52**

Kapitola 1

Úvod

Makromolekuly proteinů jsou nedílnou součástí všech živých organismů, ve kterých plní nezbytné životní funkce. Povrchová i vnitřní struktura proteinu je však poměrně komplikovaná. Je tvořena různými výběžky, rýhami, prohlubněmi a dutinami. Navíc se tato struktura vlivem molekulové dynamiky mění v čase. Význam zmíněných útvarů spočívá ve funkci a povaze daného proteinu, protože zpravidla umožňují molekulám okolních látek přístup k aktivním místům, které bývají ukryty uvnitř proteinu. V těchto místech pak dochází k reakci látek s daným proteinem. Produkt této reakce poté odchází z proteinu pryč stejnou nebo jinou cestou, avšak podobného charakteru. Složitá proteinová struktura tvořená zmíněnými útvary slouží jako selektivní filtr pro substráty. Molekuly substrátů, tj. okolních látek, mají různé tvary i velikosti a stejně tak i útvary ve struktuře proteinu jsou různě složité. Představíme-li si, například, útvar připomínající tunel nebo rouru, který vede strukturou proteinu od jeho povrchu až k aktivnímu místu, pak takový útvar díky svému průměru a tvaru dokáže propustit jen určitou část molekul látek, které vyhovují svou velikostí i tvarem, a jsou tedy schopné projít daným útvarem až k aktivnímu místu. Tímto se makromolekula proteinu z části brání reakci s nevhodnou látkou a vznikl tak nežádoucí produkt.

Studium dynamických vlastností proteinových struktur a jejich schopností reagovat s jinými látkami je velkou motivací pro biology a chemiky. Porozumět podrobně principu, jak spolu látky mohou reagovat přispívá k lepšímu pochopení reakcí a vzniku známých látek, ale také může přispět k získání nových znalostí, díky kterým budeme schopni molekuly látek pozměnit tak, aby reagovaly jinak, než je tomu doposud v přírodě. To může pomoci při vývoji a výzkumu nových látek v různých oblastech lékařství a medicíny, biochemie nebo průmyslu.

Proces hledání a lokalizace aktivních míst proteinových struktur je velice výpočetně náročný, zejména kvůli simulaci složité molekulové dynamiky. Výsledkem takových simulací je obrovské množství (sta tisíce až miliony) potenciálních výskytů zájmového útvaru v čase. Tato data je dále nutné zpracovat pomocí shlukování a v obrovském množství dat najít pouze několik nejvýznamějších výskytů tunelů. Cílem této práce je shlukování tunelů v proteinech z velkého množství statických snímků makromolekuly proteinu zachycené v čase. Shlukovací algoritmy jsou výpočetně velmi náročné a je tedy snahou hledat nové algoritmy a metody, pomocí kterých by se uvedený problém dal řešit s nižší časovou a paměťovou složitostí. Touto výzvou se zabývá i tato práce.

Kapitola 2

Úvod do biologie proteinů

2.1 Proteiny

Proteiny jsou organické makromolekulární látky, které jsou tvořeny jedním nebo více polypeptidovými řetězci. Jsou podstatou všech živých organismů, kde plní různé funkce, např. strukturní, obranné nebo senzorické. [8]

Struktura proteinů

Struktura proteinů vychází z uspořádání aminokyselin v polypeptidovém řetězci. Způsob uspořádání má vliv na funkci proteinů.
Podle úrovně abstrakce lze rozdělit strukturu proteinů do čtyř skupin (viz obr. 2.1) [20, 34]:

Primární struktura Přesné pořadí a počet aminokyselinových zbytků v polypeptidovém řetězci, ve kterém jsou tyto zbytky spojeny peptidovou vazbou.

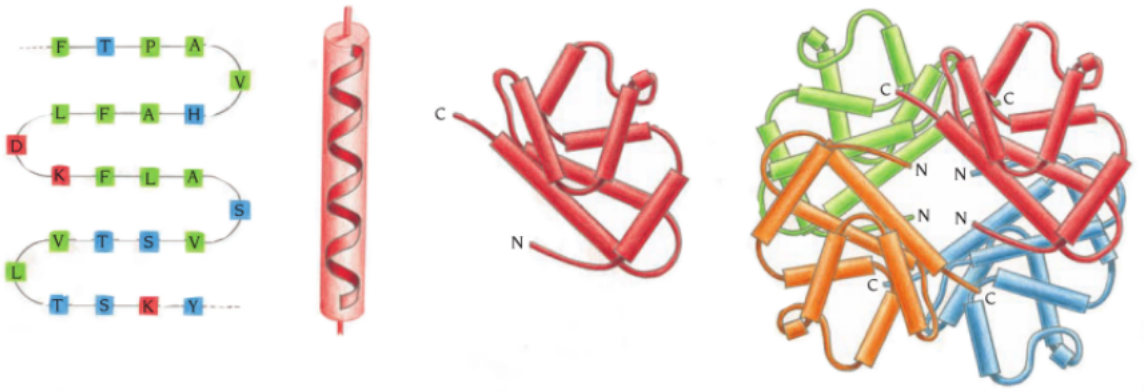
Sekundární struktura Prostorové uspořádání částí polypeptidového řetězce a spojení jednotlivých aminokyselin vodíkovými můstky, které toto spojení stabilizují. Mezi časté stavební motivy sekundární struktury patří *alfa*-šroubovice a *beta*-skládaný list.

Terciární struktura Prostorové uspořádání celé molekuly. Je charakterizována dalšími intramolekulárními vazebnými interakcemi, jako jsou disulfidické můstky, van der Waalsovy síly a iontové vazby [34]. Tato struktura je vytvořena spojením několika strukturních elementů, tj. elementů sekundární struktury do jedné, skrze interakce postranních řetězců. Druhou možností je, že struktura je tvořena několika kompaktními kulovitými jednotkami zvané domény [8].

Kvartérní struktura Protein na nejvyšší úrovni abstrakce může být složen z několika polypeptidových řetězců uspořádaných do kvartérní struktury. Podjednotky jsou spojeny nekovalentními vazbami.

Aktivní místa

Aktivní, neboli vazebná, místa jsou tzv. prázdnými místy ve struktuře proteinu. Nazývají se prázdnými, protože neobsahují žádný atom proteinu, ale jsou vyplněny malými molekulami solventu. Tato místa hrají významnou roli v interakcích s jinými molekulami a proto významně ovlivňují i samotnou funkci proteinu. Velikost aktivních míst a jejich četnost



Obrázek 2.1: Struktury proteinu. Zleva doprava: primární, sekundární, terciární, kvartérní [8].

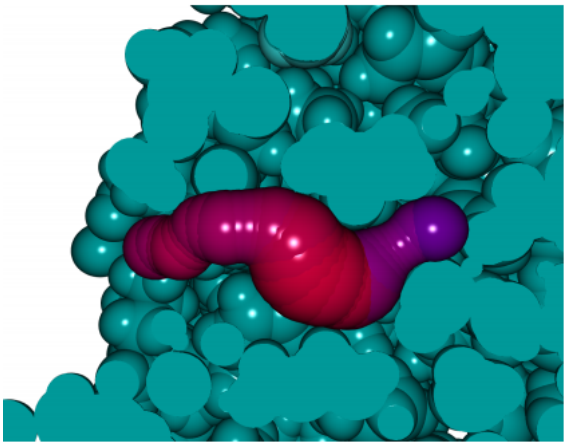
výskytu je závislá na atomovém rozložení samotného proteinu [21]. Tvoří tak specifická místa na povrchu proteinu nebo ve vnitřních dutinách. Zde na sebe dokáží navázat jiné volné molekuly, které se vyskytují v okolí proteinu a za určitých podmínek mezi nimi může proběhnout katalytická reakce, čímž dojde k transformaci volné molekuly na jinou.

U již dobře strukturně popsaných proteinů je pozice těchto míst známá, avšak u nově vzniklých proteinů je tato místa nutné najít. Algoritmus nebo jiná univerzální metoda, která by tato aktivní místa byla schopna najít se 100% úspěšností však neexistuje. Řada již existujících nástrojů se soustředí na nalezení pozice aktivního místa u specifické skupiny proteinů. Taková metoda pak u jiné skupiny proteinů často nemá tak vysokou úspěšnost.

Velký význam hledání těchto míst je např. při výrobě léků nebo jiných chemických látek. Přitom se využívá reakce aktivního místa proteinu s nějakou okolní molekulou, kdy dochází ke transformaci proteinu nebo okolní molekuly.

Jelikož struktura každého proteinu je poměrně komplikovaná, rozlišujeme několik druhů útvarů – aktivních míst – rozlišených podle svého tvaru a podle výskytu v rámci molekuly proteinu [20]. Útvary v proteinové struktuře jsou velice zajímavé z hlediska funkce proteinu, proto se vědecké komunity zabývají jejich studiem a výzkumem. Mezi takové útvary patří např. tunely, póry, kanály, dutiny a kapsy. Ve své práci se dále zaměřím podrobněji pouze na popis tunelů, kterými se má práce z velké části zabývá. Termín „tunel“ může mít dvojí význam. Může se jednat o propojení dvou míst ukrytých v makromolekule proteinu, tj. aktivní místa ve vícefunkcionálních enzymech [28] nebo o cestu ve struktuře proteinu mezi vnitřní dutinou molekuly (tzv. kavita [21]) a povrchem proteinu. Tunel tak spojuje okolní prostředí molekuly s aktivním místem, ukrytým uvnitř struktury proteinu. Ve své práci se zaměřuji právě na tento typ tunelu. Tunel tvoří přístupovou cestu pro molekuly substrátu a odchodovou cestu pro molekuly produktu. Pokud se molekula substrátu dostane až k aktivnímu místu, kde dojde k reakci, molekuly produktu se mohou, ale nemusí vracet stejnou cestou, která byla přístupovou pro substrát. Tunel svou tloušťkou (nejužším místem) limituje velikost molekuly substrátu, která je schopná tunelem projít. Tímto způsobem dochází k selekci látek, se kterými by mohlo dojít k reakci. [32]

Příklad tunelu nalezeného ve struktuře proteinu znázorňuje obr. 2.2. Existujícími nástroji zabývajícími se problematikou proteinových tunelů a jejich vyhledáváním, jsou např. programy *CAVER* [28], *MOLE* [31], *MolAxis2* [30].



Obrázek 2.2: Tunel nalezený uvnitř makromolekuly proteinu. Převzato a upraveno z [20].

Detekce tunelu

Cílem při hledání tunelů je najít nejpravděpodobnější trajektorii v makromolekule proteinu, podél které se okolní molekuly mohou dostat z vnějšího povrchu makromolekuly až k jejímu aktivnímu místu. Podle [28] je tunel definován následovně:

Definice 1 (Tunel) *Nechť M je konečná množina koulí reprezentující atomy makromolekulárního systému. Pak tunel, spojující místa A a B, je sjednocením tzv. prázdných koulí takových, že:*

- *středy těchto koulí tvoří křivku spojující body A a B,*
- *poloměr těchto koulí je maximální možný, a zároveň takový, že průnik takové koule s atomem z M je prázdný,*
- *středová trajektorie leží mezi A a B v rámci prostorových hranic M.*

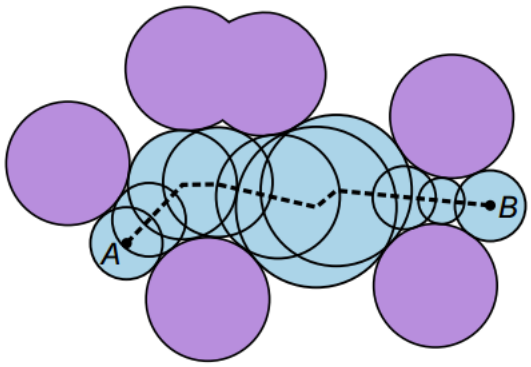
Obrázek 2.3 níže znázorňuje příklad tunelu. Úzké místo tunelu pak tvoří koule s minimálním poloměrem.

Pro výpočet polohy tunelů se používají algoritmy využívající *Delaunayho* triangulaci a konstrukci *Voroného* diagramu [28]. *Voroného* diagram dekomponuje metrický prostor do konečné množiny *Voroného* buněk. Z tohoto diagramu pak můžeme určit trajektorii, po které vnější molekula může putovat až k aktivnímu místu proteinu. Vztah mezi *Voroného* diagramem a *Delaunayho* triangulací splňuje princip duality, tzn. že mezi oběma strukturami lze libovolně přecházet. Výpočtem *Delaunayho* triangulace lze získat *Voroného* diagram a naopak. Tuto schopnost vyjadřuje obrázek 2.4. [32]

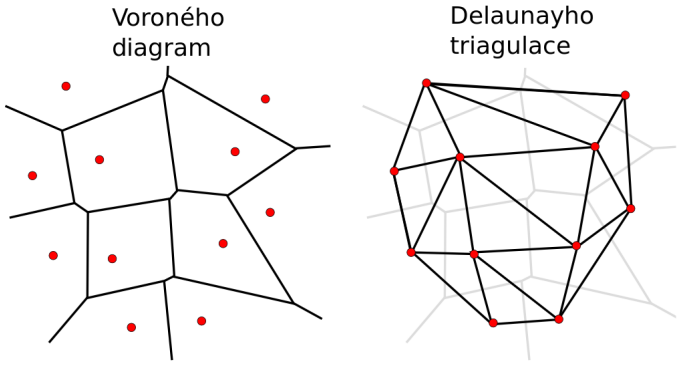
Molekulární dynamika

Při studiu tunelů a jejich detekci v rámci makromolekul proteinů nás zajímá chování celé makromolekuly v čase. Zajímáme se tedy o její molekulární dynamiku, což je simulace vývoje daného proteinu v čase, kdy se projeví jeho strukturní změny způsobené interakcemi s okolními částicemi. To zahrnuje i možné změny výskytu tunelů. V rámci makromolekuly

[1] Převzato z www.diku.dk/students/myth/eos.

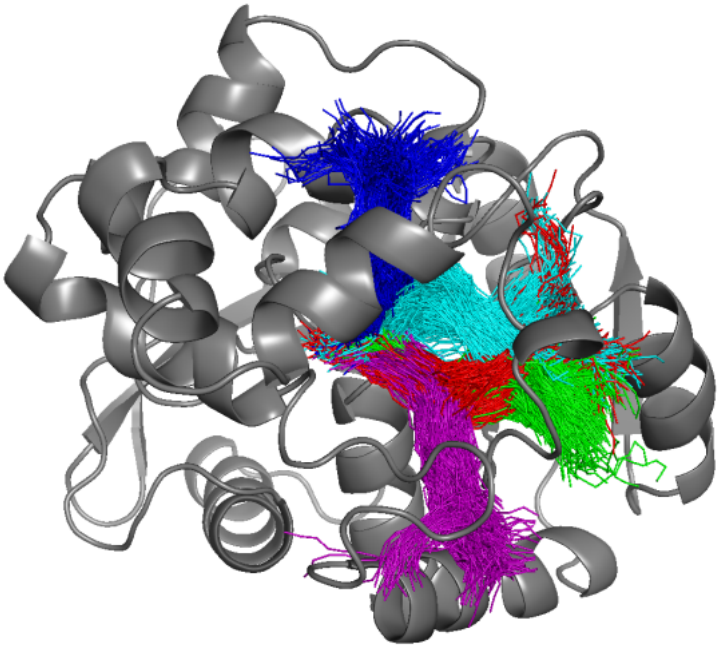


Obrázek 2.3: Tunel spojující místa A a B. Přerušovaná křivka představuje středovou trajektorii tunelu, modré kruhy představují koule formující tunel, a fialové kruhy atomy z konečné množiny koulí M, reprezentující atomy makromolekuly [28].



Obrázek 2.4: Vztah mezi *Voroného* diagramem a *Delaunayho* triangulací. Převzato[1] a upraveno.

se může vyskytovat více tunelů a zároveň se mohou různě transformovat a v daném místě vznikat i zanikat. Tento vývoj jsme schopni zachytit právě pomocí molekulární simulace. Výsledkem simulace je značné množství statických snímků proteinu, ve kterých jsou tunely lokalizovány pomocí algoritmů zmíněných v kapitole 2.1. Nad těmito daty je poté potřeba provést shlukovou analýzu a zaměřit co nejpřesněji výskyt pouze malé nejpravděpodobnější podmnožiny tunelů v daném proteinu. O tomto problému pojednává následující kapitola 3. Obrázek 2.5 znázorňuje pět různých tunelů, které se povedlo najít pomocí shlukovacího algoritmu. Z obrázku je patrno, že každý tunel je tvořen shlukem několika jeho možných výskytů.



Obrázek 2.5: Vizualizace výsledku shlukování. V makromolekule proteinu (znázorněno šedými šroubovicemi) jsou barevně znázorněny nalezené a nejlépe ohodnocené tunely [28].

Kapitola 3

Shlukovací algoritmy

Tato kapitola pojednává o vybraných typech shlukovacích algoritmů. Běžně používané shlukovací algoritmy si často neumí efektivně poradit s velkou datovou sadou, na kterou jsou aplikovány. Jejich další nepříznivou vlastností je kvadratická časová, ale i prostorová složitost [24]. Při svém studiu jsem se tedy zaměřila na hierarchické a mřížkové shlukovací algoritmy, zejména na *Gilpinův* algoritmus [13], *OptiGrid* [10, 36, 33] a *Twister Tries* [11].

3.1 Mřížkové shlukovací algoritmy

Mřížkové shlukovací algoritmy [10] rozdělují datový prostor do konečného počtu buněk, čímž vytváří mřížkovou strukturu nad tímto prostorem. Z buněk mřížky se poté vytvářejí shluky (z angl. *clusters*). Shluky odpovídají místům, kde jsou data tzv. *hustější*, než je jejich okolí, tzn. místa, kde je koncentrace bodů větší než v jejich okolí. Velkou výhodou tohoto přístupu je značná redukce časové náročnosti, zejména pro velké objemy dat. Přístup založený na mřížce shlukuje sousední data na úrovni mřížky. Nepracuje tedy přímo s datovými body, ale s jejich reprezentací pomocí buněk mřížky.

Mřížkové shlukovací algoritmy obecně pracují následujícím způsobem:

1. Rozdělení datového prostoru do mřížky o konečném počtu buněk.
2. Výpočet hustoty v každé buňce mřížky.
3. Seřazení buněk podle jejich hustoty.
4. Určení středů shluků.
5. Průchod přes sousední buňky.

Nevýhodou tohoto přístupu je jeho náchylnost k následujícím datovým problémům:

- Neuniformita – použití pravidelné mřížky pro rozdělení dat nemusí být vždy postačující k dosažení požadované shlukovací kvality a efektivity pro vysoce nepravidelné distribuce dat.
- Lokalita – efektivita je limitována předdefinovanou velikostí buněk mřížky, jejích hranic a prahem hustoty u významných buněk.

- Dimenzionalita – výkonnost závisí na velikosti mřížkových struktur, které se mohou značně zvětšit s rostoucím počtem dimenzí. Pro vícedimenzionální data pak tyto algoritmy nemusí škálovat.

K překonání prvního problému zmíněného výše lze použít adaptivní mřížkové shlukovací algoritmy, např. *AMR* [22] nebo *MAFIA* [14]. Problém vyskytující se obecně u shlukování vícedimenzionálních dat vychází z faktu, že středy shluků je mnohem obtížnější určit, vlivem velkého množství téměř totožných datových bodů, než u dat s méně dimenzemi [18].

Příklady shlukovacích algoritmů, které si umí dobře poradit i s vícedimenzionálními daty a tudíž by pro problém, který řeším (viz kapitola 4), mohly být vhodnými kandidáty, jsou např. *OptiGrid*, *CLIQUE* [6], *MAFIA* [14], *O-cluster* [26]. Při svém studiu jsem se zaměřila podrobněji na algoritmus *OptiGrid*, jehož vlastnosti nastíním v následující podkapitole.

OptiGrid

Algoritmus *OptiGrid* [10, 36] je navržen tak, aby bylo možné získat optimální mřížkové rozdělení dat, viz obr. 3.1. *OptiGrid* rekurzivně rozděluje data do podmnožin. Každá z podmnožin obsahuje minimálně jeden shluk, a je také zpracována rekurzivně. Rekurze končí v okamžiku, kdy již nelze shluk dále dělit, tj. není možné získat dobré průřezové roviny [36]. Obecně algoritmus pracuje následovně [33]:

1. Pro každou dimenzi:
   - Vygeneruj histogram hodnot.
   - Urči velikost šumu v datech. Pokud dimenzionalita není příliš vysoká, může se velikost šumu určit i manuálně, v ostatních případech je tuto fázi nutné automatizovat. Odhad velikosti šumu lze provést výpočtem funkce hustoty a její vizualizací [18].
   - Z histogramu hodnot najdi nejlevější a nejpravější maximum, a poté $(q-1)$ maxim mezi nimi, kde q je počet rovin, kterými chceme rozdělit data (vstupní parametr algoritmu).
   - Najdi q minim nacházející se mezi maximy nalezenými v předchozím kroku. Tyto body reprezentují místa pro potencionální řezy, tj. místa, kterými může procházet průřezová rovina (viz definice 2).
   - Každý nalezený potencionální řez ohodnoť hodnotou hustoty nalezeného bodu.
2. Ze všech dimenzí vyber q nejlépe ohodnocených řezů, tj. řezů s nejnižší hodnotou hustoty.
3. Z vybrané konečné množiny řezů vytvoř mřížku, která rozdělí data.
4. Najdi buňky mřížky s nejvyšší hustotou a přidej je do seznamu shluků.
5. Opakuj kroky 1.–4. pro každý shluk.

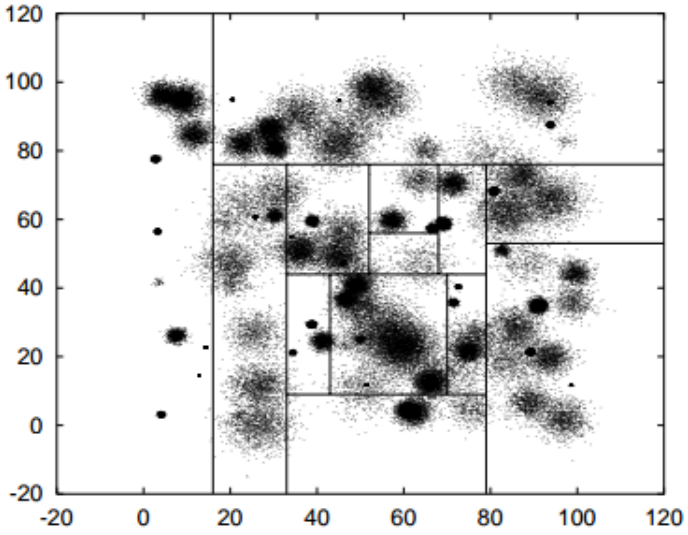
Definice 2 (Průřezová rovina) *Průřezové roviny jsou* $(d-1)$ *dimenzionální roviny, které splňují následující omezení:*

1. *procházejí oblastmi s nízkou hustotou,*
2. *jsou schopny detekovat maximální počet shluků.*

d značí počet dimenzí datové sady [36].

Časová složitost algoritmu *OptiGrid* je $\Theta(d \cdot N \cdot \log N)$, kde d je počet dimenzí a N je počet datových bodů.

Velkou nevýhodou *OptiGridu* je variabilita výběru parametrů, které je nutné znát již na začátku algoritmu. Konkrétně se jedná např. o počet řezů, který by měl být proveden (vstupní parametr q). Počet řezů nelze jednoduše zobecnit. Tento parametr se bude lišit je aplikačně závislý. Liší se tedy u každé úlohy a jeho optimální volba je možná až po dostatečně velké řadě provedených experimentů. Parametr q je vybírán tak, aby stanovil minimální dolní mez hustoty. Z tohoto důvodu jsem tento algoritmus pro svou implementaci nezvolila.
V porovnání s algoritmy *BIRCH* [37] a *DENCLUE* [17], je *OptiGrid* mnohem efektivnější a dokáže se mnohem lépe vypořádat se šumem ve vícedimenzionálních datech [18].



Obrázek 3.1: Určení prvních průřezových rovin algoritmem *OptiGrid* pro $q = 1$ [18].

3.2 Hierarchické shlukovací algoritmy

Jedná se o vícedimenzionální statistickou metodu, využívanou ke klasifikaci dat. Metoda hierarchického shlukování vytváří v konečné datové množině hierarchii – strukturu – shluků dat. Vytváří tedy systém podmnožin, kde průnikem dvou podmnožin je buď prázdná množina nebo jedna z podmnožin [23].

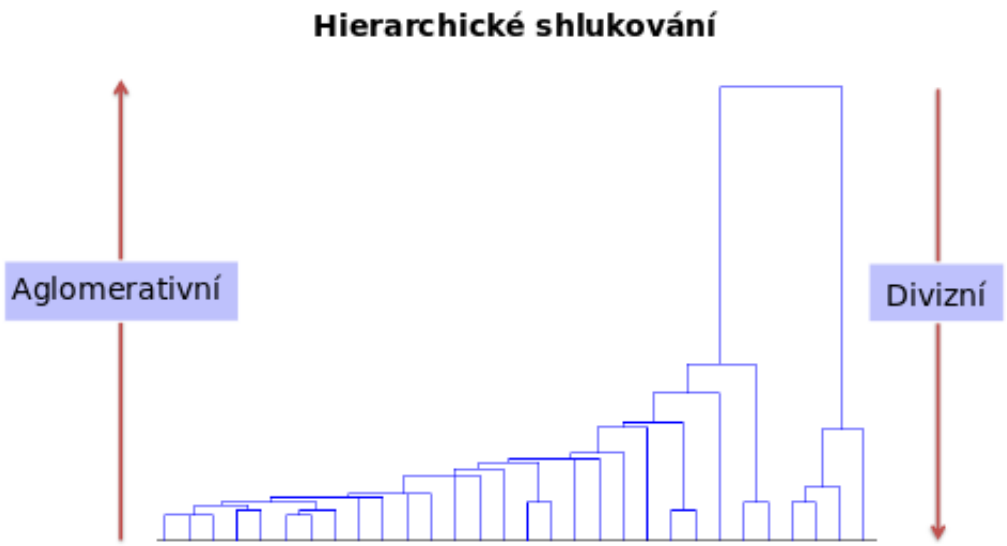
Existují dva přístupy pro tyto algoritmy [24]:

- Aglomerativní (zespodu–nahoru, z angl. *hierarchical aglomerative clustering (HAC)*). Aplikací postupného seskupování objektů se vytváří stromová struktura a výsledkem je konečný celkový shluk. Časová složitost *HAC* je nejméně kvadratická. Mezi typické metody, řadící se do této podkategorie, patří *single-linkage*, *complete-linkage* a *average-linkage* [24].

- Divizní (shora–dolů, z angl. *bandwidth-adaptive clustering (BAC)*). Tento přístup rozděluje počáteční celkový shluk do hierarchického systému objektů. Metoda je aplikována rekurzivně, dokud každý jediný objekt není ve svém vlastním shluku. V určitých případech je tato metoda schopna pracovat mnohem přesněji než *HAC*. Při rozhodování na nejvyšší úrovni těží z celkové informace o globální distribuci dat, zatímco *BAC* se rozhoduje na základě lokálních příznaků.

Bohužel, většina současných algoritmů je ve svých aplikacích limitována časovou složitostí, která je nejméně $\Theta(n^2)$, kde n je počet objektů. Mimo to je výpočet podobnosti mezi dvěma objekty sám o sobě velmi časově náročný vzhledem k velkému počtu dimenzí.

Hierarchické shlukování může nabízet více alternativních řešení. Výsledek procesu shlukování lze vyjádřit např. pomocí dendrogramu (viz obr. 3.2).



Obrázek 3.2: Ukázka dendrogramu. Na obrázku je rovněž vidět, jakým způsobem fungují aglomerativní a divizní shlukovací algoritmy. Převzato[2] a upraveno.

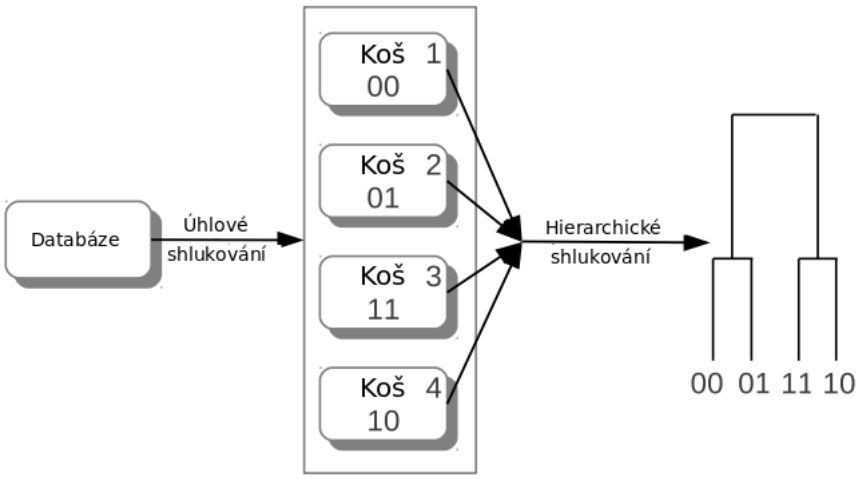
Metody hierarchického shlukování můžeme rozdělit na exaktní a přibližné. Podstatou exaktních metod je výpočet podobnosti mezi jednotlivými body ve shlucích. Představiteli těchto metod jsou např. *single-linkage*, *complete-linkage* nebo *average-linkage* [24].

Mezi představitele přibližných algoritmů hierarchického shlukování patří např. *Gilpinův* algoritmus [13], *Twister Tries* [11] a *HappieClust* [19]. Podstatou těchto metod je, že se nepočítá vzdálenost mezi jednotlivými body shluku tak, jako tomu bylo například u metody *average-linkage*, ale dochází k určitému předzpracování dat, kdy jsou data rozdělena do skupin podle určité metriky, která aproximuje jejich podobnost. Samotné shlukování pak probíhá na úrovni těchto skupin. Dochází zde tedy k určitému zanedbání přesnosti. Na druhou stranu se tyto algoritmy vyznačují menší časovou a prostorovou náročností.

[2]Převzato z www.saedsayad.com/clustering_hierarchical.htm

Gilpinův algoritmus

S *Gilpinovým* algoritmem [13] přichází mezi *HAC* algoritmy nová myšlenka, a to využití úhlového *hashování* pro výpočet souboru *hash* košů (z angl. *buckets*), což přispívá k redukci výpočtů, a tím i ke snížení časové složitosti. Myšlenka *hashování* do košů spočívá v tom, že objekty, které mezi sebou mají malou úhlovou vzdálenost, jsou *zahashovány* do stejného koše, což nám umožňuje vytvořit hierarchii objektů sestávající se z každého koše a provést hierarchické shlukování nad koši. Objektům je *hashovací* funkcí přiřazen *hashovací* kód (binární vektor).



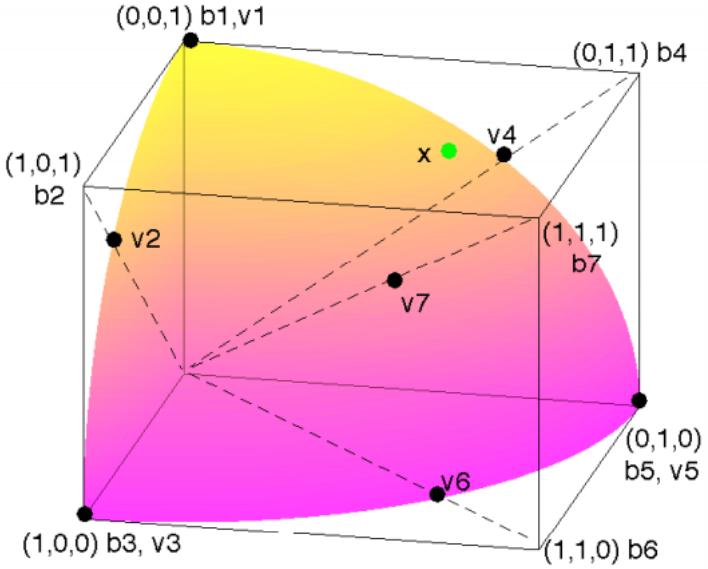
Obrázek 3.3: Schéma dvou kroků algoritmu. Po *zahashování* objektů do košů se provede hierarchické shlukování košů (využívající jejich *hash* kódů) a poté se provede hierarchické shlukování objektů v rámci jednotlivých košů. Převzato a upraveno z [13].

Rychlejšího porovnání podobnosti dvou objektů lze docílit použitím takových binárních kódů, že jejich Hammingova vzdálenost bude aproximovat kosinovou vzdálenost původních vektorů (tj. výpočet Hammingovy vzdálenosti pro binární kódy je mnohem rychlejší než výpočet kosinové vzdálenosti mezi původními vektory v plovoucí řádové čárce). Takové kódy pak lze pro celou datovou sadu zkonstruovat v lineárním čase. Toto však platí pouze pro datové sady, jejichž objekty mají kladné souřadnice, a nachází se tedy v prvním kvadrantu souřadnicového systému. V praxi se používají binární kódy s proměnlivou délkou, tj. délkou větší než je počet dimenzí datové sady.

Získání binárních kódů pro kladný reálný vektor se provede jeho namapováním na jednotkovou hyperkouli (viz obr. 3.4 – zelený bod x). Dále se provede přiřazení nejbližšího vrcholu d-dimenzionální jednotkové hyperkostky, jejíž souřadnice vrcholů představují hledané binární kódy. [15]

Mezi výhody tohoto algoritmu patří [13]:

- Lineární časová i prostorová složitost.
- Poskytuje porovnatelné výsledky s exaktními hierarchickými algoritmy (viz podkapitola 3.3) a pro některé podobnostní, tj. vzdálenostní, funkce dává dokonce lepší výsledky než klasické *HAC* algoritmy.



Obrázek 3.4: Ukázka mapování vektorů na hyperkouli a přiřazení vrcholu kostky [15].

- Díky vztahu mezi Hammingovou a kosinovou vzdáleností se rozšiřuje jeho aplikovatelnost na více typů vícedimenzionálních datových sad.

Gilpinův algoritmus jsem si pro svou implementaci nevybrala kvůli způsobu, jakým se provádí mapování bodů přes hyperkouli na hyperkostku. Systém s body tunelů obsahuje kladné i záporné souřadnice a nebyl by tedy problém počátek tohoto systému posunout tak, aby se všechny jeho body nacházely v prvním kvadrantu souřadnicového systému hyperkoule. Pokud by však nastala situace, kdy dva body odlišných tunelů leží v přímce, avšak oba představují zcela jiný tunel na jiné straně makromolekuly proteinu, mohlo by dojít k namapování obou těchto bodů na stejný bod v hyperkouli, a pak tedy i v hyperkostce. Dva naprosto rozdílné vektory by pak reprezentoval stejný binární vektor (kód) a došlo by tak k chybě v procesu shlukování.

Twister Tries

V této podkapitole se podrobněji zaměřím na popis algoritmu a datové struktury *Twister Tries* [11]. Dále na způsob *hashování*, kterého využívá, a datovou strukturu *LSH Forest* [11], ze které vychází. Na závěr se zaměřím na jeho časovou a prostorou složitost.

Hashování citlivé na lokalitu

Metoda *hashování* citlivého na lokalitu (z angl. *locality-sensitive hashing, LSH*)[29] slouží k nalezení nejbližších sousedů v rámci datové sady, tj. tzv. kandidátních párů. Nejbližším sousedem rozumíme takový bod, který splňuje zadané kritérium, kterým zpravidla bývá vzdáleností metrika, na jejímž základě se rozhoduje, zda spolu tyto body ještě korespondují či nikoliv. Mezi běžně používané vzdálenostní metriky patří např. *Euklidovská, Manhattanská, Jaccardova* nebo *Hammingova* vzdálenost [29]. Za účelem najít kandidátní páry se zavádí pojem rodiny nezávislých *hashovacích* funkcí (viz definice 3), které *hashují* podobné objekty, tj. objekty splňující požadovaná kritéria metriky, k sobě, a odlišné objekty od sebe.

V okamžiku, kdy jsou všechny objekty již *zahashovány* funkcemi z této rodiny a je vytvořena roztříděná databáze, je možné vyhledat nejbližšího souseda k zadanému bodu. K zahájení procesu vyhledávání je nutné ještě tento bod, který je předmětem vyhledávání, *zahashovat* stejnými *hashovacími* funkcemi. Výsledkem procesu vyhledávání je podmnožina objektů databáze, jejíž objekty byly *zahashovány* se stejnými výsledky jako dotazovaný bod. Tyto objekty lze z pohledu použitého rozhodovacího kritéria a metriky považovat za sobě si blízké.

Rodina *hashovacích* funkcí musí splňovat tato kritéria:

1. Funkce musí být schopné s větší pravděpodobností určit dva blízké body, tj. body mezi nimiž není vzdálenost dle použité metriky větší, než je požadovaná mez, jako kandidátní pár, než dva vzdálené.

2. Funkce musí být statisticky nezávislé, tj. schopnost odhadnout pravděpodobnost, že dvě a více funkcí budou dávat určitou odezvu danou produktovým pravidlem (viz definice 3) pro nezávislé události.

3. Musí pracovat efektivně v těchto směrech:

   (a) Schopnost identifikovat kandidátní páry v čase mnohem kratším, než by trval průchod přes všechny prvky v souboru dat.

   (b) Funkce musí být kombinovatelné. Účelem kombinovatelnosti je vytvořit funkce, které lépe zamezí výskytu chybných shod a chybných neshod. Platí pro ně, že čas nutný k identifikaci kandidátních párů je mnohem kratší, než čas potřebný na průchod přes všechny páry.

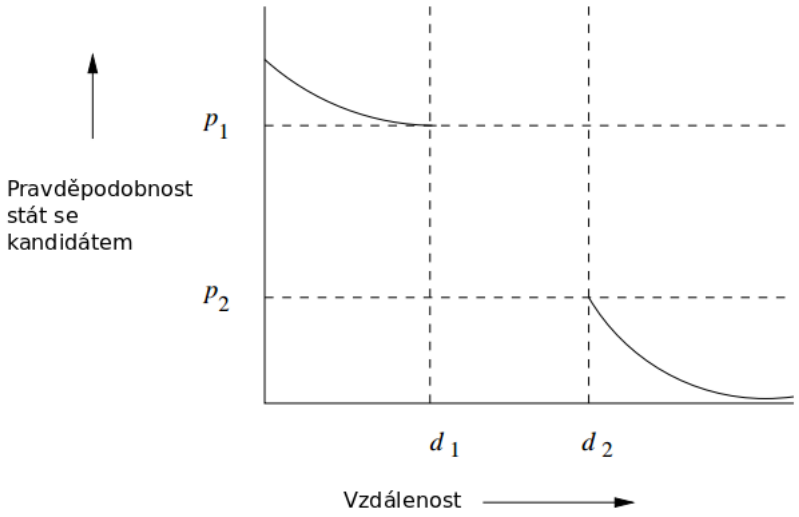
Definice 3 (Rodina funkcí citlivých na lokalitu) *Nechť H je rodina hashovacích funkcí, která provádí mapování z domény D do univerza U, a nechť d je metrika vzdálenosti definovaná na D. Pak, je-li $d_1 < d_2$, H se nazývá (d_1, d_2, p_1, p_2)-citlivá pro každou dvojici $p, q \in D$ a všechny $h \in H$, jestliže:*

- $d(p,q) \leq d_1$, *pak* $Pr[h(p) = h(q)] \geq p_1$,
- $d(p,q) \geq d_2$, *pak* $Pr[h(p) = h(q)] \leq p_2$,

kde $p_1 > p_2$. [11]

Definice 3 nám říká, že pracujeme v určité zájmové doméně D, ve které je vzdálenost mezi dvěma body definována metrikou d. Rodina funkcí H má tu vlastnost, že pokud náhodně vybereme funkci h, pak výsledkem aplikace h na body, které si jsou blízké ($d(p,q) \leq d_1$), pravděpodobně bude produkce stejných výsledků ($Pr[h(p) = h(q)] \geq p_1$). Naopak, aplikací h na body, které jsou od sebe vzdálené, bude pravděpodobnost produkce stejných výsledků velmi malá. Obr. 3.5 ilustruje předpoklad o pravděpodobnosti, s jakou *LSH* rodina označí dva objekty za kandidátní pár.

Jelikož pravděpodobnosti p_1 a p_2 si mohou být blízké, použití pouze jediné funkce h z H k rozhodnutí o podobnosti bodů nemusí být postačující. Řešením tohoto problému může být tzv. amplifikace *LSH* (viz 3.2). Rodinu *LSH* funkcí H můžeme využít ke konstrukci *hashovacích* tabulek tak, jak je ukázáno níže. Skupinu takových *hashovacích* tabulek označujeme jako *LSH Index*.



Obrázek 3.5: Chování (d_1, d_2, p_1, p_2)-závislé funkce. Převzato a upraveno z [29].

Amplifikace LSH Uvažujme (d_1, d_2, p_1, p_2)-citlivou rodinu H. Pod pojmem amplifikace *LSH* (z angl. *amplification*) budeme chápat konstrukci nové *LSH* rodiny H' takové, že H' nad H může být vytvořena pomocí [29]:

AND-konstrukce Každý člen z H' se skládá z r *hashovacích* funkcí vybraných náhodně z H, kde r je libovolné fixní kladné číslo. O takovém h, pro které platí $h \in H'$, a to, že bylo vytvořeno z konečné množiny $\{h_1, h_2, \ldots, h_r\} \in H$, můžeme říci, že $h(x) = h(y)$ tehdy, a jen tehdy když $h_i(x) = h_i(y)$, pro všechna $i = 1, 2, \ldots, r$. Takto vytvořená nová rodina se pak nazývá $(d_1, d_2, (p_1)^r, (p_2)^r)$-citlivá.
Pro libovolný bod s z množiny všech bodů S platí, že se s umístí do koše se štítkem (z angl. *label*) $g(s) = (h_1(s), h_2(s), \ldots, h_r(s))$. Zde vidíme, že každá $h_i, i > 0$, produkuje jeden symbol a každý koš je identifikován štítkem tvořeným r symboly [7].

OR-konstrukce Každý člen z H' je zkonstruován pomocí b členů z H, tj. z množiny $\{h_1, h_2, \ldots h_b\}$. Tato konstrukce definuje, že $h(x) = h(y)$ platí tehdy, a jen tehdy když, $h_i(x) = h_i(y)$, pro jednu nebo více hodnot i. Takto vytvořená rodina funkcí se nazývá $(d_1, d_2, 1-(1-p_1)^b, 1-(1-p_2)^b)$-citlivá.

AND-konstrukce snižuje všechny pravděpodobnosti, tj. produkci kolizí, zatímco *OR*-konstrukce tyto pravděpodobnosti zvyšuje. Vytvořením kaskády z *AND*- a *OR*- konstrukcí v jakémkoliv pořadí, můžeme spodní hranici pravděpodobnosti posunout blíže k hodnotě 0 a horní hranici blíž k hodnotě 1.

AND- i *OR*- konstrukce konkatenuje k rozdílných *hashovacích funkcí*, které identifikují koš. Dva vzdálené body mají pravděpodobnost kolize p_2 s jednou *hashovací* funkcí. S konkatenací však jejich pravděpodobnost kolize odpovídá $(p_2)^k$, popř. $1-(1-p_2)^k$. Z čehož plyne, že malá hodnota k bude produkovat nepravé shody (z angl. *false positives*) a naopak vysoká hodnota k bude mít za následek snížení možnosti kolize blízkých bodů. Tento problém řeší konstrukce vícenásobných *hashovacích* tabulek, která spočívá v opakované aplikaci *AND*- nebo *OR*- konstrukce. Pro vytvoření l odlišných *hashovacích* tabulek je nutné *AND*- nebo *OR*- konstrukci provést právě $l-$krát s *hashovacími* funkcemi $g_1, g_2, \ldots, g_l$ [7].

LSH Forest

LSH Forest [11, 7] je stromová datová struktura reprezentující hierarchii *LSH* funkcí pomocí *LSH Trees*. *LSH Forest* řeší určité problémy, které se vyskytují ve standardním *LSH* algoritmu, a to:

- Výpočet pouze g_j *LSH* funkcí, kde $j > 0$, nutných k rozlišení dvou bodů od sebe. Tímto dochází k úspoře výpočetního času.
- Použití štítků s variabilní velikostí, což vede k úspoře paměťového místa a eliminaci potřeby konstruovat vícenásobné LSH indexy. Tyto štítky jsou umístěny v prefixovém stromě (datová struktura *trie*) na jeho hranách a slouží jako hodnota *sub-hashovací* funkce g_j. Duplikované prefixy jsou takto uloženy pouze jednou. LSH index používal štítky o fixní velikosti k.

Problémem, který se rovněž vyskytuje i při použití struktury LSH index, je generování příliš dlouhých štítků nutných k odlišení dvou bodů, které si jsou velmi blízké. Tento problém lze řešit omezením maximální velikosti štítku. Pak délka štítku x bodu p je $g(p, x) = (h_1(p)\, h_2(p), \ldots\, h_x(p))$ [7].

LSH Tree je prefixová stromová datová struktura sestávající se ze všech štítků a jejíž listové uzly korespondují s jednotlivými body z datové množiny. $LSH\ Forest$ sestává z $l\, LSH\, Tree$ konstruovaných nezávisle z náhodné sekvence *hashovacích* funkcí H [7].

Rozbor algoritmu Twister Tries

Algoritmus *Twister Tries* [11] funguje jako aproximace k aglomerativním hierarchickým shlukovacím algoritmům. Jeho časová i prostorová složitost závisí na požadované přesnosti, tj. pravděpodobnost, že algoritmus v každé iteraci správně vybere dva shluky, které mohou být spojeny. Pokud je tento parametr fixní, časová i prostorová složitost je lineární s ohledem na počet shlukovaných položek. Rozhodnutí o tom, které shluky budou spojeny do většího shluku, je založeno na aproximaci. Standardní *HAC* algoritmy vyberou dva shluky na základě nejmenší vzdálenosti mezi nimi definované dle zvoleného algoritmu (např. *average-linkage*) a metriky. Algoritmus *Twister Tries* je však stochastickou metodou a své rozhodnutí o výběru dvou shluků provádí na základě pozice nejnižšího rozdělujícího bodu (viz definice 4), resp. spojujícího bodu (viz definice 5). Tento bod obsahuje záznamy obou shluků, tj. poshlukovaných párů v rámci jakéhokoliv stromu–*trie* (viz podkapitola 3.2).
Mezi výhody *Twister Tries*, jakožto algoritmu i datové struktury, patří:

- Výsledky shlukování jsou porovnatelné se standardními *HAC* algoritmy. Výsledkem *Twister Tries* je dendrogram.
- Díky lineární časové a prostorové složitosti, algoritmus škáluje i u datových sad s více jak jedním milionem položek.

Algoritmus využívá *LSH* funkcí ale i datové struktury typu *LSH Forest*. *Twister Tries* pracuje následujícím způsobem:

- Vytvoří shluk pro každý bod a vloží jej do struktury *LSH Forest*.
- V každém shlukovacím kroku (tato část je někdy nazývána jako *twisting* fáze):

  - Získá nejbližšího souseda pro každý shluk a uchová si pouze ten pár, mezi jehož body je nejmenší vzdálenost (určeno náhodně na základě nejnižšího spojovacího bodu).
  - Shluky spojí dohromady tak, že je nejprve vyjme ze stromové struktury a poté do ní vloží jejich sjednocení.
- Algoritmus končí v okamžiku, kdy zbývá pouze jeden shluk.

Hlavní myšlenkou tohoto přístupu je opakovatelné použití *LSH* datové struktury k nalezení nejbližšího páru. Takový výpočet může být dokončen v sub-lineárním čase pouhým zjištěním, zda koše obsahují dva rozdílné body.

Datová struktura Twister Tries

Datová struktura [11] se skládá ze souboru prefixových stromů, kde každý strom je doplněn o rozdělující *splitmap* strukturu. Shluky jsou v rámci stromu (*trie*) reprezentovány *elementy*. Výsledkem výpočtu *hashovacích* funkcí jsou štítky umístěné na hranách mezi dvěma uzly stromu. Dále, každý uzel obsahuje ukazatel na svého rodiče a každý list udržuje seznam ukazatelů na jednotlivé elementy, s kterými daný list koresponduje.

Pojmy užívané v rámci datové struktury *Twister Tries* [11]:

Definice 4 (Rozdělující bod) *Rozdělující bod (z angl. splitpoint) je uzel stromu, který má více jak jednoho potomka, nebo je listem vztahujícím se k více jak jednomu elementu. Rozdělující body mimo jiné uchovávají ukazatel na uzel v lineárním seznamu splitmapy, který ukazuje zpět na rozdělující bod.*

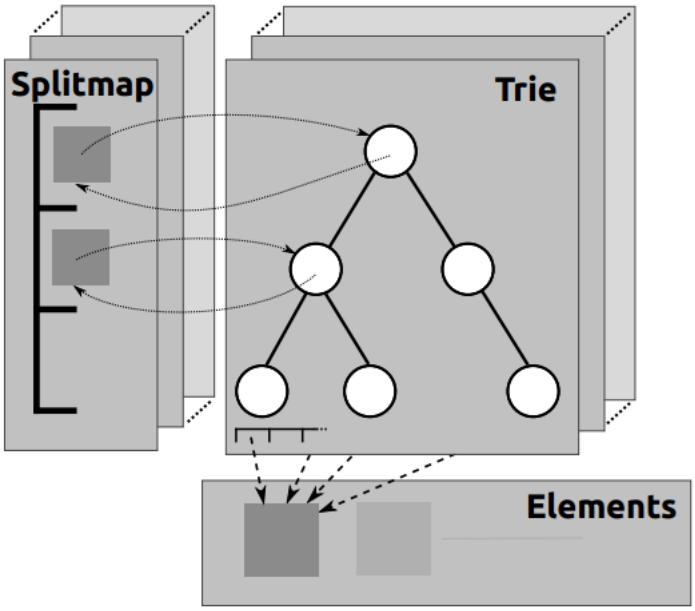
Definice 5 (Spojovací bod) *Spojovací bod (z angl. joinpoint) je nejníže položeným rozdělujícím bodem v rámci stromové struktury. Je významný z hlediska výběru kandidátů pro operaci shlukování.*

Definice 6 (Seznam rozdělujících bodů) *Seznam rozdělujících bodů (z angl. splitmap) doprovází jednotlivé stromové struktury trie. Jedná se o lineární seznam o velikosti maximální hloubky trie (kořenový uzel stromu má hloubku 0). Každá položka v rámci splitmap je sama o sobě lineární seznam obsahující ukazatele na rozdělující body (viz definice 4) stejné úrovně v rámci stromu (trie).*

Elementy představují shluky, které se momentálně vyskytují v systému. Každý element uchovává ukazatele na listové uzly v jednotlivých *trie*, které reprezentují daný shluk. Listové uzly pak zase ukazují zpět do struktury elementů na jednotlivé korespondující elementy. Výše popsané části datové struktury vystihuje obr. 3.6.

Nad datovou strukturou jsou definovány operace vložení, vyjmutí, výběru a spojení. Každá z operací musí být provedena v konstantním čase. Operace vložení je konstantní, jelikož velikost a počet *trie* je fixní, a výpočet *hashovací* funkce je proveden rovněž v konstantním čase. Přidání uzlu do *splitmap* struktury jako rozdělující bod je provedeno pouze jednou a v podstatě se jedná pouze o přidání ukazatele do lineárního seznamu. Přidání elementu do listu stromu má rovněž konstantní složitost $\Theta(1)$. Víceméně stejné důvody platí i pro ostatní operace a tudíž mohou být dokončeny v konstantním čase.

Díky předpokladu o konstantní časové složitosti těchto funkcí, je tedy celková časová složitost algoritmu lineární [11].



Obrázek 3.6: Části *Twister Trie* datové struktury [11].

3.3 Metriky pro porovnání výsledků shlukování

Vyhodnocením výsledků shukování se zabývá disciplína zvaná validace shlukové analýzy. Jejím úkolem je měřit kvalitu shlukování pro různé algoritmy nebo pro stejný algoritmus, který při výpočtu používá odlišné proměnné. Výsledky shlukování lze porovnávat za základě různých kritérií.
Metody, které vyhodnocují kvalitu shlukování daného algoritmu, pracují na pricipu porovnávání výsledků z již provedených shlukových analýz. Vstupem algoritmu porovnání je pak asociační matice a příslušnost jednotlivých objektů do shluků.

Porovnání na základě počtu párů

Porovnání dvou algoritmů je založeno na výpočtu příslušnosti každého páru bodů ke shlukům vygenerovaných pomocí obou algoritmů. Mohou tedy nastat čtyři situace, a to, že testovaný pár bodů se nachází ve shlucích obou algoritmů, ani v jednom nebo pouze v jednom shluku jednoho algoritmu.

Mezi metody, které pracují na tomto principu, řadíme výpočet *Jaccardova* indexu, *Fowlkes-Mallows* indexu, *Randova* indexu atd. [25]

Porovnání na základě shody podmnožin

Metody pracující na tomto principu porovnávají přímo body ve shlucích a hledají nejlepší shodu. Pro porovnání používají kontingenční tabulku, která statisticky vyjadřuje vztah mezi oběma shluky. Zástupcem této metody je např. *van Dongenova* metrika nebo *Larsenova* a *Aoneova* metrika. [25]

3.4 Způsoby vizualizace vícedimenzionálních dat

Vizualizací dat rozumíme grafickou reprezentaci získaných hodnot buď přímo ze shlukovacího algoritmu, nebo z různých měření či porovnání. Pro samotnou vizualizaci může být využito kombinací grafických primitiv, symbolů, čísel apod. U vizualizace vícedimenzionálních dat je nutné využít technik, které pomocí geometrických transformací převádí data z n-dimenzionálního prostoru do nejvýše 3-dimenzionálního prostoru, který je pro nás mnohem srozumitelnější. Ke zvýraznění dalších dimenzí lze použít různé barvy či tvary, avšak taková grafická reprezentace se může stát nepřehlednou. Je důležité si uvědomit, že při každé redukci dimenzí dochází k určité ztrátě informace. To pro účely vizualizace nemusí nijak výrazněji vadit, avšak pro správnou reprezentaci dat je potřebné s tímto faktem počítat. [16]

Mezi nejznámější vizualizační techniky patří[3]:

Scatter Plot Matrix (SPLOM) Tato vizualizační technika umožňuje reprezentovat vztahy mezi více parametry, resp. dimenzemi. Využívá 2D-projekce k zobrazení vztahů mezi každým ze dvou parametrů v rámci datové sady. Jedná se tedy o matici složenou z jednoduchých X-Y grafů (*Scatter Plots*), která poskytuje zobrazení každému atributu vůči všem ostatním. Ukázka této vizualizační techniky je na obr. 3.7.

***Node-link* diagramy** Tato technika se řadí mezi techniky zobrazující hierarchie. Pomocí těchto metod lze rovněž vizualizovat výsledky či průběh shlukovacích metod. *Node-link* diagram patří mezi populární metody, jak zvizualizovat stromové struktury. Kruhový diagram zobrazuje obr. 3.7. Alternativou tohoto diagramu je *dendrogram* (viz obr. 3.2), který je hojně využíván právě při vizualizaci procesu shlukování. Dendrogram umísťuje listové uzly na stejnou úroveň.

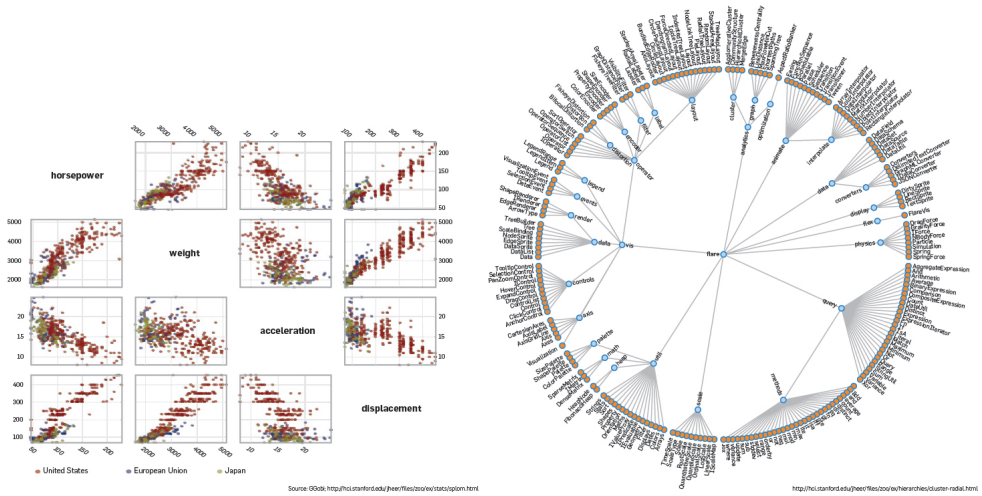
Síťové grafy Tento typ grafů je využíván zejména pro vyjádření vztahů mezi jednotlivými entitami. Z hlediska shlukování nás zajímá zejména grafová vzdálenost, jejíž způsob vyjádření popisuje obr. 3.8.

Parallel Coordinates Tato vizualizační technika nezobrazuje každý pár parametrů ve 2D zobrazení, nýbrž opakovaně vykresluje data na paralelní osy, a poté spojí korespondující údaje pomocí linek. Všechny vertikální linie představují všechny uvažované parametry. Minimální a maximální hodnoty každé dimenze jsou poté nastaveny vůči spodnímu a hornímu bodu vertikálních čar. Výhodou tohoto zobrazení je jeho relativní kompaktnost, kdy velké množství parametrů může být zobrazeno souběžně. Příklad takového zobrazení je na obr. 3.8.

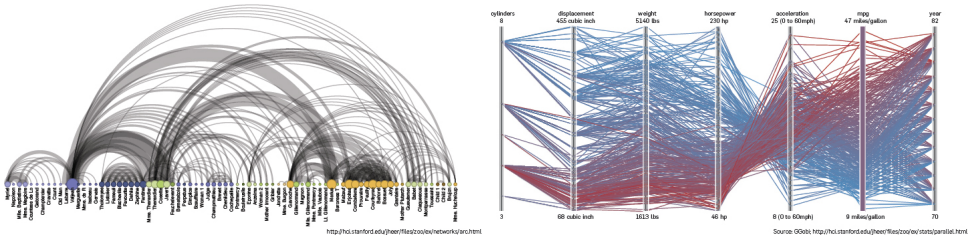
3.5 Způsoby ukládání vícedimenzionálních dat

S rostoucí velikostí datových sad a jejich rostoucí dimenzionalitou, vzniká problém, jak správně pracovat s daty, aby se co nejvíce snížily požadavky na paměťové možnosti a zvýšila se propustnost dat při manipulaci s nimi. Dále je nutné definovat způsob jejich ukládání do souboru tak, aby nedošlo k chybné interpretaci vzhledem k velikosti dat.
Jednou z nejjednodušších možností je použití obyčejného textového souboru. Tato možnost je však vhodná maximálně pro ukládání jednoduchých nebo malých datových sad. Mezi značné nevýhody textových souborů patří:

[3] Kvůli obtížné přeložitelnosti jsou některé názvy technik ponechány v angličtině.



Obrázek 3.7: Ukázka *Scatter Plot Matrix* (vlevo) a *Node-link* diagramu (vpravo) [16].



Obrázek 3.8: Ukázka síťového grafu vyjadřující vztah vzdálenosti mezi entitami (vlevo) a *Parallel Coordinates* (vpravo) [16].

- Nutnost vytvoření vlastního syntaktického analyzátoru.
- Pomalé vstupně výstupní operace.
- Rozdílná interpretace znaků na různých počítačových architekturách a vlivem použitého kódování.
- Žádná schopnost samopopisu dat a detekce chyb.

Další možností je použít binární soubory, které tyto nevýhody odstraňují a jsou přímo navrženy pro práci s velkými a vícedimenzionálními datovými sadami. Tyto soubory jsou hojně používány v aplikacích různými, nejen vědeckými, komunitami a existuje velké množství nástrojů, které takto uložená data umí přímo interpretovat a zobrazovat. Rovněž byla vytvořena řada knihoven pro různé programovací jazyky, které umí pracovat s formátem těchto datových souborů, např. *rhdf5* knihovna pro práci se soubory ve formátu *hdf* v jazyce *R* [12].

Samopopisné datové formáty

Samopopisné datové formáty [27] obsahují ve svých metadatech navigační informaci o tom, jak jsou data uložena a jak se mají sémanticky chápat. Data se do souboru ukládají po menších blocích (z angl. *chunks*). Tímto se redukuje přístupový čas k disku v případě, že přistupujeme k velkému množství dat. Ukládání po blocích rovněž umožňuje aplikaci přímo přistoupit k jakékoliv části datového souboru. Většina datových formátů určených pro vědecké aplikace ukládá data do vícedimenzionálních polí za účelem zjednodušit přístup k těmto datům z aplikace. Tyto aplikace pak na data nahlížejí přes tzv. přístupové šablony, od kterých se odvíjí způsob, jak jsou data čtena.

Mezi používané samopopisné datové formáty patří knihovny *HDF* (*Hierarchical Data Format*)[1], *NetCDF* (*Network Common Data Format*)[5] nebo *FITS* (*Flexible Image Transport System*)[2]. Pro všechny tyto formáty existuje podpora ze strany komerčních, ale i nekomerčních softwarových aplikací. Například pro knihovnu *HDF* existuje podpora pro programovací jazyky *Java, MATLAB, R, Python, C/C++, Fortran* a další.

Kapitola 4

Kritika současného stavu a cíle práce

Hlavním problémem, kterých se ve své práci zabývám, je shluková analýza vícedimenzionálních dat. Data pro shlukovou analýzu jsou získána prostřednictvím simulace molekulární dynamiky proteinu. Výstupem této simulace je velké množství statických snímků makromolekuly proteinu v čase, ve kterých se hledají charakteristické útvary uvnitř makromolekuly. V mém případě se jedná o lokalizaci tunelů. Tyto snímky tedy představují potencionální výskyty tunelů v rámci makromolekuly a je nutné mezi těmito snímky nalézt korespondence a najít tak pouze několik nejpravděpodobnějších výskytů tunelů. Uvnitř makromolekuly proteinu se však může nacházet více tunelů. Simulační data představují velkou vícedimenzionální datovou sadu a proto se pro nalezení korespondencí jeví jako vhodný kandidát shluková analýza. Pro řešení tohoto problému pomocí shlukové analýzy zde navrhuji své řešení.

4.1 Kritika současného stavu

V současné době se na fakultě informačních technologií VUT v Brně ve spolupráci s Masarykovou univerzitou vyvíjí program *CAVER* [28]. Tento program se zabývá tématikou analýzy molekulárních látek. Zabezpečuje dvě hlavní úlohy. První úlohou je detekce tunelů, kanálů a pórů v molekulárních tělesech a anorganických materiálech. Tato úloha se provádí na základě simulace molekulární dynamiky, kde se pro detekci uvažovaných struktur používá *Voroného* diagramu (viz podkapitola pojednávající o detekci tunelů 2.1). Další hlavní úlohou je pak shlukování nalezených struktur a nalezení kandidátních řešení, tj. nalezení korespondujících objektů mezi různými snímky simulace. U problematiky tunelů se počet nalezených struktur běžně pohybuje mezi desíti až sto tisíci [28]. Lze rovněž předpokládat, že s rostoucí délkou simulace, bude růst i počet struktur, které je nutné dále zpracovat pomocí shlukování.

Program *CAVER* [28] aktuálně používá pro shlukování hierarchický algoritmus *average-linkage*. Tento algoritmus počítá vzdálenost dvou shluků tak, že počítá vzdálenosti mezi všemi dvojicemi bodů mezi oběma shluky. Jeho časová i prostorová složitost je tedy vyšší než kvadratická a výpočet se pro velké datové sady stává velmi nákladným a neefektivním. Z tohoto důvodu je proces shlukování úzkým hrdlem celé aplikace.

4.2 Cíle práce

Na základě zhodnocení aktuálního stavu v řešené oblasti jsem v souladu se zadáním práce zformulovala cíle práce. Hlavním cílem mé práce je prostudovat některé existující shlukovací algoritmy a zvolit nejvhodnější pro nalezení kandidátních tunelů v makromolekule proteinu. Provést implementaci algoritmu a experimentálně ověřit jeho funkčnost.

Dílčí a postupné cíle mé práce jsou:

- Prostudování shlukovacích algoritmů používaných v dané oblasti.
- Výběr vhodného kandidáta pro shlukovací algoritmus.
- Vytvoření obecného návrhu tříd pro daný algoritmus.
- Výběr programovacího jazyka a návrh testování.
- Implementace a testování.
- Návrh dalšího postupu.

Kapitola 5

Návrh algoritmu

Z kandidátních shlukovacích algoritmů, zmíněných v kapitole 3 jsem se rozhodla implementovat algoritmus *Twister Tries* a to nejen pro jeho lineární časovou i prostorovou složitost, ale také proto, že se u něj nevyskytují problémy s velkou nejezdnoznačností reprezentace dvou rozdílných bodů jako u *Gilpinova* algoritmu (viz podkapitola 3.2) s velkou variabilitou výběru parametrů jako v případě algoritmu *OptiGrid* (viz podkapitola 3.1).

V aplikaci je tunel identifikován desíti body v 3D prostoru. S tunelem však budu pracovat jako s jedním bodem. Což znamená, že celý tunel bude určen 30 souřadnicemi a problém shlukování se tak bude řešit ve 30-dimenzionálním prostoru. Tunel T tvořený body $\{P_0[x_0, y_0, z_0], P_1[x_1, y_1, z_1], \ldots, P_9[x_9, y_9, z_9]\}$ je pak reprezentován vektorem $\vec{t} = (x_0, y_0, z_0, x_1, y_1, z_1, \ldots, x_9, y_9, z_9)$.

Návrh algoritmu jsem vytvořila tak, aby byl modulární. Algoritmus je tedy možné aplikovat na objekty různého typu, na které lze poté aplikovat různé *hashovací* funkce. Pro imlementaci byl zvolen programovací jazyk *Java* kvůli plánované integraci algoritmu do programu *CAVER* [28], který je rovněž celý implementován v jazyce *Java*. Myslím si, že pokud by byl algoritmus využíván i jinými aplikacemi, bylo by vhodnější použít programovací jazyk *C++*, jehož implementace by byla zcela jistě efektivnější. Popřípadě lze zvolit skriptovací jazyk *Python* kvůli jeho prototypovacím vlastnostem a jeho narůstajícím využitím tohoto jazyka vědeckými komunitami.

5.1 Návrh hashovací funkce

Hashovací funkci jsem navrhla tak, aby splňovala definici 3. Jejím cílem je tedy maximalizovat počet kolizí lokalitně si blízkých tunelů a minimalizovat kolize u tunelů, které jsou od sebe vzdálené.

Návrh *hashovací* funkce vychází z následujícího:

- Velikost koše, která je parametrem *hashovací* funkce, odpovídá svým rozměrem maximální požadované vzdálenosti dvou tunelů, viz obr. 5.1.
- *Hashovací* funkce pracuje ve stanoveném souřadnicovém systému, jehož hranice jsou nastaveny pomocí parametrů této funkce.
- Při výpočtech vzdáleností tunelů uvažuji *Euklidovskou* vzdálenostní metriku [29].

- Funkce převádí reálné souřadnice tunelu na diskrétní, které odpovídají číselnému označení koše.
- Tunely jsou *hashovány* více *hashovacími* funkcemi, které vytvářím prostřednictvím posunu (změnou *offsetu*) počátku souřadnicového systému, viz obr. 5.1. Velikost posunu, tj. *offsetu*, je náhodně zvolena z rozmezí souřadnicového systému dané *hashovací* funkce. Změnou *offsetu hashovacích* funkcí se mění pohled na souřadnice tunelů. Cílem je, aby tunely, které si jsou blízké, měly co největší počet shod ve stejných koších, jež jsou výsledkem různých *hashovacích* funkcí. Počet těchto funkcí je dán součinem počtu a velikosti stromů. Tyto dva parametry, velikost stromu a jejich počet, jsou v algoritmu volitelné.
- Příslušnost bodu ke koši vypočítám tak, že od reálné souřadnice bodu odečtu aktuální *offset* souřadnicového systému a toto číslo vydělím velikostí koše. Získám takto n-dimenzionální číslo odpovídající číselnému označení koše, kde n je počet dimenzí.
- Jednotlivé výsledky *hashovací* funkce ukládám vždy na hranu mezi rodičovský uzel a uzel jeho příslušného potomka v datové struktuře *trie*, viz obr. 5.3.
- Navrhla jsem celkem tři *hashovací* funkce pracující na popsaném principu. Navržené *hashovací* funkce se liší pouze v tom, v jakém rozsahu redukují dimenzionalitu problému:
  - Uvažujme tunel T daný vektorem souřadnic $\vec{d} = (d_0, d_1, \ldots, d_{29})$. Dále uvažujme výsledek *hashování* V daný vektorem:
    1. $\vec{v} = (v_0, v_1, \ldots, v_{29})$. V tomto případě nedochází k žádné redukci dimenzionality. Jedná se tedy o zobrazení, které není prosté a zobrazuje čísla typu *double* z 30-dimenzionálního prostoru souřadnic tunelů S_T na čísla typu *integer* z 30-dimenzionálního prostoru souřadnic košů (výsledků *hashovací* funkce) S_V. Toto zobrazení lze zapsat jako $f : S_T \to S_V$. Výpočet *hashovací* funkce je následující:

$$v_i = \frac{|d_i - offset|}{velikost\_koše}, \quad kde\ i = 0,\ \ldots,\ 29. \tag{5.1}$$

    2. $\vec{v} = (v_0, v_1, \ldots, v_9)$. V tomto případě dochází k redukci dimenzionality z 30 na 10. Jedná se tedy opět o zobrazení, které není prosté a zobrazuje čísla typu *double* z 30-dimenzionálního prostoru souřadnic tunelů S_T na čísla typu *integer* z 10-dimenzionálního prostoru souřadnic košů (výsledků *hashovací* funkce) S_V. Toto zobrazení lze zapsat jako $f : S_T \to S_V$. Výpočet *hashovací* funkce je následující:

$$v_i = \sum_{j=0}^{2} \frac{|d_{3\cdot i+j} - offset|}{velikost\_koše} \cdot \frac{1}{3}, \quad kde\ i = 0,\ \ldots,\ 9. \tag{5.2}$$

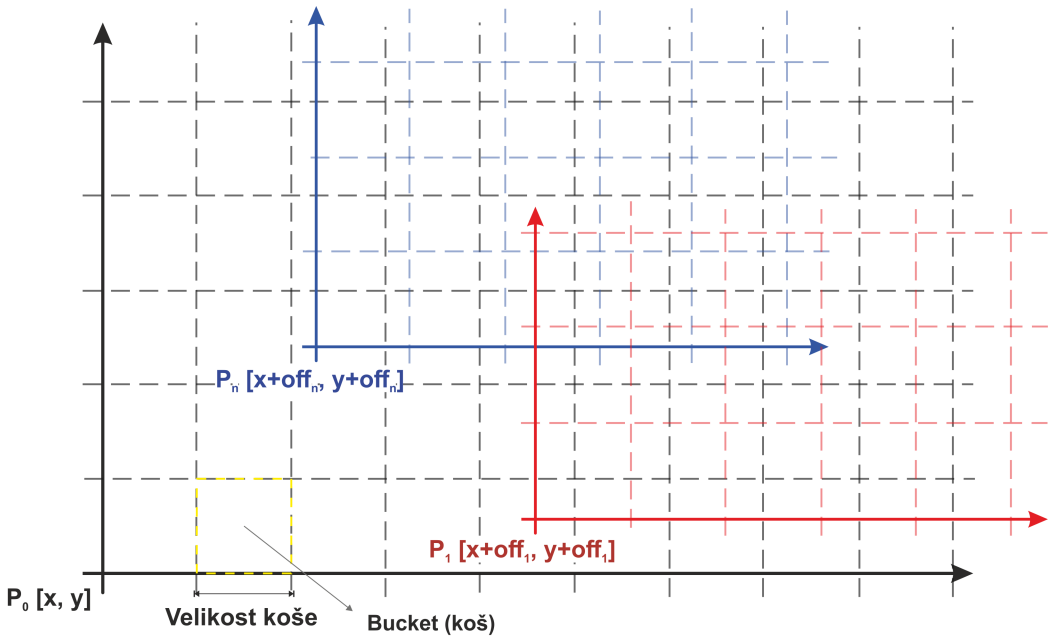
       Výpočet spočívá v tom, že 30-dimenzionální vektor tunelu T je rozdělen na deset 3D bodů. Je provedena suma výpočtu každé ze tří souřadnic, které tvoří daný bod. Tato hodnota je zprůměrována a výsledek je uložen do vektoru $\vec{v}$.

3. $\vec{v} = (v_0)$. V tomto případě dochází k největší redukci dimenzionality, a to z 30 na 1. Zobrazení, které není prosté a zobrazuje čísla typu *double* z 30-dimenzionálního prostoru souřadnic tunelů S_T na čísla typu *integer* z 1-dimenzionálního prostoru souřadnic košů (výsledků *hashovací* funkce) S_V, zapíšeme jako $f : S_T \to S_V$. Výpočet *hashovací* funkce je následující:

$$v_i = \sum_{i=0}^{29} \frac{|d_i - offset|}{velikost\_koše} \cdot \frac{1}{30} \tag{5.3}$$

Výpočet je proveden jako suma výpočtů nad jednotlivými položkami vektoru $\vec{d}$. Vypočtená hodnota je následně vydělena velikostí vektoru $\vec{d}$.

U výše popsaných *hashovacích* funkcí chci zkoumat, jak velký vliv má redukce dimenzionality na rychlost výpočtu a na přesnost řešení.



Obrázek 5.1: Pro lepší názornost je obrázek zjednodušen a objasňuje problém zredukovaný do 2D prostoru, místo původního 30D. Na obrázku lze vidět způsob generování nového souřadného systému. Generování spočívá v přičtení náhodného čísla, vygenerovaného z rozmezí souřadnicového systému dané *hashovací* funkce, k původnímu počátku souřadnicového systému P_0. Obrázek dále objasňuje pojem koše a jeho velikost.

5.2 Vstupy a výstupy

Návrh aplikace byl vytvořen v souladu s vstupně výstupním rozhraním programu *CAVER* [28], tudíž následná integrace mnou vytvořeného algoritmu do aplikačního rozhraní programu, by měla být bez problému.

V souladu s rozhraním použitým v programu, je vstupem algoritmu textový soubor, obsahující sekvenční číslo objektu, resp. tunelu, a jeho souřadnice. Výstupem algoritmu jsou následující textové soubory:

- Soubor obsahující objekty, které byly na vstupu algoritmu a objekty vzniklé v průběhu procesu shlukování. Soubor tedy obsahuje výčet všech objektů, jež se v době výpočtu nacházely v systému. Soubor dodržuje stejný formát jako vstupní soubor, tj. obsahuje vždy sekvenční číslo objektu, resp. tunelu, a jeho souřadnice. Záznamy v souboru jsou seřazeny vzestupně dle sekvenčního čísla.

- Soubor s dendrogramem. Záznamy jsou ve formátu

$$sekv\_č\_obj \quad : \quad vzdálenost[Å] \quad sekv\_č\_předka1 \quad sekv\_č\_předka2$$

  kde *sekv_č_obj* značí sekvenční číslo objektu, resp. tunelu, *sekv_č_předka* značí sekvenční číslo předka objektu. *vzdálenost* je vzdálenost mezi oběma předky udána v jednotkách *angstromů*[4]. Objekt vzniklý v procesu shlukování má vždy dva předky. Souřadnice ke všem objektům jsou dohledatelné v prvním souboru. Záznamy jsou v souboru seřazeny vzestupně, tedy tak, jak v procesu shlukování docházelo k jejich vzniku.
  Soubor s dendrogramem lze vytvořit ve třech modifikacích, a to tak, že:

  - tiskne se posledních x shluků, které v procesu shlukování vznikly,
  - tiskne se pouze prvních x shluků, které v procesu shlukování vznikly,
  - tisknou se všechny shluky kromě posledních x shluků, které v procesu shlukování vznikly,

  kde x vyjadřuje počet shluků, který je parametrem těchto funkcí. První modifikace poskytuje výpis několika posledních shluků, které již lze považovat za důvěryhodné. Volba parametru x je závislá na aplikaci a aktuální konfiguraci algoritmu. Další dvě modifikace umožňují zachytit vývoj procesu shlukování a jsou vhodné pro vizualizaci.

- Soubor s výsledky *hashování*. Jedná se o mezivýsledek procesu shlukování a slouží k rozboru použité *hashovací funkce*. Záznamy v souboru vyjadřují vývoj procesu *hashování*. Po aplikaci *hashovací* funkce je zaznamenán stav, kolik vzniklo nových košů a jak jsou do nich data rozdělena.

5.3 Návrh datové struktury

Při návrhu datové struktury jsem vycházela z článku [11] rozebraného v kapitole 3.2.

Elementární entitou v rámci datové struktury jsou shluky. Shluky představují jednotlivé tunely a udržují se v seznamu elementů. Tento seznam obsahuje ty shluky, které byly vstupem algoritmu a ty, které vznikly v průběhu shlukování. Každý shluk udržuje seznam ukazatelů na uzly, ve kterých se nachází. Návrhu těchto shluků odpovídají třídy *Element* a *Tunnel*.

Na úrovni uzlů rozlišuji listové a nelistové uzly (rozdělující a spojující body). Listovým uzlem je takový uzel, který buď ukazuje pouze na jednoho uzlového potomka nebo na jeden shluk v seznamu elementů. Nemusí se tedy nutně jednat o uzly umístěné v poslední úrovni stromu. Jedná se o označení charakteru uzlu, nikoliv o jeho pozici v rámci stromu. Rozdělujícím uzlem je jakýkoliv uzel, který ukazuje na více jak jednoho uzlového potomka nebo na

[4] *Angstrom* je jednotkou délky využívané k vyjádření velikostí atomů, molekul, mikroskopických biologických struktur atd. *1å* odpovídá $0,1nm$.

více jak jeden shluk v seznamu elementů. Rozdělující bod, který se nachází nejníže v rámci stromu, se označuje jako spojující. Toto rozlišení uzlů má svůj význam ve stromových operacích, zejména při výběru kandidátních uzlů pro operaci shlukování. Každý uzel udržuje ukazatel na strom, ve kterém se nachází, a na jeho úroveň, tj. do seznamu rozdělujících bodů.

Každý strom se sestává z několika úrovní, které svým počtem odpovídají počtu *hashovacích* funkcí, vygenerovaných pro daný strom, zvýšenému o jedničku. Uzly v rámci tohoto stromu jsou mezi úrovněmi propojeny ukazateli, a to tak, že každý uzel, s výjimkou uzlů umístěných v poslední úrovni stromu, uchovává ukazatele na své potomky. Zároveň každý uzel, s výjimkou uzlu v první úrovni stromu, uchovává ukazatel na svého předchůdce. Uzly v poslední úrovni stromu, narozdíl od jiných uzlů, uchovávají ukazatele do struktury elementů na jednotlivé shluky. Strom představuje prefixovou strukturu. Prefix každého uzlu je tvořen cestou stromem směrem od kořenového uzlu až k danému uzlu. Tato cesta je tvořena výsledky *hashovacích* funkcí z každé úrovně stromu. Uvažujme tedy rodinu *hashovacích* funkcí H, ze které je do stromu vybráno n funkcí $\{h_1, h_2, \ldots, h_n\} \in H$. Uzly $u_1, u_2, \ldots, u_m$ v poslední úrovni stromu, kde m odpovídá počtu uzlů na této úrovni, jsou tedy reprezentovány štítky s výsledky jednotlivých *hashovacích* funkcí $g(u_i) = (h_1(u_i), h_2(u_i), \ldots, h_n(u_i))$, *kde* $i \leq m$. S rostoucí hloubkou stromu je rozlišení podobnosti shluků striktnější a posiluje se tedy *AND*-vlastnost algoritmu (viz obr. 6.2). Hierarchie stromů pak tvoří les a s rostoucí velikostí lesu se posiluje *OR*-vlastnost algoritmu.

Každý strom je doprovázen datovou strukturou – seznamem rozdělujících bodů. Tento seznam má stejný počet úrovní jako strom a v každé této úrovni uchovává ukazatele na všechny uzly v odpovídající úrovni stromu.

Návrh tříd realizujících tuto datovou strukturu je podrobně popsán níže v podkapitole 5.4.

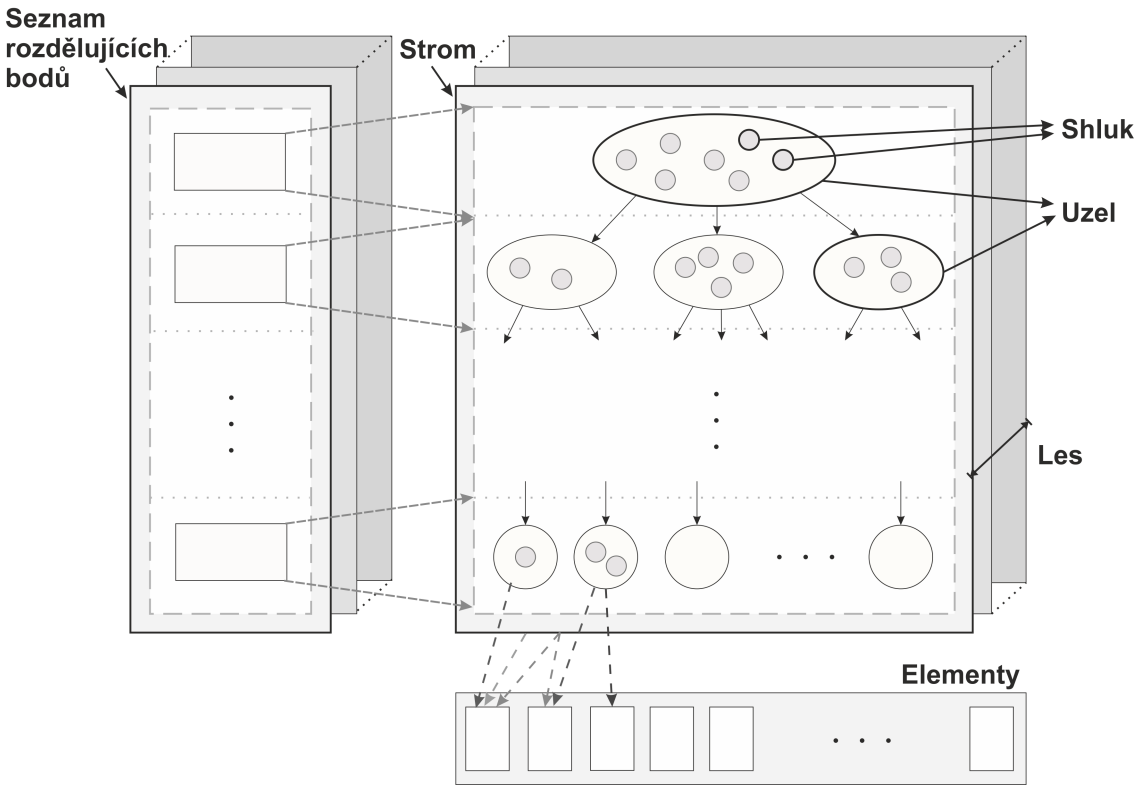
5.4 Návrh a popis tříd

V návrhu nastíním funkčnost a význam jednotlivých tříd. Konkrétní náhled na celkové rozhraní poskytuje příloha A.

Třídy Element a Tunnel Třída *Element* je abstraktní a generická. Parametrem šablony této třídy je typový parametr určující, s jakým typem objektu se bude pracovat. Třída tvoří rozhraní pro práci s objektem na nejnižší úrovni. Třída *Tunnel* implementuje rozhraní definované třídou *Element*, ze které dědí. Tato třída tedy tvoří konkrétní implementaci pro práci s 30-dimenzionálními tunely a je sestavena dle zadání.

Třídy Elements a Tunnels Třída *Elements* je abstraktní třídou a definuje rozhraní pro práci s objekty na vyšší úrovni. Seskupuje objekty třídy *Element* a pracuje s nimi jako s celkem. Implementuje datovou strukturu shodnou se strukturou elementů na obrázku 5.2. Třída *Tunnels* pak od této třídy dědí a implementuje všechny metody jejího rozhraní. Je tedy sestavena pro konkrétní aplikaci s tunely.

Třídy HashFamily a HashFunction *HashFamily* je abstraktní a generickou třídou. Parametrem šablony této třídy je datový typ. *HashFamily* ve svém rozhraní definuje ještě vnitřní abstraktní třídu *HashResult*. Vnitřní třída definuje rozhraní pro práci s výsledkem *hashovací* funkce. Třída *HashFunction* tvoří konkrétní implementaci a implementuje



Obrázek 5.2: Popis jednotlivých entit, které tvoří datovou strukturu *Twister Tries* a procesu *hashování*.

všechny metody nadřazené funkce. Její vnitřní třída je třída *BucketID*, která implementuje rozhraní definované vnitřní třídou *HashResult*.

Třída Node Třída *Node* definuje uzel stromu (viz obrázek 5.2), jeho typ a vlastnosti. Zároveň implementuje rozhraní pro práci s tímto uzlem. V průběhu *hashovací* fáze jsou jednotlivé výsledky *hashování* ukládány právě do těchto uzlů, což představuje uložení výsledku na hrany stromu mezi jednotlivé uzly (viz obrázek 5.3). Každý uzel, mimo jiné, obsahuje položku ukazatele na svého předchůdce, pokud není kořenovým uzlem, a seznam ukazatelů na své uzlové potomky, pokud nějaké má. Každý uzel, který nemá žádné uzlové potomky, obsahuje pole ukazatelů na shluky, tj. na instance třídy *Element*, ke kterým přistupuje skrze rozhraní třídy *Elements*. Podle toho, kolik referencí na uzlové listy nebo na shluky uzel obsahuje, se rozlišuje, zda se jedná o listový uzel nebo o rozdělující bod. Tato informace je rovněž uložena jako atribut třídy *Node*.

Každý uzel dále obsahuje ukazatel na strom, ve kterém se vyskytuje a na konkrétní úroveň tohoto stromu. Toto má opět velký význam při stromových operacích jako je odstranění objektu či operace shlukování.

Jelikož při operaci shlukování dochází ke slučování objektů buď na úrovni elementů nebo uzlů, je nutné uchovávat informaci o tom, kolika elementy byl shluk na počátku před touto operací shlukování tvořen. Tato informace slouží k váhování dvou shluků při jejich shlukování a výpočtu jejich spojení. Tuto informaci v sobě nese každý uzel a pokud je rušen v rámci operace shlukování, předá ji nově vzniklému uzlu.

Třídy Trie a TrieImpl Abstraktní třída *Trie* definuje atributy a metody rozhraní. V rámci této třídy se definuje struktura zvaná seznam rozdělujících bodů, viz obr. 5.2. Tato struktura sice obsahuje i listové uzly, ale ty se v procesu výběru kandidáta přehlížejí. Třída *TrieImpl* dědí z *Trie* a implementuje všechny její metody.

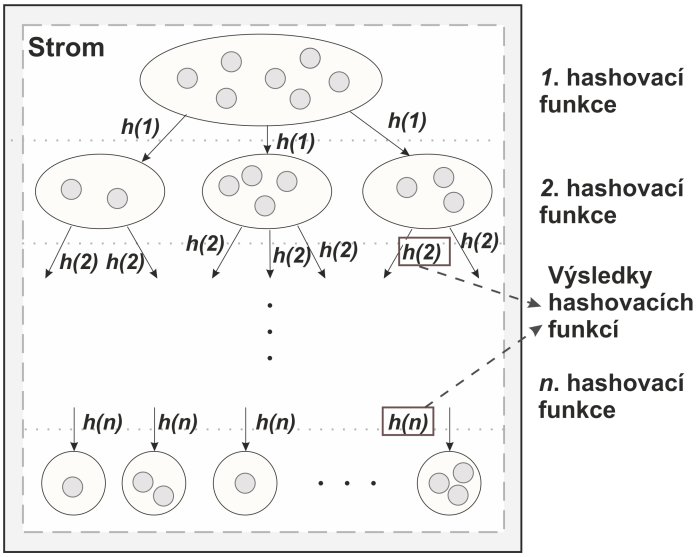
Třída Twister Tries Tato třída zastřešuje celý algoritmus a implementuje celou jeho logiku. Obsahuje čtyři stěžejní funkce, které vycházejí z článku [11], a to výběr kandidátů pro shlukování (`select()`), shlukování (`merge()`), odstranění objektu (`remove()`) a vložení objektu do stromů a jeho *zahashování* (`insert()`).

Třídy ProcessInput a ProcessOutput Třída *ProcessInput* slouží k rozparsování vstupního souboru a vytvoření bázové skupiny elementů (instance třídy *Elements*), která slouží jako vstupní parametr pro třídu *Twister Tries*. Třída *ProcessOutput* naopak využívá bázových elementů a elementů nově vzniklých v procesu shlukování k vygenerování výstupních souborů, viz podkapitola 5.2.

5.5 Návrh procesu shlukování

Algoritmus *Twister Tries* dle článku [11] pracuje ve dvou fázích:

1. *Hashovací* fáze. Na počátku této fáze je vytvořen les a do první úrovně (tj. kořene) každého stromu jsou vloženy bázové elementy tvořící shluky o velikosti 1. Na tyto shluky jsou poté postupně aplikovány *hashovací* funkce. Tento proces je naznačen na obr. 5.2 a 5.3. Na obrázcích je jasně vidět, že se původní množina shluků rozděluje v dalších úrovních stromu do uzlů. Toto odpovídá rozdělování entit do košů podle výsledků *hashovací* funkce *h(i)*, kde *i* odpovídá úrovni stromu. Shluky zakreslené



Obrázek 5.3: Způsob *hashování* jednotlivých uzlů do stromů v algoritmu *Twister Tries*. Výsledky *hashování* se ukládají na hrany mezi uzly.

v uzlech však slouží pouze k ilustraci procesu *hashování*. Samotný uzel neuchovává ukazatele na tyto shluky, s výjimkou uzlů v poslední úrovni stromu. Z každého uzlu je však možné zjistit, jaké shluky obsahoval v době, kdy na něj byla aplikována *hashovací* funkce, a to prostřednictvím ukazatelů na jeho potomky.

2. *Twisting* fáze. Tato fáze představuje proces shlukování. Shlukování probíhá od spoda nahoru. V každé iteraci jsou vybrány dva uzly, které se spojí, tj. je vypočten váhovaný průměr ze souřadnic obou shluků. Takto vzniklý nový shluk je poté *zahashován* zvlášť v každém stromě jeho *hashovacími* funkcemi a původní shluky jsou ze všech stromů odstraněny. Proces shlukování končí v okamžiku, kdy v systému existuje poslední shluk.

Kapitola 6

Implementace

Při vývoji aplikace jsem se rozhodla pracovat podle metodiky testy řízeného vývoje (z angl. *test driven development – TDD*). Vývoj spočívá v tom, že pro každý návrh třídy (viz kapitola 5.4) nejprve napíši jednotkové testy, u kterých specifikuji očekávanou funkcionalitu jednotlivých metod třídy ve formě vstupů a očekávaných výstupů. Až poté implementuji samotnou třídu a její metody. Danou třídu pak testuji svými automatickými testy pokaždé, když dojde k jakékoliv změně v kódu. Tímto přístupem se snažím zabránit situaci, ve které by se mohla zanést chyba do kódu a k jejímu odhalení by došlo až v průběhu závěrečného testování nebo při používání samotné aplikace, což obecně může být kritické zejména pro aplikace, které zasahují do bezpečnosti lidí.

V průběhu vývoje a implementace algoritmu jsem používala distribuovaný systém pro správu verzí, *GIT* [4] [9]. Pro implementaci návrhů dílčích tříd, testů a různých oprav jsem vytvářela příslušná témata (z angl. *issues*), ve kterých jsem podrobně popsala daný problém a způsob, jak jej řešit a implementovat. Jednotlivé záznamy (z angl. *commits*) pak byly vždy provázány s korespondujícím tématem a ve svém popisu se odkazovaly na jeho dílčí část a popisovaly, v jakém stavu řešení se tento dílčí problém aktuáně nachází.

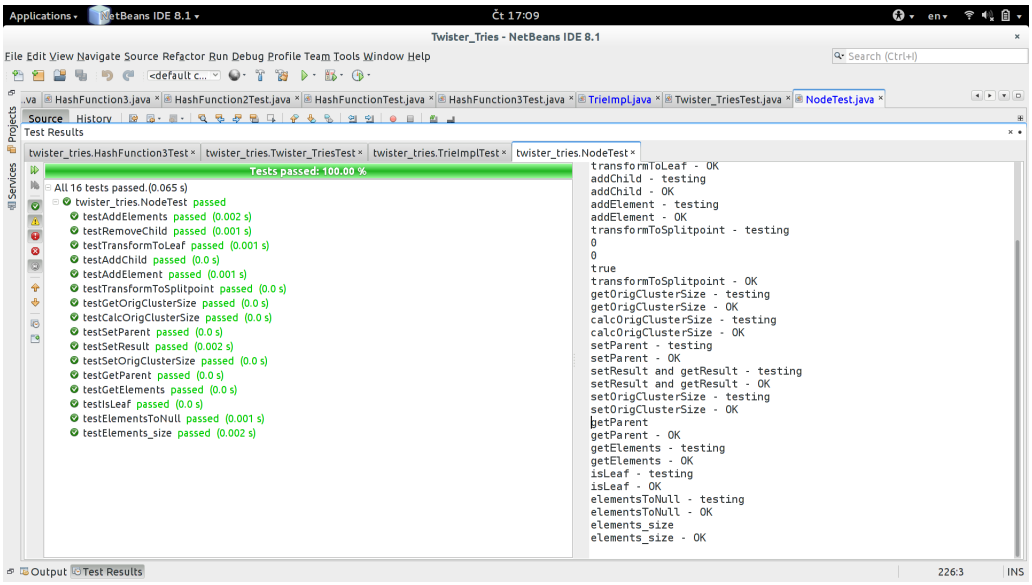
Dle návrhu (kapitola 5) jsem aplikaci vytvořila v jazyce *Java*. Tudíž jsem pro implementaci jednotkových testů použila *framework*, psaný v jazyce *Java*, *JUnit* [3] (viz obr. 6.1). Aplikaci jsem vytvořila jako projekt do vývojového prostředí *NetBeans IDE* [4].

Algoritmus *Twister Tries* jsem implementovala dle návrhu v kapitole 5. Pro detailní popis tříd a metod jsem vytvořila dokumentaci pomocí nástroje *Javadoc*. Hlavní logiku algoritmu, implementovanou ve třídě *Twister Tries*, popisuje obrázek 6.3. Část algoritmu prováděnou uvnitř modrého obdélníku vyznačeného v obrázku, se označuje jako *twisting* fáze.

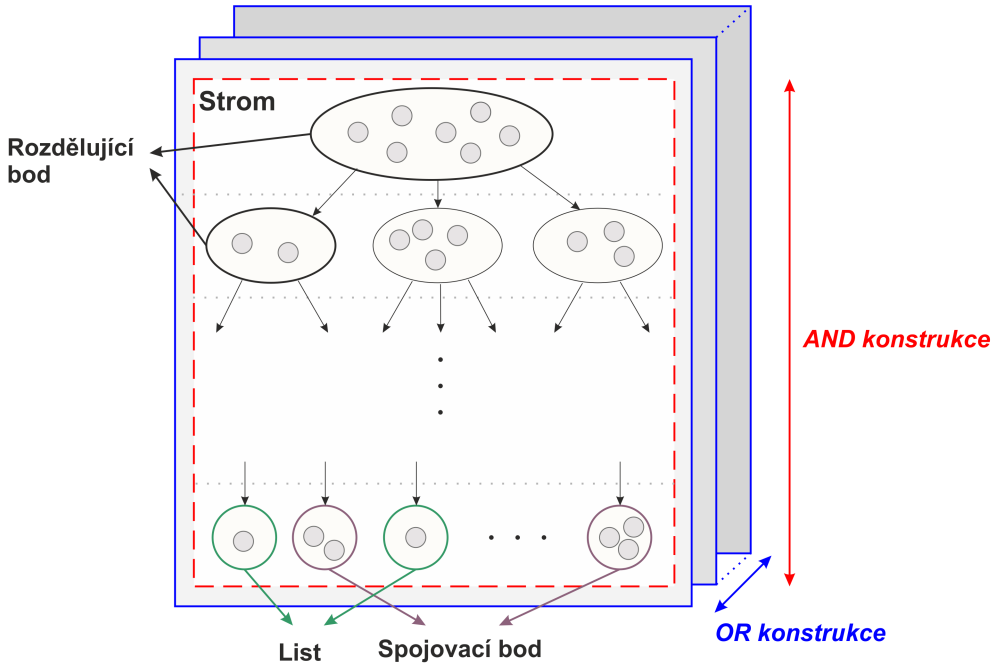
Vstupními parametry algoritmu jsou:

- Počet stromů. S rostoucím počtem stromů se posiluje *OR*-vlastnost algoritmu (viz obr. 6.2 a odstavec 3.2).
- Výška stromu. S rostoucí výškou stromu se posiluje *AND*-vlastnost algoritmu (viz obr. 6.2 a odstavec 3.2).
- Model *hashovací* funkce. Model se použije pro vytvoření kopií a rovnoměrnému rozdělení funkcí mezi jednotlivé stromy. Počet kopií je dán součinem výšky stromu a jejich počtem. V každé kopii se vygeneruje nový offset (viz podkapitola 5.1 a obrázek 5.1).

[4] Provozovaný na serveru `bitbucket.org`.



Obrázek 6.1: Ukázka použití jednotkového testu, vytvořeného ve vývojovém prostředí *Netbeans IDE* [4], k ověření správnosti implementace třídy *Node*. V levé části okna se zobrazuje, kolik procent testů proběhlo správně. V pravé části okna se zobrazují kontrolní výpisy z testovaných metod jednotkového testu.



Obrázek 6.2: *AND-OR* vlastnosti stromu a rozlišení typů uzlů v datové struktuře *Twister Tries*.

- Bázové elementy. Počáteční elementy, se kterými algoritmus začíná pracovat. Elementy jsou získány ze vstupního souboru, viz podkapitola 5.2.

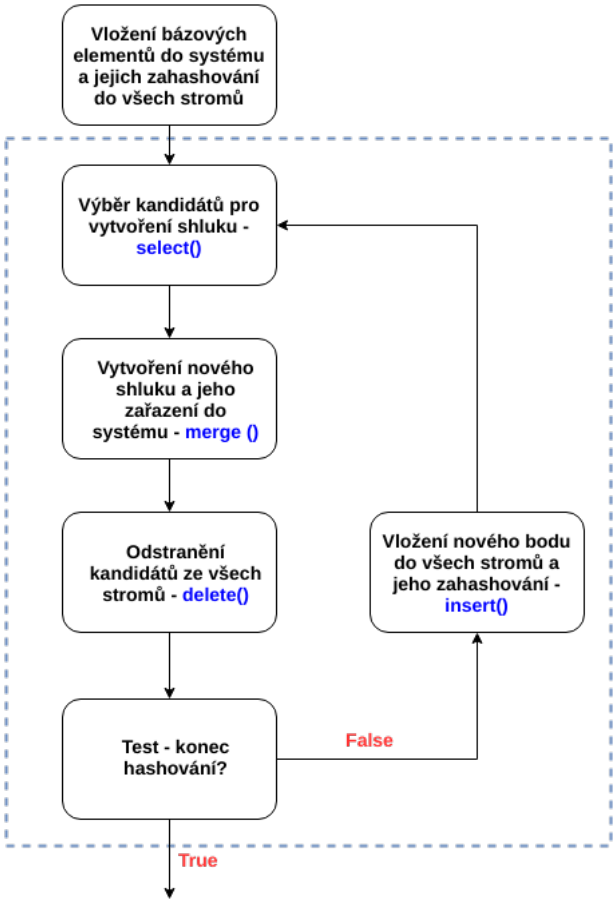
Algoritmus *Twister Tries* dle diagramu pracuje následovně:

- Bázové elementy jsou metodou `hashAllLevels()` *zahashovány* nezávisle v každém stromu. Po *zahashování* se provede ještě tzv. *zatřesení* stromem (metoda `shakeIt()`). Tato operace není řešena v popisu algoritmu dle článku [11]. Operace ošetřuje situace, kdy již na začátku došlo k *zahashování* bodů do určitého koše a další aplikace *hashování* by již tento stav nezměnily. Vytvořily se tedy větve, kdy na sebe v sérii navazuje několik listových uzlů. Operace *zatřesení* provede to, že tyto větve upraví a zkrátí bez újmy na správné funkčnosti (upraví ukazatele na potomky a rodičovské uzly). K této situaci většinou dochází v situacích, kdy je velikost koše malá, nebo je použit první typ *hashovací funkce* (viz funkce 5.1), u kterého nedochází k redukci dimenzionality a do stejného koše spadnou opravdu jen velmi blízké body.

- V cyklu:

  1. Výběr vhodných kandidátů pro operaci shlukování metodou `select()`. Výběr probíhá tak, že se postupně prohledávají nejspodnější úrovně všech stromů a hledá se nejbližší spojovací bod. Teprve až na dané úrovni v žádném stromu nejsou žádní vhodní kandidáti pro shlukování, přejde se na vyšší úroveň. Touto metodou jsou vybrány vždy dva elementy nebo dva uzly, na které ukazoval vybraný spojovací bod. Pokud je uzlových kandidátů více, vyberou se náhodně dva. Pokud je nalezeno více kandidátů mezi elementy, vyberou se první dva v pořadí.
  2. V závislosti na tom, jaký typ objektu byl vybrán metodou `select()`, volá se příslušná modifikace metody `merge()`, buď pro dva elementy nebo dva uzly. Metoda vytvoří nový uzel, do kterého uloží souřadnice nově vypočteného shluku dvou vybraných kandidátů. Zároveň vloží nově vypočtený shluk do množiny bázových elementů. V mé implementaci jde v operaci `merge()` o výpočet těžiště kandidátů. Kromě samotného výpočtu těžiště, metoda ještě zabezpečí správné nastavení všech atributů nového uzlu.
  3. Nyní je nutné tyto dva kandidátní objekty odstranit ze všech stromů, k čemuž slouží metoda `delete()`. Podle typu mazaného objektu se volá příslušná modifikace této metody. Pokud je typem objektu mazání element (shluk), dojde k vymazání reference na tento element ze všech listových uzlů, které na něj ukazovaly a k odstranění všech referencí na listové uzly uvnitř elementu. Element však není zrušen a v systému zůstává dál uložen mezi bázovými elementy. Pokud byl typem objektu uzel, musí tento uzel předat referenci na elementy, na které ukazuje, svému předchůdci. Jeho předchůdce poté ve svém atributu smaže referenci na mazaný uzel a uzel smaže referenci na svého předchůdce. Uzel je poté odstraněn ze stromu, konkrétně z příslušné úrovně seznamu rozdějících bodů, ve kterém byl umístěn. Zároveň je smazán i odkaz na daný strom a jeho úroveň z atributů uzlu. Takový uzel je pak v nejbližší době dealokován pomocí *Garbage Collectoru*, což automaticky obstarává *runtime* jazyku *Java*.
  4. Nyní dojde k otestování ukončující podmínky, tj. ve všech stromech se otestuje, zda první úroveň stromu, resp. seznamu spojujících bodů, obsahuje pouze jeden uzel, který nemá žádné potomky a neukazuje ani na žádný element. Pokud je tato podmínka splněna, algoritmus končí a vygenerují se výstupní soubory. Pokud však podmínka splněna není, pokračuje se dalším bodem.

5. Metoda `merge()` volaná v bodu 3 zajistila, že nově vzniklý element již v systému zařazený je, tj. element byl přidán mezi bázové elementy. Nyní je však nutné uzel vzniklý touto metodou vložit do všech stromů a *zahashovat* jej příslušnými *hashovacími* funkcemi. K tomu slouží metoda `insert()`, která uvnitř sebe volá metodu `hashNode()`, která zabepečuje právě ono *hashování* v rámci stromu, následně po tomto kroku se volá metoda `shakeIt()`, která zatřese daným stromem. Algoritmus pokračuje bodem 1.



Obrázek 6.3: Zjednodušený vývojový diagram algoritmu *Twister Tries*.

Kapitola 7

Testování

Správnost implementace jsem ověřila pomocí jednotkových testů. Tento způsob testování jsem již popsala v kapitole 6. Pro každou třídu jsem vytvořila testovací soubor se sadou různých testů, kterými implementace musela projít. Po každém zásahu do kódu jsem spustila sadu všech testů, čímž jsem zabezpečila správnost implementace a zabránila zanesení chyby do implementace.

Schopnost algoritmu shlukovat data dle předpokladů jsem otestovala pomocí referenční testovací sady 145 tunelů. Tato sada je získána z 50 snímků z 10ns simulace molekulární dynamiky enzymu haloalkan dehalogenáza *DhaA* [28]. Získala jsem ji pro účely závěrečného testování a jedná se o výsledky shlukování pomocí programu *CAVER* [28]. Tato testovací sada je oproti plné testovací sadě poměrně malá. Plná sada obsahuje 10-100 tisíc tunelů [28]). U algoritmu *Twister Tries* se však předpokládá, že vzhledem k jeho lineárním vlastnostem, bude aplikovatelný se stejnými parametry i na velkou testovací sadu. Právě u velké testovací sady se předpokládá akcelerace výpočtu, která na menší sadě nemusí být tolik znatelná.

Metodiku vyhodnocování získaných výstupů s referenčním řešením popisuji v podkapitole 7.1. Poznatky týkající se *hashovací* fáze popisuji v podkapitole 7.2 a poznatky dosažené z testování výsledků shlukování a jejich vizualizaci popisuji v podkapitole 7.3.

7.1 Metodika vyhodnocení výstupů

Výsledky obou algoritmů jsem porovnala pomocí *Euklidovské* metriky [29], která se mi pro tento problém zdála jak nejvíce vhodná. Vybrala jsem ji z toho důvodu, že porovnávám body v prostoru a výsledky získané touto metrikou jsou dobře pochopitelné a porovnatelné. Výpočet jsem provedla pomocí vztahu [29]:

$$d([x_1, x_2, \ldots, x_n], [y_1, y_2, \ldots, y_n]) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

Kde x, y jsou dva porovnávané tunely, n značí počet dimenzí tunelu a d je vypočtená *Euklidovská* vzdálenost mezi tunely.

Stejně jako jiné vzdálenostní metriky, musí i tato metrika splňovat axiomy:

- Axiom nezápornosti. *Euklidovská* vzdálenost vypočtená mezi dvěma body nesmí být záporná, $d(x, y) \geq 0$.
- Axiom totožnosti. Body jsou identické, pokud je mezi nimi *Euklidovská* vzdálenost nulová, $d(x, y) = 0 \Leftrightarrow x = y$.

- Axiom symetrie. Musí tedy platit $d(x, y) = d(y, x)$.
- Musí splňovat trojúhelníkovou nerovnost. Mějme tři body x, y, z v prostoru, pak musí platit $d(x, z) \leq d(x, y) + d(y, z)$.

Takto získané výsledky jsem poté vyhodnotila a vytvořila grafické výstupy v podobě grafů. Další porovnání výstupů jsem provedla na základě vizualizace, kterou rovněž provádí i program *CAVER* [28], a to následovně:

- Tunel je vyjádřen jako směrový vektor. Pro každou dimenzi x, y, z se provede výpočet průměrné hodnoty ze všech desíti bodů, které tvoří tunel. Tato průměrná hodnota je následně odečtena od první souřadnice tunelu.
- Převod kartézských souřadnic směrového vektoru na sférické pomocí vztahů:

$$r = \sqrt{x^2 + y^2 + z^2}$$

$$\psi = \text{atan2}\,(y,\, x)$$

$$\theta = \arccos\left(\frac{z}{r}\right)$$

- Využití balíčku *ggplot2* [35] pro programovací jazyk *R* pro vygenerování grafu, kde na ose x je vynesen úhel ψ (úhel odklonu polohového vektoru od osy x) a na ose y úhel θ (úhel odklonu polohového vektoru od osy z).

7.2 Výběr hashovací funkce a jejích parametrů

Při testování jsem se rovněž zaměřila na *hashovací* fázi algoritmu a jednotlivé *hashovací* funkce. Testovala jsem vliv výběru *hashovací* funkce a jejích parametrů na schopnost algoritmu řadit shluky do jednotivých košů. Dále jsem sledovala vliv *hashovacích* funkcí, velikostí stromu a počtu stromů na rychlost celého procesu shlukování. Všechna svá měření jsem provedla na 5000 běžích programu a sledovala jsem změny vypočtených středních hodnot sledovaných veličin. V grafech uvedených níže neuvádím u těchto hodnot žádné jiné statistiky, protože cílem je ukázat pouze trendy vývoje. Nejde o přesné měření hodnot. Měření doby běhu programu jsem provedla tak, že vždy na začátku algoritmu jsem zaznamenala aktuální čas systému v milisekundách. Na konci algoritmu jsem opět zaznamenala aktuální čas a rozdílem obou hodnot jsem dobu jednoho běhu programu – mezičas. Stejným způsobem jsem měření provedla pro všech 5000 běhů a jednotlivé mezičasy jsem sečetla. Takové měření však může být zatíženo určitou nejistotou, jelikož nijak nefiltruji zátěž procesoru jinými činnostmi. Pokud je tedy v průběhu měření procesor zaměstnán i jinou činností, může odebírat prostředky měřenému algoritmu a zkreslit tak čas jeho běhu.

Provedla jsem tato dílčí měření:

- Závislost délky běhu programu na počtu stromů. Tuto závislost vyjadřuje graf na obr. 7.1 vpravo. Z grafu je patrno, že výpočetně nejnáročnější se stává první typ *hashovací* funkce 5.1. Důvodem je velký počet dimenzí koše a s tím spojené i náročnější testování na rovnost označení koše (výsledek *hashování*). Dále můžeme vidět, že trend funkcí je lineární. Křivky v grafu se však nevyvíjí úplně lineárně. Tato skutečnost je způsobena zřejmě nejistotou měření časové délky běhu programu (viz výše).

- Závislost délky běhu programu na velikosti stromů. Tuto závislot vyjadřuje graf na obr. 7.1 vlevo. Z grafu je opět vidět, že výpočetně nejvíc náročným je první typ *hashovací* funkce (viz definice 5.1) pro stejné důvody uvedené výše. Trend křivek je opět lineární.

- Vliv použité *hashovací* funkce na schopnost algoritmu rozdělovat shluky do košů. Toto pozorování pro jednotlivé *hashovací* funkce vyjadřují grafy na obrázcích 7.2, 7.3 a 7.4. Z grafů na obr. 7.2 je patrno, že již od prvního procesu *hashování* v systému vzniká velké množství košů. Lze říci, že po 2-3 operacích *hashování* jsou data rozdělena do košů po jednom shluku. Vzniká tak velmi jemné rozdělení a nedochází tak úplně k naplnění myšlenky použití košů, tj. rozdělení bodů mezi koše na základě *Euklidovy* vzdálenostní metriky [29]. Tuto skutečnost ovlivňuje nastavení parametrů *hashovací* funkce, ale hlavní vliv na tuto skutečnost má vysoká dimenzionalita použitá v této funkci. Řešením je použít funkci s dostatečnou velikostí koše. Z grafu vlevo je však patrno, že proces *hashování* je i tak velmi jemný a po aplikaci dalšího *hashování* v systému rychle vzniká stejný počet košů, jako je shluků.
  Z grafů na obr. 7.3 vidíme, že situace rozdělovaní shluků do košů se vyvíjí mnohem více podle našich teoretických předpokladů (viz podkapitola 3.2). V procesu *hashování* vznikají koše mnohem pozvolněji. Pro některé parametry *hashovací* funkce, definované v 5.2, není dokonce ani po šesté operaci *hashování* v systému stejný počet košů jako shluků. Pravý graf vyjadřuje počet shluků, které spadají na jeden koš, po jednotlivých operacích *shlukování*.
  Grafy na obr. 7.4 charakterizují třetí *hashovací* funkci ((viz definice 5.3)). Tato funkce rozděluje shluky do košů velmi hrubě a na jeden koš spadá mnohem více shluků než u předchozích případů *hashovacích* funkcí. Velkou výhodou této funkce je její rychlost, viz obr. 7.1.
  U všech třech *hashovacích* funkcí je patrné, že velikost *offsetu* nemá žádný velký prokazatelný vliv na způsob ani rychlost *hashování*. Velikost koše je naopak důležitým atributem, který je nutné experimentálně získat a vhodně určit pro každou aplikaci tohoto algoritmu.

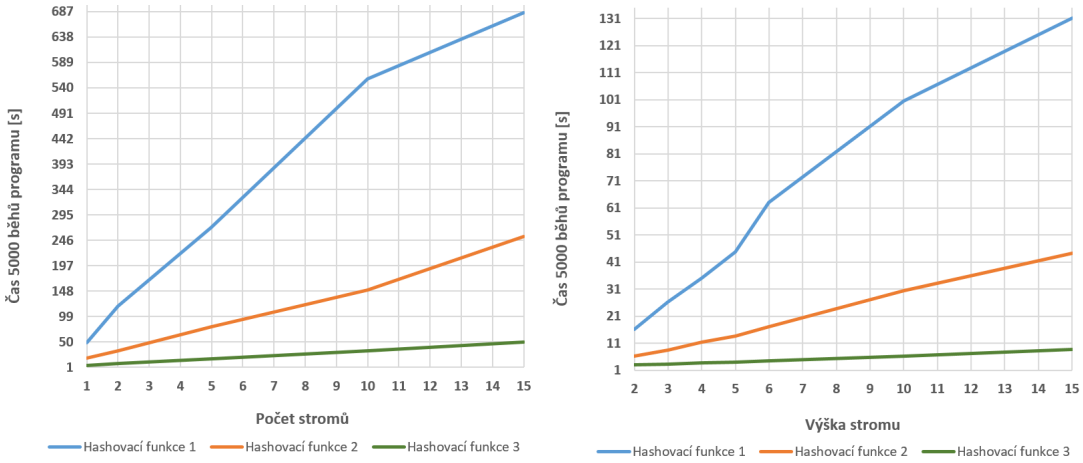
Časová náročnost referenčního algoritmu *average-linkage* je podrobně popsána v článku [28]. Referenční testovací sada, kterou jsem získala pro účely testování získaných výsledků, byla však vygenerována na jiném počítači, než na kterém jsem své testování prováděla. Tudíž nemohu dané algoritmy dostatečně přesně porovnat z hlediska časové náročnosti. Proto jsem se zaměřila na testování jevů, tj. počet a výška stromů a parametry *hashovací* funkce, které na časovou náročnost mají vliv. Testování bylo provedeno na počítači vybaveném procesorem *Intel i5-6200U 2.30GHz*[6] s pamětí typu *RAM* o velikosti *8GB*.

7.3 Výsledky shlukování

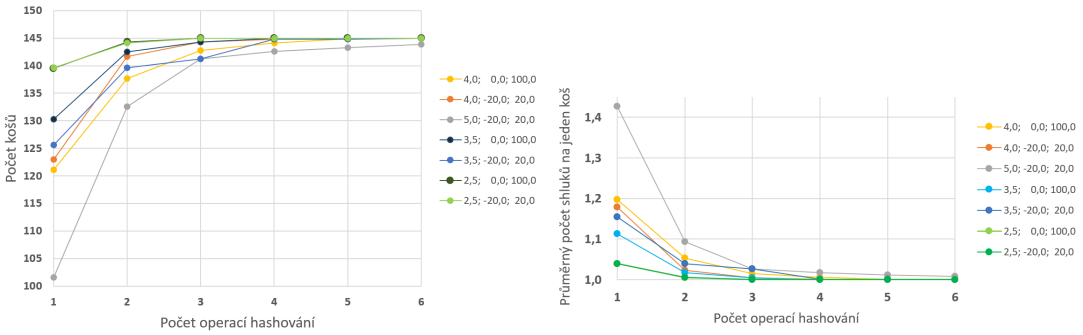
Problém porovnání výsledků řeším tak, že 30dimenzionální vektor souřadnic tunelu rozdělím na deset bodů po 3 dimenzích. Na body ze dvou tunelů, získaného algoritmem *Twister Tries* a referenčního, aplikuji *Euklidovskou* metriku následujícím způsobem:

- Z procesu shlukování obou algoritmů jsem vždy vybrala posledních pět tunelů, které v procesu vznikly.

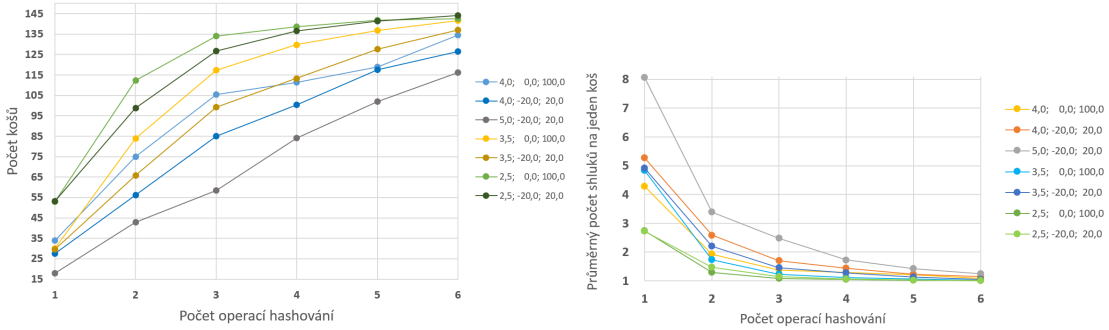
[6] http://ark.intel.com/products/88193/Intel-Core-i5-6200U-Processor-3M-Cache-up-to-2_80-GHz



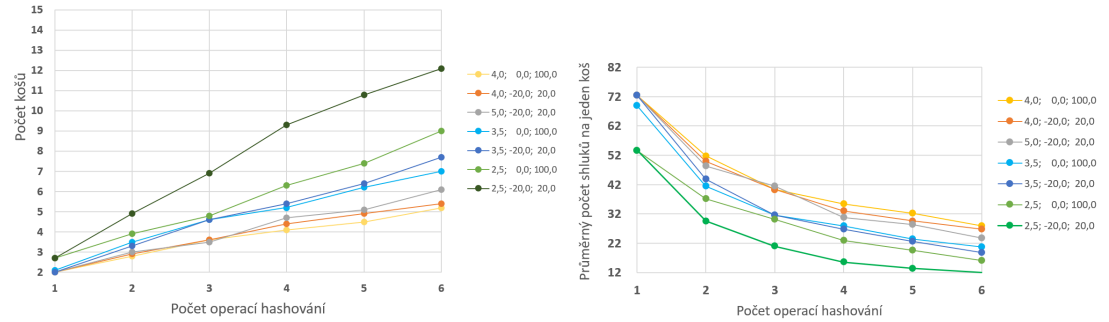
Obrázek 7.1: Graf vpravo vyjadřuje závislost délky běhu programu na počtu stromů. Graf vlevo vyjadřuje závislost délky běhu programu na velikosti jednotlivých stromů. Je vidět, že výška stromu nemá tak dramatický vliv na rychlost algoritmu jako počet stromů. Jestliže výška stromu vzroste $2\times$, časová délka programu se u prvního typu *hashovací* funkce prodlouží $2,2\times$, u druhého typu $2\times$ a u třetí typu *hashovací* funkce $1,4\times$. Pokud počet stromů zdvojnásobíme, časová délka běhu programu se u prvního typu *hashovací* funkce prodlouží $2,3\times$ a u zbylých dvou typů přibližně $1,8\times$.



Obrázek 7.2: Grafy vyjadřují vliv volby parametrů první *hashovací* funkce na schopnost algoritmu rozdělovat shluky do košů. Graf vlevo ukazuje, kolik košů vzniklo po každé operaci *hashování*. Graf vpravo ukazuje, kolik shluků průměrně spadlo do stejného koše. Legenda vpravo u každého grafu charakterizuje testované parametry dané *hashovací* funkce ve tvaru *velikost koše, minimální hodnota offsetu, maximální hodnota offsetu*.



Obrázek 7.3: Grafy vyjadřují vliv volby parametrů druhé *hashovací* funkce na schopnost algoritmu rozdělovat shluky do košů. Graf vlevo ukazuje, kolik košů vzniklo po každé operaci *hashování*. Graf vpravo ukazuje, kolik shluků průměrně spadlo do stejného koše. Legenda vpravo u každého grafu charakterizuje testované parametry dané *hashovací* funkce ve tvaru *velikost koše, minimální hodnota offsetu, maximální hodnota offsetu*.



Obrázek 7.4: Grafy vyjadřují vliv volby parametrů třetí *hashovací* funkce na schopnost algoritmu rozdělovat shluky do košů. Graf vlevo ukazuje, kolik košů vzniklo po každé operaci *hashování*. Graf vpravo ukazuje, kolik shluků průměrně spadlo do stejného koše. Legenda vpravo u každého grafu charakterizuje testované parametry dané *hashovací* funkce ve tvaru *velikost koše, minimální hodnota offsetu, maximální hodnota offsetu*.

- Vytvořila jsem kontingenční tabulku a na všechny dvojice tunelů jsem aplikovala *Euklidovskou* metriku. V tabulce jsem sledovala, jak jsou od sebe jednotlivé dvojice vzdáleny.

Pro rychlejší a srozumitelnější orientaci v takto vytvořené tabulce jsem vytvořila zjednodušený ukazatel vzdálenostní metriky. Pro každé dva tunely je vytvořen vektor obsahující deset hodnot, které odpovídají vypočtené *Euklidovské* metrice mezi každými dvěma 3D body. Jako rychlý ukazatel jsem použila střední hodnotu metriky vypočtenou z tohoto vektoru. V tabulce jsem se vždy zaměřila na minimální a maximální hodnotu ukazatele. Takové měření jsem provedla pro každý typ *hashovací* funkce, u které jsem vytvořila přibližně 50 modifikací parametrů. Došla jsem k následujícím závěrům:

- Při porovnávání výsledků shlukování na poslední úrovni byly od sebe dvě trajektorie tunelů vždy poměrně hodně vzdálené. Na této úrovni již zřejmě dochází k velké generalizaci a shluk není dobře přepočítán. Při porovnávání trajektorií tunelů na nižších úrovních bylo dosaženo mnohem lepších výsledků.

- Velikost koše je významným parametrem *hashovací* funkce, který má vliv na kvalitu shlukovaných dat. S volbou tohoto parametru souvisí parametry vytvoření lesu pro *Twister Tries*, tj. počet stromů a jejich výška. Změna offsetu *hashovací* funkce měla menší vliv pouze u třetího typu *hashovací* funkce (viz definice 7.4). Změna offsetu tak, aby těsně obaloval rozsah vstupních hodnot tunelů, někdy pomohla ke zvýšení přesnosti nalezených tunelů. Tato změna však nebyla nijak výrazná. U *hashovací* funkce typu 1 (viz definice 7.2) se však změny offsetu na kvalitu získaných dat neprojevily vůbec.

- U jednotlivých *hashovacích* funkcí se mi podařilo dosáhnout nejlepších výsledků při konfiguraci *hashovací* funkce a lesu dle tabulky 7.1.

Tabulka 7.1: Optimální konfigurace paramaterů *hashovací* funkce a parametrů stromu.

	Typ *hashovací* funkce		
	1	2	3
Velikost koše [*Å*]	$4,0 - 5,0$	$3,0 - 5,0$	$2,0 - 2,5$
Rozsah offsetu	$0 - 100$	$0 - 100, -20 - 20$	$0 - 100, -20 - 20$
Počet stromů	$5 - 15$	$5 - 15$	$15 - 25$
Výška stromu	5	$5 - 8$	$10 - 15$

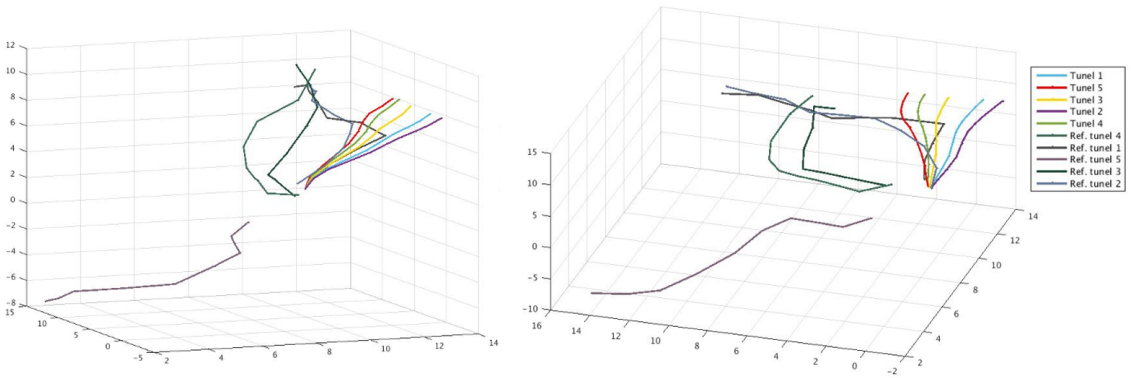
Je třeba podotknout, že vypočtené výsledky jsou datově závislé na předchozích hodnotách, tudíž každý běh aplikace může poskytovat trochu jiné výsledky. Avšak výše uvedené konfigurace poskytovaly stabilní výsledky, kde nejnižší střední hodnota *Euklidovské* vzdálenosti byla:

- $4,2$ *Å* pro první typ *hashovací* funkce.
- $2,8$ *Å* pro druhý typ *hashovací* funkce.
- $3,7$ *Å* pro třetí typ *hashovací* funkce.

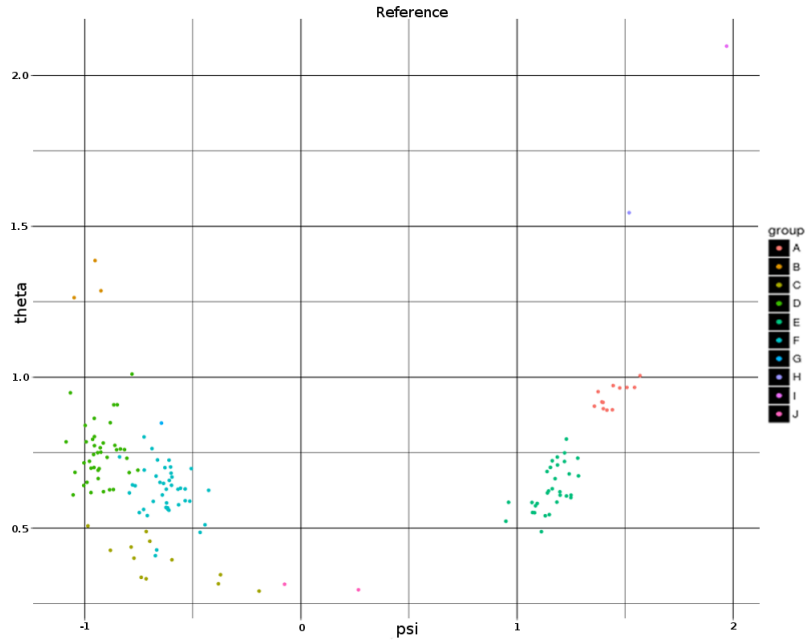
Maximální střední hodnota *Euklidovské* vzdálenosti byla pro všechny *hashovací* funkce $14,0 - 15,0\,\mathring{A}$. Avšak těchto hodnot dosahovala metrika pouze při porovnání trajektorií tunelů, které byly vygenerovány jako poslední v obou algoritmech (kořen dendrogramu). Z hlediska nalezení nejpravděpodobnějšího výskytu tunelu se jako nejslibnější jeví *hashovací* funkce typu 2 a 3. Při jejich použití bylo dosaženo výsledků, které jsou v toleranci s modelovým konceptem tunelu uvedeným v článku [28]. To ovšem neznamená, že všechny body výsledného tunelu musí nutné spadat do obalové plochy referenčního řešení [28] (viz obr. 7.5, 7.6 a 7.7).

Bohužel, podobných výsledků lze dosáhnout i s jinými konfiguracemi. Tudíž nasbírané výsledky nejsou zcela prokazatelné. Předpokládanou hypotézou této práce bylo, že i přibližným shlukovacím algoritmem budeme schopni dosáhnout stejných výsledků jako s exaktním algoritmem. Tuto hypotézu se však nepovedlo zcela potvrdit. Je však možné, že testovací sada pouze 145 tunelů byla příliš malá na to, aby se změny dostatečně projevily. Větší testovací sadu jsem však neměla k dispozici, tudíž své tvrzení nemohu potvrdit ani vyvrátit. Toto tvrzení může být ověřeno až v případně další etapě testování, kdy bude k dispozici datová sada dostatečné velikosti. Další možností vzniku nepředpokládaných výsledků může být nevhodně koncipovaná *hashovací* funkce.

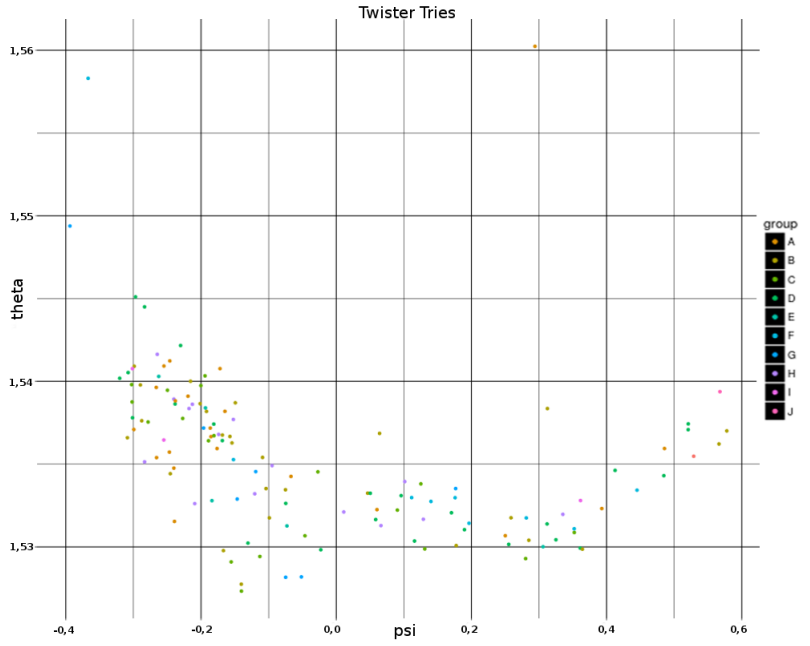
Obrázek 7.5 znázorňuje vztah mezi posledními pěti vzniklými tunely obou algoritmů promítnutý v 3D prostoru. Z grafů lze jasně vidět, že algoritmus *Twister Tries* došel k opravdu odlišným výsledkům. Je vidět, že některé trajektorie se vytvářely podobně jako referenční, avšak od určitého okamžiku došlo k velkému vychýlení. Obrázek 7.6 zachycuje referenční řešení získané algoritmem *average-linkage*. Obrázek 7.7 zachycuje graf, který odpovídá řešení získaného algoritmem *Twister Tries*. Grafy redukují problém do 2D prostoru, čímž dochází ke ztrátě části nesené informace, a vizualizují vývoj shlukování s parametrem *cut-off* o hodnotě 10. Vizualizuje tedy vývoj tohoto procesu s výjimkou zachycení vzniku posledních 10 shluků. Po porovnání získaných výsledků s referenčními, lze opět vidět, že algoritmus *Twister Tries* neshlukuje data tak, jak bylo předpokládáno a výsledky nejsou shodné s výsledky získanými pomocí exaktního algoritmu.



Obrázek 7.5: Grafy znázorňují posledních pět tunelů vzniklých v procesu shlukování pomocí 20 stromů o výšce 10, ve kterém byl použit druhý typ *hashovací* funkce s konfigurací – rozmezí offsetu: $-20 - 20$, velikost koše: $4,5\,\mathring{A}$. Jednotlivé trajektorie tvoří deset 3D bodů získaných z 30-dimenzionálních souřadnic tunelů. Grafy znázorňují stejnou situaci pod jinými pozorovacími úhly pro lepší pochopení. Grafy byly vytvořeny pomocí nástroje *MATLAB* [5].



Obrázek 7.6: Graf znázorňuje referenční řešení procesu shlukování. Barevně je rozlišeno deset posledních shluků, které se povedlo pomocí algoritmu najít.



Obrázek 7.7: Graf znázorňuje posledních deset shluků, které vznikly v procesu shlukování algoritmem *Twister Tries*. Barevně je rozlišeno deset posledních shluků, které algoritus našel. Shlukování bylo provedeno pomocí 15 stromů o výšce 10 s využitím druhého typu *hashovací* funkce s konfigurací – rozmezí offsetu: $-20 - 20$, velikost koše: $4,5\,\mathring{A}$.

[5] http://www.mathworks.com/products/matlab/

Kapitola 8

Závěr

V předložené práci jsem se zabývala výběrem vhodného shlukovacího algoritmu pro shlukování tunelů v proteinových strukturách, jeho návrhem a implementací. Dále jsem se zabývala metrikami pro porovnání vícedimenzionálních dat a jejich vizualizací. Při vývoji byla použita metodika testy řízeného vývoje a po celou dobu vývoje jsem používala distribuovaný systém pro správu verzí, *GIT* [9].

Z prostudovaných algoritmů jsem pro implementaci zvolila hierarchický algoritmus *Twister Tries* [11]. Tento algoritmus pracuje ve dvou fázích. První fáze slouží k předzpracování dat pomocí tzv. *hashovacích* funkcí. Tyto funkce jsou sestrojeny na základě vzdálenostní metriky a řadí elementy do tzv. košů. V koši se pak nacházejí takové elementy, které mají, z hlediska použité metriky, od sebe vzdálenost menší, než je stanovená mez. Výhodou těchto funkcí je, že pracují velmi rychle. Druhou fází je shlukování elementů rozdělených do košů. V této fázi se algoritmus rozhoduje stochasticky. Díky těmto vlastnostem pracuje algoritmus s lineární časovou i prostorovou složitostí.

Vytvořila jsem návrh pro tento algoritmus pro programovací jazyk *Java*, provedla jeho implementaci a otestování. Návrh byl vytvořen tak, aby implementaci bylo možné integrovat do programu *CAVER* [28], což je nástroj zabývající se detekcí útvarů v rámci proteinových struktur. V rámci hledání takových útvarů je využíváno shlukování. Zde má mnou navržený algoritmus sloužit k akceleraci stávajícího algoritmu *average-linkage* [28]. Myslím si však, že pokud by algoritmus nemusel být implementován v souladu s rozhraním pro program *CAVER* [28], bylo by jej vhodnější implementovat v jazyce *C++*, což by umožnilo dosáhnout vyšší efektivity. Popřípadě zvážit implementaci v programovacích jazyce *Python* či *R*, jelikož tyto jazyky jsou velmi často využívány právě vědeckými komunitami. Vstupně výstupní rozhraní bylo rovněž navrženo a implementováno v souladu s koncepcí programu *CAVER* [28] a využívá se tedy výhradně textových souborů v nestandardním formátu. Myslím si, že by bylo mnohem efektivnější použít pro vstupně výstupní operace binární samoopravné soubory (např. knihovna *HDF* [1]). Jejich výhodou, ne však jedinou, je, že umí efektivněji pracovat s velkými a vícedimenzionálními datovými sadami. Pro práci se soubory v těchto datových formátech existuje velká řada knihoven pro různé programovací jazyky. Dále existují nástroje, které umí takto uložená data přímo zpracovat a zvizualizovat.

Testování algoritmu jsem provedla na implementační úrovni a také na úrovni získaných výsledků ze shlukování, pro jejichž otestování oproti referenční sadě shlukování (145 tunelů nalezených v 20 tisicích snímcích z 10ns simulace molekulové dynamiky enzymu haloalkan dehalogenáza *DhaA* [28]) jsem zvolila *Euklidovskou* metriku [29]. Na úrovni testování správnosti implementace jsem vytvořila sadu jednotkových testů, které algoritmus splňuje na 100%. Z testování výsledků shlukování jsem dospěla k závěrům, že algoritmus *Twister*

Tries [11] nepracuje tak přesně jako exaktní algoritmus *average-linkage* [28], jehož výsledky shlukování byly použity jako referenční. Tato skutečnost může být způsobena nevhodnou *hashovací* funkcí. Ve své implementaci jsem použila tři typy *hashovací* funkce. Všechny pracují na stejném principu a liší se ve stupni redukce dimenzionality. Algoritmus s nimi dosahuje podobných výsledků shlukování, avšak velikost dimenzionality má veliký vliv na rychlost výpočtu. Z tohoto plyne, že pro dosažení téměř stejných výsledků shlukování nám stačí použít jednodušší funkci, která redukuje dimenzionalitu problému.

Předpokládanou hypotézou této práce bylo, že i přibližným shlukovacím algoritmem budeme schopni dosáhnout stejných výsledků jako s exaktním algoritmem. Tuto hypotézu se však nepovedlo potvrdit. Je však možné, že testovací sada pouze 145 tunelů byla příliš malá na to, aby se změny dostatečně projevily. Další možností je, že nebyla vytvořena vhodná *hashovací* funkce pro daný problém.

Literatura

[1] The HDF Group. 2014.
URL https://www.hdfgroup.org/

[2] Flexible Image Transport System. 2016.
URL http://fits.gsfc.nasa.gov/

[3] JUnit. 2016.
URL http://junit.org/junit4/

[4] NetBeans. 2016.
URL www.netbeans.org

[5] Unidata Program Center. 2016.
URL http://www.unidata.ucar.edu/software/netcdf/

[6] Agrawal, R.; Gehrke, J.; Gunopulos, D.; aj.: Automatic Subspace Clustering of High Dimensional Data for Data Mining Applications. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, SIGMOD '98, New York, NY, USA: ACM, 1998, ISBN 0-89791-995-5, s. 94–105, doi:10.1145/276304.276314.
URL http://doi.acm.org/10.1145/276304.276314

[7] Bawa, M.; Condie, T.; Ganesan, P.: LSH forest: self-tuning indexes for similarity search. *Proceedings of the 14th international conference on World Wide Web - WWW '05*, 2005: str. 651, doi:10.1145/1060745.1060840.
URL http://dl.acm.org/citation.cfm?id=1060745.1060840

[8] Branden, Carl; Tooze, J.: *Introduction to Protein Structure*. New York: Garland Science, druhé vydání, 1999, ISBN 0815323050, 410 s.

[9] Chacon, S.: *Pro Git*. Berkely, CA, USA: Apress, první vydání, 2009, ISBN 1430218339, 9781430218333.

[10] Cheng, W.; Wang, W.; Batista, S.: Grid-Based Clustering. In *Data Clustering: Algorithms and Applications*, editace C. Aggarwal, C., Charu; Reddy, K., kapitola 6, Chapman and Hall/CRC Press, 2013, ISBN 9781466558212, str. 652.

[11] Cochez, M.: Twister Tries : Approximate Hierarchical Agglomerative Clustering for Average Distance in Linear Time. 2015.

[12] Fischer, B.: rhdf5 - HDF5 interface for R. Technická zpráva, 2014.
URL http://master.bioconductor.org/packages/release/bioc/vignettes/rhdf5/inst/doc/rhdf5.pdf

[13] Gilpin, S.; Qian, B.; Davidson, I.: Efficient hierarchical clustering of large high dimensional datasets. *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management - CIKM '13*, 2013: s. 1371–1380, doi:10.1145/2505515.2505527.

[14] Goil, S.; Nagesh, H.; Choudhary, A.: MAFIA: Efficient and scalable subspace clustering for very large data sets. *Discovery and Data Mining*, ročník 5, 1999: s. 443–452, doi:CPDC-TR-9906-010.
URL http://mrl.cecsresearch.org/Resources/papers/goil99mafia.pdf

[15] Gong, Y.; Kumar, S.; Verma, V.; aj.: Angular Quantization-based Binary Codes for Fast Similarity Search. *Google Research*, 2013: s. 1–9, ISSN 10495258.

[16] Heer, J.; Bostock, M.; Ogievetsky, V.: A Tour through the Visualization Zoo. *Communications of the ACM*, ročník 53, č. 5, 2010: s. 59–67, ISSN 00010782, doi:10.1145/1743546.

[17] Hinneburg, a.; Keim, D.: DENCLUE : An efficient approach to clustering in large multimedia databases with noise. *Proceedings of 4th International Conference on Knowledge Discovery and Data Mining (KDD-98)*, , č. c, 1998: s. 58–65, doi:10.1.1.44.3961.
URL http://www.aaai.org/Papers/KDD/1998/KDD98-009.pdf

[18] Hinneburg, A.; Keim, D. a.: Optimal Grid-Clustering: Towards Breaking the Curse of Dimensionality in High-Dimensional Clustering. *International Conference on Very Large Databases (VLDB)*, 1999: s. 506–517, doi:1-55860-615-7.

[19] Kull, M.; Vilo, J.: Fast approximate hierarchical clustering using similarity heuristics. *BioData Mining*, ročník 1, č. 1, 2008: s. 1–14, ISSN 1756-0381, doi:10.1186/1756-0381-1-9.
URL http://dx.doi.org/10.1186/1756-0381-1-9

[20] Kuročenko, A.: *Vývoj algoritmu pro detekci kapes v proteinech*. Diplomová práce, Masaryk University, 2014.
URL http://is.muni.cz/th/359759/fi_m/thesis.pdf

[21] Liang, J.; Edelsbrunner, H.; Woodward, C.: Anatomy of protein pockets and cavities: measurement of binding site geometry and implications for ligand design. *Protein science : a publication of the Protein Society*, ročník 7, 1998: s. 1884–1897, ISSN 0961-8368, doi:10.1002/pro.5560070905.

[22] Liao, W.-k.: A Grid-based Clustering Algorithm using Adaptive Mesh Refinement. *Proceedings of the 7th Workshop on Mining Scientific and Engineering Datasets*, 2004, doi:10.1.1.128.5119.

[23] Lukasová, J., A. a Šarmanová: *Metody shlukové analýzy*. Praha: SNTL, 1985.

[24] Manning, C. D.; Raghavan, P.; Schütze, H.: Hierarchical clustering. In *Introduction to Information Retrieval*, kapitola 17, Cambridge University Press, 2009, ISBN 0521865719, s. 377–401, doi:10.1017/CBO9780511809071.017.
URL http://www-nlp.stanford.edu/IR-book/

[25] Meila, M.: Comparing clusterings-an information based distance. *Journal of Multivariate Analysis*, ročník 98, č. 5, 2007: s. 873–895, ISSN 0047259X, doi:10.1016/j.jmva.2006.11.013.

[26] Milenova, B. L.; Campos, M. M.: O-cluster: scalable clustering of large high dimensional data sets. In *In Data Mining, Proceedings from the IEEE International Conference*, 2002, s. 290–297, doi:10.1.1.85.866.

[27] Nam, B.; Sussman, A.: Improving access to multi-dimensional self-describing scientific datasets. *Proceedings - CCGrid 2003: 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2003: s. 172–181, doi:10.1109/CCGRID.2003.1199366.

[28] Pavelka, A.; Sebestova, E.; Kozlikova, B.; aj.: CAVER: Algorithms for Analyzing Dynamics of Tunnels in Macromolecules. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, ročník PP, č. 99, 2015: s. 1–1, ISSN 1545-5963, doi:10.1109/TCBB.2015.2459680.

[29] Rajaraman, A.; Ullman, J. D.: Mining of Massive Datasets. *Lecture Notes for Stanford CS345A Web Mining*, ročník 67, 2011: str. 328, ISSN 01420615, doi:10.1017/CBO9781139058452.
URL http://ebooks.cambridge.org/ref/id/CBO9781139058452

[30] Sambasivarao, S. V.: NIH Public Access. ročník 18, č. 9, 2013: s. 1199–1216, ISSN 1878-5832, doi:10.1016/j.micinf.2011.07.011.Innate.

[31] Sehnal, D.; Svobodova Varekova, R.; Berka, K.; aj.: MOLE 2.0: advanced approach for analysis of biomacromolecular channels. *Journal of Cheminformatics*, ročník 5, č. 1, 2013: str. 39, ISSN 1758-2946, doi:10.1186/1758-2946-5-39.
URL http://www.jcheminf.com/content/5/1/39

[32] Špačková, E.: Analýza tunelů a pórů v proteinech. 2013.
URL http://is.muni.cz/th/375733/prif_b/BP_ES.pdf

[33] Steinbach, M.; Ertoz, L.; Kumar, V.: The Challenges of Clustering High Dimensional Data. *New Directions in Statistical Physics*, 2004: s. 273–309, doi:10.1007/978-3-662-08968-2_16.

[34] Stoker, S., H.: Cengage Learning, 7 vydání, 2014, ISBN 978-1-305-08107-9, 356–403 s.

[35] Wickham, M. H.: Package 'ggplot2'. 2014.

[36] Xu, Rui; Wunsch, C., D.: *Clustering*. Wiley-IEEE Press, 2008, ISBN 978-0-470-27680-8, 368 s.

[37] Zhang, T.; Fayyad, U.: BIRCH : A New Data Clustering Algorithm and Its Applications. *Data Mining and Knowledge Discovery*, ročník 1, 1997: s. 141–182, doi:10.1.1.325.7171.

Přílohy

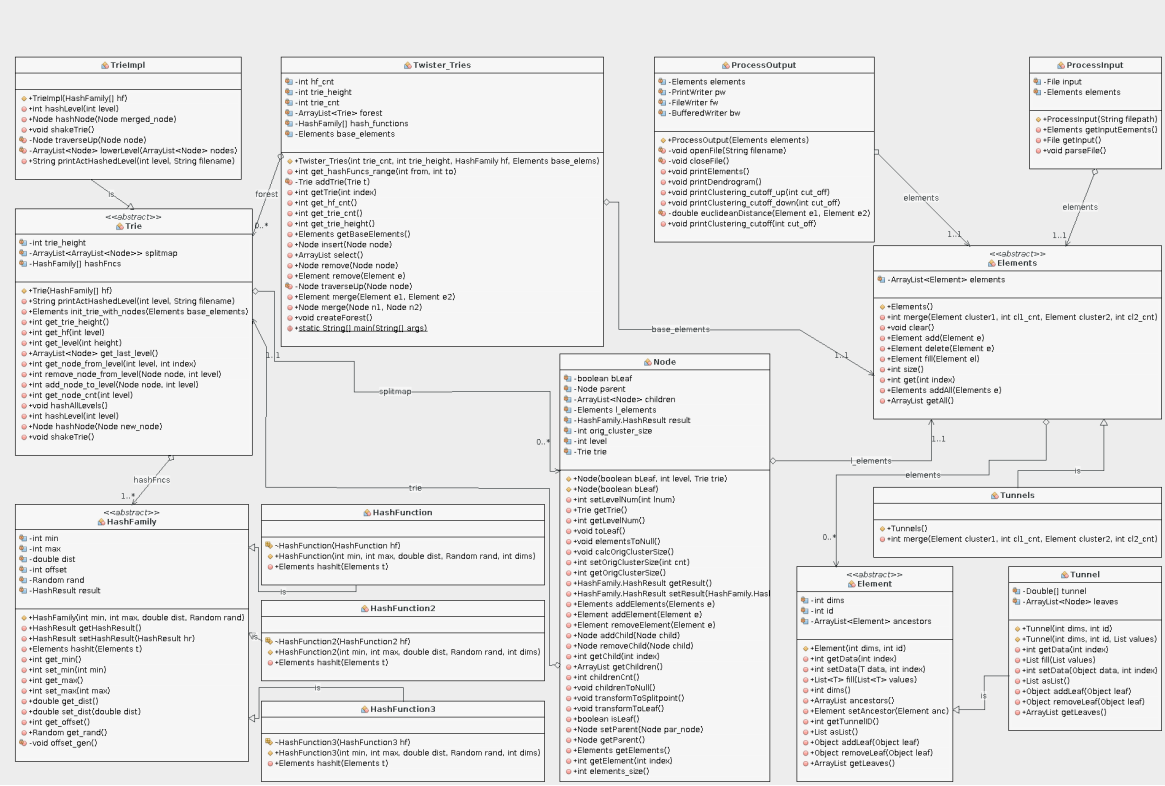
Seznam příloh

A Třídní diagram **51**

B Obsah CD **52**

Příloha A

Třídní diagram



Obrázek A.1: Třídní diagram návrhu algoritmu *Twister Tries*. Řídící třídou algoritmu je třída *Twister Tries*, která zprostředkovává stěžejní stromové operace, tj. výběr kandidátních uzlů pro operaci shlukování `select()`, operace shlukování `merge()`, vymazání uzlu ze stromů `delete()` a vložení nového uzlu do stromů `insert()`. Třídy *Element* a *Elements*, reps. *Tunnel, Tunnels*, vytvářejí rozhraní pro práci s entitami na nejnižší úrovni. Třídy *Node, Trie* a *TrieImpl* vytvářejí rozhraní pro práci se stromem, jeho uzly a seznamem rozdělujících bodů. Třídy *HashFamily* a *HashFunction* tvoří rozhraní a konkrétní implementaci výpočtu *hashovací* funkce. Třídy *HashFunction2* a *HashFunction3* pouze implementují jiné *hashovací* funkce. Třídy *ProcessInput* a *ProcessOutput* zprostředkovávají zpracování vstupního souboru pro algoritmus a vytvářejí požadované výstupy z algoritmu.

Příloha B

Obsah CD

Tabulka B.1: Obsah přiloženého CD.

Umístění	**Obsah složky**
`/`	Soubor `readme` s popisem obsahu CD a návody.
`/doc/pdf`	Text práce ve formátu *PDF*.
`/doc/latex`	Zdrojový text práce ve formátu LaTeX.
`/src/Twister_Tries`	Zdrojové kódy algoritmu.
`/src/UML_Diagrams`	UML diagramy algoritmu Twister Tries.
`/examples`	Všechny vstupní a výstupní soubory, včetně referenčních.